JMLR: Workshop and Conference Proceedings 67:15–29, 2016 PAPIs 2016

Scaling Machine Learning as a Service

Li Erran Li ERRANLI@UBER.COM
Eric Chen ECHEN@UBER.COM
Jeremy Hermann JHERMANN@UBER.COM
Pusheng Zhang PUSHENG@UBER.COM
Luming Wang LUMING.WANG@UBER.COM
Uber Technologies, Inc. 1455 Market St, San Francisco, CA 94103

Abstract

Machine learning as a service (MLaaS) is imperative to the success of many companies as they need to gain business intelligence from big data. Building a scalable MLaaS for mission-critical and real-time applications is a very challenging problem. In this paper, we present the scalable MLaaS we built for Uber that operates globally. We focus on several scalability challenges. First, how to scale feature computation for many machine learning use cases. Second, how to build accurate models using global data and account for individual city or region characteristics. Third, how to enable scalable model deployment and real-time serving for hundreds of thousands of models across multiple data centers. Our technical solutions are the design and implementation of a scalable feature computing engine and feature store, a framework to manage and train a hierarchy of models as a single logical entity, and an automated one-click deployment system and scalable real-time serving service.

Keywords: Machine learning as a service, feature store, big data, data streaming, Apache Spark

1. Introduction

Machine learning (ML) and Artificial Intelligence (AI) have been rapidly transforming many industries such as e-commerce, transportation and healthcare. ML and AI have been referred to as the "new electricity" (Ng, 2017). ML and AI have traditionally been trained in a single machine using tools such as Scikit-learn or R with trained models being deployed manually. However, in the big data and on-demand real-time service era, this severely limits the scale of application of ML. Recognizing a need, many companies such as Facebook and cloud providers such as Microsoft, Amazon, and Google have been building machine learning platforms to automate and scale machine learning.

ML is critical to Uber's business in several ways. It finds wide applicability in predicting supply and demand, recommending restaurants for UberEATS, mapping, and self-driving cars, for example. To unleash the true power of machine learning, we have been building a company-wide ML platform within Uber. The goal of the platform is to make it extremely easy and fast for any engineer or data scientist to develop and deploy ML solutions in production. This is particularly challenging for several reasons. First, Uber applications interact with many people and businesses, such as restaurants, in real-time, which generate a large amount of data. Training with big data needs to be scalable to enable fast iteration of

model development and frequent model deployment: predictions need to happen in real-time. Second, Uber operates in hundreds of cities and it is difficult to train and manage models separately for each city. Third, Uber has diverse use cases which require many different kinds of ML algorithms such as time series for demand forecasting, deep learning for mapping, unsupervised learning to group drivers into cohorts, multi-armed bandit algorithms and reinforcement learning for restaurant recommender systems and self-driving cars.

In this paper, we focus on addressing several of these key challenges and our solutions. First, to make it easy to obtain features, we have built a feature store and its associated computing engines. We scalably compute and aggregate spatial and temporal features for batch training and real-time prediction. Second, we manage and train a hierarchy of models as a single logical entity for all cities. Third, we automatically deploy and scale our model serving service. Although the platform focuses on Uber's use cases and integrates with Uber's internal data infrastructure and applications, we believe our design and lessons learned can offer valuable insights to industries and the system community, especially the ML system community, for two main reasons. First, Uber's applications are mission-critical and interact with people in real-time in the physical world. This trend will continue and ML will play an essential role. Second, major components of our platform use open-source software such as Spark (Zaharia et al., 2012), Cassandra (Apache) and Samza (Apache, 2017c).

The remainder of this paper is organized as follows: In Section 2, we present an overview of our system architecture. In Section 3, 4, 5, we focus on three key scaling challenges and solutions. We discuss lessons learned in Section 6, related work in Section 7 and conclude in Section 8.

2. Overview of System Architecture

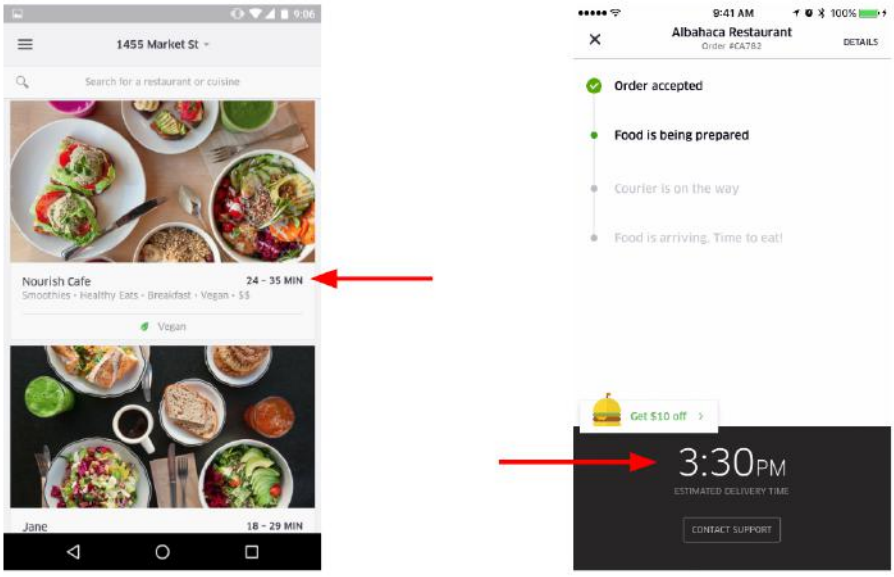

Figure 1: UberEATS estimated time to delivery (ETD) feature

We present the system architecture overview in the context of a typical ML workflow (See Figure 2). A typical ML workflow involves getting the relevant data, engineering features, training models, evaluating model performance, deploying, serving and monitoring models.

To be concrete, we illustrate our system with a ML use case for UberEATS. When a customer opens the UberEATS app, it shows the estimated time to delivery (ETD) of a food order for each restaurant (See Figure 1).

Users such as data scientists interact with our platform through the front end web UI. HTTP requests from the front end go to the API tier which communicates with both the cluster scheduler and the software deployment infrastructure. The cluster scheduler kicks off jobs preparing data, training models and making batch predictions and the software deployment infrastructure deploys models to real-time service Docker containers. Systems can also interact with our platform programmatically through the API tier.

2.1. Data Preparation

Data is collected continuously and stored in databases or distributed file systems. For training, we need feature data and outcomes (or labels). We use SQL as the common querying interface to all our data sources. The Uber query engine finds the best way to execute these queries. For example, if features or outcomes involve data in our Hadoop data lake, a Spark (Zaharia et al., 2012) job will be launched. Also, metadata is attached to all data sources collected, so different use cases can share the same data preparation job.

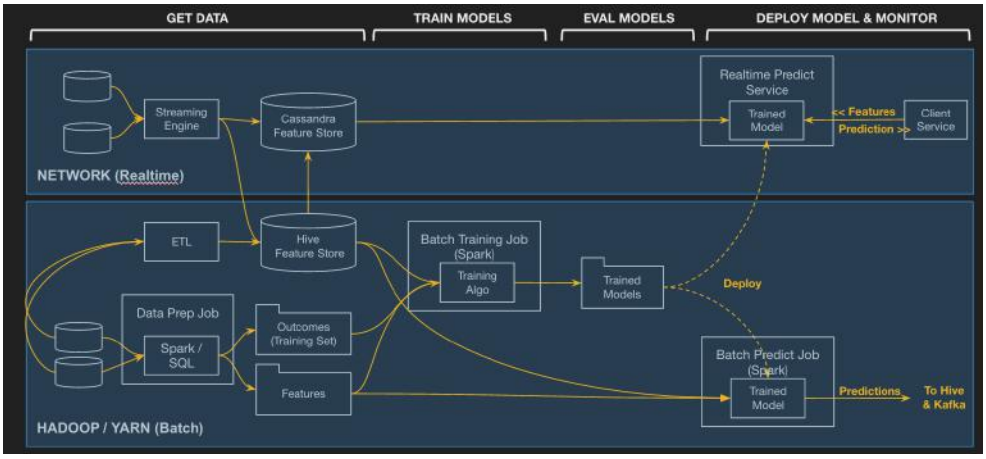

Figure 2: Overall system architecture

2.2. Feature Store

To speed up the process of building machine learning pipelines and enable real-time predictions, we needed to address the general needs of feature computing and serving as they can take significant time to compute.

We built a shared feature computation engine and feature store for both batch and real-time machine learning pipelines. In the ideal case where all features can be pulled from the Feature Store, the machine learning project's data pipeline just needs to provide the UUIDs

for the entities being modeled and all the features can be pulled by name from the Feature Store.

The Feature Store enables real-time machine learning use cases by providing low latency access to these same features in an online serving store - Cassandra (Apache). We provide unified access via canonical names to these features for batch training and prediction, and real-time prediction. This is critical because the same expressions used to retrieve features in the batch environment during training need to work at prediction time when the model is deployed as a real-time service.

2.3. Model Training

The platform needs to support a diverse set of ML algorithms such as linear regression, gradient boosted trees, autoregressive integrated moving average (ARIMA) models, and deep learning. Training needs to be scalable and reliable. For non-deep learning algorithms, we use Spark MLlib (Meng et al., 2016) and XGBoost (Chen and Guestrin, 2016) because they meet these requirements. For deep learning, we support Caffe (Jia et al., 2014) and Tensorflow (Abadi et al., 2016).

2.4. Model Deployment and Management

As we operate in new cities and regions, we would like to apply machine learning to optimize various aspects of user experience. For example, pricing to balance supply and demand, and product recommendations. This raises two challenges: Firstly, for a city recently entered, there may not be enough data to train an accurate model. In this case, we would like to fall back to a model trained at regional scale. Secondly, to train and make predictions using one separate model for each city is inefficient and hard to manage. Our solution is to organize the models in a hierarchical structure. Instead of separately managed models per city, we train a single hierarchical model for all cities and deploy the hierarchical model across all countries. We also leverage Uber's deployment infrastructure to enable one-click deployment of models for batch serving and real-time serving.

2.5. Model Serving and Live Monitoring

Models should behave the same in both real-time and batch mode when serving predictions. Our model serving core can be plugged into either a batch prediction pipeline or real-time serving containers. In batch mode, we plug the serving core into Spark jobs. In real-time mode, the serving core is hosted by Uber RPC services. The client sends Thrift (Apache, 2017d) requests via RPC to our online prediction service. In the request, basis features and the model ID are included. Based on the request, the prediction service fetches extra features from the feature store, performs feature transformation and selection, and invokes the serving core to make the actual prediction, before returning the prediction to the client. In both batch and real-time mode, all predictions are logged to Kafka (Apache, 2017b) for diagnostics and monitoring.

2.6. UberEATS Estimated Time to Delivery Use Case

Figure 3 illustrates how UberEATS ETD makes use of our platform. Both training and prediction use features from the Feature Store. The model uses gradient boosted trees and is trained using Spark MLlib. The deployed models serve the UberEATS application when users open the app.

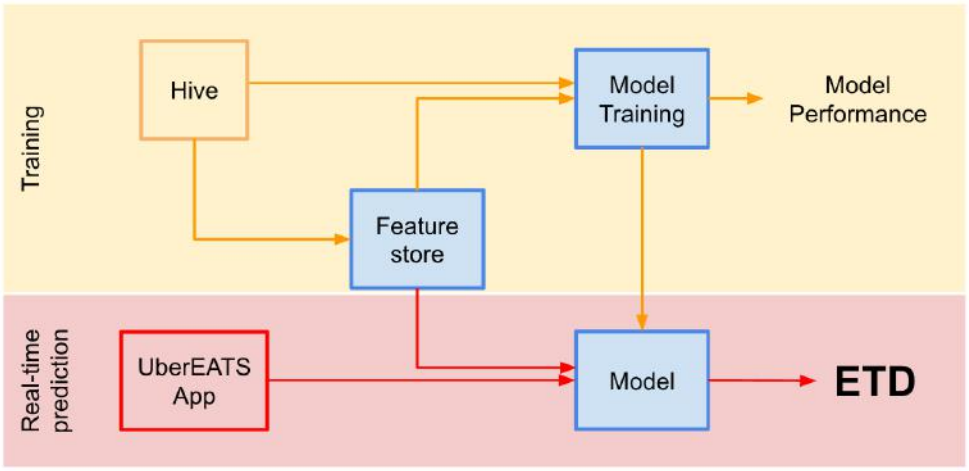

Figure 3: UberEATS ETD

3. Feature Computation and Serving

One key impediment to developing ML solutions is that it is time consuming to engineer and compute the right features or predictors. To overcome this, we would like to make them readily available for both training and real-time predictions: we need to address the general needs of feature computing and serving.

3.1. Requirements and Challenges

We first discuss our requirements and explain the challenges.

3.1.1. Requirements

We have several requirements. First, we would like to compute a diverse set of features. Features are associated with simple entities in Uber's systems, such as riders, drivers, trips, and cities. In some cases, features can be associated with composite entities such as a cohort (a set of drivers in an incentive campaign), a set of similar trips (origin and destination within one mile of one another), or geofences. We distinguish simple, relatively static attributes of the entity in question, such as a driver's signup city from aggregate features relative to a nominal time, such as the number of trips in last 30 days per driver or average rider waiting time of a geofence over the last hour. We would like to make these two types of pre-computed features available to training, offline prediction, and real-time prediction. Third, we need feature freshness and serving latency guarantees. Historical features served from the online

features store are less than 12 hours old and are available with 10 ms query latency. Near real-time features served from the online features store are less than 5 min old and available in the online serving store with 10 ms query latency.

3.1.2. Challenges on Feature Computation

Since we need to compute historical aggregate features over long durations such as weeks and near real-time features over several minutes, it is desirable, for simplicity, to have a unified batch and streaming engine. However, there are several limitations of such an engine. First, it is much less efficient to compute historical features over a long duration because of the need to maintain states for on demand update of states due to delayed data. Second, there is a lack of support for computing certain features (for example, features of driver-rider fraud graphs) in current streaming engines such as Spark streaming.

3.1.3. Challenges on Feature Serving

For batch training and prediction, we would like efficient bulk access. For real-time prediction, we need low latency per record access. Ideally, we would like a single feature store that serves both historical features and near real-time features. However, there are two main problems. First, most data stores are efficient for either point queries or for bulk access, but not both. Second, frequent bulk access can adversely impact the latency of point queries. Third, the two have different query patterns. For the online case, we only use features at the time of the query. For the offline case, we use features at the event time (nominal time) of the data.

3.2. Solution

Due to problems of unified computation engines and data stores, we made the decision to leverage the strength of separate batch computing and streaming engines, as well as separate bulk serving and online serving stores. Because of the separation, we need to address three problems. First, we require that each feature to be computed by one code path for training and prediction. We want to avoid the case where a near real-time feature is computed by one code path in a streaming system for online serving and a separate code path in the batch system for training. Second, users only need to express their selected features for training. The same expression should work for real-time prediction without user involvement. Third, batch computed historical features need to be uploaded to the online data store and real time computed features should be uploaded to bulk serving store.

We now describe our detailed design. We chose Apache Spark as our batch computing engine, Samza (Apache, 2017c) as our streaming engine, Hive (Apache, 2017a) as batch feature store, and Cassandra (Apache) as our online feature store.

3.2.1. Feature Groups

Features need to be computed, stored and accessed. To minimize operational overhead and optimize storage and access patterns, we group features together. A feature group is an atomic unit for scheduling and storage. All features in the same group are computed by the same batch job or streaming job, and they are stored in the same table.

It is up to the feature owners to define their own feature group or add features to an existing feature group owned by others. Feature groups will generally store features related to an entity that are convenient and efficient to compute and manage together. For instance, all the trip related features for a geofence would likely be in one feature group since they all can be computed efficiently by one job that scans through the trip history.

A feature group consists of a meta-data YAML file that defines its semantics and a set of source code files written in the appropriate language for the execution platform (Spark or SQL for batch, Samza for streaming). These resources are all managed with git as source code and subject to regular code review and approval.

When the job code is deployed to the execution platform, the metadata will get pushed into the feature directory (a global registry of all available features) so that it is available globally. Metadata about data availability can also be updated as the ETL jobs complete.

Once the feature groups are registered in the feature directory, they can be browsed and discovered via the REST API or web UI.

Deprecating features will require notifying users. (We log usage of features so that we know exactly who is using what features.)

Once features are active, there should never be semantic changes to those features. Semantic changes should always result in a new feature with a new name in order to not break existing consumers.

3.2.2. Batch feature computation and feature store

We chose Apache Spark as the computing engine for batch computed features and Hive as the batch feature store.

Spark ETL jobs are run to compute a set of related features (a.k.a. feature group) and they are written into a single Hive table, one column per feature. Each row is a set of features for a specific entity identified by a unique ID (e.g. driver UUID) and is relative to a specific nominal time. As features in a feature group are likely to be accessed together by nominal time, we use nominal time as the partition key. This ETL process is used to generate regular historical features used for batch training and both batch and real-time prediction.

When each ETL job finishes, the relevant rows of historical features are copied to Cassandra clusters in each data center so that the features can be queried with low latency for real-time prediction.

Both training and batch predictions use offline features. In offline mode, the primary key of a set of features is the combination of the unique ID and the nominal time. When the nominal time is given, there is no difference between historical features and near realtime features. Using unique ID and the nominal time, we associate feature store features to each data record in the training or prediction dataset. This is implemented as Spark jobs joining data sources from Parquet files and data frames from Hive.

3.2.3. Near real-time feature computation and online feature store

The online feature store serves features generated from two sources. Regular historical features are generated by Spark ETL jobs. They need to be bulk loaded into the online store periodically. The other set is near real-time features. By consuming Kafka topics, we

compute these features using streaming jobs, and update them in the online feature store directly. Cassandra supports both data loading patterns.

In online prediction, clients are responsible for sending the relevant entity UUID and any features not in the feature store. The online prediction service uses the feature accessor expressions in the feature engineering component of the deployed model to pull pre-computed features directly from Cassandra. Once all features are available, the prediction service will compute a response which involves DSL transformation of features followed by running inference of the machine learning algorithm. It is worth noting that, at batch training time, the Spark job uses expressions identical to the ones that are used in online serving to pull features from Hive and automatically joins them if they are from different Hive tables.

3.3. UberEATS ETD Use Case

The ETD model uses different type of features as follows:

- Restaurant features such as location, average meal preparation time, average delivery time, average demand during lunch
- Contextual features such as the time of day, or day of week
- Order features such as the number of items or the total cost
- Near real-time features such as information about the past N orders

UberEATS order and context features are supplied by the app. Restaurant and near real-time features are obtained by the service backend from the Feature Store.

4. Partitioned Models

Data scientists at Uber often build one ML model per city. It is time consuming and error prone for users to manage training and deployment of several hundred individual models. To simplify the process, we allow users to define a hierarchical partitioning scheme such as country and city. We automatically train a model per partition (a node in the tree). We refer to the collection of models as one partitioned model. A partitioned model with all partitions is managed and deployed as one single logical model. When one partition has insufficient data to train a model, it will fall back to use the model trained at its parent node or ancestor node if the parent partition happens to be skipped for training as well.

4.1. Challenges

First, each partition needs to create two sample sets (one training and one validation set) for model training. This results in significant data duplication, unnecessary data serialization, and a large number of files if we persist data across all partitions. Second, training several hundred models can be time consuming and cause the starvation of regular non-partitioned model training. A large number of training jobs are spawned out of training on partitioned data: for example, a partitioned model with a 3-level hierarchy for country-city includes 1 global, 60 country and 400 city partitions (a total of 461 partitions) for training the partitioned model. Thus, we need some strategy for training job management.

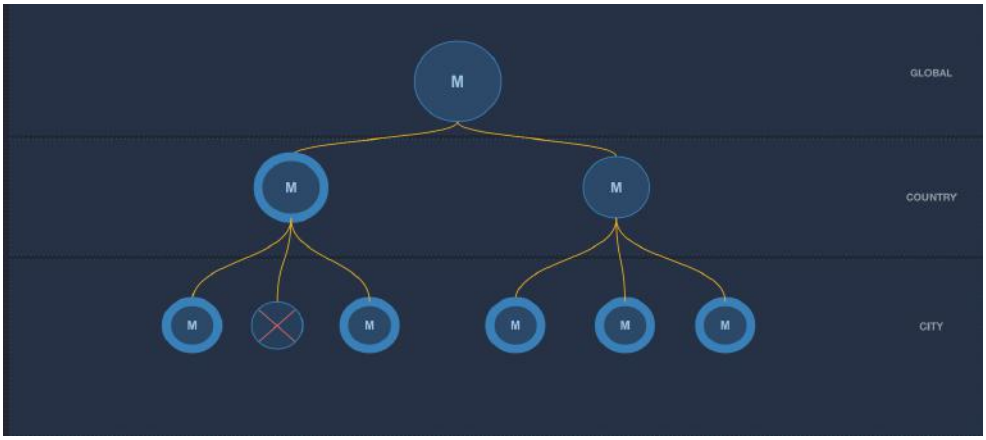

Figure 4: Illustration of a partitioned model: the red cross means that a model is not trained due to lack of data in this partition, or that its model performance is worse than the one from its parent's.

4.2. Solution

We now describe our solution for scalable data preparation and training of partitioned models in detail.

4.2.1. Scalable data preparation

We persist the global training and validation sets with the metadata of partitions so that we can create partitioned training and validation sets on-the-fly to avoid duplicate data serialization and storage on partition levels. The data persistence enables data checkpoint and failover on training partition models, i.e., each partition can be re-trained if the previous attempt of model training fails on a partition due to some transient issue.

4.2.2. Scheduling training jobs

There are three possible approaches: (1) rely on a workflow scheduler such as Apache Oozie, (2) an external controller with all the scheduling logic, (3) write a single Spark application. For the first approach, we would need to construct the training jobs in the workflow scheduler. Workflow schedulers typically are not equipped to process input data and construct training jobs based on information extracted dynamically. For example, we need to extract the keys of the partition model and train per node in the tree, and we only train when we have enough data. Writing an external controller will be much more involved and we cannot do it in our front end or API tier. So we decided to take the third approach.

Fortunately, Spark has good support internally to schedule jobs fairly and in parallel. Therefore, we can train from top to bottom, and parallelize training per level. All partition training is encapsulated in a single Spark application so that it does not flood the job scheduler and can make use of sharing the global data across all partitions training under the same Spark application.

4.3. UberEATS ETD Use Case

UberEATS operates food delivery across many cities and the model of estimated time to delivery (ETD) is trained as a partitioned model, i.e., the ETD model is trained using data from each city when it has sufficient data. This greatly simplified model management by training and deploying a single ETD model for all cities.

5. Real-Time Prediction and Live Monitoring

Many applications of ML such as UberEATS ETD require real-time prediction. The deployed ML models need to be monitored live for their performance. It is challenging to scale ML services for real-time prediction and live monitoring of model performance, and automate model deployment and retirement.

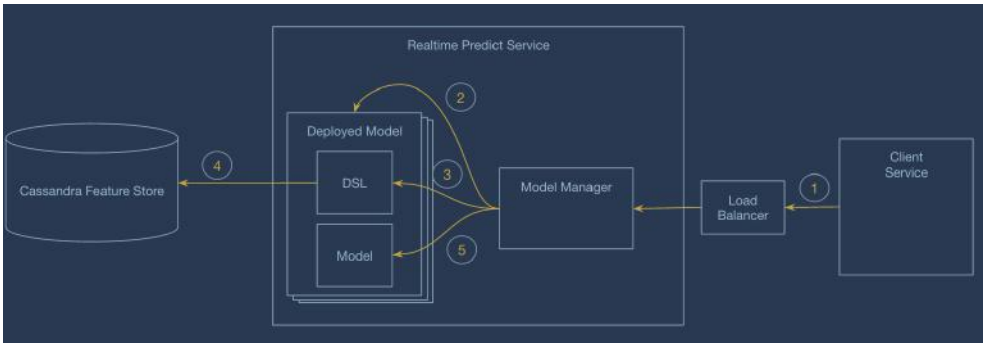

Figure 5: Illustration of real-time serving.

5.1. Challenges

The service needs to be low latency and meet latency bounds even during peak usage. When deploying new models, the services need to keep serving prediction requests. The deployed models need to be monitored live, and degradation of model performance should be detected before seriously impacting on external customer (e.g. riders and drivers) experiences.

5.2. Solution

We now describe our technical solution in detail.

5.2.1. Real-time serving

For real-time prediction, we adopt a high performance RPC service using TChannel as the networking framing protocol (Corbin et al., 2017). TChannel was developed at Uber and has been open sourced. TChannel supports out-of-order responses at extremely high performance where intermediaries can make a forwarding decision quickly. Client service specifies the model to use in the TChannel request transport headers. The predict service inspects these headers and routes internally to the appropriate model. In the case of experimentation, it is up to the client service to decide with each request which version of the model it wants

to invoke. The prediction service is managed by Uber's infrastructure. The infrastructure provides all key support to host services with high availability. It shards the requests of one logical service into multiple physical instances of the service, manages all services using Docker containers, provides rolling updates and monitors service health and payloads.

For the prediction service, our models are packed into docker images. When a physical instance of a service boots up, it loads into memory and warms up all models in the container, and then registers itself as a live instance. After that, it starts accepting prediction requests. Each service instance also periodically checks new models deployed in the Docker image, and loads newly detected models into memory.

5.2.2. Live monitoring of model performance

To ensure deployed models are making good predictions, we log predictions in Kafka (Apache, 2017b) and join logged predictions with actual outcomes. We then publish metrics for monitoring and alerting.

5.2.3. Automated model deployment and retirement

Owners of each ML prediction service decide when to deploy a new model and when to retire an old model. Deployment or retirement requests go through our API tier. Requests are put into a reliable distributed message queue. The message will be sent to our deployment service. For deployment, the service will be responsible to package required files, validate the model can be run correctly, and deploy into running Docker containers. To avoid interference of concurrent deployment requests of the same prediction service, we use Redis distributed locks (Da Silva and Tavares, 2015). For model retirement, the deployment service will purge the model from the Docker containers of the prediction service.

5.3. UberEATS ETD Use Case

UberEATS has bursty requests, especially during lunch and dinner time. Our real-time services can scale with the demand gracefully. We have 99 percentile latency of 20 ms.

6. Lessons Learned

Developing, maintaining and operating a ML system as a service in production is a massive undertaking with many practical challenges as discussed in Sculley et al. (2015). We feel it is beneficial to share some important lessons we learned with the community.

First, throughout the whole ML workflow, a user's jobs can fail for various reasons such as schema mismatch, missing data, Spark job failures, Cassandra failures, etc. Debuggability is very important. This includes surfacing meaningful error messages and logging error messages. It is particularly hard to assign the right amount of resources to Spark MLlib jobs so that training can succeed within the user expected durations.

Second, fault tolerance is very important. Partitioned model training contains hundreds of sub-models. This increases the chance of failure. Cross data center replication is important for seamless real-time prediction.

Third, we designed our system with abstractions at the machine learning algorithm level so that we can easily plug in different ML libraries. This allowed us to easily support

MLlib, XGBoost (Chen and Guestrin, 2016), Caffe (Jia et al., 2014) and TensorFlow (Abadi et al., 2016). Configuration of hyperparameters, evaluation metrics and visualization, and prediction all have APIs common at the algorithm level.

Fourth, automating ML workflow and ease of use is essential. For many common use cases, users do not need to write any code. Interacting with the web UI, all users need to do is specify what features to use from the feature store, and what algorithms and corresponding hyperparameters to use for the model. Models are training automatically and standard performance metrics (e.g. ROC, AUC precision and recall) are provided after training. Users can then select the best model based on performance metrics and deploy to real-time service with one-click.

Fifth, the ML system must be extensible and be easy for other systems to interact with programmatically. Other teams programmatically interact with the platform to train their models. Jupyter notebooks are integrated with the platform through REST. Data scientists find it very useful to iterate on their models in our scalable platform via the notebook.

7. Related Work

We make heavy use of open source components: HDFS (Apache, 2014), Spark, Samza streaming (Apache, 2017c), Cassandra (Apache), Hive (Apache, 2017a), Mesos and Docker (Docker Inc., 2017). We provide users with a standard set of algorithms, a row-based DSL, feature store and one-click deployment of real-time prediction service. Our ML platform tightly integrates with Uber infrastructure for ETL, deployment, prediction and data store. We now discuss closely related ML systems in industry and academia.

Facebook FBLearner Flow (Dunn, 2017) is a workflow scheduling system with an experiment management UI. It composes operators (programs) to construct and execute ML pipelines. It provides a number of standard ML algorithms that users can reuse. Serving is not part of FBLearner Flow. In contrast, our ML platform currently does not support flexible pipeline construction. We manage the whole workflow of training and serving. Amazon, Microsoft and Google all provide cloud services for users to construct ML workflows to train models and prediction services. They also offer high level APIs for computer vision, speech and natural language. Turi (2017) uses the public cloud to offer ML services. However, there is a lack of technical details from which we cannot make an informed comparison.

ML systems have received limited attention in academia. Clipper (Crankshaw et al., 2017) focuses on online serving. It tries to address three key challenges of prediction serving: latency, throughput, and accuracy. Clipper provides a common prediction interface that isolates end-user applications from the variability and diversity in machine learning frameworks. Clipper has a model abstraction layer and a model selection layer. It addressed the challenges of prediction serving latency and throughput with caching and adaptive batching strategies in the model abstraction layer. It addressed the challenges of accuracy by dynamically selecting and combining predictions from each model with straggler mitigation in the model selection layer.

8. Conclusion and Future Work

We have built a scalable ML as a service company-wide platform. We have many internal customers actively training models and many deployed models serving real-time predictions such as UberEATS ETD. Although not a focus of this paper, our platform also provides deep learning as a service. We have many use cases for deep learning including computer vision and natural language processing, and demand forecasting.

For our ongoing work, we are in the process of supporting custom machine learning models for sophisticated customers, and custom performance evaluations, and making pipeline construction more customizable. We are also working on simplifying the whole workflow, scaling training of deep learning models, optimizing real-time serving, and automated and efficient cluster resource allocation including GPUs.

9. Acknowledgements

We would like to acknowledge everyone in the ML platform team at Uber for their contributions. In particular, we would like to thank Aiden Scandella, Alex Sergeev, Anne Holler, Austin Greco, Bhavya Agarwal, Cheng Bai, Chris Chen, Ellie Zhang, Jai Ranganathan, Jennifer Anderson, Joseph Wang, Lezhi Li, Logan Jeya, Manish Chablani, Michael Grazziani, Michael Mui, Mike Del Balso, Mingshi Wang, Mingyang Zhao, Naveen Somasundaram, Nikunj Aggarwal, Paul Mikesell, Qianjun Xu, Raya Karova, Ron Tal, Sayantan Ghosh, Shagandeep Kaur, Simon Cochrane, Srikar Yekollu, Stephanie Blotner, Wai Yip Tung, Wesam Manassra, Will Glad, Yandong Liu and Yun Ni

References

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, GA, 2016. USENIX Association. ISBN 978-1-931971-33-1. URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi.

Apache. Apache Cassandra. http://cassandra.apache.org.

Apache. HDFS architecture. http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html, 2014.

Apache. Apache Hive. https://hive.apache.org/, 2017a.

Apache. Kafka: a distributed streaming platform. https://kafka.apache.org/, 2017b.

Apache. Apache Samza. http://samza.apache.org/, 2017c.

Apache. Apache Thrift. https://thrift.apache.org/, 2017d.

Tianqi Chen and Carlos Guestrin. XGBoost: A scalable tree boosting system. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 785–794, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4232-2. doi: 10.1145/2939672.2939785. URL http://doi.acm.org/10.1145/2939672.2939785.

Joshua T Corbin, Jake Verbaten, Prashant Varanasi, Juncao Li, and Junchao Wu. TChannel: High performance forwarding for general RPC. https://uber.github.io/tchannel/, 2017.

Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017.

Maxwell Dayvson Da Silva and Hugo Lopes Tavares. *Redis Essentials*. Packt Publishing, 2015. ISBN 1784392456, 9781784392451.

Docker Inc. Docker: Build, ship, and run any app, anywhere. https://www.docker.com/, 2017.

Jeffrey Dunn. Introducing FBLearner Flow: Facebook's AI backbone. https://code.facebook.com/posts/1072626246134461/introducing-fblearner-flow-facebook-s-ai-backbone/, 2017.

Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22Nd ACM International Conference on Multimedia*, MM '14, pages 675–678, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3063-3. doi: 10.1145/2647868.2654889. URL http://doi.acm.org/10.1145/2647868.2654889.

Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. Mllib: Machine learning in apache spark. *J. Mach. Learn. Res.*, 17(1):1235–1241, January 2016. ISSN 1532-4435. URL http://dl.acm.org/citation.cfm?id=2946645.2946679.

Andrew Ng. Andrew Ng: Why AI is the new electricity. http://news.stanford.edu/thedish/2017/03/14/andrew-ng-why-ai-is-the-new-electricity/, 2017.

D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-Francois Crespo, and Dan Dennison. Hidden technical debt in machine learning systems. In *Proceedings of the 28th International Conference on Neural Information Processing Systems*, NIPS'15, pages 2503–2511, Cambridge, MA, USA, 2015. MIT Press. URL http://dl.acm.org/citation.cfm?id=2969442.2969519.

Turi. Turi: GraphLab create, a machine learning modeling tool for developers and data scientists. https://turi.com/, 2017.

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2228298.2228301.