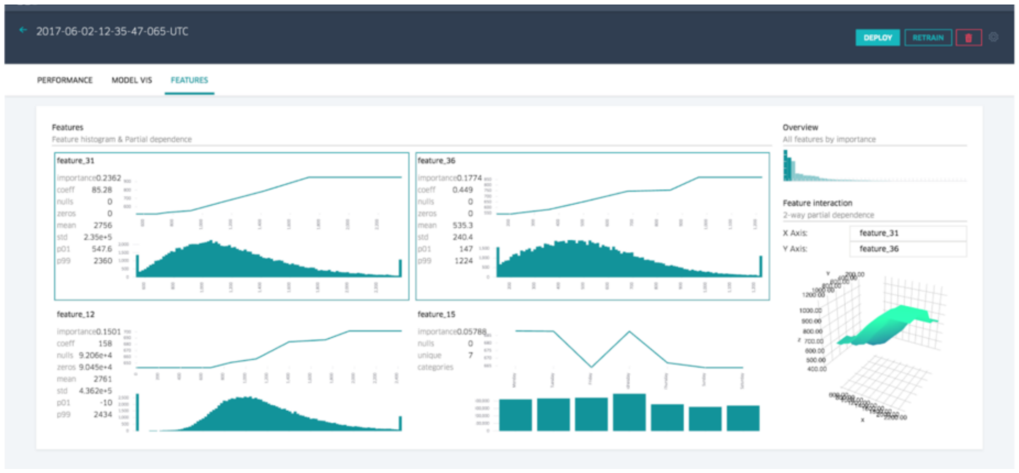

Uber Engineering is committed to developing technologies that create seamless, impactful experiences for our customers. We are increasingly investing in artificial intelligence (AI) and machine learning (ML) to fulfill this vision. At Uber, our contribution to this space is Michelangelo, an internal ML-as-a-service platform that democratizes machine learning and makes scaling AI to meet the needs of business as easy as requesting a ride.

Michelangelo enables internal teams to seamlessly build, deploy, and operate machine learning solutions at Uber's scale. It is designed to cover the end-to-end ML workflow: manage data, train, evaluate, and deploy models, make predictions, and monitor predictions. The system also supports traditional ML models, time series forecasting, and deep learning.

Michelangelo has been serving production use cases at Uber for about a year and has become the de-facto system for machine learning for our engineers and data scientists, with dozens of teams building and deploying models. In fact, it is deployed across several Uber data centers, leverages specialized hardware, and serves predictions for the highest loaded online services at the company.

In this article, we introduce Michelangelo, discuss product use cases, and walk through the workflow of this powerful new ML-as-a-service system.

Motivation behind Michelangelo

Before Michelangelo, we faced a number of challenges with building and deploying machine learning models at Uber related to the size and scale of our operations. While data scientists were using a wide variety of tools to create predictive models (R, scikit-learn, custom algorithms, etc.), separate engineering teams were also building bespoke one-off systems to use these models in production. As a result, the impact of ML at Uber was limited to what a few data scientists and engineers could build in a short time frame with mostly open source tools.

Specifically, there were no systems in place to build reliable, uniform, and reproducible pipelines for creating and managing training and prediction data at scale. Prior to Michelangelo, it was not possible to train models larger than what would fit on data scientists' desktop machines, and there was neither a standard place to store the results of training experiments nor an easy way to compare one experiment to another. Most importantly, there was no established path to deploying a model into production–in most cases, the relevant engineering team had to create a custom serving container specific to the project at hand. At the same time, we were starting to see signs of many of the ML anti-patterns documented by Scully et al.

Michelangelo is designed to address these gaps by standardizing the workflows and tools across teams though an end-to-end system that enables users across the company to easily build and operate machine learning systems at scale. Our goal was not only to solve these immediate problems, but also create a system that would grow with the business.

When we began building Michelangelo in mid 2015, we started by addressing the challenges around scalable model training and deployment to production serving containers. Then, we focused on building better systems for managing and sharing feature pipelines. More recently, the focus shifted to developer productivity–how to speed up the path from idea to first production model and the fast iterations that follow.

In the next section, we look at an example application to understand how Michelangelo has been used to build and deploy models to solve specific problems at Uber. While we highlight a specific use case for UberEATS, the platform manages dozens of similar models across the company for a variety of prediction use cases.

Use case: UberEATS estimated time of delivery model

UberEATS has several models running on Michelangelo, covering meal delivery time predictions, search rankings, search autocomplete, and restaurant rankings. The delivery time models predict how much time a meal will take to prepare and deliver before the order is issued and then again at each stage of the delivery process.

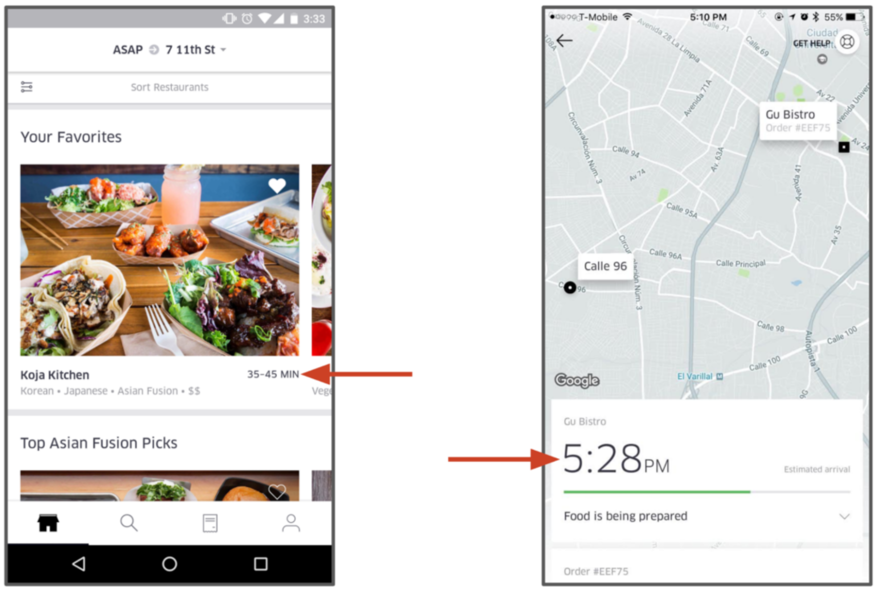

Figure 1: The UberEATS app hosts an estimated delivery time feature powered by machine learning models built on Michelangelo.

Predicting meal estimated time of delivery (ETD) is not simple. When an UberEATS customer places an order it is sent to the restaurant for processing. The restaurant then needs to acknowledge the order and prepare the meal which will take time depending on the complexity of the order and how busy the restaurant is. When the meal is close to being ready, an Uber delivery-partner is dispatched to pick up the meal. Then, the delivery-partner needs to get to the restaurant, find parking, walk inside to get the food, then walk back to the car, drive to the customer's location (which depends on route, traffic, and other factors), find parking, and walk to the customer's door to complete the delivery. The goal is to predict the total duration of this complex multi-stage process, as well as recalculate these time-to-delivery predictions at every step of the process.

On the Michelangelo platform, the UberEATS data scientists use gradient boosted decision tree regression models to predict this end-to-end delivery time. Features for the model include information from the request (e.g., time of day, delivery location), historical features (e.g. average meal prep time for the last seven days), and near-realtime calculated features (e.g., average meal prep time for the last one hour). Models are deployed across Uber's data centers to Michelangelo model serving containers and are invoked via network requests by the UberEATS microservices. These predictions are displayed to UberEATS customers prior to ordering from a restaurant and as their meal is being prepared and delivered.

System architecture

Michelangelo consists of a mix of open source systems and components built in-house. The primary open sourced components used are HDFS, Spark, Samza, Cassandra, MLLib, XGBoost, and TensorFlow. We generally prefer to use mature open source options where possible, and will fork, customize, and contribute back as needed, though we sometimes build systems ourselves when open source solutions are not ideal for our use case.

Michelangelo is built on top of Uber's data and compute infrastructure, providing a data lake that stores all of Uber's transactional and logged data, Kafka brokers that aggregate logged messages from all Uber's services, a Samza streaming compute engine, managed Cassandra clusters, and Uber's in-house service provisioning and deployment tools.

In the next section, we walk through the layers of the system using the UberEATS ETD models as a case study to illustrate the technical details of Michelangelo.

Machine learning workflow

The same general workflow exists across almost all machine learning use cases at Uber regardless of the challenge at hand, including classification and regression, as well as time series forecasting. The workflow is generally implementation-agnostic, so easily expanded to support new algorithm types and frameworks, such as newer deep learning frameworks. It also applies across different deployment modes such as both online and offline (and in-car and in-phone) prediction use cases.

We designed Michelangelo specifically to provide scalable, reliable, reproducible, easy-to-use, and automated tools to address the following six-step workflow:

1. Manage data
2. Train models
3. Evaluate models
4. Deploy models
5. Make predictions
6. Monitor predictions

Next, we go into detail about how Michelangelo's architecture facilitates each stage of this workflow.

Manage data

Finding good features is often the hardest part of machine learning and we have found that building and managing data pipelines is typically one of the most costly pieces of a complete machine learning solution.

A platform should provide standard tools for building data pipelines to generate feature and label data sets for training (and re-training) and feature-only data sets for predicting. These tools should have deep integration with the company’s data lake or warehouses and with the company’s online data serving systems. The pipelines need to be scalable and performant, incorporate integrated monitoring for data flow and data quality, and support both online and offline training and predicting. Ideally, they should also generate the features in a way that is shareable across teams to reduce duplicate work and increase data quality. They should also provide strong guard rails and controls to encourage and empower users to adopt best practices (e.g., making it easy to guarantee that the same data generation/preparation process is used at both training time and prediction time).

The data management components of Michelangelo are divided between online and offline pipelines. Currently, the offline pipelines are used to feed batch model training and batch prediction jobs and the online pipelines feed online, low latency predictions (and in the near future, online learning systems).

In addition, we added a layer of data management, a feature store that allows teams to share, discover, and use a highly curated set of features for their machine learning problems. We found that many modeling problems at Uber use identical or similar features, and there is substantial value in enabling teams to share features between their own projects and for teams in different organizations to share features with each other.

Figure 2: Data preparation pipelines push data into the Feature Store tables and training data repositories.

Offline

Uber’s transactional and log data flows into an HDFS data lake and is easily accessible via Spark and Hive SQL compute jobs. We provide containers and scheduling to run regular jobs to compute features which can be made private to a project or published to the Feature Store (see below) and shared across teams, while batch jobs run on a schedule or a trigger and are integrated with data quality monitoring tools to quickly detect regressions in the pipeline–either due to local or upstream code or data issues.

Online

Models that are deployed online cannot access data stored in HDFS, and it is often difficult to compute some features in a performant manner directly from the online databases that back Uber’s production services (for instance, it is not possible to directly query the UberEATS order service to compute the average meal prep time for a restaurant over a specific period of time). Instead, we allow features needed for online models to be precomputed and stored in Cassandra where they can be read at low latency at prediction time.

We support two options for computing these online-served features, batch precompute and near-real-time compute, outlined below:

- **Batch precompute.** The first option for computing is to conduct bulk precomputing and loading historical features from HDFS into Cassandra on a regular basis. This is simple and efficient, and generally works well for historical features where it is acceptable for the features to only be updated every few hours or once a day. This system guarantees that the same data and batch pipeline is used for both training and serving. UberEATS uses this system for features like a '*restaurant's average meal preparation time over the last seven days*.'
- **Near-real-time compute.** The second option is to publish relevant metrics to Kafka and then run Samza-based streaming compute jobs to generate aggregate features at low latency. These features are then written directly to Cassandra for serving and logged back to HDFS for future training jobs. Like the batch system, near-real-time compute ensures that the same data is used for training and serving. To avoid a cold start, we provide a tool to "backfill" this data and generate training data by running a batch job against historical logs. UberEATS uses this near-realtime pipeline for features like a *'restaurant's average meal preparation time over the last one hour.'*

Shared feature store

We found great value in building a centralized Feature Store in which teams around Uber can create and manage canonical features to be used by their teams and shared with others. At a high level, it accomplishes two things:

1. It allows users to easily add features they have built into a shared feature store, requiring only a small amount of extra metadata (owner, description, SLA, etc.) on top of what would be required for a feature generated for private, project-specific usage.
2. Once features are in the Feature Store, they are very easy to consume, both online and offline, by referencing a feature's simple canonical name in the model configuration. Equipped with this information, the system handles joining in the correct HDFS data sets for model training or batch prediction and fetching the right value from Cassandra for online predictions.

At the moment, we have approximately 10,000 features in Feature Store that are used to accelerate machine learning projects, and teams across the company are adding new ones all the time. Features in the Feature Store are automatically calculated and updated daily.

In the future, we intend to explore the possibility of building an automated system to search through Feature Store and identify the most useful and important features for solving a given prediction problem.

Domain specific language for feature selection and transformation

Often the features generated by data pipelines or sent from a client service are not in the proper format for the model, and they may be missing values that need to be filled. Moreover, the model may only need a subset of features provided. In some cases, it may be more useful for the model to transform a timestamp into an hour-of-day or day-of-week to better capture seasonal patterns. In other cases, feature values may need to be normalized (e.g., subtract the mean and divide by standard deviation).

To address these issues, we created a DSL (domain specific language) that modelers use to select, transform, and combine the features that are sent to the model at training and prediction times. The DSL is implemented as sub-set of Scala. It is a pure functional language with a complete set of commonly used functions. With this DSL, we also provide the ability for customer teams to add their own user-defined functions. There are accessor functions that fetch feature values from the current context (data pipeline in the case of an offline model or current request from client in the case of an online model) or from the Feature Store.

It is important to note that the DSL expressions are part of the model configuration and the same expressions are applied at training time and at prediction time to help guarantee that the same final set of features is generated and sent to the model in both cases.

Train models

We currently support offline, large-scale distributed training of decision trees, linear and logistic models, unsupervised models (k-means), time series models, and deep neural networks. We regularly add new algorithms in response to customer need and as they are developed by Uber's AI Labs and other internal researchers. In addition, we let customer teams add their own model types by providing custom training, evaluation, and serving code. The distributed model training system scales up to handle billions of samples and down to small datasets for quick iterations.

A model configuration specifies the model type, hyper-parameters, data source reference, and feature DSL expressions, as well as compute resource requirements (the number of machines, how much memory, whether or not to use GPUs, etc.). It is used to configure the training job, which is run on a YARN or Mesos cluster.

After the model is trained, performance metrics (e.g., ROC curve and PR curve) are computed and combined into a model evaluation report. At the end of training, the original configuration, the learned parameters, and the evaluation report are saved back to our model repository for analysis and deployment.

In addition to training single models, Michelangelo supports hyper-parameter search for all model types as well as partitioned models. With partitioned models, we automatically partition the training data based on configuration from the user and then train one model per partition, falling back to a parent model when needed (e.g. training one model per city and falling back to a country-level model when an accurate city-level model cannot be achieved).

Training jobs can be configured and managed through a web UI or an API, often via Jupyter notebook. Many teams use the API and workflow tools to schedule regular re-training of their models.

Figure 3: Model training jobs use Feature Store and training data repository data sets to train models and then push them to the model repository.

Evaluate models

Models are often trained as part of a methodical exploration process to identify the set of features, algorithms, and hyper-parameters that create the best model for their problem. Before arriving at the ideal model for a given use case, it is not uncommon to train hundreds of models that do not make the cut. Though not ultimately used in production, the performance of these models guide engineers towards the model configuration that results in the best model performance. Keeping track of these trained models (e.g. who trained them and when, on what data set, with which hyper-parameters, etc.), evaluating them, and comparing them to each other are typically big challenges when dealing with so many models and present opportunities for the platform to add a lot of value.

For every model that is trained in Michelangelo, we store a versioned object in our model repository in Cassandra that contains a record of:

- Who trained the model
- Start and end time of the training job
- Full model configuration (features used, hyper-parameter values, etc.)
- Reference to training and test data sets
- Distribution and relative importance of each feature
- Model accuracy metrics
- Standard charts and graphs for each model type (e.g. ROC curve, PR curve, and confusion matrix for a binary classifier)
- Full learned parameters of the model
- Summary statistics for model visualization

The information is easily available to the user through a web UI and programmatically through an API, both for inspecting the details of an individual model and for comparing one or more models with each other.

Model accuracy report

The model accuracy report for a regression model shows standard accuracy metrics and charts. Classification models would display a different set, as depicted below in Figures 4 and 5:

Figure 4: Regression model reports show regression-related performance metrics.

Figure 5: Binary classification performance reports show classification-related performance metrics.

Decision tree visualization

For important model types, we provide sophisticated visualization tools to help modelers understand why a model behaves as it does, as well as to help debug it if necessary. In the case of decision tree models, we let the user browse through each of the individual trees to see their relative importance to the overall model, their split points, the importance of each feature to a particular tree, and the distribution of data at each split, among other variables. The user can specify feature values and the visualization will depict the triggered paths down the decision trees, the prediction per tree, and the overall prediction for the model, as pictured in Figure 6 below:

Figure 6: Tree models can be explored with powerful tree visualizations.

Feature report

Michelangelo provides a feature report that shows each feature in order of importance to the model along with partial dependence plots and distribution histograms. Selecting two features lets the user understand the feature interactions as a two-way partial dependence diagram, as showcased below:

Figure 7: Features, their impact on the model, and their interactions can be explored though a feature report.

Deploy models

Michelangelo has end-to-end support for managing model deployment via the UI or API and three modes in which a model can be deployed:

1. **Offline deployment.** The model is deployed to an offline container and run in a Spark job to generate batch predictions either on demand or on a repeating schedule.
2. **Online deployment.** The model is deployed to an online prediction service cluster (generally containing hundreds of machines behind a load balancer) where clients can send individual or batched prediction requests as network RPC calls.
3. **Library deployment.** We intend to launch a model that is deployed to a serving container that is embedded as a library in another service and invoked via a Java API. (It is not shown in Figure 8, below, but works similarly to online deployment).

Figure 8: Models from the model repository are deployed to online and offline containers for serving.

In all cases, the required model artifacts (metadata files, model parameter files, and compiled DSL expressions) are packaged in a ZIP archive and copied to the relevant hosts across Uber's data centers using our standard code deployment infrastructure. The prediction containers automatically load the new models from disk and start handling prediction requests.

Many teams have automation scripts to schedule regular model retraining and deployment via Michelangelo's API. In the case of the UberEATS delivery time models, training and deployment are triggered manually by data scientists and engineers through the web UI.

Make predictions

Once models are deployed and loaded by the serving container, they are used to make predictions based on feature data loaded from a data pipeline or directly from a client service. The raw features are passed through the compiled DSL expressions which can modify the raw features and/or fetch additional features from the Feature Store. The final feature vector is constructed and passed to the model for scoring. In the case of online models, the prediction is returned to the client service over the network. In the case of offline models, the predictions are written back to Hive where they can be consumed by downstream batch jobs or accessed by users directly through SQL-based query tools, as depicted below:

Figure 9: Online and offline prediction services use sets of feature vectors to generate predictions.

Referencing models

More than one model can be deployed at the same time to a given serving container. This allows safe transitions from old models to new models and side-by-side A/B testing of models. At serving time, a model is identified by its UUID and an optional tag (or alias) that is specified during deployment. In the case of an online model, the client service sends the feature vector along with the model UUID or model tag that it wants to use; in the case of a tag, the container will generate the prediction using the model most recently deployed to that tag. In the case of batch models, all deployed models are used to score each batch data set and the prediction records contain the model UUID and optional tag so that consumers can filter as appropriate.

If both models have the same signature (i.e. expect the same set of features) when deploying a new model to replace an old model, users can deploy the new model to the same tag as the old model and the container will start using the new model immediately. This allows customers to update their models without requiring a change in their client code. Users can also deploy the new model using just its UUID and then modify a configuration in the client or intermediate service to gradually switch traffic from the old model UUID to the new one.

For A/B testing of models, users can simply deploy competing models either via UUIDs or tags and then use Uber's experimentation framework from within the client service to send portions of the traffic to each model and track performance metrics.

Scale and latency

Since machine learning models are stateless and share nothing, they are trivial to scale out, both in online and offline serving modes. In the case of online models, we can simply add more hosts to the prediction service cluster and let the load balancer spread the load. In the case of offline predictions, we can add more Spark executors and let Spark manage the parallelism.

Online serving latency depends on model type and complexity and whether or not the model requires features from the Cassandra feature store. In the case of a model that does not need features from Cassandra, we typically see P95 latency of less than 5 milliseconds (ms). In the case of models that do require features from Cassandra, we typically see P95 latency of less than 10ms. The highest traffic models right now are serving more than 250,000 predictions per second.

Monitor predictions

When a model is trained and evaluated, historical data is always used. To make sure that a model is working well into the future, it is critical to monitor its predictions so as to ensure that the data pipelines are continuing to send accurate data and that production environment has not changed such that the model is no longer accurate.

To address this, Michelangelo can automatically log and optionally hold back a percentage of the predictions that it makes and then later join those predictions to the observed outcomes (or labels) generated by the data pipeline. With this information, we can generate ongoing, live measurements of model accuracy. In the case of a regression model, we publish R-squared/coefficient of determination, root mean square logarithmic error (RMSLE), root mean square error (RMSE), and mean absolute error metrics to Uber's time series monitoring systems so that users can analyze charts over time and set threshold alerts, as depicted below:

Figure 10: Predictions are sampled and compared to observed outcomes to generate model accuracy metrics.

Management plane, API, and web UI

The last important piece of the system is an API tier. This is the brains of the system. It consists of a management application that serves the web UI and network API and integrations with Uber's system monitoring and alerting infrastructure. This tier also houses the workflow system that is used to orchestrate the batch data pipelines, training jobs, batch prediction jobs, and the deployment of models both to batch and online containers.

Users of Michelangelo interact directly with these components through the web UI, the REST API, and the monitoring and alerting tools.

Building on the Michelangelo platform

In the coming months, we plan to continue scaling and hardening the existing system to support both the growth of our set of customer teams and Uber's business overall. As the platform layers mature, we plan to invest in higher level tools and services to drive democratization of machine learning and better support the needs of our business:

- **AutoML.** This will be a system for automatically searching and discovering model configurations (algorithm, feature sets, hyper-parameter values, etc.) that result in the best performing models for given modeling problems. The system would also automatically build the production data pipelines to generate the features and labels needed to power the models. We have addressed big pieces of this already with our Feature Store, our unified offline and online data pipelines, and hyper-parameter search feature. We plan to accelerate our earlier data science work through AutoML. The system would allow data scientists to specify a set of labels and an objective function, and then would make the most privacy-and security-aware use of Uber's data to find the best model for the problem. The goal is to amplify data scientist productivity with smart tools that make their job easier.
- **Model visualization.** Understanding and debugging models is increasingly important, especially for deep learning. While we have made some important first steps with visualization tools for tree-based models, much more needs to be done to enable data scientists to understand, debug, and tune their models and for users to trust the results.
- **Online learning**. Most of Uber's machine learning models directly affect the Uber product in real time. This means they operate in the complex and ever-changing environment of moving things in the physical world. To keep our models accurate as this environment changes, our models need to change with it. Today, teams are regularly retraining their models in Michelangelo. A full platform solution to this use case involves easily updateable model types, faster training and evaluation architecture and pipelines, automated model validation and deployment, and sophisticated monitoring and alerting systems. Though a big project, early results suggest substantial potential gains from doing online learning right.
- **Distributed deep learning.** An increasing number of Uber's machine learning systems are implementing deep learning technologies. The user workflow of defining and iterating on deep learning models is sufficiently different from the standard workflow such that it needs unique platform support. Deep learning use cases typically handle a larger quantity of data, and different hardware requirements (i.e. GPUs) motivate further investments into distributed learning and a tighter integration with a flexible resource management stack.

If you are interesting in tackling machine learning challenges that push the limits of scale, consider applying for a role on our team!

Jeremy Hermann is an Engineering Manager and Mike Del Balso is a Product Manager on Uber's Machine Learning Platform team.

Comments

Jeremy Hermann

Jeremy Hermann is an engineering manager on Uber's Michelangelo team.

Mike Del Balso

Mike Del Balso is a product manager on Uber’s Machine Learning Platform team.

Get the App →

Engineering

Become a Driver

Contact Us

ubereng@uber.com

UberEngineering

@ubereng

UberEngineering

Uber Engineering Blog Categories

AI

General Engineering

Architecture

Mobile

Backend

Open Source

Culture

Team Profile

Developers

Uber Data

Uber Links

Uber.com

Help

Uber Eats

Newsroom

UberRUSH

Careers

Uber for Business

Uber Open Source

Privacy Policy

Terms and Conditions