Using Machine Learning to Predict Value of Homes On Airbnb

Introduction

Data products have always been an instrumental part of Airbnb's service. However, we have long recognized that it's costly to make data products. For example, personalized search ranking enables guests to more easily discover homes, and smart pricing allows hosts to set more competitive prices according to supply and demand. However, these projects each required a lot of dedicated data science and engineering time and effort.

Recently, advances in Airbnb's machine learning infrastructure have lowered the cost significantly to deploy new machine learning models to production. For example, our ML Infra team built a general feature repository that allows users to leverage high quality, vetted, reusable features in their models. Data scientists have started to incorporate several AutoML tools into their workflows to speed up model selection and performance benchmarking. Additionally, ML infra created a new framework that will automatically translate Jupyter notebooks into Airflow pipelines.

In this post, I will describe how these tools worked together to expedite the modeling process and hence lower the overall development costs for a specific use case of LTV modeling — predicting the value of homes on Airbnb.

What Is LTV?

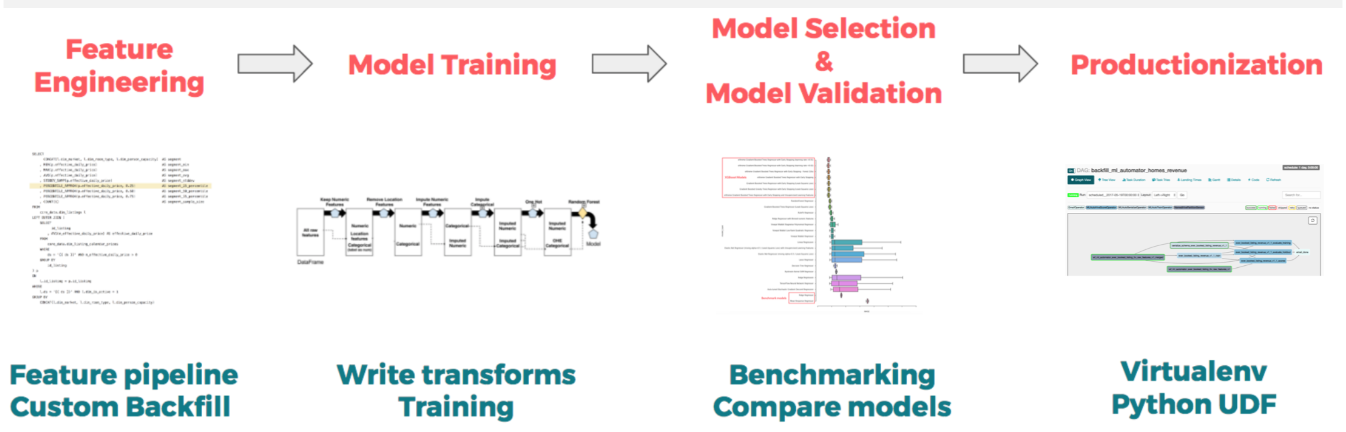
Customer Lifetime Value (LTV), a popular concept among e-commerce and marketplace companies, captures the projected value of a user for a fixed time horizon, often measured in dollar terms.

At e-commerce companies like Spotify or Netflix, LTV is often used to make pricing decisions like setting subscription fees. At marketplace companies like Airbnb, knowing users' LTVs enable us to allocate budget across different marketing channels more efficiently, calculate more precise bidding prices for online marketing based on keywords, and create better listing segments.

While one can use past data to calculate the historical value of existing listings, we took one step further to predict LTV of new listings using machine learning.

Machine Learning Workflow For LTV Modeling

Data scientists are typically accustomed to machine learning related tasks such as feature engineering, prototyping, and model selection. However, taking a model prototype to production often requires an orthogonal set of data engineering skills that data scientists might not be familiar with.


Feature Engineering
Model Training
Model Selection & Model Validation
Productionization
Feature pipeline Custom Backfill
Write transforms Training
Benchmarking Compare models
Virtualenv Python UDF

Luckily, At Airbnb we have machine learning tools that abstract away the engineering work behind productionizing ML models. In fact, we could not have put our model into production without these amazing tools. The remainder of this post is organized into four topics, along with the tools we used to tackle each task:

- **Feature Engineering:** Define relevant features
- **Prototyping and Training:** Train a model prototype
- **Model Selection & Validation:** Perform model selection and tuning
- **Productionization:** Take the selected model prototype to production

Feature Engineering

> ***Tool used: Airbnb's internal feature repository — Zipline***

One of the first steps of any supervised machine learning project is to define relevant features that are correlated with the chosen outcome variable, a process called feature engineering. For example, in predicting LTV, one might compute the percentage of the next 180 calendar dates that a listing is available or a listing's price relative to comparable listings in the same market.

At Airbnb, feature engineering often means writing Hive queries to create features from scratch. However, this work is tedious and time consuming as it requires specific domain knowledge and business logic, which means the feature pipelines are often not easily sharable or even reusable. To make this work more scalable, we developed **Zipline** — a training feature repository that provides features at different levels of granularity, such as at the host, guest, listing, or market level.

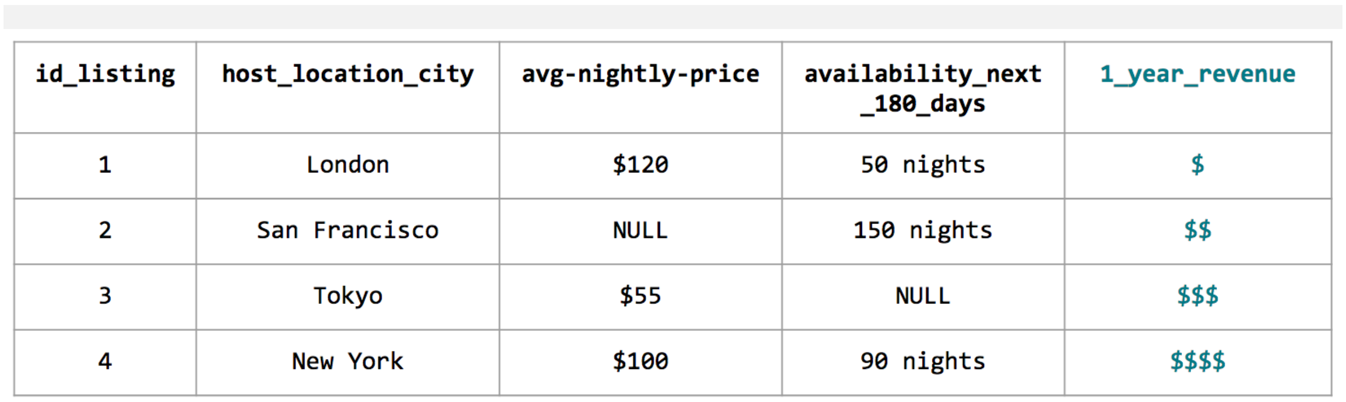
The **crowdsourced** nature of this internal tool allows data scientists to use a wide variety of high quality, vetted features that others have prepared for past projects. If a desired feature is not available, a user can create her own feature with a feature configuration file like the following:

```
source: {
type: hive
query:"""
SELECT
id_listing as listing
, dim_city as city
, dim_country as country
, dim_is_active as is_active
, CONCAT(ds, ' 23:59:59.999') as ts
FROM
core_data.dim_listings
WHERE
ds BETWEEN '{{ start_date }}' AND '{{ end_date }}'
"""
dependencies: [core_data.dim_listings]
is_snapshot: true
start_date: 2010-01-01
}
features: {
city: "City in which the listing is located."
country: "Country in which the listing is located."
is_active: "If the listing is active as of the date partition."
}
```

zipline_example.conf hosted with ♡ by GitHub view raw

will automatically perform intelligent key joins and backfill the training dataset behind the scenes. For the listing LTV model, we used existing Zipline features and also added a handful of our own. In sum, there were over 150 features in our model, including:

- **Location**: country, market, neighborhood and various geography features
- **Price**: nightly rate, cleaning fees, price point relative to similar listings
- **Availability**: Total nights available, % of nights manually blocked
- **Bookability**: Number of bookings or nights booked in the past X days
- **Quality**: Review scores, number of reviews, and amenities



id_listing	host_location_city	avg-nightly-price	availability_next_180_days	1_year_revenue
1	London	$120	50 nights	$
2	San Francisco	NULL	150 nights	$$
3	Tokyo	$55	NULL	$$$
4	New York	$100	90 nights	$$$$

A example training dataset

With our features and outcome variable defined, we can now train a model to learn from our historical data.

Prototyping and Training

> ***Tool used: Machine learning Library in Python — scikit-learn***

As in the example training dataset above, we often need to perform additional data processing before we can fit a model:

- **Data Imputation:** We need to check if any data is missing, and whether that data is missing at random. If not, we need to investigate why and understand the root cause. If yes, we should impute the missing values.
- **Encoding Categorical Variables**: Often we cannot use the raw categories in the model, since the model doesn't know how to fit on strings. When the number of categories is low, we may consider using one-hot encoding. However, when

the cardinality is high, we might consider using ordinal encoding, encoding by frequency count of each category.

In this step, we don't quite know what is the best set of features to use, so writing code that allows us to rapidly iterate is essential. The pipeline construct, commonly available in open-source tools like Scikit-Learn and Spark, is a very convenient tool for prototyping. Pipelines allow data scientists to specify high-level blueprints that describe how features should be transformed, and which models to train. To make it more concrete, below is a code snippet from our LTV model pipeline:

```
transforms = []

transforms.append(
('select_binary', ColumnSelector(features=binary))
)

transforms.append(
('numeric', ExtendedPipeline([
('select', ColumnSelector(features=numeric)),
('impute', Imputer(missing_values='NaN', strategy='mean', axis=0)),
]))
)

for field in categorical:
transforms.append(
(field, ExtendedPipeline([
('select', ColumnSelector(features=[field])),
('encode', OrdinalEncoder(min_support=10))
])
)
)

features = FeatureUnion(transforms)
```

pipeline_example.py hosted with ♡ by GitHub view raw

features, depending on whether these features are of type binary, categorical, or numeric. FeatureUnion at the end simply combines the features column-wise to create the final training dataset.

The advantage of writing prototypes with pipelines is that it abstracts away tedious data transformations using data transforms. Collectively, these transforms ensure that data will be transformed consistently across training and scoring, which solves a common problem of data transformation inconsistency when translating a prototype into production.

Furthermore, pipelines also separates data transformations from model fitting. While not shown in the code above, data scientists can add a final step to specify an estimator for model fitting. By exploring different estimators, data scientists can perform model selection to pick the best model to improve the model's out of sample error.

Performing Model Selection

> ***Tool used: Various AutoML frameworks***

As mentioned in the previous section, we need to decide which candidate model is the best to put into production. To make such a decision, we need to weigh the tradeoffs between model interpretability and model complexity. For example, a sparse linear model might be very interpretable but not complex enough to generalize well. A tree based model might be flexible enough to capture non-linear patterns but not very interpretable. This is known as the **Bias-Variance tradeoff**.

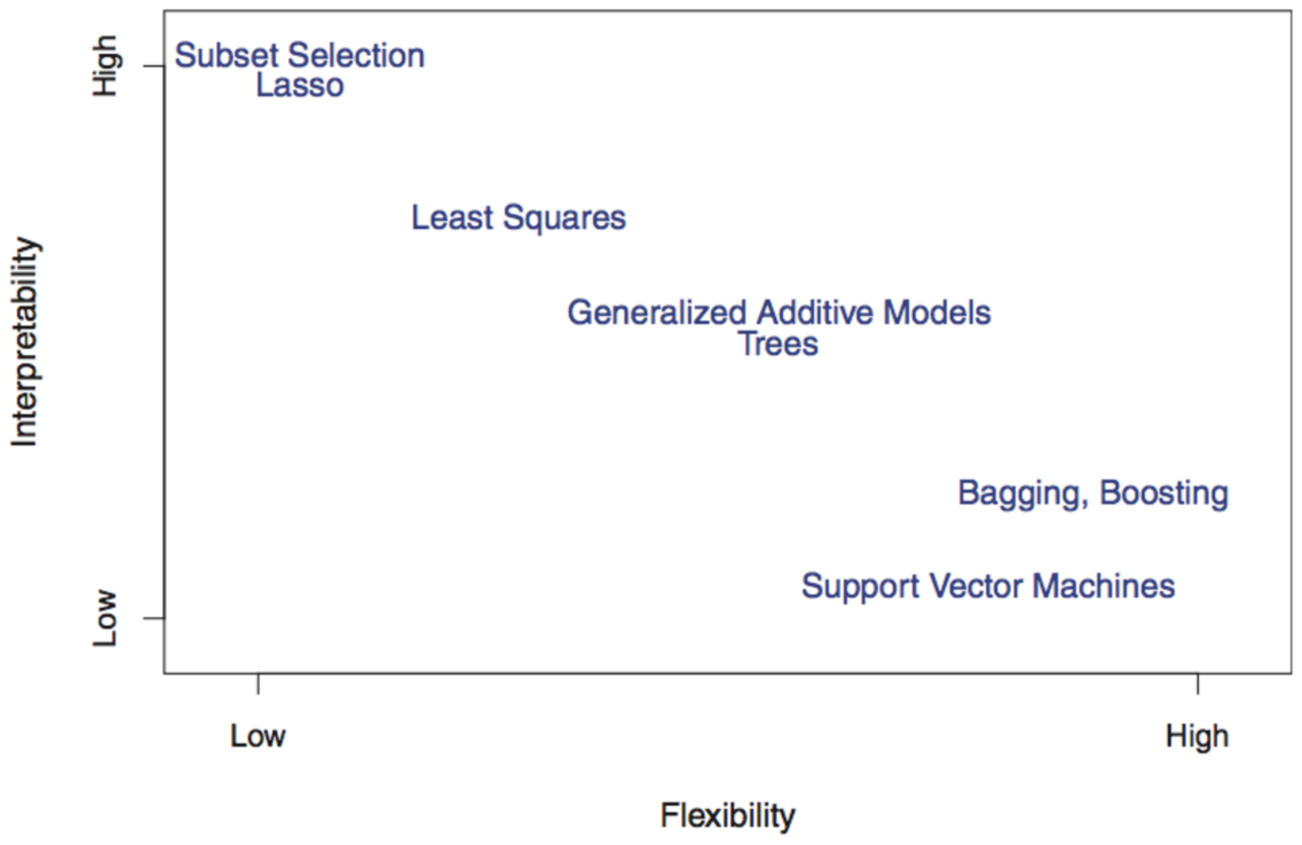
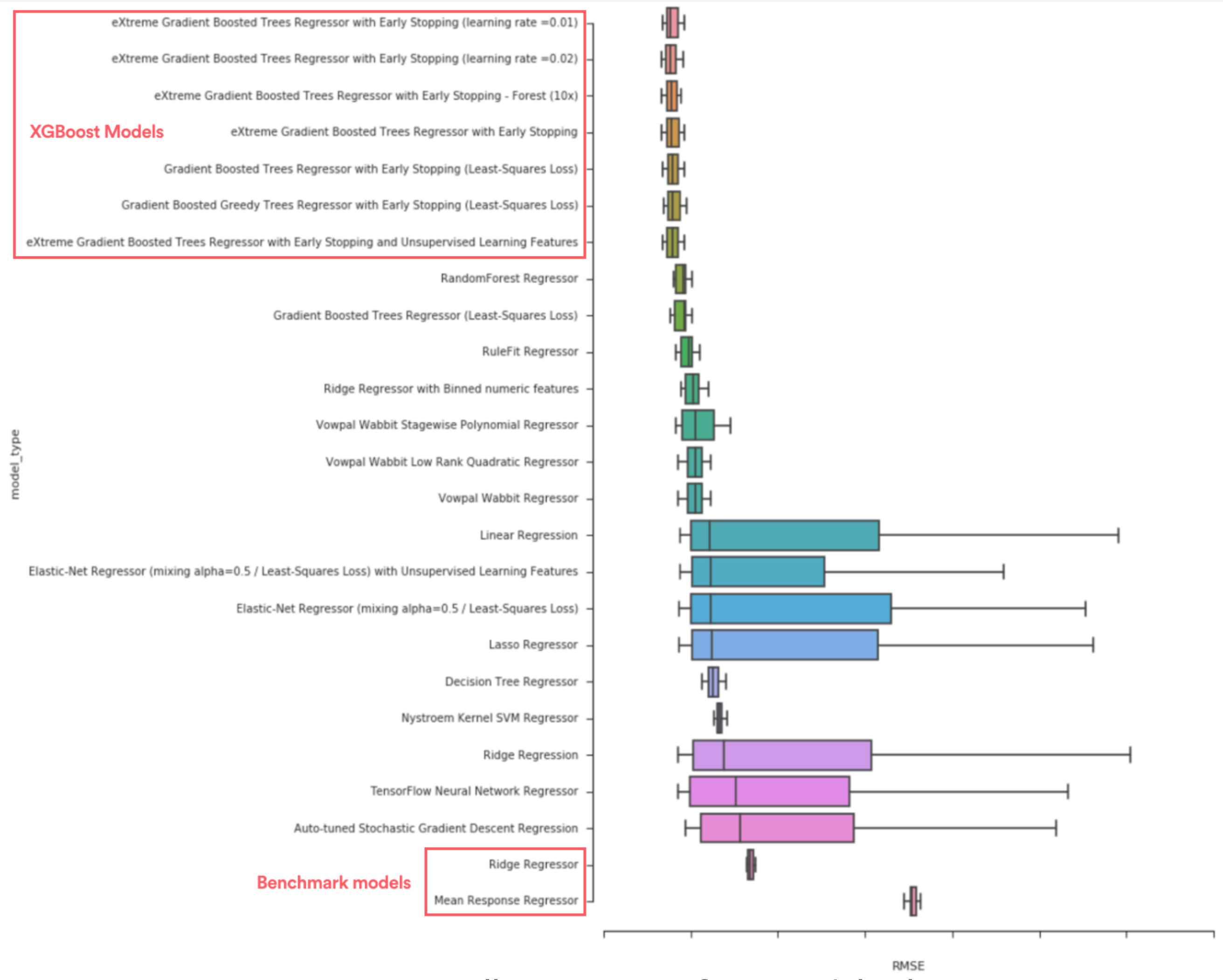


Figure referenced from Introduction to Statistical Learning with R by James, Witten, Hastie, and Tibshirani

In applications such as insurance or credit screening, a model needs to be interpretable because it's important for the model to avoid inadvertently discriminating against certain customers. In applications such as image classification, however, it is much more important to have a performant classifier than an interpretable model.

Given that model selection can be quite time consuming, we experimented with using various AutoML tools to speed up the process. By exploring a wide variety of models, we found which types of models tended to perform best. For example, we learned that eXtreme gradient boosted trees (XGBoost) significantly outperformed benchmark models such as mean response models, ridge regression models, and single decision trees.



Comparing RMSE allows us to perform model selection

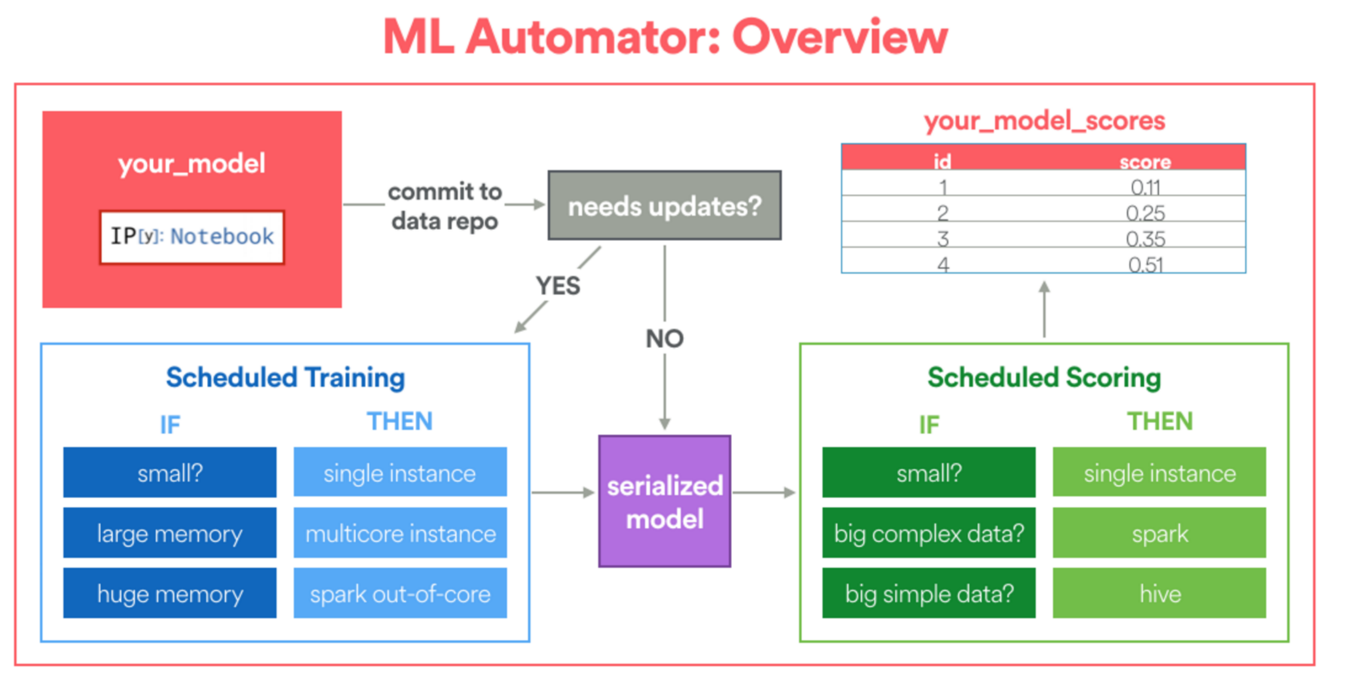
Given that our primary goal was to predict listing values, we felt comfortable productionizing our final model using XGBoost, which favors flexibility over interpretability.

Taking Model Prototypes to Production

> ***Tool used: Airbnb's notebook translation framework — ML Automator***

As we alluded to earlier, building a production pipeline is quite different from building a prototype on a local laptop. For example, how can we perform periodic re-training? How do we score a large number of examples efficiently? How do we build a pipeline to monitor model performance over time?

At Airbnb, we built a framework called **ML Automator** that automagically translates a Jupyter notebook into an Airflow machine learning pipeline. This framework is designed specifically for data scientists who are already familiar with writing prototypes in Python, and want to take their model to production with limited experience in data engineering.



id	score
1	0.11
2	0.25
3	0.35
4	0.51

A simplified overview of the ML Automator Framework (photo credit: Aaron Keys)

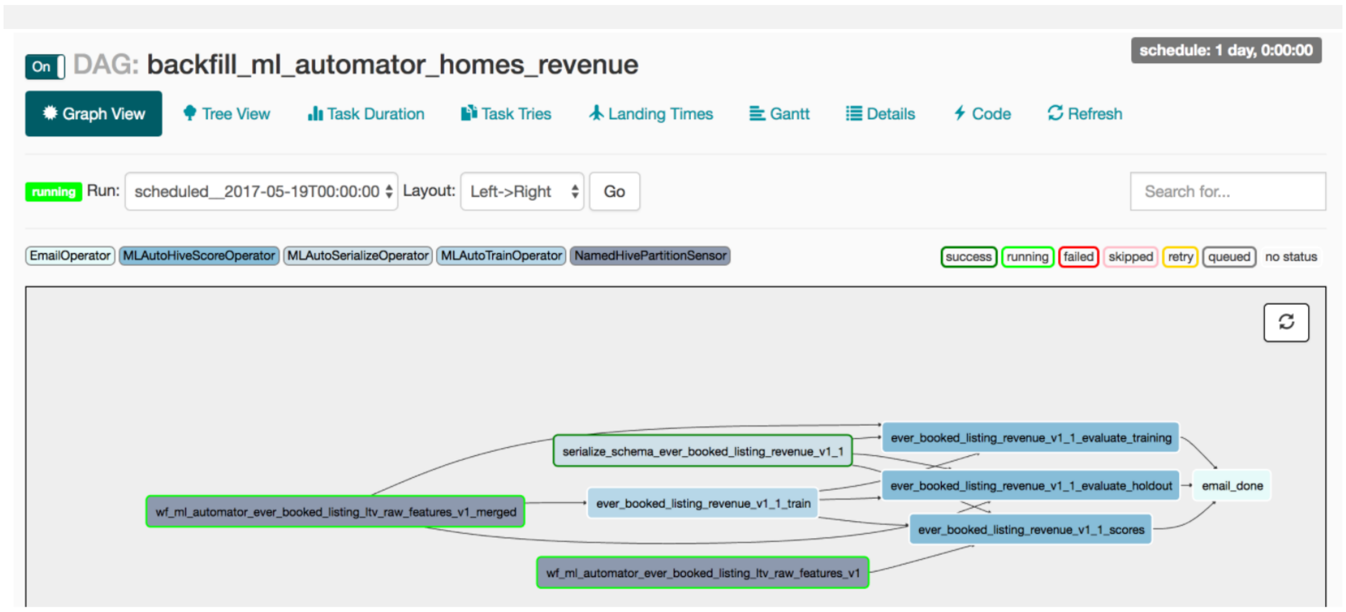
- First, the framework requires a user to specify a model config in the notebook. The purpose of this model config is to tell the framework where to locate the training table, how many compute resources to allocate for training, and how scores will be computed.

- Additionally, data scientists are required to write specific *fit* and *transform* functions. The fit function specifies how training will be done exactly, and the transform function will be wrapped as a Python UDF for distributed scoring (if needed).

Here is a code snippet demonstrating how the *fit* and *transform* functions are defined in our LTV model. The fit function tells the framework that a XGBoost model will be trained, and that data transformations will be carried out according to the pipeline we defined previously.

```python
def fit(X_train, y_train):
import multiprocessing
from ml_helpers.sklearn_extensions import DenseMatrixConverter
from ml_helpers.data import split_records
from xgboost import XGBRegressor

global model

model = {}
n_subset = N_EXAMPLES
X_subset = {k: v[:n_subset] for k, v in X_train.iteritems()}
model['transformations'] = ExtendedPipeline([
('features', features),
('densify', DenseMatrixConverter()),
]).fit(X_subset)

# apply transforms in parallel
Xt = model['transformations'].transform_parallel(X_train)

# fit the model in parallel
model['regressor'] = XGBRegressor().fit(Xt, y_train)

def transform(X):
# return dictionary
global model
Xt = model['transformations'].transform(X)
return {'score': model['regressor'].predict(Xt)}
```

tasks such as data serialization, scheduling of periodic re-training, and distributed scoring are all encapsulated as a part of this daily batch job. As a result, this framework significantly lowers the cost of model development for data scientists, as if there was a dedicated data engineer working alongside the data scientists to take the model into production!


On
DAG: backfill_ml_automator_homes_revenue
schedule: 1 day, 0:00:00
Graph View
Tree View
Task Duration
Task Tries
Landing Times
Gantt
Details
Code
Refresh
running
Run: scheduled__2017-05-19T00:00:00
Layout: Left->Right
Go
Search for...
EmailOperator
MLAutoHiveScoreOperator
MLAutoSerializeOperator
MLAutoTrainOperator
NamedHivePartitionSensor
success
running
failed
skipped
retry
queued
no status
serialize_schema_ever_booked_listing_revenue_v1_1
ever_booked_listing_revenue_v1_1_evaluate_training
ever_booked_listing_revenue_v1_1_evaluate_holdout
email_done
wf_ml_automator_ever_booked_listing_ltv_raw_features_v1_merged
ever_booked_listing_revenue_v1_1_train
ever_booked_listing_revenue_v1_1_scores
wf_ml_automator_ever_booked_listing_ltv_raw_features_v1
A graph view of our LTV Airflow DAG, running in production

Note: *Beyond productionization, there are other topics, such as tracking model performance over time or leveraging elastic compute environment for modeling, which we will not cover in this post. Rest assured, these are all active areas under development.*

Lessons Learned & Looking Ahead

In the past few months, data scientists have partnered very closely with ML Infra, and many great patterns and ideas arose out of this collaboration. In fact, we believe that these tools will unlock a new paradigm for how to develop machine learning models at Airbnb.

- **First, the cost of model development is significantly lower**: by combining disparate strengths from individual tools: Zipline for feature engineering, Pipeline for model prototyping, AutoML for model selection and benchmarking, and finally ML Automator for productionization, we have shortened the development cycle tremendously.
- **Second, the notebook driven design reduces barrier to entry**: data scientists who are not familiar with the framework have immediate access to a plethora of real life examples. Notebooks used in production are guaranteed to

be correct, self-documenting, and up-to-date. This design drives strong adoption from new users.

- **As a result, teams are more willing to invest in ML product ideas**: At the time of this post's writing, we have several other teams exploring ML product ideas by following a similar approach: prioritizing the listing inspection queue, predicting the likelihood that listings will add cohosts, and automating flagging of low quality listings.

We are very excited about the future of this framework and the new paradigm it brought along. By bridging the gap between prototyping and productionization, we can truly enable data scientists and engineers to pursue end-to-end machine learning projects and make our product better.