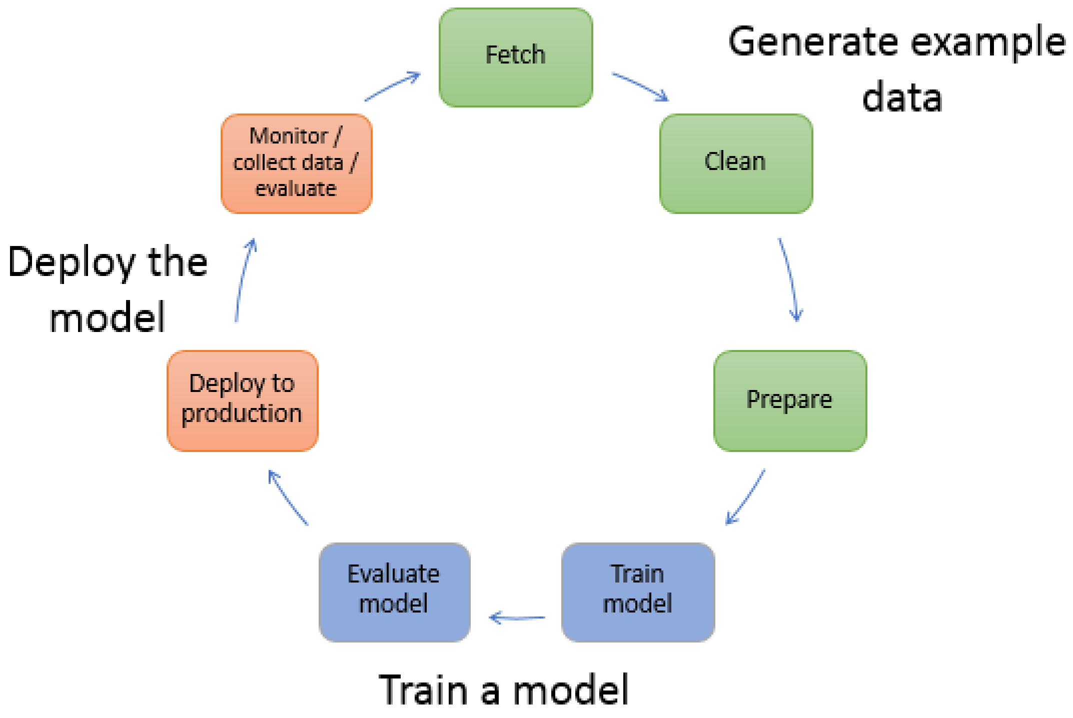
# Digging into AWS SageMaker — First Look

philarmour Follow
Jan 10, 2018 · 14 min read

I'm going to start out right away by admitting that I am no data scientist. In fact, I'm barely even a data science wannabe. However, I am a cloud developer and a system architect, and my knowledge of those fields tells me that the new SageMaker offering from AWS shows a lot of potential. Unsurprisingly, I wanted to dig in and see how true that really is. Going forward, I plan to spend more time on architecture and general AWS stuff than a typical SageMaker user would care for, so stay tuned for my series of posts that continue this investigation! Hopefully, this post will provide some useful technical tidbits and, at the very least, chronicle my first look at SageMaker. Here we go!



General Machine Learning Pipeline

### Scratching the Surface

My first impression of SageMaker is that it's basically a few AWS services (EC2, ECS, S3) cobbled together into an orchestrated set of actions — well this is AWS we're talking about so of course that's what it is! From the console, they tout Notebook instances, Jobs, Models, and Endpoints.

In addition, AWS adds a bunch of value under the covers via the open source SageMaker libraries (Python and Spark). The libraries provide a unified wrapper for many best-in-class algorithms, plus frameworks such as TensorFlow and Apache MXNet for deep learning.

Another huge asset provided by SageMaker is their published docker images for each of the built-in algorithms. These docker images provide consistent training and inference interfaces for each of the algorithms, allowing them to play nicely within the SageMaker ecosystem. We'll touch more on these later.

Most importantly, the published docker images provide a one-stop shop for examples — how to use the algorithms, actual data sets to use, and runnable notebooks that exercise the SageMaker Service components.

Now that sounds pretty awesome! Let's get into it and see what happens.

## Notebook Instances

### Creating a Notebook Instance

These are stupidly simple, but sometimes there's value in simplicity. SageMaker provides a mechanism for easily deploying an EC2 instance, loaded with all the goodies a data scientist could want (Anaconda packages and libraries for common deep learning platforms). It's very spartan with respect to the instance types you can choose (there's currently 3 options) and they only give you 5GB of EBS storage attached — but in this case, simple seems to be good enough!

Next, I focused on the IAM role configuration in the wizard since, from a security standpoint, it's good to restrict as much as possible. With this task, AWS exposes how the fundamentals of the service work. There's a nice review of the permissions required in the docs, and it shows how SageMaker will use lots of S3 (shocker!), ECR for the algorithm Docker images, Cloudwatch and Logs for metrics and logging and, of course, the new SageMaker API actions (i.e. SageMaker).

The last nugget in there is iam:PassRole, which is used to initiate SageMaker resources with the roles we have appropriately scoped.

Now that I started my notebook instance, I have a few questions:

- Why is it taking so long? It took about 5 minutes for my ml.t2.medium to become available. On second glance, this is only slightly longer than a vanilla Linux instance takes in EC2 — so maybe I've been working with Docker too long! 😜
- Where is it? (*read: how do I know how much my AWS bill is going to be?*) According to the pricing page, we're still paying by the hour for these instances, so it would be nice to suspend anything I'm not using

Both answers may be related in a single answer: the notebook is provisioned and managed under the new SageMaker service — you're not going to see it listed as a running EC2 instance. Similar to how RDS or EMR works, SageMaker has a dedicated pool of resources and their own management layer. As they detail here, the notebook creation steps probably have a ways to go for optimization of startup speed.

## Using a Notebook Instance

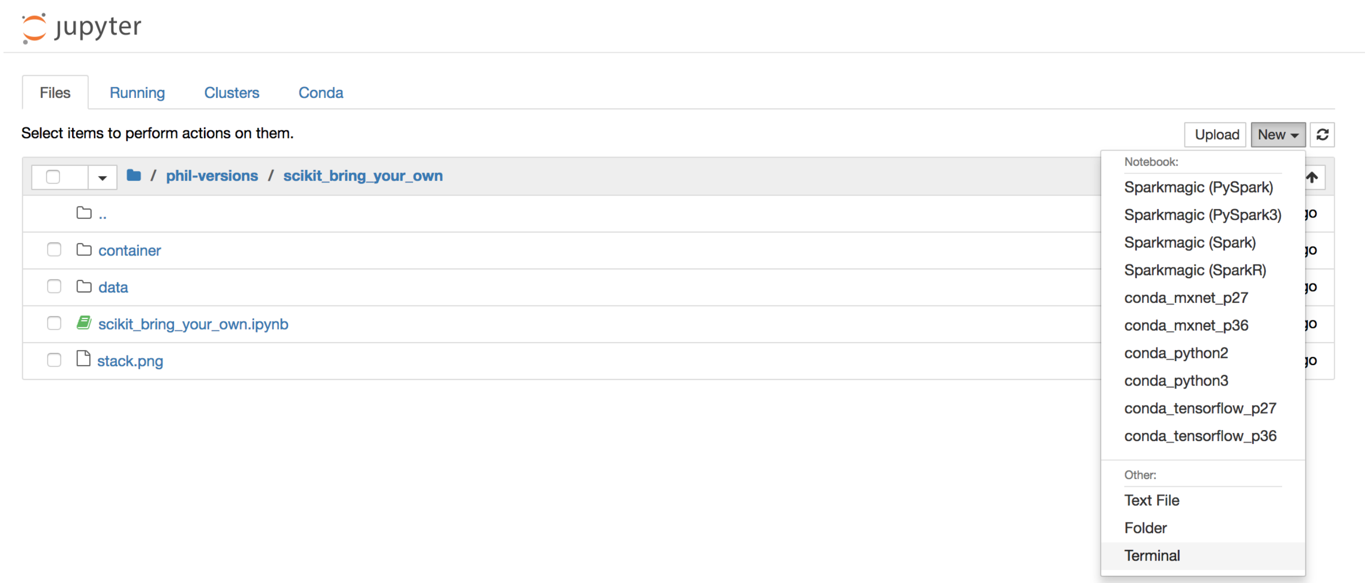
Next question; how do I access the notebook? The console makes it simple by providing a button that silently generates a publicly available, pre-signed notebook instance URL (but uses the same IAM SSO from the web console) that exposes the Jupyter server for the notebook — right there in the browser. This means no more VPNs, bastion hosts, or other elaborate setups just to get data scientists into the AWS environment.

OK, now I'm in and working with the first SageMaker construct, the Notebook! A few things jump out at me immediately:

- The samples from Github are easily available
- The typical Jupyter diagnostics like running notebooks

- The conda tab shows the environments, as well as installed and available packages

Another key menu is the "New" dropdown. Here's where you can see what kinds of pre-packaged environments the nice folks at AWS have created for us. The current list is:



Yet another key tool available in this list is the ability to create a new terminal. I opened one of these before diving into any of the examples. Let's poke around a bit first.

## Hackers Gonna Hack

Let's start by looking at the metadata:

- We can get the ami-id from the local meta-data like so: `curl http://169.254.169.254/latest/meta-data/ami-id`

- We can also see the user-data for the image like so: `curl http://169.254.169.254/latest/user-data`

TONS of good stuff in there. It shows the volume mount, folder structure, user permissions, networking and firewall settings, docker status, and Jupyter server status. (Too deep? Maybe, but it's a good technique to keep in your pocket for trouble-shooting and getting to the bottom of things.)

Another nice thing about having the terminal is that you can easily clean up your Notebook instance, in case you want to move files around or rearrange the files into folders. For example, if you created a copy of one of the sample notebooks, it would be placed in that same directory. If you wanted to move it to the root directory instead, you could do something like:

mv ~/SageMaker/sample-notebooks/introduction_to_amazon_algorithms/xgboost_abalone/xgboost_abalone-Copy1.ipynb ~/SageMaker/my_xgboost_abalone.ipynb

## Don't Let Your Notebooks Go Stale

I also found that after keeping my notebook around for a while the Samples became out-of-sync with the latest from Github. I issued a few commands to change my notebook and clone the latest from Github, giving me an easy way to pull the latest updates into my notebook environment:

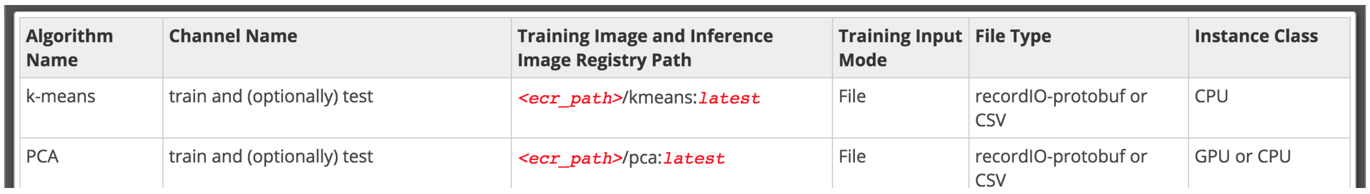
```
cd /home/ec2-user
git clone https://github.com/awslabs/amazon-sagemaker-examples.git
ln -sfn ~/amazon-sagemaker-examples ~/SageMaker/sample-notebooks
```

This actually brings up a larger point: we need to think about how to persist notebooks and other artifacts that we create within the notebook environment. Git seems like the obvious choice, but we need to get the user's Git credentials onto the machine. S3 is going to be easier to pull off, but it feels messy. I'll revisit this!

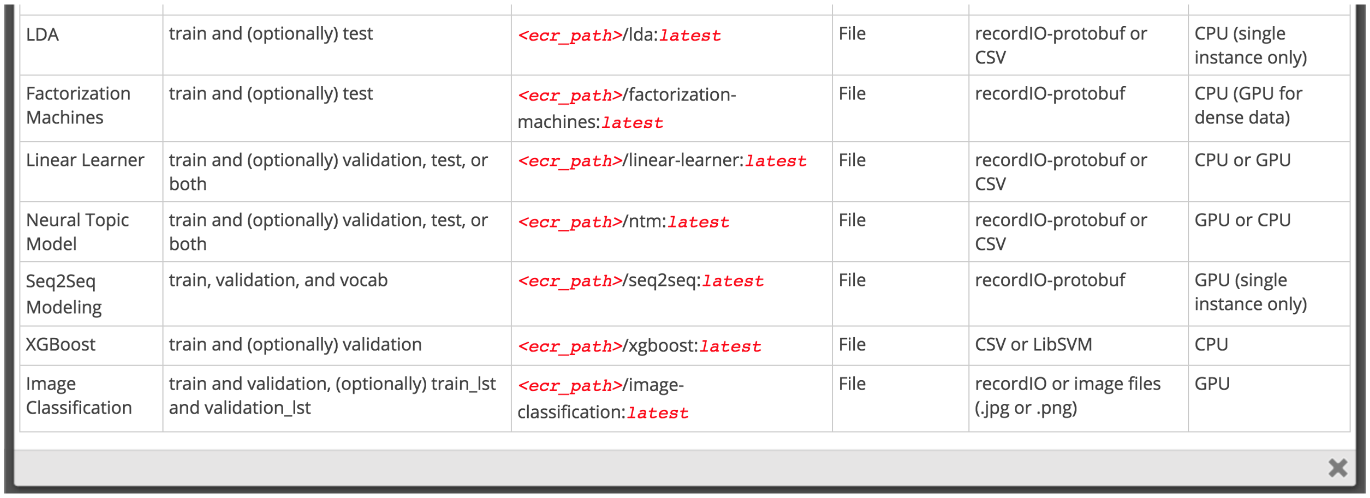
## Learning by Doing (a Sample)

On to my first sample notebook, I chose xgboost_abalone. Seeming like a fairly straight-forward sample set of data, XGBoost trains on labeled, libsvm data and generates a model (more on this later) which can then be loaded into an inference container and used to predict values. Sounds good.

This table from the doc shows the input mode and file type for each of the implemented algorithms:



| Algorithm Name | Channel Name | Training Image and Inference Image Registry Path | Training Input Mode | File Type | Instance Class |
|---|---|---|---|---|---|
| k-means | train and (optionally) test | *<ecr_path>*/kmeans:*latest* | File | recordIO-protobuf or CSV | CPU |
| PCA | train and (optionally) test | *<ecr_path>*/pca:*latest* | File | recordIO-protobuf or CSV | GPU or CPU |



| | | | | | |
|---|---|---|---|---|---|
| LDA | train and (optionally) test | *<ecr_path>*/lda:*latest* | File | recordIO-protobuf or CSV | CPU (single instance only) |
| Factorization Machines | train and (optionally) test | *<ecr_path>*/factorization-machines:*latest* | File | recordIO-protobuf | CPU (GPU for dense data) |
| Linear Learner | train and (optionally) validation, test, or both | *<ecr_path>*/linear-learner:*latest* | File | recordIO-protobuf or CSV | CPU or GPU |
| Neural Topic Model | train and (optionally) validation, test, or both | *<ecr_path>*/ntm:*latest* | File | recordIO-protobuf or CSV | GPU or CPU |
| Seq2Seq Modeling | train, validation, and vocab | *<ecr_path>*/seq2seq:*latest* | File | recordIO-protobuf | GPU (single instance only) |
| XGBoost | train and (optionally) validation | *<ecr_path>*/xgboost:*latest* | File | CSV or LibSVM | CPU |
| Image Classification | train and validation, (optionally) train_lst and validation_lst | *<ecr_path>*/image-classification:*latest* | File | recordIO or image files (.jpg or .png) | GPU |

The columns in this table will make more sense after going through the rest of the sample.

The first section of this sample (and I would imagine *all* of the samples) involves getting access to and prepping the data. We need to partition the labeled data into training, validation and test sets and the training job (more on this in a second) needs to be able to access those data sets. Assuming the notebook code needs to create/modify the data sets, it too needs to have access to the data. For the example an S3 bucket is used to read and write the data sets, and the samples use a heavy dose of boto3 boilerplate like: `boto3.Session().resource('s3').Bucket(bucket).Object(key).upload_fileobj(fobj)`.

In practice, I think we would want to build a pattern for data pipelines. At the very least, we would want an organized data lake that makes the necessary data available for the training and validation step, without the need for a notebook step to prep the data. Surely another idea to revisit as I continue to explore!

## Jobs

### Creating a Training Job

The next section of the sample involves creating and running a training job. I find it interesting that this sample (presumably the others too) uses the AWS SDK to interact with the SageMaker service via their API. It makes sense since we're already writing Python and using boto3, but it's an interesting mix of console and code.

The code from the sample is pretty self-explanatory in how it defines the job. It identifies the ECR image that has the XGBoost algorithm, instructs usage of a particular-sized single instance, points to the training and validation data sets, sets some initial hyperparameters and, **most importantly**, indicates where (in S3) to store the resulting trained model. Then it kicks off the job and waits for it to finish.

## Running the Training Job

Running the job posed a problem initially in that I was trying to use an S3 bucket in the wrong region (different from my SageMaker resources) — I was sad to learn that after 5 minutes of waiting, my training job failed due to not being able to get the data. In hindsight, 5 minutes was probably just the time required to provision and configure the instance (see gripe about slowness above), but since we're talking about ECR images for these training jobs, the wait seems unwarranted. Unfortunately, getting the data seems like a pre-requisite to even start the job.

After creating a new bucket in the correct region with the necessary files, I was finally off and training my model. After approximately 6 minutes my job was done. This training data set was tiny at 2923 rows and 177 Kb — my guess is that the first 5 minutes were dedicated to provisioning the machines, then the actual algorithm ran.

## Distributed Training Jobs

Just for fun, I ran it again with the job configuration changed from 1 instance to 2. Could it possibly half the time required to train the data? Nope, it actually took longer (7 minutes)! Assuming the startup time is so costly, this isn't a very useful test for a small data set. I will need to revisit with a much larger data set and possibly a lot more iterations of the algorithm.

I reviewed the logs from the distributed training run (more on this in a second) and gained some insight into how this works. With this info I was able to develop a good explanation as to why it took longer for this job: it sets up a Hadoop YARN cluster and distributes the job. The single node training job did use YARN and simply ran the training iterations locally on the node.

From the logs:

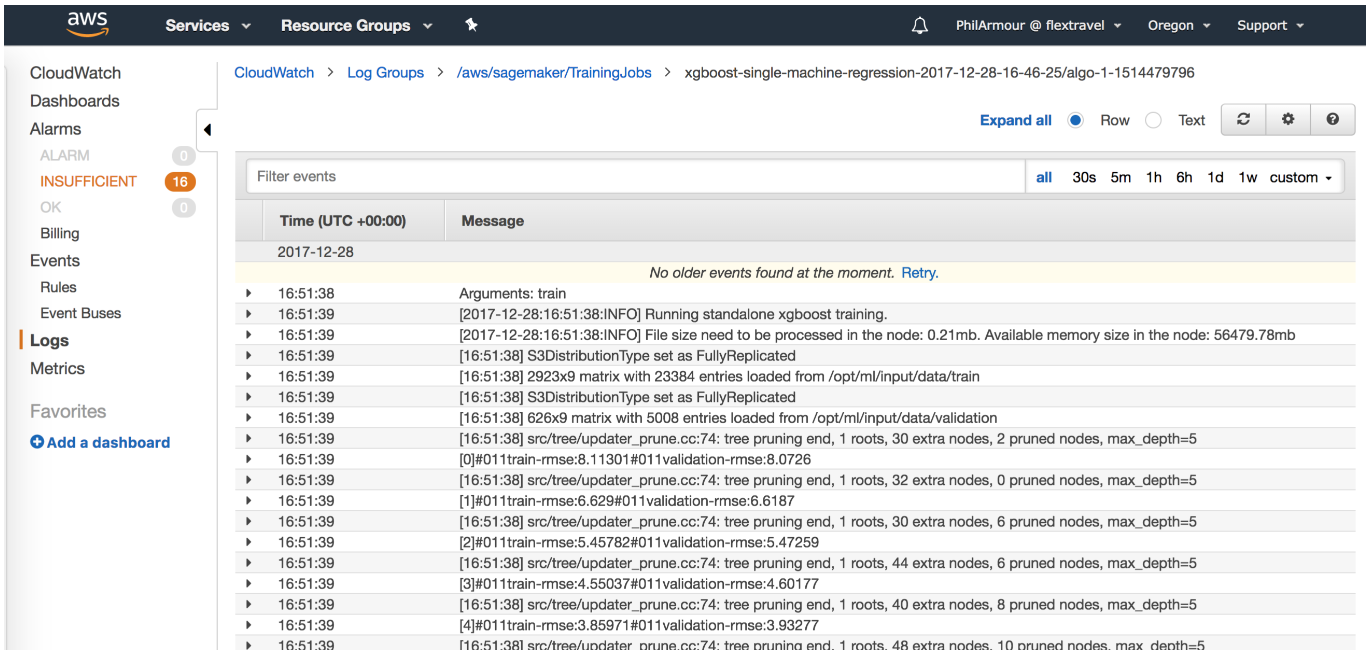
```
Arguments: train
[2017-12-29:17:35:21:INFO] Running distributed xgboost training.
[2017-12-29:17:35:25:INFO] Number of hosts: 2, master IP address:
10.40.0.3, host IP address: 10.32.0.4.
[2017-12-29:17:35:25:INFO] Finished Yarn configuration files setup.
```

```
chown: missing operand after ‘/opt/amazon/hadoop/logs’
Try 'chown --help' for more information.
starting datanode, logging to /opt/amazon/hadoop/logs/hadoop--
datanode-aws.out
Arguments: train
[2017-12-29:17:35:21:INFO] Running distributed xgboost training.
starting nodemanager, logging to /opt/amazon/hadoop/logs/yarn--
nodemanager-aws.out
[2017-12-29:17:35:25:INFO] Number of hosts: 2, master IP address:
10.40.0.3, host IP address: 10.40.0.3.
[2017-12-29:17:35:25:INFO] Finished Yarn configuration files setup.
```

This is very cool! I literally changed a `1` to a `2` and they took care of all the overhead associated with managing the cluster and distributing the work.

## Logging Data from Training Instances

Okay, I ran a training job, but what really happened? To access the logs you need to go to CloudWatch Log. Specifically, SageMaker created a log group called /aws/sagemaker/TrainingJobs and started logging to a log stream for each instance running the training job.



```
Time (UTC +00:00)   Message
2017-12-28
                    No older events found at the moment. Retry.
16:51:38            Arguments: train
16:51:39            [2017-12-28:16:51:38:INFO] Running standalone xgboost training.
16:51:39            [2017-12-28:16:51:38:INFO] File size need to be processed in the node: 0.21mb. Available memory size in the node: 56479.78mb
16:51:39            [16:51:38] S3DistributionType set as FullyReplicated
16:51:39            [16:51:38] 2923x9 matrix with 23384 entries loaded from /opt/ml/input/data/train
16:51:39            [16:51:38] S3DistributionType set as FullyReplicated
16:51:39            [16:51:38] 626x9 matrix with 5008 entries loaded from /opt/ml/input/data/validation
16:51:39            [16:51:38] src/tree/updater_prune.cc:74: tree pruning end, 1 roots, 30 extra nodes, 2 pruned nodes, max_depth=5
16:51:39            [0]#011train-rmse:8.11301#011validation-rmse:8.0726
16:51:39            [16:51:38] src/tree/updater_prune.cc:74: tree pruning end, 1 roots, 32 extra nodes, 0 pruned nodes, max_depth=5
16:51:39            [1]#011train-rmse:6.629#011validation-rmse:6.6187
16:51:39            [16:51:38] src/tree/updater_prune.cc:74: tree pruning end, 1 roots, 30 extra nodes, 6 pruned nodes, max_depth=5
16:51:39            [2]#011train-rmse:5.45782#011validation-rmse:5.47259
16:51:39            [16:51:38] src/tree/updater_prune.cc:74: tree pruning end, 1 roots, 44 extra nodes, 6 pruned nodes, max_depth=5
16:51:39            [3]#011train-rmse:4.55037#011validation-rmse:4.60177
16:51:39            [16:51:38] src/tree/updater_prune.cc:74: tree pruning end, 1 roots, 40 extra nodes, 8 pruned nodes, max_depth=5
16:51:39            [4]#011train-rmse:3.85971#011validation-rmse:3.93277
16:51:39            [16:51:38] src/tree/updater_prune.cc:74: tree pruning end, 1 roots, 48 extra nodes, 10 pruned nodes, max_depth=5
```

As you can see (or know from past experience), navigating logs in the AWS console is kind of painful. Instead I prefer to use one 2 tools for watching CloudWatch logs (*read: tailing the log*): cwtail or awslogs. I prefer the former (but it's Node), while the latter is Python and might be more natural for most folks following along.

I ran each locally right before I initiated the training job like so:

```
npm install cwtail
./node_modules/.bin/cwtail -e -n 5 -f /aws/sagemaker/TrainingJobs
```

or

```
pip install awslogs
awslogs get /aws/sagemaker/TrainingJobs ALL --watch --aws-region us-
west-2
```

Both result in the logs streaming into your terminal from the training instances!

## The Trained Model

Sure enough, after running a successful training job, I can see the serialized model artifact in my S3 bucket: model.tar.gz (and it's nested in a prefix with the job name and the word output — not to mention some extra prefix cruft from the sample notebook). Naming and organization is another area that requires planning to create a clean and consistent platform for creating, training, and retraining multiple models across a team or enterprise.

Now to take a deeper look at the resulting serialized model artifact. But first we need to get it into an easy working environment — I choose to pull it onto the Notebook instance.

```
# Copy the model file from S3 to the Notebook server

bucket = 'my_bucket_name'
key= 'my/long/and/convoluted/key'  #ends in /model.tar.gz in my case
filename = 'model.tar.gz'

boto3.Session().resource('s3').Bucket(bucket).download_file(key,
filename)
```

In the next cell of your notebook use a shell command to unzip the archive like so:

```
!tar -xzvf model.tar.gz
!ls -al
```

In my case it unzipped the archive into a new field called xgboost-model. Names seems pretty self-explanatory, so let's get XGBoost installed in our Notebook and take a look. Here I used another shell command to install via conda:

```
!conda install -y -c conda-forge xgboost
```

(*Note: for this install i read somewhere that it requires Anaconda with Python3 — so be sure your notebook kernel is set to* `conda_python3` )

Now here's the initial code to load the model from the file:

```
import xgboost as xgb

unzipped_model = 'xgboost-model'

booster = xgb.Booster()
booster.load_model(unzipped_model)
```

It worked! Now we can inspect all kinds of fun things like:

```
# Get the feature importance of each feature:
booster.get_fscore()

# Plot the tree and weights:
from xgboost import to_graphviz

to_graphviz(bt._Booster)
```

Or even perform local predictions!

## Models

The next resource type listed in the SageMaker dashboard is Models, and based on my success in the last step, you would think that I created a model. Not so fast! The models in API are more about the metadata that defines a model, including a *pointer* to the actual serialized model artifact. The next step in the sample notebook walks you through using the boto3 SDK to create the model. It consists of a name, and IAM role, the ECR image (same one as before), and the path to the trained model. That's all there is to it.

## Endpoints

The last resource type listed on the SageMaker dashboard is the endpoint. This is the runtime setup that provides inferences based on the trained model. Like the other components (Notebooks and Jobs), these instances are fully managed by SageMaker and are exposed via an HTTP endpoint that can be invoked a couple of ways.

### Creating the Endpoint Configuration

Getting back to the sample notebook, the endpoint configuration is similar to the Model creation in that it's really just a bunch of metadata that describes how to route traffic to one or more models. This is where we could introduce A/B testing of 2 competing models or possibly do canary testing to slowly introduce a new model. The endpoint configuration also indicates the instance count and type to be used for running each model.

### Creating the Endpoint

Once the endpoint configuration is created, creating the endpoint is just a formality. Using boto3 again, we provide a name and the endpoint configuration to use, then we kick off the provisioning process for the actual endpoint instances and the network routing required to make the endpoint accessible. After about 11 minutes, my single instance endpoint was available.

### Testing the Endpoint

As seen in the sample, it's really easy to invoke the endpoint via the boto3 SDK:

```
runtime = boto3.Session().client('runtime.sagemaker')
response = runtime.invoke_endpoint(EndpointName=endpoint_name,
                                   ContentType='text/csv',
                                      Body=payload)
result = json.loads(response[‘Body’].read().decode())
```

What about plain old `curl`? I used Postman as a bit of a crutch since it has a nice ability to do the authorization handshake with AWS and get the signature to put into the Authorization header for the request. From there, I generated the following curl command:

```
curl -X POST \
  https://runtime.sagemaker.us-west-
2.amazonaws.com/endpoints/XGBoostEndpoint-2017-12-28-17-39-
56/invocations \
  -H 'Accept: text/plain' \
  -H 'Authorization: AWS4-HMAC-SHA256 Credential=AKXXX/00000000/us-
west-2/sagemaker/aws4_request, SignedHeaders=accept;content-
length;content-type;host;x-amz-date, Signature=XXXX' \
  -H 'Cache-Control: no-cache' \
  -H 'Content-Type: text/x-libsvm' \
  -H 'Postman-Token: XXX' \
  -H 'X-Amz-Date: 20171228T182622Z' \
  -d '10 1:2 2:0.69 3:0.58 4:0.195 5:1.658 6:0.708 7:0.3615
8:0.4715'
```

and it worked as expected!

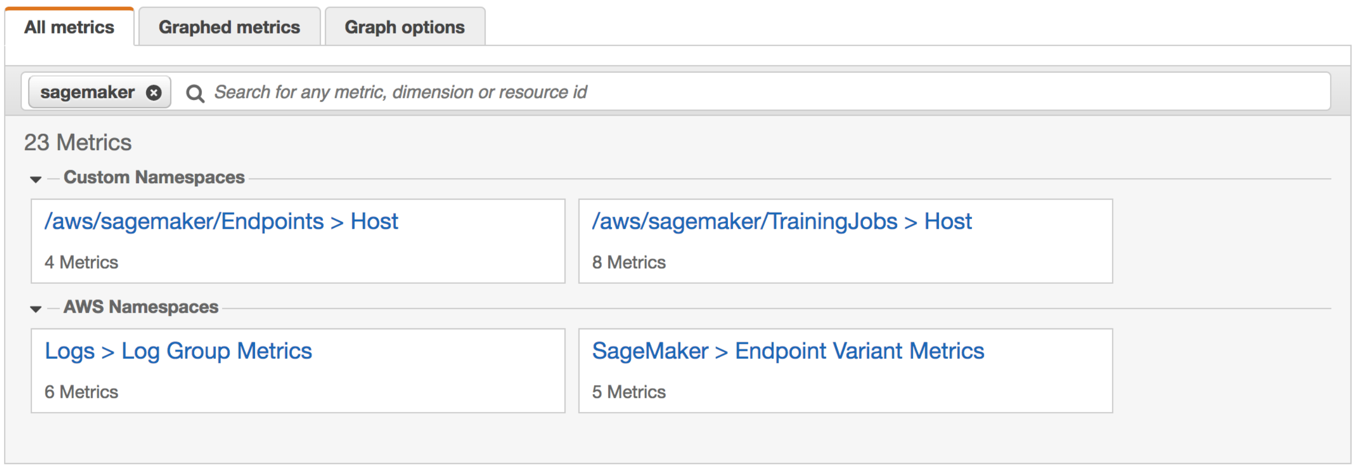
```
Connection →keep-alive

Content-Length →13

Content-Type →text/csv; charset=utf-8

Date →Thu, 28 Dec 2017 18:26:23 GMT

x-Amzn-Invoked-Production-Variant →AllTraffic

11.9059553146
```

Note that I determined the endpoint URL to use for the direct HTTP request from the AWS console screen for the endpoint. However, it’s also fairly easy to build with info from the API documentation.

## CloudWatch Metrics

There’s more info about the training process and the endpoints runtime. These are typical AWS metrics available via CloudWatch, although they don’t yet appear to be supported by DataDog (which we typically use rather than CloudWatch directly).

The metrics break out into 3 groups:


All metrics | Graphed metrics | Graph options

sagemaker ⊗ Search for any metric, dimension or resource id

23 Metrics

Custom Namespaces

/aws/sagemaker/Endpoints > Host
4 Metrics

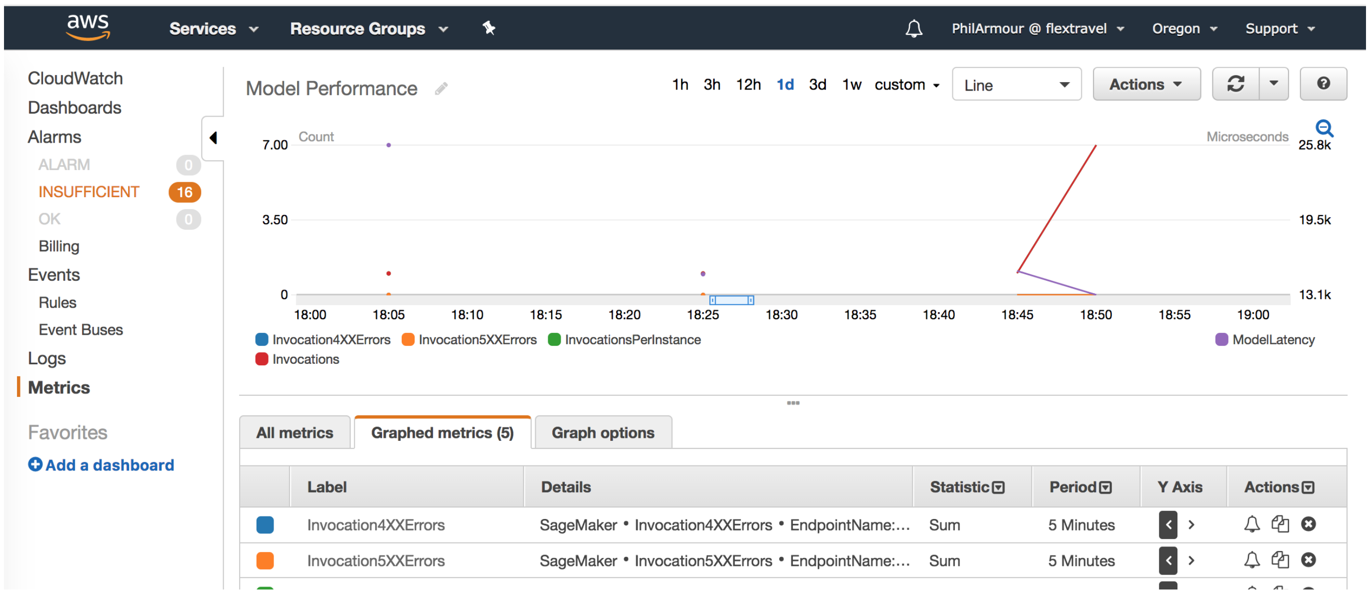
/aws/sagemaker/TrainingJobs > Host
8 Metrics

AWS Namespaces

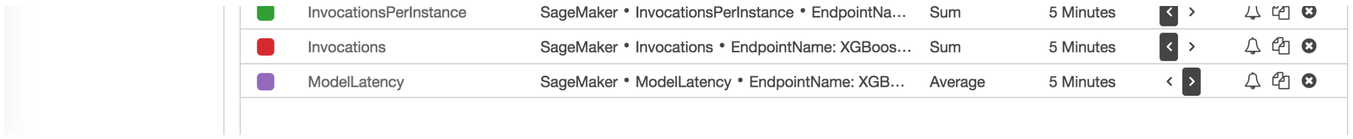
Logs > Log Group Metrics
6 Metrics

SageMaker > Endpoint Variant Metrics
5 Metrics

- TrainingJobs: consists of the host metrics for each training instances (CPU and memory utilization)
- Endpoint: also consists of host metrics for each of the endpoint instances (CPU and memory utilization)
- Endpoint Variant Metrics: consist of the interesting values for the runtime environment including invocations, errors, and latency stats.



| | Label | Details | Statistic | Period | Y Axis | Actions |
|---|---|---|---|---|---|---|
| 🟦 | Invocation4XXErrors | SageMaker • Invocation4XXErrors • EndpointName:... | Sum | 5 Minutes | | |
| 🟧 | Invocation5XXErrors | SageMaker • Invocation5XXErrors • EndpointName:... | Sum | 5 Minutes | | |



| | | | | |
|---|---|---|---|---|
| 🟩 | InvocationsPerInstance | SageMaker • InvocationsPerInstance • EndpointNa... | Sum | 5 Minutes |
| 🟥 | Invocations | SageMaker • Invocations • EndpointName: XGBoos... | Sum | 5 Minutes |
| 🟪 | ModelLatency | SageMaker • ModelLatency • EndpointName: XGB... | Average | 5 Minutes |

## Endpoint Logs

Similarly, the Endpoint instances emit logs via CloudWatch Logs. Each endpoint creates a log group named with the pattern /aws/sagemaker/Endpoints/endpoint_name. Within the log group, each instance logs in its own log stream. Again, I used cwtail to watch the logs as the endpoint responds to requests for inferences:

```
./node_modules/.bin/cwtail -e -n 5 -f
/aws/sagemaker/Endpoints/XGBoostEndpoint-2017-12-29-18-14-56
```

There's a slight delay, but the logs show up.

That's it! This sample provided a great way for me to test a bunch of the basic building blocks that make up SageMaker. It especially helped me learn how they fit into the rest of the AWS ecosystem. In case it wasn't clear, my goal was to determine how the services and tooling could integrate with our development process and, hopefully, how we might be able to operationalize it!

## Tearing Down

A few final thoughts on cleaning up and containing costs. Notebooks can easily be stopped and started. This is great because you don't have to pay for those machines when they're not in use, and when you need it, it's a lot faster to start the existing instance rather than provision a whole new instance. There's probably some risk that AWS released new versions and additional libraries over time, so making new notebook instances from time to time might make sense. The other risk of a long-lived notebook is creating code and artifacts that only reside on the notebook instance. As mentioned earlier, you should combat this with git/S3 and lots of discipline!

Job instances are automatically terminated when the job is finished (or terminated) so there's not much teardown required here.

The endpoint instances are a place where costs can add up quickly. These instances are typically a lot larger than the notebook instances, so if you're not using an endpoint, you really should turn it down. Now here's the catch: you can't simply stop the endpoint instance, you need to delete it. This isn't a huge deal, since it's trivial to define a new endpoint based on your pre-existing endpoint configuration.

## What's Next?

I plan to continue to explore more areas in SageMaker as well as start building on some of the ideas mentioned in this post. Specially I plan to look into:

- Roll your own image for training / inference deployment
- Faster training through scaling up
- Better processes for validating models
- Review Other algorithms
- Use of protocol buffers
- Writing standards/best practices for working with SageMaker

. . .

If this post piqued your interest, check out our open engineering positions and start working with Phil at Upside today: http://upsd.io/2DhbOY8

AWS Machine Learning

About Help Legal