Get started with MLOps

Hands-on Tutorials

A comprehensive MLOps tutorial with open source tools

People vector created by pch.vector — www.freepik.com

Getting machine learning (ML) models into production is hard work. Depending on the level of ambition, it can be surprisingly hard, actually. In this post, I'll go over my personal thoughts (with implementation examples) on principles suitable for the journey of putting ML models into production within a regulated industry; i.e., when everything needs to be auditable, compliant, and in control — a situation where a hacked together API deployed on an EC2 instance is not going to cut it.

Machine Learning Operations (MLOps) refers to an approach where a combination of DevOps and software engineering is leveraged in a manner that enables deploying and maintaining ML models in production reliably and efficiently. Plenty of information can be found online discussing the conceptual ins and outs of MLOps, so instead, this article will focus on being pragmatic with a lot of hands-on code, etc., basically setting up a proof of concept MLOps framework based on open-source tools. You can find the final code on GitHub.

It is all about getting ML models into production; but what does that mean? For this post, I will consider the following list of concepts, which I think should be considered as part of an MLOps framework:

1. **Development platform:** a collaborative platform for performing ML experiments and empowering the creation of ML models by data scientists should be considered part of the MLOps framework. This platform should enable secure access to data sources (e.g., from data engineering workflows). We want the handover from ML training to deployment to be as smooth as possible, which is more likely the case for such a platform than ML models developed in different local environments.
2. **Model unit testing:** every time we create, change, or retrain a model, we should automatically validate the integrity of the model, e.g.
   - should meet minimum performance on a test set
   - should perform well on synthetic use case-specific datasets
3. **Versioning:** it should be possible to go back in time to inspect everything relating to a given model, e.g., what data & code was used. Why? Because if something breaks, we need to be able to go back in time and see why.
4. **Model registry:** there should be an overview of deployed & decommissioned ML models, their version history, and the deployment stage of each version. Why? If something breaks, we can roll back a previously archived version back into production.
5. **Model Governance:** only certain people should have access to see training related to any given model. There should be access control for who can request/reject/approve transitions between deployment stages (e.g., dev to test to prod) in the model registry.
6. **Deployments:** deployment can be many things, but in this post, I consider the case where we want to deploy a model to cloud infrastructure and expose an API, which enables other people to consume and use the model, i.e., I'm not considering cases where we want to deploy ML models into embedded systems. Efficient model deployments on appropriate infrastructure should:
   - support multiple ML frameworks + custom models
   - have well-defined API spec (e.g., Swagger/OpenAPI)
   - support containerized model servers
7. **Monitoring:** tracking performance metrics (throughput, uptime, etc.). Why? If suddenly a model starts returning errors or being unexpectedly slow, we need to know before the end-user complains so that we can fix it.
8. **Feedback:** we need to feedback information to the model on how well it is performing. Why? Typically we run predictions on new samples where we do not yet know the ground truth. As we learn the truth, however, we need to inform the model to report on how well it is actually doing.
9. **A/B testing:** no matter how solid cross-validation we think we're doing, we never know how the model will perform until it actually gets deployed. It should be easy to perform A/B experiments with live models within the MLOps framework.

10. **Drift detection:** typically, the longer time a given model is deployed, the worse it becomes as circumstances change compared to when the model was trained. We can try to monitor and alert on these different circumstances, or “drifts” before they get too severe:
    - *Concept drift:* when the relation between input and output has changed
    - *Prediction drift:*changes in predictions, but the model still holds
    - *Label drift:* change in the model’s outcomes compared to training data
    - *Feature drift:* change in the distribution of model input data
11. **Outlier detection:** if a deployed model receives an input sample that is significantly different from anything observed during training, we can try to identify this sample as a potential outlier, and the returned prediction should be marked as such, indicating that the end-user should be careful in trusting the prediction.
12. **Adversarial Attack Detection:** we should be warned when adversarial samples attack our models (e.g., someone trying to abuse/manipulate the outcome of our algorithms).
13. **Interpretability:** the ML deployments should support endpoints returning the explanation of our prediction, e.g., through SHAP values. Why? for a lot of use cases, a prediction is not enough, and the end-user needs to know why a given prediction was made.
14. **Governance of deployments:** we not only need access restrictions on who can see the data, trained models, etc., but also on who can eventually use the deployed models. These deployed models can often be just as confidential as the data they were trained on.
15. **Data-centricity:** rather than focus on model performance & improvements, it makes sense that an MLOps framework also enables an increased focus on how the end-user can improve data quality and breadth.

Of course above tenets are in addition to normal best practices for DevOps and software engineering in terms of infrastructure as code, documentation, unit tests, CI/CD, etc. Having explored the application of various solutions for MLOps (while considering the above points), the solution I will be exploring in this blog post is a combination of mlflow + Seldon-core — the advantages of these are that there is no need for expensive subscriptions. They are mostly open-source, meaning we can jump right into setting things up and trying them out 🙌 ... I will go through above points one by one to see how this stack addresses each challenge. This is quite a lot to cover, so maybe grab a coffee before reading on. ☕

PS: I’m not affiliated with the mlflow or Seldon-core teams. Although this post contains a lot of code for these specific frameworks, this post comes from a technology-agnostic mindset trying to learn more about the challenges above.

1. Development

There are multiple tools out there that address ML experiment management and collaboration for data scientists, some examples being Neptune.ai, Weights & Biases, Comet.ml, and mlflow. Here we will focus on mlflow, which has both an open-source version as well as a managed Databricks solution. We can deploy Databricks as part of our cloud infrastructure (no upfront cost, pay per use), giving us:

- A solid foundation for data engineering workflows (which will not be addressed in this post). This data, which could be highly confidential and potentially could be something we are not allowed to keep on local computers, is needed for training our models.
- A scalable and collaborative platform for running data science experiments within notebooks on actual data, very similar to jupyter notebooks.
- Versioning of both the data and the code being used to train a given ML model, as well as information relating to model training (through tools Delta Lake, Jobs API, and mlflow)
- User management, which, e.g., in the case of Azure Databricks, can link up with Azure AD, as used by many corporates. This allows us to very finely control who can see what and do what with our models.

To test databricks+mlflow, the first thing to do is to create some infrastructure . In this case, we'll deploy to Azure using terraform, terraform being a tool for easily provisioning infrastructure across different cloud providers. We will keep it simple and simply run the CLI command `terraform init && terraform apply` within a folder with the following script:

Terraform script for setting up Databricks workspace and a blob storage container in Azure. Code by author.

Running the above script through Terraform sets up a Databricks workspace on your Azure account — if you navigate to the created Databricks resource in the Azure Portal, you should be able to click "Launch Workspace," which will send you to your newly created Databricks workspace; so far so good. 👌

Now that we have Databricks up and running, we need to train an ML model and log it into mlflow — this can be done for any kind of custom model, but mlflow provides a lot of convenience functions when we're using popular frameworks — in this case, we'll take advantage of this mlflow magic and train a classic Keras model for classifying hand-written digits in the MNIST dataset. Navigating to the "Workspace" tab on the left in Databricks, we can create a notebook and run the following snippet:

Python code for training MNIST classifier with Keras and logging results to mlflow. Code by author.

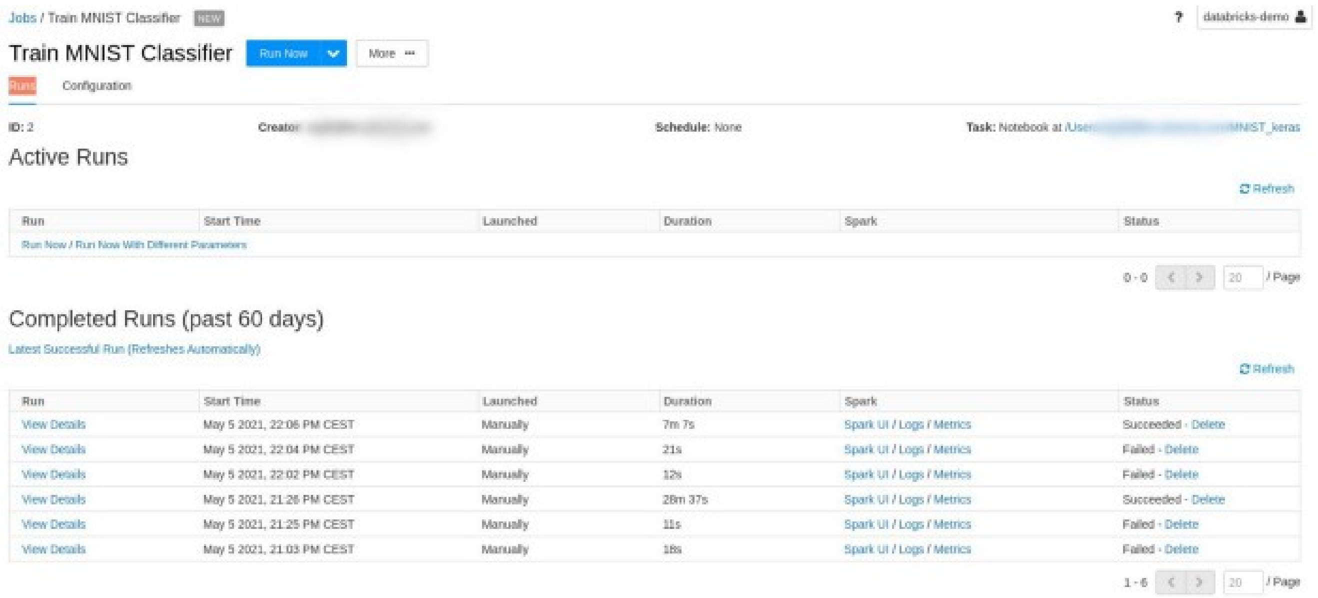
Note: to able to run the notebook, you must attach it to a cluster, but this can easily be created using the UI.

2. Model Unit Testing

Databricks enable us to define 'jobs,' which lets us run a given notebook with a set of input parameters. We can use this to bundle up our training code so that when we wish to re-train our model, we simply need to re-run our job, which could be done with a simple REST request, instead of manually going into the training notebook and re-training the model. A crucial part of this kind of "operationalization" of our model training is that we include checks within the notebook to ensure that the model is still working as expected on, e.g., test datasets or synthetic datasets. An example could be to add the following to our previous notebook:

Code snippet for evaluating our model on a test dataset, failing if accuracy too low. Code by author

Having added these checks, or "tests," in the notebook, we can create a "job" in Databricks, pointing to the notebook. To re-train our model, we simply tell the job to create a "run" with the notebook (which can be done in the UI or through a REST call).



ID: 2 Creator: [redacted] Schedule: None Task: Notebook at /User[redacted]MNIST_keras

Active Runs

Run	Start Time	Launched	Duration	Spark	Status
Run Now / Run Now With Different Parameters					

0 - 0 20 / Page

Completed Runs (past 60 days)

Latest Successful Run (Refreshes Automatically)

Run	Start Time	Launched	Duration	Spark	Status
View Details	May 5 2021, 22:06 PM CEST	Manually	7m 7s	Spark UI / Logs / Metrics	Succeeded - Delete
View Details	May 5 2021, 22:04 PM CEST	Manually	21s	Spark UI / Logs / Metrics	Failed - Delete
View Details	May 5 2021, 22:02 PM CEST	Manually	12s	Spark UI / Logs / Metrics	Failed - Delete
View Details	May 5 2021, 21:26 PM CEST	Manually	28m 37s	Spark UI / Logs / Metrics	Succeeded - Delete
View Details	May 5 2021, 21:25 PM CEST	Manually	11s	Spark UI / Logs / Metrics	Failed - Delete
View Details	May 5 2021, 21:03 PM CEST	Manually	18s	Spark UI / Logs / Metrics	Failed - Delete

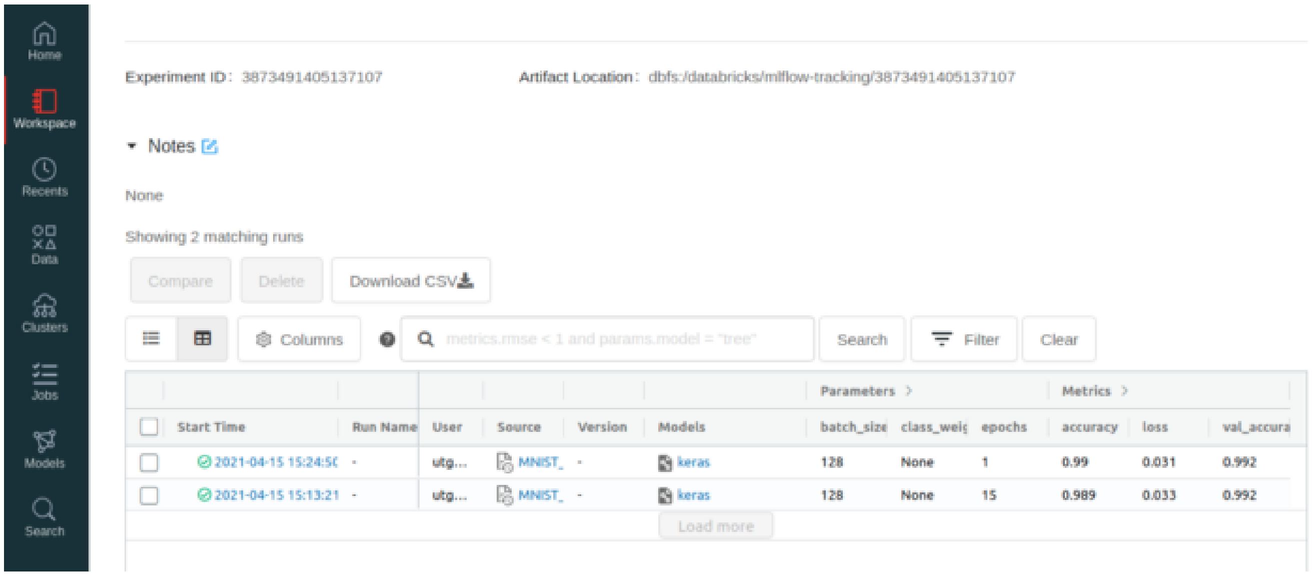
1 - 6 20 / Page

Screenshot showing "runs" for our MNIST job—screenshot by author.

This is exactly what we want; if, for instance, we automatically send a re-train REST request to Databricks when the performance of a deployed model is starting to decrease too much, but for some reason, new corrupt data has been introduced into the dataset since last training thereby breaking the model, we do not want that model to proceed into production.

3. Versioning

Once training of our model is finished, we can view the mlflow 'experiment' attached to this notebook — here we get an overview of the models trained as part of this notebook and metrics and parameters for each model (e.g., loss over time, etc.). We can click the 'source' buttons (see below screenshot), which will show us exactly what code was used to train the model; this means we will be able to audit any model in production to see exactly who trained it and how.



Experiment ID: 3873491405137107   Artifact Location: dbfs:/databricks/mlflow-tracking/3873491405137107

None

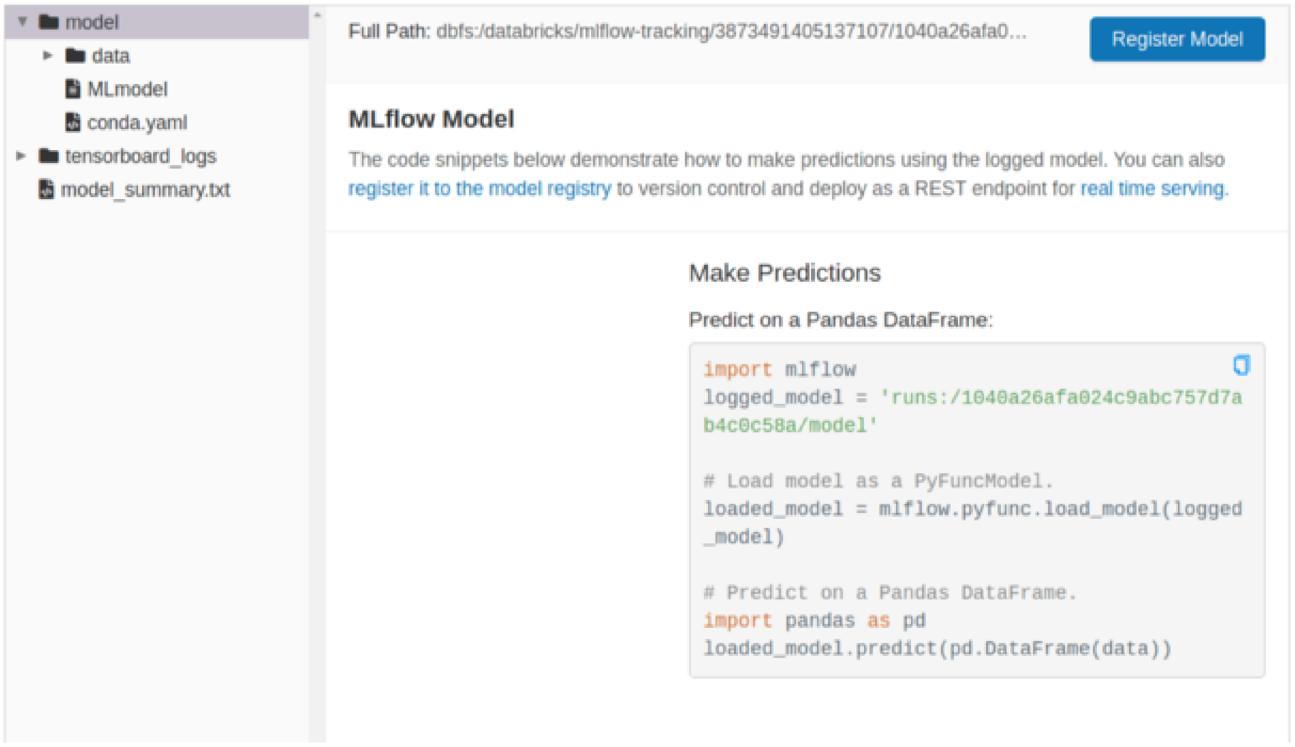
Showing 2 matching runs

metrics.rmse < 1 and params.model = "tree"

						Parameters			Metrics		
Start Time	Run Name	User	Source	Version	Models	batch_size	class_weig	epochs	accuracy	loss	val_accura
2021-04-15 15:24:5(	-	utg...	MNIST_	-	keras	128	None	1	0.99	0.031	0.992
2021-04-15 15:13:21	-	utg...	MNIST_	-	keras	128	None	15	0.989	0.033	0.992

Screenshot showing mlflow experiment with our two trained MNIST models. Screenshot by author.

In the above screenshot, there is also a “Models” column, where for each run, mlflow creates a “mlflow model” object, which encapsulates the model at the end of that run. Clicking one of these, we get to a page with details about the model, including information about loss and accuracy during training, training parameters, and finally, a section with artifacts, where we can see the encapsulated “mlflow model,” which contains information about the `conda.yaml` environment required by the model, the trained model weights, etc. By clicking the “Register Model” button, we can enter any of these models into the mlflow model registry (this could, of course, also be done in the code)



Full Path: dbfs:/databricks/mlflow-tracking/3873491405137107/1040a26afa0...

MLflow Model

The code snippets below demonstrate how to make predictions using the logged model. You can also register it to the model registry to version control and deploy as a REST endpoint for real time serving.

Make Predictions

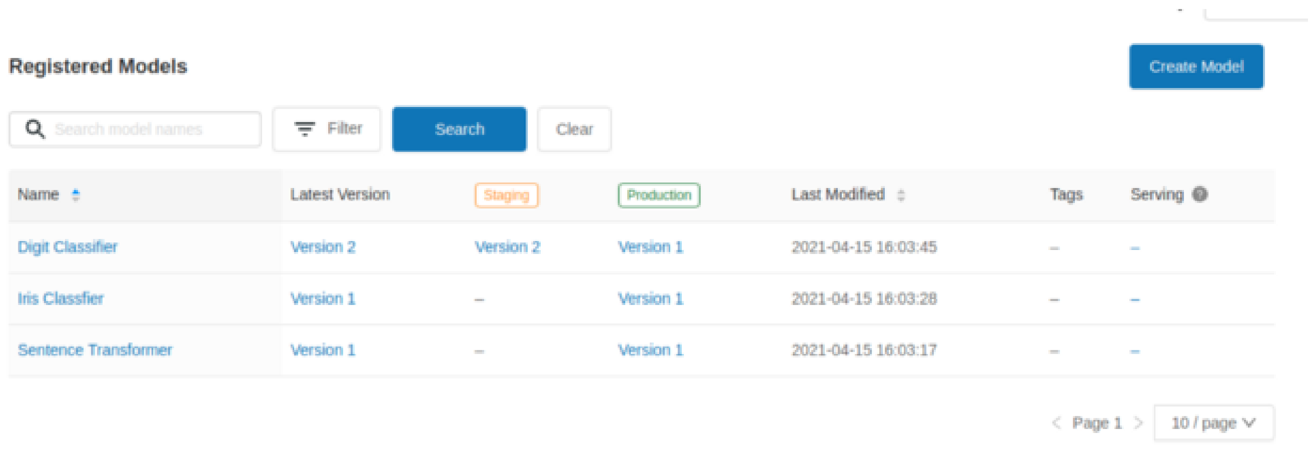
Predict on a Pandas DataFrame:

```
import mlflow
logged_model = 'runs:/1040a26afa024c9abc757d7a
b4c0c58a/model'

# Load model as a PyFuncModel.
loaded_model = mlflow.pyfunc.load_model(logged
_model)

# Predict on a Pandas DataFrame.
import pandas as pd
loaded_model.predict(pd.DataFrame(data))
```

Screenshot of mlflow artifacts for a given run specifically focused on the trained mlflow model.

4. Model Registry

Once we've logged one or more models into our model registry, we can click the "Models" tab on the left in Databricks in order to get an overview of our model registry with all the models and their current stages:



Registered Models

Name	Latest Version	Staging	Production	Last Modified	Tags	Serving
Digit Classifier	Version 2	Version 2	Version 1	2021-04-15 16:03:45	–	–
Iris Classfier	Version 1	–	Version 1	2021-04-15 16:03:28	–	–
Sentence Transformer	Version 1	–	Version 1	2021-04-15 16:03:17	–	–

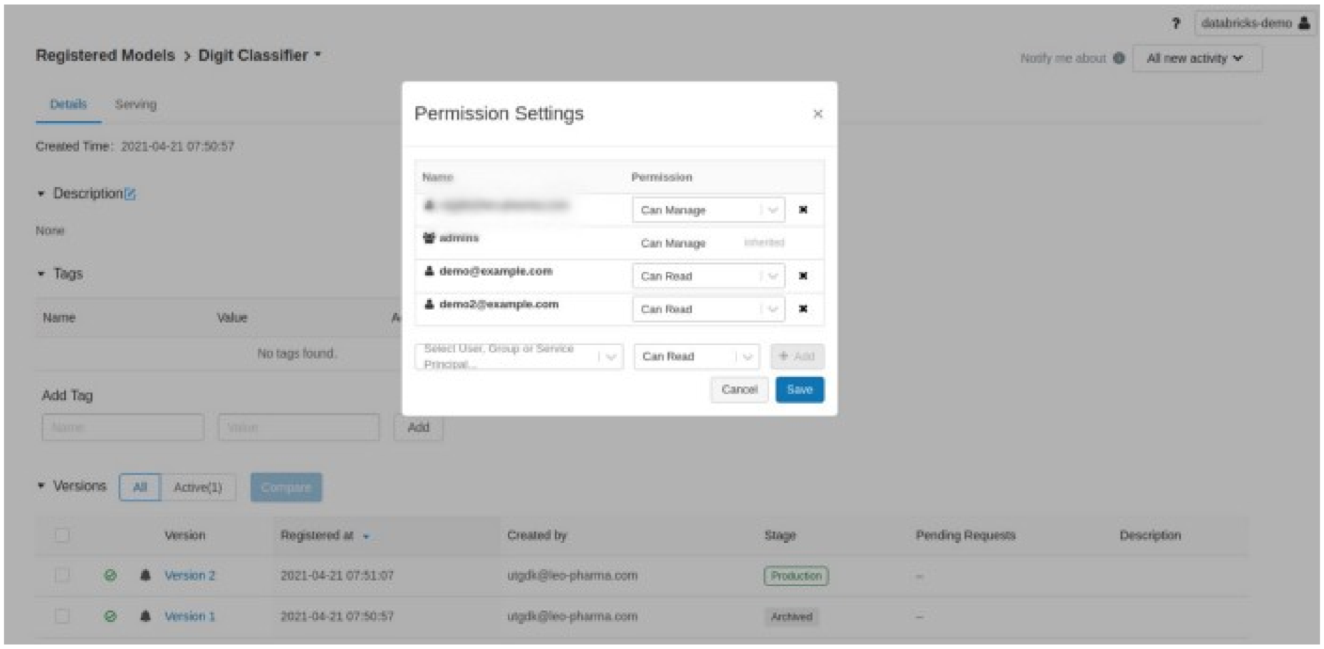
The model registry from Databricks shows each model, its latest version, deployment stages, etc.

This gives us a lot of transparency in terms of what is currently being deployed, which versions were previously deployed, with links to all the model objects, allowing us to inspect what code and data they were trained on, etc.

5. Model Governance

For each trained model registered into our model registry, we can control the permission settings, meaning that we can control who can see which models in the model registry, as well as who can authorize transitions from, e.g., Staging to Production; this is exactly what we want, in that we wish to have a control gate for which models go into production; e.g., maybe the manager of the data scientist who created a given model needs to validate and approve a given model before it transitions into deployment, checking things such as:

- Ensure that the model was tested against a known test set or through cross-validation to confirm that it works.
- Ensure that the model was tested against synthetic datasets with a known "signal." If it fails this type of test, maybe an error was introduced.



Created Time: 2021-04-21 07:50:57

Description: None

Name	Value
No tags found.	

Permission Settings

Name	Permission
[redacted]	Can Manage
admins	Can Manage (Inherited)
demo@example.com	Can Read
demo2@example.com	Can Read

Select User, Group or Service Principal... | Can Read | Add

Cancel | Save

Version	Registered at	Created by	Stage	Pending Requests	Description
Version 2	2021-04-21 07:51:07	utgdk@leo-pharma.com	Production	–	
Version 1	2021-04-21 07:50:57	utgdk@leo-pharma.com	Archived	–	

Screenshot shows permission settings of a given model in the model registry, demonstrating how we can give different permissions to different users. Screenshot by author.

6. Deployment

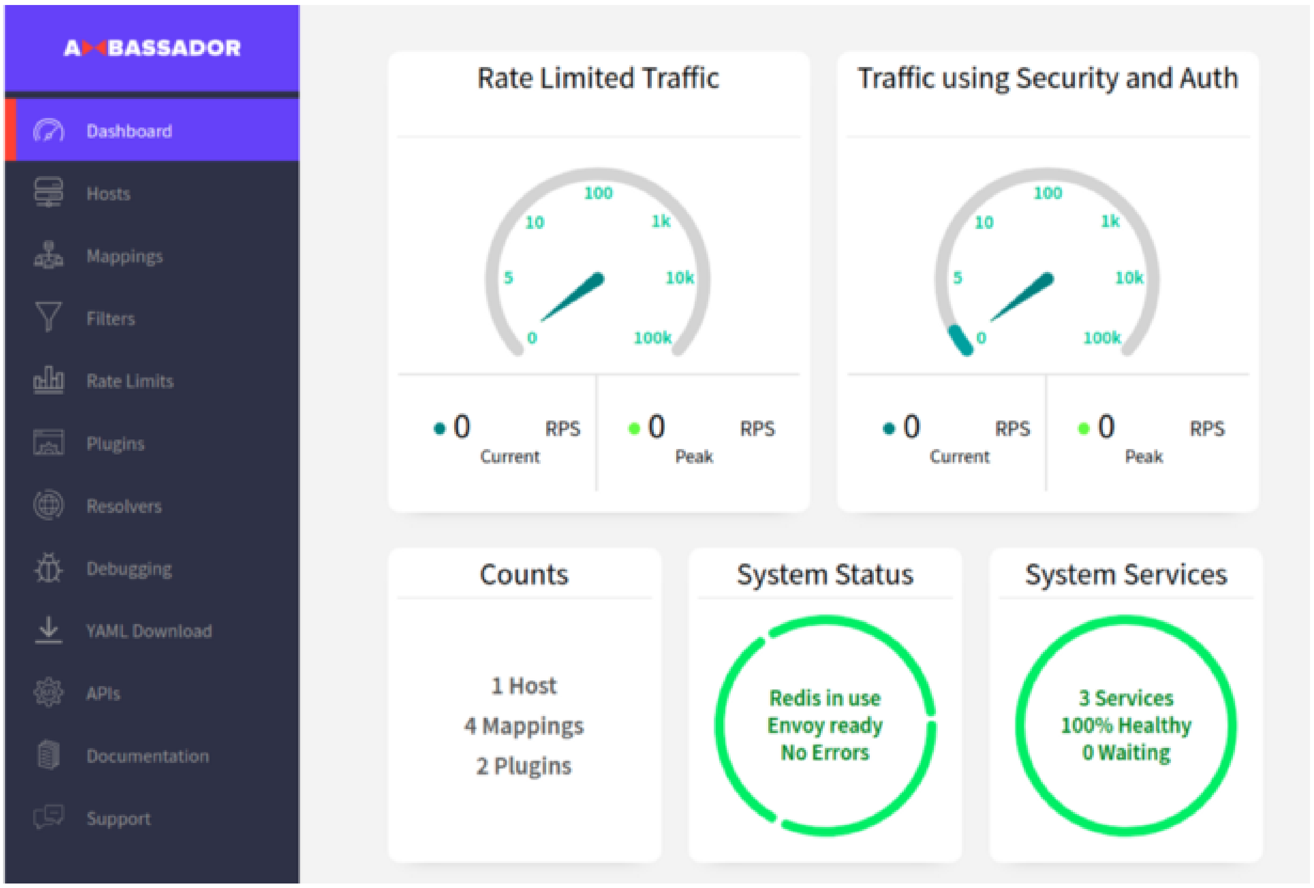
Once a model in the model registry has been transitioned into the "Production" stage, the next phase is to push that model to a deployment server so that the consumers of that model can call it using, e.g., REST API. In this post, Seldon-core will handle these deployments. Briefly, Seldon-core wraps and deploys ML models onto a Kubernetes cluster, Kubernetes being a common open-source system for automating deployment, scaling, and management of containerized applications. To test all this out, first, we need a Kubernetes cluster, which we can do by adding a bit of configuration to our terraform script so as to set up an azure-managed Kubernetes cluster:

Extension of previous terraform script to also create a Kubernetes cluster on Azure. Code by author.

Adding the above to our previous terraform script and re-running `terraform apply` should update our infrastructure with a very bare-bones Kubernetes cluster. Now that we have a cluster up and running, we need to install Seldon-core + associated packages, which can be achieved with the following commands (taken from official docs):

Bash commands for install Seldon-core on our newly created Kubernetes cluster

And that's it 👌 .. now we have a basic Kubernetes cluster running in the cloud with Seldon-core installed. As part of the setup, the above commands also install Ambassador, which controls requests into our Kubernetes cluster, and when you run the `edgectl install` above, it should have opened your browser to your newly set up domain for ambassador, which gives us a nice interface for inspecting what we're exposing from our Kubernetes cluster, how it's running, etc.



Screenshot of Ambassador interface for our new cluster. The screenshot was taken by the author.

Now that we've set up a fundamental version of Seldon-core, we can start exploring how to deploy our mlflow model. Sadly we will have to do a bit of plumping to make this happen — i.e., although there is support for mlflow models in Seldon-core, it is not in the form of a big nice button that would automagically deploy our models. Keep in mind, though, that a lot of the steps involved in deploying a model could be automated — I discuss this more thoroughly in my next blog post, which can be found here:

Making MLOps easy for End-Users

A tutorial on simplifying MLOps using open source tools

towardsdatascience.com

Okay. We want to take our mlflow model from the Databricks model registry and deploy it into Seldon-core. Seldon-core does come with an out-of-the-box mlflow wrapper for deploying mlflow models, but for me, that kept crashing with the model in question. Instead, I went with a bit more custom method, where we need to create a Seldon-compatible docker image, which we can deploy onto Seldon-core. I actually find this preferable since it will allow us to explore a few more edge applications of Seldon in the later sections relating to drift and outlier detection. As the first thing, we'll need to download our mlflow model from the Databricks mlflow server:

Get an Azure AD token, and use it to authenticate the Databricks CLI against your workspace.

Script used for downloading model from Databricks hosted mlflow to a local folder

With the above script, we can readily download a given model from mlflow, e.g., in my case, I ran `python download_mlflow_model.py --model-name “Digit Classifier" --model_stage Production`. This will download the trained mlflow model into a `model/` folder. From here, we just need to write a small re-useable python wrapper, which lets Seldon know how to load this mlflow model and perform predictions with it by defining `load()` and `predict()` methods. This `MyMlflowModel` python wrapper is really the core piece of interface we have between mlflow and Seldon, and we will be returning to it several times throughout this post.

Python interface for our mlflow to be used by Seldon-core. Model is loaded from our downloader /model folder.

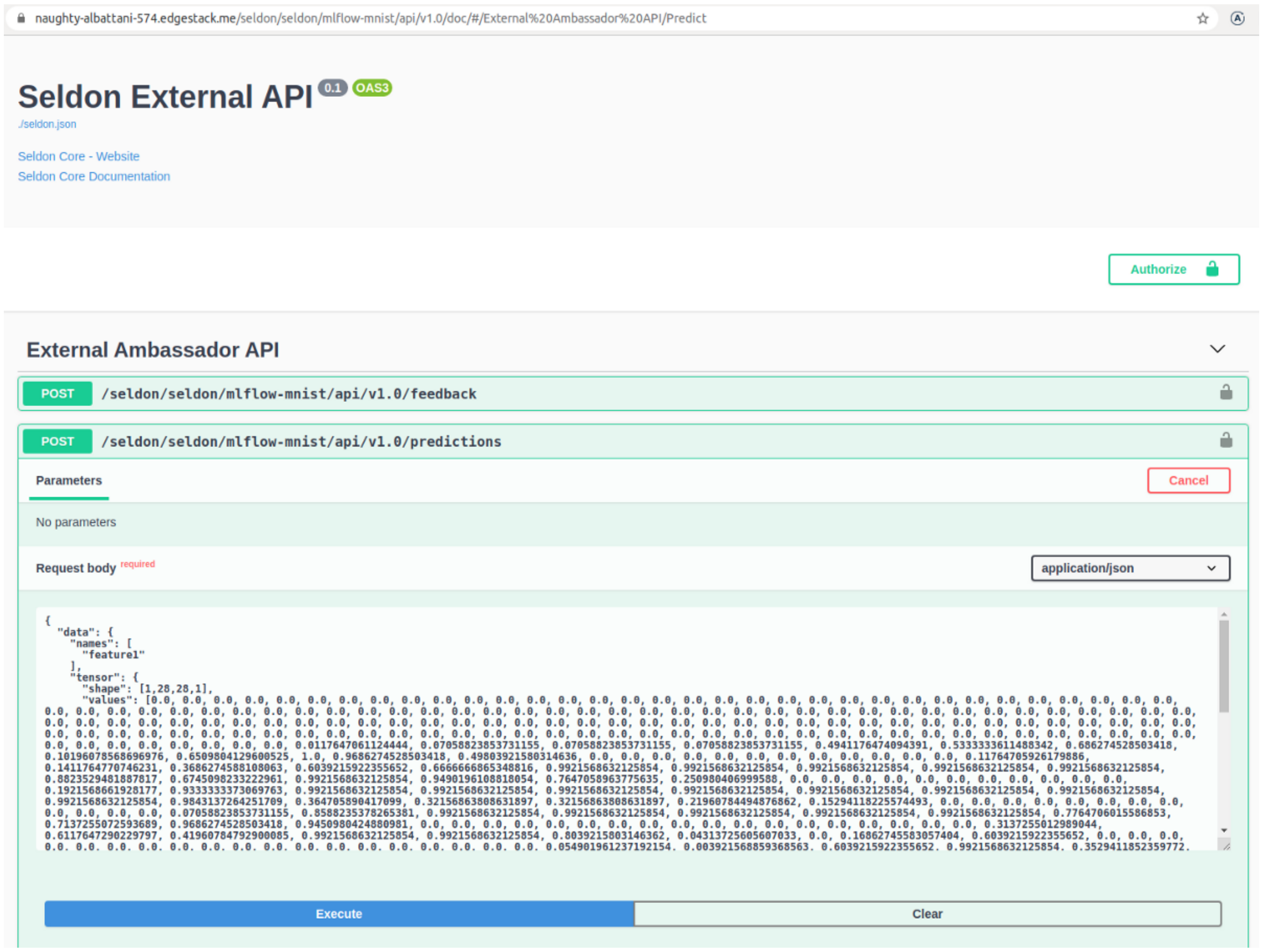
With the downloaded model and the python wrapper in hand, we are ready to package up an appropriate docker image with our mlflow model. This can be easily accomplished using the command-line tool `s2i`, which lets us inject code into a docker image maintained by Seldon:

Bundling up our mlflow model in a docker image, which is pushed to dockerhub

Reading through the Seldon-core documentation, there is a multitude of ways to test this docker image locally before deploying, e.g., by launching the docker image and sending requests towards it or using various command-line tools provided by Seldon-core. Once confirmed that it works locally, we have to push our image to some container repository — in this case, I just use public dockerhub, but we have to remember that models can be highly confidential as well, so in a real-world setup, security of the docker image repository has to be considered as well. Finally, we can push our trained mlflow model to our Kubernetes cluster using a super basic Kubernetes `mlflow.yaml` file, which is applied with `kubectl apply -f mlflow.yaml`:

Configuration file for our mlflow model deployment on Kubernetes

Once the `kubectl apply` command succeeds, you should be able to go to your ambassador endpoint (e.g., https://YOUR_URL/seldon/mlflowmodels-namespace/mlflow-mnist/api/v1.0/doc/) to get the documentation for the REST API of your newly deployed model. This page will also enable you to quickly start testing out the API by sending requests to it directly from the browser:


naughty-albattani-574.edgestack.me/seldon/seldon/mlflow-mnist/api/v1.0/doc/#/External%20Ambassador%20API/Predict

Seldon External API 0.1 OAS3

./seldon.json

Seldon Core - Website

Seldon Core Documentation

External Ambassador API

POST /seldon/seldon/mlflow-mnist/api/v1.0/feedback

POST /seldon/seldon/mlflow-mnist/api/v1.0/predictions

Parameters

No parameters

Request body required

application/json

```
{
  "data": {
    "names": [
      "feature1"
    ],
    "tensor": {
      "shape": [1,28,28,1],
      "values": [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0117647061124444, 0.07058823853731155, 0.07058823853731155, 0.07058823853731155, 0.4941176474094391, 0.5333333611488342, 0.686274528503418,
0.10196078568696976, 0.6509804129600525, 1.0, 0.9686274528503418, 0.49803921580314636, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.11764705926179886,
0.1411764770746231, 0.3686274588108063, 0.6039215922355652, 0.6666666865348816, 0.9921568632125854, 0.9921568632125854, 0.9921568632125854, 0.9921568632125854, 0.9921568632125854,
0.8823529481887817, 0.6745098233222961, 0.9921568632125854, 0.9490196108818054, 0.7647058963775635, 0.250980406999588, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.1921568661928177, 0.9333333373069763, 0.9921568632125854, 0.9921568632125854, 0.9921568632125854, 0.9921568632125854, 0.9921568632125854, 0.9921568632125854, 0.9921568632125854,
0.9921568632125854, 0.9843137264251709, 0.364705890417099, 0.32156863808631897, 0.32156863808631897, 0.21960784494876862, 0.15294118225574493, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.07058823853731155, 0.8588235378265381, 0.9921568632125854, 0.9921568632125854, 0.9921568632125854, 0.9921568632125854, 0.9921568632125854, 0.7764706015586853,
0.7137255072593689, 0.9686274528503418, 0.9450980424880981, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.3137255012989044,
0.6117647290229797, 0.41960784792900085, 0.9921568632125854, 0.9921568632125854, 0.8039215803146362, 0.04313725605607033, 0.0, 0.16862745583057404, 0.6039215922355652, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.054901961237192154, 0.003921568859368563, 0.6039215922355652, 0.9921568632125854, 0.3529411852359772,
```

Alternatively, we can use python to hit our API with the entire test dataset:

Python script for sending the entire test MNIST dataset to our deployed model endpoint.

7. Monitoring

Our deployed docker image automatically sets up a lot of monitoring boilerplate that expose a multitude of metrics, which can be scraped by Prometheus (a common monitoring solution). Seldon-core comes with tooling for quickly setting all this up, enabling us to visualize these metrics in Grafana. Essentially, all we have to do is to install `seldon-core-analytics` on our Kubernetes cluster, which we can do by running the following commands:

Bash commands for installing Seldon-core-analytics and port-forwarding Prometheus and Grafana to localhost

With these commands, we have installed Prometheus and Grafana on our cluster and port-forwarded both to our localhost so we can go to `localhost:3001` to view the Prometheus interface, which lets us send queries for all logged metrics, or we can go to `localhost:3000` to see a pre-populated Grafana dashboard giving us an overview of request rates, latencies, etc., for all currently deployed models. Essentially, setting up monitoring is super easy with Seldon-core, and Grafana makes it convenient to customize the dashboards based on the needs and wishes of the user.

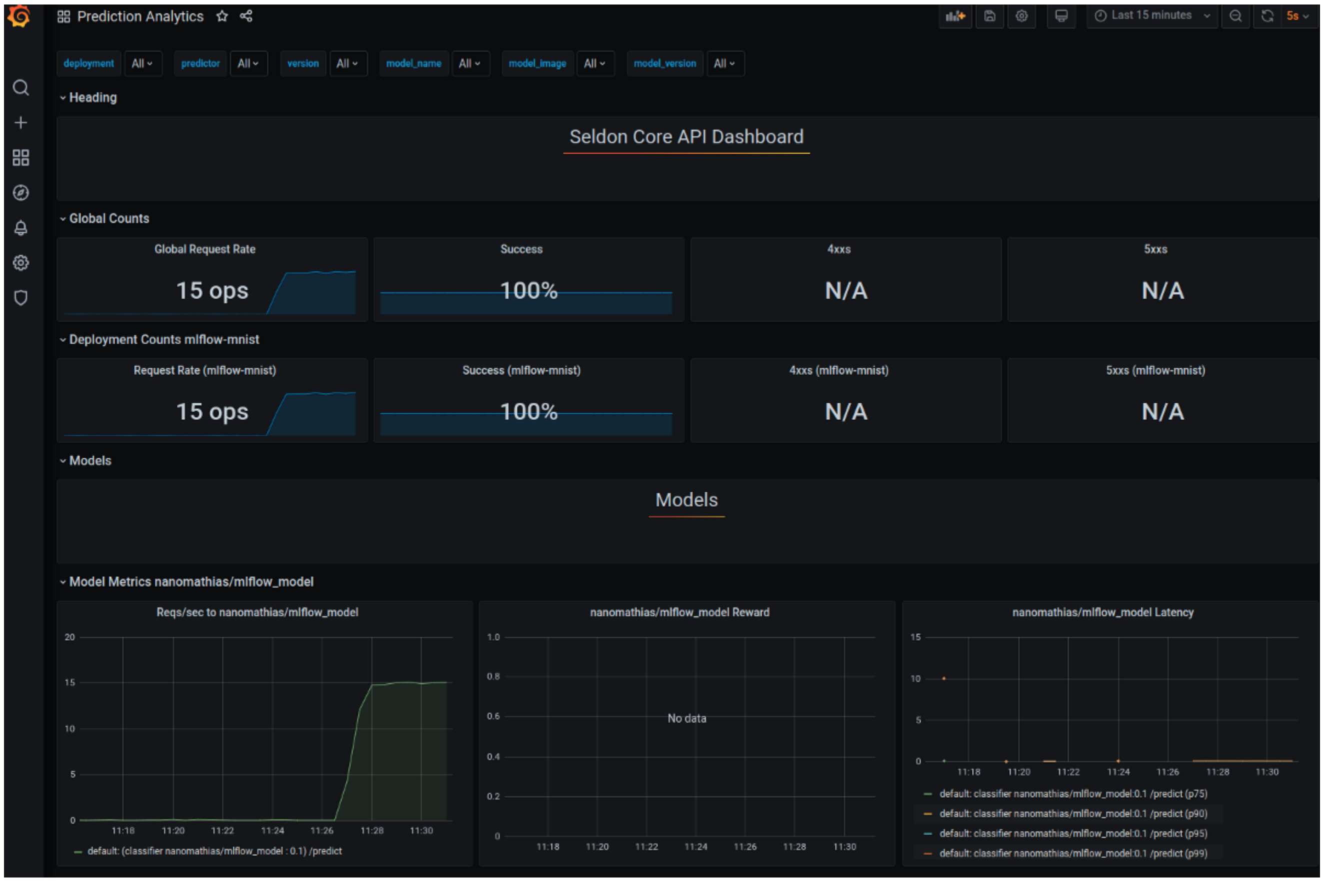
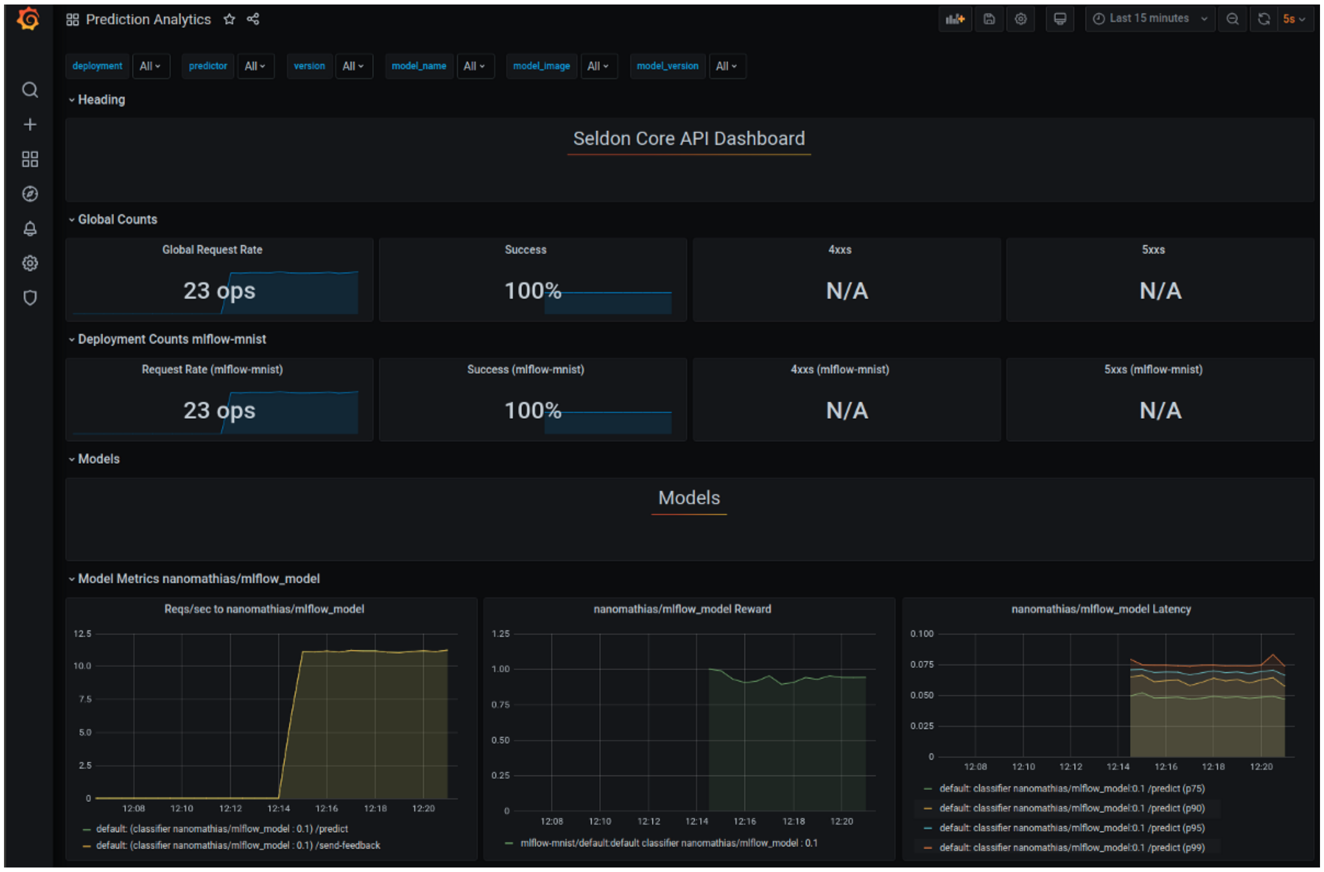

Prediction Analytics
Last 15 minutes
5s
deployment All
predictor All
version All
model_name All
model_image All
model_version All
Heading
Seldon Core API Dashboard
Global Counts
Global Request Rate
15 ops
Success
100%
4xxs
N/A
5xxs
N/A
Deployment Counts mlflow-mnist
Request Rate (mlflow-mnist)
15 ops
Success (mlflow-mnist)
100%
4xxs (mlflow-mnist)
N/A
5xxs (mlflow-mnist)
N/A
Models
Models
Model Metrics nanomathias/mlflow_model
Reqs/sec to nanomathias/mlflow_model
default: (classifier nanomathias/mlflow_model : 0.1) /predict
nanomathias/mlflow_model Reward
No data
nanomathias/mlflow_model Latency
default: classifier nanomathias/mlflow_model:0.1 /predict (p75)
default: classifier nanomathias/mlflow_model:0.1 /predict (p90)
default: classifier nanomathias/mlflow_model:0.1 /predict (p95)
default: classifier nanomathias/mlflow_model:0.1 /predict (p99)

Image by author, showing Grafana dashboard with our deployed mlflow model.

8. Feedback

As you may have noticed from the API documentation exposed by Seldon, it automatically exposes a `/feedback` endpoint as well as `/predict` endpoint. In addition, the Grafana dashboard also, by default, provides a graph showing the “reward” signal from our model. “Feedback” and “rewards” are what enable us to evaluate our model once it is deployed, as we progressively start getting additional data with known labels. Getting new data where we know the true labels, we can send a given reward to the `/feedback` endpoint, registering whether or not a model has been performing well (high rewards) or poorly (low rewards). As an example, we can test the endpoint out by looping through our test data and sending rewards back to the model based on how well it did with its predictions:

Loop over test set, get predictions from the model, feedback rewards to the model based on performance.



Prediction Analytics
Last 15 minutes
5s
deployment All
predictor All
version All
model_name All
model_image All
model_version All
Heading
Seldon Core API Dashboard
Global Counts
Global Request Rate
23 ops
Success
100%
4xxs
N/A
5xxs
N/A
Deployment Counts mlflow-mnist
Request Rate (mlflow-mnist)
23 ops
Success (mlflow-mnist)
100%
4xxs (mlflow-mnist)
N/A
5xxs (mlflow-mnist)
N/A
Models
Models
Model Metrics nanomathias/mlflow_model
Reqs/sec to nanomathias/mlflow_model
default: (classifier nanomathias/mlflow_model : 0.1) /predict
default: (classifier nanomathias/mlflow_model : 0.1) /send-feedback
nanomathias/mlflow_model Reward
mlflow-mnist/default:default classifier nanomathias/mlflow_model : 0.1
nanomathias/mlflow_model Latency
default: classifier nanomathias/mlflow_model:0.1 /predict (p75)
default: classifier nanomathias/mlflow_model:0.1 /predict (p90)
default: classifier nanomathias/mlflow_model:0.1 /predict (p95)
default: classifier nanomathias/mlflow_model:0.1 /predict (p99)

Screenshot of Grafana dashboards showing our model performing quite well as determined by reward signal.

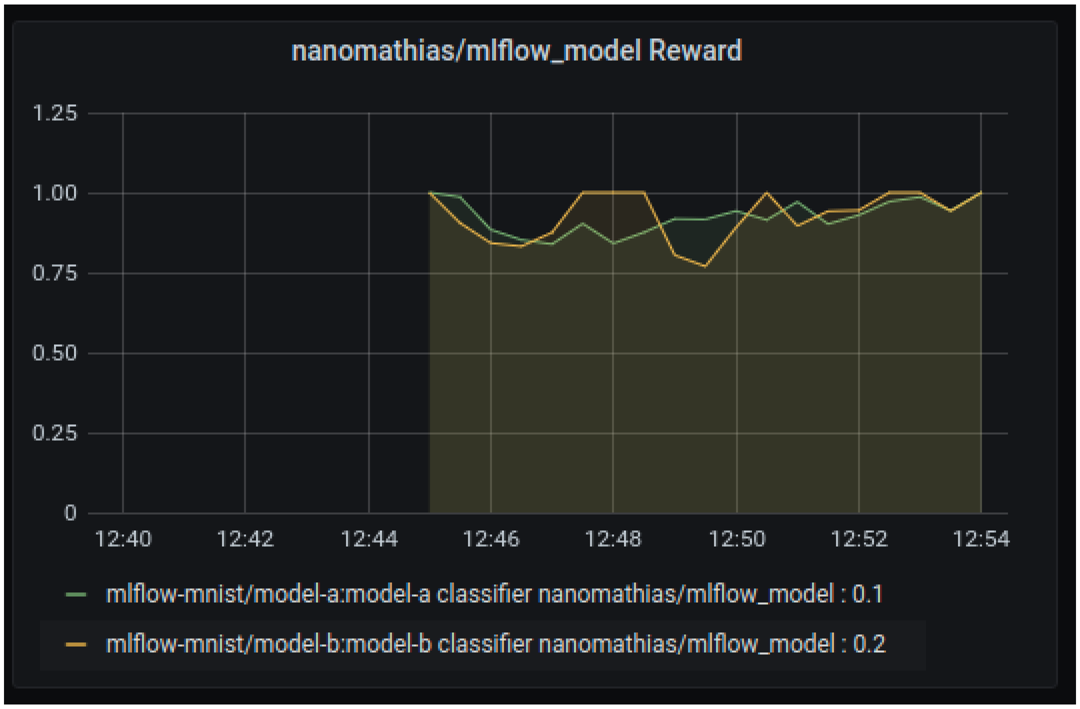
Instead of sending the reward directly, it is worth noting that we could also overwrite the `send_feedback()` method on our python model wrapper, enabling us to send the data sample and the true label to the `/feedback` endpoint and then calculating the "reward" (e.g., in terms of accuracy) on the backend — essentially there is a lot of flexibility in terms of how to track live model performance.

Note that we could quite easily implement a threshold in our Python wrapper, which would automatically send a re-training request to Databricks when the average reward reduces below the given threshold. 🚀

9. A/B Testing

When introducing a new version of your model, you may wish to validate that it works better than your previous model. As such, we may wish to run AB testing live in production to see which model is performing better. Seldon-core makes doing this quite easy in that we essentially only need to add `traffic` keywords to our previous `yaml` file and add another predictor. As an example, let us try adding another MNIST model which accidentally has only been trained for 1 epoch. Before fully replacing our old model, we will direct 80% of the traffic to the old model and 20% to the new one:

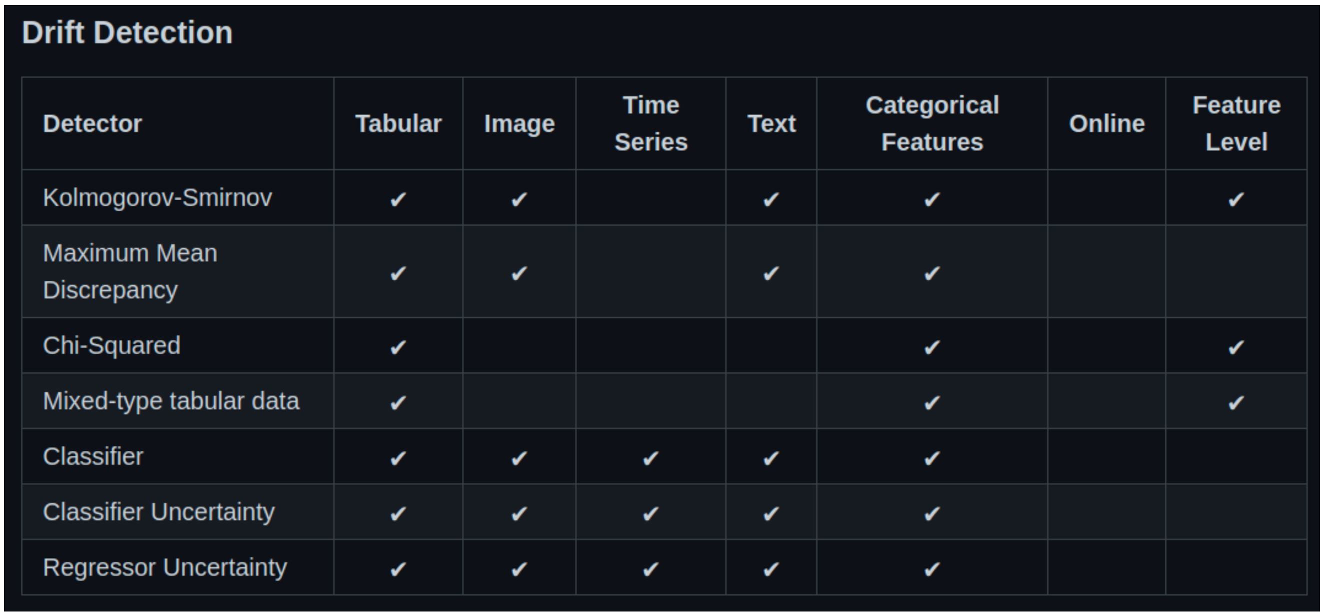
With this, 80% of the traffic is directed towards the old model (version 0.1) and 20% towards the new model (version 0.2), and both report to our Grafana dashboard, where we can compare their performance as they receive feedback signals.



In this case, both models actually perform about the same, so only training the MNIST classifier for 1 epoch did not result in that much worse of a model but also didn't significantly improve results. Errors in model implementation or training data could, however, in some cases, pass through all our previous checks, so it can be a good idea to perform these AB tests for mission-critical deployments, and Seldon-core makes it very easy to do.

10. Drift Detection

One of the reasons for choosing Seldon-core in this post is that they maintain and integrate the library Alibi-Detect, which provides pre-implemented algorithms for drift detection, adversarial attack detection, and outlier detection — having these pre-implemented can be a big time-saver as there is no one-algorithm-fits-all, and it would be a pain to implement them all ourselves.



Drift Detection

Detector	Tabular	Image	Time Series	Text	Categorical Features	Online	Feature Level
Kolmogorov-Smirnov	✔	✔		✔	✔		✔
Maximum Mean Discrepancy	✔	✔		✔	✔		
Chi-Squared	✔				✔		✔
Mixed-type tabular data	✔				✔		✔
Classifier	✔	✔	✔	✔	✔		
Classifier Uncertainty	✔	✔	✔	✔	✔		
Regressor Uncertainty	✔	✔	✔	✔	✔		

Drift detection algorithms support in Alibi-Detect (4th of May, 2021)

Let us say we wish to apply one of the drift detection algorithms to our MNIST classifier. To do this in the manner recommended by Seldon-core, we have to introduce yet another tool, namely "KNative," which allows us to pick up payload information from the Seldon-core API and process it asynchronously, see figure below:

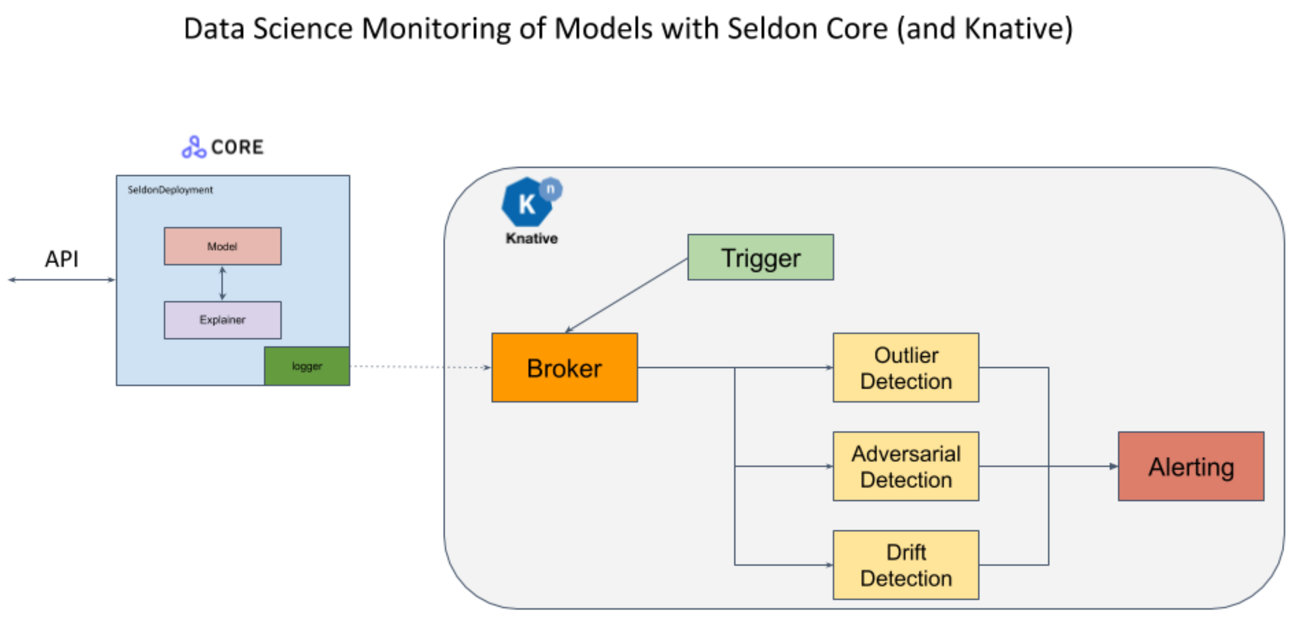


Image from Seldon-Core documentation

Setting this up is a little cumbersome, as it would involve installing Knative and setting it up to work with Ambassador properly, training an Alibi-Detect model and pushing it to blob storage, and finally setting up the required Knative resources for our model. In the end, though, after all the setup, deploying a drift detection algorithm is no more complicated than applying another `yaml` file to Kubernetes, pointing to the pre-trained drift-detection algorithm.

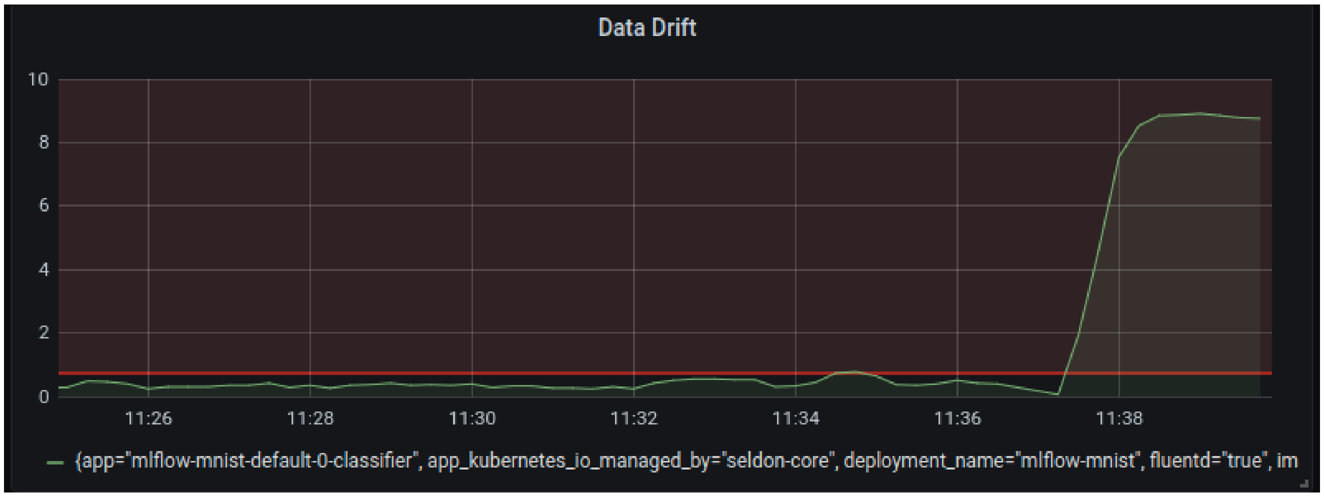
For the purposes of keeping this post somewhat succinct, I've opted not to go through this Knative process but instead mention that there are examples in the Seldon-core docs on how to proceed. As an alternative, it is worth pointing out that we do have the option of implementing custom metrics directly on our python model wrapper class, which means we could include pre-trained drift-detection models (based on Alibi-Detect) in the model itself, thereby letting it evaluate synchronously with each request, and reporting the drift live in Grafana. To do this, we can go back to our Databricks notebook and train a Kolmogorov-Smirnov drift detection algorithm (checks for differences in data distribution):

Training a KS drift detector for MNIST dataset and adding to mlflow model artifacts

With this, we have trained & added our drift-detection algorithm to our mlflow model, which we can then re-package into a Seldon-core docker image and deploy to our Kubernetes cluster. Before that, though, we'll update our python wrapper class as well to perform the drift detection on each prediction:

Python wrapper for our mlflow model with added drift detector.

With this model deployed, the drift is now scraped by Prometheus, and we can easily create a graph in Grafana showing the drift live. We can even add alert thresholds, such that if data drifts too far away from training data, relevant users are notified. Here is an example where initially I fed the model lots of normal MNIST images, and all of a sudden, a lot of distorted samples, which is readily picked up by the detector:

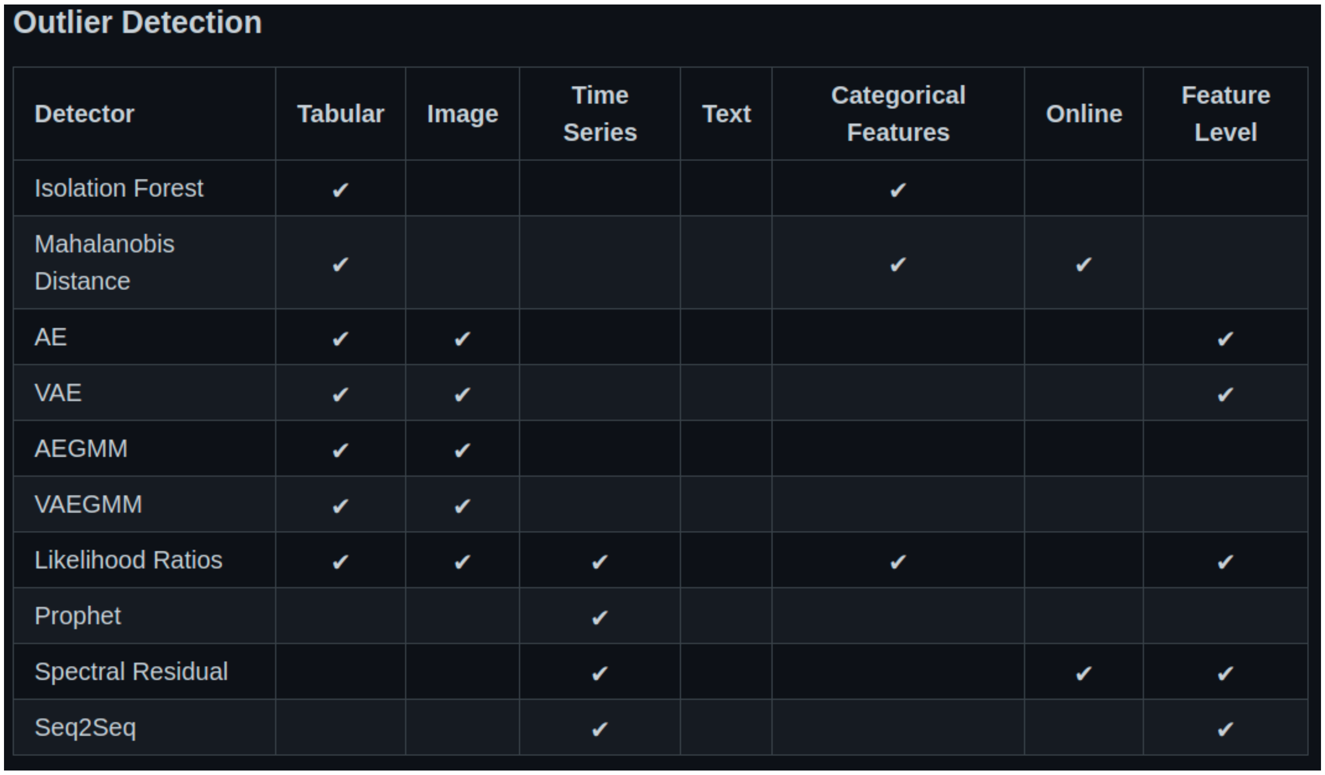


Screenshot from author taken from Grafana dashboard

This specific type of drift is known as 'feature drift' or data drift, i.e., a change in the incoming data features. Any other type of drift detection could, however, also readily be implemented directly into the python wrapper.

11. Outlier Detection

Similar to drift detection, Alibi-Detect implements various algorithms for identifying that an incoming sample is an outlier. The recommended implementation in Seldon-core is similar to that of drift detection in that it involves setting up Knative and processing whether a given sample is an outlier asynchronously. Again I will skip this implementation for the sake of brevity and instead link to the documentation example.



Outlier Detection

Detector	Tabular	Image	Time Series	Text	Categorical Features	Online	Feature Level
Isolation Forest	✔				✔		
Mahalanobis Distance	✔				✔	✔	
AE	✔	✔					✔
VAE	✔	✔					✔
AEGMM	✔	✔					
VAEGMM	✔	✔					
Likelihood Ratios	✔	✔	✔		✔		✔
Prophet			✔				
Spectral Residual			✔			✔	✔
Seq2Seq			✔				✔

Outlier detection algorithms support in Alibi-Detect (4th of May, 2021)

Instead of setting up Knative, it is worth mentioning that similarly to how we can implement custom metrics on our python model wrapper, we can also implement custom tags to be returned on prediction — as such, one could bundle the outlier detector together with the model as well, and synchronously process each sample going into the `/predict` endpoint in the outlier detector. One idea could be that if a sample is an outlier, to return a tag stating that the sample is expected to be an outlier along with the prediction. If we were to try doing this, we first go back to our Databricks notebook, where we can train the outlier detector and bundle it with our mlflow model:

Python script for training an outlier predictor for MNIST dataset and bundling it as mlflow model artifact.

Then we just need to add the outlier detector to our current python wrapper:

Python wrapper for our mlflow model with added drift and outlier detector

With this, we can bundle up our new mlflow model in a new Seldon-core docker image and deploy it to Kubernetes. When sending a REST request to the prediction endpoint with a sample that is clearly an outlier (e.g., a completely white image), we now get the following response:

Sample response from deploy mlflow model with drift and outlier detector, when we send a white image.

Now the end-user will immediately be able to see that this prediction is fairly unsure since the input data is tagged as an outlier as compared to the training data.

12. Adversarial Attack Detection

Machine learning algorithms can be brittle in the sense that by very slightly altering the input we give to a model, we can sometimes radically change its output — I'm sure you've all seen examples online where an imperceptible change to a given image can change its classification from, e.g., panda to a gibbon. In the context of MLOps, we need to consider that a bad actor may exploit this fact about some of our deployed machine learning models in an attempt to determine how predictions by the algorithm can be manipulated optimally. Depending on the use case, this can be more or less critical. Again the Alibi-Detect package comes to the rescue as it implements an algorithm for detecting "adversarial" (manipulated) samples using an adversarial autoencoder, enabling us to get alerts when an unusual amount of adversarial data is encountered by our deployed models.

As with outlier and drift detection, we could include the adversarial detector directly within the model code; an added benefit of this is actually that we can use the adversarial detector not only to detect attacks of our ML model but also correct the adversarial examples so as to make attacks much more difficult, see paper for details. Implementation of this would quite trivially follow the steps for the outlier and drift detectors, and therefore I opt for not including it here.

13. Interpretability

In addition to Alibi-Detect, Seldon also maintains and integrates a library Alibi, which implements various algorithms for model interpretation (e.g., SHAP, integrated gradients, anchors, and more). These are implemented in such as manner that they provide an additional `/explain` endpoint in our REST API so that any user who is interested can easily query for an explanation for a given prediction. This is awesome because it means that when an end-user is in doubt about why a given prediction is returned, he does not need to perform extensive analysis manually but can just call the `/explain` endpoint.

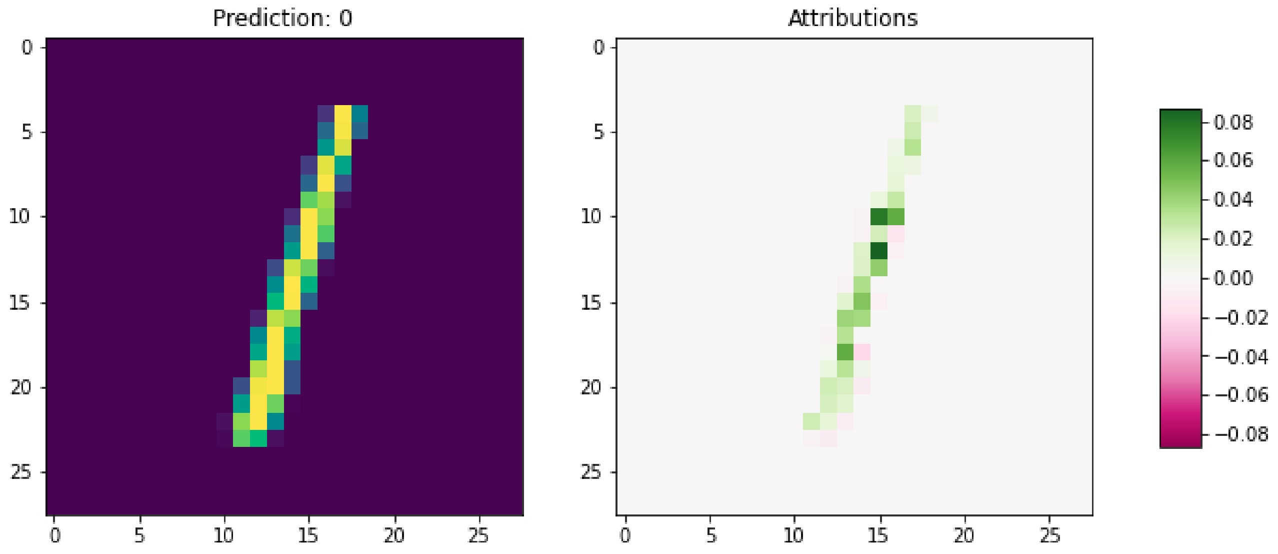
Let us try to add an "integrated gradients" explainer to our MNIST model — different explainers have different requirements, but in this case, the explainer needs access to our Keras model directly — as such, first, we will save the model in our Databricks notebook, and upload to the Azure blob we created with Terraform initially:

Script for saving our Keras model into Azure blob storage in the cloud.

Note: the `AZURE_CONNECT_STRING` we get from the storage account in Azure. Now that we have uploaded our explainer to the blob storage, we can extend the `yaml` -file of our Seldon deployments to include the `explainer` section:

With this, we can send requests to an endpoint with the format: https://[url]/seldon/[space]/[name]-explainer/default/api/v1.0/explain, and we will receive a response that contains the contribution of each pixel towards the classification. Visualizing these for a given sample, we would get a plot like follows, showing which pixels caused the given positive classification of the given label.

Calling the /explain endpoint with a sample image from the test set and plotting returned explanation



The resulting plot showing the sample sent to explain endpoint (left) and the pixel attributions (right)

This works quite well, and we can see that the model focuses on sensible pixels for making its prediction here.

14. Deployment Governance

It can be important that not all users have access to a given deployed model — basically, the principle of least privilege should always hold: allow people to do what they need to do, and nothing more. For instance, we might not want someone from the finance department poking around a model trained on highly confidential data from the R&D department. Luckily, there is plenty of tooling around Kubernetes to solve these issues, e.g., in our case, where we are using a managed Azure cluster with Ambassador; we could, for instance, use Azure AD to authenticate our endpoints; see this link for more details. The details of these security aspects are beyond the scope of this post, but solutions are definitely out there.

An interesting complication in terms of governance is that access permissions on the final endpoints should likely align with the permissions set within our model registry in Databricks/mlflow, and so some additional tooling for tightly syncing these needs to be implemented. The details of such a tool are also outside the scope of this blog post.

15. Data-centricity

The final but perhaps most important point is how we can change the primary focus of MLOps away from the models, their performance, their monitoring, etc., and back towards the data. As anyone who has worked within data science will know, systematic improvement of data quality, consistency, and breadth (covering edge cases) will lead to much better model performance than chasing more complex and fancy models.

How to make an MLOps pipeline data-centric is a little bit of an open question, and the answer depends on the use case, but a few observations are:

- The implementations of feedback monitoring, drift detection, and outlier identifications already tell us something about how well our model is doing on specific new samples, how different they are from what we've already observed, and whether we should be suspicious of the accuracy of an incoming new sample, respectively. These are important metrics to keep in mind prior to re-training our model; e.g., if we identified that a sample is an outlier (e.g., a broken sensor), we should not include it in our re-training and samples where we know the old model performed poorly we should probably investigate a bit more thoroughly before adding to our dataset.
- In some applications where we can choose what data to collect, an additional model suitable for Bayesian optimization could be deployed; in this way, the deployed model can guide the user on which additional data points to collect.
- Similarly to re-training upon feedback, we can easily imagine a scenario where we have received so much new data that a re-training of the model would actually increase performance, even though the performance of the old model has not decayed. As such, not only changes in model performance should determine when we re-train but also changes in available training data.

Conclusions

I am frequently approached by companies wanting to introduce their latest and greatest MLOps framework, and it can be quite difficult to evaluate the pros and cons between each of these based merely on marketing material. As I hope this post demonstrates, implementing a comprehensive MLOps framework is no by no means a trivial task, and there are a lot of considerations and pitfalls that should be taken into account.

The proof of concept solution presented in this post covers quite a lot of ground with the perspective of the 15 principles posed in this post, and beyond that, the systems in question (mlflow + Seldon) support a lot of additional concepts that might also be relevant for a given MLOps solution. The solution of mlflow+seldon is, however, by no means perfect, and I had to jump through a lot of hoops when debugging deployments in this framework to get them working. In an ideal scenario, I would want to see data scientists be able to focus on implementing ML models that conform to some python interface implementing methods such as `predict()` , `feedback()` , `drift()` , `outlier()` , `transform()` Etc., and then be able to push a big green "Deploy" button, initiating an automated pipeline that containerizes the interface and deploys the ML model with all the bells and whistles discussed throughout this post. Requiring that the average citizen data scientist writes and debugs Kubernetes deployments (along with artifact deployments to secure blob storage etc.) is not realistic. Also, doing every deployment of a new model in this kind of manual fashion does not seem scalable or very *MLOps*-ish to me, even less so if all of the sudden multiple people are required to get one model into production.

A vision could be to implement a tool for more tightly linking mlflow to Seldon-core. The tool would be responsible for ensuring that only models that conform to a proper interface could be pushed to the "Production" stage in the mlflow model registry, and once a model does get promoted to the Production stage, the tool should be responsible for ensuring that the model gets deployed, that permissions are propagated from mlflow to Kubernetes, that monitoring is functional, etc. The vision of such as tool ties into a final principle which I did not touch much upon during the blog post, and that is the principle of *automation*. Essentially for an MLOps framework to be considered "complete," it should incorporate all the best practices we see in fields such as DevOps and automate all the things that can be automated through code.

Although I do not think that mlflow + Seldon is a "seamless" MLOps framework, I do think that it represents a potent combination, and I think both are very awesome tools. At the end of the day, I don't see a comprehensive MLOps system ever being trivial, and as such, the fact that the presented solutions are open source is a huge benefit in my book in that it means we can readily fill any missing holes ourselves.