The objective of this exercise to build an end-to-end pipeline that is satisfies all the requirements mentioned in the `Task.md` file.

There is an option to work with Apache Airflow or Kubeflow. Since the developer is new to both, it is fairly straightforward to go with the Airflow approach and builds DAGs (ingest >> preprocess >> train >> evaluate >> deploy). Airflow offers numerous operator and the `PythonOperator`can easily help plug the functions and build the pipeline.

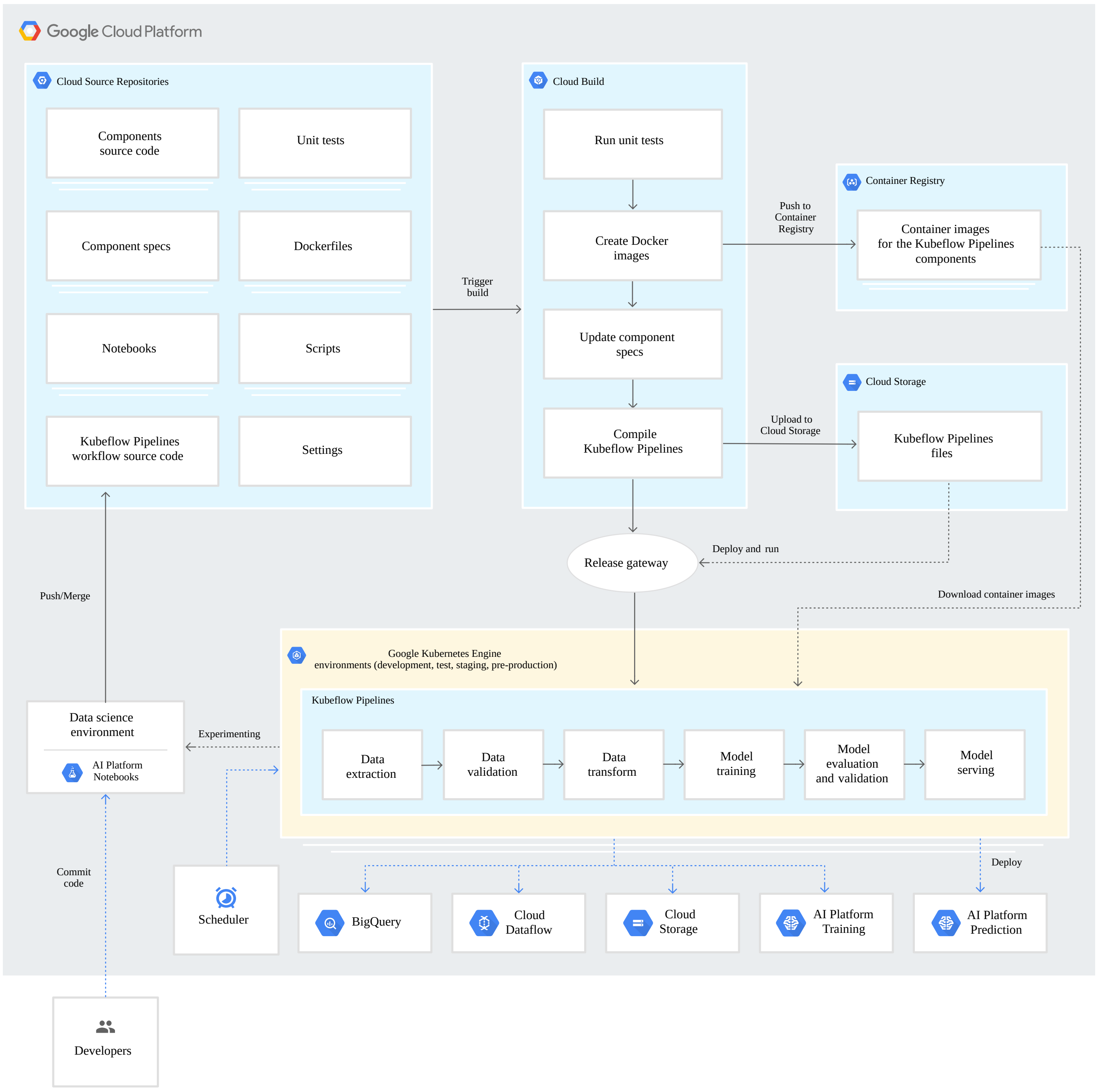
However, the developer knows that such an implementation comes at a cost. For instance, he will need to move to `KubernetesOperator` to run the tasks on a Kubernetes cluster for scalability. Additionally, he will need to opt for `CeleryExecutor` or any other distributed executor because sequential execution may become a bottleneck and under-utilize workers. Although, there should be a work around to use Airflow for the most part, it needs further evaluation.

Considering the above points (and knowing that Unity is using GCP services), it is a good idea to follow Kubernetes route.

Now, focusing on the architecture, there are broadly 3 steps:

- **Data** ingestion, validation, transformation, feature-generation, splitting.
- **Model** training (+ hyper-params tuning), evaluation, serving.
- **Life cycle** monitoring, logging, iterating model.

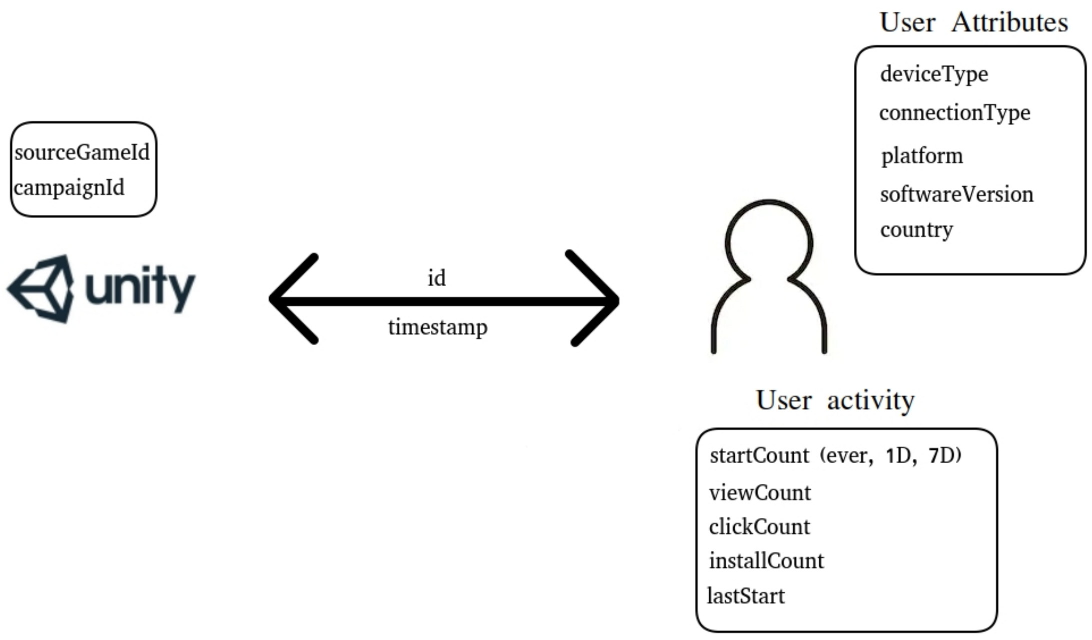
There are numerous ways to approach the above steps and below is a purely GCP-based example for the same, courtesy Google blogs.



Solution

Data understanding

After an initial overview of all the features, we can bucket features in 4 broader categories, namely, company-side features, interaction features, user attributes and user activity.



Based on the provided field descriptors, we will represent the information flow in a logical and consistent way that binds back to how this data may have been generated.

A user arrives with the user attributes {device type, connection type, platform, software version, country} and a video ad is shown. The ad is associated with a campaign (campaign ID) and promotes a game (game ID). This leads to an impression event, which is the granularity of the provided data.

Conversion funnel: (Assumptions based on the provided definition)

startCount: This refers to all the videos ads ever started on the user's device

viewCount: Since all impressions are implicitly viewed; the meaning of view, here, can imply that the ad was viewed for more than a specific time interval or completely. Thus, this refers to all the video ads viewed, as per definition, by the user.

clickCount: If the user click's on the ad while or after watching. This refers to all the ads that we clicked.

installCount: If the ad led to an installation. This refers to all previous installation via video ads.

Note: The definition says 'how many times the user has installed games from this network ever'. There is not further information about what a network is so we will assume that it means the Unity ad network.

User journey: When a user is impressed with an ad, the user can view -> click -> install. If the installation happens, install flag is set to 1 for the impression and other information is logged as usual.

Other factors: The installation flag also derives from factors outside the logged information. For example, the user may have clicked but the app/play store didn't function. Or the user was trying to press the skip/close button on the ad but clicked wrong and arrived at the app/play store. Overall, the user's propensity to install depends on the following factors:

- the game type or the user-recommendation match
- the ad placement; between some loading period vs in the middle of the play
- Funnel attributes with given user attributes i.e. ease of skipping or clicking, re-direct to app/play store, installation size, paid/free app.
- unknowns

With the above analysis, we can say that a campaign is a collection of video ads from various games and a game can be a part of multiple campaigns. There is no information if the video ad corresponding to a gameId-campaignId pair is unique for that pair or reused across all campaigns for that game.

Components for pipeline

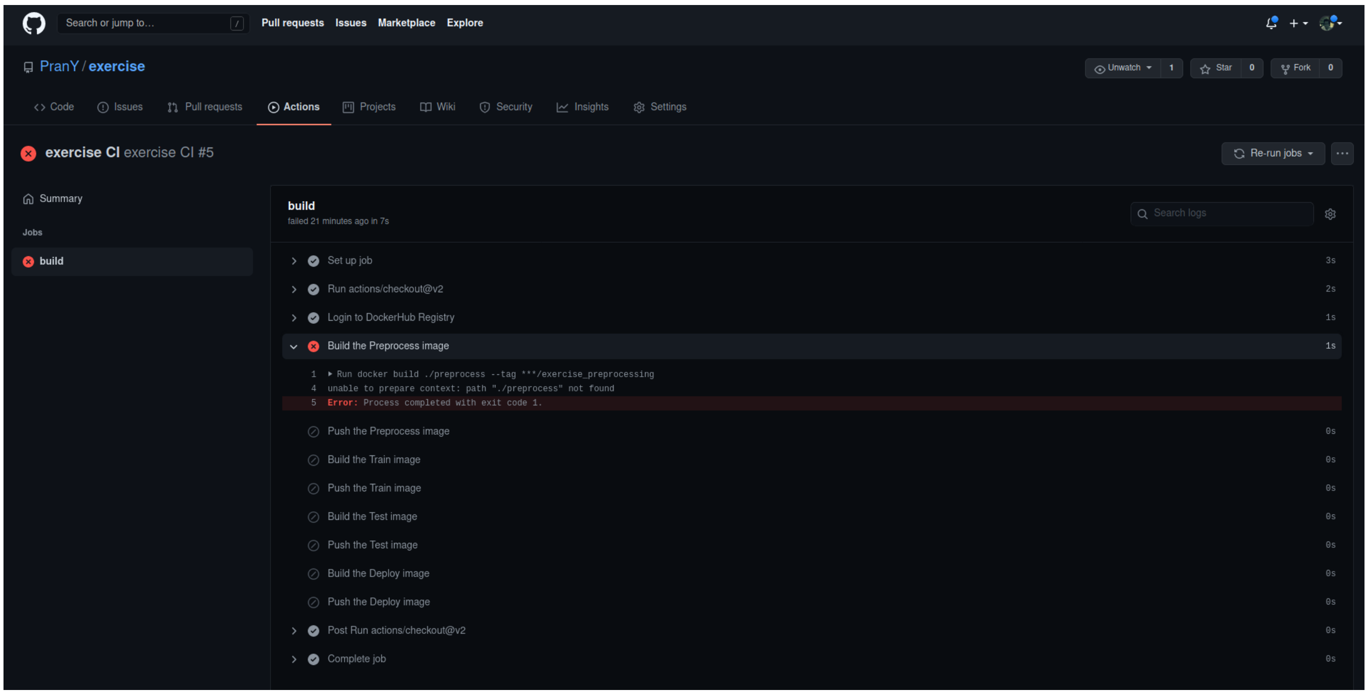
After the EDA, we can start designing the components for our kubeflow pipeline. We will follow `preprocess >> train(+evaluation) >> test(prediction) >> deploy`. We will remove the ingest component as the data is locally available.

> If time permits, we can add basic spark functionality coupled with a querying mechanism that can pull data dynamically form RDS, as an example to containerize the ingestion process.

ML pipeline

Building components and running CI

The pipeline solution will follow a similar but scaled-down pattern as shown in the above GCP image. The component structure will be `preprocess >> train >> test >> deploy`. After investigating the kubeflow-pipeline, we observed that we needed our workflow to push the docker containers to a hosting or manually inject the files in pods. **Pushing containers to dockerhub is not an option due to private data**. We will leave the pipeline functional and locally run the pipeline to generate all the files. Additionally, Github actions are also setup for the CI process in case we want an alternate pipeline build process (image below).



exercise CI exercise CI #5

build
failed 21 minutes ago in 7s

```
Set up job                                   3s
Run actions/checkout@v2                      2s
Login to DockerHub Registry                  1s
Build the Preprocess image                   1s
  ▸ Run docker build ./preprocess --tag ***/exercise_preprocessing
  unable to prepare context: path "./preprocess" not found
  Error: Process completed with exit code 1.
Push the Preprocess image                    0s
Build the Train image                        0s
Push the Train image                         0s
Build the Test image                         0s
Push the Test image                          0s
Build the Deploy image                       0s
Push the Deploy image                        0s
Post Run actions/checkout@v2                 0s
Complete job                                 0s
```

Note: This is expected to fail and ideally the file reading process will be from S3 or anything other location.

Model versioning and hyper-param tuning

Since the model performance holds less weightage in this exercise, we will focus on the next steps. Ideally, we should implement MLFlow with all our components for file versioning and use it for hyper parameter tuning in the train component. We can scale the pods for the training step in case we need to cover a wider param scan. Another alternate is Pachyderm but evaluating it against MLflow is not required at the moment.

Serving

Initially, the plan was to use kf serving but after reviewing documentation and downstream feature richness, we have decidede to move with Seldon. Here are a few benefits

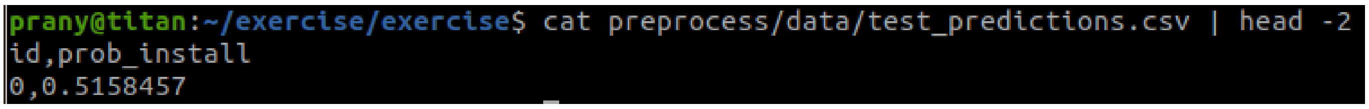
- Reusable inference servers - It provides out-of-the-box model servers for SKLearn, XGBoost, TF and MLFlow.
- Better documentation
- Coverts ML models to REST/gRPC microservices
- Advanced metrics and logging (Prometheus + Grafana)
- Explainers for models and a feedback mechanism for live learnings.
- A/B testing with canary, green-blue, shadow deployments.
- Otlier detection and more.

> The seldon-core installation with Helm consumed about 20% of the time in the process. Helm is confusing and lacks documentation, use Kustomize.

> Seldon-core's XGBoost server is the easiest way to deploy our model but due to a bug, it took almost a day to figure out why the deployments didn't work. We have raised a *Bug Report* on github and found a way around.

Proof of a working deployment

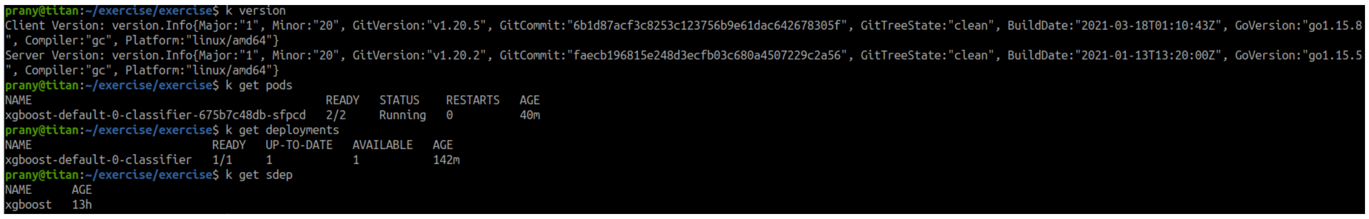
Let's start with a prediction generated using our pipeline for reference.



```
prany@titan:~/exercise/exercise$ cat preprocess/data/test_predictions.csv | head -2
id,prob_install
0,0.5158457
```

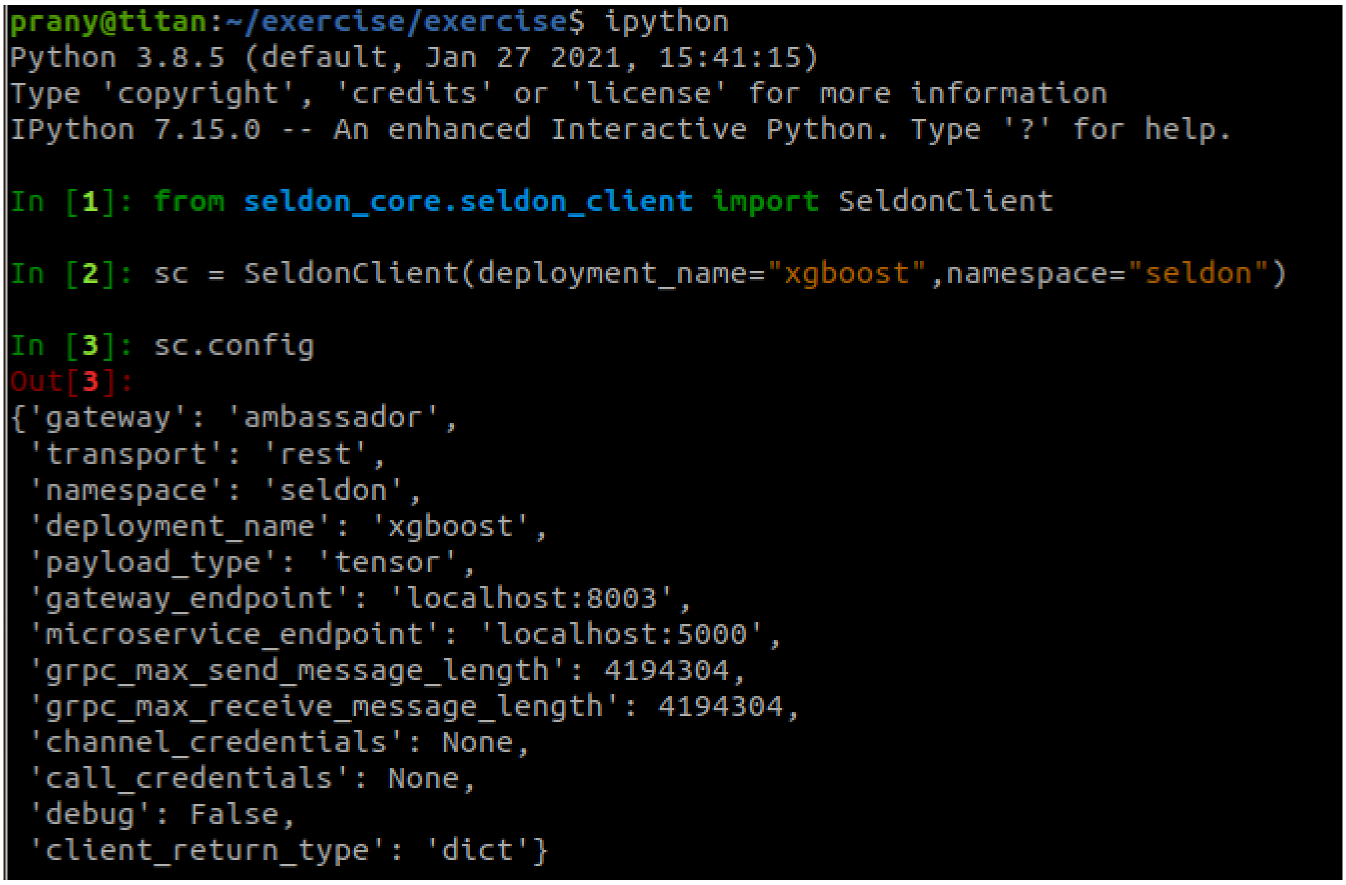
The is the probability of install for the test case in the first-row. We will ensure that we get the same probability from the seldon deployment.

Let's take a look at our deployment and ensure everything is working as expected.



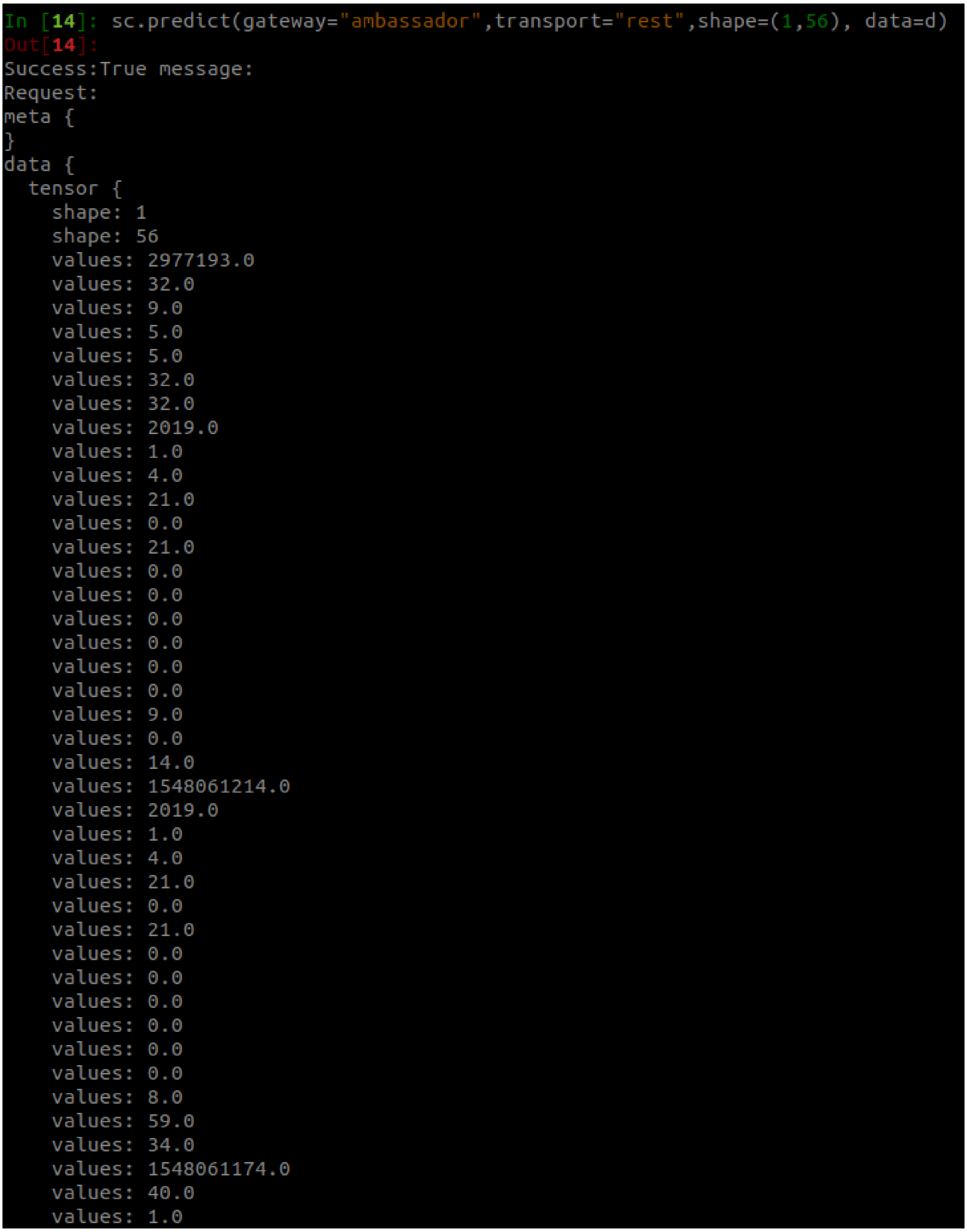
```
prany@titan:~/exercise/exercise$ k version
Client Version: version.Info{Major:"1", Minor:"20", GitVersion:"v1.20.5", GitCommit:"6b1d87acf3c8253c123756b9e61dac642678305f", GitTreeState:"clean", BuildDate:"2021-03-18T01:10:43Z", GoVersion:"go1.15.8
", Compiler:"gc", Platform:"linux/amd64"}
Server Version: version.Info{Major:"1", Minor:"20", GitVersion:"v1.20.2", GitCommit:"faecb196815e248d3ecfb03c680a4507229c2a56", GitTreeState:"clean", BuildDate:"2021-01-13T13:20:00Z", GoVersion:"go1.15.5
", Compiler:"gc", Platform:"linux/amd64"}
prany@titan:~/exercise/exercise$ k get pods
NAME                                      READY   STATUS    RESTARTS   AGE
xgboost-default-0-classifier-675b7c48db-sfpcd   2/2     Running   0          40m
prany@titan:~/exercise/exercise$ k get deployments
NAME                           READY   UP-TO-DATE   AVAILABLE   AGE
xgboost-default-0-classifier   1/1     1            1           142m
prany@titan:~/exercise/exercise$ k get sdep
NAME      AGE
xgboost   13h
```

Everything looks okay (finally! after reading the source-code of seldon-core for a day). Now is a good time to test our server.

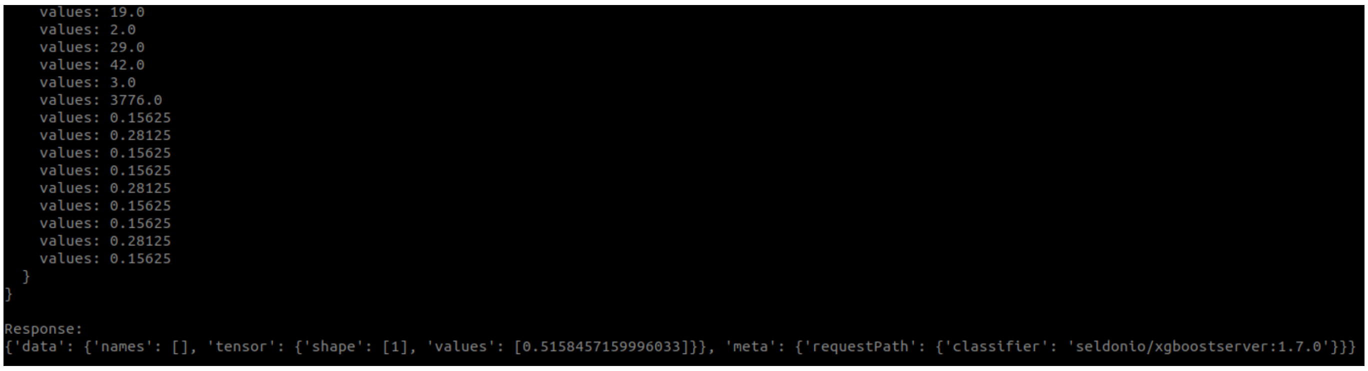


```
prany@titan:~/exercise/exercise$ ipython
Python 3.8.5 (default, Jan 27 2021, 15:41:15)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.15.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: from seldon_core.seldon_client import SeldonClient

In [2]: sc = SeldonClient(deployment_name="xgboost",namespace="seldon")

In [3]: sc.config
Out[3]:
{'gateway': 'ambassador',
 'transport': 'rest',
 'namespace': 'seldon',
 'deployment_name': 'xgboost',
 'payload_type': 'tensor',
 'gateway_endpoint': 'localhost:8003',
 'microservice_endpoint': 'localhost:5000',
 'grpc_max_send_message_length': 4194304,
 'grpc_max_receive_message_length': 4194304,
 'channel_credentials': None,
 'call_credentials': None,
 'debug': False,
 'client_return_type': 'dict'}
```

Initialization seems to be okay, now we will test the first row from our test data to see if the deployment works.



```
In [14]: sc.predict(gateway="ambassador",transport="rest",shape=(1,56), data=d)
Out[14]:
Success:True message:
Request:
meta {
}
data {
  tensor {
    shape: 1
    shape: 56
    values: 2977193.0
    values: 32.0
    values: 9.0
    values: 5.0
    values: 5.0
    values: 32.0
    values: 32.0
    values: 2019.0
    values: 1.0
    values: 4.0
    values: 21.0
    values: 0.0
    values: 21.0
    values: 0.0
    values: 0.0
    values: 0.0
    values: 0.0
    values: 0.0
    values: 0.0
    values: 9.0
    values: 0.0
    values: 14.0
    values: 1548061214.0
    values: 2019.0
    values: 1.0
    values: 4.0
    values: 21.0
    values: 0.0
    values: 21.0
    values: 0.0
    values: 0.0
    values: 0.0
    values: 0.0
    values: 0.0
    values: 0.0
    values: 8.0
    values: 59.0
    values: 34.0
    values: 1548061174.0
    values: 40.0
    values: 1.0
```



```
    values: 19.0
    values: 2.0
    values: 29.0
    values: 42.0
    values: 3.0
    values: 3776.0
    values: 0.15625
    values: 0.28125
    values: 0.15625
    values: 0.15625
    values: 0.28125
    values: 0.15625
    values: 0.15625
    values: 0.28125
    values: 0.15625
  }
}

Response:
{'data': {'names': [], 'tensor': {'shape': [1], 'values': [0.5158457159996033]}}, 'meta': {'requestPath': {'classifier': 'seldonio/xgboostserver:1.7.0'}}}
```

Yay! It works.

Model Life Cycle

Currently, we are not epxloring the logging, monitoring and A/B testing options but, **these are all available with the deployment**. If time permits, we can simulate a live server that samples 5K records per sec from the test data and requests the SeldonClient. That way we can simulate live traffic and can test our A/B tests or bandit stratergies with a simple change in our deployment yamls.