Deploying Deep Learning Models

OSCON Tensorflow Day 2018

Hannes Hapke

@hanneshapke

Slides available at https://github.com/hanneshapke/Deploying-Deep-Learning-Models

Does the following scenario sound familiar?

Joe (data scientist): Hey Jane, my model is validated and tested. I would like to deploy it.

Jane (back-end engineer): Great, do you have an API for it?

Joe (data scientist): Hey Jane, my model is validated and tested. I would like to deploy it.

Jane (back-end engineer): Great, do you have an API for it?

Joe: API? Our model runs on TF/Python. The entire back-end runs on Ruby. I haven't written Ruby in years ...

Jane: Ufff, I have never written Tensorflow code. Is that a Python library?

Joe (data scientist): Hey Jane, my model is validated and tested. I would like to deploy it.

Jane (back-end engineer): Great, do you have an API for it?

Joe: API? Our model runs on TF/Python. The entire back-end runs on Ruby. I haven't written Ruby in years ...

Jane: Ufff, I have never written Tensorflow code. Is that a Python library?

Joe: Hm, I guess, I'll write some Ruby API code then.

What's the problem?

Who owns the API?

Data science code deployed to API instances?

Different language expertises are needed

Coordinate release cycles between teams?

Coordination about model versioning

Hi, I'm Hannes.

Data Science Engineer at Cambia Health Solutions

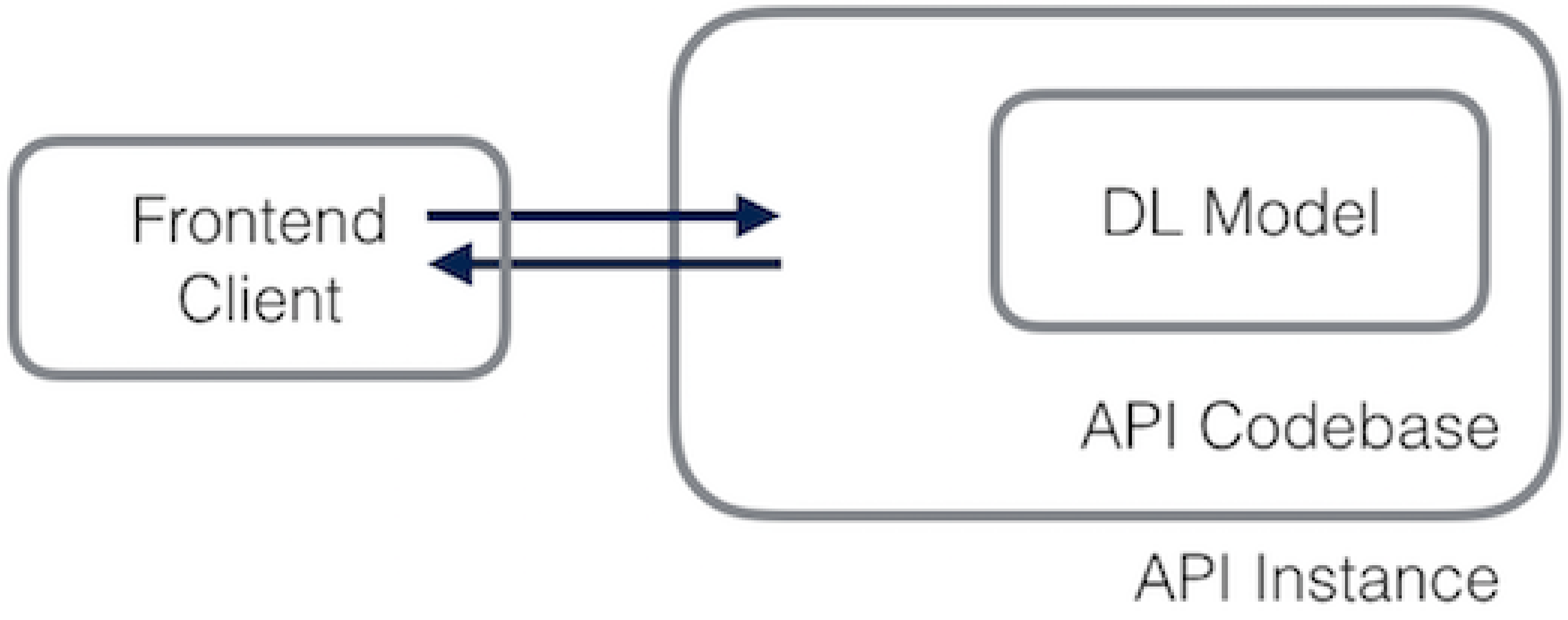
Agenda

- Requirements for Model Deployments
- Sample project
- How not to deploy models
- Deploying Models
    - with Tensorflow Serving on premise
    - in the Cloud
    - with Kubeflow
- Other deployment options

Infrastructure Architectures

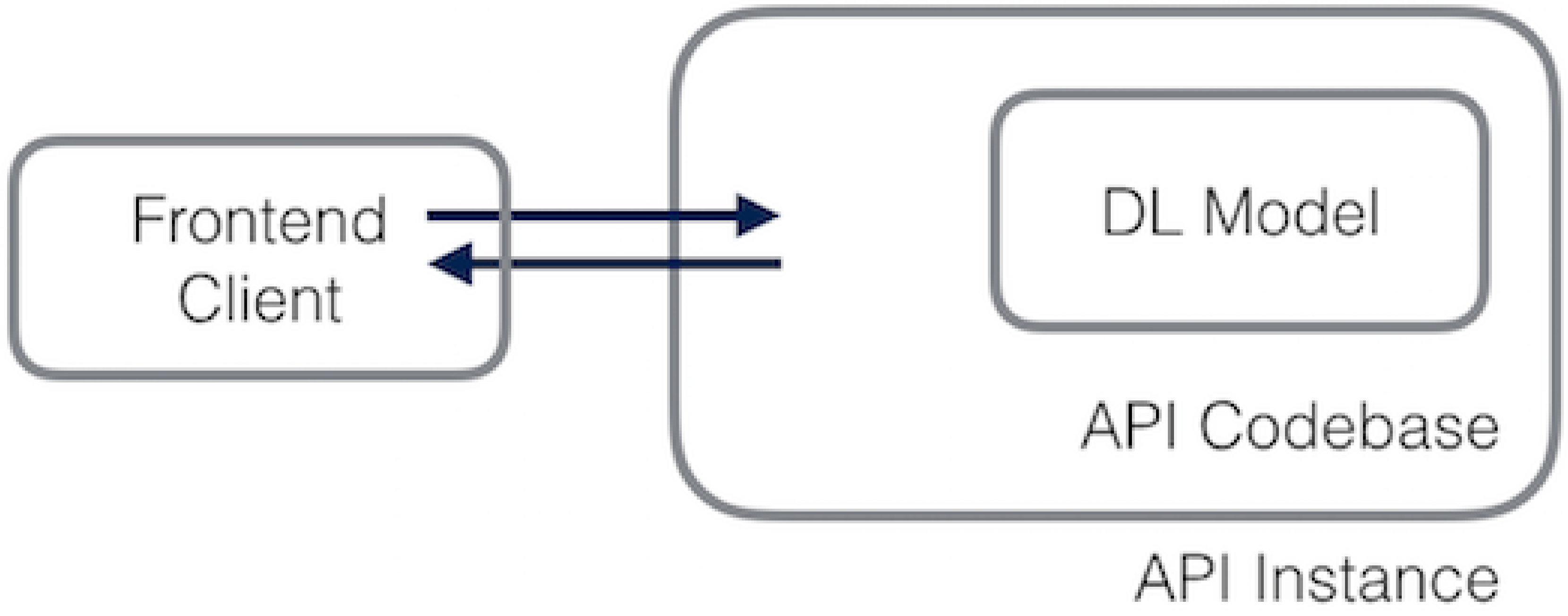
Infrastructure Architectures

Loading models on the backend server

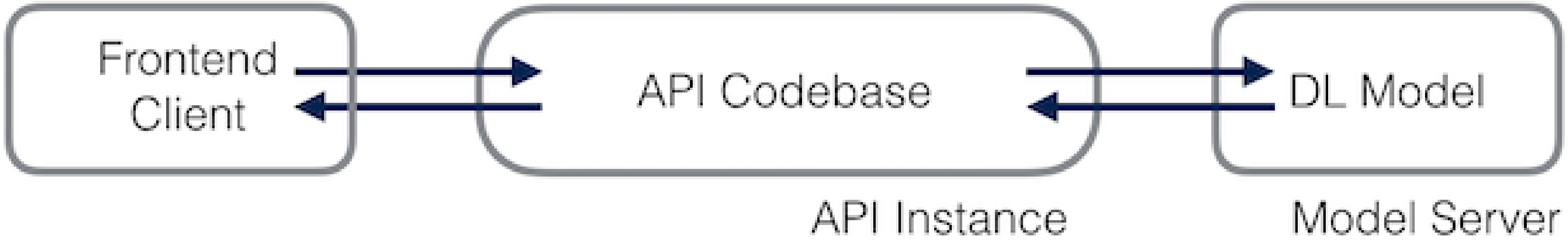


Infrastructure Architectures

Loading models on the backend server



Using a model server



Model deployments should ...

Model deployments should …

1. Separate data science code from backend code

Model deployments should …

1. Separate data science code from backend code

2. Reduce boilerplate code

Model deployments should ...

1. Separate data science code from backend code
2. Reduce boilerplate code
3. Allow isolation of memory and CPU requirements

Model deployments should ...

1. Separate data science code from backend code

2. Reduce boilerplate code

3. Allow isolation of memory and CPU requirements

4. Support multiple models

Model deployments should ...

1. Separate data science code from backend code
2. Reduce boilerplate code
3. Allow isolation of memory and CPU requirements
4. Support multiple models
5. Server should handle requests (e.g. timeouts)

Sample Project

Model Structure

Let's predict Amazon product ratings based on the comments with a small LSTM network.

```
model_input = Input(shape=(MAX_TOKENS,))
x = Embedding(input_dim=len(CHARS), output_dim=10, input_length=MAX_TOKENS)(model_input)
x = LSTM(128)(text_input)
output = Dense(5, activation='softmax')(x)
model = Model(inputs=text_input, outputs=output)
optimizer = RMSprop(lr=0.01)
model.compile(loss='categorical_crossentropy', optimizer=optimizer)
```

```
model.fit(x_train, y_train,
          batch_size=BATCH_SIZE, epochs=EPOCHS,
          verbose=1, validation_data=(x_test, y_test),
          callbacks=[keras.callbacks.ModelCheckpoint('/tmp/amazon_ratings',
                                                     monitor='val_loss',
                                                     verbose=1, save_best_only=True,
                                                     save_weights_only=False,
                                                     mode='auto', period=1)])
```

Testing our Model

Negative Review

```
>> test_sentence = "horrible book, don't buy it"
>> test_vector = clean_data(test_sentence, max_tokens=MAX_TOKENS, sup_chars=CHARS)
>> model.predict(test_vector.reshape(1, MAX_TOKENS, len(CHARS)))
[[0.5927979  0.23748466 0.10798287 0.03301411 0.02872046]]
```

Positive Review

```
>> test_sentence = "Awesome product."
>> test_vector = clean_data(test_sentence, max_tokens=MAX_TOKENS, sup_chars=CHARS)
>> model.predict(test_vector.reshape(1, MAX_TOKENS, len(CHARS)))
[[0.03493131 0.0394276  0.08326671 0.2957105  0.5466638 ]]
```

How not to deploy a model …

Deploy with Flask + Keras

```python
@app.route("/predict", methods=["POST"])
def predict():
    # initialize the data dictionary that will be returned from the
    # view
    data = {"success": False}

    # ensure an image was properly uploaded to our endpoint
    if flask.request.method == "POST":
        if flask.request.files.get("image"):
            # read the image in PIL format
            image = flask.request.files["image"].read()
            image = Image.open(io.BytesIO(image))

            # preprocess the image and prepare it for classification
            image = prepare_image(image, target=(224, 224))

            # classify the input image and then initialize the list
            # of predictions to return to the client
            preds = model.predict(image)
            results = imagenet_utils.decode_predictions(preds)
            data["predictions"] = []

            # loop over the results and add them to the list of
            # returned predictions
            for (imagenetID, label, prob) in results[0]:
                r = {"label": label, "probability": float(prob)}
                data["predictions"].append(r)

            # indicate that the request was a success
            data["success"] = True

    # return the data dictionary as a JSON response
    return flask.jsonify(data)
```

Code snippet from [Keras Blog]

Don't deploy that way if can avoid it.

Why?

Why?

1. Mix of data science and backend code

Why?

1. Mix of data science and backend code

2. Boilerplate API code

Why?

1. Mix of data science and backend code

2. Boilerplate API code

3. API instances need enough memory to load models

Why?

1. Mix of data science and backend code

2. Boilerplate API code

3. API instances need enough memory to load models

4. Multiple models?

Why?

1. Mix of data science and backend code

2. Boilerplate API code

3. API instances need enough memory to load models

4. Multiple models?

5. No timeout handling

Use Tensorflow Serving instead.

But before that, let's chat about some terms.

Important Terms

Protocol Buffers

Protobufs are a method of serializing structured data. Binary format.

Bazel

Automation tool to build software. Similar to Make or Apache Maven.

gRPC

(Google) Remote Procedure Call. HTTP/2 based. Uses ProtoBuf.

REST

Representational State Transfer. Architectural style for web services.

Welcome Tensorflow Serving!

Steps to deploy a model

Steps to deploy a model

1. Export model structure weights, as well as model signatures as protobuf

Steps to deploy a model

1. Export model structure weights, as well as model signatures as protobuf

2. Set up the Tensorflow Server

Steps to deploy a model

1. Export model structure weights, as well as model signatures as protobuf

2. Set up the Tensorflow Server

3. Create a gRPC client

Steps to deploy a model

1. Export model structure weights, as well as model signatures as protobuf

2. Set up the Tensorflow Server

3. Create a gRPC client

4. Load the model

Export our Keras model to Protobuf

```
import os
from keras import backend as K
import tensorflow as tf

tf.app.flags.DEFINE_integer('training_iteration', 1000, 'number of training iterations.')
tf.app.flags.DEFINE_integer('model_version', 1, 'version number of the model.')
tf.app.flags.DEFINE_string('work_dir', '/tmp', 'Working directory.')
FLAGS = tf.app.flags.FLAGS

export_path_base = '/tmp/amazon_reviews'
export_path = os.path.join(tf.compat.as_bytes(export_path_base),
               tf.compat.as_bytes(str(FLAGS.model_version)))

builder = tf.saved_model.builder.SavedModelBuilder(export_path)

signature = tf.saved_model.signature_def_utils.predict_signature_def(
    inputs={'input': model.input}, outputs={'rating_prob': model.output})

builder.add_meta_graph_and_variables(
    sess=K.get_session(), tags=[tf.saved_model.tag_constants.SERVING],
    signature_def_map={
        tf.saved_model.signature_constants.DEFAULT_SERVING_SIGNATURE_DEF_KEY: signature })
builder.save()
```

Let's unpack what we just saw.

Flags

- Let's set flags with information relevant for the model

```
import tensorflow as tf

tf.app.flags.DEFINE_integer('training_iteration', 1000,
                            'number of training iterations.')
tf.app.flags.DEFINE_integer('model_version', 1,
                            'version number of the model.')
tf.app.flags.DEFINE_string('work_dir', '/tmp',
                           'Working directory.')
FLAGS = tf.app.flags.FLAGS
```

Model Signatures

- Tensorflow Serving requires that every model has a model signature
- Signature define the generic *inputs* and *outputs* of a function

```
signature = tf.saved_model.signature_def_utils.predict_signature_def(
    inputs={model_name + '_input': model.input}, outputs={model_name + '_output': model.output})

builder.add_meta_graph_and_variables(
      sess=K.get_session(), tags=[tf.saved_model.tag_constants.SERVING],
      signature_def_map={
          tf.saved_model.signature_constants.DEFAULT_SERVING_SIGNATURE_DEF_KEY:
              signature
      })
```

Exporting the Model

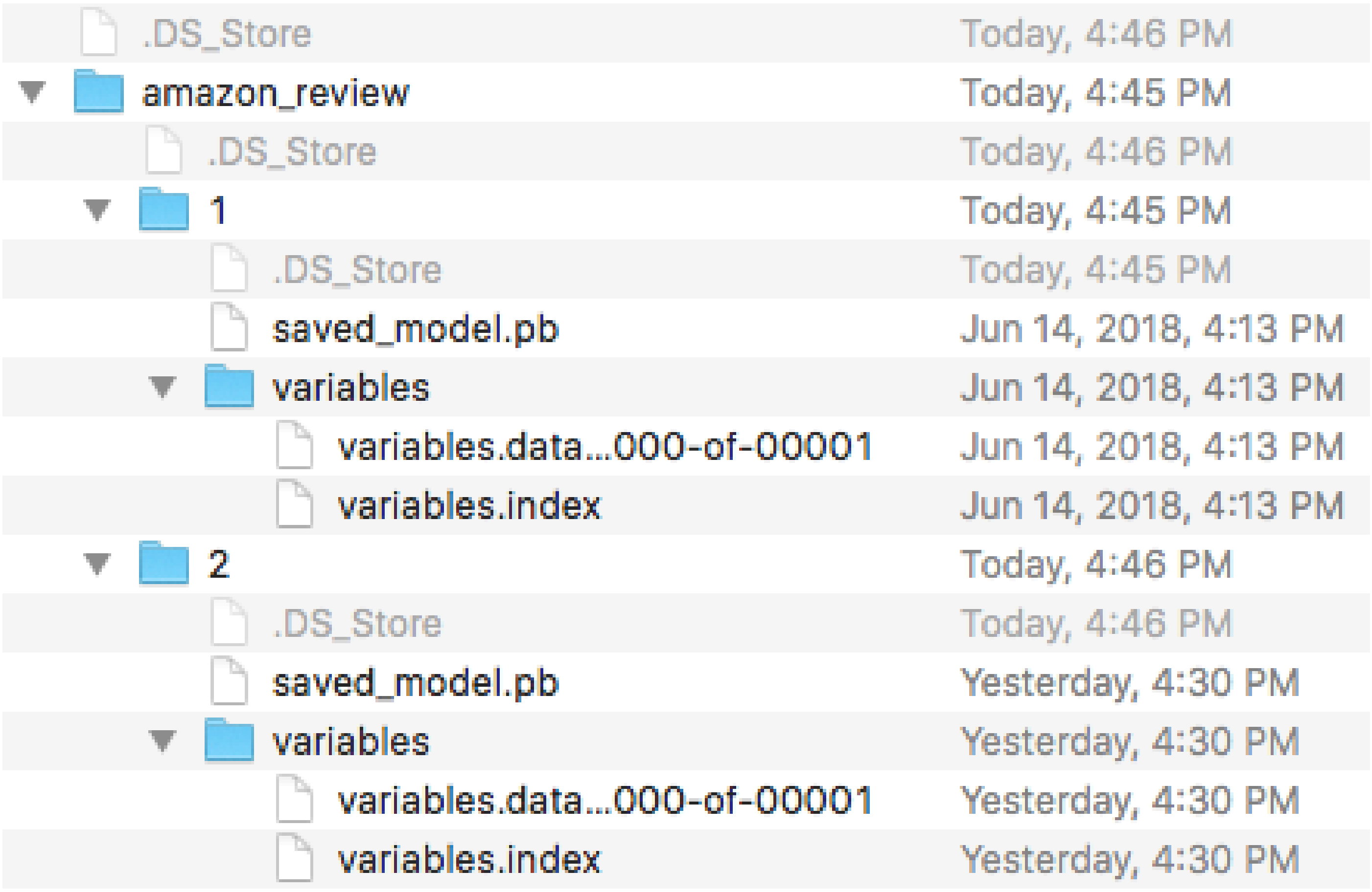
- The *SavedModelBuilder* will export your model to a predefined ProtoBuf format

```
export_path_base = '/tmp/amazon_reviews'
export_path = os.path.join(
      tf.compat.as_bytes(export_path_base),
      tf.compat.as_bytes(str(FLAGS.model_version)))

print('Exporting trained model to', export_path)

builder = tf.saved_model.builder.SavedModelBuilder(export_path)
...
builder.save()
```

Now you have exported your model.

Exported Models

- You should find these protobuf files in your folder structure



- The files should include a
  - saved_model.pb
  - variable.index
  - one or more variable.data* files.

Let's set up your Tensorflow server

Creating a Tensorflow Serving Environment

- If you need optimizations, clone the TF Serving repo and build your server with Bazel
- Otherwise install Tensorflow server in a Docker container

```
$ git clone git@github.com:hanneshapke/Deploying_Deep_Learning_Models.git

$ docker build --pull -t $USER/tensorflow-serving-devel-cpu \
                     -f {path to repo}/\
                     Deploying_Deep_Learning_Models/\
                     examples/Dockerfile .
```

Starting up the Server

- Start up the container with

```
$ docker run -it -p 8500:8500 -p 8501:8501
             -v {model_path}/exported_models/amazon_review/:/models
             $USER/tensorflow-serving-devel-cpu:latest /bin/bash
```

What's happening inside the Docker container?

- Starting up the Tensorflow Serving instance

```
$[docker bash] tensorflow_model_server --port=8500
                                       --model_name={model_name}
                                       --model_base_path=/models/{model_name}
```

What's happening inside the Docker container?

- Starting up the Tensorflow Serving instance

```
$[docker bash] tensorflow_model_server --port=8500
                                       --model_name={model_name}
                                       --model_base_path=/models/{model_name}
```

- This should generate similar output like this

```
2018-06-29 00:02:05.611608:
    tensorflow_serving/model_servers/server_core.cc:444 Adding/updating models.
2018-06-29 00:02:05.611712:
    tensorflow_serving/model_servers/server_core.cc:499  (Re-)adding model: amazon_review
2018-06-29 00:02:05.729657:
    tensorflow_serving/core/basic_manager.cc:716
    Successfully reserved resources to load servable {name: amazon_review version: 2}
2018-06-29 00:02:05.729731:
    tensorflow_serving/core/loader_harness.cc:66
    Approving load for servable version {name: amazon_review version: 2}
2018-06-29 00:02:05.729761:
    tensorflow_serving/core/loader_harness.cc:74
    Loading servable version {name: amazon_review version: 2}
...
2018-06-29 00:02:05.855197:
    tensorflow_serving/core/loader_harness.cc:86
    Successfully loaded servable version {name: amazon_review version: 2}
2018-06-29 00:02:05.863820:
    tensorflow_serving/model_servers/main.cc:323 Running ModelServer at 0.0.0.0:8500 ...
2018-06-29 00:02:05.870805:
    tensorflow_serving/model_servers/main.cc:333 Exporting HTTP/REST API at:localhost:8501 ...
evhttp_server.cc : 235 RAW: Entering the event loop ...
```

Let's export a new model version

```
$ python examples/export_keras_model.py
```

Let's export a new model version

```
$ python examples/export_keras_model.py
```

- Tensorflow Serving will detect the new version and load it automatically

```
2018-07-15 20:39:02.561131:
    external/org_tensorflow/tensorflow/contrib/session_bundle/bundle_shim.cc:360 Attempting to load native SavedModelBundle in bundle-shim from:
    /models/amazon_review/3
2018-07-15 20:39:02.561509:
    external/org_tensorflow/tensorflow/cc/saved_model/loader.cc:242 Loading SavedModel with tags: { serve }; from: /models/amazon_review/3
2018-07-15 20:39:02.593076:
    external/org_tensorflow/tensorflow/cc/saved_model/loader.cc:161 Restoring SavedModel bundle.
2018-07-15 20:39:02.621946:
    external/org_tensorflow/tensorflow/cc/saved_model/loader.cc:196 Running LegacyInitOp on SavedModel bundle.
2018-07-15 20:39:02.627210:
    external/org_tensorflow/tensorflow/cc/saved_model/loader.cc:291 SavedModel load for tags { serve }; Status: success.
    Took 65974 microseconds.
2018-07-15 20:39:02.637227:
    tensorflow_serving/core/loader_harness.cc:86 Successfully loaded servable version {name: amazon_review version: 3}
2018-07-15 20:39:02.637451:
    tensorflow_serving/core/loader_harness.cc:137 Quiescing servable version {name: amazon_review version: 2}
2018-07-15 20:39:02.637751:
    tensorflow_serving/core/loader_harness.cc:144 Done quiescing servable version {name: amazon_review version: 2}
2018-07-15 20:39:02.639501:
    tensorflow_serving/core/loader_harness.cc:119 Unloading servable version {name: amazon_review version: 2}
2018-07-15 20:39:02.645189:
    ./tensorflow_serving/core/simple_loader.h:294 Calling MallocExtension_ReleaseToSystem() after servable unload with 251467
2018-07-15 20:39:02.645312:
    tensorflow_serving/core/loader_harness.cc:127 Done unloading servable version {name: amazon_review version: 2}
```

Useful tips

Inspect your models

```
$ saved_model_cli show --dir=/models/{model_name}/{version_number}
                       --tag_set serve
                       --signature_def serving_default
```

Useful tips

Inspect your models

```
$ saved_model_cli show --dir=/models/{model_name}/{version_number}
                        --tag_set serve
                        --signature_def serving_default
```

- `saved_model_cli` should return the signature information

```
The given SavedModel SignatureDef contains the following input(s):
  inputs['amazon_review_input'] tensor_info:
      dtype: DT_FLOAT
      shape: (-1, 50)
      name: input_1:0
The given SavedModel SignatureDef contains the following output(s):
  outputs['amazon_review_output'] tensor_info:
      dtype: DT_FLOAT
      shape: (-1, 5)
      name: dense_1/Softmax:0
Method name is: tensorflow/serving/predict
```

Tensorflow Serving Client

Dependecies

- tensorflow_serving
- grpc

```
$ pip install tensorflow-serving-api grpc
```

Using Python 3? [Python 3 compliance](#) [Fix Python 3 issues](#)

Tensorflow Serving Client

Connecting to the RPC host

```
from grpc.beta import implementations
from tensorflow_serving.apis import prediction_service_pb2

def get_stub(host='127.0.0.1', port='8500'):
    channel = implementations.insecure_channel(host, int(port))
    stub = prediction_service_pb2.beta_create_PredictionService_stub(channel)
    return stub
```

Tensorflow Serving Client

Request prediction using gRPC

- Very barebone implementation!

```
def get_model_prediction(model_input, stub,
                         model_name='amazon_reviews',
                         signature_name='serving_default'):

    request = predict_pb2.PredictRequest()
    request.model_spec.name = model_name
    request.model_spec.signature_name = signature_name

    request.inputs['input'].CopyFrom(
        tf.contrib.util.make_tensor_proto(
            model_input.reshape(1, 50),
            verify_shape=True, shape=(1, 50)))

    response = stub.Predict.future(request, 5.0)  # wait max 5s
    return response.result().outputs["rating_prob"].float_val
```

Tensorflow Serving Client

Request prediction using gRPC

- Very barebone implementation!

```
>>> sentence = "this product is really helpful"
>>> model_input = clean_data_encoded(sentence)

>>> get_model_prediction(model_input, stub)
[0.02509271726012230, 0.03738045319914818, 0.09454590082168579,
0.33069494366645813, 0.5122858881950378]
```

Tensorflow Serving Client

Request prediction from a specific model version

- You can specify the specific model version
- If no model version is provided, TF Serving loads the model with the latest model version

```
request = predict_pb2.PredictRequest()
request.model_spec.name = 'amazon_reviews'
request.model_spec.version.value = 1
```

Tensorflow Serving Client

Obtain model meta data

```
def get_model_meta(model_name, stub):
    request = get_model_metadata_pb2.GetModelMetadataRequest()
    request.model_spec.name = model_name
    request.metadata_field.append("signature_def")
    response = stub.GetModelMetadata(request, 5)
    return response.metadata['signature_def']

>>> meta = get_model_meta(model_name, stub)
>>> print(meta.SerializeToString().decode("utf-8", 'ignore'))
type.googleapis.com/tensorflow.serving.SignatureDefMap
serving_default
amazon_review_input
        input_1:0
               2@
amazon_review_output(
dense_1/Softmax:0
               tensorflow/serving/predict
```

Tensorflow Serving Client

Obtain model version

```
def get_model_version(model_name, stub):
    request = get_model_metadata_pb2.GetModelMetadataRequest()
    request.model_spec.name = model_name
    request.metadata_field.append("signature_def")
    response = stub.GetModelMetadata(request, 5)
    return response.model_spec.version.value
```

```
>>> model_name = 'amazon_review'
>>> stub = get_stub()

>>> get_model_version(model_name, stub)
2L
```

Tensorflow Serving Client using REST

- Tensorflow Serving supports REST requests since release 1.8

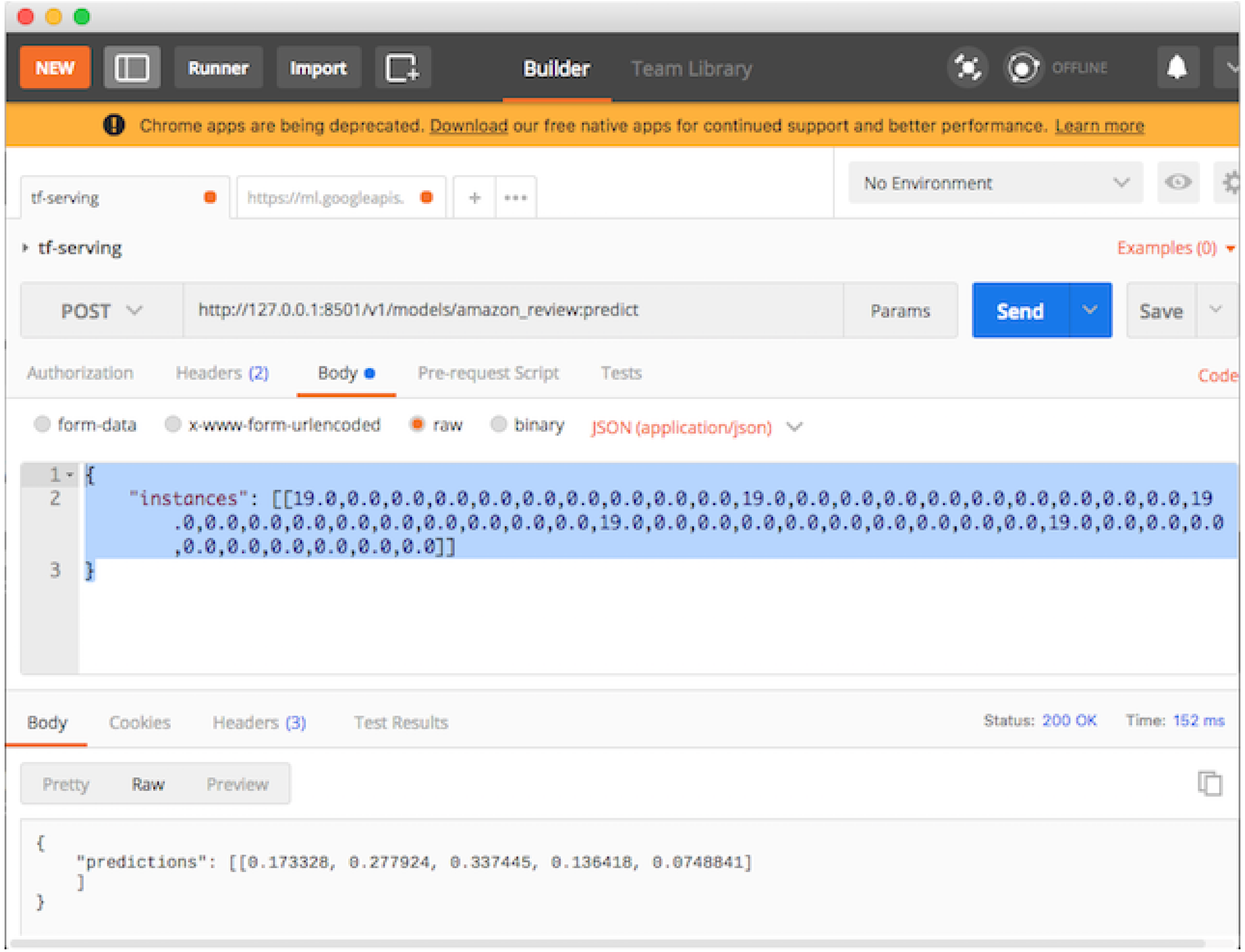
```
$[docker bash] tensorflow_model_server --port=8500
                                        --rest_api_port=8501
                                        --model_name={model_name}
                                        --model_base_path=/models/{model_name}
```

Tensorflow Serving Client using REST

- The URI should be
    - `http://host:port/<URI>:<VERB>`
    - URI `/v1/models/{model_name}/versions/{model_version}`
    - verb `classify|regress|predict`
- Use the Python *requests* package for REST calls.

```
def get_model_prediction(model_input, model_name='amazon_review', signature_name='serving_default
    url = get_rest_url(model_name)
    data = {"instances": [model_input.tolist()]}

    rv = requests.post(url, data=json.dumps(data))
    if rv.status_code != requests.codes.ok:
        rv.raise_for_status()

    return rv.json()['predictions']
```

Tensorflow Serving Client using REST



Tensorflow Serving

Serve multiple models

- Provide a server config file *config.file*

```
model_config_list: {
    config:{
        name:"amazon_reviews",
        base_path:"/models/{model_name}",
        model_platform:"tensorflow",
        model_version_policy: { all: {} }
    },
    config:{
        name:"amazon_ratings",
        base_path:"/models/{other_model_name}",
        model_platform:"tensorflow",
        model_version_policy: { all: {} }
    }
}
```

Tensorflow Serving

Serve multiple models

- Start the server using config file

Instead of

```
$ tensorflow_model_server --port=8500
                         --model_name={model_name}
                         --model_base_path=/models/{model_name}
```

use

```
$ tensorflow_model_server --port=8500
                         --model_config_file=/path/to/\
                           config/file.config
```

How to do A/B Testing?

- Easily possible since multiple versions can be served
- A/B testing of models can be performed by selecting the models from the client side
- Set the specific version in your gRPC or REST request

```
from random import random
def get_rest_url(model_name, host='127.0.1', port='8501',
                 verb='predict', version=None):
    url = "http://{host}:{port}/v1/models/{model_name}".format(
        host=host, port=port, model_name=model_name)
    if version:
        url += 'versions/{version}'.format(version=version)
    url += ':{verb}'.format(verb=verb)
    return url

# 10% of requests to the latest model
version = 1 if random() > 0.1 else None
url = get_rest_url('amazon_review', version=version)
```

[Discussion on the TF.org mailing list](#)

Good idea?

Good idea?

1. No mix of data science and backend code

Good idea?

1. No mix of data science and backend code
2. No boilerplate API code

Good idea?

1. No mix of data science and backend code
2. No boilerplate API code
3. APIs can be serverless

Good idea?

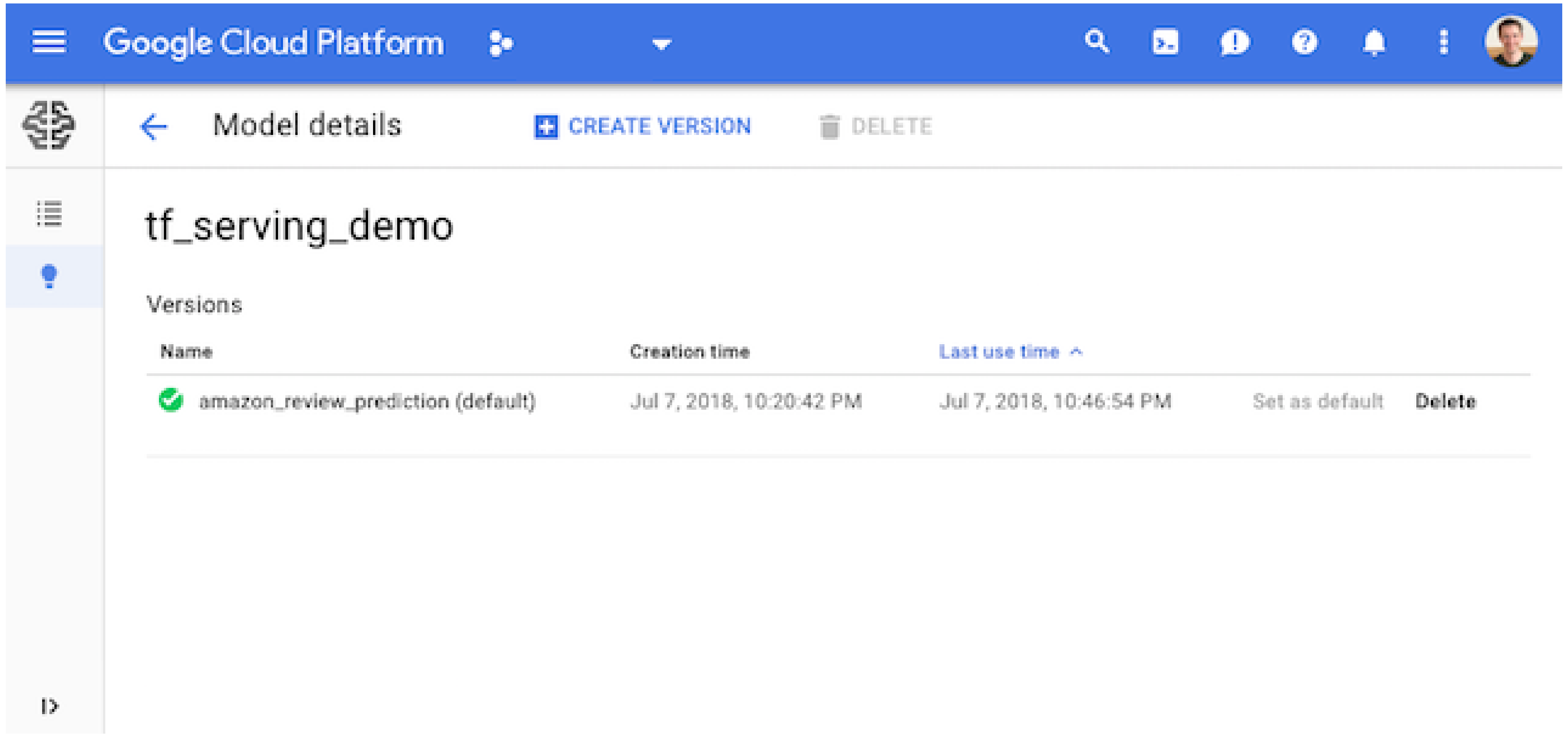
1. No mix of data science and backend code
2. No boilerplate API code
3. APIs can be serverless
4. Multiple models? Of course.

Good idea?

1. No mix of data science and backend code
2. No boilerplate API code
3. APIs can be serverless
4. Multiple models? Of course.
5. Asyncronous requests. Heck yes!

Serving Models via the Cloud

- Exported Tensorflow or Keras models can be served via the *Google Cloud Platform* and Google Cloud ML Engine
- Detailed information on GCP Model Deployments



Serving Models via Google Cloud ML Engine

- Copy the exported model to a Google storage bucket

```
$ gsutil cp -r inception gs://<bucket-name>
$ gsutil ls -r gs://<bucket-name>
gs://<bucket-name>/amazon_review/:

gs://<bucket-name>/amazon_review/1/:
gs://<bucket-name>/amazon_review/1/saved_model.pb
...
```

- Create a json file with the request data

```
$ MODEL_NAME="tf_serving_demo"
$ INPUT_DATA_FILE="request_data.json"
$ VERSION_NAME="amazon_review_prediction"
$ gcloud ml-engine predict --model $MODEL_NAME  \
                           --version $VERSION_NAME \
                           --json-instances $INPUT_DATA_FILE
```

install of gsutils: https://cloud.google.com/storage/docs/gsutil_install

Kubeflow for all

- Scalable ML stack for Kubernetes
- Supports Jupyter notebooks, and Tensorflow Jobs
- Kubeflow integrations with Tensorflow Serving
- Kubernetes takes care of scaling your ML infrastructure

Other Deployment Options

Other Deployment Options

Seldon

- Deployment solution for ML on Kubernetes
- Supports Scikit, H2O and Tensorflow
- Supports REST and gRPC end-points
- Provides model routers (e.g. for server side A/B testing)

Other Deployment Options

Seldon

- Deployment solution for ML on Kubernetes
- Supports Scikit, H2O and Tensorflow
- Supports REST and gRPC end-points
- Provides model routers (e.g. for server side A/B testing)

MLflow

- Databricks package supports deployments
- Supports Scikit and Tensorflow
- Provides REST end-points
- Supports deployments to AzureML, Amazon Sagemaker, Spark clusters

Conclusion

Conclusion

- Reasons to serve models with Tensorflow Serving
  - Separates data science code from API code
  - No boilerplate code
  - Can handle multiple models and versions

Conclusion

- Reasons to serve models with Tensorflow Serving
  - Separates data science code from API code
  - No boilerplate code
  - Can handle multiple models and versions
- Steps to deploy
  - Export your model
  - Setup your server
  - Request predictions via gRPC or REST

Thank you and happy deploying!

@hanneshapke