Operationalizing the Model

Advanced ML with TensorFlow on GCP

End-to-End Lab on Structured Data ML

Production ML Systems

Image Classification Models

Sequence Models

Recommendation Systems

Steps involved in doing ML on GCP

1 Explore the dataset

2 Create the dataset

3 Build the model

4 **Operationalize the model**

Building an ML model involves:

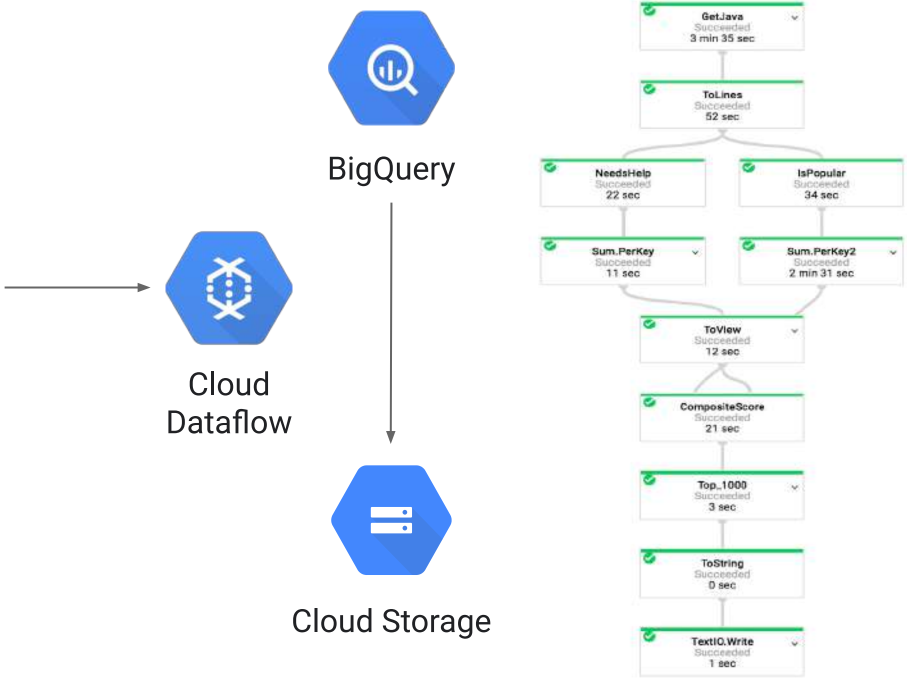
Creating
the dataset

Building
the model

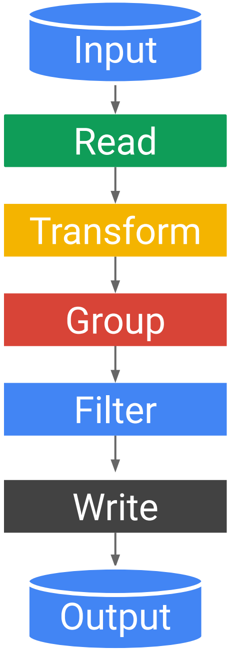
Operationalizing
the model

Beam is a way to write elastic data processing pipelines

```
def packageHelp(record, keyword):
   count=0
   package_name=''
   if record is not None:
     lines=record.split('\n')
     for line in lines:
       if line.startswith(keyword):
         package_name=line
       if 'FIXME' in line or 'TODO' in l
         count+=1
     packages = (getPackages(package_nam
     for p in packages:
         yield (p,count)
```



Open-source API, Google infrastructure



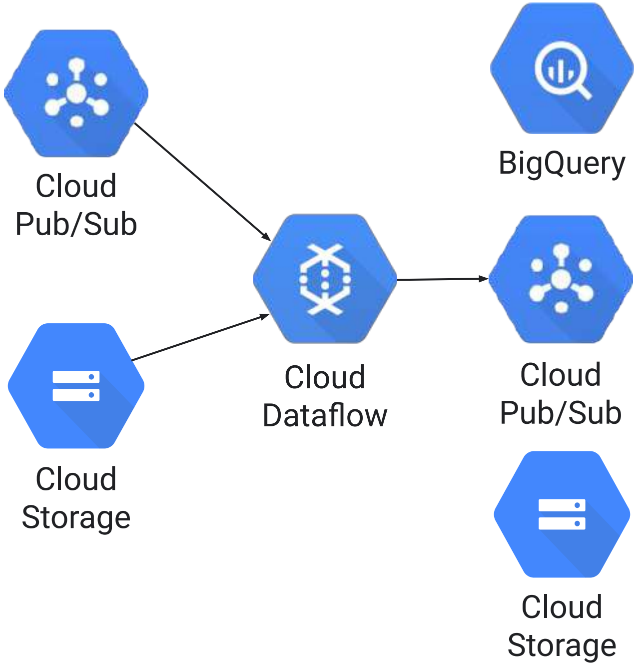

```
p = beam.Pipeline()
(p

      | beam.io.ReadFromText('gs://..')

      | beam.Map(Transform)

      | beam.GroupByKey()

      | beam.FlatMap(Filter)

      | beam.io.WriteToText('gs://...')
 )

p.run();
```

Open-source API (Apache Beam) can be executed on Flink, Spark, etc. also

Parallel tasks (autoscaled by execution framework)

```
def Transform(line):
     return (parse_custid(line), 1)

def Filter(key, values):
     return sum(values) > 10
```

The code is the same between real-time and batch




```
p = beam.Pipeline()
(p
      | beam.io.ReadStringsFromPubSub('project/topic')
      | beam.WindowInto(SlidingWindows(60))
      | beam.Map(Transform)
      | beam.GroupByKey()
      | beam.FlatMap(Filter)
      | beam.io.WriteToBigQuery(table)
)
p.run()
```

An example Beam pipeline for BigQuery->CSV on cloud

```
import apache_beam as beam

def transform(rowdict):
   import copy
   result = copy.deepcopy(rowdict)
   if rowdict['a'] > 0:
      result['c'] = result['a'] * result['b']
      yield ','.join([ str(result[k]) if k in result else 'None' for k in ['a','b','c'] ])

if __name__ == '__main__':
   p = beam.Pipeline(argv=sys.argv)
   selquery = 'SELECT a,b FROM someds.sometable'
   (p
      | beam.io.Read(beam.io.BigQuerySource(query = selquery,
                                   use_standard_sql = True)) # read input
      | beam.Map(transform_data) # do some processing
      | beam.io.WriteToText('gs://...') # write output
   )
   p.run() # run the pipeline
```

Executing pipeline (Python)

Simply running main() runs pipeline locally.

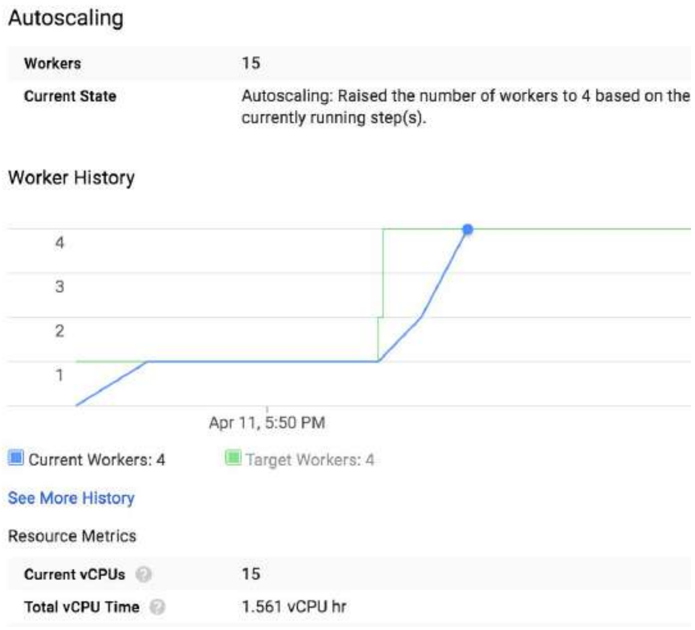
```
python ./etl.py
```

To run on cloud, specify cloud parameters.

```
python ./etl.py \
      --project=$PROJECT \
      --job_name=myjob \
      --staging_location=gs://$BUCKET/staging/ \
      --temp_location=gs://$BUCKET/staging/ \
      --runner=DataflowRunner    # DirectRunner would be local
```

Split the full dataset into train/eval and do preprocessing

BigQuery -> Dataflow -> CSV

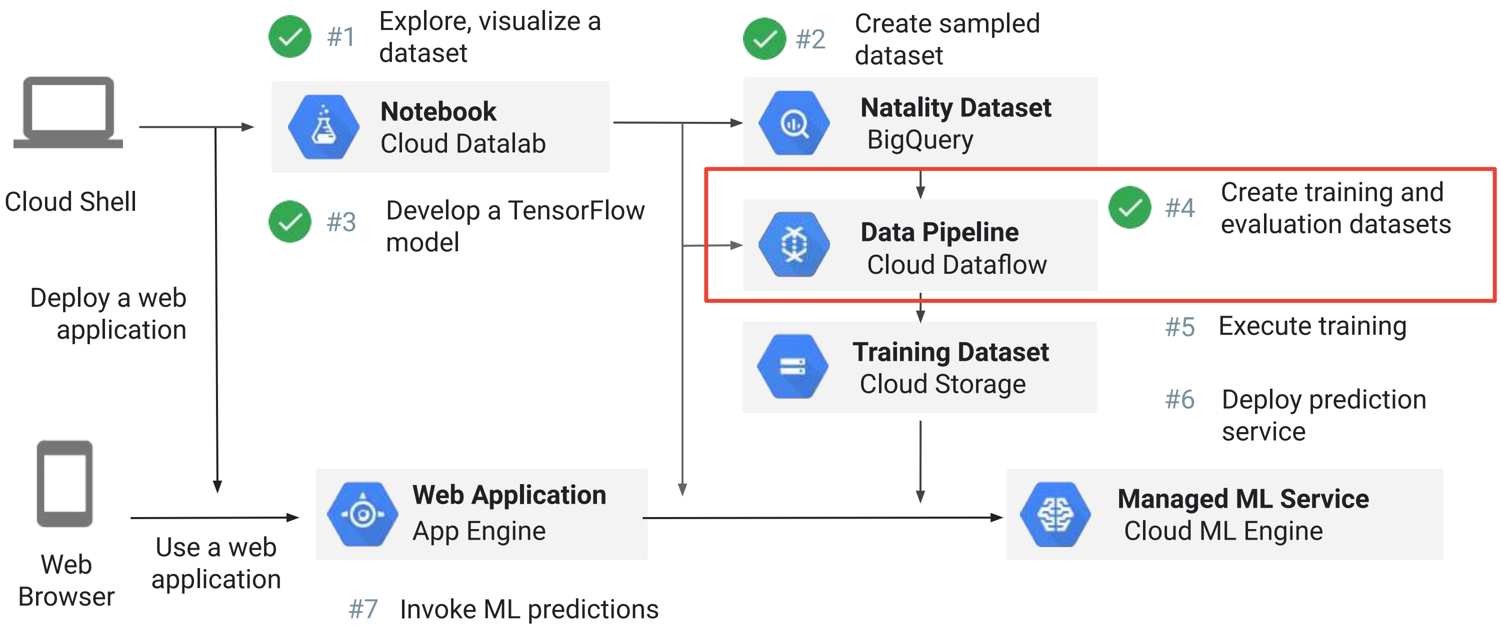


Lab

Preprocessing using Cloud Dataflow

In this lab, you use Cloud Dataflow to create datasets for Machine Learning.

The end-to-end process



Building an ML model involves:

Creating
the dataset

Building
the model

Operationalizing
the model

Create task.py to parse command-line parameters and send to train_and_evaluate

task.py

```
parser.add_argument(
    '--train_data_paths', required=True)
parser.add_argument(
    '--train_steps', ...
```

model.py

```
def train_and_evaluate(args):
    estimator = tf.estimator.DNNRegressor(
                     model_dir=args['output_dir'],
                     feature_columns=feature_cols,
                     hidden_units=args['hidden_units'])
    train_spec=tf.estimator.TrainSpec(
                     input_fn=read_dataset(args['train_data_paths'],
                                    batch_size=args['train_batch_size'],
                                    mode=tf.contrib.learn.ModeKeys.TRAIN),
                     max_steps=args['train_steps'])
    exporter = tf.estimator.LatestExporter('exporter', serving_input_fn)
    eval_spec=tf.estimator.EvalSpec(...)
    tf.estimator.train_and_evaluate(estimator, train_spec, eval_spec)
```

The model.py contains the ML model in TensorFlow (Estimator API)

	Example of the code in model.py (see Lab #3)

Training and evaluation input functions	`CSV_COLUMNS = ...`
	`def read_dataset(filename, mode, batch_size=512):`
	`    ...`

Feature columns	`INPUT_COLUMNS = [` `    tf.feature_column.numeric_column('gestation_weeks'),`

Feature engineering	`def add_more_features(feats):`
	`  # feature crosses etc.`
	`  return feats`

Serving input function	`def serving_input_fn():`
	`    ...`
	`    return tf.estimator.export.ServingInputReceiver(features, feature_pholders)`

Train and evaluate loop	`def train_and_evaluate(args):`
	`    ...`
	`    tf.estimator.train_and_evaluate(estimator, train_spec, eval_spec)`

Package TensorFlow model as a Python package

```
taxifare/
taxifare/PKG-INFO
taxifare/setup.cfg
taxifare/setup.py
taxifare/trainer/
taxifare/trainer/__init__.py
taxifare/trainer/task.py
taxifare/trainer/model.py
```

Python packages need to contain an `__init__.py` in every folder.

Verify that the model works as a Python package

```
export PYTHONPATH=${PYTHONPATH}:/somedir/babyweight
python -m trainer.task \
  --train_data_paths="/somedir/datasets/*train*" \
  --eval_data_paths=/somedir/datasets/*valid*  \
  --output_dir=/somedir/output \
  --train_steps=100 --job-dir=/tmp
```

You use distributed TensorFlow on Cloud ML Engine

Run TF at scale

tf.estimator	High-level API for distributed training
tf.layers, tf.losses, tf.metrics	Components useful when building custom NN models
Core TensorFlow (Python)	Python API gives you full control
Core TensorFlow (C++)	C++ API is quite low level
CPU, GPU, TPU, Android	TF runs on different hardware

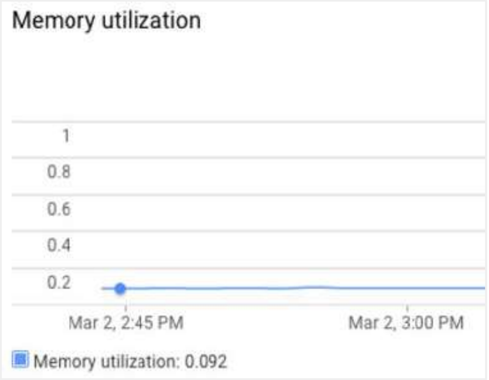
Cloud ML Engine

Use the gcloud command to submit the training job either locally or to the cloud

```
gcloud ml-engine local train \
   --module-name=trainer.task \
   --package-path=/somedir/babyweight/trainer \
   -- \
   --train_data_paths etc.
   REST as before
```

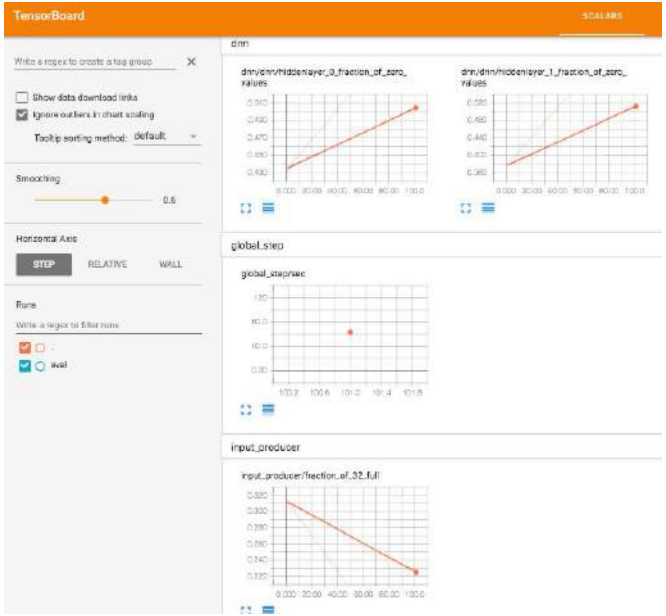
```
gcloud ml-engine jobs submit training $JOBNAME \
   --region=$REGION \
   --module-name=trainer.task \
   --job-dir=$OUTDIR --staging-bucket=gs://$BUCKET \
   --scale-tier=BASIC \
   REST as before
```

Monitor training jobs with GCP Console



You can also view CPU and Memory utilization charts for this training job with StackDriver Monitoring.

Monitor training jobs with TensorBoard



Pre-made estimators automatically populate summary data that you can examine and visualize using TensorBoard.

Lab

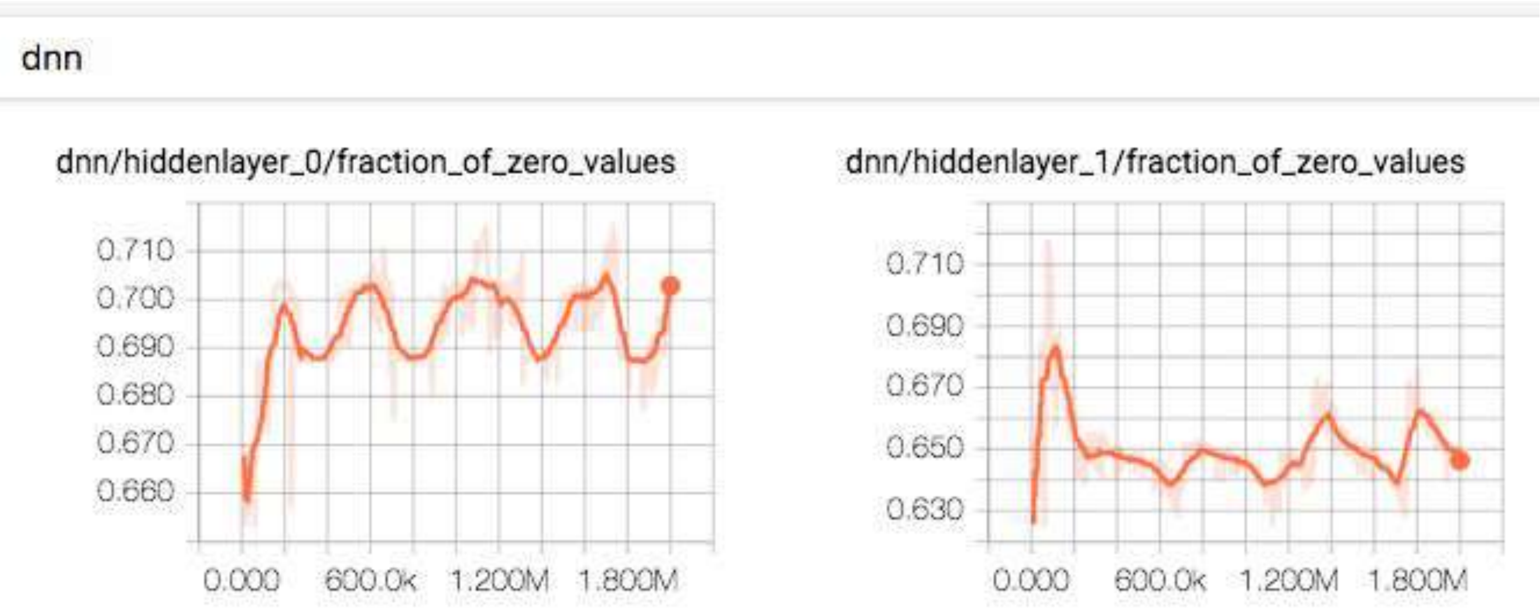
Training on Cloud ML Engine

In this lab, you will do distributed training using Cloud ML Engine, and improve model accuracy using hyperparameter tuning.

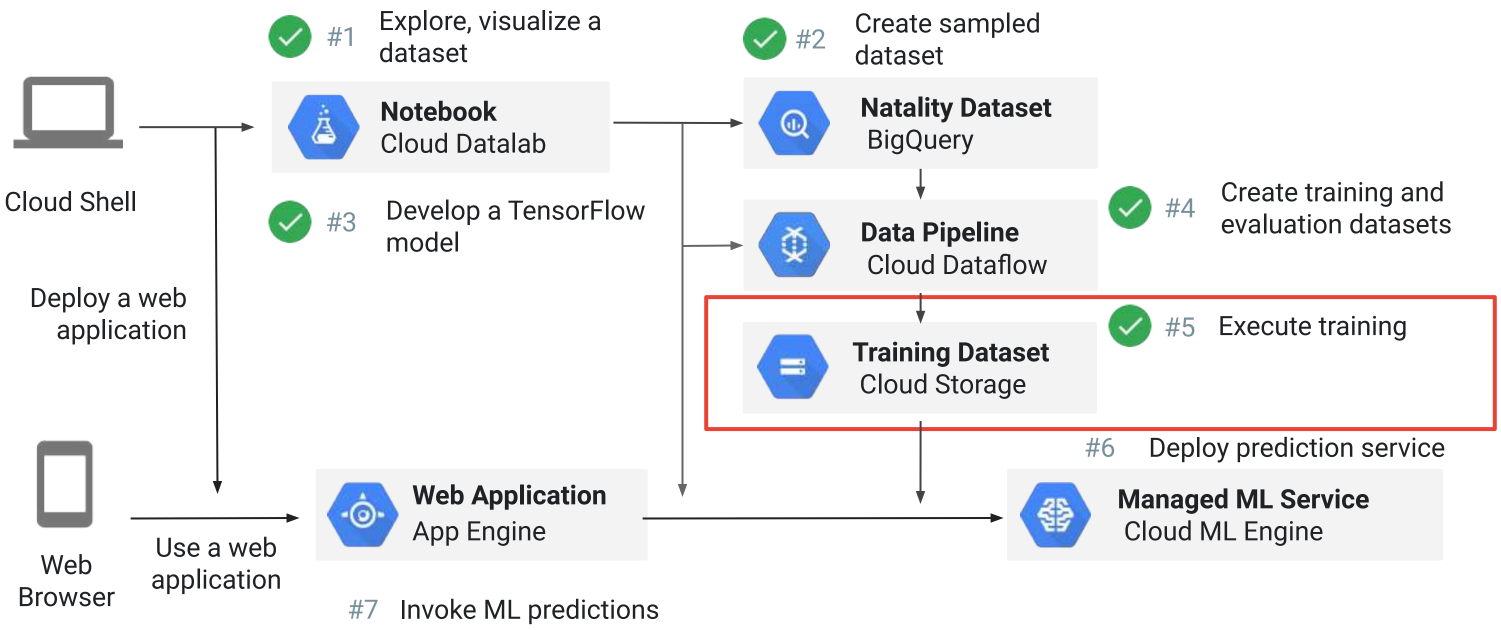
Lab Steps

1. Change the batch size if necessary.
2. Calculate the train steps based on the # examples.
3. Make hyperparameter command-line parameters.

Submit the training job on the full dataset and monitor using TensorBoard



The end-to-end process

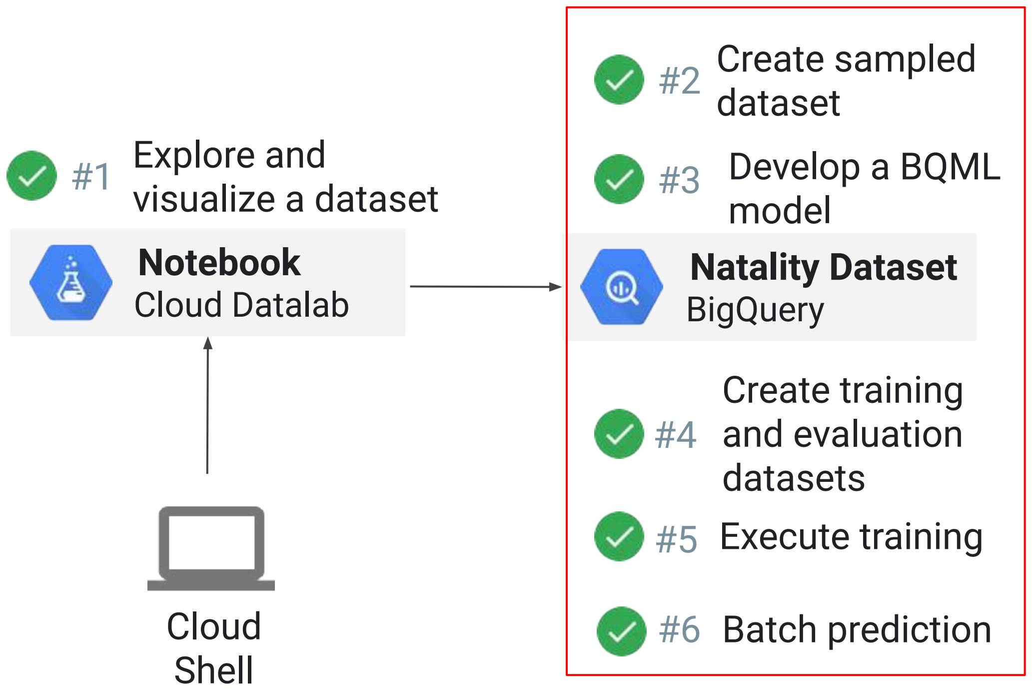


It can take days to months to create an ML model

Simplify model development with BigQuery ML

1. Use familiar SQL for machine learning.
2. Train models over all their data in BigQuery.
3. Don’t worry about hypertuning or feature transformations.

Simplify model development with BigQuery ML



Behind the scenes

With 2 lines of code:

- Leverages BigQuery's processing power to build a model.
- Auto-tunes learning rate.
- Auto-splits data into training and test.

For the advanced user:

- L1/L2 regularization.
- 3 strategies for training/test split: Random, Sequential, Custom.
- Set learning rate.

Supported features

1. StandardSQL and UDFs within the ML queries.
2. Linear Regression (Forecasting).
3. Binary Logistic Regression (Classification).
4. Model evaluation functions for standard metrics, including ROC and precision-recall curves.
5. Model weight inspection.
6. Feature distribution analysis through standard functions.

The end-to-end BQML process

ETL into BigQuery 1

- BQ Public Data Sources
- Google Marketing Platform
  - Analytics
  - Ads
- YouTube
- Your Datasets

Preprocess Features 2

- Explore
- Join
- Create Train / Test Tables

3

```
#standardSQL
CREATE MODEL
ecommerce.classification

OPTIONS
  (

model_type='logistic_reg',
input_label_cols =
['will_buy_later']

    ) AS

# SQL query with training data
```

4

```
#standardSQL
SELECT
        roc_auc,
        accuracy,
    precision,
    recall
FROM
  ML.EVALUATE(MODEL
ecommerce.classification

# SQL query with eval data
```

5

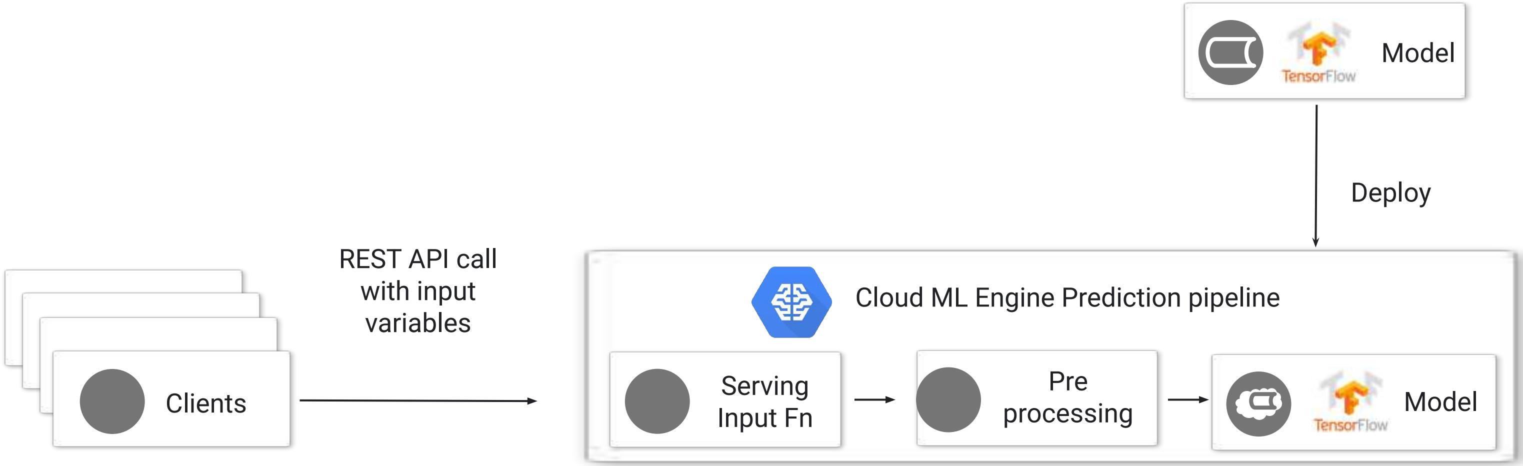
```
#standardSQL
SELECT * FROM
   ML.PREDICT
(MODEL ecommerce.classification,
(

# SQL query with test data
```

Lab

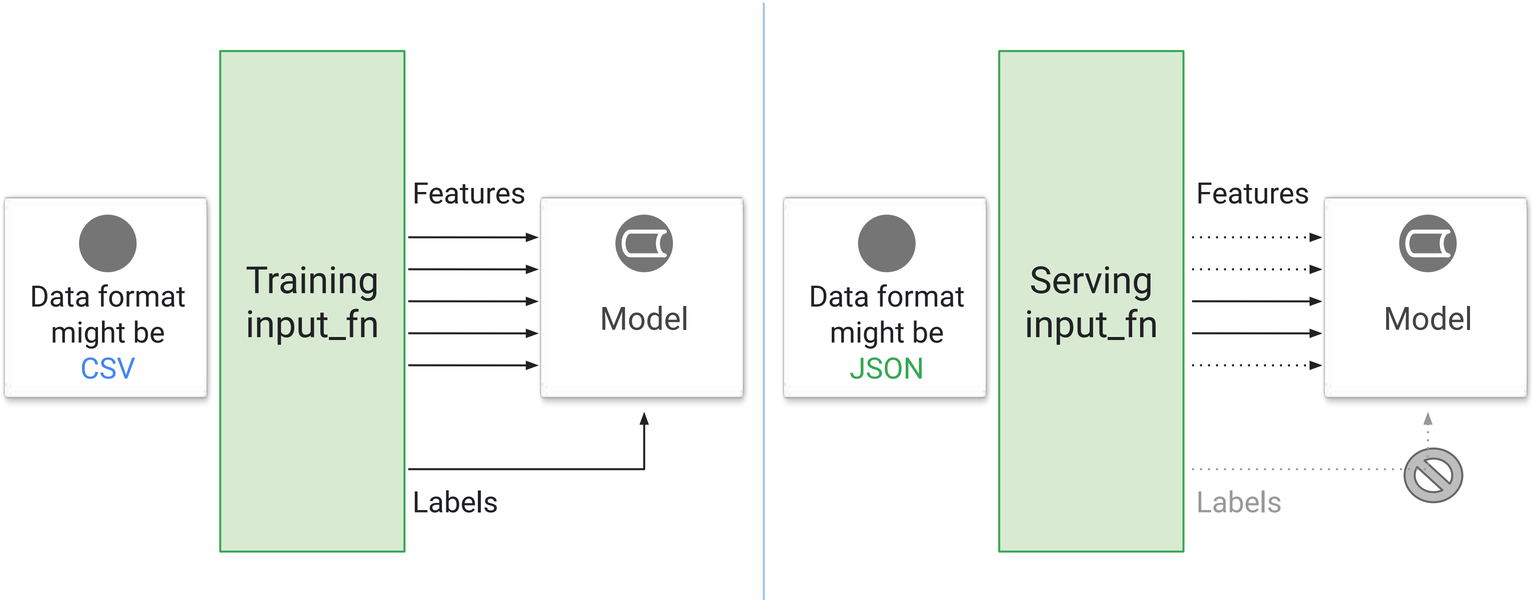
Predicting baby weight with BigQuery ML

In this lab, you will do the model training, evaluation, and prediction, all within BigQuery.

Cloud ML Engine makes deploying models and scaling the prediction infrastructure easy



You can't reuse the training input function for serving



1. The serving_input_fn specifies what the caller of the predict() method must provide

```
def serving_input_fn():
    feature_placeholders = {
      'pickuplon' : tf.placeholder(tf.float32, [None]),
      'pickuplat' : tf.placeholder(tf.float32, [None]),
      'dropofflat' : tf.placeholder(tf.float32, [None]),
      'dropofflon' : tf.placeholder(tf.float32, [None]),
      'passengers' : tf.placeholder(tf.float32, [None]),
    }
    features = {
        key: tf.expand_dims(tensor, -1)
        for key, tensor in feature_placeholders.items()
    }
    return tf.estimator.export.ServingInputReceiver(features,
                                                    feature_placeholders)
```

2. Deploy a trained model to GCP

```
MODEL_NAME="taxifare"
MODEL_VERSION="v1"
MODEL_LOCATION="gs://${BUCKET}/taxifare/smallinput/taxi_trained/export/exporter
/.../"

gcloud ml-engine models create ${MODEL_NAME} --regions $REGION
gcloud ml-engine versions create ${MODEL_VERSION} --model ${MODEL_NAME}
--origin ${MODEL_LOCATION}
```

Could also be a locally trained model.

3. Client code can make REST calls

```
credentials = GoogleCredentials.get_application_default()
api = discovery.build('ml', 'v1', credentials=credentials,
discoveryServiceUrl='https://storage.googleapis.com/cloud-ml/discovery/ml_v1beta1_
discovery.json')
request_data = [
    {'pickup_longitude': -73.885262,
     'pickup_latitude': 40.773008,
     'dropoff_longitude': -73.987232,
     'dropoff_latitude': 40.732403,
     'passenger_count': 2}]
parent = 'projects/%s/models/%s/versions/%s' % ('cloud-training-demos',
'taxifare', 'v1')
response = api.projects().predict(body={'instances': request_data},
name=parent).execute()
```

Lab

Deploying and Predicting with Cloud ML Engine

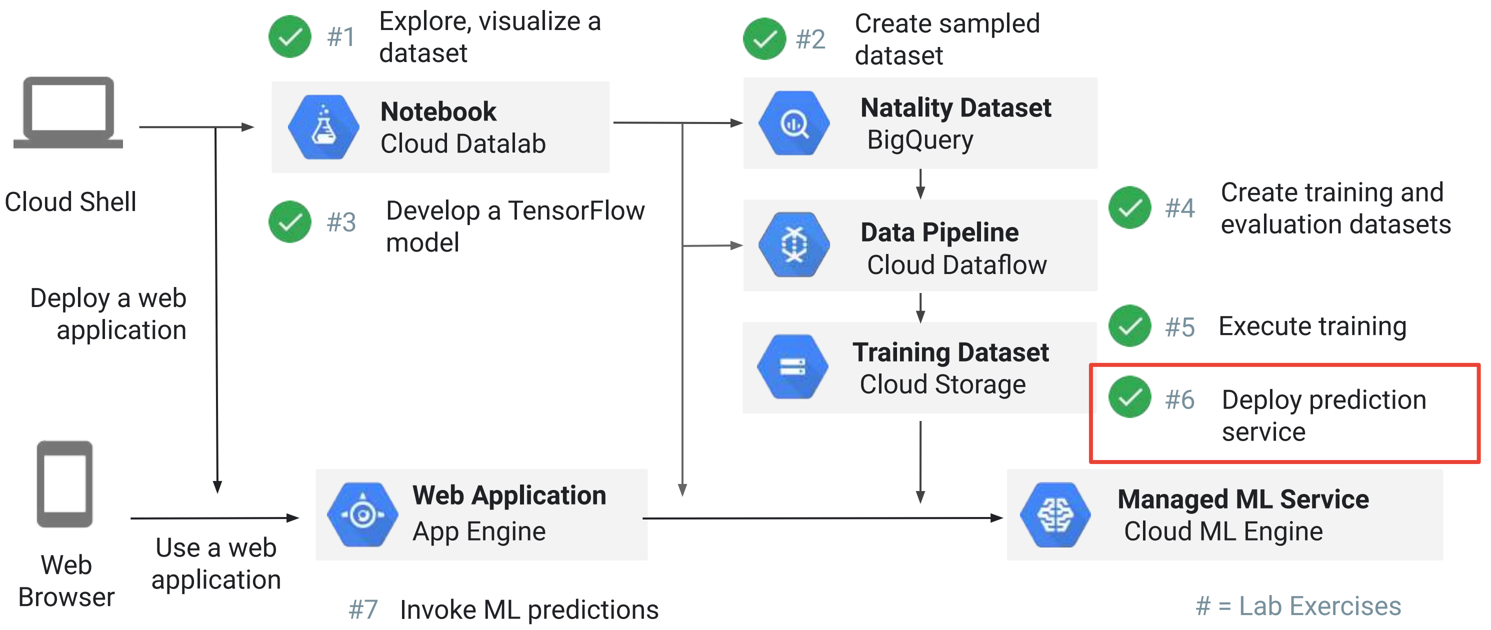
In this lab, you will deploy the trained model to act as a REST web service, and send a JSON request to the endpoint of the service to make it predict a baby's weight.

Lab Steps

1 Deploy a trained model to Cloud ML Engine.

2 Send a JSON request to model to get predictions.

The end-to-end process

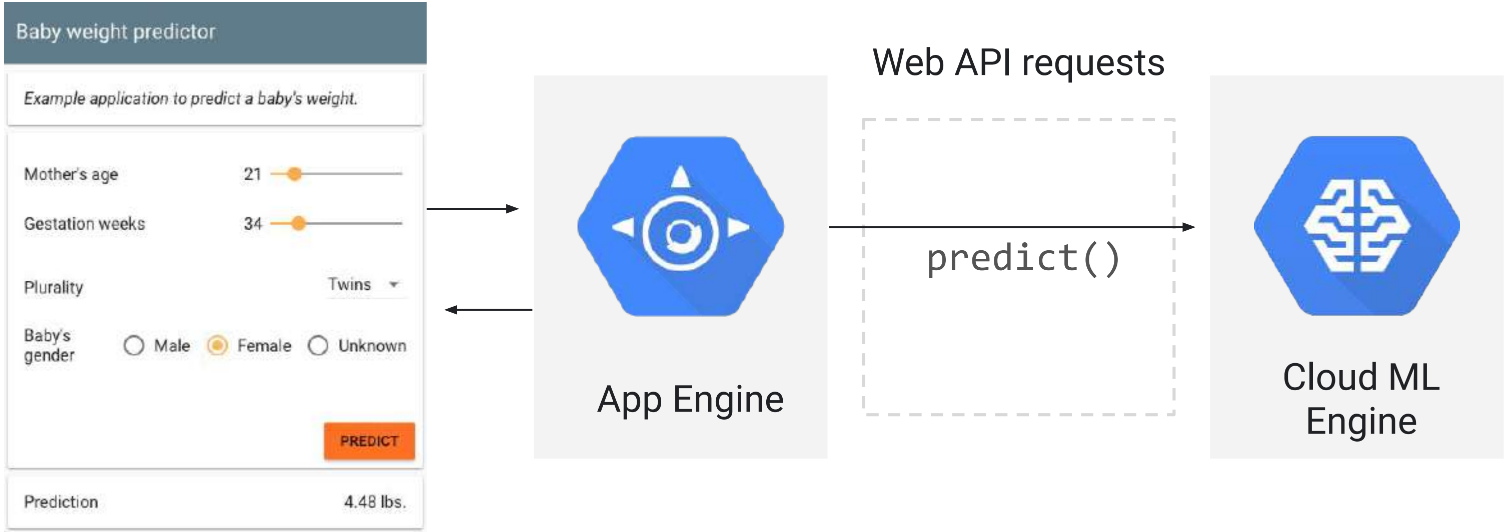


Lab

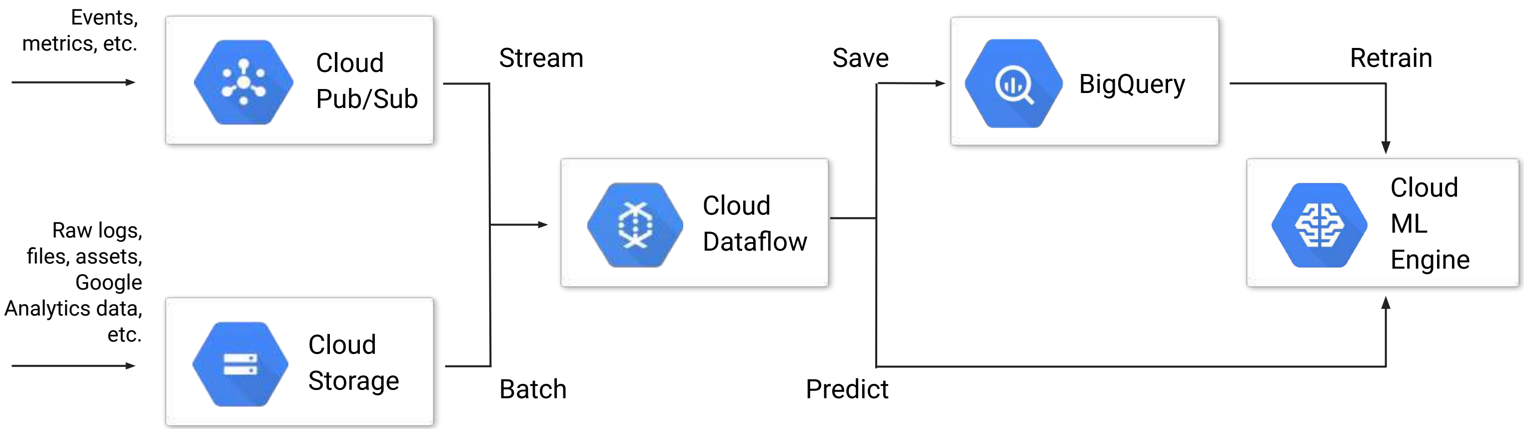
Building an App Engine app to serve ML predictions

In this lab, you will deploy a python Flask app as a App Engine web application, and use the App Engine app to post JSON data, based on user interface input, to the deployed ML model and get predictions.

Use App Engine to invoke ML predictions



You can also invoke the ML service from Cloud Dataflow and save predictions to BigQuery



cloud.google.com