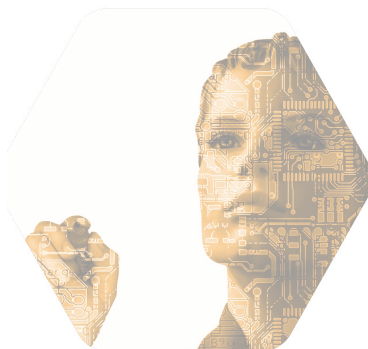Google Cloud

# Scaling TF models with Cloud ML Engine

Data Engineering on Google Cloud Platform
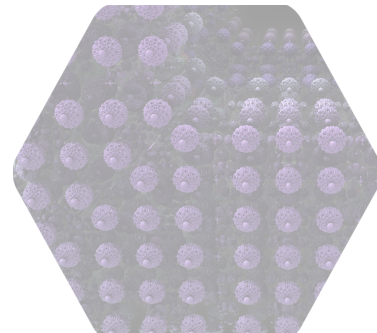
Google Cloud

# What's left? Ways to build effective ML



Big Data

Feature Engineering

Model Architectures

**Notes:**

https://pixabay.com/en/large-data-dataset-word-895563/ (cc0)
https://pixabay.com/en/fractal-complexity-render-3d-1232494/ (cc0)
https://pixabay.com/en/robot-artificial-intelligence-woman-507811/ (cc0)

Now that you know *how* to build ML, let's learn how to do it well in the rest of the course.
Ordered from easiest to most difficult

In the previous chapter, we refactored the model to make it easier to eim, but we haven't actually done anything to improve it.
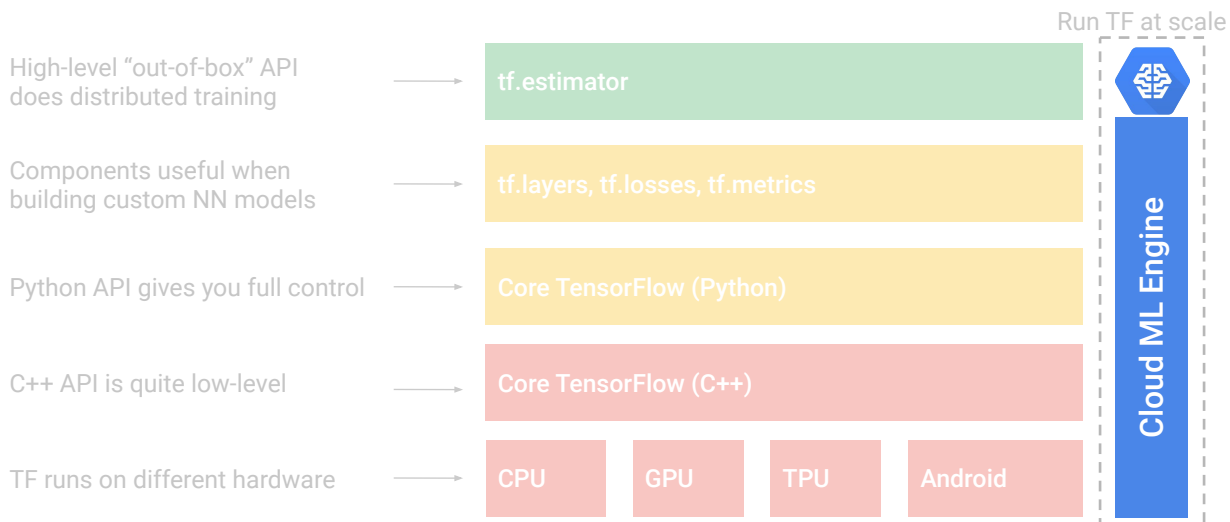
# Agenda

Why Cloud ML Engine?

Packing up a TensorFlow model + Lab

Kubeflow Pipelines

**Notes:**

Now to look at Cloud ML Engine which helps you run TF at scale

# Many Machine Learning frameworks can handle toy problems

**Notes:**

If you have data that fits in memory, pretty much any ML framework will work. R, Python, etc. have statistical packages that are often 3-4 lines of code that will work. Tf.learn when you use it on small in-memory datasets has an api that is like sci-kit learn and is very easy. But these are in-memory datasets. Once your datasets get larger (and they do get larger because of things like one-of-k encodings that you do on your columns), those packages won't work.

# As your data size increases, batching and distribution become important



Inputs → Train model → Model

**Notes:**

You will need to split your data into batches, and train. However, you you'll also need to distribute your training over many machines. This is not as simple as MapReduce where things are embarrassingly parallel. Things like gradient descent optimization are not embarrassingly parallel -- you will need parameter servers that form a shared memory that is updated during each epoch. Sometimes people think they can take a shortcut in order to keep the training simple by getting a bigger and bigger machine with lots of GPUs. They live to regret that decision, because at some point you will hit the limits of whatever single machine you are using. Scaling out is the answer, not scaling up. Another common shortcut that people take is to sample their data so that it is small enough to do ML with on the hardware they happen to have. They are leaving substantial performance gains on the table -- using all of the available data (and devising a plan to collect 10x the data that you currently have) is often the difference between ML that doesn't work and ML that appears magical.

# Input necessary transformations
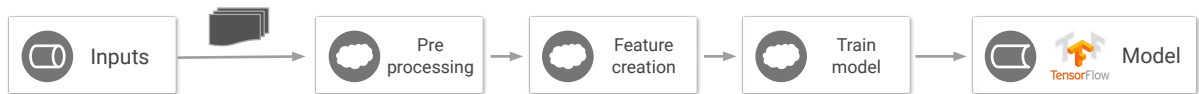
**Notes:**

Some of the most major improvements to Machine Learning happen when human insights come into the problem.

In ML, you bring human insights, what your experts know about the problem, in the form of new features. You will also need to preprocess your raw data -- scale it, encode it, etc. and these two things on a large dataset also need to be distributed and done on the cloud.

# Hyperparameter tuning might be nice

Hyper-parameter
tuning

Inputs → Pre processing → Feature creation → Train model → Model (TensorFlow)
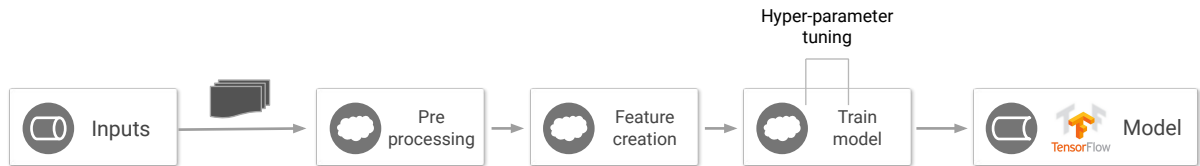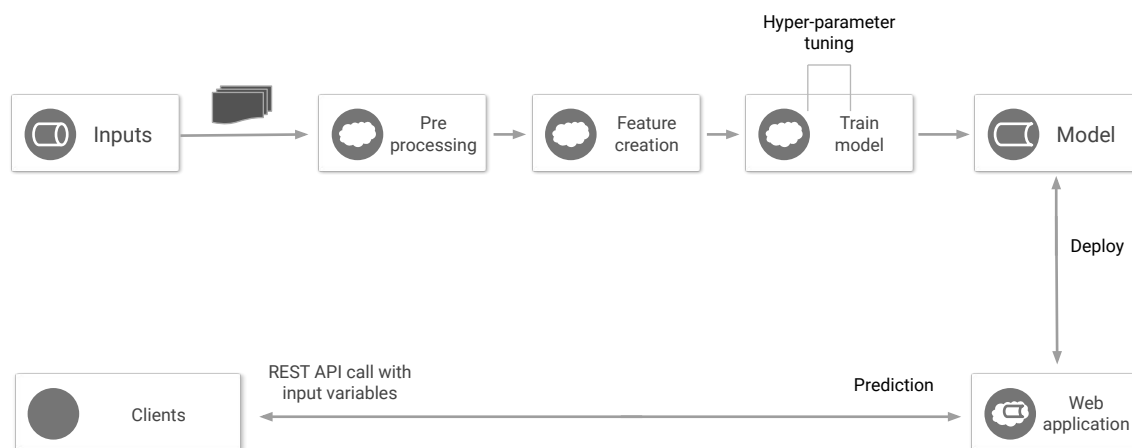
**Notes:**

When you do ML, you often pick a number of things arbitrarily -- the number of nodes, the embedding, the stride size on your convolution layer … and as your models get more complex, you start to wonder whether you picked the right things. You'll have to do some search in the hyperparameter space to see if there are better choices you could have made.

How many layers, how many nodes is the obvious hyper-parameter. But as we'll see in this course, it is good to take the preprocessing parameters (such as number of buckets) and make a hyperparameter out of it.

# Need to autoscale prediction code

**Notes:**

That is on the training side. You will want to take your trained model and deploy and at that point, the performance characteristic changes. Instead of thinking of how long it takes to train on your training dataset, you start to need to support N queries/second. That requires autoscaling the prediction code as necessary to support the users who need those predictions.

Do you want all your client code creating a DNNRegressor(dirname) and calling predict() on it? What if your model changes? Hyperparameters change? Number of inputs changes? Do you really want to expose this to your users? What if that client is not in Python?

Can invoke the API from pretty much any programming language. Can put the server on the cloud, and scale to beaucoup queries per second as needed.
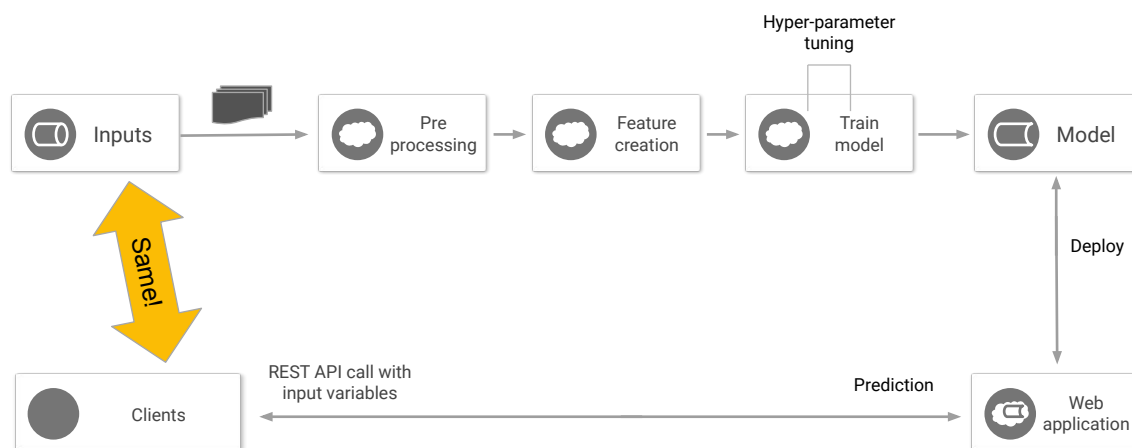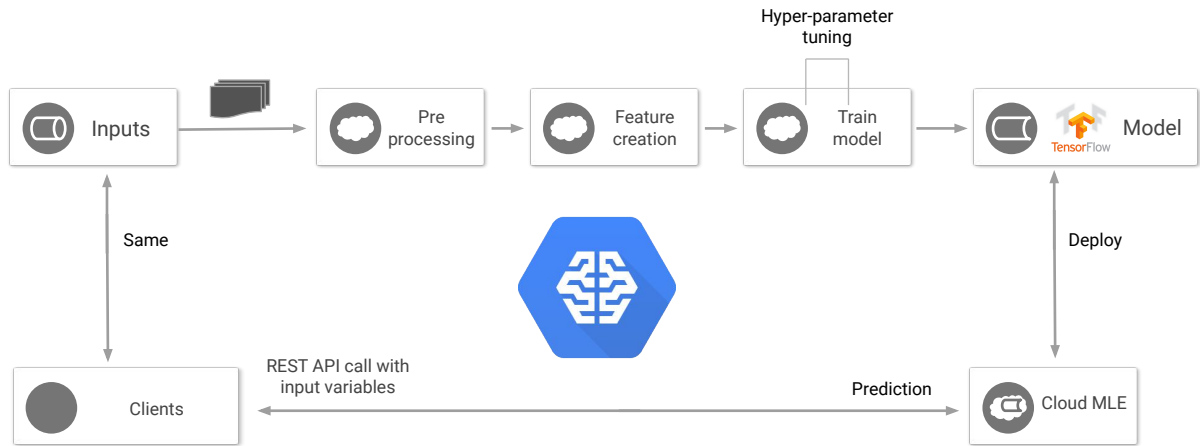
**Notes:**

Not as obvious … who will do the input transformations on behalf of the client code? You can't pass in the raw input variables to the trained model -- it expects scaled, transformed inputs!

You also have to worry about model changes -- when you do a bag-of-words, for example, with IBM=32, the embedding might change in the next model run because your input data is larger. Similarly, in scaling, min/max/stdev can all change. Doing the bookkeeping associated with preprocessing and feature crosses is painful and a major source of error. It is also near-impossible to find, so there are probably many ML models out there that have a "**training/serving skew**" (yes, this is a real thing, with a real jargon word for it, but it is rarely discussed because the majority of ML research papers are from college settings where routine model updates are not a concern.)

**Notes:**

Summary:

Repeatable:
Using TensorFlow would mean keeping track of all kinds of things such as the order of preprocessing operations, what scaling parameters were, etc. Cloud ML simplifies the bookkeeping, ensuring that the trained model is what you actually run; i.e., it helps you handle training/serving skew. It can be quite easy, for example, for the training pipeline to do something that the prediction pipeline doesn't do.
The key thing is that the input variables that clients are expected to provide is the same data as was provided to ML pipeline -- Cloud ML will keep track of all the preprocessing and feature engineering steps.
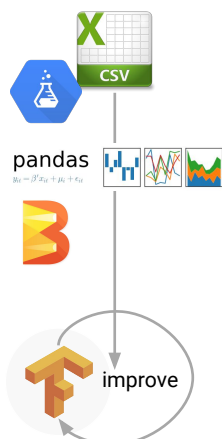
Scalable:
In training, Cloud ML will help you distribute preprocessing, training model (multiple times iteratively, due to hyper-parameter training) and deploy your trained model to the cloud. In prediction, the ML model is accessible via a REST API and it includes all the preprocessing, feature creation that you did, so that client code can supply simply the raw input variables (exactly what you collected out of the log file or sensor or database) and get back a prediction.

Cloud ML will also scale this (distribute it) so as to reach high # of queries per second. This is important -- you need high-quality execution both at training and at prediction time. While computation of a tensorflow model once is relatively cheap, the point of a ML model is often to do predictions for lots of requests (this is where TPUs come in).

Tuned:
You can use Cloud ML to do hyperparameter tuning and Cloud ML will remember these hyperparameters.

**In Datalab, start locally on sampled dataset**

**Notes:**

- The fail fast tool to analyze and learn from your data
- Interactively explore data
  - Define features with rich visualization support
  - Launch training and evaluation
  - ML lifecycle support
  - Combine code, results, visualizations, and documentation in notebook format
  - Share results with your team
  - Pick from a rich set of tutorials and samples to learn and get started with your project

# Then, scale it out to GCP using serverless technology



store

improve/
serverless/
hypertune

# Agenda

Packing up a TensorFlow model + Lab

# Training your model with Cloud ML Engine

| Use TensorFlow to create computation graph and training application | Package your trainer application | Configure and start a Cloud ML Engine job |
| --- | --- | --- |
| Step 1 | Step 2 | Step 3 |
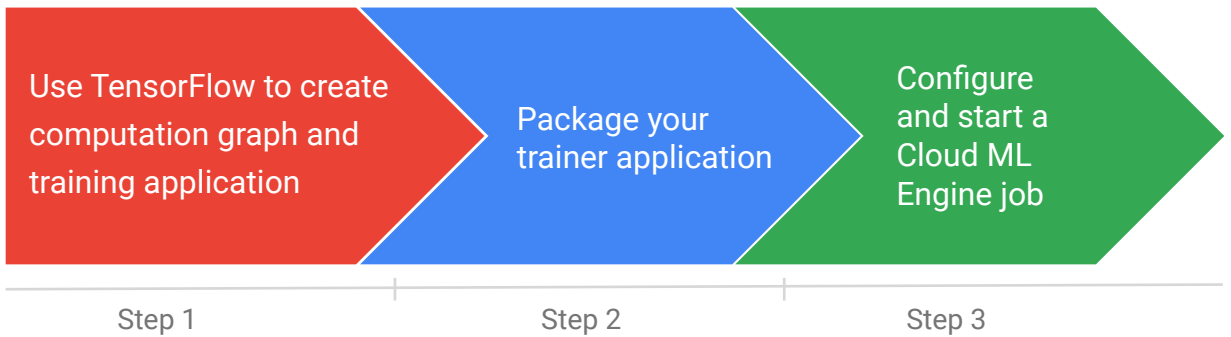
Before you begin though, be sure to (1) gather and prepare (clean, split, engineer features, preprocess features) training data, and (2) put your training data in an online source that Cloud MLE can access (e.e. Cloud storage)

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

**task.py**

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

**model.py**

```python
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)
```

Google Cloud

Training and Certification

**Notes:**

Task.py calls model.py

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

| | Example of the code in model.py (see previous module) |
|---|---|
| Training and evaluation input functions | ```CSV_COLUMNS = ...```<br>```def read_dataset(filename, mode, batch_size=512):```<br>```    ...``` |
| Feature columns | ```INPUT_COLUMNS = [```<br>```    tf.feature_column.numeric_column('pickuplon'),``` |
| Feature engineering | ```def add_more_features(feats):```<br>```  # will be covered in next chapter; for now, just a no-op```<br>```  return feats``` |
| Serving input function | ```def serving_input_fn():```<br>```    ...```<br>```    return tf.estimator.export.ServingInputReceiver(features,```<br>```feature_pholders)``` |
| Train and evaluate loop | ```def train_and_evaluate(args):```<br>```    ...```<br>```    tf.estimator.train_and_evaluate(estimator, train_spec, eval_spec)``` |

**Notes:**

Should be familiar.

# Package up TensorFlow model as 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

**Notes:**

Cloud MLE involves packaging up TensorFlow models. Not very different from how you need to create a web archive file (.war) to deploy a Java web application to Tomcat. Similarly, you put your TensorFlow code in a very specific packaging structure, and deploy it to the cloud.

The nice thing is that this is the standard way to create Python modules:
http://python-packaging.readthedocs.io/en/latest/minimal.html

# Verify that the model works as a Python package

```
export PYTHONPATH=${PYTHONPATH}:/somedir/taxifare
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
```

Then use the `gcloud` command to submit
the training job, either locally or to cloud

Proprietary + Confidential

```
gcloud ml-engine local train \
    --module-name=trainer.task \
    --package-path=/somedir/taxifare/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
```

**Notes:**

https://cloud.google.com/ml/pricing#ml_training_units_by_scale_tier  for explanation
of tiers and what the relative costs/performance are.
You can use estimator.train_and_evaluate() for distributed training

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

Model

Deploy

Cloud ML Engine Prediction pipeline

Clients

REST API call with input variables

Serving Input Fn

Pre processing

Model

# We can not reuse the training input function for serving

Features

Data format
might be CSV

Training
input_fn

Model

Labels

Features

Data format
might be JSON

Serving
input_fn

Model

Labels

- Serving input function is what will be used to parse what comes from the user at predict time.
- Serving input function might need to parse different input format
- Serving input function doesn't need labels
  - Features received from user might be a smaller set of variables than during training
    - E.g., train with hour-of-day, day-of-week as inputs to NN, but during prediction, can get these from system clock, so user doesn't need to supply them.

# The `serving_input_fn` specifies what the caller of the `predict()` method will have to provide

```python
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)
```

If we need to get the current time and add the day-of-the-week (Monday, Wed, etc.), this is where it would go.

# 2. Deploy trained model to GCP

```
MODEL_NAME="taxifare"          COULD ALSO BE A LOCALLY-TRAINED MODEL
MODEL_VERSION="v1"
MODEL_LOCATION="gs://${BUCKET}/taxifare/smallinput/taxi_trained/export/Servo/..
./"


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

# 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()
```

**Notes:**

This is serverless; Cloud ML will distribute the queries.

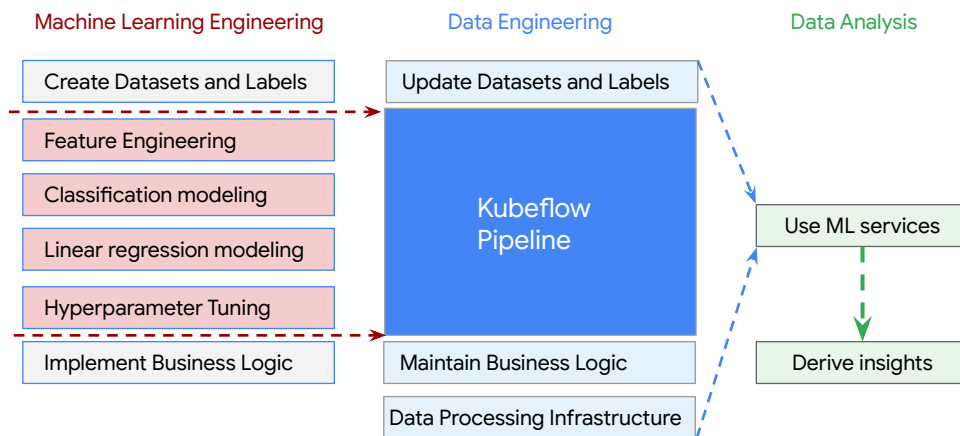# Lab 6: Scaling up ML using Cloud ML Engine

In this lab you will:

1. Package up the code
2. Find absolute paths to data
3. Run the Python module from the command line
4. Run locally using gcloud
5. Submit training job using gcloud
6. Deploy model
7. Prediction
8. Train on a larger dataset
9. 1-million row dataset



Cloud Machine Learning—Repeatable, Scalable, Tuned

Google Cloud
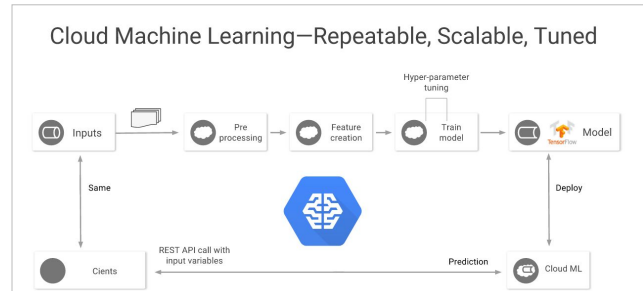
# Agenda

Kubeflow Pipelines

# Kubeflow Pipelines provides reusability and portability

**Machine Learning Engineering**

Create Datasets and Labels

Feature Engineering

Classification modeling

Linear regression modeling

Hyperparameter Tuning

Implement Business Logic

**Data Engineering**

Update Datasets and Labels

Kubeflow Pipeline

Maintain Business Logic

Data Processing Infrastructure

**Data Analysis**

Use ML services

Derive insights

Google Cloud

Training and Certification

Kubeflow Pipelines reusability helps separate the work which enables people to specialize.

In this example a Machine Learning Engineer can focus on Feature Engineering and Linear Regression Modeling and Hyperparameter Tuning.
Their solutions are bundled up into the Kubeflow Pipeline which can be used by a Data Engineer as part of a data engineering solution.
The solution appears as a service that is used by a Data Analyst to derive business insights.

# Kubeflow Pipelines package multi-step ML workflows
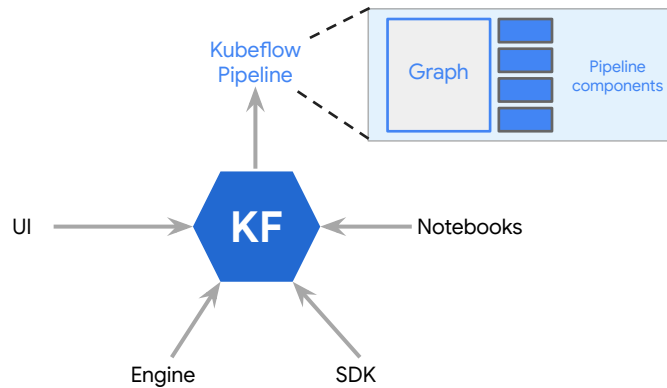
Kubeflow is an open project that packages machine learning code for Kubernetes.
Kubeflow PIpelines is a platform for composing, deploying, and managing end-to-end
machine learning workflows.

The main components of Kubeflow Pipelines includes:
A user interface (UI) for managing and tracking experiments, jobs, and runs.
An engine for scheduling multi-step ML workflows.
An SDK for defining and manipulating pipelines and components.
Notebooks for interacting with the system using the SDK.

These tools are used to define, experiment with, run, and share pipelines.
The pipeline consists of pipeline components, which are ML steps, that are
assembled into a graph that describes the execution pattern.

Key benefits are reusability and portability.

# Resources

- Cloud ML
  https://cloud.google.com/ml/docs/
- Blog on Big Data + ML
- https://cloud.google.com/blog/big-data/2017/07/new-hands-on-labs-for-scientific-data-processing-on-google-cloud-platform

cloud.google.com