Machine Learning with Apache Beam and TensorFlow

cloud.google.com/dataflow/docs/samples/molecules-walkthrough

This walkthrough shows you how to preprocess, train, and make predictions on a machine learning model, using Apache Beam, Google Dataflow, and TensorFlow.

To demonstrate these concepts, this walkthrough uses the Molecules code sample. Given molecular data as input, the Molecules code sample creates and trains a machine learning model to predict molecular energy.

Note: The Molecules code sample uses the TensorFlow Estimators API. If you are using the TensorFlow Keras API but want your project to more closely match this walkthrough, you can convert your Keras model to an Estimator.

Costs

This walkthrough potentially uses billable components of Google Cloud, including one or more of:

- Dataflow
- Cloud Storage
- AI Platform

Use the Pricing Calculator to generate a cost estimate based on your projected usage.

Overview

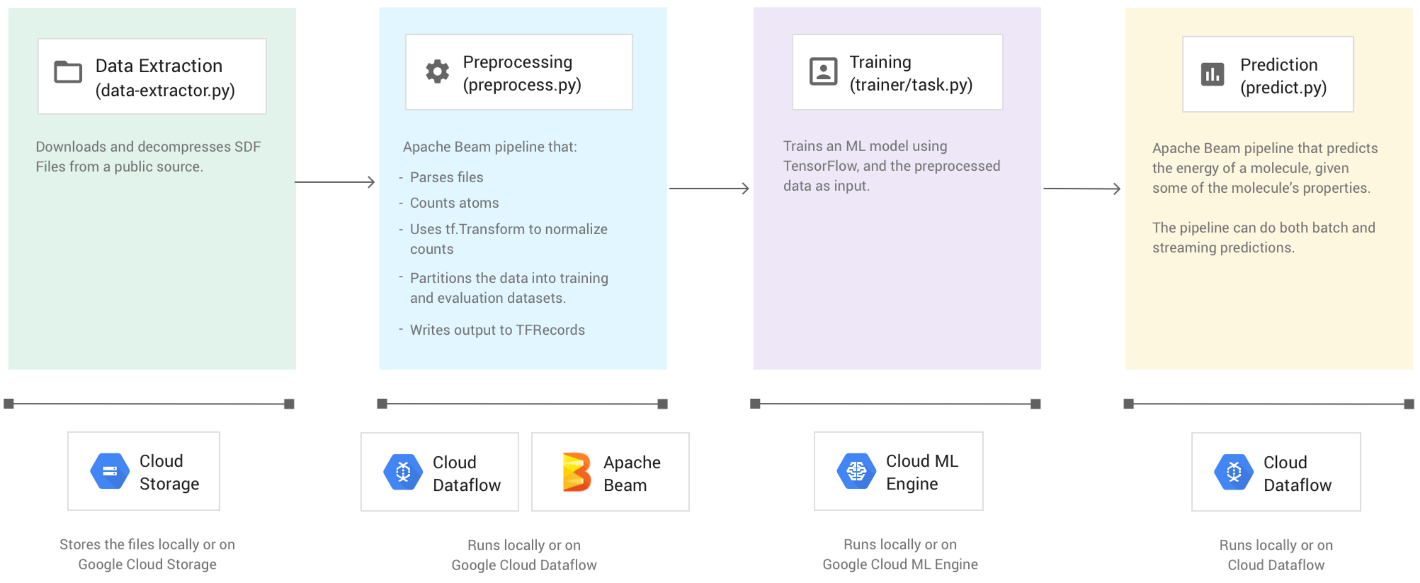
The Molecules code sample extracts files that contain molecular data and counts the number of carbon, hydrogen, oxygen, and nitrogen atoms are in each molecule. Then, the code normalizes the counts to values between 0 and 1, and feeds the values into a TensorFlow Deep Neural Network estimator. The Neural Network estimator trains a machine learning model to predict molecular energy.

The code sample consists of four phases:

1. Data extraction (`data-extractor.py`)
2. Preprocessing (`preprocess.py`)
3. Training (`trainer/task.py`)
4. Prediction (`predict.py`)

The sections below walk through the four phases, but this walkthrough focuses more on the phases that use Apache Beam and Dataflow: the preprocessing phase and the prediction phase. The preprocessing phase also uses the TensorFlow Transform library (commonly known as `tf.Transform`).

The following image shows the workflow of the Molecules code sample.


Data Extraction
(data-extractor.py)
Downloads and decompresses SDF Files from a public source.
Preprocessing
(preprocess.py)
Apache Beam pipeline that:
- Parses files
- Counts atoms
- Uses tf.Transform to normalize counts
- Partitions the data into training and evaluation datasets.
- Writes output to TFRecords
Training
(trainer/task.py)
Trains an ML model using TensorFlow, and the preprocessed data as input.
Prediction
(predict.py)
Apache Beam pipeline that predicts the energy of a molecule, given some of the molecule's properties.
The pipeline can do both batch and streaming predictions.
Cloud Storage
Stores the files locally or on Google Cloud Storage
Cloud Dataflow
Apache Beam
Runs locally or on Google Cloud Dataflow
Cloud ML Engine
Runs locally or on Google Cloud ML Engine
Cloud Dataflow
Runs locally or on Cloud Dataflow

Run the code sample

To set up your environment, follow the instructions in the README of the Molecules GitHub repository. Then, run the Molecules code sample using one of the provided wrapper scripts, `run-local` or `run-cloud`. These scripts automatically run all four phases of the code sample (extraction, preprocessing, training, and prediction).

Alternatively, you can run each phase manually using the commands provided in the sections in this document.

Run locally

To run the Molecules code sample locally, run the `run-local` wrapper script:

```
./run-local
```

The output logs show you when the script runs each of the four phases (data extraction, preprocessing, training, and prediction).

The `data-extractor.py` script has a required argument for the number of files. For ease of use, the `run-local` script and `run-cloud` scripts have a default of 5 files for this argument. Each file contains 25,000 molecules. Running the code sample should take approximately 3-7 minutes from start to finish. The time to run will vary depending on your computer's CPU.

Note: The `run-local` script includes optional arguments. See the README for information about how to use the optional arguments.

Run on Google Cloud

To run the Molecules code sample on Google Cloud, run the `run-cloud` wrapper script. All input files must be in Cloud Storage.

Set the `--work-dir` parameter to the Cloud Storage bucket:

```
./run-cloud --work-dir gs://<your-bucket-name>/cloudml-samples/molecules
```

Note: The `run-cloud` script includes optional arguments. See the README for information about how to use the optional arguments.

Phase 1: Data extraction

Source code: `data-extractor.py`

The first step is to extract the input data. The `data-extractor.py` file extracts and decompresses the specified SDF files. In later steps, the example preprocesses these files and uses the data to train and evaluate the machine learning model. The file extracts the SDF files from the public source and stores them in a subdirectory inside the specified working directory. The default working directory (`--work-dir`) is `/tmp/cloudml-samples/molecules` .

Store the extracted files

Store the extracted data files locally:

```
python data-extractor.py --max-data-files 5
```

Or, store the extracted data files in a Cloud Storage location:

```
WORK_DIR=gs://<your bucket name>/cloudml-samples/moleculespython data-extractor.py --work-dir
$WORK_DIR --max-data-files 5
```

Note: Storing files in Cloud Storage incurs charges on your Google Cloud project. See Cloud Storage pricing for more information.

Phase 2: Preprocessing

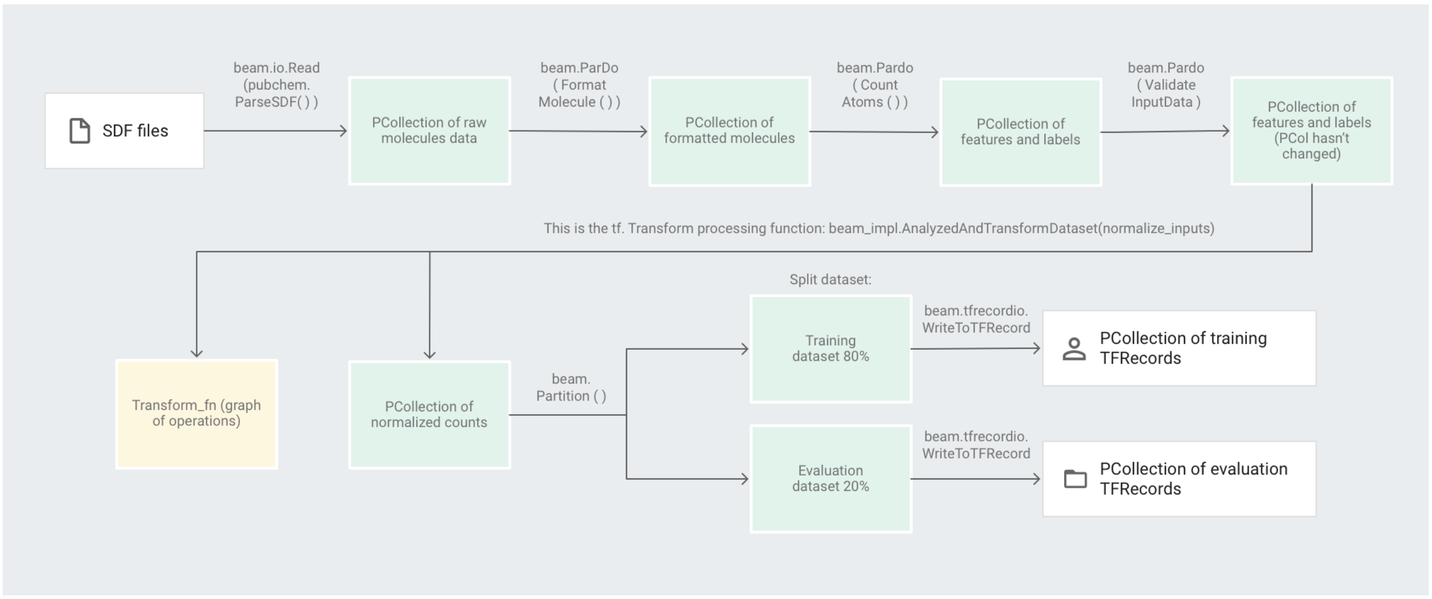
Source code: `preprocess.py`

The Molecules code sample uses an Apache Beam pipeline to preprocess the data. The pipeline performs the following preprocessing actions:

1. Reads and parses the extracted SDF files.
2. Counts the number of different atoms in each of the molecules in the files.
3. Normalizes the counts to values between 0 and 1 using `tf.Transform` .
4. Partitions the dataset into a training dataset and an evaluation dataset.
5. Writes the two datasets as `TFRecord` objects.

Apache Beam transforms can efficiently manipulate single elements at a time, but transforms that require a full pass of the dataset cannot easily be done with only Apache Beam and are better done using tf.Transform. Because of this, the code uses Apache Beam transforms to read and format the molecules, and to count the atoms in each molecule. The code then uses tf.Transform to find the global minimum and maximum counts in order to normalize the data.

The following image shows the steps in the pipeline.


SDF files
beam.io.Read (pubchem. ParseSDF( ))
PCollection of raw molecules data
beam.ParDo (Format Molecule ( ))
PCollection of formatted molecules
beam.Pardo (Count Atoms ( ))
PCollection of features and labels
beam.Pardo (Validate InputData)
PCollection of features and labels (PCol hasn't changed)
This is the tf. Transform processing function: beam_impl.AnalyzedAndTransformDataset(normalize_inputs)
Transform_fn (graph of operations)
PCollection of normalized counts
beam. Partition ( )
Split dataset:
Training dataset 80%
Evaluation dataset 20%
beam.tfrecordio. WriteToTFRecord
beam.tfrecordio. WriteToTFRecord
PCollection of training TFRecords
PCollection of evaluation TFRecords

Note: Some of the logic in the preprocessing pipeline (`preprocess.py`) is shared with the pipeline responsible for making predictions (`predict.py`). To avoid code duplication, this shared logic is located in the `pubchem/pipeline.py` and `pubchem/sdf.py` files. Apache Beam's pipeline dependency management encourages placing imported files in a separate module.

Applying element-based transforms

The `preprocess.py` code creates an Apache Beam pipeline.

See the tensorflow_transform/beam/impl.py code.

```
# Build and run a Beam Pipeline
with beam.Pipeline(options=beam_options) as p, \
     beam_impl.Context(temp_dir=tft_temp_dir):
```

Next, the code applies a `feature_extraction` transform to the pipeline.

```
# Transform and validate the input data matches the input schema
dataset = (
    p
    | 'Feature extraction' >> feature_extraction
```

The pipeline uses `SimpleFeatureExtraction` as its `feature_extraction` transform.

molecules/preprocess.py
View on GitHub Feedback

```
pubchem.SimpleFeatureExtraction(pubchem.ParseSDF(data_files_pattern)),
```

The `SimpleFeatureExtraction` transform, defined in `pubchem/pipeline.py` , contains a series of transforms that manipulate all elements independently. First, the code parses the molecules from the source file, then formats the molecules to a dictionary of molecule properties, and finally, counts the atoms in the molecule. These counts are the features (inputs) for the machine learning model.

molecules/pubchem/pipeline.py
View on GitHub Feedback

```
class SimpleFeatureExtraction(beam.PTransform):


"""The feature extraction (element-wise transformations).

  We create a `PTransform` class. This `PTransform` is a bundle of
  transformations that can be applied to any other pipeline as a step.

  We'll extract all the raw features here. Due to the nature of `PTransform`s,
  we can only do element-wise transformations here. Anything that requires a
  full-pass of the data (such as feature scaling) has to be done with
  tf.Transform.
  """
  def __init__(self, source):
    super(SimpleFeatureExtraction, self).__init__()
    self.source =

 source


  def expand(self, p):
    # Return the preprocessing pipeline. In this case we're reading the PubChem
    # files, but the source could be any Apache Beam source.
    return (p
        | 'Read raw molecules' >> self.source
        | 'Format molecule' >> beam.ParDo(FormatMolecule())
        | 'Count atoms' >> beam.ParDo(CountAtoms())
    )
```

The read transform `beam.io.Read(pubchem.ParseSDF(data_files_pattern))` reads SDF files from a custom source.

The custom source, called `ParseSDF`, is defined in `pubchem/pipeline.py`. `ParseSDF` extends `FileBasedSource` and implements the `read_records` function that opens the extracted SDF files.

When you run the Molecules code sample on Google Cloud, multiple workers (VMs) can simultaneously read the files. To ensure that no two workers read the same content in the files, each file uses a `range_tracker`.

The pipeline groups the raw data into sections of relevant information needed for the next steps. Each section in the parsed SDF file is stored in a dictionary (see `pipeline/sdf.py`), where the keys are the section names and the values are the raw line contents of the corresponding section.

Note: The Molecules code sample extracts only a few of the molecules' features. To see a more complex feature extraction, see this project.

The code applies `beam.ParDo(FormatMolecule())` to the pipeline. The `ParDo` applies the `DoFn` named `FormatMolecule` to each molecule. `FormatMolecule` yields a dictionary of formatted molecules. The following snippet is an example of an element in the output PCollection:

```
{
  'atoms': [
    {
      'atom_atom_mapping_number': 0,
      'atom_stereo_parity': 0,
      'atom_symbol': u'O',
      'charge': 0,
      'exact_change_flag': 0,
      'h0_designator': 0,
      'hydrogen_count': 0,
      'inversion_retention': 0,
      'mass_difference': 0,
      'stereo_care_box': 0,
      'valence': 0,
      'x': -0.0782,
      'y': -1.5651,
      'z': 1.3894,
    },
    ...
  ],
  'bonds': [
    {
      'bond_stereo': 0,
      'bond_topology': 0,
      'bond_type': 1,
      'first_atom_number': 1,
      'reacting_center_status': 0,
      'second_atom_number': 5,
    },
    ...
  ],
  '<PUBCHEM_COMPOUND_CID>': ['3\n'],
  ...
  '<PUBCHEM_MMFF94_ENERGY>': ['19.4085\n'],
  ...
}
```

Then, the code applies `beam.ParDo(CountAtoms())` to the pipeline. The `DoFn` `CountAtoms` sums the number of carbon, hydrogen, nitrogen, and oxygen atoms each molecule has. `CountAtoms` outputs a PCollection of features and labels. Here is an example of an element in the output PCollection:

```
{
  'ID': 3,
  'TotalC': 7,
  'TotalH': 8,
  'TotalO': 4,
  'TotalN': 0,
  'Energy': 19.4085,
}
```

The pipeline then validates the inputs. The `ValidateInputData` `DoFn` validates that every element matches the metadata given in the `input_schema`. This validation ensures that the data is in the correct format when it's fed into TensorFlow.

molecules/preprocess.py
View on GitHub Feedback

```
| 'Validate inputs' >> beam.ParDo(ValidateInputData(
    input_feature_spec)))
```

Applying full-pass transforms

The Molecules code sample uses a Deep Neural Network Regressor to make predictions. The general recommendation is to normalize the inputs before feeding them into the ML model. The pipeline uses tf.Transform to normalize the counts of each atom to values between 0 and 1. To read more about normalizing inputs, see feature scaling.

Normalizing the values requires a full pass through the dataset, recording the minimum and maximum values. The code uses tf.Transform to go through the entire dataset and apply full-pass transforms.

To use tf.Transform, the code must provide a function that contains the logic of the transform to perform on the dataset. In `preprocess.py`, the code uses the `AnalyzeAndTransformDataset` transform provided by tf.Transform. Learn more about how to use tf.Transform.

molecules/preprocess.py
View on GitHub Feedback

```
# Apply the tf.Transform preprocessing_fn
input_metadata = dataset_metadata.DatasetMetadata(
    dataset_schema.from_feature_spec(input_feature_spec))dataset_and_metadata, transform_fn = (
    (dataset, input_metadata)
    | 'Feature scaling' >> beam_impl.AnalyzeAndTransformDataset(
        feature_scaling))
dataset, metadata = dataset_and_metadata
```

In `preprocess.py`, the `feature_scaling` function used is `normalize_inputs`, which is defined in `pubchem/pipeline.py`. The function uses the tf.Transform function `scale_to_0_1` to normalize the counts to values between 0 and 1.

molecules/pubchem/pipeline.py
View on GitHub Feedback

```
def normalize_inputs(inputs):


"""Preprocessing function for tf.Transform (full-pass transformations).

  Here we will do any preprocessing that requires a full-pass of the dataset.
  It takes as inputs the preprocessed data from the `PTransform` we specify, in
  this case `SimpleFeatureExtraction`.

  Common operations might be scaling values to 0-1, getting the minimum or
  maximum value of a certain field, creating a vocabulary for a string field.

  There are two main types of transformations supported by tf.Transform, for
  more information, check the following modules:
    - analyzers: tensorflow_transform.analyzers.py
    - mappers:   tensorflow_transform.mappers.py

  Any transformation done in tf.Transform will be embedded into the TensorFlow
  model itself.
  """
  return {
      # Scale the input features for normalization
      'NormalizedC': tft.scale_to_0_1(inputs['TotalC']),
      'NormalizedH': tft.scale_to_0_1(inputs['TotalH']),
      'NormalizedO': tft.scale_to_0_1(inputs['TotalO']),
      'NormalizedN': tft.scale_to_0_1(inputs['TotalN']),
      # Do not scale the label since we want the absolute number for prediction
      'Energy': inputs['Energy'],
  }
```

Normalizing the data manually is possible, but if the dataset is large, it's faster to use Dataflow. Using Dataflow allows the pipeline to run on multiple workers (VMs) as necessary.

Partitioning the dataset

Next, the `preprocess.py` pipeline partitions the single dataset into two datasets. It allocates approximately 80% of the data to be used as training data, and approximately 20% of the data to be used as evaluation data.

molecules/preprocess.py
View on GitHub Feedback

```
# Split the dataset into a training set and an evaluation set
assert 0 < eval_percent < 100, 'eval_percent must in the range (0-100)'
train_dataset, eval_dataset = (
    dataset
    | 'Split dataset' >> beam.Partition(
        lambda elem, _: int(random.uniform(0, 100) < eval_percent), 2))
```

Writing the output

Finally, the `preprocess.py` pipeline writes the two datasets (training and evaluation) using the `WriteToTFRecord` transform.

molecules/preprocess.py

View on GitHub Feedback

```
# Write the datasets as TFRecords
coder = example_proto_coder.ExampleProtoCoder(metadata.schema)train_dataset_prefix =
os.path.join(train_dataset_dir, 'part')
_ = (
    train_dataset
    | 'Write train dataset' >> tfrecordio.WriteToTFRecord(
        train_dataset_prefix, coder))eval_dataset_prefix = os.path.join(eval_dataset_dir, 'part')
_ = (
    eval_dataset
    | 'Write eval dataset' >> tfrecordio.WriteToTFRecord(
        eval_dataset_prefix, coder))
# Write the transform_fn
_ = (
    transform_fn
    | 'Write transformFn' >> transform_fn_io.WriteTransformFn(work_dir))
```

Run the preprocessing pipeline

Run the preprocessing pipeline locally:

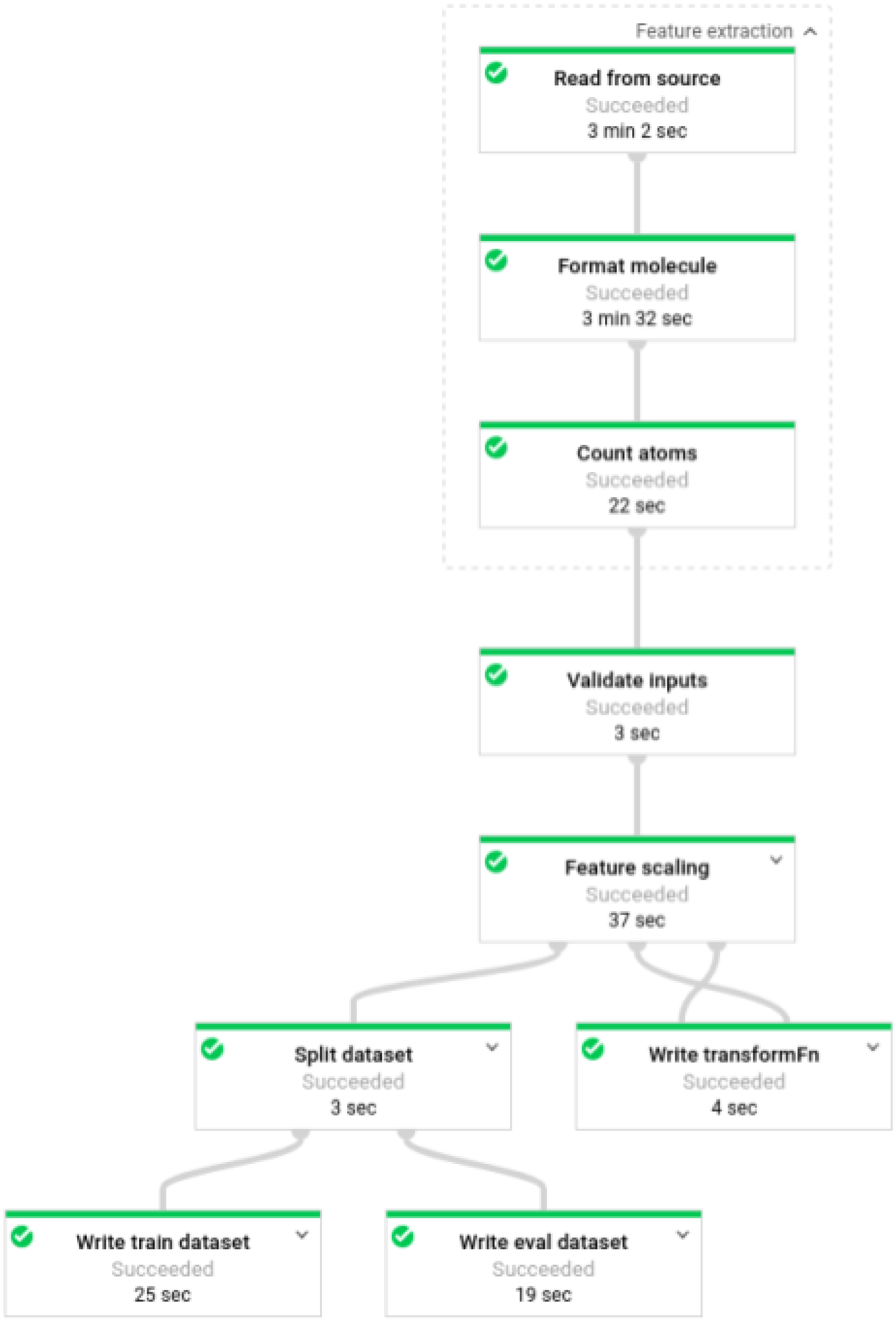
```
python preprocess.py
```

Or, run the preprocessing pipeline on Dataflow:

```
PROJECT=$(gcloud config get-value project)
WORK_DIR=gs://<your bucket name>/cloudml-samples/molecules
python preprocess.py \
  --project $PROJECT \
  --runner DataflowRunner \
  --temp_location $WORK_DIR/beam-temp \
  --setup_file ./setup.py \
  --work-dir $WORK_DIR
```

Note: Running Dataflow pipelines incurs charges on your Google Cloud project. See Dataflow pricing for more information.

After the pipeline is running, you can view the pipeline’s progress in the Dataflow Monitoring Interface:



Phase 3: Training

Source code: `trainer/task.py`

Recall that at the end of the preprocessing phase, the code split the data into two datasets (training and evaluation).

The sample uses TensorFlow to train the machine learning model. The `trainer/task.py` file in the Molecules code sample contains the code for training the model. The main function of `trainer/task.py` loads the data that was processed in the preprocessing phase.

The Estimator uses the training dataset to train the model, and then uses the evaluation dataset to verify that the model accurately predicts molecular energy given some of the molecule’s properties.

You can view the training job details in TensorBoard, by running:

```
tensorboard --logdir $WORK_DIR/model
```

Access the results in your browser at `localhost:6006`.

Learn more about training an ML model.

Train the model

Train the model locally:

```
python trainer/task.

py


# To get the path of the trained model
EXPORT_DIR=/tmp/cloudml-samples/molecules/model/export/final
MODEL_DIR=$(ls -d -1 $EXPORT_DIR/* | sort -r | head -n 1)
```

Or, train the model on AI Platform:

First select a supported region to run the training job

```
gcloud config set compute/region $REGION
```

Launch the training job

```
JOB="cloudml_samples_molecules_$(date +%Y%m%d_%H%M%S)"
BUCKET=gs://<your bucket name>
WORK_DIR=$BUCKET/cloudml-samples/molecules
gcloud ai-platform jobs submit training $JOB \
  --module-name trainer.task \
  --package-path trainer \
  --staging-bucket $BUCKET \
  --runtime-version 1.13 \
  --stream-logs \
  -- \
  --work-

dir $WORK_DIR


# To get the path of the trained model:
EXPORT_DIR=$WORK_DIR/model/export
MODEL_DIR=$(gsutil ls -d $EXPORT_DIR/* | sort -r | head -n 1)
```

Note: Making predictions using AI Platform incurs charges on your Google Cloud project. See AI Platform pricing for more information.

Phase 4: Prediction

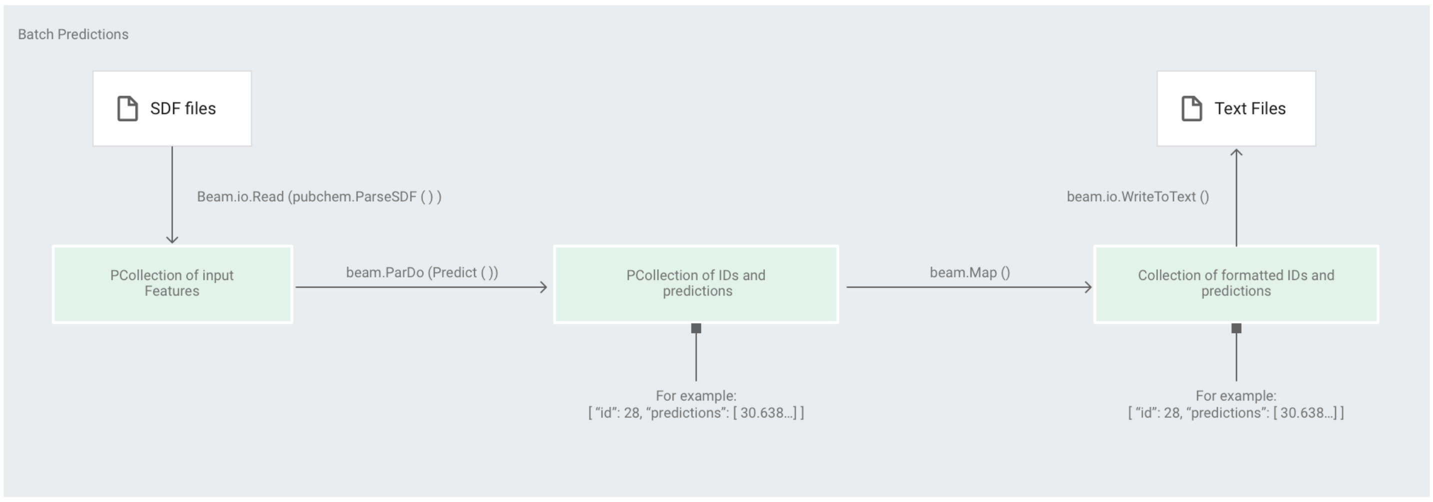
Source code: `predict.py`

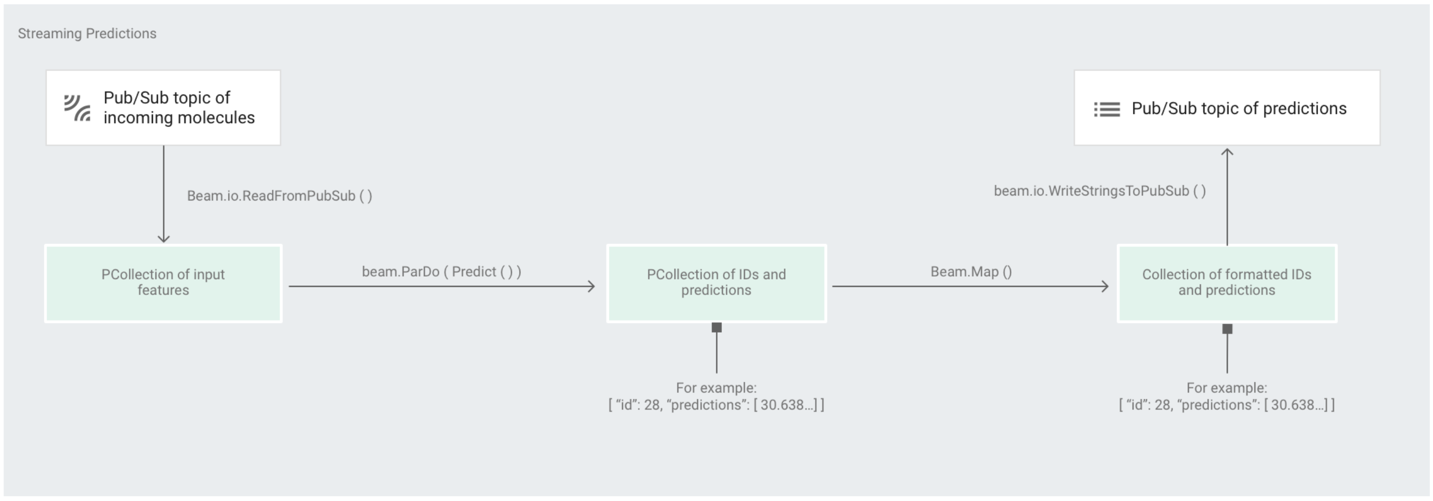
After the Estimator trains the model, you can provide the model with inputs and it will make predictions. In the Molecules code sample, the pipeline in `predict.py` is responsible for making predictions. The pipeline can act as either a batch pipeline or a streaming pipeline.

The code for the pipeline is the same for batch and streaming, except for the source and sink interactions. If the pipeline runs in batch mode, it reads the input files from the custom source and writes the output predictions as text files to the specified working directory. If the pipeline runs in streaming mode, it reads the input molecules, as they arrive, from a Pub/Sub topic. The pipeline writes the output predictions, as they are ready, to a different Pub/Sub topic.

```
if args.verb == 'batch':
  data_files_pattern = os.path.join(args.inputs_dir, '*.sdf')
  results_prefix = os.path.join(args.outputs_dir, 'part')
  source = pubchem.ParseSDF(data_files_pattern)
  sink = beam.io.WriteToText(results_prefix)
elif args.verb == 'stream':
  if not project:
    parser.print_usage()
    print('error: argument --project is required for streaming')
    sys.exit(1)  beam_options.view_as(StandardOptions).streaming = True
  source = beam.io.ReadFromPubSub(topic='projects/{}/topics/{}'.format(
      project, args.inputs_topic))
  sink = beam.io.WriteToPubSub(topic='projects/{}/topics/{}'.format(
      project, args.outputs_topic))
```

The following image shows the steps in the prediction pipelines (batch and streaming).


Batch Predictions
SDF files
Beam.io.Read (pubchem.ParseSDF ( ))
PCollection of input Features
beam.ParDo (Predict ( ))
PCollection of IDs and predictions
For example: ["id": 28, "predictions": [30.638...]]
beam.Map ()
Collection of formatted IDs and predictions
For example: ["id": 28, "predictions": [30.638...]]
beam.io.WriteToText ()
Text Files


Streaming Predictions
Pub/Sub topic of incoming molecules
Beam.io.ReadFromPubSub ( )
PCollection of input features
beam.ParDo (Predict ( ))
PCollection of IDs and predictions
For example: ["id": 28, "predictions": [30.638...]]
Beam.Map ()
Collection of formatted IDs and predictions
For example: ["id": 28, "predictions": [30.638...]]
beam.io.WriteStringsToPubSub ( )
Pub/Sub topic of predictions

In `predict.py`, the code defines the pipeline in the `run` function:

```
def run(model_dir, feature_extraction, sink, beam_options=None):
  print('Listening...')
  with beam.Pipeline(options=beam_options) as p:
    _ = (p
        | 'Feature extraction' >> feature_extraction
        | 'Predict' >> beam.ParDo(Predict(model_dir, 'ID'))
        | 'Format as JSON' >> beam.Map(json.dumps)
        | 'Write predictions' >> sink)
```

The code calls the `run` function with the following parameters:

```
run(
    args.model_dir,
    pubchem.SimpleFeatureExtraction(source),
    sink,
    beam_options)
```

First, the code passes the `pubchem.SimpleFeatureExtraction(source)` transform as the `feature_extraction` transform. This transform, which was also used in the preprocessing phase, is applied to the pipeline:

```
class SimpleFeatureExtraction(beam.PTransform):


"""The feature extraction (element-wise transformations).

  We create a `PTransform` class. This `PTransform` is a bundle of
  transformations that can be applied to any other pipeline as a step.

  We'll extract all the raw features here. Due to the nature of `PTransform`s,
  we can only do element-wise transformations here. Anything that requires a
  full-pass of the data (such as feature scaling) has to be done with
  tf.Transform.
  """
  def __init__(self, source):
    super(SimpleFeatureExtraction, self).__init__()
    self.source =

 source


  def expand(self, p):
    # Return the preprocessing pipeline. In this case we're reading the PubChem
    # files, but the source could be any Apache Beam source.
    return (p
        | 'Read raw molecules' >> self.source
        | 'Format molecule' >> beam.ParDo(FormatMolecule())
        | 'Count atoms' >> beam.ParDo(CountAtoms())
    )
```

The transform reads from the appropriate source based on the pipeline's execution mode (batch or streaming), formats the molecules, and counts the different atoms in each molecule.

Next, `beam.ParDo(Predict(…))` is applied to the pipeline that performs the prediction of the molecular energy. `Predict`, the `DoFn` that's passed, uses the given dictionary of input features (atom counts), to predict the molecular energy.

The next transform applied to the pipeline is `beam.Map(lambda result: json.dumps(result))`, which takes the prediction result dictionary and serializes it into a JSON string.

Finally, the output is written to the sink (either as text files to the working directory for batch mode, or as messages published to a Pub/Sub topic for streaming mode).

Batch predictions

Batch predictions are optimized for throughput rather than latency. Batch predictions work best if you're making many predictions and you can wait for all of them to finish before getting the results.

Run the prediction pipeline in batch mode

Run the batch prediction pipeline locally:

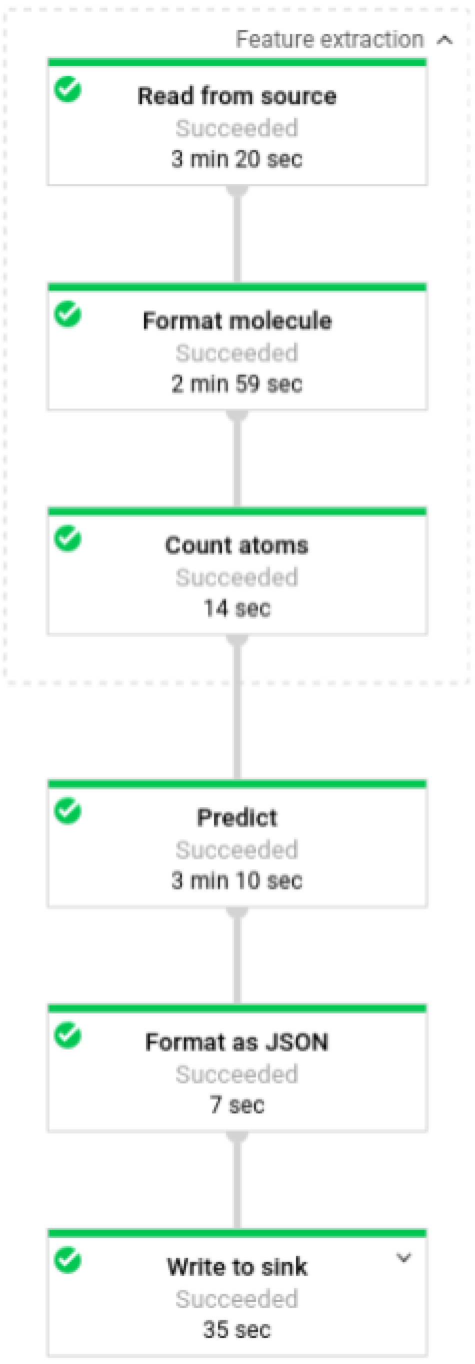
```
# For simplicity, we'll use the same files we used for training
python predict.py \
  --model-dir $MODEL_DIR \
  batch \
  --inputs-dir /tmp/cloudml-samples/molecules/data \
  --outputs-dir /tmp/cloudml-samples/molecules/predictions
```

Or, run the batch prediction pipeline on Dataflow:

```
# For simplicity, we'll use the same files we used for training
PROJECT=$(gcloud config get-value project)
WORK_DIR=gs://<your bucket name>/cloudml-samples/molecules
python predict.py \
  --work-dir $WORK_DIR \
  --model-dir $MODEL_DIR \
  batch \
  --project $PROJECT \
  --runner DataflowRunner \
  --temp_location $WORK_DIR/beam-temp \
  --setup_file ./setup.py \
  --inputs-dir $WORK_DIR/data \
  --outputs-dir $WORK_DIR/predictions
```

Note: Running Dataflow pipelines incurs charges on your Google Cloud project. See Dataflow pricing for more information.

After the pipeline is running, you can view the pipeline’s progress in the Dataflow Monitoring Interface:



Streaming predictions

Streaming predictions are optimized for latency rather than throughput. Streaming predictions work best if you're making sporadic predictions but want to get the results as soon as possible.

The prediction service (the streaming prediction pipeline) receives the input molecules from a Pub/Sub topic and publishes the output (predictions) to another Pub/Sub topic.

Create the inputs Pub/Sub topic:

```
gcloud pubsub topics create molecules-inputs
```

Create the outputs Pub/Sub topic:

```
gcloud pubsub topics create molecules-predictions
```

Run the streaming prediction pipeline locally:

```
# Run on terminal 1
PROJECT=$(gcloud config get-value project)
python predict.py \
  --model-dir $MODEL_DIR \
  stream \
  --project $PROJECT
  --inputs-topic molecules-inputs \
  --outputs-topic molecules-predictions
```

Or, run the streaming prediction pipeline on Dataflow:

```
# Run on terminal 1
PROJECT=$(gcloud config get-value project)
WORK_DIR=gs://<your bucket name>/cloudml-samples/molecules
python predict.py \
  --work-dir $WORK_DIR \
  --model-dir $MODEL_DIR \
  stream \
  --project $PROJECT
  --runner DataflowRunner \
  --temp_location $WORK_DIR/beam-temp \
  --setup_file ./setup.py \
  --inputs-topic molecules-inputs \
  --outputs-topic molecules-predictions
```

After you have the prediction service (the streaming prediction pipeline) running, you need to run a publisher to send molecules to the prediction service and a subscriber to listen for prediction results. The Molecules code sample provides publisher (`publisher.py`) and subscriber (`subscriber.py`) services.

The publisher parses SDF files from a directory and publishes them to the inputs topic. The subscriber listens for prediction results and logs them. For simplicity, this example uses the same SDF files used in the training phase.

Note: You need to run these as different processes concurrently, so you'll need to have a different terminal to run each command. Remember to activate the `virtualenv` on each terminal.

Run the subscriber:

```
# Run on terminal 2
python subscriber.py \
  --project $PROJECT \
  --topic molecules-predictions
```

Run the publisher:

```
# Run on terminal 3
python publisher.py \
  --project $PROJECT \
  --topic molecules-inputs \
  --inputs-dir $WORK_DIR/data
```

After the publisher starts parsing and publishing molecules, you'll start seeing predictions from the subscriber.

Clean up

After you've finished running the streaming predictions pipeline, stop your pipeline to prevent incurring charges.

What's Next

- See the Apache Beam documentation
- See the tf.Transform documentation
- See another code sample that uses Apache Beam and tf.Transform