Open in Colab

Ungraded Lab (Optional): ETL Pipelines and Batch Predictions with Apache Beam and Tensorflow

In this lab, you will create, train, evaluate, and make predictions on a model using Apache Beam and TensorFlow. In particular, you will train a model to predict the molecular energy based on the number of carbon, hydrogen, oxygen, and nitrogen atoms.

This lab is marked as optional because you will not be interacting with Beam-based systems directly in future exercises. Other courses of this specialization also use tools that abstract this layer. Nonetheless, it would be good to be familiar with it since it is used under the hood by TFX which is the main ML pipelines framework that you will use in other labs. Seeing how these systems work will let you explore other codebases that use this tool more freely and even make contributions or bug fixes as you see fit. If you don't know the basics of Beam yet, we encourage you to look at the Minimal Word Count example here for a quick start and use the Beam Programming Guide to look up concepts if needed.

The entire pipeline can be divided into four phases:

1. Data extraction
2. Preprocessing the data
3. Training the model
4. Doing predictions

You will focus particularly on Phase 2 (Preprocessing) and a bit of Phase 4 (Predictions) because these use Beam in its implementation.

Let's begin!

Note: This tutorial uses code, images, and discussion from this article. We highlighted a few key parts and updated some of the code to use more recent versions. Also, we focused on making the lab running locally. The original article linked above contain instructions on running it in GCP. Just take note that it will have associated costs depending on the resources you use.

Initial setup

You will first download the scripts that you will use in the lab.

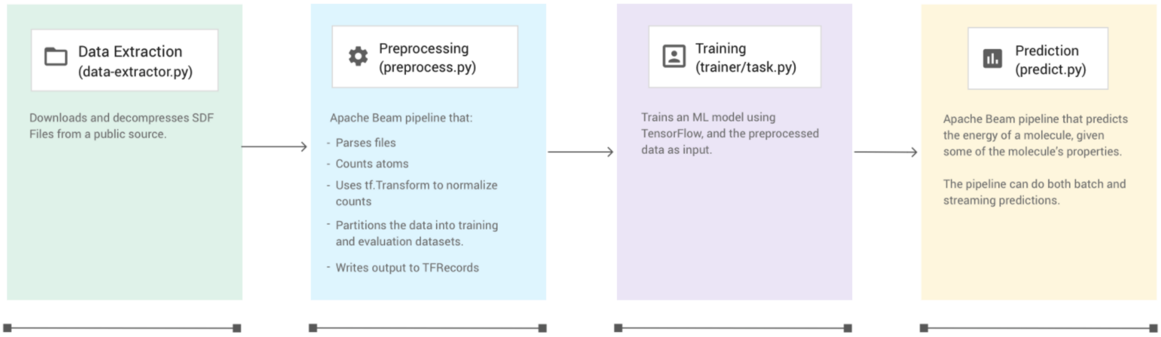
In [1]:

```
# Download the scripts
!wget https://github.com/https-deeplearning-ai/machine-learning-engineering-for-production-public

# Unzip the archive
!tar -xvzf molecules.tar.gz
```

```
--2021-09-24 13:35:37--  https://github.com/https-deeplearning-ai/machine-learning-engineering-fo
r-production-public/raw/main/course4/week2-ungraded-labs/C4_W2_Lab_4_ETL_Beam/data/molecules.tar.
gz
Resolving github.com (github.com)... 140.82.113.4
Connecting to github.com (github.com)|140.82.113.4|:443... connected.
HTTP request sent, awaiting response... 302 Found
Location: https://raw.githubusercontent.com/https-deeplearning-ai/machine-learning-engineering-fo
r-production-public/main/course4/week2-ungraded-labs/C4_W2_Lab_4_ETL_Beam/data/molecules.tar.gz
[following]
--2021-09-24 13:35:38--  https://raw.githubusercontent.com/https-deeplearning-ai/machine-learning
-engineering-for-production-public/main/course4/week2-ungraded-labs/C4_W2_Lab_4_ETL_Beam/data/mol
ecules.tar.gz
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.109.133, 185.199.111.1
33, 185.199.108.133, ...
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.199.109.133|:443... conne
cted.
HTTP request sent, awaiting response... 200 OK
Length: 10622 (10K) [application/octet-stream]
Saving to: ‘molecules.tar.gz’

molecules.tar.gz    100%[===================>]  10.37K  --.-KB/s    in 0s

2021-09-24 13:35:38 (77.0 MB/s) - ‘molecules.tar.gz’ saved [10622/10622]

molecules/
```

```
molecules/pubchem/
molecules/pubchem/pipeline.py
molecules/pubchem/sdf.py
molecules/pubchem/__init__.py
molecules/data-extractor.py
molecules/predict.py
molecules/requirements.txt
molecules/trainer/
molecules/trainer/task.py
molecules/trainer/__init__.py
molecules/preprocess.py
```

The `molecules` directory you downloaded mainly contain 4 scripts that encapsulate all phases of the workflow you will execute in this lab. It is summarized by the figure below:


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.

In addition, it also contains these additional files and directories:

- `pubchem` - subdirectory which contains common modules (i.e. `pipeline.py` and `sdf.py`) shared by the preprocessing and predicition phases. If you look at `preprocess.py` and `predict.py`, you can see the line `import as pubchem` at the top.
- `requirements.txt` - contains packages to install in this Colab. These are Apache Beam and Tensorflow Transform. These allows you to create Extract-Transform-Load (ETL) pipelines and preprocess data. Let's install them in the next cell.

In [ ]:

```
# Install required packages
!pip install -r ./molecules/requirements.txt
```

Note: In Google Colab, you need to restart the runtime at this point to finalize updating the packages you just installed. You can do so by clicking the `Restart Runtime` button at the end of the output cell above (after installation), or by selecting `Runtime > Restart Runtime` in the Menu bar. ***Please do not proceed to the next section without restarting.***

Next, you'll define the working directory to contain all the results you will generate in this exercise. After each phase, you can open the Colab file explorer on the left and look under the `results` directory to see the new files and directories generated.

In [1]:

```
# Define working directory
WORK_DIR = "results"
```

With that, you are now ready to execute the pipeline. As shown in the figure earlier, the logic is already implemented in the four main scripts. You will run them one by one and the next sections will discuss relevant detail and the outputs generated.

Phase 1: Data extraction

The first step is to extract the input data. The dataset is stored as `SDF` files and is extracted from the National Center for Biotechnology Information (FTP source). Chapter 6 of this document shows a more detailed description of the SDF file format.

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.

As you can see here, the complete set of files is huge and can easily exceed storage limits in Colab. For this exercise, you

In [2]:

```
# Print the help documentation. You can ignore references to GCP because you will be running ever
!python ./molecules/data-extractor.py --help
```

```
usage: data-extractor.py [-h] --work-dir WORK_DIR
                         [--data-sources DATA_SOURCES [DATA_SOURCES ...]]
                         [--filter-regex FILTER_REGEX] --max-data-files
                         MAX_DATA_FILES

optional arguments:
  -h, --help            show this help message and exit
  --work-dir WORK_DIR   Directory for staging and working files. This can be a
                        Google Cloud Storage path. (default: None)
  --data-sources DATA_SOURCES [DATA_SOURCES ...]
                        Data source location where SDF file(s) are stored.
                        Paths can be local, ftp://<path>, or gcs://<path>.
                        Examples: ftp://hostname/path
                        ftp://username:password@hostname/path (default: ['ftp:
                        //anonymous:guest@ftp.ncbi.nlm.nih.gov/pubchem/Compoun
                        d_3D/01_conf_per_cmpd/SDF'])
  --filter-regex FILTER_REGEX
                        Regular expression to filter which files to use. The
                        regular expression will be searched on the full
                        absolute path. Every match will be kept. (default:
                        \.sdf)
  --max-data-files MAX_DATA_FILES
                        Maximum number of data files for every file pattern
                        expansion. Set to -1 to use all files. (default: None)
```

In [3]:

```
# Run the data extractor
!python ./molecules/data-extractor.py --max-data-files 1 --work-dir={WORK_DIR}
```

```
Found 6227 files, using 1
Extracting data files...
Extracted results/data/00000001_00025000.sdf
```

You should now have a new folder in your work directory called `data`. This will contain the SDF file you downloaded.

In [4]:

```
# List working directory
!ls {WORK_DIR}
```

```
data
```

In the SDF Documentation linked earlier, it shows that one record is terminated by `$$$$`. You can use the command below to print the first one in the file. As you'll see, just one record is already pretty long. In the next phase, you'll feed these records in a pipeline that will transform these into a form that can be consumed by our model.

In [5]:

```
# Print one record
!sed '/$$$$/q' {WORK_DIR}/data/00000001_00025000.sdf
```

```
1
  -OEChem-09192103043D

 31 30  0     1  0  0  0  0  0999 V2000
    0.3387    0.9262    0.4600 O   0  0  0  0  0  0  0  0  0  0  0  0
    3.4786   -1.7069   -0.3119 O   0  5  0  0  0  0  0  0  0  0  0  0
    1.8428   -1.4073    1.2523 O   0  0  0  0  0  0  0  0  0  0  0  0
    0.4166    2.5213   -1.2091 O   0  0  0  0  0  0  0  0  0  0  0  0
   -2.2359   -0.7251    0.0270 N   0  3  0  0  0  0  0  0  0  0  0  0
   -0.7783   -1.1579    0.0914 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.1368   -0.0961   -0.5161 C   0  0  2  0  0  0  0  0  0  0  0  0
   -3.1119   -1.7972    0.6590 C   0  0  0  0  0  0  0  0  0  0  0  0
   -2.4103    0.5837    0.7840 C   0  0  0  0  0  0  0  0  0  0  0  0
   -2.6433   -0.5289   -1.4260 C   0  0  0  0  0  0  0  0  0  0  0  0
    1.4879   -0.6438   -0.9795 C   0  0  0  0  0  0  0  0  0  0  0  0
    2.3478   -1.3163    0.1002 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.4627    2.1935   -0.0312 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.6678    3.1549    1.1001 C   0  0  0  0  0  0  0  0  0  0  0  0
   -0.7073   -2.1051   -0.4563 H   0  0  0  0  0  0  0  0  0  0  0  0
   -0.5669   -1.3392    1.1503 H   0  0  0  0  0  0  0  0  0  0  0  0
   -0.3089    0.3239   -1.4193 H   0  0  0  0  0  0  0  0  0  0  0  0
   -2.9705   -2.7295    0.1044 H   0  0  0  0  0  0  0  0  0  0  0  0
   -2.8083   -1.9210    1.7028 H   0  0  0  0  0  0  0  0  0  0  0  0
   -4.1563   -1.4762    0.6031 H   0  0  0  0  0  0  0  0  0  0  0  0
   -2.0398    1.4170    0.1863 H   0  0  0  0  0  0  0  0  0  0  0  0
   -3.4837    0.7378    0.9384 H   0  0  0  0  0  0  0  0  0  0  0  0
   -1.9129    0.5071    1.7551 H   0  0  0  0  0  0  0  0  0  0  0  0
   -2.2450    0.4089   -1.8190 H   0  0  0  0  0  0  0  0  0  0  0  0
   -2.3000   -1.3879   -2.0100 H   0  0  0  0  0  0  0  0  0  0  0  0
   -3.7365   -0.4723   -1.4630 H   0  0  0  0  0  0  0  0  0  0  0  0
    1.3299   -1.3744   -1.7823 H   0  0  0  0  0  0  0  0  0  0  0  0
    2.0900    0.1756   -1.3923 H   0  0  0  0  0  0  0  0  0  0  0  0
   -0.1953    3.1280    1.7699 H   0  0  0  0  0  0  0  0  0  0  0  0
    0.7681    4.1684    0.7012 H   0  0  0  0  0  0  0  0  0  0  0  0
    1.5832    2.9010    1.6404 H   0  0  0  0  0  0  0  0  0  0  0  0
  1  7  1  0  0  0  0
  1 13  1  0  0  0  0
  2 12  1  0  0  0  0
  3 12  2  0  0  0  0
  4 13  2  0  0  0  0
  5  6  1  0  0  0  0
  5  8  1  0  0  0  0
  5  9  1  0  0  0  0
  5 10  1  0  0  0  0
  6  7  1  0  0  0  0
  6 15  1  0  0  0  0
  6 16  1  0  0  0  0
  7 11  1  0  0  0  0
  7 17  1  0  0  0  0
  8 18  1  0  0  0  0
  8 19  1  0  0  0  0
  8 20  1  0  0  0  0
  9 21  1  0  0  0  0
  9 22  1  0  0  0  0
  9 23  1  0  0  0  0
 10 24  1  0  0  0  0
 10 25  1  0  0  0  0
 10 26  1  0  0  0  0
 11 12  1  0  0  0  0
 11 27  1  0  0  0  0
 11 28  1  0  0  0  0
 13 14  1  0  0  0  0
 14 29  1  0  0  0  0
 14 30  1  0  0  0  0
 14 31  1  0  0  0  0
M  CHG  2   2  -1   5   1
M  END
> <PUBCHEM_COMPOUND_CID>
1

> <PUBCHEM_CONFORMER_RMSD>
0.6

> <PUBCHEM_CONFORMER_DIVERSEORDER>
2
43
65
46
25
35
57
19
53
42
34
37
41
50
30
14
13
10
56
28
55
22
17
44
52
48
21
7
61
16
66
36
12
32
40
1
24
29
63
47
9
39
60
5
```

```
20
31
62
51
4
59
67
8
18
11
33
26
6
27
64
15
58
54
23
38
3
45
49

> <PUBCHEM_MMFF94_PARTIAL_CHARGES>
14
1 -0.43
10 0.5
11 -0.11
12 0.91
13 0.66
14 0.06
2 -0.9
3 -0.9
4 -0.57
5 -1.01
6 0.5
7 0.28
8 0.5
9 0.5

> <PUBCHEM_EFFECTIVE_ROTOR_COUNT>
6

> <PUBCHEM_PHARMACOPHORE_FEATURES>
5
1 2 acceptor
1 3 acceptor
1 4 acceptor
1 5 cation
3 2 3 12 anion

> <PUBCHEM_HEAVY_ATOM_COUNT>
14

> <PUBCHEM_ATOM_DEF_STEREO_COUNT>
0

> <PUBCHEM_ATOM_UDEF_STEREO_COUNT>
1

> <PUBCHEM_BOND_DEF_STEREO_COUNT>
0

> <PUBCHEM_BOND_UDEF_STEREO_COUNT>
0

> <PUBCHEM_ISOTOPIC_ATOM_COUNT>
0

> <PUBCHEM_COMPONENT_COUNT>
1

> <PUBCHEM_CACTVS_TAUTO_COUNT>
1

> <PUBCHEM_CONFORMER_ID>
0000000100000002

> <PUBCHEM_MMFF94_ENERGY>
37.801

> <PUBCHEM_FEATURE_SELFOVERLAP>
25.427

> <PUBCHEM_SHAPE_FINGERPRINT>
1 1 17907859857256425260
```

```
13132413 78 18339935856441330356
16945 1 18127404777055172104
17841504 4 18338806718360982307
18410436 195 18412821378365737484
20361792 2 18413103948606886951
20645477 70 18193836175106948431
20653091 64 18337681930618404851
20711985 327 18273495675867710310
20711985 344 18052533275153547866
21041028 32 18342473533857807689
21061003 4 18410298003707379195
21524375 3 17335906067529293413
22112679 90 18128282041358100696
23419403 2 17977062926062270852
23552423 10 18193564595396549919
23557571 272 18127697028774774262
23598294 1 17832149325056171186
2748010 2 18339911658547624660
305870 269 17981602981145137625
31174 14 18192722361058170003
528862 383 18124596637411617035
7364860 26 18197783412505576099
81228 2 18051694343465326048
81539 233 17831573545929999781

> <PUBCHEM_SHAPE_MULTIPOLES>
259.66
4.28
3.04
1.21
1.75
2.55
0.16
-3.13
-0.22
-2.18
-0.56
0.21
0.17
0.09

> <PUBCHEM_SHAPE_SELFOVERLAP>
494.342

> <PUBCHEM_SHAPE_VOLUME>
160.7

> <PUBCHEM_COORDINATE_TYPE>
2
5
10
```

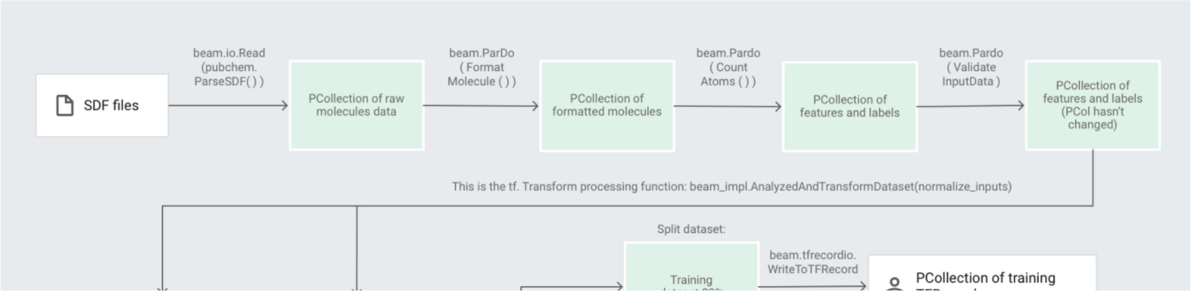
Phase 2: Preprocessing

The next script: `preprocess.py` 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)
Split dataset:
Training
beam.tfrecordio. WriteToTFRecord
PCollection of training

Run the preprocessing pipeline

You will run the script first and the following sections will discuss the relevant parts of this code. This will take around 6 minutes to run.

In [6]:

```
# Print help documentation
!python ./molecules/preprocess.py --help
```

```
usage: preprocess.py [-h] --work-dir WORK_DIR

optional arguments:
  -h, --help           show this help message and exit
  --work-dir WORK_DIR  Directory for staging and working files. This can be a
                       Google Cloud Storage path. (default: None)
```

In [7]:

```
# Run the preprocessing script
!python ./molecules/preprocess.py --work-dir={WORK_DIR}
```

```
2021-09-24 13:42:16.163587: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successfu
l NUMA node read from SysFS had negative value (-1), but there must be at least one NUMA node, so
returning NUMA node zero
2021-09-24 13:42:16.608316: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successfu
l NUMA node read from SysFS had negative value (-1), but there must be at least one NUMA node, so
returning NUMA node zero
2021-09-24 13:42:16.609275: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successfu
l NUMA node read from SysFS had negative value (-1), but there must be at least one NUMA node, so
returning NUMA node zero
2021-09-24 13:42:16.610603: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successfu
l NUMA node read from SysFS had negative value (-1), but there must be at least one NUMA node, so
returning NUMA node zero
2021-09-24 13:42:16.611476: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successfu
l NUMA node read from SysFS had negative value (-1), but there must be at least one NUMA node, so
returning NUMA node zero
2021-09-24 13:42:16.612296: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successfu
l NUMA node read from SysFS had negative value (-1), but there must be at least one NUMA node, so
returning NUMA node zero
2021-09-24 13:42:21.896424: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successfu
l NUMA node read from SysFS had negative value (-1), but there must be at least one NUMA node, so
returning NUMA node zero
2021-09-24 13:42:21.897359: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successfu
l NUMA node read from SysFS had negative value (-1), but there must be at least one NUMA node, so
returning NUMA node zero
2021-09-24 13:42:21.898091: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successfu
l NUMA node read from SysFS had negative value (-1), but there must be at least one NUMA node, so
returning NUMA node zero
2021-09-24 13:42:21.898932: W tensorflow/core/common_runtime/gpu/gpu_bfc_allocator.cc:39] Overrid
ing allow_growth setting because the TF_FORCE_GPU_ALLOW_GROWTH environment variable is set. Origi
nal config value was 0.
2021-09-24 13:42:21.899005: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1510] Created devi
ce /job:localhost/replica:0/task:0/device:GPU:0 with 10819 MB memory:  -> device: 0, name: Tesla
K80, pci bus id: 0000:00:04.0, compute capability: 3.7
2021-09-24 13:42:22.723867: I tensorflow/compiler/mlir/mlir_graph_optimization_pass.cc:185] None
of the MLIR Optimization Passes are enabled (registered 2)
WARNING:root:Make sure that locally built Python SDK docker image has Python 3.7 interpreter.
2021-09-24 13:42:24.251828: W tensorflow/python/util/util.cc:348] Sets are not currently consider
ed sequences, but this may change in the future, so consider avoiding using them.
WARNING:apache_beam.io.tfrecordio:Couldn't find python-snappy so the implementation of _TFRecordU
til._masked_crc32c is not as fast as it could be.
```

You should now have a few more outputs in your work directory. Most important are:

- **transform_fn** - the transformation graph describing how to transform SDF inputs to tensors
- **transformed_metadata** - contains the metadata describing the data types of the `tf.Transform` outputs
- **train-dataset** - contains the transformed training dataset
- **eval-dataset** - contains the transformed evaluation dataset
- **PreprocessData** - pickled file containing variables you will use in the training phase

In [8]:

```
# List working directory
!ls {WORK_DIR}
```

```
data          PreprocessData  train-dataset         transform_fn
eval-dataset  tft-temp        transformed_metadata
```

The training and evaluation datasets contain TFRecords and you can view them by running the helper function in the cells below.

In [9]:

```
from google.protobuf.json_format import MessageToDict

# Define a helper function to get individual examples
def get_records(dataset, num_records):
    '''Extracts records from the given dataset.
    Args:
        dataset (TFRecordDataset): dataset saved in the preprocessing step
        num_records (int): number of records to preview
    '''

    # initialize an empty list
    records = []

    # Use the `take()` method to specify how many records to get
    for tfrecord in dataset.take(num_records):

        # Get the numpy property of the tensor
        serialized_example = tfrecord.numpy()

        # Initialize a `tf.train.Example()` to read the serialized data
        example = tf.train.Example()

        # Read the example data (output is a protocol buffer message)
        example.ParseFromString(serialized_example)

        # convert the protocol bufffer message to a Python dictionary
        example_dict = (MessageToDict(example))

        # append to the records list
        records.append(example_dict)

    return records
```

In [10]:

```
import tensorflow as tf
from pprint import pprint

# Create TF Dataset from TFRecord of training set
train_data = tf.data.TFRecordDataset(f'{WORK_DIR}/train-dataset/part-00000-of-00001')

# Print two records
test_data = get_records(train_data, 2)

pprint(test_data)
```

```
[{'features': {'feature': {'Energy': {'floatList': {'value': [37.801]}},
                           'NormalizedC': {'floatList': {'value': [0.21428572]}},
                           'NormalizedH': {'floatList': {'value': [0.265625]}},
                           'NormalizedN': {'floatList': {'value': [0.083333336]}},
                           'NormalizedO': {'floatList': {'value': [0.1904762]}}}}},
 {'features': {'feature': {'Energy': {'floatList': {'value': [44.1107]}},
                           'NormalizedC': {'floatList': {'value': [0.21428572]}},
                           'NormalizedH': {'floatList': {'value': [0.28125]}},
                           'NormalizedN': {'floatList': {'value': [0.083333336]}},
                           'NormalizedO': {'floatList': {'value': [0.1904762]}}}}}]
```

Note: From the output cell above, you might concur that we'll need more than the atom counts to make better predictions. You'll notice that the counts are identical in both records but the `Energy` value is different. Thus, you cannot expect the model to have a low loss during the training phase later. For simplicity, we'll just use atom counts in this exercise but feel free to revise later to have more predictive features. You can share your findings in our Discourse community to discuss with other learners who are interested in the same problem.

The `PreprocessData` contains Python objects needed in the training phase such as:

- the filename patterns of the training and eval set directories
- spec file describing the input features

- name of the label column

These are saved in a serialized file using dill when you ran the `preprocess` script earlier and you can deserialize it using the cell below to view its contents

In [11]:

```
import dill as pickle

# Helper function to load the serialized file
def load(filename):
  with tf.io.gfile.GFile(filename, 'rb') as f:
    return pickle.load(f)

# Load PreprocessData
preprocess_data = load('/content/results/PreprocessData')

# Print contents
pprint(vars(preprocess_data))
```

```
{'eval_files_pattern': 'results/eval-dataset/part*',
 'input_feature_spec': {'Energy': FixedLenFeature(shape=[], dtype=tf.float32, default_value=Non
e),
                        'TotalC': FixedLenFeature(shape=[], dtype=tf.int64, default_value=None),
                        'TotalH': FixedLenFeature(shape=[], dtype=tf.int64, default_value=None),
                        'TotalN': FixedLenFeature(shape=[], dtype=tf.int64, default_value=None),
                        'TotalO': FixedLenFeature(shape=[], dtype=tf.int64, default_value=None)},
 'labels': ['Energy'],
 'train_files_pattern': 'results/train-dataset/part*'}
```

The next sections will describe how these are implemented as a Beam pipeline in `preprocess.py`. You can open this file in a separate text editor so you can look at it more closely while reviewing the snippets below.

Applying element-based transforms

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

▸ Click here to see the code snippet

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

▸ Click here to see the code snippet

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

▸ Click here to see the code snippet

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.

▸ Click here to see the code snippet

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. 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. 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`:

▸ Click here to see a sample output of *beam.ParDo(FormatMolecule())*

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`:

▸ Click here to see a sample output of *beam.ParDo(CountAtoms())*

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.

▸ Click here to see the code snippet

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.

▸ Click here to see the code snippet

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.

▸ Click here to see the code snippet

Writing the output

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

▸ Click here to see the code snippet

Phase 3: Training

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

The script uses a simple dense neural network for the regression problem. The `trainer/task.py` file contains the code for training the model. The main function of `trainer/task.py` loads the parameters needed from the preprocessing phase and passes it to the task runner function (i.e. `run_fn`).

In this exercise, we will not focus too much on the training metrics (e.g. accuracy). That is discussed in other courses of this specialization. The main objective is to look at the outputs and how it is connected to the prediction phase.

In [12]:

```
# Print help documentation
!python ./molecules/trainer/task.py --help
```

```
usage: task.py [-h] --work-dir WORK_DIR

optional arguments:
  -h, --help           show this help message and exit
  --work-dir WORK_DIR  Directory for staging and working files. This can be a
                       Google Cloud Storage path. (default: None)
```

In [13]:

```
# Run the trainer.
!python ./molecules/trainer/task.py --work-dir {WORK_DIR}
```

```
2021-09-24 13:44:02.730636: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successfu
l NUMA node read from SysFS had negative value (-1), but there must be at least one NUMA node, so
returning NUMA node zero
2021-09-24 13:44:02.740568: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successfu
l NUMA node read from SysFS had negative value (-1), but there must be at least one NUMA node, so
returning NUMA node zero
2021-09-24 13:44:02.741365: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successfu
l NUMA node read from SysFS had negative value (-1), but there must be at least one NUMA node, so
returning NUMA node zero
2021-09-24 13:44:02.742710: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successfu
l NUMA node read from SysFS had negative value (-1), but there must be at least one NUMA node, so
returning NUMA node zero
2021-09-24 13:44:02.743489: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successfu
l NUMA node read from SysFS had negative value (-1), but there must be at least one NUMA node, so
returning NUMA node zero
```

```
2021-09-24 13:44:02.744351: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successfu
l NUMA node read from SysFS had negative value (-1), but there must be at least one NUMA node, so
returning NUMA node zero
2021-09-24 13:44:03.208617: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successfu
l NUMA node read from SysFS had negative value (-1), but there must be at least one NUMA node, so
returning NUMA node zero
2021-09-24 13:44:03.209463: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successfu
l NUMA node read from SysFS had negative value (-1), but there must be at least one NUMA node, so
returning NUMA node zero
2021-09-24 13:44:03.210284: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successfu
l NUMA node read from SysFS had negative value (-1), but there must be at least one NUMA node, so
returning NUMA node zero
2021-09-24 13:44:03.210990: W tensorflow/core/common_runtime/gpu/gpu_bfc_allocator.cc:39] Overrid
ing allow_growth setting because the TF_FORCE_GPU_ALLOW_GROWTH environment variable is set. Origi
nal config value was 0.
2021-09-24 13:44:03.211057: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1510] Created devi
ce /job:localhost/replica:0/task:0/device:GPU:0 with 10697 MB memory:  -> device: 0, name: Tesla
K80, pci bus id: 0000:00:04.0, compute capability: 3.7
Model: "model"
__________________________________________________________________________________________________
_
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
=
NormalizedC (InputLayer)        [(None, 1)]          0
__________________________________________________________________________________________________
_
NormalizedH (InputLayer)        [(None, 1)]          0
__________________________________________________________________________________________________
_
NormalizedN (InputLayer)        [(None, 1)]          0
__________________________________________________________________________________________________
_
NormalizedO (InputLayer)        [(None, 1)]          0
__________________________________________________________________________________________________
_
concatenate (Concatenate)       (None, 4)            0           NormalizedC[0][0]
                                                                 NormalizedH[0][0]
                                                                 NormalizedN[0][0]
                                                                 NormalizedO[0][0]
__________________________________________________________________________________________________
_
dense (Dense)                   (None, 128)          640         concatenate[0][0]
__________________________________________________________________________________________________
_
dense_1 (Dense)                 (None, 64)           8256        dense[0][0]
__________________________________________________________________________________________________
_
dense_2 (Dense)                 (None, 1)            65          dense_1[0][0]
==================================================================================================
=
Total params: 8,961
Trainable params: 8,961
Non-trainable params: 0
__________________________________________________________________________________________________
_
2021-09-24 13:44:03.654134: I tensorflow/compiler/mlir/mlir_graph_optimization_pass.cc:185] None
of the MLIR Optimization Passes are enabled (registered 2)
100/100 [==============================] - 6s 31ms/step - loss: 661.6962 - accuracy: 0.0012 - val
_loss: 324.6600 - val_accuracy: 0.0017
WARNING:absl:Function `serve_tf_examples_fn` contains input name(s) ID, TotalC, TotalH, TotalN, T
otalO with unsupported characters which will be renamed to id, totalc, totalh, totaln, totalo in
the SavedModel.
2021-09-24 13:44:11.548611: W tensorflow/python/util/util.cc:348] Sets are not currently consider
ed sequences, but this may change in the future, so consider avoiding using them.
```

The outputs of this phase are in the `model` directory. This will be the trained model that you will use for predictions.

In [14]:

```
!ls {WORK_DIR}/model
```

```
assets  keras_metadata.pb  saved_model.pb  variables
```

The important thing to note in the training script is it also exports the transformation graph with the model. That is shown in these lines:

▸ Click here to see the code snippet

The implementation of `_get_serve_tf_examples_fn()` is as follows:

▸ Click here to see the code snippet

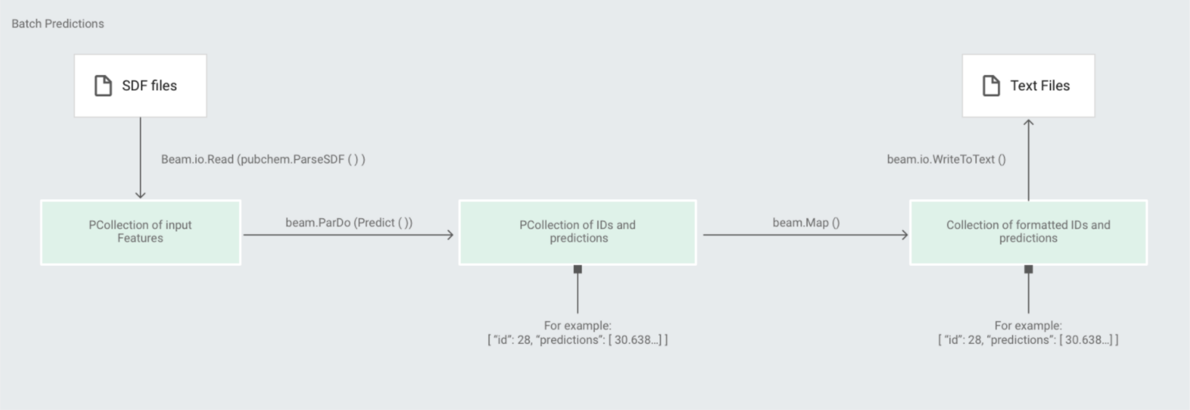
The use of `model.tft_layer` means that your model can accept raw data and it will do the transformation before feeding it to make predictions. It implies that when you serve your model for predictions, you don't have to worry about creating a pipeline to transform new data coming in. The model will already do that for you through this serving input

Phase 4: Prediction

After training the model, you can provide the model with inputs and it will make predictions. The pipeline in `predict.py` is responsible for making predictions. It reads the input files from the custom source and writes the output predictions as text files to the specified working directory.

▸ Click here to see the code snippet

The following image shows the steps in the prediction pipeline:


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

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

▸ Click here to see the code snippet

The code calls the run function with the following parameters:

▸ Click here to see the code snippet

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:

▸ Click here to see the code snippet

The transform reads from the appropriate source based on the pipeline's execution mode (i.e. batch), 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.

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. You can run the following cells to use the script to run batch predictions. For simplicity, you will use the same file you used for training. If you want however, you can use the data extractor script earlier to grab a different SDF file and feed it here.

In [15]:
```
# Print help documentation. You can ignore references to GCP and streaming data.
!python ./molecules/predict.py --help
```

```
usage: predict.py [-h] --work-dir WORK_DIR --model-dir MODEL_DIR
                  {batch,stream} ...

positional arguments:
  {batch,stream}
    batch               Batch prediction
    stream              Streaming prediction

optional arguments:
  -h, --help            show this help message and exit
  --work-dir WORK_DIR   Directory for temporary files and preprocessed
                        datasets to. This can be a Google Cloud Storage path.
                        (default: None)
  --model-dir MODEL_DIR
                        Path to the exported TensorFlow model. This can be a
                        Google Cloud Storage path. (default: None)
```

In [16]:

```
# Define model, input and output data directories
MODEL_DIR = f'{WORK_DIR}/model'
DATA_DIR = f'{WORK_DIR}/data'
PRED_DIR = f'{WORK_DIR}/predictions'
```

In [17]:

```
# Run batch prediction. This will take around 7 minutes.
!python ./molecules/predict.py \
  --model-dir {MODEL_DIR} \
  --work-dir {WORK_DIR} \
  batch \
  --inputs-dir {DATA_DIR} \
  --outputs-dir {PRED_DIR}
```

```
Listening...
WARNING:root:Make sure that locally built Python SDK docker image has Python 3.7 interpreter.
2021-09-24 13:44:20.134857: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successfu
l NUMA node read from SysFS had negative value (-1), but there must be at least one NUMA node, so
returning NUMA node zero
2021-09-24 13:44:20.143180: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successfu
l NUMA node read from SysFS had negative value (-1), but there must be at least one NUMA node, so
returning NUMA node zero
2021-09-24 13:44:20.144007: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successfu
l NUMA node read from SysFS had negative value (-1), but there must be at least one NUMA node, so
returning NUMA node zero
2021-09-24 13:44:20.145301: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successfu
l NUMA node read from SysFS had negative value (-1), but there must be at least one NUMA node, so
returning NUMA node zero
2021-09-24 13:44:20.146103: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successfu
l NUMA node read from SysFS had negative value (-1), but there must be at least one NUMA node, so
returning NUMA node zero
2021-09-24 13:44:20.146891: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successfu
l NUMA node read from SysFS had negative value (-1), but there must be at least one NUMA node, so
returning NUMA node zero
2021-09-24 13:44:20.642580: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successfu
l NUMA node read from SysFS had negative value (-1), but there must be at least one NUMA node, so
returning NUMA node zero
2021-09-24 13:44:20.643555: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successfu
l NUMA node read from SysFS had negative value (-1), but there must be at least one NUMA node, so
returning NUMA node zero
2021-09-24 13:44:20.644353: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successfu
l NUMA node read from SysFS had negative value (-1), but there must be at least one NUMA node, so
returning NUMA node zero
2021-09-24 13:44:20.645084: W tensorflow/core/common_runtime/gpu/gpu_bfc_allocator.cc:39] Overrid
ing allow_growth setting because the TF_FORCE_GPU_ALLOW_GROWTH environment variable is set. Origi
nal config value was 0.
2021-09-24 13:44:20.645149: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1510] Created devi
ce /job:localhost/replica:0/task:0/device:GPU:0 with 10697 MB memory:  -> device: 0, name: Tesla
K80, pci bus id: 0000:00:04.0, compute capability: 3.7
2021-09-24 13:44:21.084460: I tensorflow/compiler/mlir/mlir_graph_optimization_pass.cc:185] None
of the MLIR Optimization Passes are enabled (registered 2)
```

The results should now be in the `predictions` folder. This is just a text file so you can easily print the output.

In [18]:

```
# List working directory
!ls {WORK_DIR}
```

```
data          model        PreprocessData  train-dataset         transform_fn
eval-dataset  predictions  tft-temp        transformed_metadata
```

In [19]:

```
# Print the first 100 results
!head -n 100 /content/results/predictions/part-00000-of-00001
```

```
{"id": [1], "predictions": [[34.93013381958008]]}
{"id": [2], "predictions": [[35.41752624511719]]}
{"id": [3], "predictions": [[33.27965545654297]]}
{"id": [4], "predictions": [[-2.5731775760650635]]}
{"id": [5], "predictions": [[35.57941436767578]]}
{"id": [6], "predictions": [[21.5170841217041]]}
{"id": [7], "predictions": [[-26.260202407836914]]}
{"id": [8], "predictions": [[34.458534240722656]]}
{"id": [9], "predictions": [[83.24590301513672]]}
{"id": [11], "predictions": [[-11.163381576538086]]}
{"id": [12], "predictions": [[31.53417205810547]]}
{"id": [13], "predictions": [[-8.567987442016602]]}
{"id": [14], "predictions": [[64.65614318847656]]}
{"id": [16], "predictions": [[16.081748962402344]]}
```

```
{"id": [17], "predictions": [[28.514808654785156]]}
{"id": [18], "predictions": [[61.488948822021484]]}
{"id": [19], "predictions": [[32.30487060546875]]}
{"id": [20], "predictions": [[35.795833587646484]]}
{"id": [21], "predictions": [[33.48374557495117]]}
{"id": [22], "predictions": [[31.738262176513672]]}
{"id": [23], "predictions": [[38.174808502197266]]}
{"id": [26], "predictions": [[24.55595588684082]]}
{"id": [28], "predictions": [[27.744110107421875]]}
{"id": [29], "predictions": [[14.797242164611816]]}
{"id": [30], "predictions": [[18.084117889404297]]}
{"id": [31], "predictions": [[63.15522003173828]]}
{"id": [32], "predictions": [[51.62486267089844]]}
{"id": [33], "predictions": [[-1.990780234336853]]}
{"id": [34], "predictions": [[-1.0159938335418701]]}
{"id": [35], "predictions": [[31.046775817871094]]}
{"id": [36], "predictions": [[44.889225006103516]]}
{"id": [37], "predictions": [[52.80373001098633]]}
{"id": [38], "predictions": [[33.48374557495117]]}
{"id": [39], "predictions": [[22.84896469116211]]}
{"id": [40], "predictions": [[44.1185302734375]]}
{"id": [41], "predictions": [[53.291133880615234]]}
{"id": [42], "predictions": [[55.1158332824707]]}
{"id": [43], "predictions": [[41.398258209228516]]}
{"id": [44], "predictions": [[36.93250274658203]]}
{"id": [45], "predictions": [[37.90729904174805]]}
{"id": [46], "predictions": [[82.27111053466797]]}
{"id": [47], "predictions": [[23.823749542236328]]}
{"id": [48], "predictions": [[26.973413467407227]]}
{"id": [49], "predictions": [[22.07826805114746]]}
{"id": [50], "predictions": [[62.4637336730957]]}
{"id": [51], "predictions": [[40.4234733581543]]}
{"id": [58], "predictions": [[20.33278465270996]]}
{"id": [59], "predictions": [[58.6894645690918]]}
{"id": [61], "predictions": [[88.15682983398438]]}
{"id": [62], "predictions": [[2.0667941570281982]]}
{"id": [63], "predictions": [[5.812937259674072]]}
{"id": [64], "predictions": [[12.21418285369873]]}
{"id": [65], "predictions": [[11.239397048950195]]}
{"id": [66], "predictions": [[1.5794004201889038]]}
{"id": [67], "predictions": [[61.488948822021484]]}
{"id": [68], "predictions": [[81.98780822753906]]}
{"id": [69], "predictions": [[33.27965545654297]]}
{"id": [70], "predictions": [[23.823749542236328]]}
{"id": [71], "predictions": [[42.1689567565918]]}
{"id": [72], "predictions": [[32.30487060546875]]}
{"id": [73], "predictions": [[75.5761947631836]]}
{"id": [75], "predictions": [[-3.547964096069336]]}
{"id": [77], "predictions": [[44.889225006103516]]}
{"id": [78], "predictions": [[5.624638080596924]]}
{"id": [79], "predictions": [[-16.6355037689209]]}
{"id": [80], "predictions": [[42.93965530395508]]}
{"id": [81], "predictions": [[93.67658233642578]]}
{"id": [82], "predictions": [[95.98868560791016]]}
{"id": [83], "predictions": [[38.394683837890625]]}
{"id": [85], "predictions": [[23.241352081298828]]}
{"id": [86], "predictions": [[18.854814529418945]]}
{"id": [87], "predictions": [[21.307571411132812]]}
{"id": [89], "predictions": [[28.98641014099121]]}
{"id": [91], "predictions": [[26.135839462280273]]}
{"id": [92], "predictions": [[32.882102966308594]]}
{"id": [93], "predictions": [[42.1689567565918]]}
{"id": [95], "predictions": [[62.4637336730957]]}
{"id": [96], "predictions": [[20.33278465270996]]}
{"id": [98], "predictions": [[18.58730125427246]]}
{"id": [101], "predictions": [[12.984879493713379]]}
{"id": [102], "predictions": [[13.95966625213623]]}
{"id": [104], "predictions": [[31.53417205810547]]}
{"id": [105], "predictions": [[57.71467971801758]]}
{"id": [106], "predictions": [[45.23940658569336]]}
{"id": [107], "predictions": [[16.475847244262695]]}
{"id": [108], "predictions": [[9.194817543029785]]}
{"id": [109], "predictions": [[25.432022094726562]]}
{"id": [110], "predictions": [[37.90729904174805]]}
{"id": [111], "predictions": [[12.752662658691406]]}
{"id": [112], "predictions": [[68.63275146484375]]}
{"id": [113], "predictions": [[69.12014770507812]]}
{"id": [114], "predictions": [[12.685784339904785]]}
{"id": [116], "predictions": [[42.43647003173828]]}
{"id": [117], "predictions": [[-2.646965265274048]]}
{"id": [118], "predictions": [[-1.8024810552597046]]}
{"id": [119], "predictions": [[7.857514381408691]]}
{"id": [120], "predictions": [[30.55938720703125]]}
{"id": [122], "predictions": [[53.37034606933594]]}
{"id": [123], "predictions": [[-12.401966094970703]]}
```

Wrap Up

You've now completed all phases of the Beam-based pipeline! Similar processes are done under the hood by higher-level frameworks such as TFX and you can use the techniques here to understand their codebase better or to extend them for your own needs. As mentioned earlier, the original article also offers the option to use GCP and to perform online predictions as well. Feel free to try it out but be aware of the recurring costs.

On to the next part of the course!