Case study: A machine learning pipeline for virtual material testing

By Quantiphi, Inc. (/solutions/partners/www.quantiphi.com)

This article describes how Quantiphi leveraged Google Cloud to build an end-to-end machine learning (ML) pipeline for a global manufacturing client to help them design brake pad materials. The article provides a high-level description of how Quantiphi used BigQuery to handle, clean, and transform large sensor data files and training ML classifiers using the TensorFlow Estimator API on Compute Engine. This ultimately let the client predict material composition that led to optimized friction performance through a virtual material testing framework.

Process flow

Any ML learning pipeline goes through the following processes:

1. Data ingestion
2. Data preprocessing
3. Feature engineering
4. Model training and evaluation
5. Deployment

The following sections elaborate on each of these processes in the customer solution created by Quantiphi.

Data ingestion

The customer was interested in testing friction for brake pads and had sensors attached to testing devices. To prepare for the process flow described in this section, material scientists and chemists manually created an on-premises system that consisted of standardized files of configured material sensor data generated by brake pad testing machines. Adding to the complexity of exercise, the sensor data files were generated by brake pad testing machines at millisecond intervals.

Quantiphi used the Cloud SDK `gsutil` tool to transfer to Cloud Storage over 20 years' worth of historical sensor data and manufacturing process data. Together, the two data sources amounted to a massive 2 TB of uncompressed data.

Data preprocessing

The sensor data files (in zip files) were fed as the source into Dataflow. There they were uncompressed and files with the extension `.RES` were extracted. At this stage, the file name and test year metadata was added. Feature extraction and cleaning of the data were then undertaken. This import process was performed every year for the entire set of data. The resulting `.RES` files were aggregated into a separate bucket.

The following example shows a code snippet from the Dataflow pipeline:

```
opts = beam.pipeline.PipelineOptions(flags = [], **options)
    p = beam.Pipeline(options = opts)

    preprocessing = (p

    | 'ReadFromGCS' >> beam.io.ReadFromText('gs://example-friction/ziplist.tx
    | 'ExtractZIP' >> beam.ParDo(extractZIP())
    | 'ExtractRES' >> beam.ParDo(extractRES())
    | 'ETL' >> beam.ParDo(doETL())
    )

    result = p.run()
    result.wait_until_finish()
```

Sensor data files and column names were not in English, and they had to be translated for analysis. Furthermore, because of a shift in technology stacks over the years, data files didn't follow a coherent schema. For example, often an underlying feature value was in two different columns with different names. This made the data inconsistent for any downstream analysis. In order to render data usable for business or any domain-specific analysis, Quantiphi first used the Cloud Translation API to translate column headers from the domain language to English.

Figure 1 outlines the end-to-end machine learning pipeline on Google Cloud.

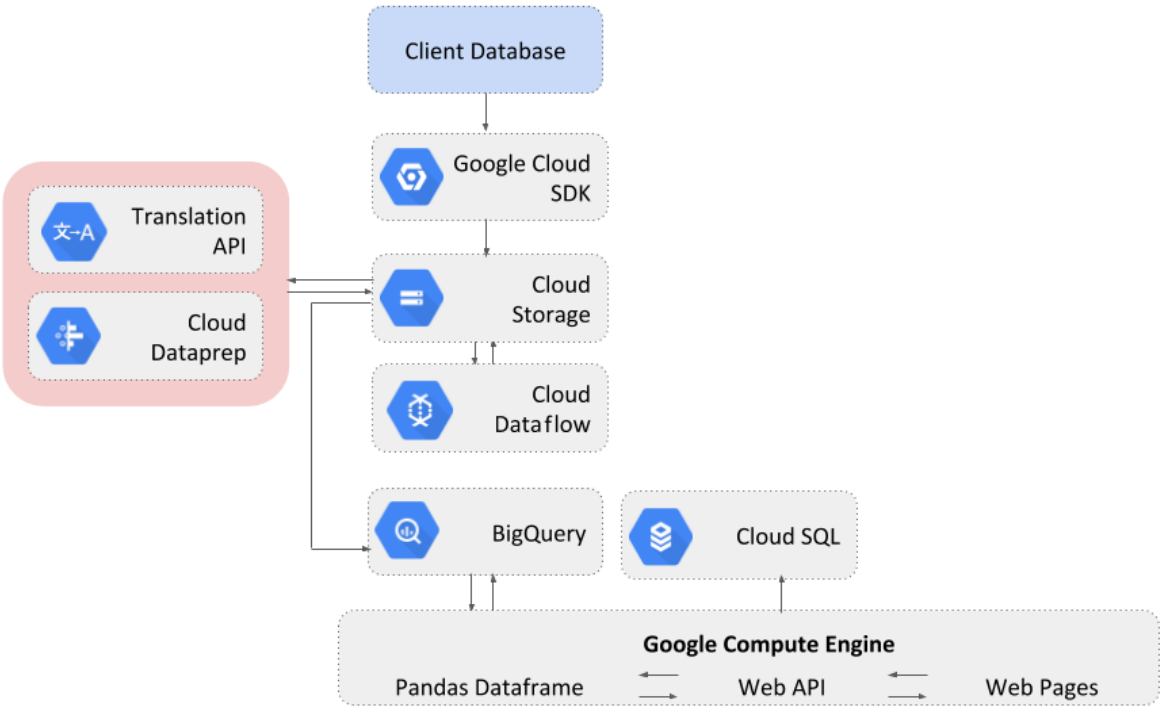

Figure 1. The architecture of the data processing pipeline on Google Cloud

The pooled `.RES` files, along with manufacturing process data, were uploaded to BigQuery. Quantiphi aggregated and joined features from disparate database tables in BigQuery. For example, because manufacturing process variables and raw material compositions were generated along different stages of the material-design lifecycle, they were stored in separate log files that come from different physical datastores. Figure 2 shows the BigQuery schema.

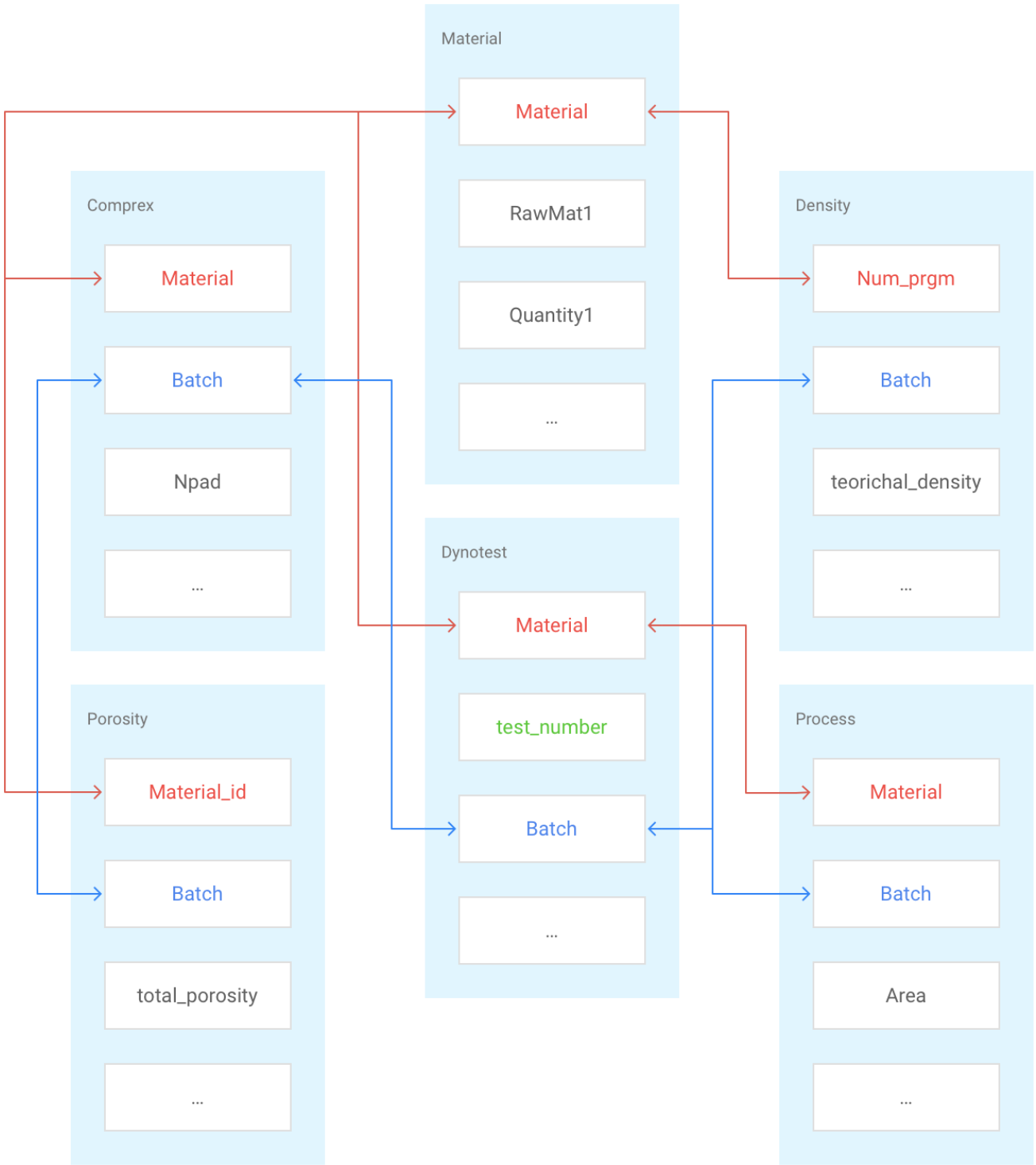

Figure 2. Database schema in BigQuery. Each table corresponds to a subset of the input feature space

Next, data was preprocessed in batches and sequential operations for transforming and mapping values were applied to the raw feature set. Each feature type (for example, categorical, numerical, and string) had to be encoded and vectorized using specific techniques. Data at this stage formed the initial input feature matrix to the machine learning model.

Feature engineering

The cleaned input feature matrix was sparse and high dimensional, with close to 2000 features. Machine learning models overfit and fail to generalize well to unseen data when confronted with a large number of input features. This results in estimates that have large variances, making the model unreliable in production. In practice, this can often be mitigated by using dimensionality reduction or feature hashing, or by modifying features using business understanding. This process is known as *feature engineering* and lies at the heart of structured machine learning.

During the scoping phase of the project, an important requirement of the final solution was interpretability. The end users wanted a clear understanding of how each input feature influenced the predictions of the model. This would enable them to optimize their underlying manufacturing process and consume predictions of the ML model to disqualify inferior friction-pad designs. Quantiphi first ranked each input feature by importance to the business stakeholder. Dimensionality techniques like PCA were ruled out because it modifies the feature space by generating linear combinations of inputs, thus sacrificing interpretability for performance. Null and NA values were removed and imputed; correlated variables and zero-variance variables were excluded; highly diverse features were aggregated, and one-hot encoding was performed to convert categorical features into numerical features. This custom feature engineering effort contracted the dimensionality while retaining the most salient business indicators.

One of the best examples of single-variable text decryption was performed as a part of the feature engineering efforts. Each variable contained information related to a process that had 10 to 12 subset variables, with all information encrypted in a string format. The model treated the variable as a string, converting the subset variables into their respective quantitative values to make them machine-learning ready.

Model training and evaluation

TensorFlow Estimators are included in a high-level Tensorflow API that simplifies machine learning by encapsulating training, evaluation, prediction, and export for serving the model. Developers can either use the pre-made Estimators or write custom Estimators. All Estimators—whether pre-made or custom—are classes based on the `tf.estimator.Estimator` class.

For this use case, Quantiphi used a pre-made Estimator known as `TensorForestEstimator` to train the classification model.

Figure 3 shows the pipeline for training and serving the machine learning classifier.

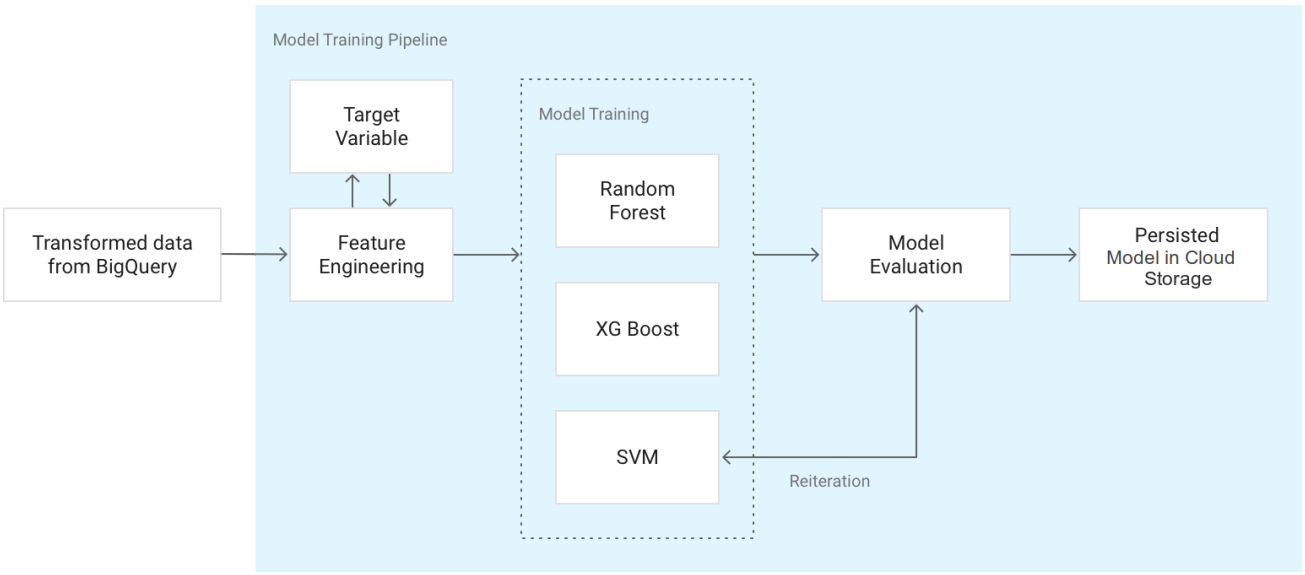

Figure 3. The end-to-end pipeline used to train and evaluate machine learning classifiers on Compute Engine

The steps involved in training using the Estimator API were as follows:

1. The dataset was split into training and test sets using the `split_dataset` function. This function takes input data and splits it into training and test sets based on the evaluation size and target variable as specified by the user.
2. After the splitting, the datatypes of the input features and target variables were converted into `float32` and `int32`, respectively, by using the `convert_datatype` function. This was required because the TensorFlow Estimator allows only float and integer datatype functions.
3. The input data had to be converted into a format mandated for the TensorFlow Estimator. To accomplish this, input data was passed to `generate_tf_input_fn`. This returns a function that allows the Pandas dataframe to be fed into the TensorFlow Estimator.
4. The Estimator was instantiated and returned using the `instatantiate_estimator` function. This function takes the number of classes present in a target variable as well as the number of features present in the input dataset, along with other hyperparameters.
5. The model was trained by calling the `fit_model` function. This function takes both the instantiated Estimator from the preceding step and the input function from step 3 to fit a model using the `fit` method of the TensorFlow Estimator.
6. Finally, the trained model was evaluated by calling the `evaluate_model` function. This function takes the model and test dataset as input and returns the confusion matrix

to assess the model quality.

The following listing shows the code snippet developed by Quantiphi for training the model using the TensorFlow Estimator API.

```
from sklearn.cross_validation import StratifiedKFold
import matplotlib.pyplot as plt
import tensorflow as tf
from sklearn.metrics import confusion_matrix,precision_recall_fscore_support
import warnings
import numpy as np
warnings.filterwarnings("ignore")
def split_dataset(data,evaluation_size,target_variable):
    X = data.drop([target_variable],axis=1)
    y = data[target_variable]
    np.random.seed(100)
    eval_size = evaluation_size
    kf = StratifiedKFold(y,round(1./eval_size))
    train_indices, test_indices = next(iter(kf))
    X_train, y_train = X.iloc[train_indices], y.iloc[train_indices]
    X_test, y_test = X.iloc[test_indices], y.iloc[test_indices]
    return ((X_train,y_train),(X_test,y_test))



def convert_datatype(X_train,Y_train):
    tf.set_random_seed(100)
    np.random.seed(100)
    train_X = X_train.astype(dtype = np.float32)
    train_Y = Y_train.astype(dtype = np.int32)
    return(train_X,train_Y)



def generate_tf_input_fn(x_input,y_input,num_epochs,target_variable):
    # this is the function we are generating
    def _input_fn_():
        # generate a standard input function
        train_input_fn = tf.estimator.inputs.pandas_input_fn(
            x= x_input,
            y= y_input,
            num_epochs=num_epochs,
            shuffle=False,
            target_column=target_variable,
        )
        # execute the standard input function
        x, y = train_input_fn()
```

```
        # expand the shape of the results (necessary for TensorForest)
        for name in x:
            x[name] = tf.expand_dims(x[name], 1, name= name)
        return x, y
        return _input_fn_



def instantiate_estimator(n_class,n_features,n_trees,bag_fraction,feat_bagging
     params=tf.contrib.tensor_forest.python.tensor_forest.ForestHParams(num_c

    classifier=tf.contrib.tensor_forest.client.random_forest.TensorForestEstir
    return classifier




def fit_model(classifier,forest_train_input_fn):
    model=classifier.fit(input_fn=forest_train_input_fn)
    return model



def evaluate_model(model,X_test,y_test,num_epochs,average_type):
    test_X = X_test.astype(dtype = np.float32)
    test_Y = y_test.astype(dtype = np.int32)
    forest_cv_input_fn = generate_tf_input_fn(test_X, test_Y, num_epochs=num_e
    accuracy_score = model.evaluate(input_fn=forest_cv_input_fn)["accuracy"]
    print("\n Test Accuracy: {0:f}\n".format(accuracy_score))
    predictions = list(model.predict(input_fn=forest_cv_input_fn))
    predicted_classes = [p["classes"] for p in predictions]
    print("New Samples, Class Predictions:{}\n".format(predicted_classes))
    cm=confusion_matrix(test_Y,predicted_classes)
           eval_metrics=precision_recall_fscore_support(test_Y,predicted_class
    return (cm,eval_metrics)
```

The best performing classifier for Quantiphi's model achieved a precision measure of 83%.

Deployment

The final model was made persistent and didn't need to be retrained every time a call to **`predict`** was made. A containerized web service was built in Flask and deployed on a Compute Engine. Figure 4 illustrates how the web service takes live user data that is entered using a web interface as inputs on which validation is performed. Validated user

inputs then make a call to the deployed model that ultimately predicts the output class and its corresponding confidence estimate.

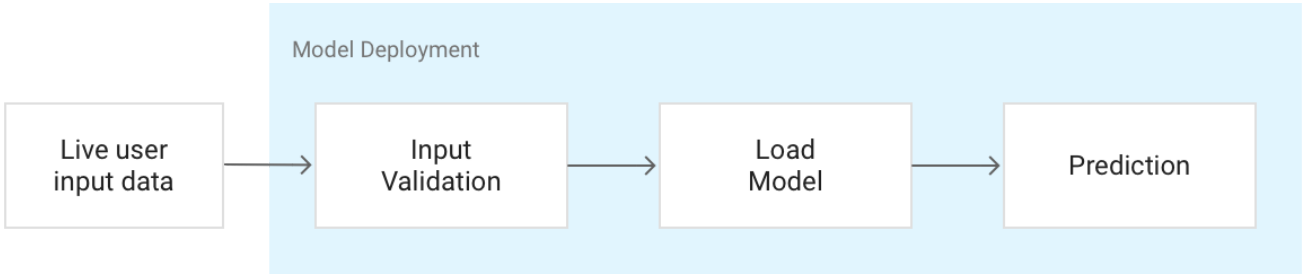

Figure 4. Visualization of how a web service takes user data as input and produces a prediction

What's next

- Quantiphi overview: Distributed training using TensorFlow Estimator APIs (/solutions/partners/quantiphi-distributed-training-using-tensorflow)
- Quantiphi case study: Automating metadata extraction for video commercials (/solutions/partners/quantiphi-video-metadata-extraction)
- Try out other Google Cloud features for yourself. Have a look at our tutorials (/docs/tutorials).

Last updated 2020-11-16 UTC.