Google Cloud

ASL Tech Talk:

Time Series Anomaly Detection

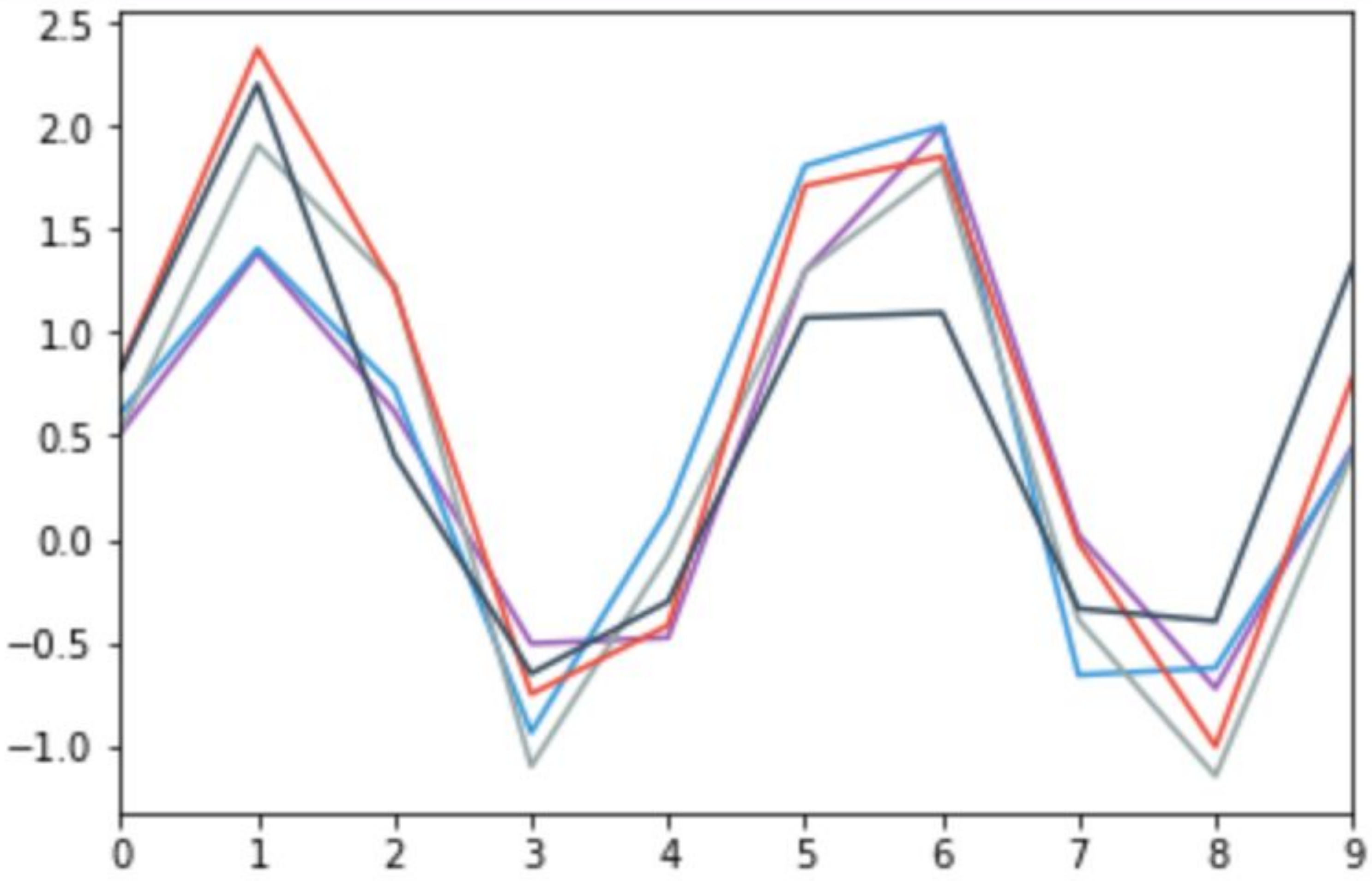
What is anomaly detection?

- Often times in data mining/analysis we want to be able to find outliers; rare events, occurrences, etc. that don't belong to our distribution of interest.
- Important for many industries such as banking fraud detection, IoT predictive maintenance, cybersecurity threat detection.

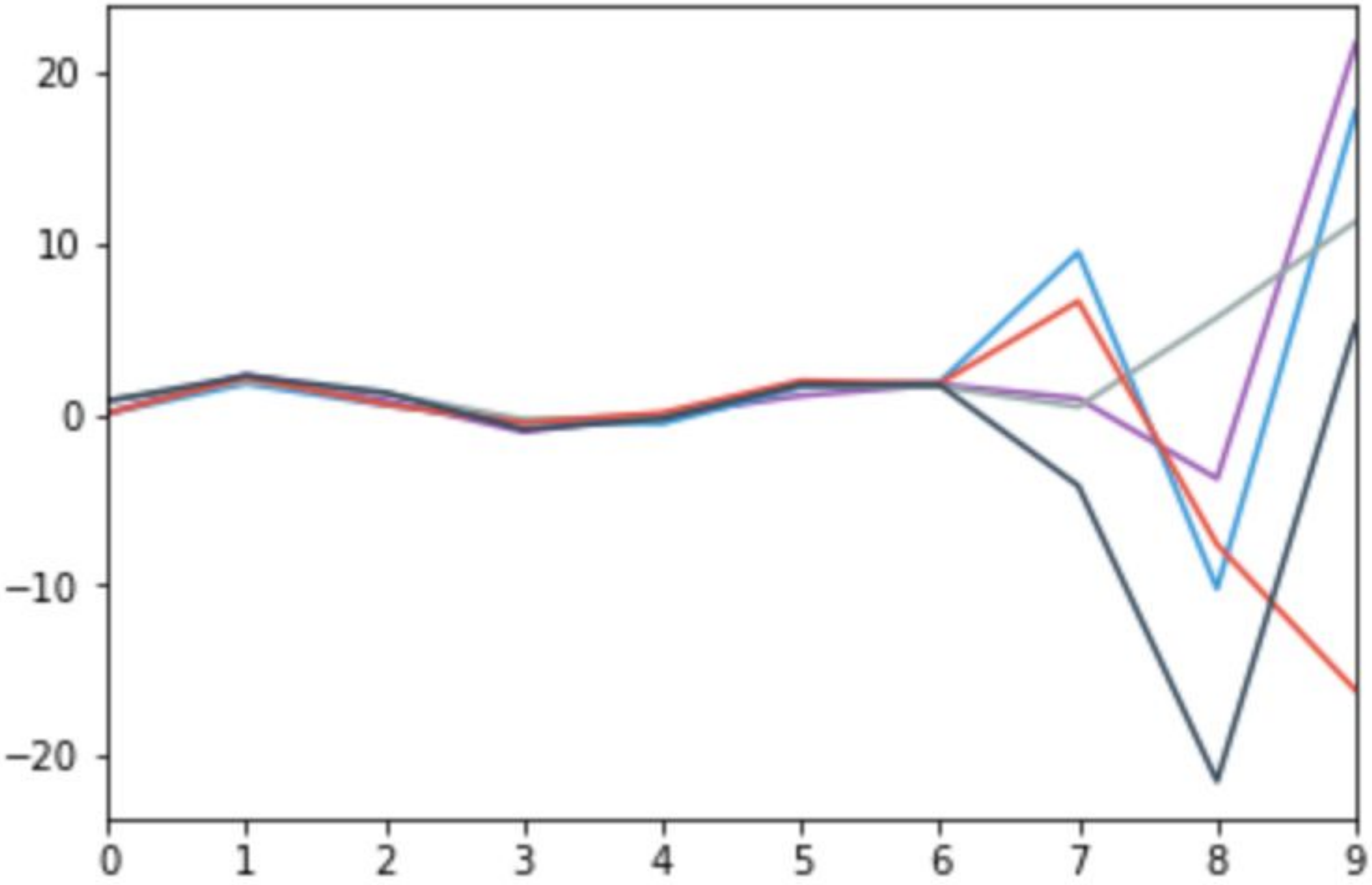
With time series?

- When most people think of anomaly detection, they first go to a point prediction.
- However, a point by itself may be very hard to flag.
- Often, it is a sequence of activity that together indicates an anomaly!
- Can look at this from both a time or feature major view.

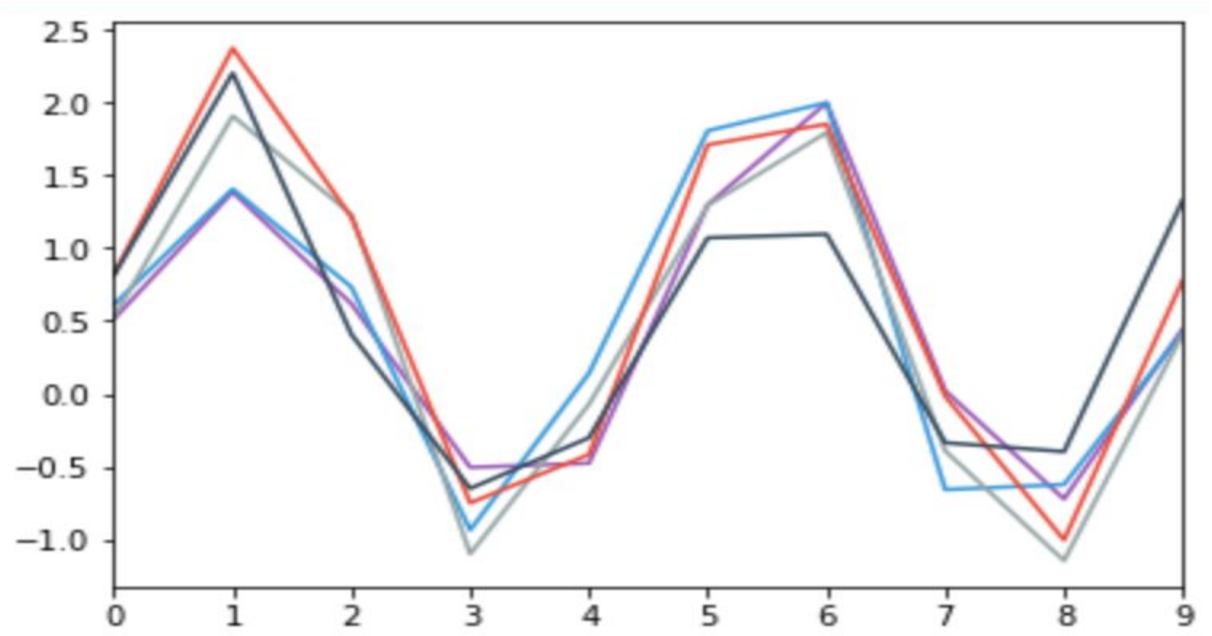
Is this anomalous?

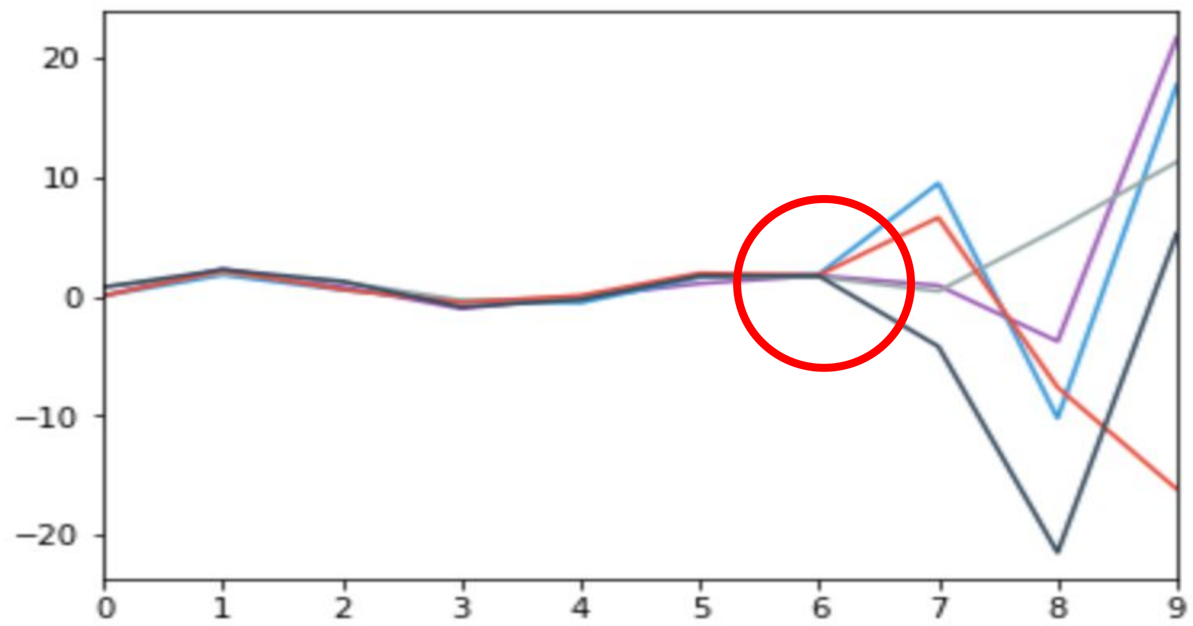


Is this anomalous?



Is this anomalous?

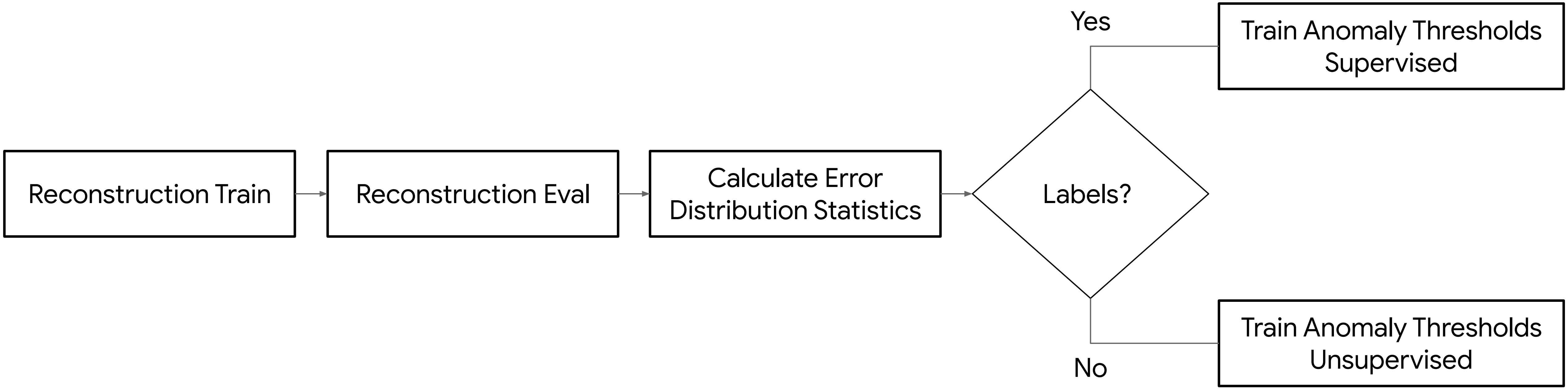




How to measure?

- There are many methods for time series anomalies detection.
- A common method is reconstruction where the predictions are the original inputs.
- We then calculate the distribution of errors so we run new data through it.
- Lastly, we tune anomaly thresholds so we can draw a line in the sand on what we're saying is what.

Training High-level System



LSTM-based Encoder-Decoder for Multi-sensor Anomaly Detection
https://arxiv.org/abs/1607.00148

Datasets

- Since there are multiple stages there are multiple datasets.

- sn1: Large num of norm seqs to be used in reconstruction training.
- vn1: Smaller num of norm seqs to be used for reconstruction evaluation and calculating error distributions.
- vn2/va: Smaller num of norm and anom seqs mixed together.
- tn/ta: Smaller num of norm and anom seqs mixed together.

Reconstruction Training

- The main goal for reconstruction is to bottleneck the inputs through the system so that a compressed representation will be learned (and not the identity function).
- Can do this with autoencoders, dimensionality reduction, density estimation, etc.
- For this solution example, used a meta-model with a dense network, LSTM encoder-decoder, and PCA.

Reconstruction Evaluation

- Obviously, we want to be able to predict "normal" sequences very well since it is the reconstruction error that we will use to flag for anomalies or not.
- This is a great point to use Hyperparameter Tuning to try and get the best reconstruction possible.

Calculate Error Distribution Statistics

- We now have a trained reconstruction model, but what to do with it?
- Our hypothesis is that since it was trained on "normal" sequences then it should have low error when a sequence should be "normal".
- However, if there is high error this might be an indication of an anomaly.

Calculate Error Distribution Statistics

- Therefore we need to learn a distribution so that we can have some semblance of distance from it for each example.
- A common assumption is that our error is normally distributed therefore the usual approach is to perform Maximum Likelihood Estimation (MLE).
- Once we have found the mean and covariance matrix, we can calculate the Mahalanobis Distance for each example.

Mahalanobis Distance, generalized n-dimensional z-score

Training

$$\mu_j = \frac{1}{n}\sum_{i=1}^{n} X_{ij}$$

$$\Sigma = (X - \mu)^T (X - \mu)$$

Inference

$$MD = diag((X - \mu)\Sigma^{-1}(X - \mu)^T)$$

Calculate Error Distribution Statistics

- For this solution, since our examples are being read in batch, there was a need to create variables to store the running counts, column means, and covariance matrices.
- This needs to be run with the Estimator in TRAIN mode so that they will be written to a checkpoint.

Tuning Anomaly Thresholds

- We've learned the reconstruction error distribution, now we need to set a good threshold so that we minimize the numbers of false positives and negatives, each of which will have different costs.
- The solution branches here depending on if you have labeled data where sequences are annotated as anomalous or not.

Tuning Anomaly Thresholds: Supervised

- If we're lucky enough to have labels that human knowledge will help in our classification task.
- The dataset is a mix of normal and anomalous sequences.
- Remember this isn't an annotation per timestep or per feature, but overall for each matrix in the batch.
- Here we are maximizing an F-$\boldsymbol{\beta}$ score (with $\boldsymbol{\beta}$<1 due to the rarity and wanting to catch all of them).

Tuning Anomaly Thresholds: Supervised

- Sometimes you can cheat a little with supervision...
- For instance, for IoT predictive maintenance, there is a high probability that right after a device begins collecting data, it is probably normal.
- Likewise, right before a device crashes, there was probably anomalous activity just before.
- With some logic based on domain knowledge you can fuzzily assign labels.

Tuning Anomaly Thresholds: Unsupervised

- If we don't have labels, then we have to resort to unsupervised methods.
- In this solution, we calculate the MLE this time for the Mahalanobis Distance and allow a certain number of standard deviations from the mean to flag a sequence as normal.
- This won't have as good of tuning as in the supervised case, but at least it uses the data rather than just be arbitrarily set by a human.

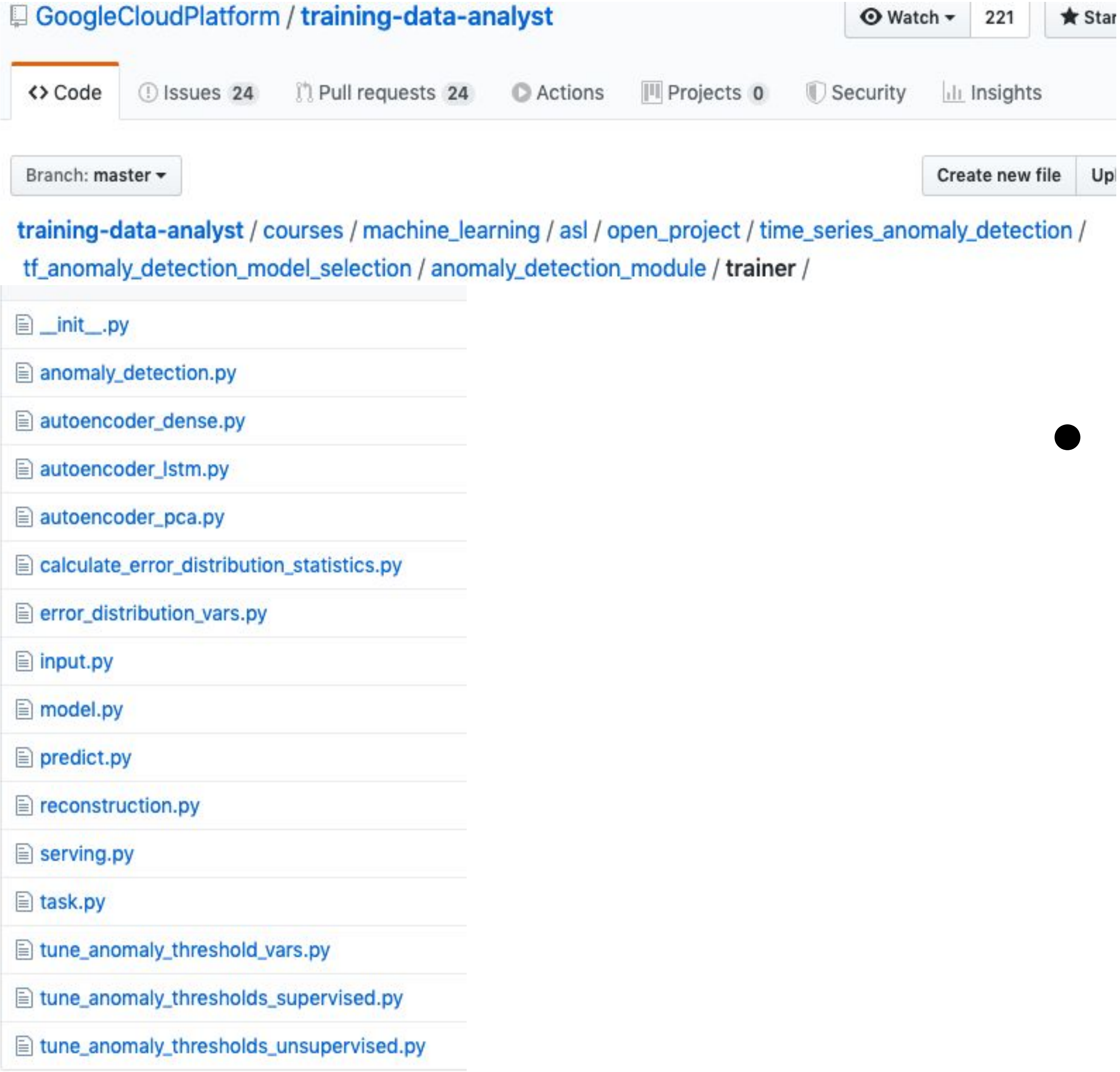
Inference

- For inference, just like in training the examples go through the reconstruction model.
- Then their reconstruction errors are calculated.
- Calculate next their Mahalanobis Distances.
- Then compare with trained anomaly thresholds.
- If any timestep is flagged as anomalous then the whole sequence is flagged in the time major view.
- Likewise for features in the feature major view.

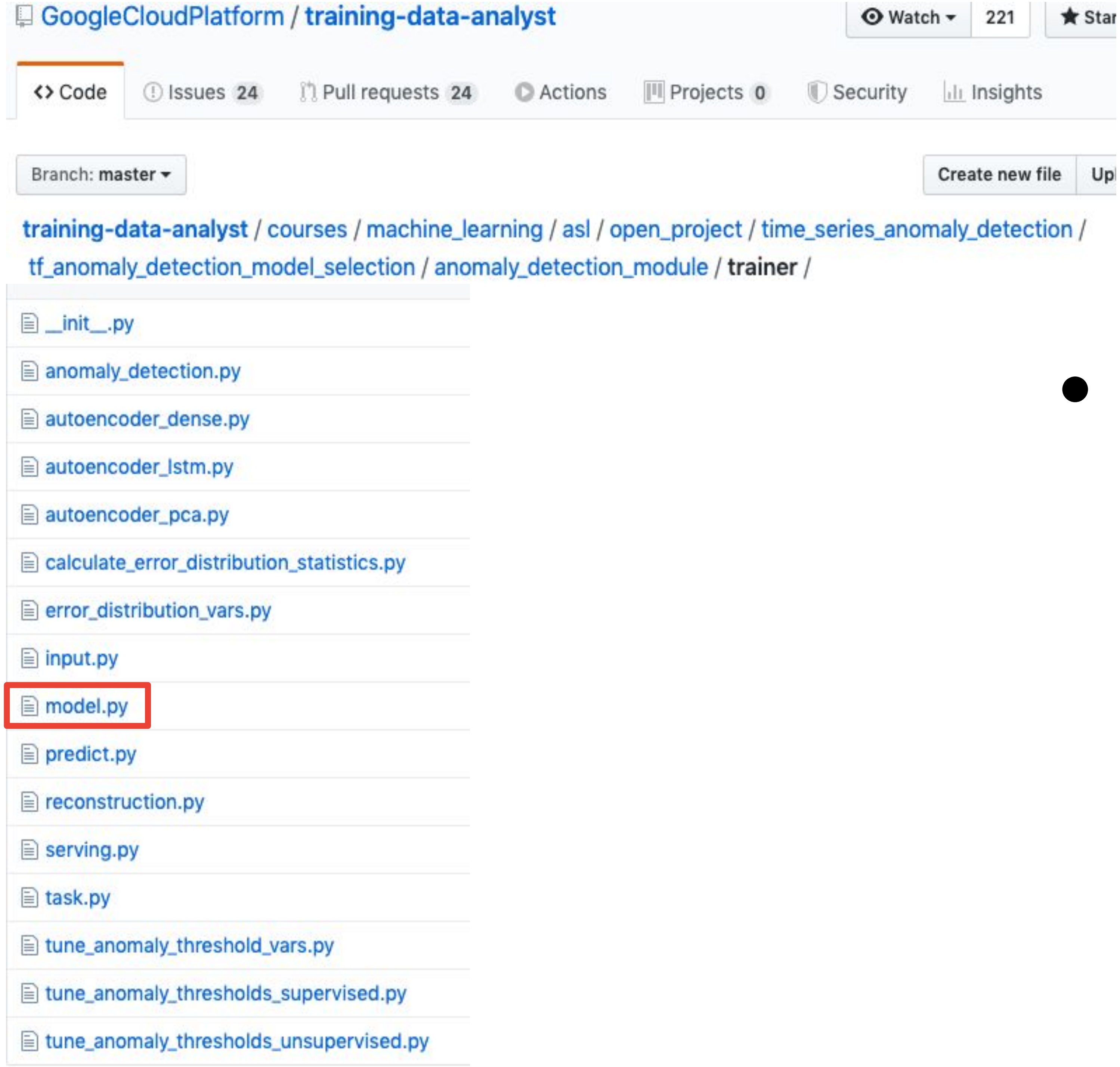
Non-traditional ML Code

- To have all of these pieces of this complex model work together, there were a lot of non-standard ways to do things in the code.
- A lot of gotchas and subtle nuances, so a lot of testing is a must.

Repo



- Python trainer module is currently made up of 16 files.



model.py

- model.py contains the estimator instantiation and the calls to train_and_evaluate.

model.py

```python
def train_and_evaluate(args):
  # Create our custom estimator using our model function
  estimator = tf.estimator.Estimator(...)

  if args["training_mode"] == "reconstruction":
    # Calculate max_steps
    max_steps = ...

    # Create eval spec to read in our validation data
    eval_spec = tf.estimator.EvalSpec(...)

    if args["model_type"] == "pca":
      # Create train spec to read in our training data
      train_spec = tf.estimator.TrainSpec(...)
      # Check to see if we need to additionally tune principal components
      if not args["autotune_principal_components"]:
        # Create train and evaluate loop to train and evaluate our estimator
        tf.estimator.train_and_evaluate(...)
      else:
          if (args["k_principal_components_time"] is None or
              args["k_principal_components_feat"] is None):
            # Create train and evaluate loop to train and evaluate our estimator
            tf.estimator.train_and_evaluate(...)
    else:  # dense_autoencoder or lstm_enc_dec_autoencoder
      # Create early stopping hook to help reduce overfitting
      early_stopping_hook = tf.contrib.estimator.stop_if_no_decrease_hook(...)

      # Create train spec to read in our training data
      train_spec = tf.estimator.TrainSpec(...)

      # Create train and evaluate loop to train and evaluate our estimator
      tf.estimator.train_and_evaluate(...)
```

- Cloud AI Platform needs train_and_evaluate and can't do just train or evaluate with the Estimator.
- This is just the first training phase, so we don't export a saved_model yet. Just checkpoints!

model.py

```python
  else:
    # Calculate max_steps
    max_steps = ...

    # if args["training_mode"] == "calculate_error_distribution_statistics"
    # Get final mahalanobis statistics over the entire val_1 dataset

    # if args["training_mode"] == "tune_anomaly_thresholds"
    # Tune anomaly thresholds using val_2 and val_anom datasets
    train_spec = tf.estimator.TrainSpec(...)

    if args["training_mode"] == "calculate_error_distribution_statistics":
      # Don't create exporter for serving yet since anomaly thresholds
      # aren't trained yet
      exporter = None
    elif args["training_mode"] == "tune_anomaly_thresholds":
      # Create exporter that uses serving_input_fn to create saved_model
      # for serving
      exporter = tf.estimator.LatestExporter(...)
    else:
      print("{0} isn't a valid training mode!".format(args["training_mode"]))

    # Create eval spec to read in our validation data and export our model
    eval_spec = tf.estimator.EvalSpec(...)

  if (args["training_mode"] == "calculate_error_distribution_statistics" or
      args["training_mode"] == "tune_anomaly_thresholds"):
    # Create train and evaluate loop to train and evaluate our estimator
    tf.estimator.train_and_evaluate(...)

  return
```

- We now handle the other two training modes.
- We export a saved_model only during the third training mode for tuning anomaly thresholds.

model.py

```python
# Calculate max_steps
max_steps = int(args["reconstruction_epochs"] * args["train_examples"])
max_steps = max_steps // args["train_batch_size"]
max_steps += args["previous_train_steps"]
```

```python
if args["model_type"] == "pca":
  # Create train spec to read in our training data
  train_spec = tf.estimator.TrainSpec(
      input_fn=read_dataset(
          filename=args["train_file_pattern"],
          mode=tf.estimator.ModeKeys.EVAL,  # read through train data once
          batch_size=args["train_batch_size"],
          params=args),
      max_steps=max_steps)
  # Check to see if we need to additionally tune principal components
  if not args["autotune_principal_components"]:
    # Create train and evaluate loop to train and evaluate our estimator
    tf.estimator.train_and_evaluate(
        estimator=estimator, train_spec=train_spec, eval_spec=eval_spec)
  else:
      if (args["k_principal_components_time"] is None or
          args["k_principal_components_feat"] is None):
        # Create train and evaluate loop to train and evaluate our estimator
        tf.estimator.train_and_evaluate(
            estimator=estimator, train_spec=train_spec, eval_spec=eval_spec)
```

- Since we have multiple training phases AND have to use train_and_evaluate, we must increment max_steps.

model.py

```python
# Calculate max_steps
max_steps = int(args["reconstruction_epochs"] * args["train_examples"])
max_steps = max_steps // args["train_batch_size"]
max_steps += args["previous_train_steps"]
```

```python
if args["model_type"] == "pca":
  # Create train spec to read in our training data
  train_spec = tf.estimator.TrainSpec(
      input_fn=read_dataset(
          filename=args["train_file_pattern"],
          mode=tf.estimator.ModeKeys.EVAL,  # read through train data once
          batch_size=args["train_batch_size"],
          params=args),
      max_steps=max_steps)
  # Check to see if we need to additionally tune principal components
  if not args["autotune_principal_components"]:
    # Create train and evaluate loop to train and evaluate our estimator
    tf.estimator.train_and_evaluate(
        estimator=estimator, train_spec=train_spec, eval_spec=eval_spec)
  else:
      if (args["k_principal_components_time"] is None or
          args["k_principal_components_feat"] is None):
        # Create train and evaluate loop to train and evaluate our estimator
        tf.estimator.train_and_evaluate(
            estimator=estimator, train_spec=train_spec, eval_spec=eval_spec)
```

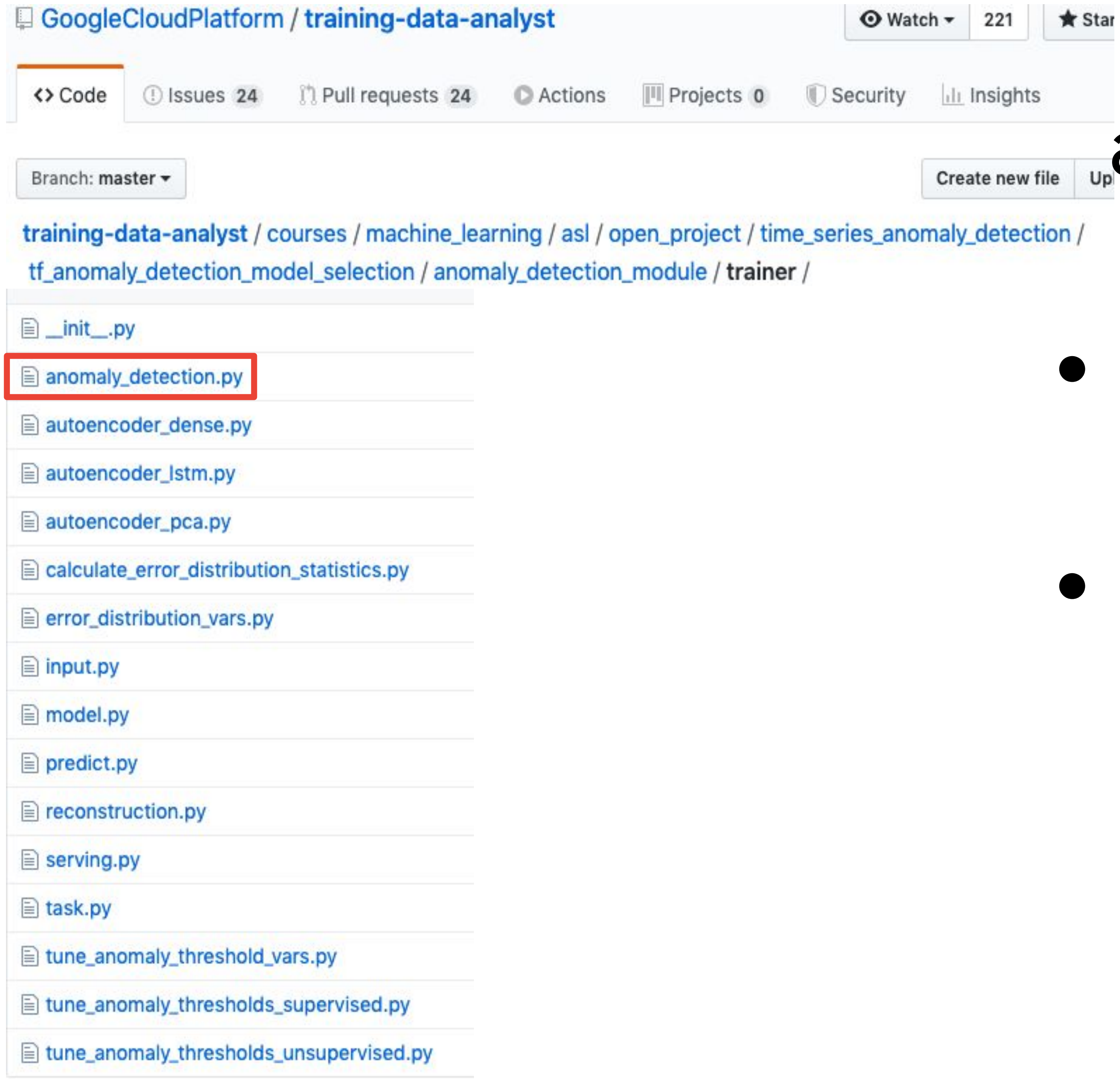
- For PCA, added a branch for the number of principal components to get automatically tuned in parallel. This saves us a possibly long hyperparameter tuning job.

model.py

```
# Create early stopping hook to help reduce overfitting
early_stopping_hook = tf.contrib.estimator.stop_if_no_decrease_hook(
    estimator=estimator,
    metric_name="rmse",
    max_steps_without_decrease=100,
    min_steps=1000,
    run_every_secs=60,
    run_every_steps=None)

# Create train spec to read in our training data
train_spec = tf.estimator.TrainSpec(
    input_fn=read_dataset(
        filename=args["train_file_pattern"],
        mode=tf.estimator.ModeKeys.TRAIN,
        batch_size=args["train_batch_size"],
        params=args),
    max_steps=max_steps,
    hooks=[early_stopping_hook])

# Create train and evaluate loop to train and evaluate our estimator
tf.estimator.train_and_evaluate(
    estimator=estimator, train_spec=train_spec, eval_spec=eval_spec)
```

- For non-PCA, added an early stopping hook to the TrainSpec to help combat overfitting.

anomaly_detection.py



- anomaly_detction.py contains the custom estimator EstimatorSpec.
- Essentially it ties everything together for our custom model function.

anomaly_detection.py

```python
# Create our model function to be used in our custom estimator
def anomaly_detection(features, labels, mode, params):
  """Custom Estimator model function for anomaly detection.
  Given dictionary of feature tensors, labels tensor, Estimator mode, and
  dictionary for parameters, return EstimatorSpec object for custom Estimator.
  Args:
    features: Dictionary of feature tensors.
    labels: Labels tensor or None.
    mode: Estimator ModeKeys. Can take values of TRAIN, EVAL, and PREDICT.
    params: Dictionary of parameters.
  Returns:
    EstimatorSpec object.
  """
  # Get input sequence tensor into correct shape
  # Get dynamic batch size in case there was a partially filled batch
  cur_batch_size = tf.shape(
      input=features[params["feat_names"][0]], out_type=tf.int64)[0]

  # Stack all of the features into a 3-D tensor
  # shape = (cur_batch_size, seq_len, num_feat)
  X = tf.stack(
      values=[features[key] for key in params["feat_names"]], axis=2)
```

- Get dynamic current batch size in case there is a partial batch.
- Create 3D features tensor X with shape [batch_size, seq_len, num_feat].

anomaly_detection.py

```python
# Important to note that flags determining which variables should be created
# need to remain the same through all stages or else they won't be in the
# checkpoint.

# Variables for calculating error distribution statistics
(...) = create_both_mahalanobis_dist_vars(...)

# Variables for automatically tuning anomaly thresh
if params["labeled_tune_thresh"]:
  (...) = create_both_confusion_matrix_thresh_vars(...)
else:
  (...) = create_both_mahalanobis_unsupervised_thresh_vars(...)

with tf.variable_scope(
    name_or_scope="mahalanobis_dist_thresh_vars", reuse=tf.AUTO_REUSE):
  time_anom_thresh_var = tf.get_variable(
      name="time_anom_thresh_var",
      dtype=tf.float64,
      initializer=tf.zeros(shape=[], dtype=tf.float64),
      trainable=False)

  feat_anom_thresh_var = tf.get_variable(
      name="feat_anom_thresh_var",
      dtype=tf.float64,
      initializer=tf.zeros(shape=[], dtype=tf.float64),
      trainable=False)

# Variables for tuning anomaly thresh evaluation
if params["labeled_tune_thresh"]:
  (...) = create_both_confusion_matrix_thresh_vars(...)

# Create dummy variable for graph dependency requiring a gradient for TRAIN
dummy_var = tf.get_variable(
    name="dummy_var",
    dtype=tf.float64,
    initializer=tf.zeros(shape=[], dtype=tf.float64),
    trainable=True)
```

- Create all variables that will be needed across the various phases in highest scope as possible.
- They will need to be accessible by all of the other submodules.

```python
# Important to note that flags determining which variables should be created
# need to remain the same through all stages or else they won't be in the
# checkpoint.

# Variables for calculating error distribution statistics
(...) = create_both_mahalanobis_dist_vars(...)

# Variables for automatically tuning anomaly thresh
if params["labeled_tune_thresh"]:
  (...) = create_both_confusion_matrix_thresh_vars(...)
else:
  (...) = create_both_mahalanobis_unsupervised_thresh_vars(...)

with tf.variable_scope(
    name_or_scope="mahalanobis_dist_thresh_vars", reuse=tf.AUTO_REUSE):
  time_anom_thresh_var = tf.get_variable(
      name="time_anom_thresh_var",
      dtype=tf.float64,
      initializer=tf.zeros(shape=[], dtype=tf.float64),
      trainable=False)

  feat_anom_thresh_var = tf.get_variable(
      name="feat_anom_thresh_var",
      dtype=tf.float64,
      initializer=tf.zeros(shape=[], dtype=tf.float64),
      trainable=False)

# Variables for tuning anomaly thresh evaluation
if params["labeled_tune_thresh"]:
  (...) = create_both_confusion_matrix_thresh_vars(...)

# Create dummy variable for graph dependency requiring a gradient for TRAIN
dummy_var = tf.get_variable(
    name="dummy_var",
    dtype=tf.float64,
    initializer=tf.zeros(shape=[], dtype=tf.float64),
    trainable=True)
```

anomaly_detection.py

- The only way to save variables to checkpoints is when mode == TRAIN.
- We don't want our variables to go through backprop.
- However, we MUST have a trainable variable for estimator to function.
- Hence dummy_var!

anomaly_detection.py

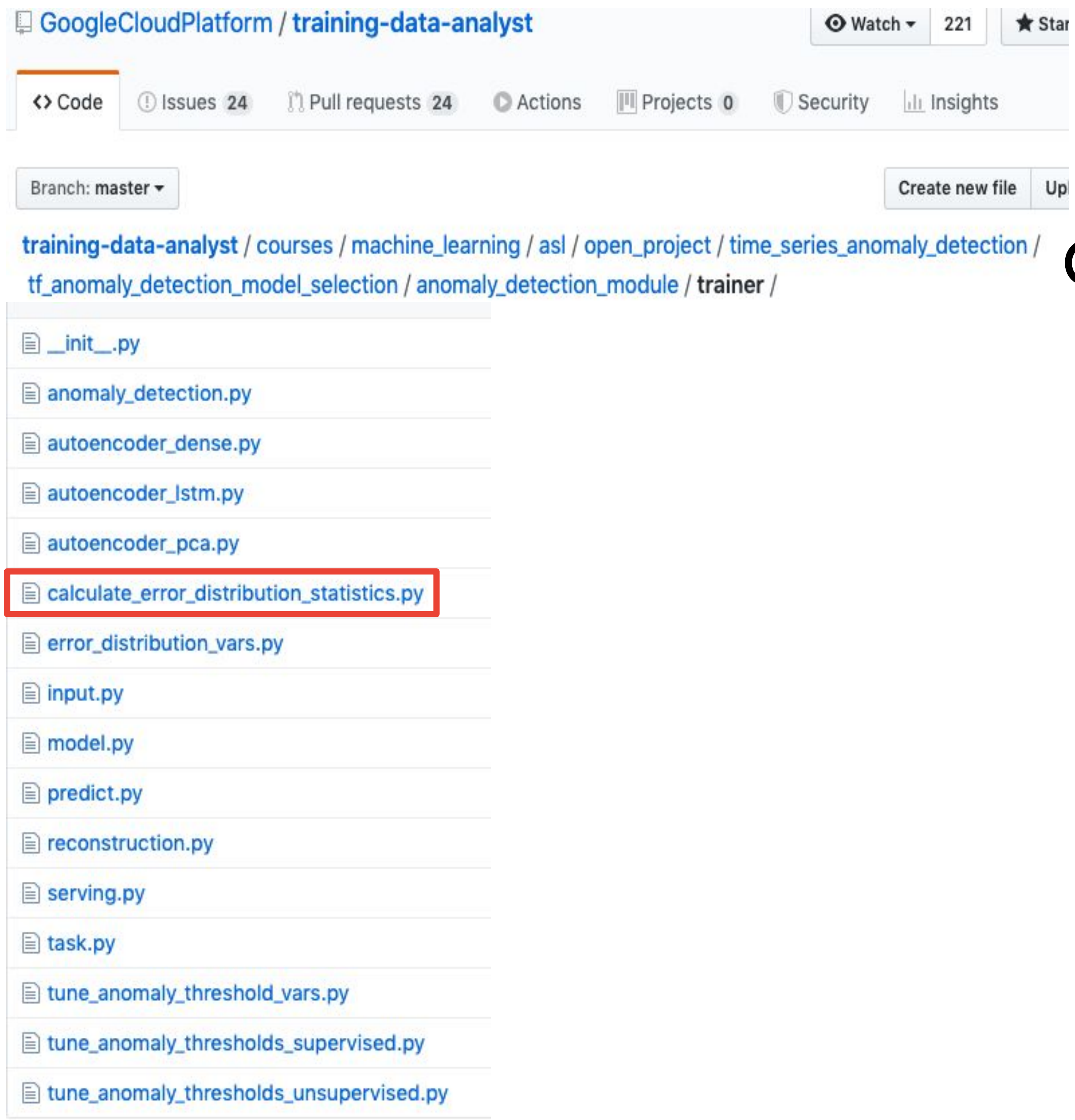
```python
predictions_dict = None
loss = None
train_op = None
eval_metric_ops = None
export_outputs = None

# Now branch off based on which mode we are in

# Call specific model
model_functions = {
    "dense_autoencoder": dense_autoencoder_model,
    "lstm_enc_dec_autoencoder": lstm_enc_dec_autoencoder_model,
    "pca": pca_model}

# Get function pointer for selected model type
model_function = model_functions[params["model_type"]]

# Build selected model
loss, train_op, X_time_orig, X_time_recon, X_feat_orig, X_feat_recon = \
  model_function(X, mode, params, cur_batch_size, dummy_var)
```

- Initialize EstimatorSpec parameters to None
- Build selected model function subgraph and connect it to main graph.
- So far all of this code applies to all model types.

```python
if not (mode == tf.estimator.ModeKeys.TRAIN and
        params["training_mode"] == "reconstruction"):
  # shape = (cur_batch_size * seq_len, num_feat)
  X_time_abs_recon_err = tf.abs(
      x=X_time_orig - X_time_recon)

  # Features based
  # shape = (cur_batch_size * num_feat, seq_len)
  X_feat_abs_recon_err = tf.abs(
      x=X_feat_orig - X_feat_recon)

  if (mode == tf.estimator.ModeKeys.TRAIN and
      params["training_mode"] == "calculate_error_distribution_statistics"):
    loss, train_op = calculate_error_distribution_statistics_training(...)
  elif (mode == tf.estimator.ModeKeys.EVAL and
        params["training_mode"] != "tune_anomaly_thresholds"):
    loss, eval_metric_ops = reconstruction_evaluation(
        X_time_orig, X_time_recon, params["training_mode"])
  elif (mode == tf.estimator.ModeKeys.PREDICT or
        ((mode == tf.estimator.ModeKeys.TRAIN or
          mode == tf.estimator.ModeKeys.EVAL) and
         params["training_mode"] == "tune_anomaly_thresholds")):
    with tf.variable_scope(
        name_or_scope="mahalanobis_dist_vars", reuse=tf.AUTO_REUSE):
      # Time based
      # shape = (cur_batch_size, seq_len)
      mahalanobis_dist_time = mahalanobis_dist(...)

      # Features based
      # shape = (cur_batch_size, num_feat)
      mahalanobis_dist_feat = mahalanobis_dist(...)

    if mode != tf.estimator.ModeKeys.PREDICT:
      if params["labeled_tune_thresh"]:
        labels_norm_mask = tf.equal(x=labels, y=0)
        labels_anom_mask = tf.equal(x=labels, y=1)

        if mode == tf.estimator.ModeKeys.TRAIN:
          loss, train_op = tune_anomaly_thresholds_supervised_training(...)
        elif mode == tf.estimator.ModeKeys.EVAL:
          loss, eval_metric_ops = tune_anomaly_thresholds_supervised_eval(...)
      else:  # not params["labeled_tune_thresh"]
        if mode == tf.estimator.ModeKeys.TRAIN:
          loss, train_op = tune_anomaly_thresholds_unsupervised_training(...)
        elif mode == tf.estimator.ModeKeys.EVAL:
          loss, eval_metric_ops = tune_anomaly_thresholds_unsupervised_eval(...)
    else:  # mode == tf.estimator.ModeKeys.PREDICT
      predictions_dict, export_outputs = anomaly_detection_predictions(...)

# Return EstimatorSpec
return tf.estimator.EstimatorSpec(...)
```

anomaly_detection.py

- Now the code does different things depending on the combination of the training_mode and tf.estimator.ModeKeys.



calculate_error_distribution_statistics.py

- Calculates the maximum likelihood estimation of the error distribution.
- Using running variables to track statistics.

calculate_error_distribution_statistics.py

```python
with tf.variable_scope(
    name_or_scope="mahalanobis_dist_vars", reuse=tf.AUTO_REUSE):
  # Time based
  singleton_time_condition = tf.equal(
      x=cur_batch_size * params["seq_len"], y=1)

  cov_time_var, mean_time_var, count_time_var = tf.cond(
      pred=singleton_time_condition,
      true_fn=lambda: singleton_batch_cov_variable_updating(
          params["seq_len"],
          X_time_abs_recon_err,
          abs_err_count_time_var,
          abs_err_mean_time_var,
          abs_err_cov_time_var),
      false_fn=lambda: non_singleton_batch_cov_variable_updating(
          cur_batch_size,
          params["seq_len"],
          X_time_abs_recon_err,
          abs_err_count_time_var,
          abs_err_mean_time_var,
          abs_err_cov_time_var))
```

- Use tf.cond to have multiple subgraphs.
- This handles if the batch is just a singleton which will require different statistics updating.
- Beware of returning dead tensors!

calculate_error_distribution_statistics.py

```python
  # Assign values to variables, use control dependencies around return to
  # enforce the mahalanobis variables to be assigned, the control order matters,
  # hence the separate contexts.
  with tf.control_dependencies(
      control_inputs=[tf.assign(ref=cov_variable, value=cov_tensor)]):
    with tf.control_dependencies(
        control_inputs=[tf.assign(ref=mean_variable, value=mean_tensor)]):
      with tf.control_dependencies(
          control_inputs=[tf.assign(ref=count_variable, value=count_tensor)]):

        return (tf.identity(input=cov_variable),
                tf.identity(input=mean_variable),
                tf.identity(input=count_variable))
```

- Control dependencies!
- Since there is nothing tying any of this to our EstimatorSpec, (i.e. loss) it will not be in graph dependency tree and hence won't get run!
- Also, order matters if you don't want an assignment to happen before another one.

calculate_error_distribution_statistics.py

```python
# Lastly use control dependencies around loss to enforce the mahalanobis
# variables to be assigned, the control order matters, hence the separate
# contexts
with tf.control_dependencies(
    control_inputs=[cov_time_var, cov_feat_var]):
  with tf.control_dependencies(
      control_inputs=[mean_time_var, mean_feat_var]):
    with tf.control_dependencies(
        control_inputs=[count_time_var, count_feat_var]):
      # Time based
      # shape = (num_feat, num_feat)
      abs_err_inv_cov_time_tensor = \
        tf.matrix_inverse(input=cov_time_var + \
          tf.eye(num_rows=tf.shape(input=cov_time_var)[0],
                 dtype=tf.float64) * params["eps"])
      # Features based
      # shape = (seq_len, seq_len)
      abs_err_inv_cov_feat_tensor = \
        tf.matrix_inverse(input=cov_feat_var + \
          tf.eye(num_rows=tf.shape(input=cov_feat_var)[0],
                 dtype=tf.float64) * params["eps"])

      with tf.control_dependencies(
          control_inputs=[tf.assign(ref=abs_err_inv_cov_time_var,
                                    value=abs_err_inv_cov_time_tensor),
                          tf.assign(ref=abs_err_inv_cov_feat_var,
                                    value=abs_err_inv_cov_feat_tensor)]):
        loss = tf.reduce_sum(
            input_tensor=tf.zeros(shape=(), dtype=tf.float64) * dummy_var)

        train_op = tf.contrib.layers.optimize_loss(
            loss=loss,
            global_step=tf.train.get_global_step(),
            learning_rate=params["learning_rate"],
            optimizer="SGD")

return loss, train_op
```

- More control dependencies!
- Remember, order matters!
- Notice the loss and train_op are encapsulated within the control dependencies.
- Notice dummy_var in the loss calculation to check the box for at least one trainable variable.

Run model module on GCP with unlabeled threshold tuning

```
import os
PROJECT = "PROJECT" # REPLACE WITH YOUR PROJECT ID
BUCKET = "BUCKET" # REPLACE WITH A BUCKET NAME
REGION = "us-central1" # REPLACE WITH YOUR REGION e.g. us-central1

# Import os environment variables
os.environ["PROJECT"] = PROJECT
os.environ["BUCKET"] =  BUCKET
os.environ["REGION"] = REGION
os.environ["TFVERSION"] = "1.13"
```

Copy data over to bucket

```
%%bash
gsutil -m cp -r data/* gs://$BUCKET/anomaly_detection/data
```

```
# Import os environment variables for global sequence shape hyperparameters
os.environ["SEQ_LEN"] = str(30)
os.environ["NUM_FEAT"] = str(5)

# Import os environment variables for global feature hyperparameters
os.environ["FEAT_NAMES"] = (",").join(["tag_{}".format(i) for i in range(int(os.environ["NUM_FEA
T"]))])
os.environ["FEAT_DEFAULTS"] = (",").join([(";").join(["0.0"] * int(os.environ["SEQ_LEN"]))] * int
(os.environ["NUM_FEAT"]))

# Import os environment variables for global training hyperparameters
os.environ["START_DELAY_SECS"] = str(60)
os.environ["THROTTLE_SECS"] = str(120)

# Import os environment variables for global threshold hyperparameters
os.environ["LABELED_TUNE_THRESH"] = "False"

# Import global dense hyperparameters
os.environ["ENC_DNN_HIDDEN_UNITS"] = "64,32,16"
os.environ["LATENT_VECTOR_SIZE"] = str(8)
os.environ["DEC_DNN_HIDDEN_UNITS"] = "16,32,64"
os.environ["TIME_LOSS_WEIGHT"] = str(1.0)
os.environ["FEAT_LOSS_WEIGHT"] = str(1.0)

# Import global lstm hyperparameters
os.environ["REVERSE_LABELS_SEQUENCE"] = "True"
os.environ["ENC_LSTM_HIDDEN_UNITS"] = "64,32,16"
os.environ["DEC_LSTM_HIDDEN_UNITS"] = "16,32,64"
os.environ["LSTM_DROPOUT_OUTPUT_KEEP_PROBS"] = "0.9,0.95,1.0"
os.environ["DNN_HIDDEN_UNITS"] = "1024,256,64"
```

Train reconstruction variables

```
# Import os environment variables for reconstruction training hyperparameters
os.environ["TRAIN_FILE_PATTERN"] = "gs://{}/anomaly_detection/data/train_norm_seq.csv".format(BUCK
ET)
os.environ["EVAL_FILE_PATTERN"] = "gs://{}/anomaly_detection/data/val_norm_1_seq.csv".format(BUCKE
T)
os.environ["PREVIOUS_TRAIN_STEPS"] = str(0)
os.environ["RECONSTRUCTION_EPOCHS"] = str(1.0)
os.environ["TRAIN_EXAMPLES"] = str(64000)
os.environ["LEARNING_RATE"] = str(0.1)
os.environ["TRAINING_MODE"] = "reconstruction"
```

Dense Autoencoder

```
%%bash
OUTDIR=gs://$BUCKET/anomaly_detection/trained_model/dense_unlabeled
JOBNAME=job_anomaly_detection_reconstruction_dense_unlabeled_$(date -u +%y%m%d_%H%M%S)
echo $OUTDIR $REGION $JOBNAME
gsutil -m rm -rf $OUTDIR
gcloud ml-engine jobs submit training $JOBNAME \
  --region=$REGION \
  --module-name=trainer.task \
  --package-path=$PWD/anomaly_detection_module/trainer \
  --job-dir=$OUTDIR \
  --staging-bucket=gs://$BUCKET \
  --scale-tier=STANDARD_1 \
  --runtime-version=1.13 \
  -- \
  --train_file_pattern=$TRAIN_FILE_PATTERN \
  --eval_file_pattern=$EVAL_FILE_PATTERN \
  --output_dir=$OUTDIR \
  --job-dir=./tmp \
  --seq_len=$SEQ_LEN \
  --num_feat=$NUM_FEAT \
  --feat_names=$FEAT_NAMES \
  --feat_defaults=$FEAT_DEFAULTS \
  --train_batch_size=32 \
  --eval_batch_size=32 \
  --previous_train_steps=$PREVIOUS_TRAIN_STEPS \
  --reconstruction_epochs=$RECONSTRUCTION_EPOCHS \
  --train_examples=$TRAIN_EXAMPLES \
  --learning_rate=$LEARNING_RATE \
  --start_delay_secs=$START_DELAY_SECS \
  --throttle_secs=$THROTTLE_SECS \
  --model_type="dense_autoencoder" \
  --enc_dnn_hidden_units=$ENC_DNN_HIDDEN_UNITS \
  --latent_vector_size=$LATENT_VECTOR_SIZE \
  --dec_dnn_hidden_units=$DEC_DNN_HIDDEN_UNITS \
  --time_loss_weight=$TIME_LOSS_WEIGHT \
  --feat_loss_weight=$FEAT_LOSS_WEIGHT \
  --training_mode=$TRAINING_MODE \
  --labeled_tune_thresh=$LABELED_TUNE_THRESH
```

Hyperparameter tuning of reconstruction hyperparameters

Dense Autoencoder ¶

```
%%writefile hyperparam_reconstruction_dense.yaml
trainingInput:
  scaleTier: STANDARD_1
  hyperparameters:
    hyperparameterMetricTag: rmse
    goal: MINIMIZE
    maxTrials: 30
    maxParallelTrials: 1
    params:
    - parameterName: enc_dnn_hidden_units
      type: CATEGORICAL
      categoricalValues: ["64 32 16", "256 128 16", "64 64 64"]
    - parameterName: latent_vector_size
      type: INTEGER
      minValue: 8
      maxValue: 16
      scaleType: UNIT_LINEAR_SCALE
    - parameterName: dec_dnn_hidden_units
      type: CATEGORICAL
      categoricalValues: ["16 32 64", "16 128 256", "64 64 64"]
    - parameterName: train_batch_size
      type: INTEGER
      minValue: 8
      maxValue: 512
      scaleType: UNIT_LOG_SCALE
    - parameterName: learning_rate
      type: DOUBLE
      minValue: 0.001
      maxValue: 0.1
      scaleType: UNIT_LINEAR_SCALE
```

Train error distribution variables

```
# Import os environment variables for error dist training hyperparameters
os.environ["TRAIN_FILE_PATTERN"] = "gs://{}/anomaly_detection/data/val_norm_1_seq.csv".format(BUCK
ET)
os.environ["EVAL_FILE_PATTERN"] = "gs://{}/anomaly_detection/data/val_norm_1_seq.csv".format(BUCKE
T)
os.environ["PREVIOUS_TRAIN_STEPS"] = str(2000)
os.environ["TRAIN_EXAMPLES"] = str(6400)
os.environ["TRAINING_MODE"] = "calculate_error_distribution_statistics"
os.environ["EPS"] = "1e-12"
```

Dense Autoencoder

```
%%bash
OUTDIR=gs://$BUCKET/anomaly_detection/trained_model/dense_unlabeled
JOBNAME=job_anomaly_detection_calculate_error_distribution_statistics_dense_unlabeled_$(date -u +%
y%m%d_%H%M%S)
echo $OUTDIR $REGION $JOBNAME
gcloud ml-engine jobs submit training $JOBNAME \
  --region=$REGION \
  --module-name=trainer.task \
  --package-path=$PWD/anomaly_detection_module/trainer \
  --job-dir=$OUTDIR \
  --staging-bucket=gs://$BUCKET \
  --scale-tier=STANDARD_1 \
  --runtime-version=1.13 \
  -- \
  --train_file_pattern=$TRAIN_FILE_PATTERN \
  --eval_file_pattern=$EVAL_FILE_PATTERN \
  --output_dir=$OUTDIR \
  --job-dir=./tmp \
  --seq_len=$SEQ_LEN \
  --num_feat=$NUM_FEAT \
  --feat_names=$FEAT_NAMES \
  --feat_defaults=$FEAT_DEFAULTS \
  --train_batch_size=32 \
  --eval_batch_size=32 \
  --previous_train_steps=$PREVIOUS_TRAIN_STEPS \
  --train_examples=$TRAIN_EXAMPLES \
  --start_delay_secs=$START_DELAY_SECS \
  --throttle_secs=$THROTTLE_SECS \
  --model_type="dense_autoencoder" \
  --enc_dnn_hidden_units=$ENC_DNN_HIDDEN_UNITS \
  --latent_vector_size=$LATENT_VECTOR_SIZE \
  --dec_dnn_hidden_units=$DEC_DNN_HIDDEN_UNITS \
  --time_loss_weight=$TIME_LOSS_WEIGHT \
  --feat_loss_weight=$FEAT_LOSS_WEIGHT \
  --training_mode=$TRAINING_MODE \
  --labeled_tune_thresh=$LABELED_TUNE_THRESH \
  --eps=$EPS
```

Tune anomaly thresholds

```
# Import os environment variables for tune threshold training hyperparameters
os.environ["PREVIOUS_TRAIN_STEPS"] = str(2200)
os.environ["TRAIN_EXAMPLES"] = str(12800)
os.environ["TRAINING_MODE"] = "tune_anomaly_thresholds"
```

Unlabeled

```
# Import os environment variables for unlabeled tune threshold training hyperparameters
os.environ["TRAIN_FILE_PATTERN"] = "gs://{}/anomaly_detection/data/unlabeled_val_mixed_seq.csv".fo
rmat(BUCKET)
os.environ["EVAL_FILE_PATTERN"] = "gs://{}/anomaly_detection/data/unlabeled_val_mixed_seq.csv".for
mat(BUCKET)
os.environ["TIME_THRESH_SCL"] = str(2.0)
os.environ["FEAT_THRESH_SCL"] = str(2.0)
```

Dense Autoencoder

```
%%bash
OUTDIR=gs://$BUCKET/anomaly_detection/trained_model/dense_unlabeled
JOBNAME=job_anomaly_detection_tune_anomaly_thresholds_dense_unlabeled_$(date -u +%y%m%d_%H%M%S)
echo $OUTDIR $REGION $JOBNAME
gcloud ml-engine jobs submit training $JOBNAME \
  --region=$REGION \
  --module-name=trainer.task \
  --package-path=$PWD/anomaly_detection_module/trainer \
  --job-dir=$OUTDIR \
  --staging-bucket=gs://$BUCKET \
  --scale-tier=STANDARD_1 \
  --runtime-version=1.13 \
  -- \
  --train_file_pattern=$TRAIN_FILE_PATTERN \
  --eval_file_pattern=$EVAL_FILE_PATTERN \
  --output_dir=$OUTDIR \
  --job-dir=./tmp \
  --seq_len=$SEQ_LEN \
  --num_feat=$NUM_FEAT \
  --feat_names=$FEAT_NAMES \
  --feat_defaults=$FEAT_DEFAULTS \
  --train_batch_size=32 \
  --eval_batch_size=32 \
  --previous_train_steps=$PREVIOUS_TRAIN_STEPS \
  --train_examples=$TRAIN_EXAMPLES \
  --start_delay_secs=$START_DELAY_SECS \
  --throttle_secs=$THROTTLE_SECS \
  --model_type="dense_autoencoder" \
  --enc_dnn_hidden_units=$ENC_DNN_HIDDEN_UNITS \
  --latent_vector_size=$LATENT_VECTOR_SIZE \
  --dec_dnn_hidden_units=$DEC_DNN_HIDDEN_UNITS \
  --time_loss_weight=$TIME_LOSS_WEIGHT \
  --feat_loss_weight=$FEAT_LOSS_WEIGHT \
  --training_mode=$TRAINING_MODE \
  --labeled_tune_thresh=$LABELED_TUNE_THRESH \
  --time_thresh_scl=$TIME_THRESH_SCL \
  --feat_thresh_scl=$FEAT_THRESH_SCL
```

Deploy

Dense Autoencoder

```
%%bash
MODEL_NAME="anomaly_detection_dense_unlabeled"
MODEL_VERSION="v1"
MODEL_LOCATION=$(gsutil ls gs://$BUCKET/anomaly_detection/trained_model/dense_unlabeled/export/exp
orter/ | tail -1)
echo "Deleting and deploying $MODEL_NAME $MODEL_VERSION from $MODEL_LOCATION ... this will take a
few minutes"
#gcloud ml-engine versions delete ${MODEL_VERSION} --model ${MODEL_NAME}
#gcloud ml-engine models delete ${MODEL_NAME}
gcloud ml-engine models create $MODEL_NAME --regions $REGION
gcloud ml-engine versions create $MODEL_VERSION --model $MODEL_NAME --origin $MODEL_LOCATION --run
time-version 1.13
```

Prediction

```
UNLABELED_CSV_COLUMNS = ["tag_{0}".format(tag) for tag in range(0, 5)]
```

```
import numpy as np
unlabeled_test_mixed_sequences_array = np.loadtxt(
    fname="data/unlabeled_test_mixed_seq.csv", dtype=str, delimiter=",")
print("unlabeled_test_mixed_sequences_array.shape = {}".format(
    unlabeled_test_mixed_sequences_array.shape))
```

```
unlabeled_test_mixed_sequences_array.shape = (12800, 5)
```

```
number_of_prediction_instances = 10
print("labels = {}".format(
  unlabeled_test_mixed_sequences_array[0:number_of_prediction_instances, -1]))
```

Dense Autoencoder

```python
# Send instance dictionary to receive response from ML-Engine for online prediction
from googleapiclient import discovery
from oauth2client.client import GoogleCredentials
import json

credentials = GoogleCredentials.get_application_default()
api = discovery.build("ml", "v1", credentials = credentials)

request_data = {"instances": instances}

parent = "projects/%s/models/%s/versions/%s" % (PROJECT, "anomaly_detection_dense_unlabeled", "v
1")
response = api.projects().predict(body = request_data, name = parent).execute()
print("response = {}".format(response))
```


```
response = {'predictions': [{'feat_anom_flags': 0, 'X_time_abs_recon_err': [[0.08630851, 0.0171070
3, 0.94729535, 0.39874107, 0.10912429], [2.37045866, 1.43010645, 2.04044194, 2.00116413, 1.2177937
4], [1.20743555, 0.40862645, 1.47843099, 0.31937228, 1.52916082], [1.14710842, 0.54163604, 0.16625
841, 0.16407445, 0.177806], [0.48609804, 2.06510263, 0.75320741, 1.40468418, 0.54337799], [1.50719
389, 0.00407659, 0.41766578, 0.85744544, 0.44055714], [2.19601683, 0.12560559, 1.258229, 0.9678330
1, 1.27923318], [0.10701992, 1.91115068, 1.52247393, 0.09800176, 1.18207574], [1.24018965, 0.08341
453, 0.72566737, 1.56246727, 0.84995686], [0.19057113, 0.99214653, 0.61741024, 0.24066002, 0.08906
497], [1.8352755, 1.59915954, 0.55178713, 0.20638952, 0.65940998], [1.752288, 0.90426161, 0.438672
16, 1.49479056, 1.61871863], [0.68841143, 1.14516381, 1.09453834, 0.8453935, 0.81842511], [1.25716
387, 0.92210796, 1.59920523, 0.65794037, 0.96566587], [0.38385502, 0.52287498, 0.34086948, 0.81768
388, 0.60382206], [2.6536265, 0.74294574, 0.8572519, 1.53697035, 1.69003285], [0.79187743, 0.70845
168, 0.9924474, 0.60762906, 1.66798859], [1.04243811, 1.03833225, 0.89896643, 0.41178175, 0.316967
74], [0.52800537, 0.4001544, 1.69688929, 1.97426229, 0.408903], [1.76135306, 0.95978758, 1.060538
1, 1.27885467, 0.25624363], [2.42560225, 1.08263277, 0.45109315, 0.99065918, 1.39733346], [0.47596
221, 0.31584913, 0.96045705, 0.75421874, 1.38432466], [0.93880634, 0.00862613, 0.59540668, 2.28180
555, 0.19920215], [0.34620397, 1.30877146, 1.24977148, 0.0776754, 0.79272972], [1.77331762, 0.2204
8323, 2.10291277, 0.52754049, 0.38229627], [1.74868177, 0.24728125, 0.88623454, 2.00615088, 1.1370
7338], [0.03435227, 1.4809888, 0.87614424, 0.92636092, 1.69781059], [0.88146609, 0.20183649, 0.545
96172, 1.06403935, 0.19309917], [0.34844452, 0.16185678, 0.32117828, 0.65159031, 0.97771496], [2.2
8759207, 1.24482918, 1.82773091, 2.25599744, 0.44386982]], 'X_feat_abs_recon_err': [[0.08630851,
```

cloud.google.com