Machine Learning - Enterprise Ready: Part II

Coding for ML use cases

Repeated content is inevitable and intended

Resources:

The code for this lecture (https://github.com/smurve/HSR2019)

Academic References

TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems, Abadi et al 2016 (https://arxiv.org/pdf/1603.04467.pdf)

TensorFlow Estimators:..., Cheng et al 2017 (https://arxiv.org/pdf/1708.02637.pdf)

Popular References

Blog: Framework Comparison (TF, Theano, Keras, DL4J, and others), towardsdatascience.com (https://towardsdatascience.com/deep-learning-framework-power-scores-2018-23607ddf297a)

Tensorflow Documentation (https://tensorflow.org)

What happened before

Data Engineering is Software Engineering

Architectural View on ML in the Enterprise

Parallelize with Computational Graphs

Storing and Retrieving Terabytes

ML Engineering: Data for the Data Scientist

The Estimator Concept

Agenda

1) Requirements Engineering

1) HPE: High Performance Engineering

2) Apache Beam Programming Model

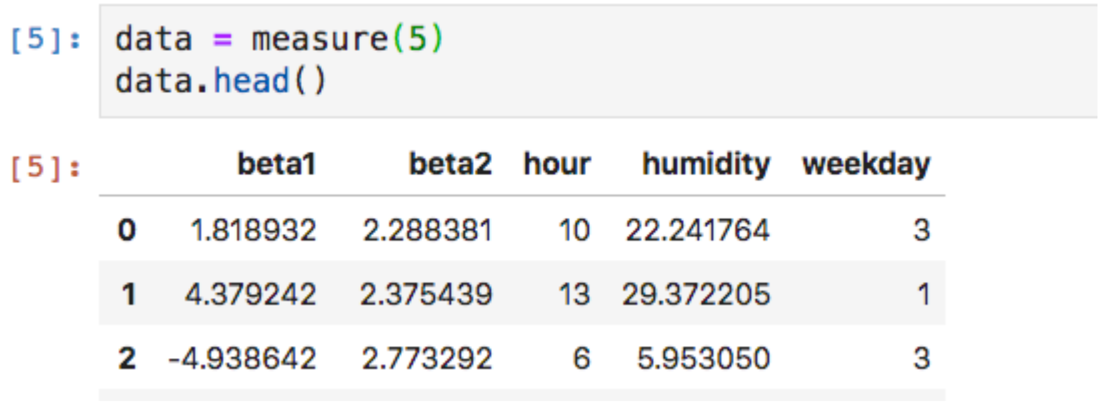
3) TF Transform

4) Ingesting data - fast

5) The Estimator Concept

The Return of the Baking Powder Machine

In the previous exercise...



[5]:
```
data = measure(5)
data.head()
```

[5]:

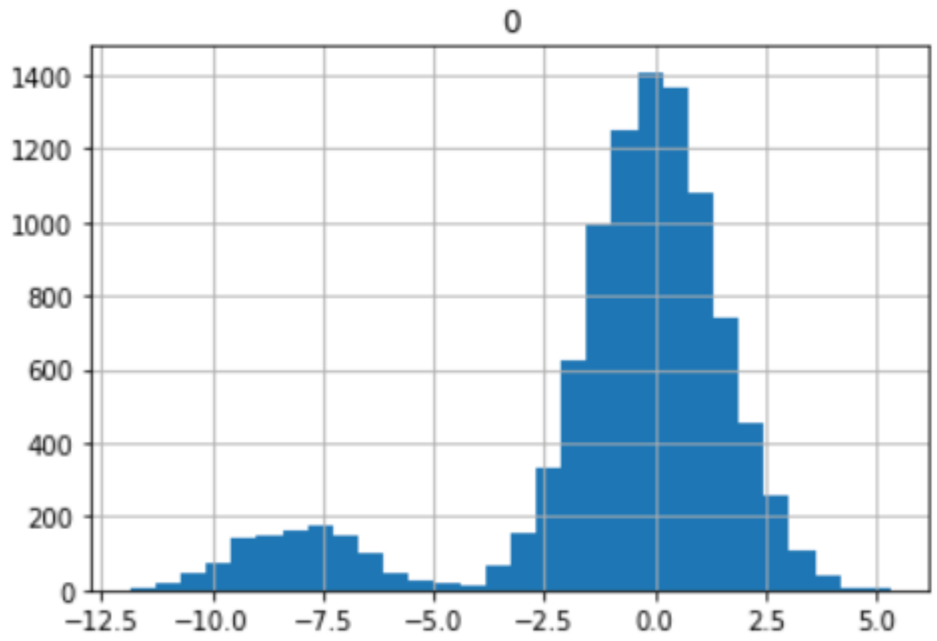
	beta1	beta2	hour	humidity	weekday
0	1.818932	2.288381	10	22.241764	3
1	4.379242	2.375439	13	29.372205	1
2	-4.938642	2.773292	6	5.953050	3

We tried the hyptothesis:

$$h = A_1 \cdot \beta_1 + A_2 \cdot \beta_2 + C$$

… we failed to explain the data

```
[32]: pd.DataFrame(errors[0]).hist(bins=30);
```



Requirements Engineering

Build a high-performance training application for the data scientists' model

F1: Provide the input data at high speed in the desired 170-dimensional format

F2: Reuse transformations from the preprocessing pipeline

F3: Monitor performance as the training continues

F4: Use save points to protect valuable intermediate results

F5: Provide a simple interface to the model (hypothesis)

High Performance Engineering

Mama, look: No for-loops!

Pre-compute and lookup

Use Hardware efficiently with dedicated libraries

Program in computational graphs that can be executed anywhere

Mama look: No for-loops

See collateral/No_For_Loops.ipynb (collateral/No_For_Loops.ipynb) for more.

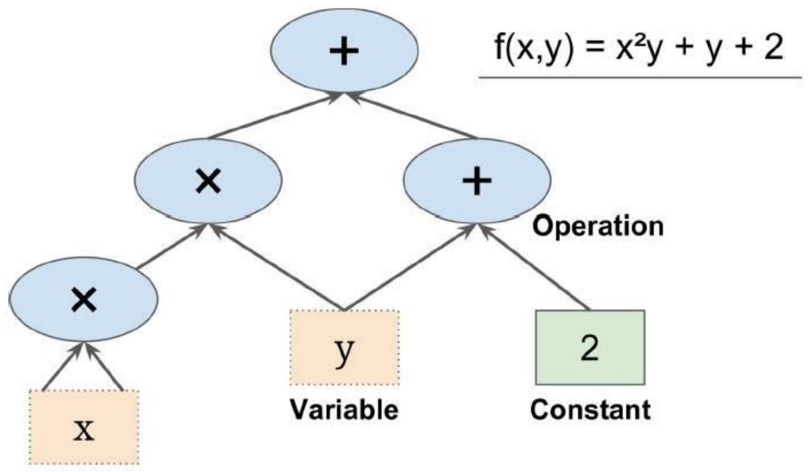
The classical approach:

```
def count_num_samples_with_row_of_3_with_forloop(samples):
    sum = 0
    for sample in samples:
        for r in range(3):
            for c in range(3):
                if sample[r][c]==1 and sample[r+1][c+1]==1 and sample[r+2][c+2
]== 1:
                    sum+=1
    return sum
```

A super-fast one-liner

```
np.sum(np.matmul(detector, np.transpose(np.reshape(samples, [2000, 25])))==3)
```

Computational Graphs



Tensors and Graphs of Tensors

collateral/Tensors_Graphs_Sessions.ipynb (collateral/Tensors_Graphs_Sessions.ipynb)

Structural elements

`Placeholder` s take regular numbers and arrays as input for execution (x)

`Constant` s represent numbers that are known before execution time.

`Variable` s can be changed during graph execution

All operators create operator nodes, rather than execute directly

`tf.gradient(...)` provides means for gradient computations.

`Graph` s represent the structure of a subset of tensors.

`Session` s represent the *current* state of a `Graph` .

Tensorflow Code Example

```python
import tensorflow as tf
x = tf.placeholder(shape=(None,1), dtype=tf.float32)
a = tf.Variable([[.5]], name="weights", dtype=tf.float32)
b = tf.Variable([[-2.]], name="bias", dtype=tf.float32)
y = tf.matmul(x, a) + b
init = tf.global_variables_initializer()
with tf.Session() as sess:
    sess.run(init)
    print(sess.run(y, feed_dict={x: [[2.0], [4.0]]}))
```

Data Processing Pipelines

Built for highly optimized parallel execution of massive workloads

Apache Beam somewhat de facto standard

Alternatives: Spark, Storm, ...

Same interface in batch and real-time (only Java) mode.

Functional Semantics: `Map(...)` and `FlatMap(...)`

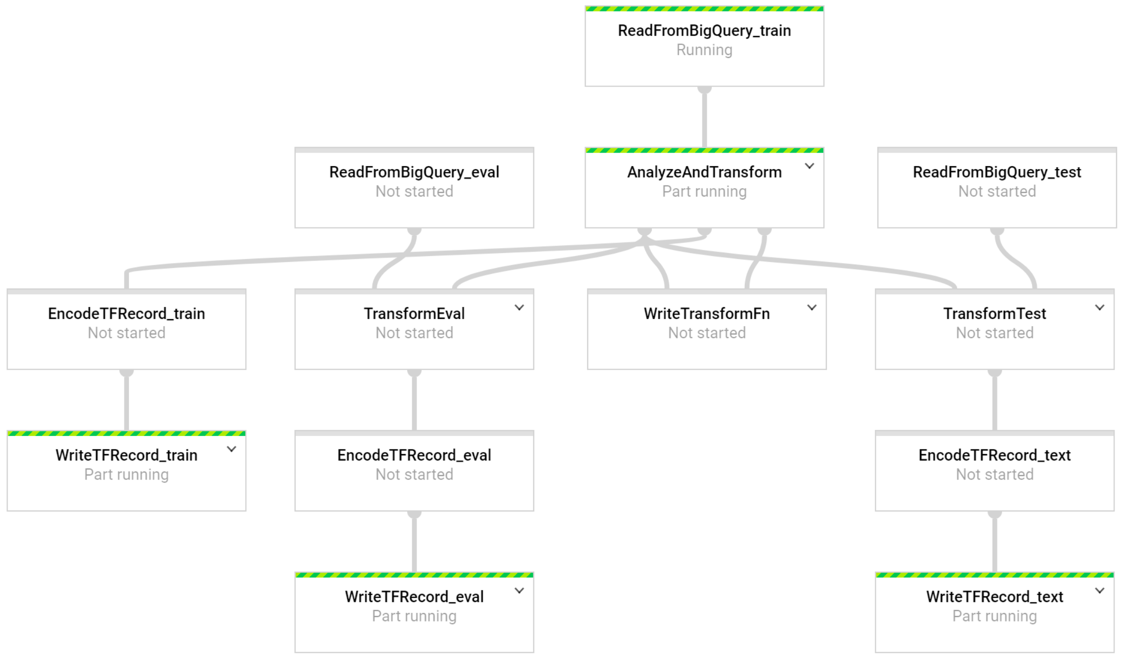
Nodes must produce serializable output

Pipelines support *fork* and *join* architectures.

Apache Beam Pipeline Code

```
with beam.Pipeline('DirectRunner', PipelineOptions()) as p:

    csv_encoder = tft.coders.CsvCoder(ORDERED_SIGNATURE_COLUMNS, schema)

    _ = (p
         | 'read_from_csv' >> beam.io.ReadFromText(
             file_pattern=signature_csv_train, coder=csv_encoder, skip_header_
lines=1)

         | 'process_records' >> beam.Map(process_data)

         | 'write_to_csv' >> beam.io.WriteToText(
             file_path_prefix=training_csv, coder=csv_encoder, header=header)
        )
```

A Production Beam Pipeline in action



Analyze and Transform

Scaling requires first evaluating min_k and max_k

And then, in a second run, compute

$$\beta'_{i,k} = \frac{\beta_{i,k} - min_k(\beta_{i,k})}{max_k(\beta_{i,k}) - min_k(\beta_{i,k})}$$

The `tf.transform` library achieves all of that with a single line of code:

```
def process_data(row):
    for c in ['beta1', 'beta2']:
        row[c] = tft.scale_to_0_1(row[c])
    return row
```

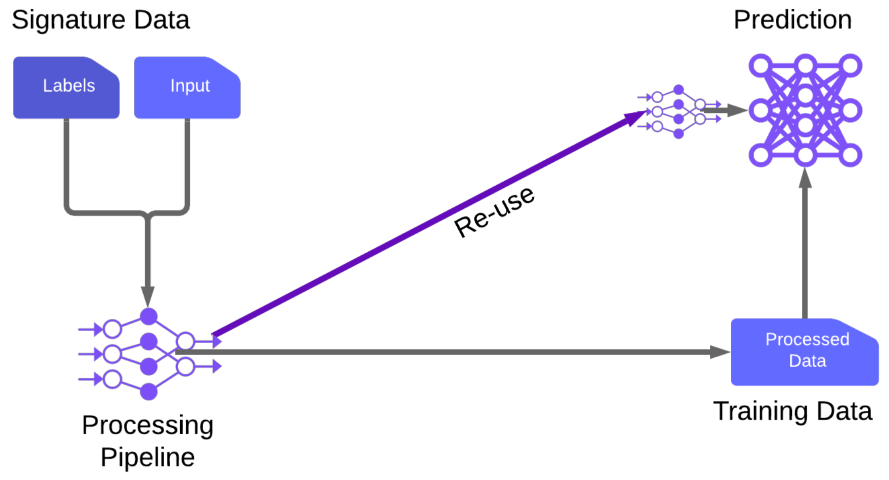
Re-use the transform function

The transform function can be saved for re-use at prediction time:

```
data_and_metadata, transform_fn = (
    signature_data | "AnalyzeAndTransform"
    >> beam_impl.AnalyzeAndTransformDataset(process_data))

#
# Eventually, save the transform function for re-use at prediction time.
#
_ = (transform_fn | 'WriteTransformFn'
     >> transform_fn_io.WriteTransformFn(metadata_dir))
```

Signature vs Training Stage

- Reproduce all pre-processing steps during prediction!
- Failure leads to "training-serving skew"



Signature vs Training stage

Signature data is what comes during prediction time

It obeys the interface signature of the prediction service

Training data is pre-processed to facilitate effective training

The differences must be carefully dealt with

Failure to do so results in the so-called *training-serving skew*

Requirements for Input Functions

Process any number of files

Create a continuous stream of decoded records

Repeat the data stream (epochs)

Shuffle the data to stabilize learning

Split the data in efficient batch sizes

Automatically iterate over those batches

Prefetch data, use multiple threads in parallel

Use frameworks for infrastructure requirements

See: [collateral/InputFunctions.ipynb (collateral/InputFunctions.ipynb)]

```
def _input_fn():
    dataset = tf.data.experimental.make_batched_features_dataset(
        file_pattern=filename_pattern,
        batch_size=batch_size,
        features=feature_spec,
        shuffle_buffer_size=options['shuffle_buffer_size'],
        prefetch_buffer_size=options['prefetch_buffer_size'],
        reader_num_threads=options['reader_num_threads'],
        parser_num_threads=options['parser_num_threads'],
        sloppy_ordering=options['sloppy_ordering'],
        num_epochs=options['num_epochs'],
        label_key='humidity')

    return dataset.make_one_shot_iterator().get_next()
```

Feature Engineering

collateral/Feature_Engineering.ipynb (collateral/Feature_Engineering.ipynb)

Data in files are not always ideally encoded for ML

Categorical data has to be transformed to numerical data

Days and hours are best *one-hot* encoded

Feature crosses help detect non-trivial dependencies (e.g. hour of week)

Embeddings help reduce dimensions for large sparse input

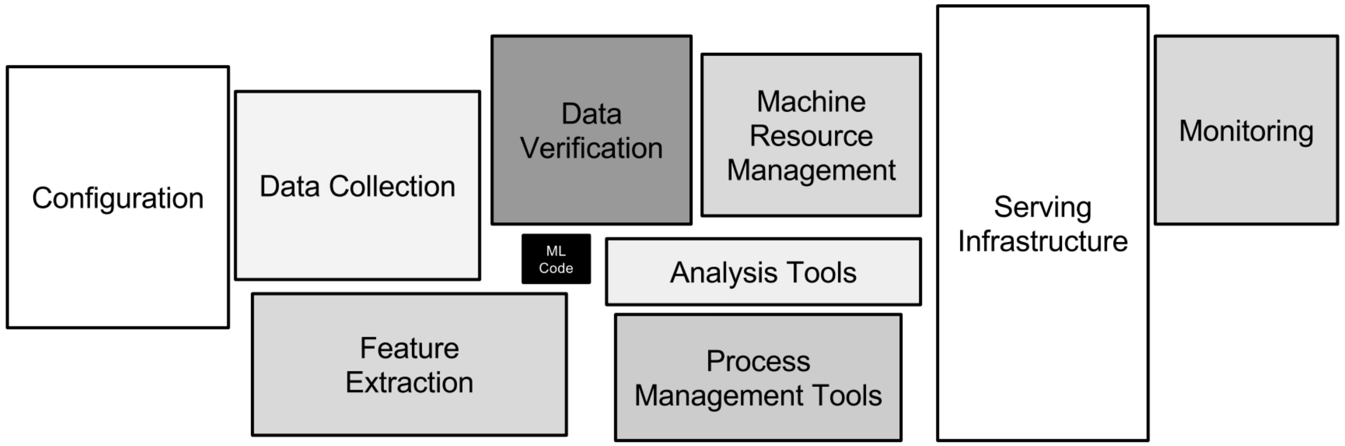
These re-encodings trade memory or speed for effectiveness

Creating an input layer for the model

`input_layer`: the x-interface to the data scientist's work

```
weekday = categorical_column_with_identity('weekday', num_buckets=7)
hour = categorical_column_with_identity('hour', num_buckets=24)
hour_of_week = indicator_column(crossed_column([weekday, hour], 24*7))

all_feature_columns = [beta1, beta2, hour_of_week]

input_layer = tf.feature_column.input_layer(
    features,
    feature_columns=[beta1, beta2, hour_of_week])
```

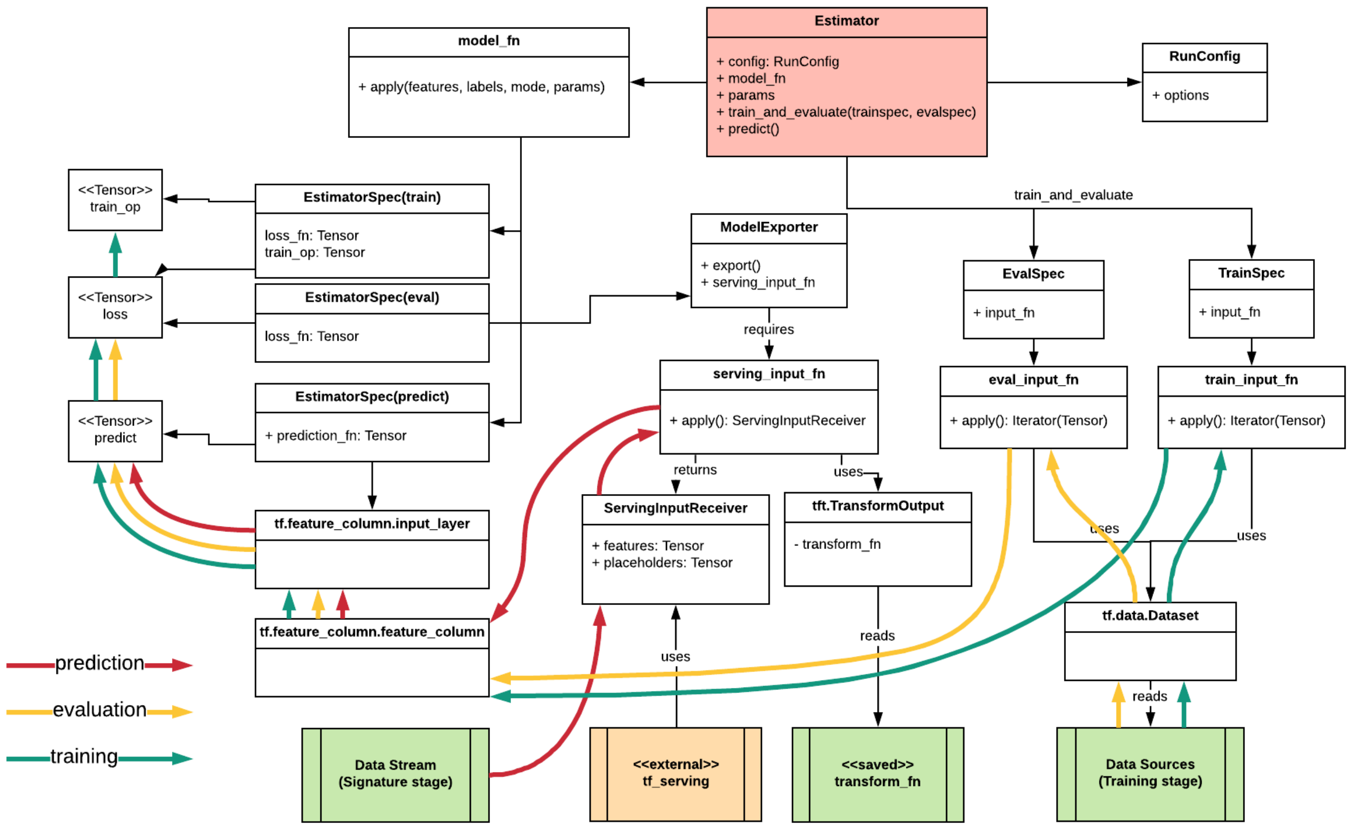
Entering the *black box* of ML



From 4 numbers per record create 170

```
array([[0.8050443 , 0.8593288 , 0.        , 0.        , 1.        ,
        0.        , 0.        , 0.        , 0.        , 0.        ,
        0.        , 0.        , 0.        , 0.        , 0.        ,
        0.        , 0.        , 0.        , 0.        , 0.        ,
        0.        , 0.        , 0.        , 0.        , 0.        ,
        0.        , 0.        , 0.        , 0.        , 0.        ,
        0.        , 0.        , 0.        , 0.        , 0.        ,
        0.        , 0.        , 0.        , 0.        , 0.        ,
        0.        , 0.        , 0.        , 0.        , 0.        ,
        0.        , 0.        , 0.        , 0.        , 0.        ,
        0.        , 0.        , 0.        , 0.        , 0.        ,
        0.        , 0.        , 0.        , 0.        , 0.        ,
        0.        , 0.        , 0.        , 0.        , 0.        ,
        0.        , 0.        , 0.        , 0.        , 0.        ,
        0.        , 0.        , 0.        , 0.        , 0.        ,
        0.        , 0.        , 0.        , 0.        , 0.        ,
        0.        , 0.        , 0.        , 0.        , 0.        ,
        0.        , 0.        , 0.        , 0.        , 0.        ,
        0.        , 0.        , 0.        , 0.        , 0.        ,
        0.        , 0.        , 0.        , 0.        , 0.        ,
```

Tensorflow: Programming model and data flow



Tensorflow Estimator

The estimator manages graph, session, checkpoints, logging and lifecycle

The estimator MUST create all tensors in its own context

We provide functions that create tensors - for the estimator to call

- We provide a model function (or maybe, the data scientist)
- We provide input functions
- We provide specifications for the lifecycle
- We provide a general configuration
- We provide an exporter that saves the entire graph (incl. transform functions!)

The model function

```
def model_function(features, labels, mode):

    my_input_layer = input_layer(features)
    linreg = tf.layers.Dense(name="LinReg", units=1)
    hypothesis =linreg(my_input_layer)

    #
    # For predictions, we just need the hypothesis.
    #
    if mode == tf.estimator.ModeKeys.PREDICT:
        return tf.estimator.EstimatorSpec(
            tf.estimator.ModeKeys.PREDICT,
            predictions=hypothesis)
```

The model function for training

```
...
    optimizer = tf.train.AdamOptimizer(learning_rate=1e-0)
    train_op = optimizer.minimize(loss,...)

    return tf.estimator.EstimatorSpec(
        tf.estimator.ModeKeys.TRAIN,
        loss = loss,
        train_op = train_op)
    ...
```

Construct the estimator

```
config = RunConfig(
    model_dir               = model_dir,
    save_summary_steps      = 1,
    save_checkpoints_steps  = 100,
    log_step_count_steps    = 10)

estimator = tf.estimator.Estimator(
    config=config,
    model_fn=model_function)
```

Let it train

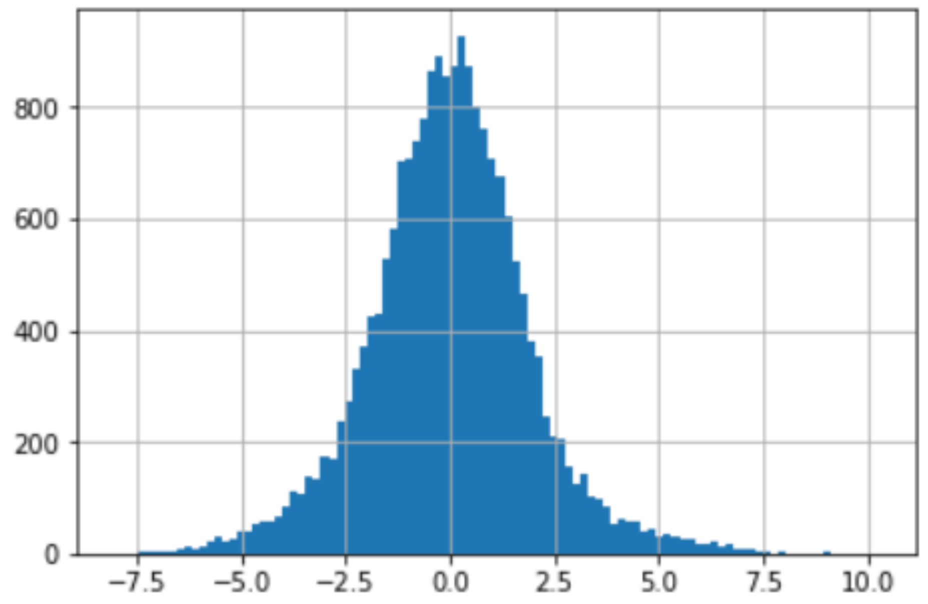
```
train_spec = tf.estimator.TrainSpec(
    input_fn=train_input_fn,
    max_steps=max_steps)

...

tf.estimator.train_and_evaluate(
    estimator,
    train_spec=train_spec,
    eval_spec=eval_spec)
```

Using the trained model

```
estimator = tf.contrib.predictor.from_saved_model(latest_model)

sample = {
    'beta1': [[1.234],[1.234]],
    'beta2': [[1.234],[1.234]],
    'weekday': [[5], [6]],
    'hour': [[16], [17]]
}

result = estimator(sample)
```

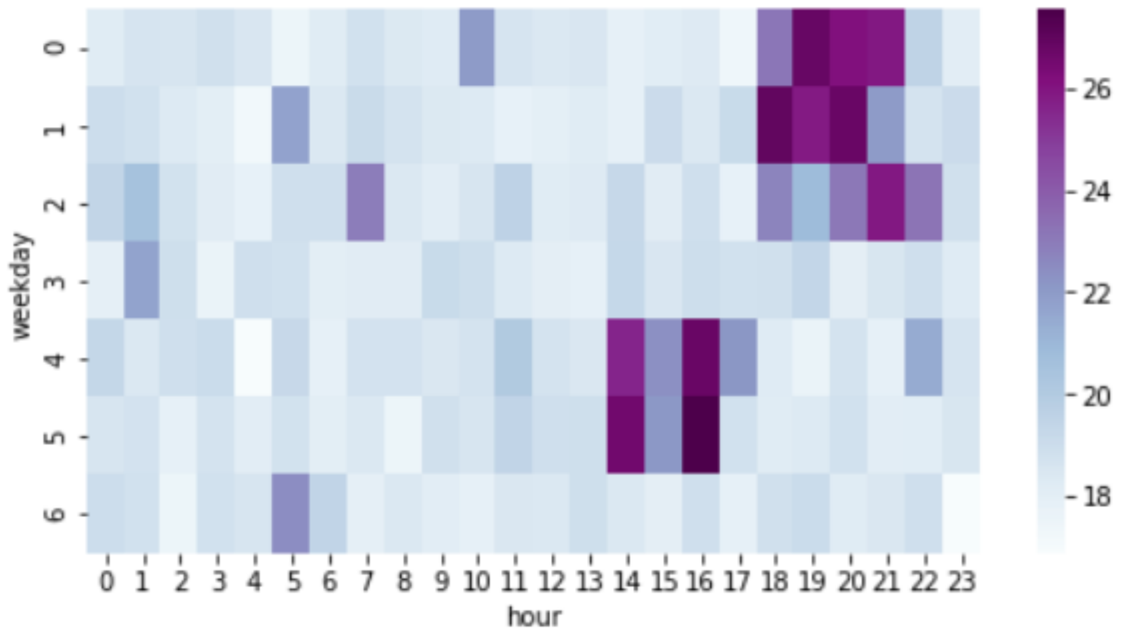
Now, we can explain the data

```
[187]: test_data['diff'].hist(bins=100);
```



The model is able to predict the anomalies

```
[26]: from matplotlib import pyplot as plt
      plt.figure(figsize=(8,4))
      sns.heatmap(test_data.pivot_table(
          index='weekday', columns='hour',
          values='predicted', aggfunc='mean'), cmap='BuPu');
```


weekday
hour