Lagrangian Approaches for Knowledge Injection

When soft constraints are enough

Constrained Machine Learning

Let's consider ML problem with constrained output

In particular, let's focus on problems in the form:

$$\text{argmin}_\theta \left\{ L(y) \mid y = f(\hat{x}, \omega), g(y) \leq 0 \right\}$$

Where:

- L is the loss
- $\hat{x}$ is the training input
- y is the ML model output, i.e. $f(x, \omega)$
- ω is the parameter vector (we assume a parameterized model)
- g is a constraint function

Equality csts. can be viewed as double-inequalities (but they admit simplifications)

Constrained Machine Learning

Example 1: logical rules

E.g. hiearchies in multi-class classification ("A dog is also an animal"):

$$y_{i,dog} \leq y_{i,animal}$$

- This constraint is defined over individual examples

Example 2: shape constraints

E.g. input x_j cannot cause the output to decrease (monotonicity)

$$y_i \leq y_k \quad \forall i, k : x_{i,j} \leq x_{k,j} \wedge x_{i,h} = x_{k,h} \forall h \neq j$$

- This is a relational constraint, i.e. defined over multiple examples

Lagrangian Relaxations for Constrained ML

One way to deal with this problem is to rely on a Lagrangian Relaxation

Main idea: we turn the constraints into penalty terms:

$$\text{argmin}_\theta \left\{ L(y) + \lambda^T \max(0, g(y)) \mid y = f(\hat{x}, \omega) \right\}$$

Or, alternatively:

$$\text{argmin}_\theta \left\{ L(y) + \lambda^T \max(0, g(y))^2 \mid y = f(\hat{x}, \omega) \right\}$$

- We use a vector of multipliers λ to weight the constraint violations

This is a popular approach for ML with constraints

- One of the first occurrences as Semantic Based Regularization
- The constraints are "distilled" in the model parameters

Lagrangian Relaxations for Constrained ML

Equality constraints can be treated as double inequalities

...Or via the simplified form:

$$\mathop{\arg\min}_{\theta} \left\{ L(\mathbf{y}) + \lambda^T g(\mathbf{y})^2 \right\} \text{ with: } \mathbf{y} = f(\mathbf{x}; \theta)$$

There are a few big caveats, in particular:

- The degree of constraint satisfaction depends on the multipliers
- There is no trivial guarantee that constraints are eventually satisfied

...But neither point matters if we just want to inject some expert knowledge in a ML model

A Case Study: Scarce Labels in Predictive Maintenance

Scarce Labels in RUL Predictions

RUL estimation is a major goal for predictive maintenance

However, ground truth for RUL is hard to come by:

- Run-to-failure experiments are time consuming
- They may not be viable for large and complex machines

Typically, only a few runs are available

However, data about normal operation may still be abundant

- This may come from test runs, installed machines, etc.
- It looks exactly like the input data for our RUL prediction model
- ...And it will still show sign of component wear

However, the true RUL value in this case will be unknown

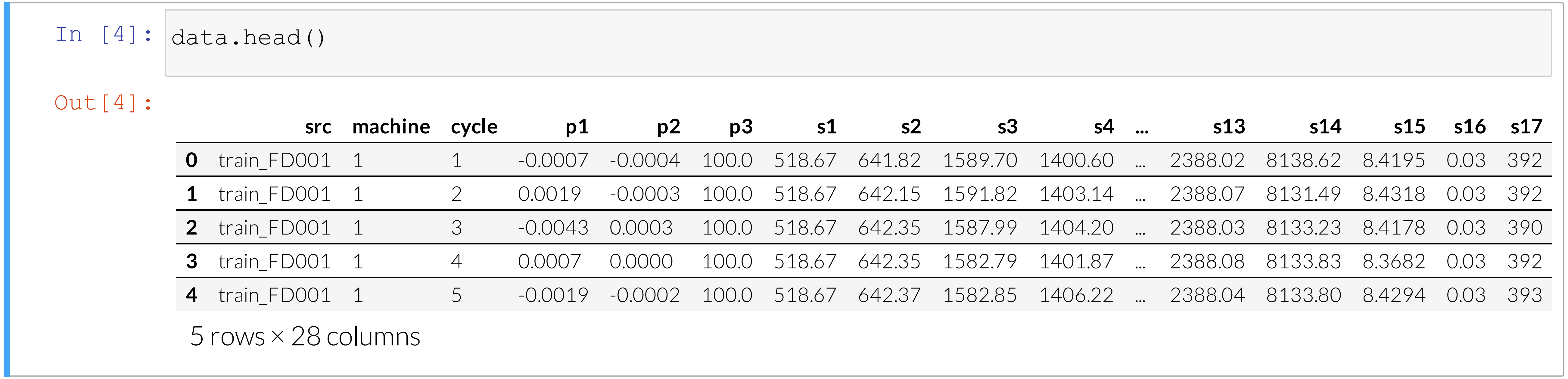
Can we still take advantage of this data?

Our Baseline Approach

Data Loading and Preparation

We will rely on the NASA C-MAPPS dataset

...Which contains simulated run-to-failure experiments for turbo-fan engines



In [4]:

```
data.head()
```

Out[4]:

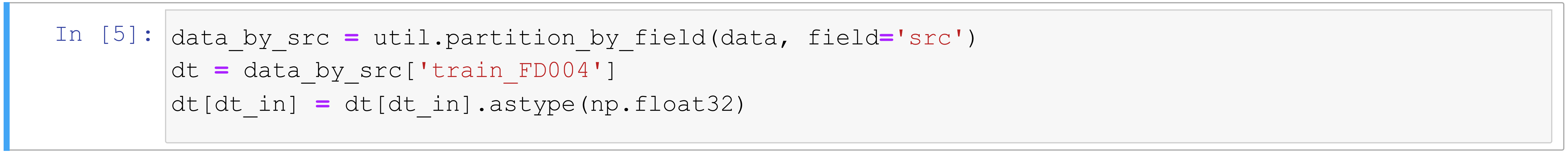
	src	machine	cycle	p1	p2	p3	s1	s2	s3	s4	...	s13	s14	s15	s16	s17
0	train_FD001	1	1	-0.0007	-0.0004	100.0	518.67	641.82	1589.70	1400.60	...	2388.02	8138.62	8.4195	0.03	392
1	train_FD001	1	2	0.0019	-0.0003	100.0	518.67	642.15	1591.82	1403.14	...	2388.07	8131.49	8.4318	0.03	392
2	train_FD001	1	3	-0.0043	0.0003	100.0	518.67	642.35	1587.99	1404.20	...	2388.03	8133.23	8.4178	0.03	390
3	train_FD001	1	4	0.0007	0.0000	100.0	518.67	642.35	1582.79	1401.87	...	2388.08	8133.83	8.3682	0.03	392
4	train_FD001	1	5	-0.0019	-0.0002	100.0	518.67	642.37	1582.85	1406.22	...	2388.04	8133.80	8.4294	0.03	393

5 rows × 28 columns

- There are four sub-datasets (column `src`)
- Columns `p1-3` represent control parameters
- Columns `s1-21` are sensor readings

Data Loading and Preparation

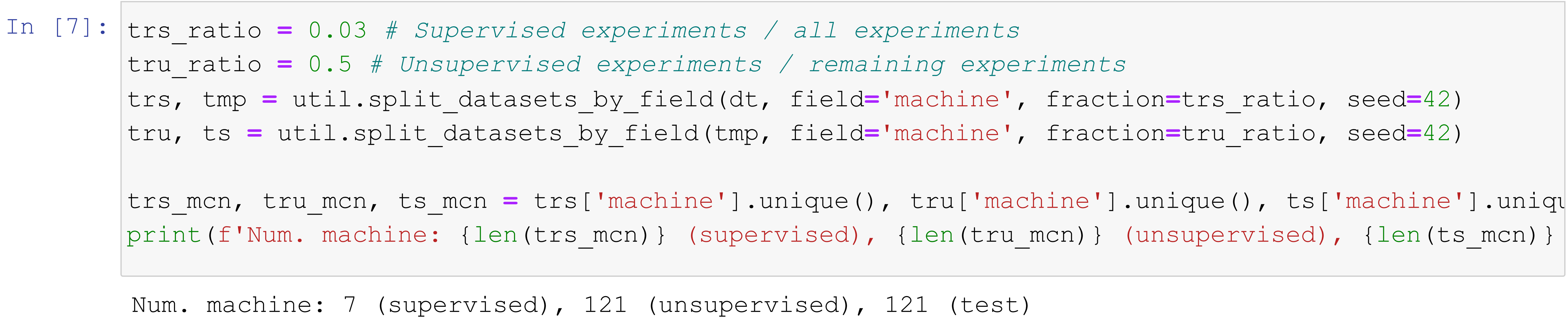
We will focus on the FD004 dataset (the hardest)



```
In [5]: data_by_src = util.partition_by_field(data, field='src')
        dt = data_by_src['train_FD004']
        dt[dt_in] = dt[dt_in].astype(np.float32)
```

Then we separate two sets for training and one for testing

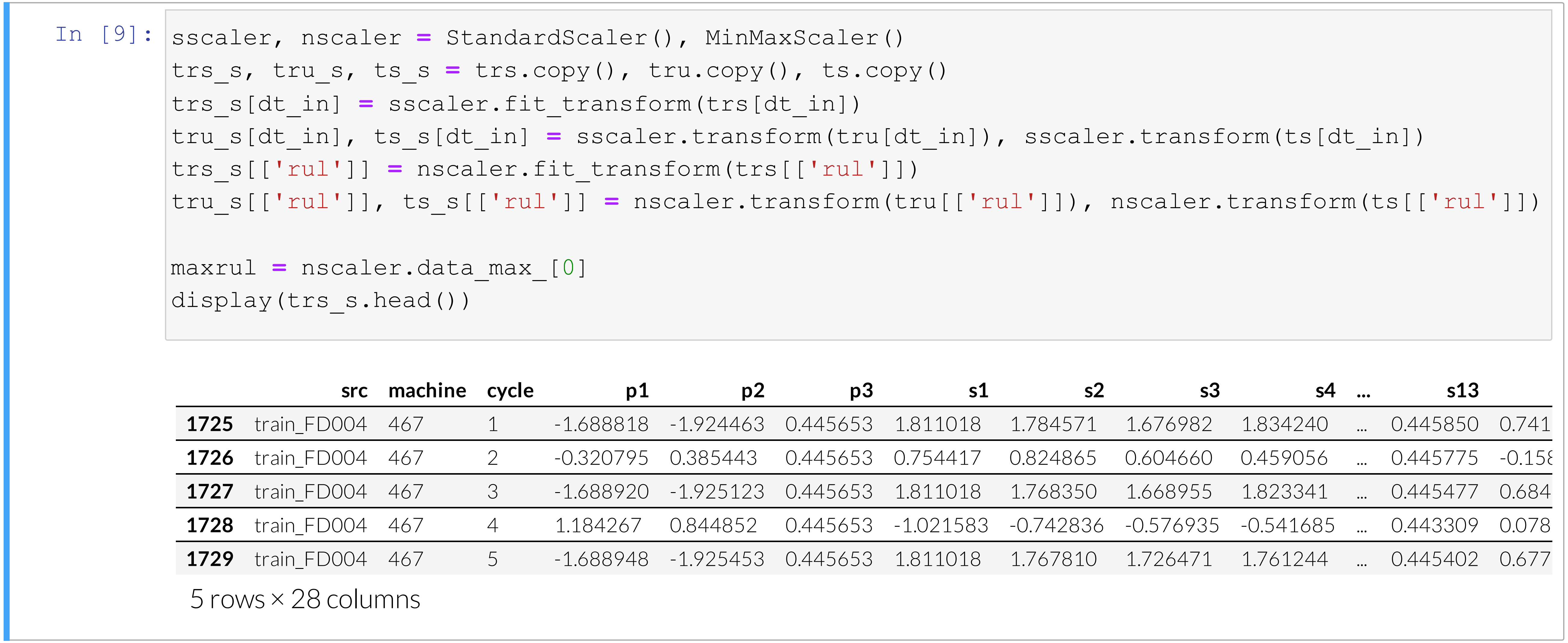
- The first trainign set will contain finished experiments (supervised)
- ...The second will contain data for still running machines (unsupervised)



```
In [7]: trs_ratio = 0.03 # Supervised experiments / all experiments
        tru_ratio = 0.5 # Unsupervised experiments / remaining experiments
        trs, tmp = util.split_datasets_by_field(dt, field='machine', fraction=trs_ratio, seed=42)
        tru, ts = util.split_datasets_by_field(tmp, field='machine', fraction=tru_ratio, seed=42)

        trs_mcn, tru_mcn, ts_mcn = trs['machine'].unique(), tru['machine'].unique(), ts['machine'].uniqu
        print(f'Num. machine: {len(trs_mcn)} (supervised), {len(tru_mcn)} (unsupervised), {len(ts_mcn)}

        Num. machine: 7 (supervised), 121 (unsupervised), 121 (test)
```

Data Loading and Preparation

Then we standardize the input data



In [9]:

```python
sscaler, nscaler = StandardScaler(), MinMaxScaler()
trs_s, tru_s, ts_s = trs.copy(), tru.copy(), ts.copy()
trs_s[dt_in] = sscaler.fit_transform(trs[dt_in])
tru_s[dt_in], ts_s[dt_in] = sscaler.transform(tru[dt_in]), sscaler.transform(ts[dt_in])
trs_s[['rul']] = nscaler.fit_transform(trs[['rul']])
tru_s[['rul']], ts_s[['rul']] = nscaler.transform(tru[['rul']]), nscaler.transform(ts[['rul']])

maxrul = nscaler.data_max_[0]
display(trs_s.head())
```

	src	machine	cycle	p1	p2	p3	s1	s2	s3	s4	...	s13	
1725	train_FD004	467	1	-1.688818	-1.924463	0.445653	1.811018	1.784571	1.676982	1.834240	...	0.445850	0.741
1726	train_FD004	467	2	-0.320795	0.385443	0.445653	0.754417	0.824865	0.604660	0.459056	...	0.445775	-0.158
1727	train_FD004	467	3	-1.688920	-1.925123	0.445653	1.811018	1.768350	1.668955	1.823341	...	0.445477	0.684
1728	train_FD004	467	4	1.184267	0.844852	0.445653	-1.021583	-0.742836	-0.576935	-0.541685	...	0.443309	0.078
1729	train_FD004	467	5	-1.688948	-1.925453	0.445653	1.811018	1.767810	1.726471	1.761244	...	0.445402	0.677

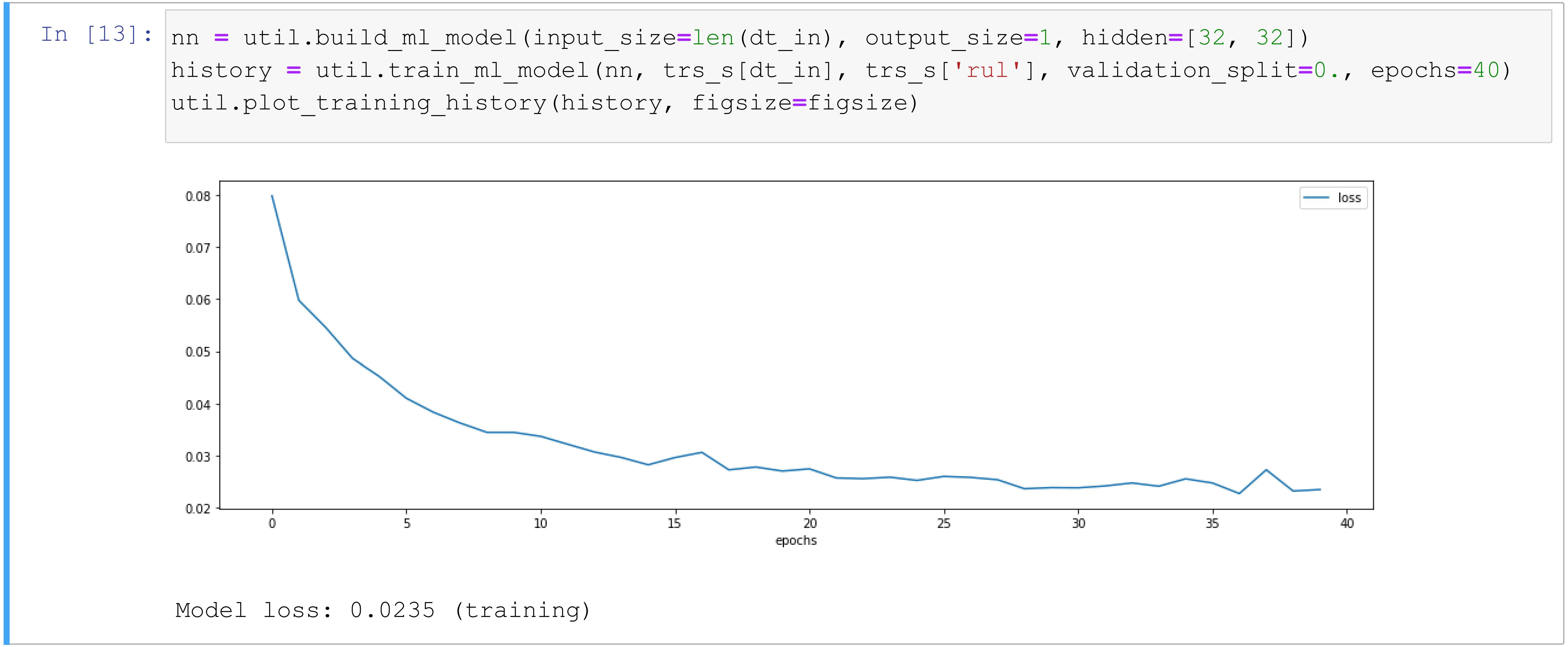
5 rows × 28 columns

Later, we will need the maximum RUL value on the training set

MLP with Scarce Labels

As a baseline, we will train a MLP model on the supervised data

We do not split a validation set, given we have scarce data



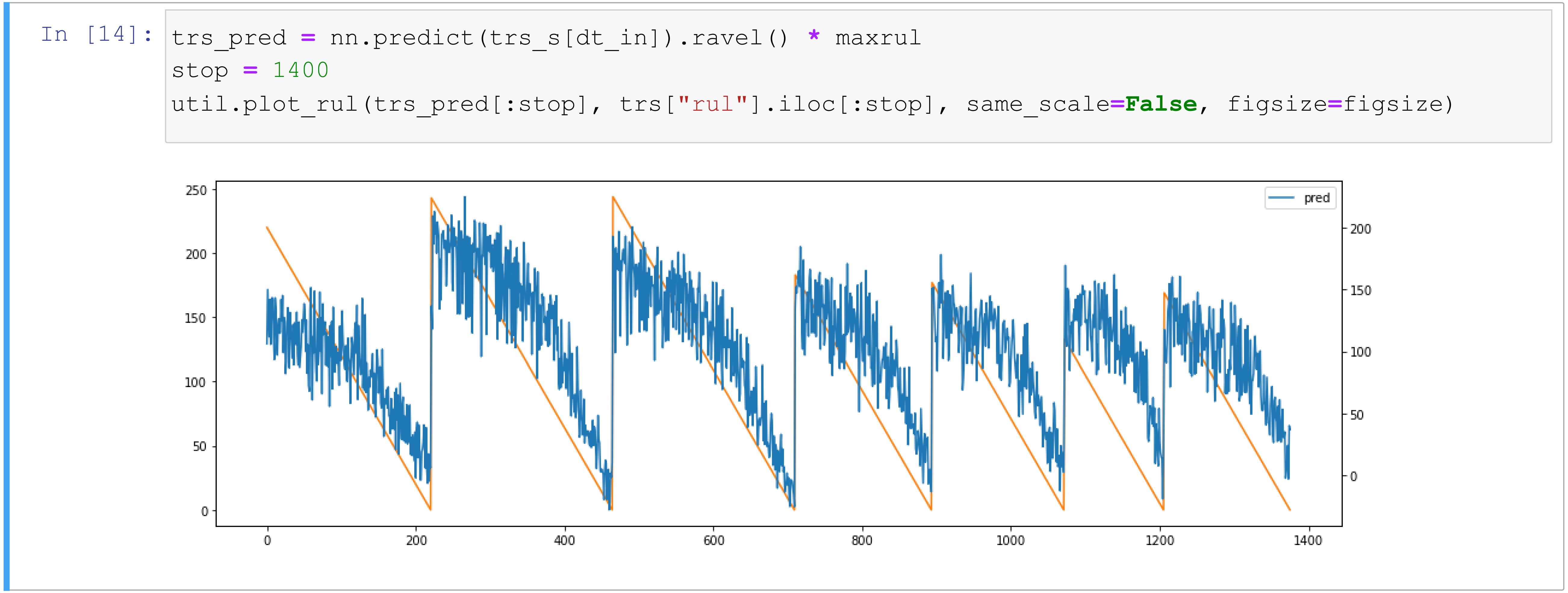
In [13]:

```
nn = util.build_ml_model(input_size=len(dt_in), output_size=1, hidden=[32, 32])
history = util.train_ml_model(nn, trs_s[dt_in], trs_s['rul'], validation_split=0., epochs=40)
util.plot_training_history(history, figsize=figsize)
```

```
Model loss: 0.0235 (training)
```

Evaluation

Let's inspect the predictions



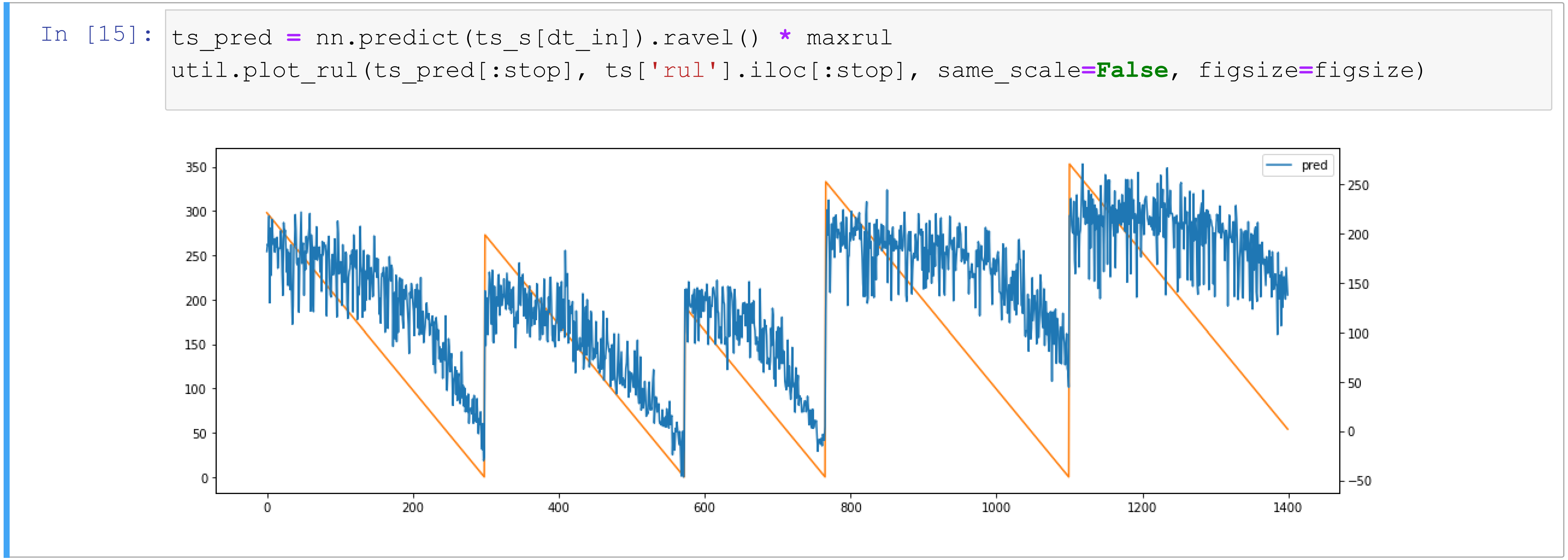
```
In [14]: trs_pred = nn.predict(trs_s[dt_in]).ravel() * maxrul
         stop = 1400
         util.plot_rul(trs_pred[:stop], trs["rul"].iloc[:stop], same_scale=False, figsize=figsize)
```

- The predictions have a decreasing trend (which is good)
- ...But they are very noisy (which is bad)

Evaluation

The behavior on the test data has a similar trend

...And it is similarly noisy



```
In [15]: ts_pred = nn.predict(ts_s[dt_in]).ravel() * maxrul
         util.plot_rul(ts_pred[:stop], ts['rul'].iloc[:stop], same_scale=False, figsize=figsize)
```

Cost Model

The RUL estimator is meant to be used to define a policy

Namely, we stop operations when:

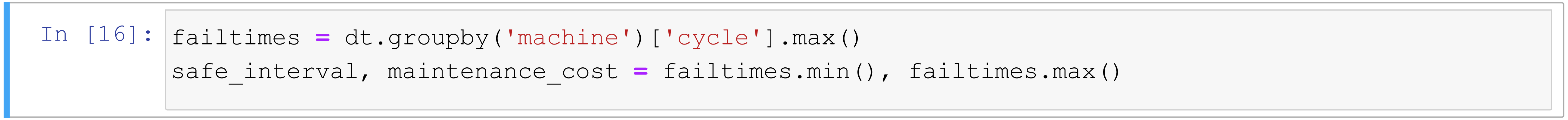
$$f(x, \omega) \leq \theta$$

- Where $f(x, \omega)$ is the estimated output and θ is threshold

Calibrating θ is best done by relying on a cost model

- We assume that operating for a time step generates 1 unit of profit
- ...And that failing looses C units of profits w.r.t. performing maintenance
- We also assume we never stop a machine before a "safe" interval s

Both C and s are calibrated on data in our example:

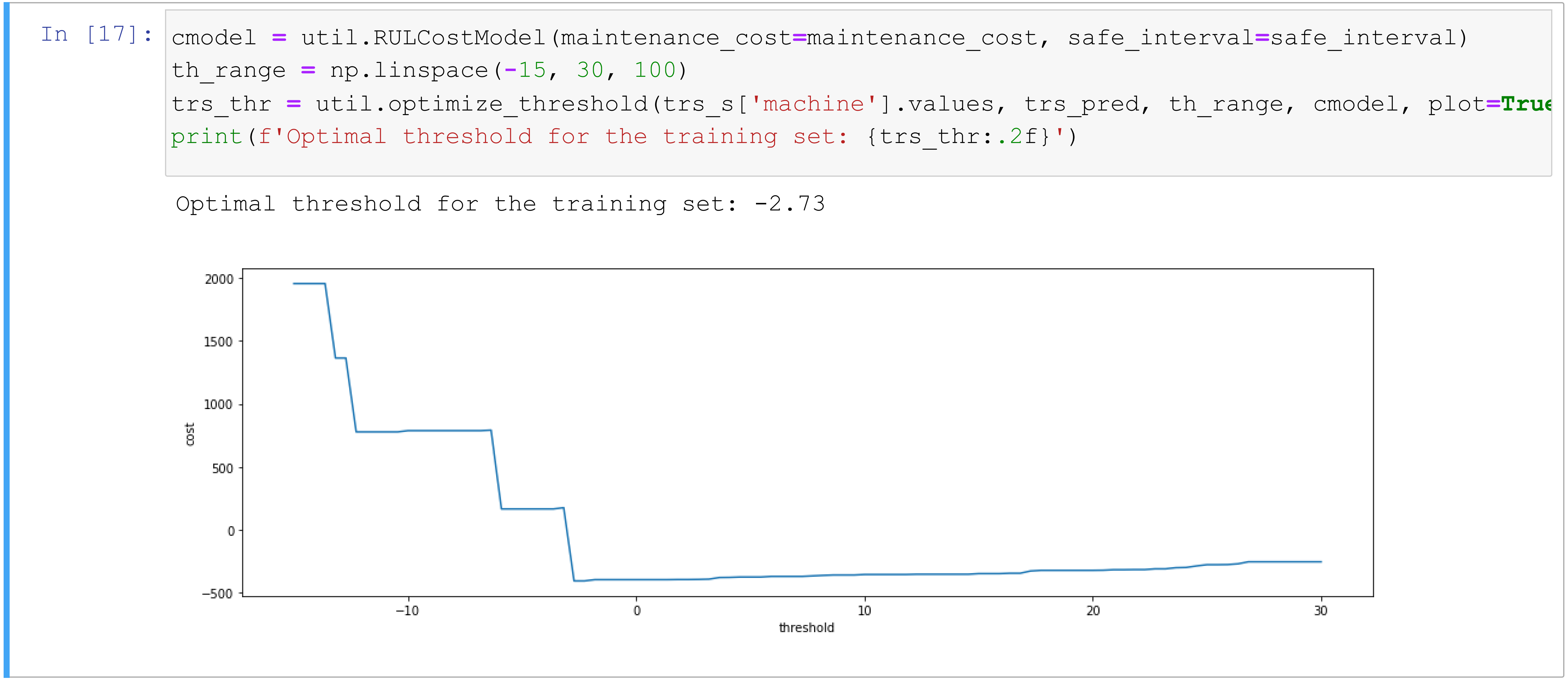


In [16]:

```
failtimes = dt.groupby('machine')['cycle'].max()
safe_interval, maintenance_cost = failtimes.min(), failtimes.max()
```

Cost Model and Threshold Optimization

We then proceed to choose θ to optimize the cost



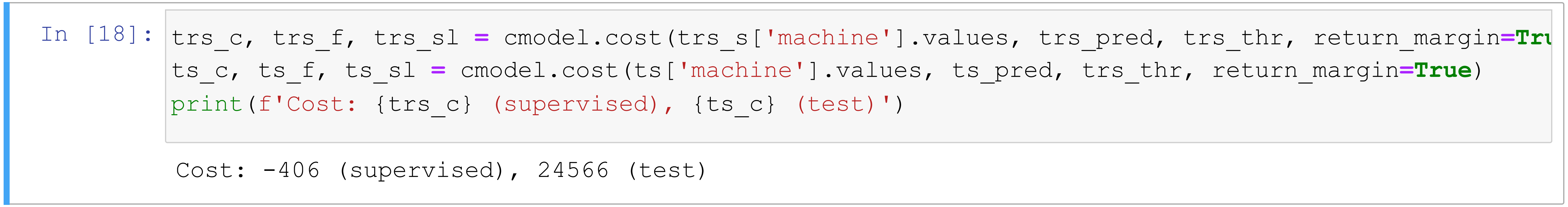
In [17]:

```
cmodel = util.RULCostModel(maintenance_cost=maintenance_cost, safe_interval=safe_interval)
th_range = np.linspace(-15, 30, 100)
trs_thr = util.optimize_threshold(trs_s['machine'].values, trs_pred, th_range, cmodel, plot=True
print(f'Optimal threshold for the training set: {trs_thr:.2f}')
```

```
Optimal threshold for the training set: -2.73
```

Cost Results

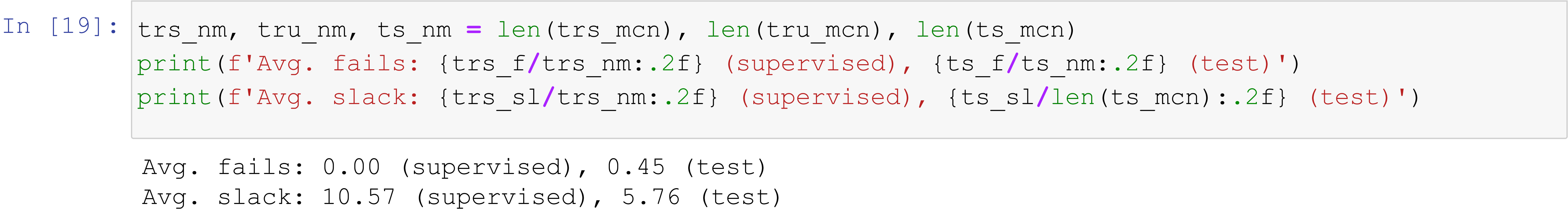
Let's now check the costs on all datasets



In [18]:
```
trs_c, trs_f, trs_sl = cmodel.cost(trs_s['machine'].values, trs_pred, trs_thr, return_margin=Tru
ts_c, ts_f, ts_sl = cmodel.cost(ts['machine'].values, ts_pred, trs_thr, return_margin=True)
print(f'Cost: {trs_c} (supervised), {ts_c} (test)')
```

```
Cost: -406 (supervised), 24566 (test)
```

- The cost for the training set is good (negative)
- ...But that is not the case for the training set



In [19]:
```
trs_nm, tru_nm, ts_nm = len(trs_mcn), len(tru_mcn), len(ts_mcn)
print(f'Avg. fails: {trs_f/trs_nm:.2f} (supervised), {ts_f/ts_nm:.2f} (test)')
print(f'Avg. slack: {trs_sl/trs_nm:.2f} (supervised), {ts_sl/len(ts_mcn):.2f} (test)')
```

```
Avg. fails: 0.00 (supervised), 0.45 (test)
Avg. slack: 10.57 (supervised), 5.76 (test)
```

- In particular, there is a very high failure rate on unseen data

Domain Knowledge as Constraints

Domain Knowledge as Constraints

We know that the RUL decreases at a fixed rate

- After 1 time step, the RUL will have decreased by 1 unit
- After 2 time steps, the RUL will have decreased by 2 units and so on

In general, let $\hat{x}_i$ and $\hat{x}_j$ be the i-th and j-th samples for a given component

Then we know that:

$$f(\hat{x}_i, \theta) - f(\hat{x}_j, \theta) = j - i \qquad \forall i, j = 1..m \text{ s.t. } c_i = c_j$$

- c_i, c_j are the machine for (respectively) sample i and j
- Samples are assumed to be temporally sorted
- The left-most terms is the difference between the predicted RULs
- $j - i$ is the difference between the sequential indexes of the two samples
- ...Which by construction should be equal to the RUL difference

Domain Knowledge as Constraints

The relation we identified is a constraint

$$f(\hat{x}_i, \theta) - f(\hat{x}_j, \theta) = j - i \qquad \forall i, j = 1..m \text{ s.t. } c_i = c_j$$

It represents domain knowledge that should (in principle) hold for our problem

- It is fine to treat this as a soft constraint

As a regularization term, we will use:

$$\lambda\Big(f(\hat{x}_i, \omega) - f(\hat{x}_j, \omega) - (j - i)\Big)^2$$

Using the absolute value (h1 norm) may also work

- The constraint involves pairs of example, i.e. it is a relational constraint
- In principle we should consider all pairs, but that may scale poorly

Our Regularizer

We can focus on contiguous pairs, leading to the loss

$$L(\hat{x}, \omega) + \lambda \sum_{\substack{i \prec j \\ c_i = c_j}} \left(f(\hat{x}_i, \omega) - f(\hat{x}_j, \omega) - (j - i) \right)^2$$

- Where $i \prec j$ iff j is the next sample for after i for a given machine
- This approach requires a linear (rather than quadratic) number of constraints

It can work with mini-batches

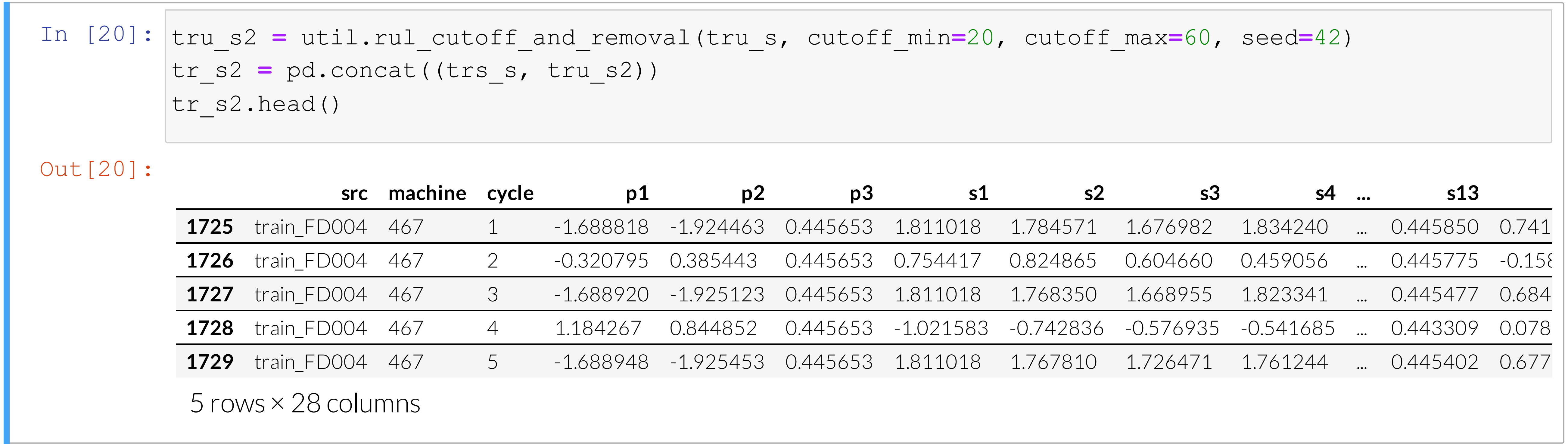
- In this case, $\prec$ will refer to contiguous samples in the same batch
- ...And of course for the same component

We will now see how to implement this approach

Removing RUL Values

We start by preparing a bit more the unsupervised data

- We remove the end of the unsupervised data sequences
- Then, we replace RUL values with -1 (invalid)
- Finally, we merge supervised and unsupervised data in a single dataset



In [20]:

```
tru_s2 = util.rul_cutoff_and_removal(tru_s, cutoff_min=20, cutoff_max=60, seed=42)
tr_s2 = pd.concat((trs_s, tru_s2))
tr_s2.head()
```

Out[20]:

	src	machine	cycle	p1	p2	p3	s1	s2	s3	s4	...	s13	
1725	train_FD004	467	1	-1.688818	-1.924463	0.445653	1.811018	1.784571	1.676982	1.834240	...	0.445850	0.741
1726	train_FD004	467	2	-0.320795	0.385443	0.445653	0.754417	0.824865	0.604660	0.459056	...	0.445775	-0.15
1727	train_FD004	467	3	-1.688920	-1.925123	0.445653	1.811018	1.768350	1.668955	1.823341	...	0.445477	0.684
1728	train_FD004	467	4	1.184267	0.844852	0.445653	-1.021583	-0.742836	-0.576935	-0.541685	...	0.443309	0.078
1729	train_FD004	467	5	-1.688948	-1.925453	0.445653	1.811018	1.767810	1.726471	1.761244	...	0.445402	0.677

5 rows × 28 columns

Generating Batches from the Same Machine

Our regularizer requires to have sorted samples from the same machine

The easiest way to ensure we have enough is using a custom `DataGenerator`

```
class SMBatchGenerator(tf.keras.utils.Sequence):
    def __init__(self, data, in_cols, batch_size, seed=42): ...
    def __len__(self): ...
    def __getitem__(self, index): ...
    def on_epoch_end(self): ...
    def __build_batches(self): ...
```

- `__len__` is called to know how many batches are left
- `__getitem__` should return one batch
- `on_epoch_end` should take care (e.g.) of shuffling

Generating Batches from the Same Machine

The `__init__` method takes care of the initial setup

```python
def __init__(self, data, in_cols, batch_size, seed=42):
    super(SMBatchGenerator).__init__()
    self.data = data
    self.in_cols = in_cols
    self.dpm = split_by_field(data, 'machine')
    self.rng = np.random.default_rng(seed)
    self.batch_size = batch_size
    # Build the first sequence of batches
    self.__build_batches()
```

- We store some fields
- We split the data by machine
- We build a dedicated RNG
- ...And finally we call the custom-made `__build_batches` method

Generating Batches from the Same Machine

The `__build_batches` method prepares the batches for one full epoch

```python
def __build_batches(self):
    self.batches, self.machines = [], []
    mcns = list(self.dpm.keys())
    self.rng.shuffle(mcns) # sort the machines at random
    for mcn in mcns: # Loop over all machines
        index = self.dpm[mcn].index # sample indexes for this machine
        ...
        self.rng.shuffle(idx) # shuffle sample indexes for this machine
        bt = idx.reshape(-1, self.batch_size) # split into batches
        bt = np.sort(bt, axis=1) # sort every batch individually
        self.batches.append(bt) # store the batch
        self.machines.append(np.repeat([mcn], len(bt))) # add machine information
    self.batches = np.vstack(self.batches) # concatenate
    self.machines = np.hstack(self.machines)
```

Generating Batches from the Same Machine

We rebuild batches after each epoch

```python
def on_epoch_end(self):
    self.__build_batches()
```

Most of the remaining work is done in the `__getiitem__` method:

```python
def __getitem__(self, index):
    idx = self.batches[index]
    x = self.data[self.in_cols].loc[idx].values
    y = self.data['rul'].loc[idx].values
    flags = (y != -1)
    info = np.vstack((y, flags, idx)).T
    return x, info
```

- The RUL value is -1 for the unsupervised data: we flag the meaningful RULs
- ...We pack indexes, RUL values, and flags into a single `info` tensor

Custom Training Step

We then enforce the constraints by means of a custom training step

```
class CstRULRegressor(keras.Model):
    def __init__(self, rul_pred, alpha, beta, maxrul): ...

    def train_step(self, data): ...

    def call(self, data): return self.rul_pred(data)

    ...
```

- We use a custom `keras.Model` subclass
- ...And accept an externally built RUL prediction model (`rul_pred`)
- The custom training step is implemented in `train_step`
- The `call` method relies on the external model for RUL prediction

Custom Training Step

In the `__init__` function:

```
def __init__(self, rul_pred, alpha, beta, maxrul):
    super(CstRULRegressor, self).__init__(input_shape, hidden)
    # Store the base RUL prediction model
    self.rul_pred = rul_pred
    # Weights
    self.alpha = alpha
    self.beta = beta
    self.maxrul = maxrul
    ...
```

- `beta` is the regularizer weight, `alpha` is a weight for the loss function itself
- We also store the maximum RUL

Custom Training Step

In the custom training step:

```
def train_step(self, data):
    x, info = data
    y_true, flags, idx = info[:, 0:1], info[:, 1:2], info[:, 2:3]
    with tf.GradientTape() as tape:
        y_pred = self(x, training=True) # predictions
        mse = k.mean(flags * k.square(y_pred-y_true)) # MSE loss
        delta_pred = y_pred[1:] - y_pred[:-1] # pred. difference
        delta_rul = -(idx[1:] - idx[:-1]) /self.maxrul # index difference
        deltadiff = delta_pred - delta_rul # difference of differences
        cst = k.mean(k.square(deltadiff)) # regualization term
        loss = self.alpha * mse + self.beta * cst # loss
    ...
```

- We unpack the `info` tensor
- Inside a `GradientTape`, we construct our regularized loss

Custom Training Step

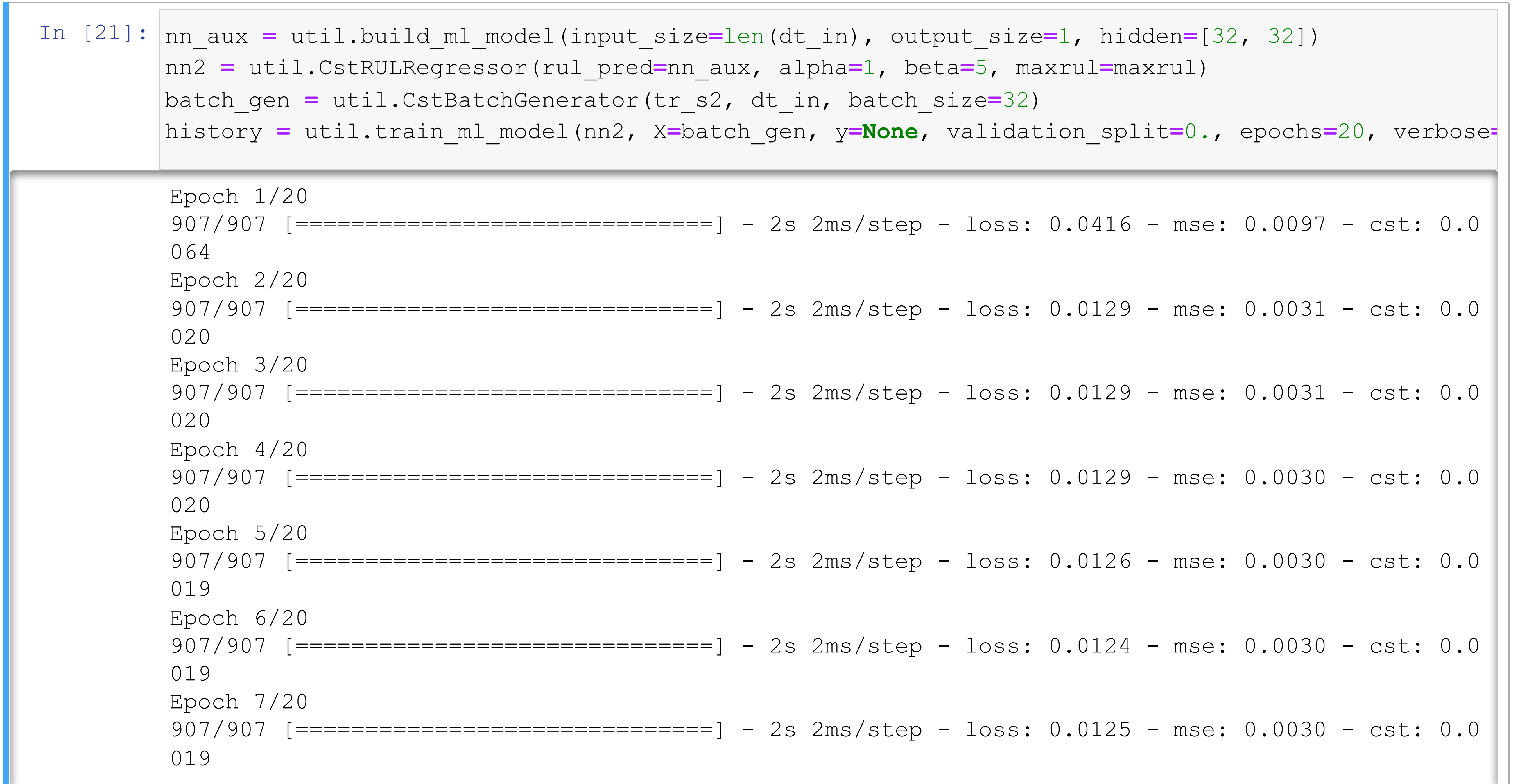
In the custom training step:

```python
def train_step(self, data):
    ...
    tr_vars = self.trainable_variables
    grads = tape.gradient(loss, tr_vars) # gradient computation

    self.optimizer.apply_gradients(zip(grads, tr_vars)) # weight update

    ...
```

- We then apply the (Stochastic) Gradient Descent step
- Then we update and retun the loss trackers

The SBR Approach

We can now test our SBR approach



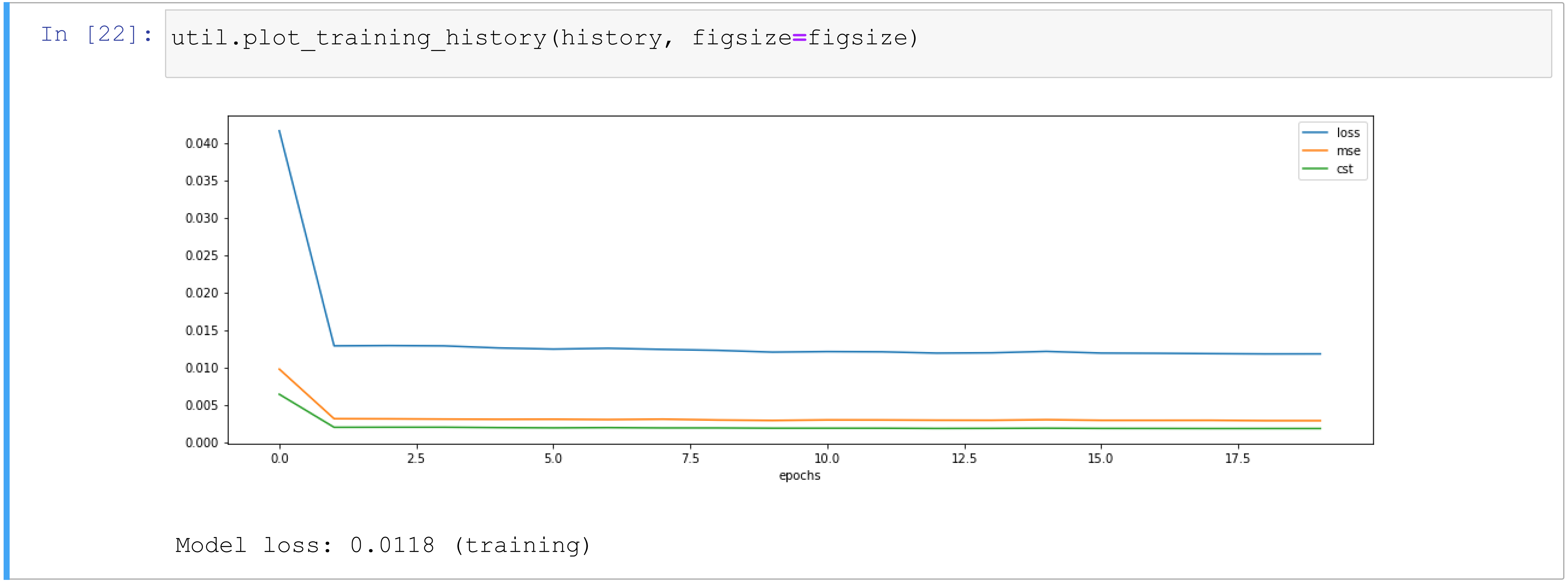
In [21]:

```
nn_aux = util.build_ml_model(input_size=len(dt_in), output_size=1, hidden=[32, 32])
nn2 = util.CstRULRegressor(rul_pred=nn_aux, alpha=1, beta=5, maxrul=maxrul)
batch_gen = util.CstBatchGenerator(tr_s2, dt_in, batch_size=32)
history = util.train_ml_model(nn2, X=batch_gen, y=None, validation_split=0., epochs=20, verbose=
```

```
Epoch 1/20
907/907 [==============================] - 2s 2ms/step - loss: 0.0416 - mse: 0.0097 - cst: 0.0
064
Epoch 2/20
907/907 [==============================] - 2s 2ms/step - loss: 0.0129 - mse: 0.0031 - cst: 0.0
020
Epoch 3/20
907/907 [==============================] - 2s 2ms/step - loss: 0.0129 - mse: 0.0031 - cst: 0.0
020
Epoch 4/20
907/907 [==============================] - 2s 2ms/step - loss: 0.0129 - mse: 0.0030 - cst: 0.0
020
Epoch 5/20
907/907 [==============================] - 2s 2ms/step - loss: 0.0126 - mse: 0.0030 - cst: 0.0
019
Epoch 6/20
907/907 [==============================] - 2s 2ms/step - loss: 0.0124 - mse: 0.0030 - cst: 0.0
019
Epoch 7/20
907/907 [==============================] - 2s 2ms/step - loss: 0.0125 - mse: 0.0030 - cst: 0.0
019
```

The SBR Approach

...And we can check the training curve

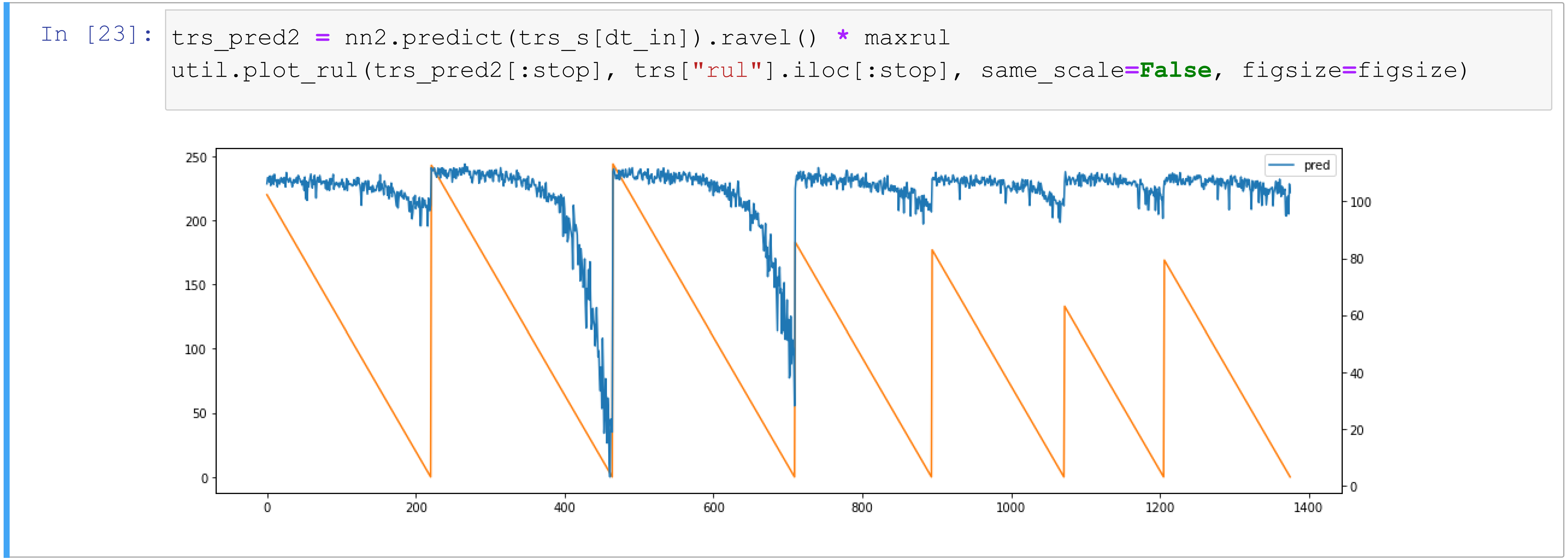


```
In [22]: util.plot_training_history(history, figsize=figsize)
```

```
Model loss: 0.0118 (training)
```

Inspecting the Predictions

Let's have a look at the predictions on the supervised data

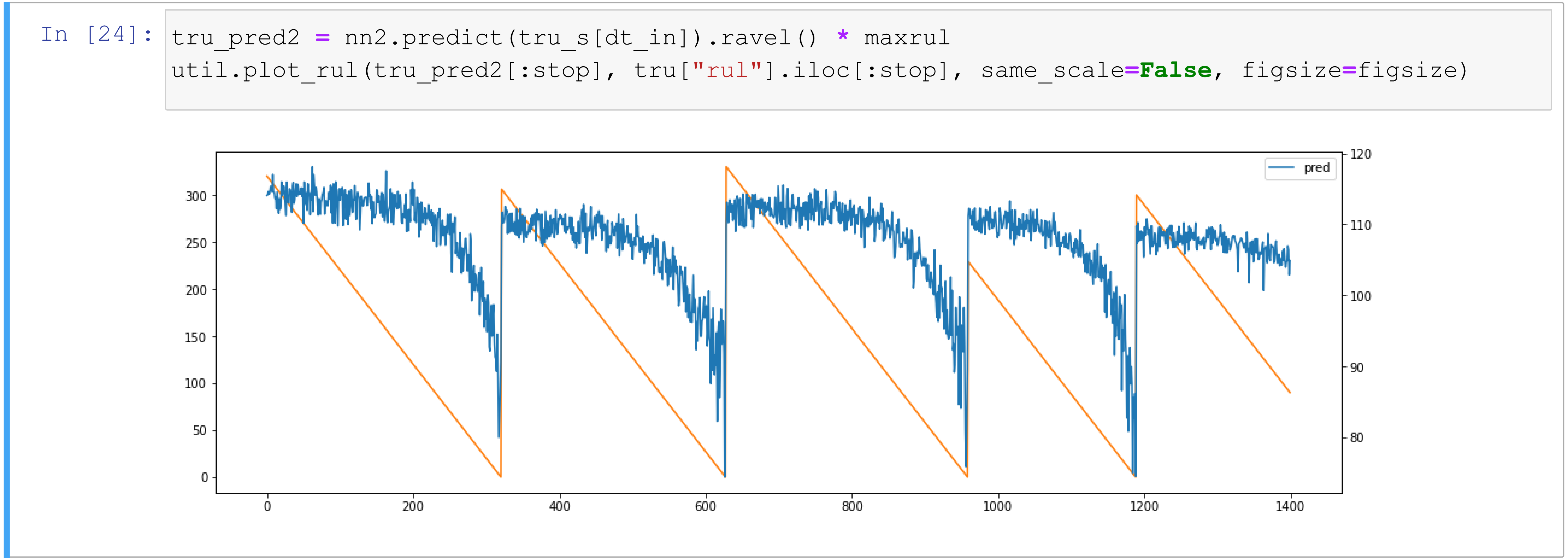


```
In [23]: trs_pred2 = nn2.predict(trs_s[dt_in]).ravel() * maxrul
         util.plot_rul(trs_pred2[:stop], trs["rul"].iloc[:stop], same_scale=False, figsize=figsize)
```

- The signal is much more stable
- The scale is still off, but we can fix that with a well chosen threshold

Inspecting the Predictions

Then let's do the same for the unsupervised data

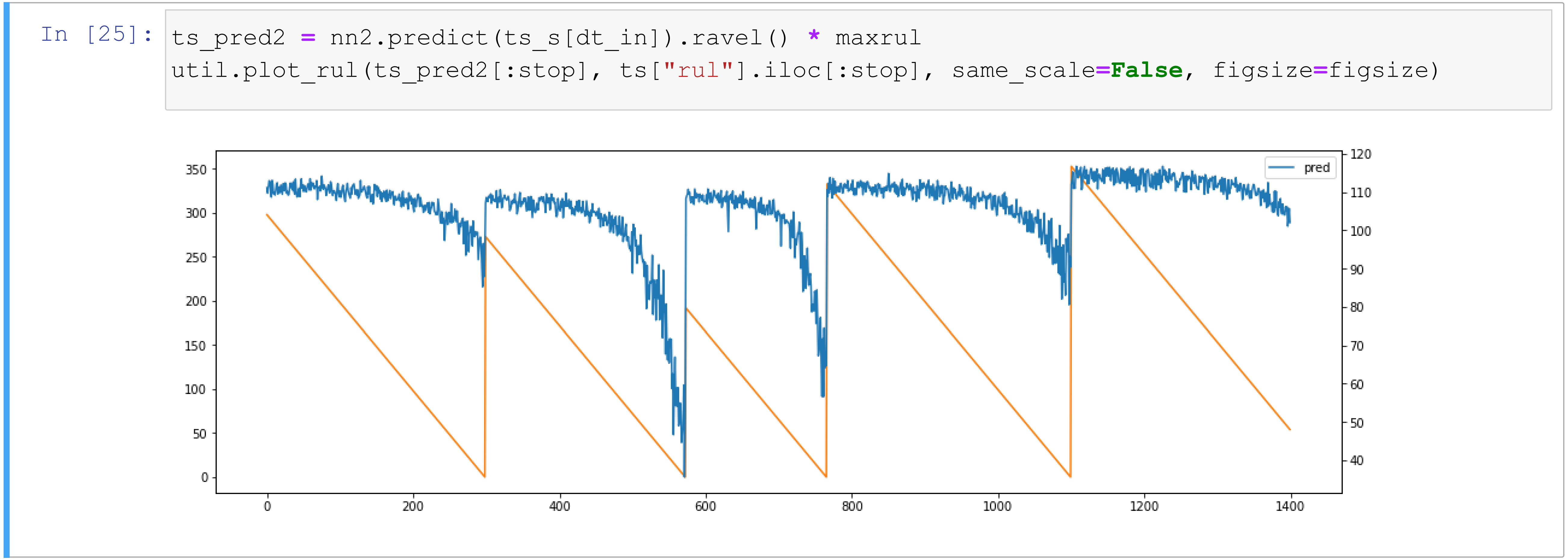


In [24]:

```
tru_pred2 = nn2.predict(tru_s[dt_in]).ravel() * maxrul
util.plot_rul(tru_pred2[:stop], tru["rul"].iloc[:stop], same_scale=False, figsize=figsize)
```

Inspecting the Predictions

Then let's do the same for the test data



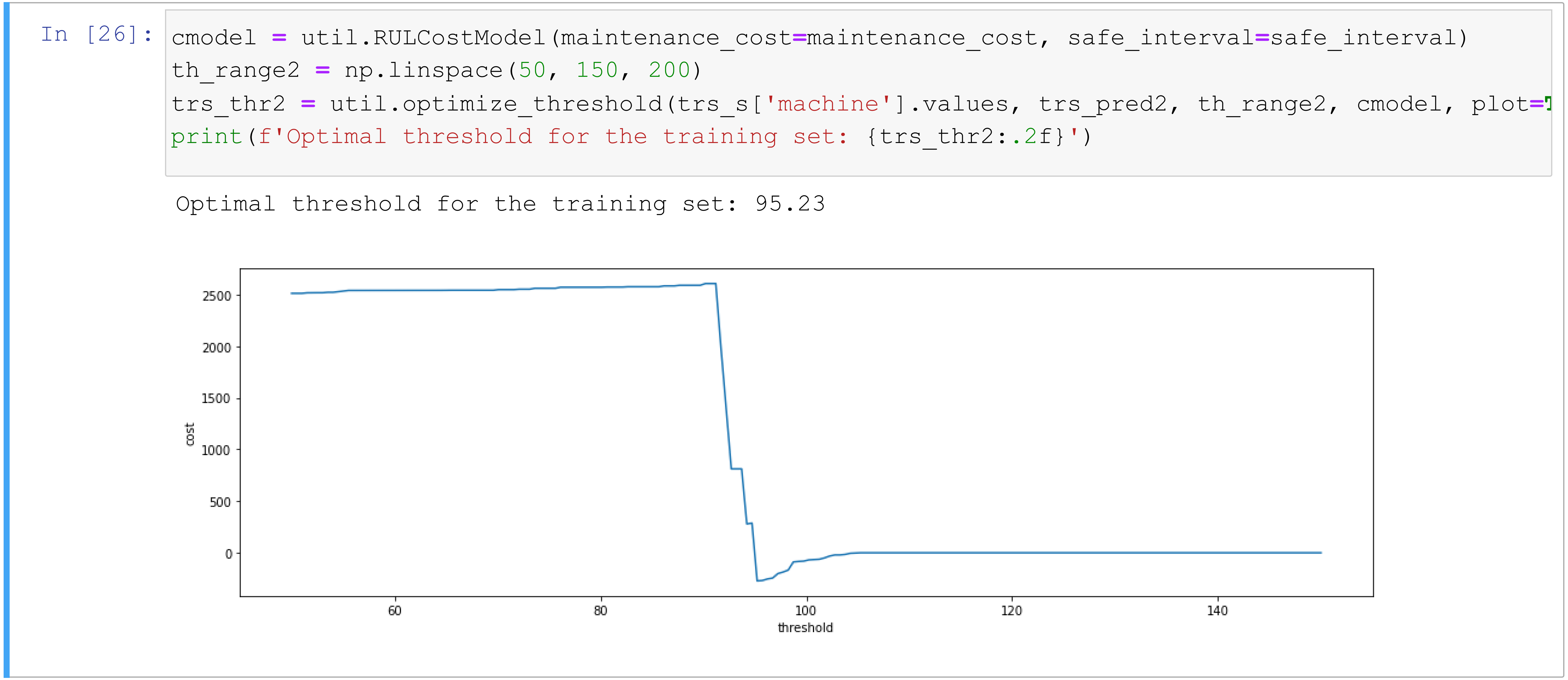
In [25]:

```
ts_pred2 = nn2.predict(ts_s[dt_in]).ravel() * maxrul
util.plot_rul(ts_pred2[:stop], ts["rul"].iloc[:stop], same_scale=False, figsize=figsize)
```

The behavior is more stable and consistent than before

Threshold Optimization and Cost Evaluation

We can now optimize the threshold optimization (on the supervised data)



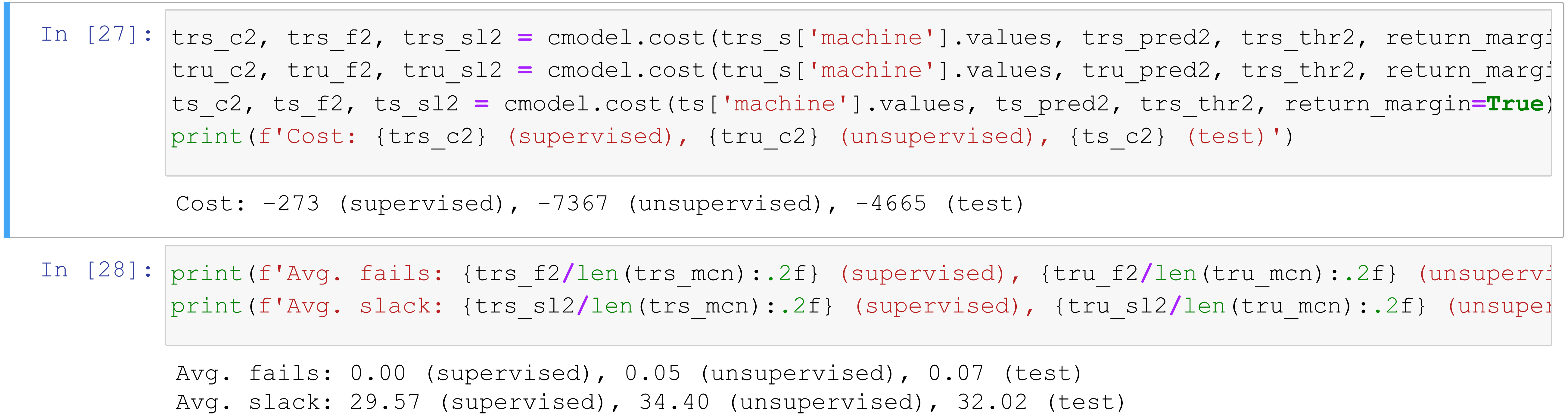
In [26]:

```
cmodel = util.RULCostModel(maintenance_cost=maintenance_cost, safe_interval=safe_interval)
th_range2 = np.linspace(50, 150, 200)
trs_thr2 = util.optimize_threshold(trs_s['machine'].values, trs_pred2, th_range2, cmodel, plot=
print(f'Optimal threshold for the training set: {trs_thr2:.2f}')
```

```
Optimal threshold for the training set: 95.23
```

Threshold Optimization and Cost Evaluation

Finally, we can evaluate the SBR approach in terms of cost



```
In [27]: trs_c2, trs_f2, trs_sl2 = cmodel.cost(trs_s['machine'].values, trs_pred2, trs_thr2, return_margi
         tru_c2, tru_f2, tru_sl2 = cmodel.cost(tru_s['machine'].values, tru_pred2, trs_thr2, return_margi
         ts_c2, ts_f2, ts_sl2 = cmodel.cost(ts['machine'].values, ts_pred2, trs_thr2, return_margin=True)
         print(f'Cost: {trs_c2} (supervised), {tru_c2} (unsupervised), {ts_c2} (test)')
```

```
Cost: -273 (supervised), -7367 (unsupervised), -4665 (test)
```

```
In [28]: print(f'Avg. fails: {trs_f2/len(trs_mcn):.2f} (supervised), {tru_f2/len(tru_mcn):.2f} (unsupervi
         print(f'Avg. slack: {trs_sl2/len(trs_mcn):.2f} (supervised), {tru_sl2/len(tru_mcn):.2f} (unsuper
```

```
Avg. fails: 0.00 (supervised), 0.05 (unsupervised), 0.07 (test)
Avg. slack: 29.57 (supervised), 34.40 (unsupervised), 32.02 (test)
```

- The number of fails has decreased very significantly
- The slack is still contained

And we did this with just a handful of run-to-failure experiments