Mastering Gradient Boosting with CatBoost

In this tutorial we will use dataset Amazon Employee Access Challenge from Kaggle (https://www.kaggle.com) competition for our experiments. Here (https://www.kaggle.com/c/amazon-employee-access-challenge/data) is the link to the challenge, that we will be exploring.

Libraries installation

In [2]:

```
# !pip install --user --upgrade catboost
# !pip install --user --upgrade ipywidgets
# !pip install shap
# !pip install sklearn
# !jupyter nbextension enable --py widgetsnbextension
```

In [3]:

```
import os
import pandas as pd
import numpy as np
np.set_printoptions(precision=4)

import catboost
print(catboost.__version__)

print(np.version.version)
# !pip install shap
# import shap
```

```
0.17.1
1.17.2
```

Reading the data

In [4]:

```
from catboost.datasets import amazon

# If you have "URLError: SSL: CERTIFICATE_VERIFY_FAILED" uncomment next two
# import ssl
# ssl._create_default_https_context = ssl._create_unverified_context

# If you have any other error:
# Download datasets from http://bit.ly/2ZUXTSv and uncomment next line:
# train_df = pd.read_csv('train.csv', sep=',', header='infer')

(train_df, test_df) = amazon()
```

In [5]:
```
train_df.head()

# feature values look like hashes and they are really hashes
# so this features should be hanled as categorical features
# and catboost allows it do out of the box
```

Out[5]:

	ACTION	RESOURCE	MGR_ID	ROLE_ROLLUP_1	ROLE_ROLLUP_2	ROLE_DEPTNAME	ROLE_TI
0	1	39353	85475	117961	118300	123472	117
1	1	17183	1540	117961	118343	123125	118
2	1	36724	14457	118219	118220	117884	117
3	1	36135	5396	117961	118343	119993	118
4	1	42680	5905	117929	117930	119569	119

In [6]:
```
train_df.tail()
```

Out[6]:

	ACTION	RESOURCE	MGR_ID	ROLE_ROLLUP_1	ROLE_ROLLUP_2	ROLE_DEPTNAME	ROLI
32764	1	23497	16971	117961	118300	119993	
32765	1	25139	311198	91261	118026	122392	
32766	1	34924	28805	117961	118327	120299	
32767	1	80574	55643	118256	118257	117945	
32768	1	14354	59575	117916	118150	117920	

Exploring the data

Label values extraction

In [7]:
```
# separate labels from features into separate Series structure
y = train_df.ACTION
print('Type of y:', type(y))
print('What labels look like:')
print(y, '\n')

# create a feature matrix
X = train_df.drop('ACTION', axis=1)
print('Type of X matrix:', type(X))
X.head()
```

```
Type of y: <class 'pandas.core.series.Series'>
What labels look like:
0        1
1        1
2        1
3        1
4        1
        ..
32764    1
32765    1
32766    1
32767    1
32768    1
Name: ACTION, Length: 32769, dtype: int64

Type of X matrix: <class 'pandas.core.frame.DataFrame'>
```

Out[7]:

	RESOURCE	MGR_ID	ROLE_ROLLUP_1	ROLE_ROLLUP_2	ROLE_DEPTNAME	ROLE_TITLE	ROLI
0	39353	85475	117961	118300	123472	117905	
1	17183	1540	117961	118343	123125	118536	
2	36724	14457	118219	118220	117884	117879	
3	36135	5396	117961	118343	119993	118321	
4	42680	5905	117929	117930	119569	119323	

Categorical features declaration

In [8]:
```
# all the features are categorical
cat_features = list(range(0, X.shape[1]))
print(cat_features)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8]
```

Looking on label balance in dataset

In [9]:
```
print('Labels: {}'.format(set(y)))
print('len(y):', len(y))
print('sum(y):', sum(y))
print('Zero count = {}, One count = {}'.format(len(y) - sum(y), sum(y)))
```

```
Labels: {0, 1}
len(y): 32769
sum(y): 30872
Zero count = 1897, One count = 30872
```

So the dataset is pretty unbalancedю

Training the first model

The first model is going to be super simple and week due to low number of iterations: 100. Normally a value about a 1000 is used. 100 is never enough.

In [10]:
```
from catboost import CatBoostClassifier

# we create the object of CatBoostClassifier class
model = CatBoostClassifier(iterations=100)

# we are passing categorical features as parameters here
# verbose = 10 outputs only each 10th tree
model.fit(X, y, cat_features=cat_features, verbose=10)
```

```
Learning rate set to 0.349945
0:	learn: 0.4668712	total: 61ms	remaining: 6.04s
10:	learn: 0.1791505	total: 112ms	remaining: 902ms
20:	learn: 0.1684338	total: 165ms	remaining: 619ms
30:	learn: 0.1650651	total: 236ms	remaining: 526ms
40:	learn: 0.1640099	total: 296ms	remaining: 425ms
50:	learn: 0.1619138	total: 349ms	remaining: 335ms
60:	learn: 0.1606667	total: 411ms	remaining: 263ms
70:	learn: 0.1595925	total: 472ms	remaining: 193ms
80:	learn: 0.1578885	total: 529ms	remaining: 124ms
90:	learn: 0.1566430	total: 595ms	remaining: 58.9ms
99:	learn: 0.1556847	total: 643ms	remaining: 0us
```

Out[10]:
```
<catboost.core.CatBoostClassifier at 0x1087a8590>
```

predtict_proba returns an array where for each object we get two numbers representing the probability of belongning to each class.

In [11]:
```
model.predict_proba(X)
```

Out[11]:
```
array([[0.0073, 0.9927],
       [0.0078, 0.9922],
       [0.045 , 0.955 ],
       ...,
       [0.0128, 0.9872],
       [0.1204, 0.8796],
       [0.024 , 0.976 ]])
```

Working with dataset

There are several ways of passing dataset to training - using X,y (the initial matrix) or using Pool class. Pool class is the class for storing the dataset. In the next few blocks we'll explore the ways to create a Pool object.

You can use Pool class if the dataset has more than just X and y (for example, it has sample weights or groups) or if the dataset is large and it takes long time to read it into python.

In [12]:
```
from catboost import Pool
pool = Pool(data=X, label=y, cat_features=cat_features)
```

Split your data into train and validation

In [13]:
```
from sklearn.model_selection import train_test_split

# random_state is the seed used by the random number generator
data = train_test_split(X, y, test_size=0.2, random_state=0)
X_train, X_validation, y_train, y_validation = data

# create train_pool object
train_pool = Pool(
    data=X_train,
    label=y_train,
    cat_features=cat_features
)

# create validation_pool object
validation_pool = Pool(
    data=X_validation,
    label=y_validation,
    cat_features=cat_features
)
```

Selecting the objective function

Possible options for binary classification:

`Logloss` for binary target.

`CrossEntropy` for probabilities in target.

For binary classification there are basically two options to use: Logloss and CrossEntropy. If we have the probabilities in our labels then we use CrossEntropy, if we have 1-0 labels, we use Logloss.

In [14]:
```
# loss function in our case is selected automatically
model = CatBoostClassifier(
    iterations=5,
    learning_rate=0.1,
    # loss_function='CrossEntropy'
)
model.fit(train_pool, eval_set=validation_pool, verbose=False)

print('Model is fitted: {}'.format(model.is_fitted()))
print('Model params:{}'.format(model.get_params()))
```

```
Model is fitted: True
Model params:{'iterations': 5, 'learning_rate': 0.1}
```

Stdout of the training

If we don't specify the learning rate, it is set automatically using the properties of the dataset (number of samples, number of features) and properties of the model (like number of the iterations). Learning rate in our case is 0.5 and it is really big learning rate that is selected mainly because we have too little iteratins.

In [15]:
```
model = CatBoostClassifier(
    iterations=15,
    )

model.fit(train_pool, eval_set=validation_pool);
```

```
Learning rate set to 0.5
0:	learn: 0.3971379	test: 0.3960691 best: 0.3960691 (0)	t
otal: 7.01ms	remaining: 98.1ms
1:	learn: 0.2948071	test: 0.2924021 best: 0.2924021 (1)	t
otal: 13ms	remaining: 84.7ms
2:	learn: 0.2485015	test: 0.2455237 best: 0.2455237 (2)	t
otal: 18.2ms	remaining: 73ms
3:	learn: 0.2234262	test: 0.2192359 best: 0.2192359 (3)	t
otal: 24.1ms	remaining: 66.3ms
4:	learn: 0.2003506	test: 0.1938956 best: 0.1938956 (4)	t
otal: 29.5ms	remaining: 58.9ms
5:	learn: 0.1916473	test: 0.1831990 best: 0.1831990 (5)	t
otal: 34.5ms	remaining: 51.7ms
6:	learn: 0.1842038	test: 0.1759780 best: 0.1759780 (6)	t
otal: 39.4ms	remaining: 45.1ms
7:	learn: 0.1808767	test: 0.1722588 best: 0.1722588 (7)	t
otal: 43.8ms	remaining: 38.3ms
8:	learn: 0.1783738	test: 0.1678080 best: 0.1678080 (8)	t
otal: 49.3ms	remaining: 32.9ms
9:	learn: 0.1769061	test: 0.1658153 best: 0.1658153 (9)	t
otal: 53.8ms	remaining: 26.9ms
10:	learn: 0.1761268	test: 0.1653031 best: 0.1653031 (10)	t
otal: 58.1ms	remaining: 21.1ms
11:	learn: 0.1752620	test: 0.1645631 best: 0.1645631 (11)	t
otal: 62.5ms	remaining: 15.6ms
12:	learn: 0.1749475	test: 0.1643973 best: 0.1643973 (12)	t
otal: 67ms	remaining: 10.3ms
13:	learn: 0.1746794	test: 0.1643479 best: 0.1643479 (13)	t
otal: 71.6ms	remaining: 5.12ms
14:	learn: 0.1739538	test: 0.1632657 best: 0.1632657 (14)	t
otal: 75.9ms	remaining: 0us

bestTest = 0.1632656889
bestIteration = 14
```

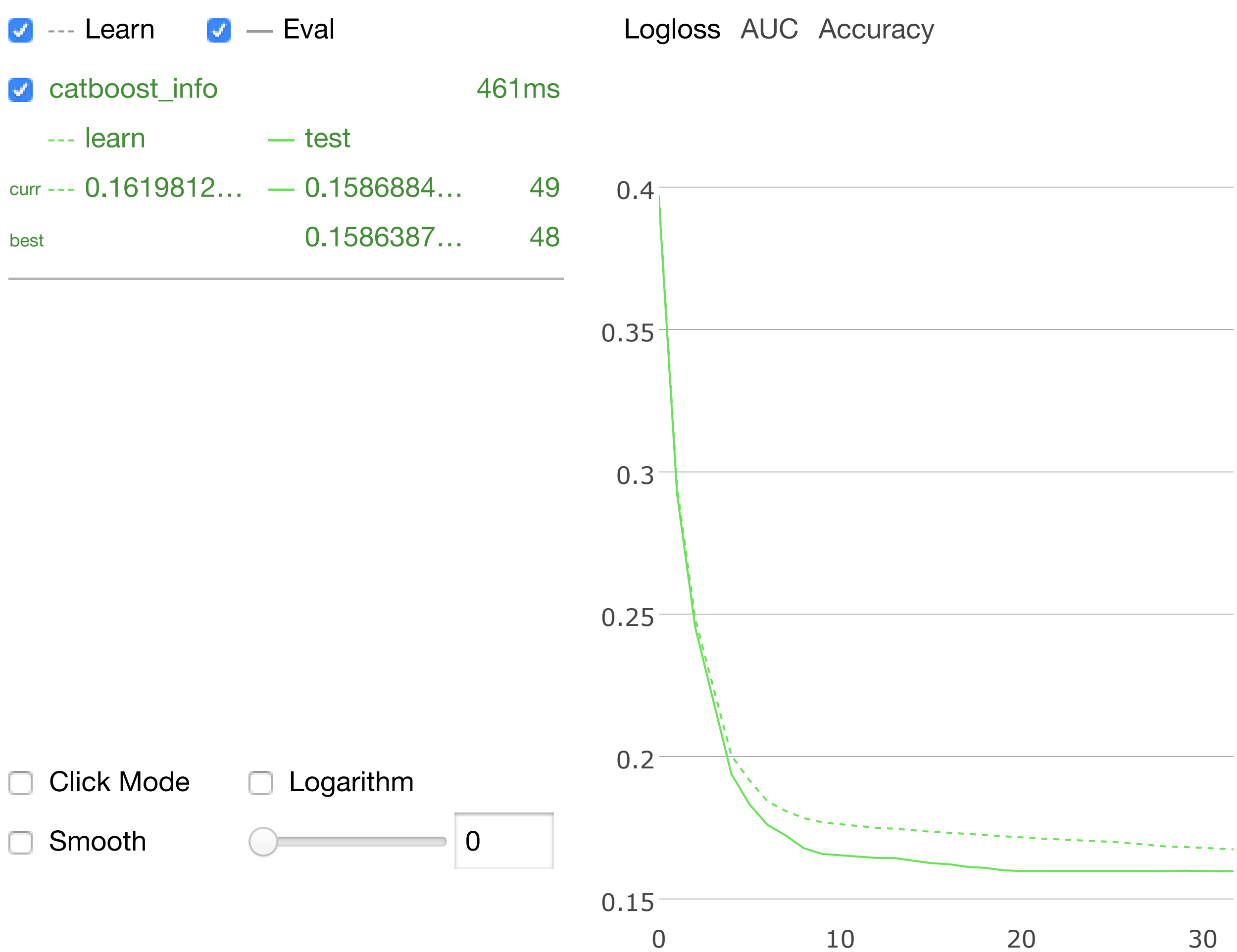
The learning rate is set to high value to prevent underfitting in the face of little amount of iterations, but we are underfitting anyway.

The number of best iteration based on validation error will at some point stope increasing in case of using many iterations.

Remaining time is a useful and accurate indication of how much we have to wait until the end of the training.

Metrics calculation and graph plotting

In [16]:
```python
model = CatBoostClassifier(
    iterations=50,
    learning_rate=0.5,
    custom_loss=['AUC', 'Accuracy']
)

model.fit(
    train_pool,
    eval_set=validation_pool,
    verbose=False,
    plot=True
);

# the dotted line is train error
# the solid line is validation error
```



Model comparison

The model with a big learning rate will overfit immediately.

In [17]:
```
model1 = CatBoostClassifier(
    learning_rate=0.7,
    iterations=100,
    train_dir='learing_rate_0.7'
)

model2 = CatBoostClassifier(
    learning_rate=0.01,
    iterations=100,
    train_dir='learing_rate_0.01'
)

model1.fit(train_pool, eval_set=validation_pool, verbose=20)
model2.fit(train_pool, eval_set=validation_pool, verbose=20);
```

```
0:      learn: 0.3264513        test: 0.3248170 best: 0.3248170 (0)     t
otal: 8.65ms    remaining: 856ms
20:     learn: 0.1683868        test: 0.1595638 best: 0.1594426 (19)    t
otal: 165ms     remaining: 621ms
40:     learn: 0.1623000        test: 0.1604900 best: 0.1592935 (29)    t
otal: 277ms     remaining: 399ms
60:     learn: 0.1566107        test: 0.1607838 best: 0.1592935 (29)    t
otal: 376ms     remaining: 240ms
80:     learn: 0.1526952        test: 0.1603468 best: 0.1592935 (29)    t
otal: 492ms     remaining: 116ms
99:     learn: 0.1485242        test: 0.1604420 best: 0.1592935 (29)    t
otal: 613ms     remaining: 0us

bestTest = 0.1592934723
bestIteration = 29

Shrink model to first 30 iterations.
0:      learn: 0.6853769        test: 0.6853610 best: 0.6853610 (0)     t
otal: 8.84ms    remaining: 875ms
20:     learn: 0.5575636        test: 0.5568281 best: 0.5568281 (20)    t
otal: 91.2ms    remaining: 343ms
40:     learn: 0.4678357        test: 0.4664710 best: 0.4664710 (40)    t
otal: 175ms     remaining: 252ms
60:     learn: 0.4029487        test: 0.4011429 best: 0.4011429 (60)    t
otal: 249ms     remaining: 159ms
80:     learn: 0.3552680        test: 0.3531846 best: 0.3531846 (80)    t
otal: 320ms     remaining: 75ms
99:     learn: 0.3210422        test: 0.3186747 best: 0.3186747 (99)    t
otal: 387ms     remaining: 0us

bestTest = 0.3186746933
bestIteration = 99
```

In [18]:
```python
from catboost import MetricVisualizer
MetricVisualizer(['learing_rate_0.7', 'learing_rate_0.01']).start()
```

--- Learn — Eval

learing_rate_0.7 612ms

--- learn — test

curr --- 0.148524209 — 0.1604420… 99

best 0.1592934… 29

learing_rate_0.01 386ms

--- learn — test

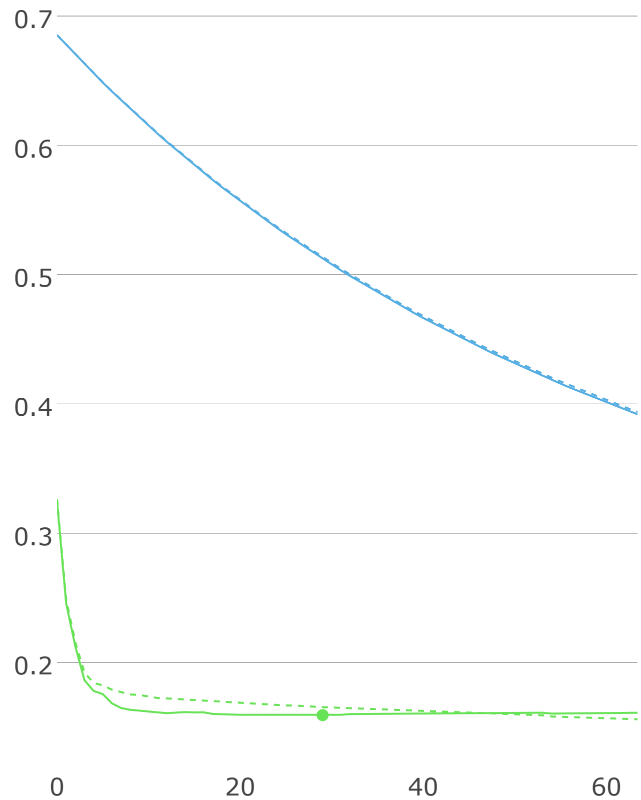
curr --- 0.3210422… — 0.3186746… 99

best 0.3186746… 99

Click Mode Logarithm

Smooth 0

Logloss

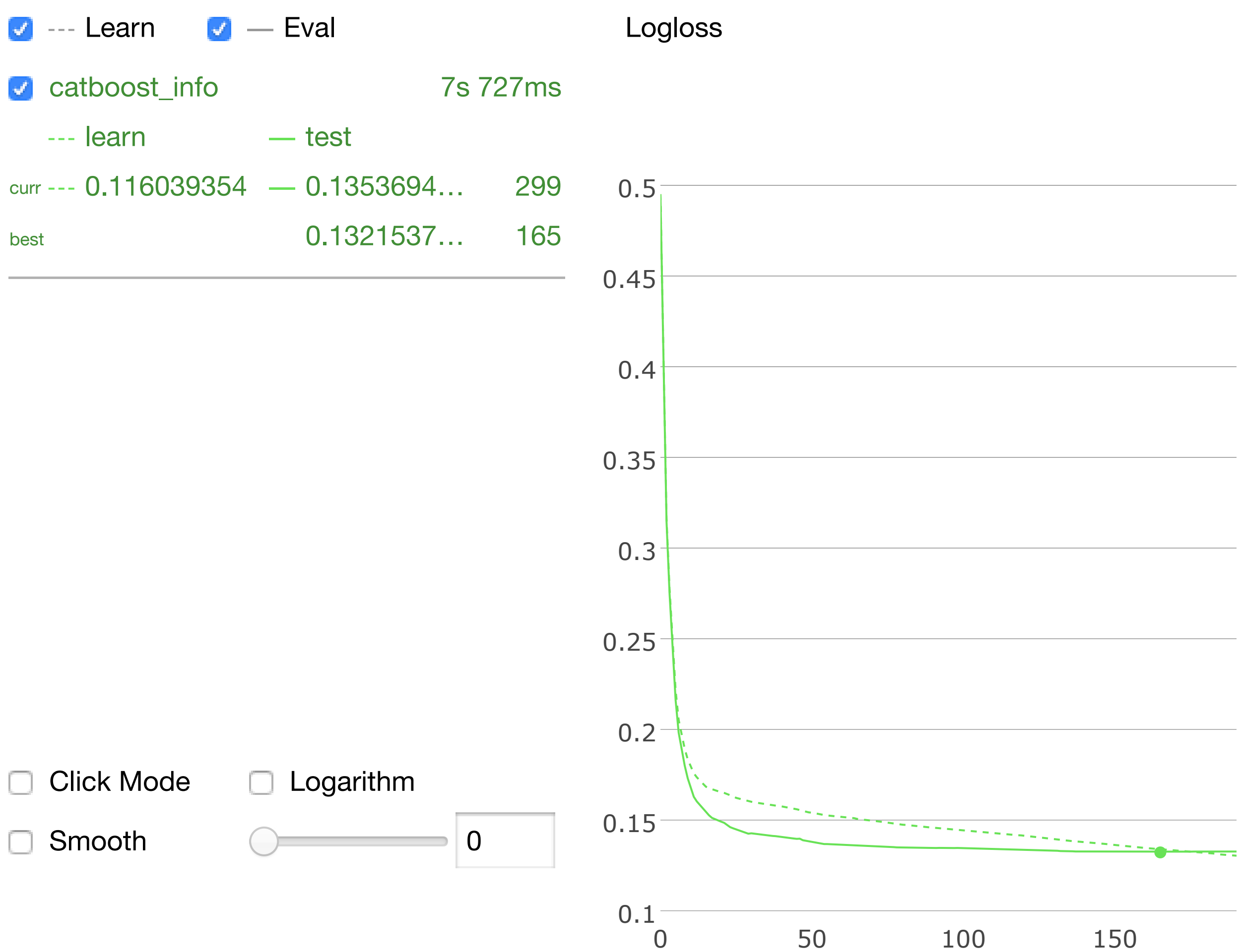


With high learning rate we overfit after like 25 iterations. All the trees after tree 25 just increase the validation error.

So we always want to cut our ensemble to the best iteration. And that is what CatBoost does by default.

Best iteration

In [19]:
```
%%time

# if we don't want to cut the ensemble we set use_best_model flag to False
model = CatBoostClassifier(
    iterations=300,
#     use_best_model=False
)

model.fit(
    train_pool,
    eval_set=validation_pool,
    verbose=False,
    plot=True
);
```




```
CPU times: user 31.7 s, sys: 8.03 s, total: 39.8 s
Wall time: 7.86 s
```

Out[19]: `<catboost.core.CatBoostClassifier at 0x1087b6c10>`

In [20]:
```
print('Tree count: ' + str(model.tree_count_))
```

```
Tree count: 166
```

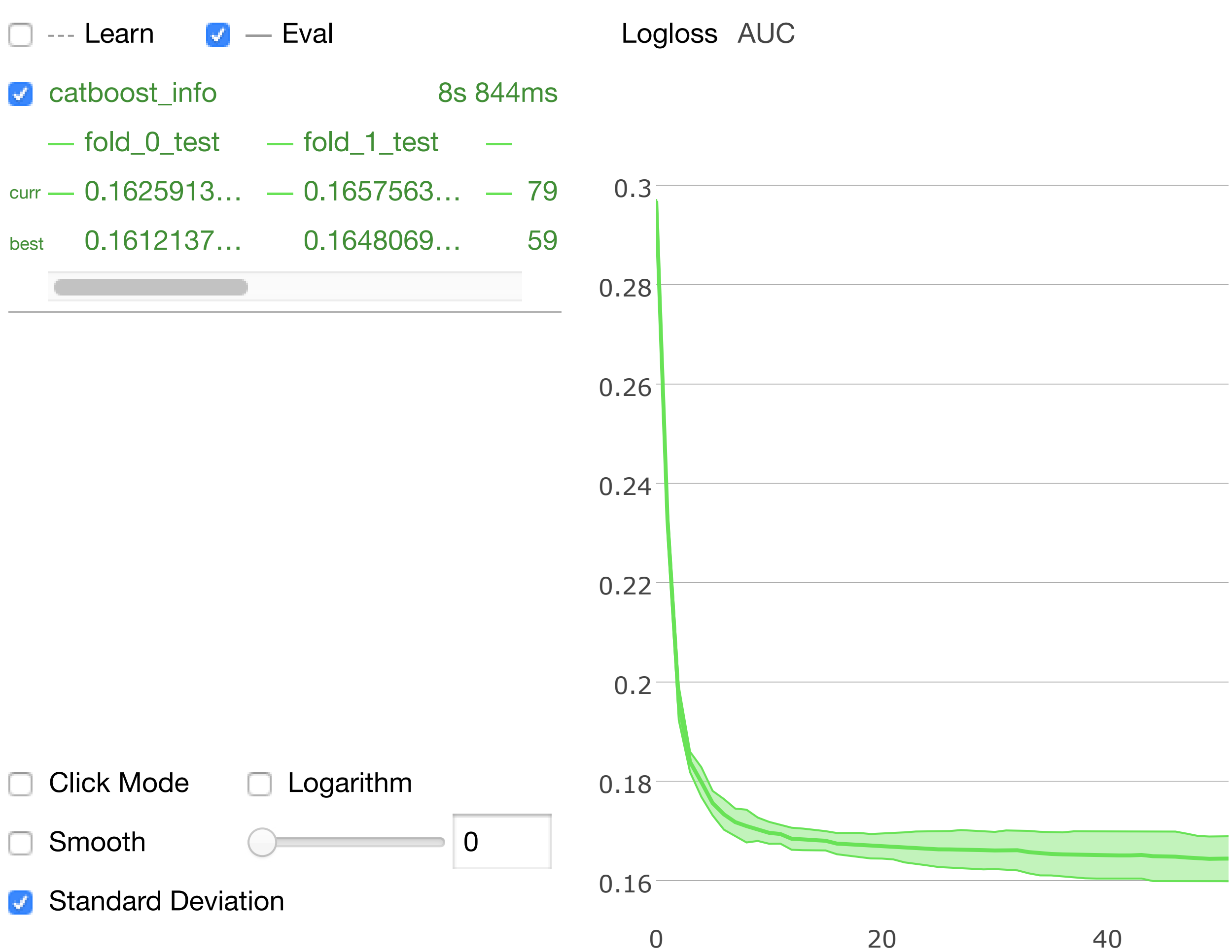
Cross-validation

Regular cross-validation shuffles the dataset and separates into several partitions (folds) and we perform 5 different folds for validation and training.

Stratified cross-validation makes sure that in each fold the classes are represented evenly. For imbalanced dataset stratification is necessary because on of the classes may be absent in the sample.

On the ouput graph we are averaging the validation errors of all the 5 trainings and we plot the mean ans standard deviation.

In [21]:
```python
from catboost import cv

params = {
    'loss_function': 'Logloss',
    'iterations': 80,
    'custom_loss': 'AUC',
    'learning_rate': 0.5,
}

# 5-fold cross-validation

cv_data = cv(
    params = params,
    pool = train_pool,
    fold_count=5,
    shuffle=True,
    partition_random_seed=0,
    plot=True,
    verbose=False
)
```



In [24]:
```
cv_data.head(10)
```

Out[24]:

	iterations	test-Logloss-mean	test-Logloss-std	train-Logloss-mean	train-Logloss-std	test-AUC-mean	test-AUC-std
0	0	0.296883	0.000377	0.299092	0.000514	0.553022	0.012227
1	1	0.232718	0.002843	0.233899	0.002472	0.585269	0.034036
2	2	0.195725	0.003420	0.202234	0.001366	0.769897	0.027029
3	3	0.183894	0.002052	0.193075	0.001714	0.793926	0.011553
4	4	0.179853	0.002984	0.189851	0.001177	0.800056	0.009765
5	5	0.175610	0.002463	0.186751	0.000560	0.811968	0.004840
6	6	0.173346	0.003080	0.185030	0.001422	0.817370	0.004980
7	7	0.171790	0.002720	0.183753	0.001353	0.821758	0.004355
8	8	0.170988	0.003301	0.182701	0.001891	0.825372	0.006429
9	9	0.170347	0.002355	0.182051	0.001876	0.828278	0.003643

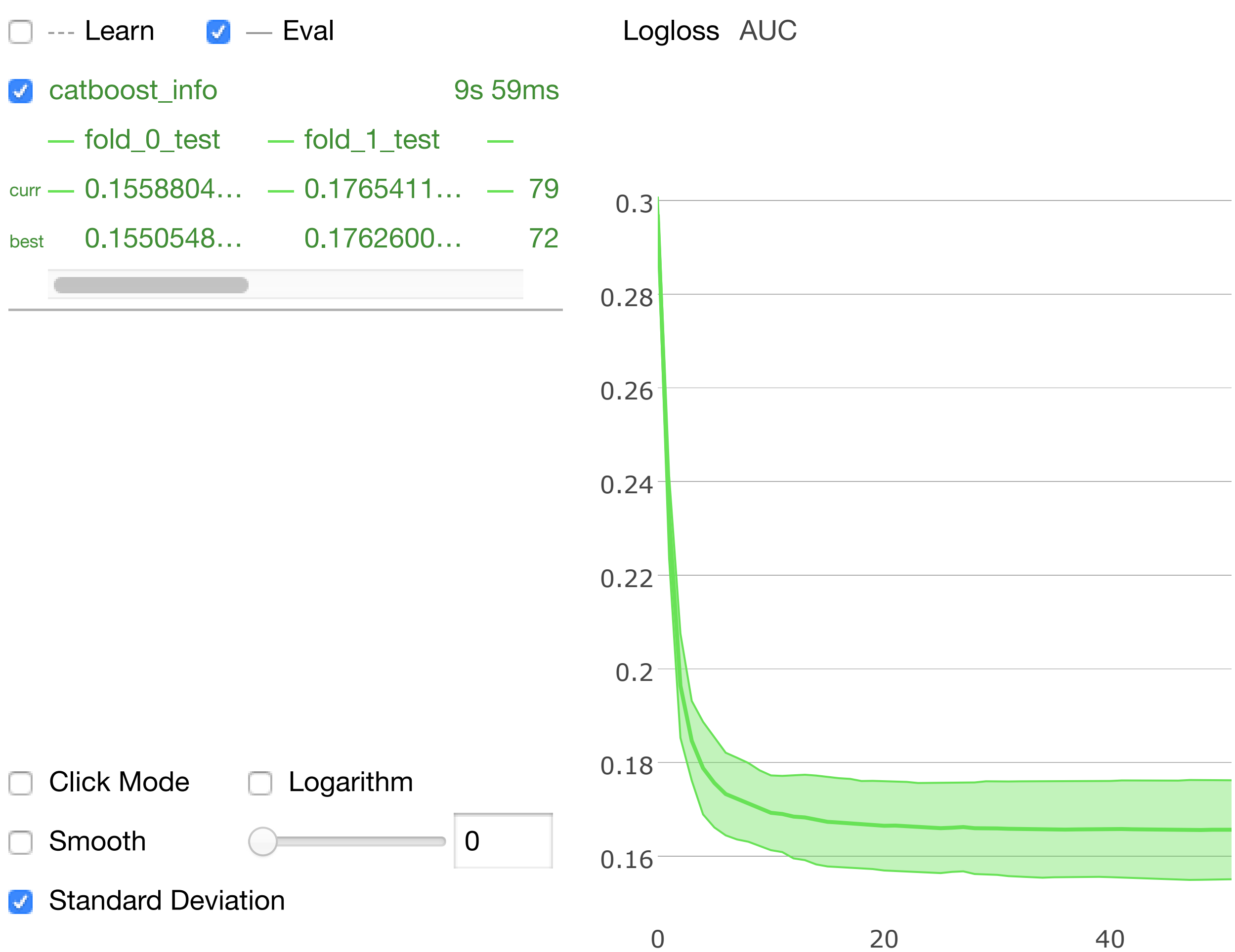
In [25]:
```
best_value = np.min(cv_data['test-Logloss-mean'])
best_iter = np.argmin(cv_data['test-Logloss-mean'])

print('Best validation Logloss score, not stratified: {:.4f}±{:.4f} on step
    best_value,
    cv_data['test-Logloss-std'][best_iter],
    best_iter)
)
```

```
Best validation Logloss score, not stratified: 0.1643±0.0046 on step 59
```

AUC is incredibly expensive to calculate and slows down the training. But we can calculate the expensive metric on every 10th iteration, not every time.

Important note: we've set stratified to False in this example, thus increasing the error.

In [26]:
```python
from catboost import cv

params = {
    'loss_function': 'Logloss',
    'iterations': 80,
    'custom_loss': 'AUC',
    'learning_rate': 0.5,
}

cv_data = cv(
    params = params,
    pool = train_pool,
    fold_count=5,
    shuffle=True,
    partition_random_seed=0,
    plot=True,
    # note that stratified is set to False
    stratified=False,
    verbose=False
)
```



In [27]:
```
cv_data.head(10)
```

Out[27]:

	iterations	test-Logloss-mean	test-Logloss-std	train-Logloss-mean	train-Logloss-std	test-AUC-mean	test-AUC-std
0	0	0.297006	0.003831	0.299197	0.002463	0.551899	0.013442
1	1	0.232408	0.008830	0.233780	0.002059	0.582268	0.037729
2	2	0.196398	0.011134	0.202640	0.003035	0.758677	0.049942
3	3	0.184656	0.008498	0.193563	0.002871	0.791557	0.015219
4	4	0.178804	0.009882	0.189045	0.002064	0.811128	0.009515
5	5	0.175633	0.009497	0.186465	0.002587	0.817355	0.008410
6	6	0.173271	0.008820	0.184606	0.002908	0.820856	0.010284
7	7	0.172298	0.008724	0.183976	0.003359	0.823645	0.008414
8	8	0.171498	0.008408	0.183364	0.003626	0.827071	0.007094
9	9	0.170316	0.008016	0.181797	0.004490	0.829916	0.005094

In [28]:
```
cv_data.tail(5)
```

Out[28]:

	iterations	test-Logloss-mean	test-Logloss-std	train-Logloss-mean	train-Logloss-std	test-AUC-mean	test-AUC-std
75	75	0.165463	0.010833	0.166431	0.003687	0.844698	0.013998
76	76	0.165497	0.010803	0.166384	0.003712	0.844603	0.013860
77	77	0.165495	0.010784	0.166350	0.003757	0.844607	0.013839
78	78	0.165395	0.010741	0.166202	0.003740	0.844840	0.013789
79	79	0.165371	0.010631	0.166134	0.003740	0.844936	0.013721

In [29]:
```
best_value = cv_data['test-Logloss-mean'].min()
best_iter = cv_data['test-Logloss-mean'].values.argmin()

print('Best validation Logloss score, stratified: {:.4f}±{:.4f} on step {}'
    best_value,
    cv_data['test-Logloss-std'][best_iter],
    best_iter)
)
```

```
Best validation Logloss score, stratified: 0.1654±0.0108 on step 72
```

Sklearn Grid Search

Grid search runs training with different parameters and selects model with the best cross-validation. Will go through just the learning rate.

In [30]:
```
%%time

from sklearn.model_selection import GridSearchCV

param_grid = {
    "learning_rate": [0.001, 0.01, 0.5],
}

clf = CatBoostClassifier(
    iterations=500,
    cat_features=cat_features,
    verbose=100
)

grid_search = GridSearchCV(clf, param_grid=param_grid, cv=3)

results = grid_search.fit(X_train, y_train)

results.best_estimator_.get_params()
```

```
0:	learn: 0.6919176	total: 15.3ms	remaining: 7.65s
100:	learn: 0.5818334	total: 1.93s	remaining: 7.64s
200:	learn: 0.4976909	total: 3.71s	remaining: 5.53s
300:	learn: 0.4333175	total: 5.61s	remaining: 3.71s
400:	learn: 0.3834238	total: 7.66s	remaining: 1.89s
499:	learn: 0.3454978	total: 9.5s	remaining: 0us
0:	learn: 0.6919115	total: 21.9ms	remaining: 11s
100:	learn: 0.5814778	total: 1.8s	remaining: 7.13s
200:	learn: 0.4969201	total: 3.49s	remaining: 5.19s
300:	learn: 0.4318546	total: 5.24s	remaining: 3.46s
400:	learn: 0.3820989	total: 7.09s	remaining: 1.75s
499:	learn: 0.3440486	total: 9.18s	remaining: 0us
0:	learn: 0.6919131	total: 21.7ms	remaining: 10.9s
100:	learn: 0.5818466	total: 1.7s	remaining: 6.72s
200:	learn: 0.4978899	total: 3.27s	remaining: 4.86s
300:	learn: 0.4326251	total: 5s	remaining: 3.3s
400:	learn: 0.3823160	total: 6.97s	remaining: 1.72s
499:	learn: 0.3439355	total: 8.99s	remaining: 0us
0:	learn: 0.6809382	total: 19.6ms	remaining: 9.79s
100:	learn: 0.2433405	total: 2.01s	remaining: 7.94s
200:	learn: 0.1873043	total: 4.51s	remaining: 6.71s
300:	learn: 0.1776406	total: 7.17s	remaining: 4.74s
400:	learn: 0.1725217	total: 10.1s	remaining: 2.5s
499:	learn: 0.1697762	total: 12.6s	remaining: 0us
0:	learn: 0.6808786	total: 21.7ms	remaining: 10.8s
100:	learn: 0.2420824	total: 2.04s	remaining: 8.04s
200:	learn: 0.1879515	total: 4.44s	remaining: 6.6s
300:	learn: 0.1774887	total: 6.97s	remaining: 4.61s
400:	learn: 0.1732318	total: 10s	remaining: 2.47s
499:	learn: 0.1704898	total: 13.5s	remaining: 0us
0:	learn: 0.6808948	total: 23.3ms	remaining: 11.6s
100:	learn: 0.2398495	total: 1.97s	remaining: 7.79s
200:	learn: 0.1856434	total: 4.28s	remaining: 6.36s
300:	learn: 0.1728246	total: 7.45s	remaining: 4.92s
400:	learn: 0.1678552	total: 10.7s	remaining: 2.64s
499:	learn: 0.1653753	total: 13.4s	remaining: 0us
0:	learn: 0.3028666	total: 23.5ms	remaining: 11.7s
```

```
100:	learn: 0.1484704	total: 3.16s	remaining: 12.5s
200:	learn: 0.1371610	total: 6.48s	remaining: 9.64s
300:	learn: 0.1315518	total: 9.58s	remaining: 6.33s
400:	learn: 0.1245843	total: 12.7s	remaining: 3.14s
499:	learn: 0.1202641	total: 15.6s	remaining: 0us
0:	learn: 0.3019334	total: 21ms	remaining: 10.5s
100:	learn: 0.1502387	total: 3s	remaining: 11.9s
200:	learn: 0.1368593	total: 6.07s	remaining: 9.03s
300:	learn: 0.1291684	total: 9.07s	remaining: 6s
400:	learn: 0.1191529	total: 12.1s	remaining: 2.98s
499:	learn: 0.1130065	total: 15s	remaining: 0us
0:	learn: 0.3021188	total: 20.8ms	remaining: 10.4s
100:	learn: 0.1439649	total: 3.05s	remaining: 12s
200:	learn: 0.1327602	total: 6.08s	remaining: 9.05s
300:	learn: 0.1246817	total: 9.14s	remaining: 6.04s
400:	learn: 0.1170070	total: 12.1s	remaining: 2.99s
499:	learn: 0.1129490	total: 15s	remaining: 0us
0:	learn: 0.6806462	total: 43.1ms	remaining: 21.5s
100:	learn: 0.2367254	total: 2.6s	remaining: 10.3s
200:	learn: 0.1805753	total: 5.68s	remaining: 8.45s
300:	learn: 0.1685944	total: 9.28s	remaining: 6.14s
400:	learn: 0.1640030	total: 13.2s	remaining: 3.25s
499:	learn: 0.1616702	total: 16.8s	remaining: 0us
CPU times: user 9min 14s, sys: 2min 6s, total: 11min 21s
Wall time: 2min 11s
```

Out[30]:

```
{'iterations': 500,
 'verbose': 100,
 'cat_features': [0, 1, 2, 3, 4, 5, 6, 7, 8],
 'learning_rate': 0.01}
```

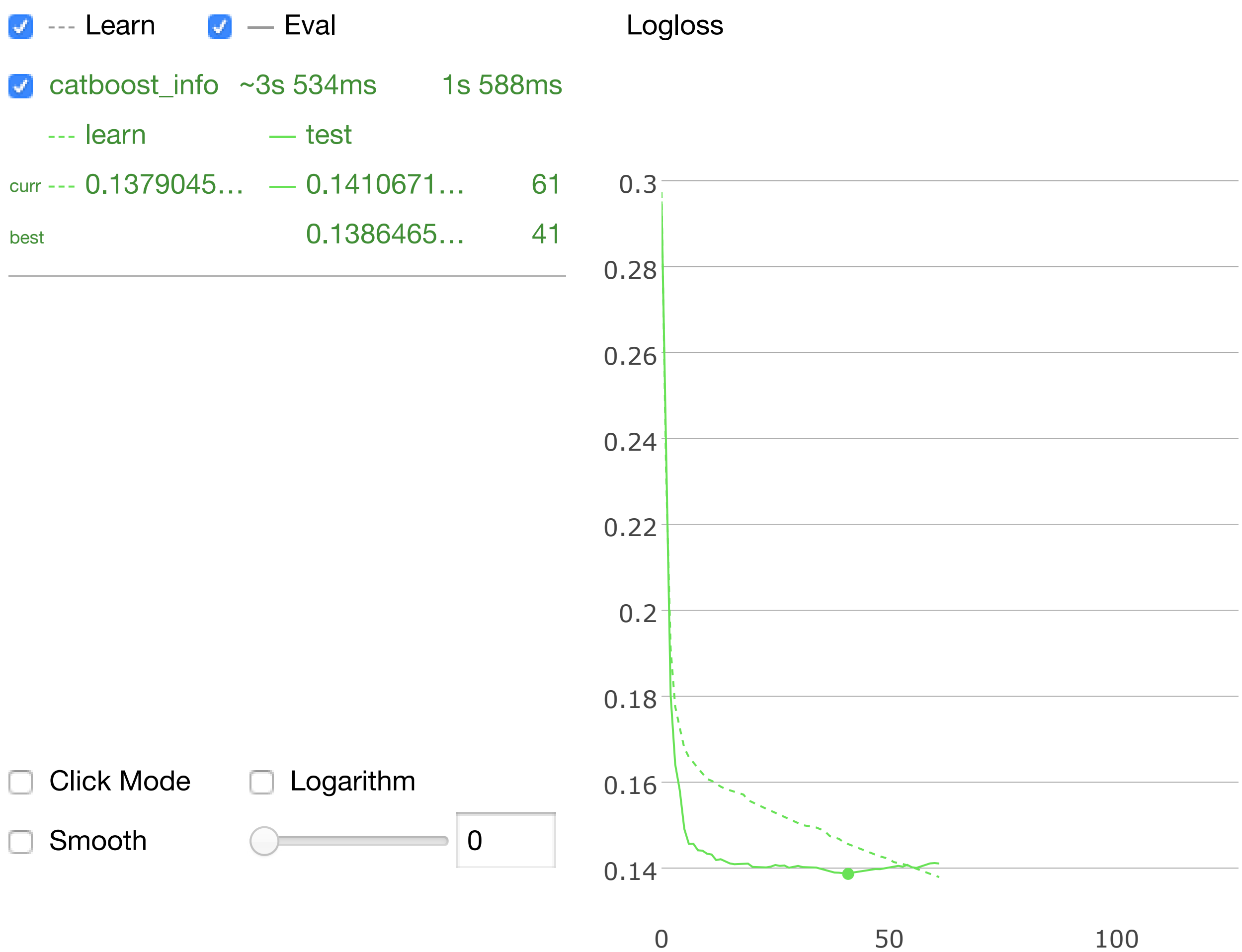
Overfitting Detector

When we set early_stopping_rounds=20 that means that we would like to stop training if the validation error does not improve after 20 iterations.

If we are training a model with like thousands of trees, 50 is a good vallue.

Side note: 0.03 is a default value for the learning rate for situations with non-binary classification.

In [31]:
```python
model_with_early_stop = CatBoostClassifier(
    iterations=200,
    learning_rate=0.5,
    early_stopping_rounds=20
)

model_with_early_stop.fit(
    train_pool,
    eval_set=validation_pool,
    verbose=False,
    plot=True
);
```



In [32]:
```python
print(model_with_early_stop.tree_count_)
```

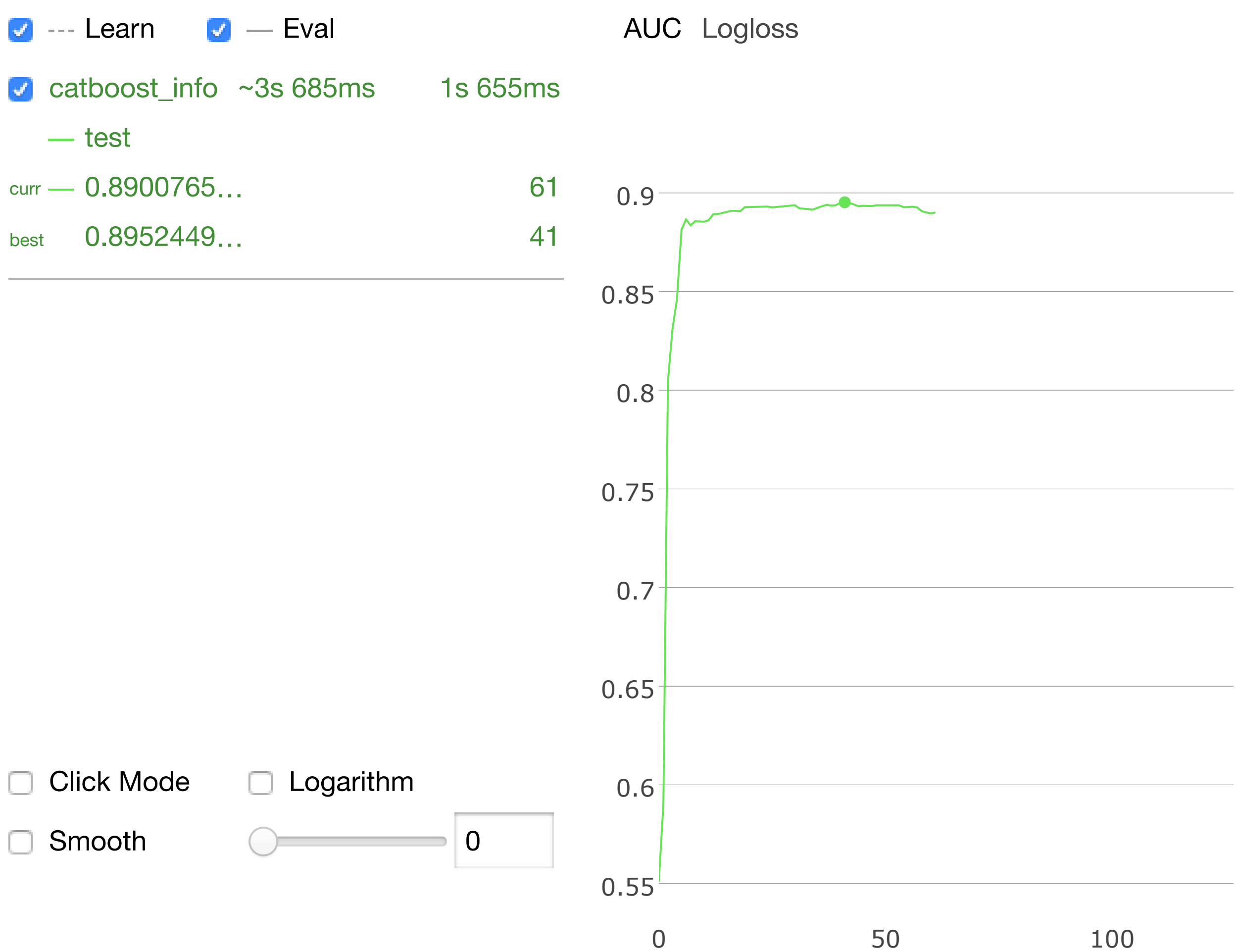
```
42
```

Overfitting Detector with eval metric

For binary classification the default metric is Logloss, but we can use a custom metric using eval_metric and perform early stopping with ensemble cutting.

In the case below AUC will be used for both overfitting detector and for early stopping.

In [33]:
```python
model_with_early_stop = CatBoostClassifier(
    eval_metric='AUC',
    iterations=200,
    learning_rate=0.5,
    early_stopping_rounds=20
)

model_with_early_stop.fit(
    train_pool,
    eval_set=validation_pool,
    verbose=False,
    plot=True
);
```



In [40]:
```python
print(model_with_early_stop.tree_count_)
```

```
42
```

Model predictions

In [41]:
```
model = CatBoostClassifier(iterations=200, learning_rate=0.03)
model.fit(train_pool, verbose=50);
```

```
0:      learn: 0.6562528        total: 16ms     remaining: 3.18s
50:     learn: 0.1918730        total: 809ms    remaining: 2.36s
100:    learn: 0.1661941        total: 1.89s    remaining: 1.85s
150:    learn: 0.1599724        total: 3.05s    remaining: 990ms
199:    learn: 0.1572903        total: 4.3s     remaining: 0us
```

The predict method outputs a class value for the object. It is either 0 or 1 in case of binary classification. In this case the default border of 0.5 probability is used.

In [42]:
```
print(model.predict(X_validation))
```

```
[1. 1. 1. ... 1. 1. 1.]
```

In [43]:
```
print(model.predict_proba(X_validation))
```

```
[[0.0278 0.9722]
 [0.0201 0.9799]
 [0.0109 0.9891]
 ...
 [0.0316 0.9684]
 [0.047  0.953 ]
 [0.0221 0.9779]]
```

In case of binary classification CatBoost uses sigmoid to output the probability of an object to be a certain class. It is important to remember that the sum of the leaf values that is obtained from the emsemble of trees is just a real value, and we calculate the sigmoid on top of that.

Raw predictions are important when we are trying to analyze the model.

In [44]:
```
raw_pred = model.predict(
    X_validation,
    prediction_type='RawFormulaVal'
)

print(raw_pred)
```

```
[3.556  3.8871 4.5108 ... 3.4235 3.0102 3.7883]
```

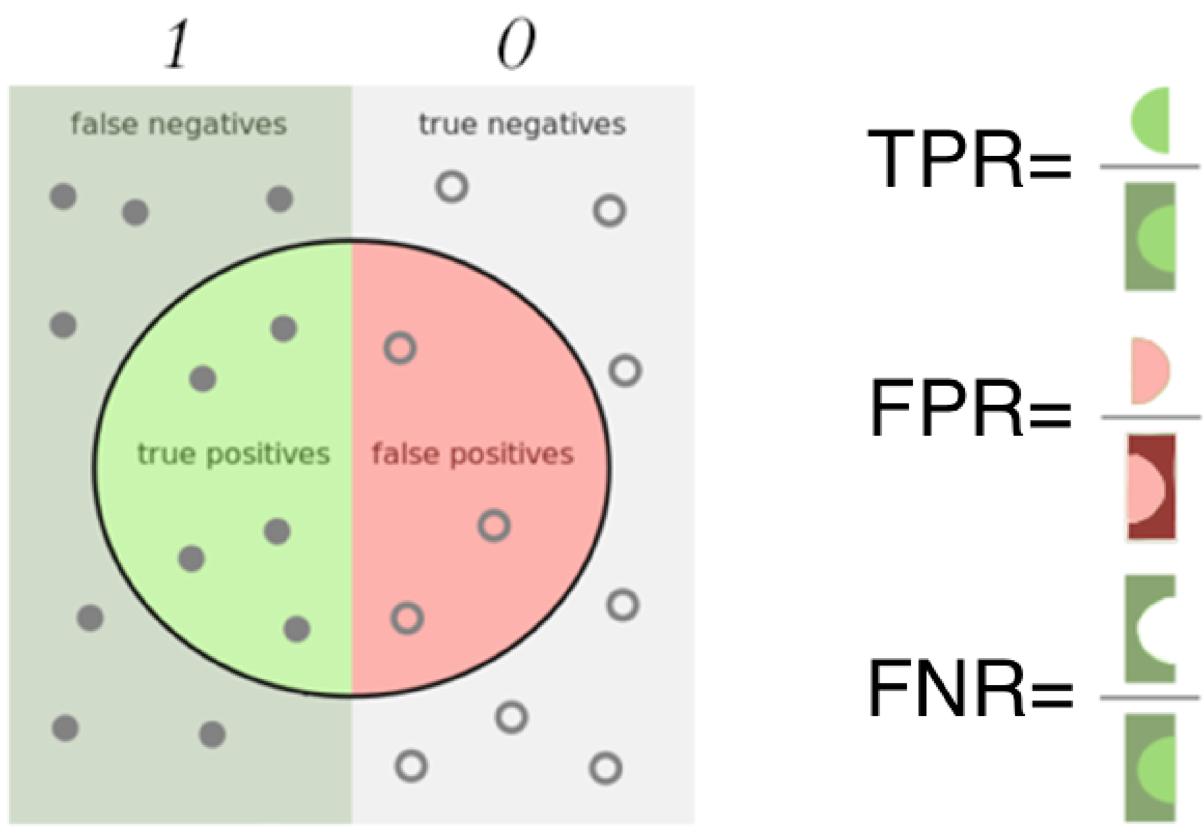
So we will write our own sigmoid function and check that sigmoids of the raw output correspond to the predict_proba values above.

In [45]:
```
from numpy import exp

sigmoid = lambda x: 1 / (1 + exp(-x))

probabilities = sigmoid(raw_pred)

print(probabilities)
```

```
[0.9722 0.9799 0.9891 ... 0.9684 0.953  0.9779]
```

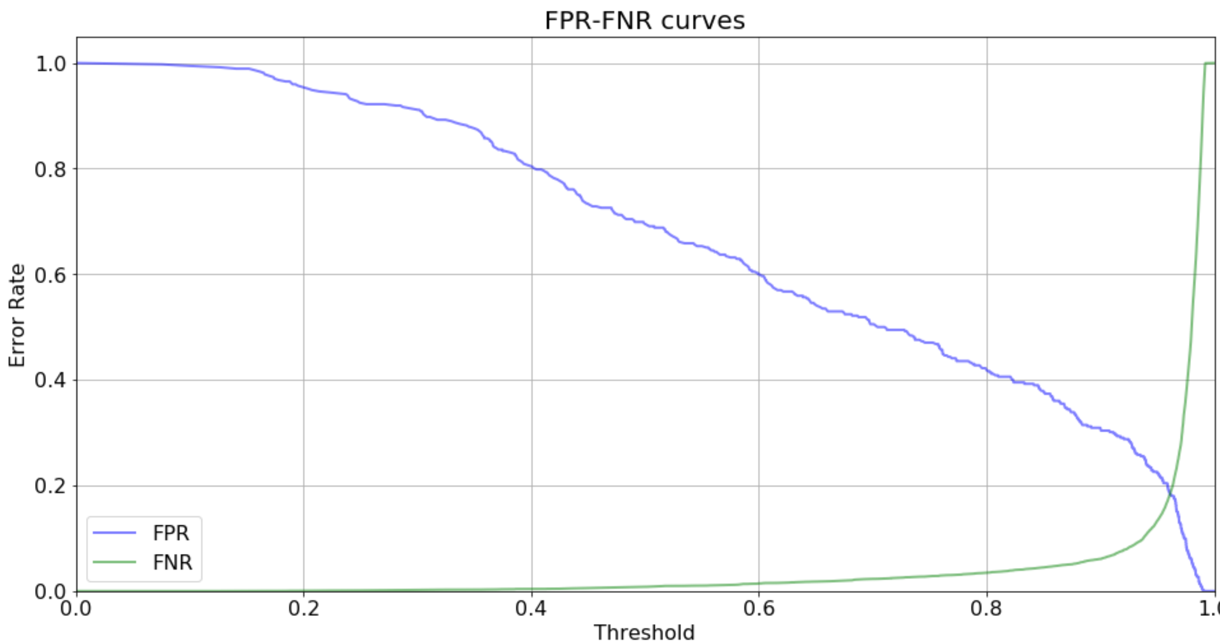
Select decision boundary

How we can choose better border that 0.5?



In [46]:
```
import matplotlib.pyplot as plt
from catboost.utils import get_roc_curve
from catboost.utils import get_fpr_curve
from catboost.utils import get_fnr_curve

curve = get_roc_curve(model, validation_pool)
(fpr, tpr, thresholds) = curve

(thresholds, fpr) = get_fpr_curve(curve=curve)
(thresholds, fnr) = get_fnr_curve(curve=curve)
```

In [47]:
```
plt.figure(figsize=(16, 8))
style = {'alpha':0.5, 'lw':2}

plt.plot(thresholds, fpr, color='blue', label='FPR', **style)
plt.plot(thresholds, fnr, color='green', label='FNR', **style)

plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xticks(fontsize=16)
plt.yticks(fontsize=16)
plt.grid(True)
plt.xlabel('Threshold', fontsize=16)
plt.ylabel('Error Rate', fontsize=16)
plt.title('FPR-FNR curves', fontsize=20)
plt.legend(loc="lower left", fontsize=16);
```



We pass the FNR or the FPR that we are ok to allow into select threshold function and get the desired threshold.

In [48]:
```
from catboost.utils import select_threshold

print(select_threshold(model, validation_pool, FNR=0.01))
print(select_threshold(model, validation_pool, FPR=0.01))
```

```
0.5348175863612155
0.9885678343152173
```

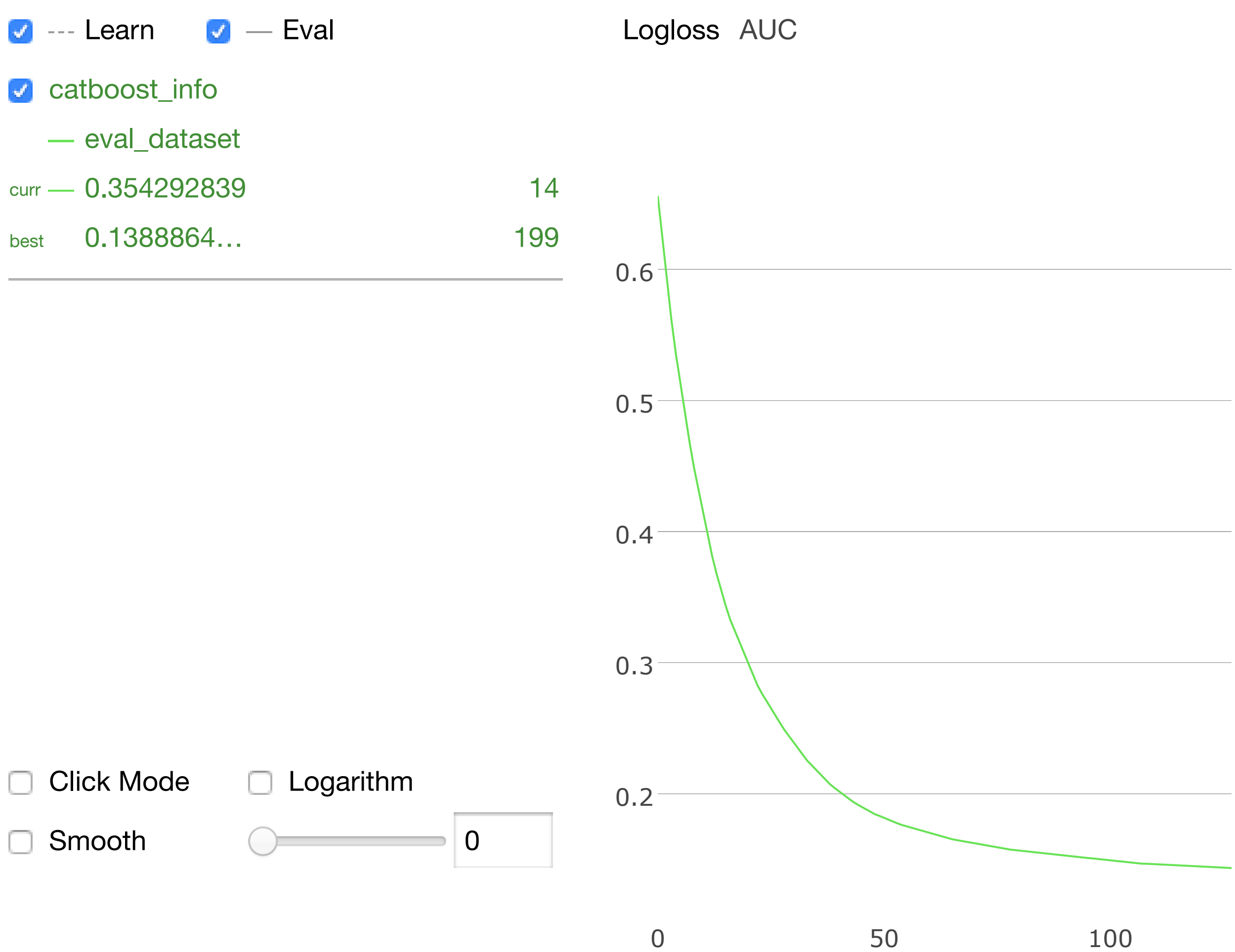
And we get very different thresholds!

Metric evaluation on a new dataset

Now will analyze the model!

eval_period is at which each iteration we evaluate the metric.

In [49]:
```
metrics = model.eval_metrics(
    data=validation_pool,
    metrics=['Logloss','AUC'],
    ntree_start=0,
    ntree_end=0,
    eval_period=1,
    plot=True
)
```



In [50]:
```
print('AUC values:\n{}'.format(np.array(metrics['AUC'])))
```

```
AUC values:
[0.5509 0.5509 0.6438 0.6438 0.633  0.633  0.6536 0.6536 0.6529 0.6528
 0.6843 0.6843 0.7023 0.7231 0.729  0.7287 0.7264 0.7264 0.7293 0.7282
 0.7323 0.7407 0.7442 0.7463 0.7518 0.7515 0.7514 0.7633 0.7784 0.7832
 0.792  0.7998 0.8072 0.8209 0.8232 0.8312 0.8337 0.8365 0.8393 0.8406
 0.8417 0.8449 0.8458 0.8469 0.8481 0.8491 0.8497 0.8507 0.8505 0.8503
 0.8504 0.8515 0.8527 0.853  0.8531 0.8528 0.8528 0.8532 0.8536 0.8538
 0.8539 0.854  0.8552 0.8561 0.8568 0.8575 0.8574 0.8578 0.8582 0.8592
 0.8624 0.8646 0.866  0.8673 0.8682 0.8693 0.8704 0.8709 0.8709 0.8712
 0.8721 0.8731 0.8728 0.8739 0.8742 0.8751 0.8757 0.8764 0.8767 0.8771
 0.8776 0.8784 0.879  0.8796 0.88   0.8806 0.881  0.8822 0.8832 0.883
 0.8841 0.8847 0.8861 0.8868 0.8876 0.8877 0.8886 0.8892 0.8892 0.8895
 0.89   0.8903 0.8904 0.8909 0.891  0.8911 0.8915 0.8914 0.8915 0.8913
 0.8913 0.8917 0.8919 0.892  0.8923 0.8923 0.8926 0.8927 0.8929 0.8931
 0.8929 0.893  0.8931 0.8933 0.8933 0.8934 0.8935 0.8936 0.8937 0.8939
 0.8939 0.8943 0.8943 0.8943 0.8943 0.8943 0.8944 0.8944 0.8945 0.8944
 0.8945 0.8945 0.8946 0.8947 0.8947 0.8948 0.8948 0.8948 0.8948 0.8949
 0.8949 0.895  0.895  0.895  0.8951 0.8951 0.8951 0.8951 0.895  0.895
 0.8951 0.8951 0.8954 0.8954 0.8955 0.8956 0.8955 0.8955 0.8955 0.8956
 0.8956 0.8957 0.8957 0.8957 0.8957 0.8958 0.896  0.896  0.8959 0.896
 0.8961 0.8961 0.8961 0.896  0.896  0.896  0.8961 0.8961 0.8961 0.8961]
```

Feature importances

Prediction values change

Default feature importances for binary classification is PredictionValueChange - how much on average does the model change when the feature value changes. These feature importances are non negative. They are normalized and sum to 1, so you can look on these values like percentage of importance.

These feature importances are really useful, but for some cases like ranking modes those feature importances may be misleading.

In [51]:
```
np.array(model.get_feature_importance(prettified=True))
```

Out[51]:
```
array([['ROLE_DEPTNAME', 18.097358505877374],
       ['RESOURCE', 17.007672800864107],
       ['MGR_ID', 15.862823716064335],
       ['ROLE_FAMILY_DESC', 9.90442304977881],
       ['ROLE_TITLE', 9.14577006148845],
       ['ROLE_ROLLUP_2', 8.92813526946003],
       ['ROLE_CODE', 8.26251859955471],
       ['ROLE_FAMILY', 8.177698118200851],
       ['ROLE_ROLLUP_1', 4.6135998787114065]], dtype=object)
```

Loss function change

The non default feature importance approximates how much the optimized loss function will change if the value of the feature changes. This importances might be negative if the feature has bad influence on the loss function. The importances are not normalized, the absolute value of the importance has the same scale as the optimized loss value. To calculate this importance value you need to pass train_pool as an argument.

In [52]:
```
np.array(model.get_feature_importance(
    train_pool,
    'LossFunctionChange',
    prettified=True
))
```

Out[52]:
```
array([['RESOURCE', 0.019038982743035844],
       ['MGR_ID', 0.015818393592726486],
       ['ROLE_FAMILY_DESC', 0.008814099362800407],
       ['ROLE_DEPTNAME', 0.007849253135864176],
       ['ROLE_TITLE', 0.007443902692415564],
       ['ROLE_ROLLUP_2', 0.005479810070508932],
       ['ROLE_CODE', 0.004590483798868571],
       ['ROLE_FAMILY', 0.002286688589620905],
       ['ROLE_ROLLUP_1', 0.00011584285785687146]], dtype=object)
```

Shap values

Shap values are calculated per object. So each object has a set of his shap values. All the importances sum up to the prediction of the object.

In [53]:
```
print(model.predict_proba([X.iloc[1,:]]))
print(model.predict_proba([X.iloc[91,:]]))
```

```
[[0.0114 0.9886]]
[[0.375 0.625]]
```

In [54]:
```
shap_values = model.get_feature_importance(
    validation_pool,
    'ShapValues'
)
expected_value = shap_values[0,-1]
shap_values = shap_values[:,:-1]
print(shap_values.shape)
```

```
(6554, 9)
```

In [57]:
```
proba = model.predict_proba([X.iloc[1,:]])[0]
raw = model.predict([X.iloc[1,:]], prediction_type='RawFormulaVal')[0]
print('Probabilities', proba)
print('Raw formula value %.4f' % raw)
print('Probability from raw value %.4f' % sigmoid(raw))
```

```
Probabilities [0.0114 0.9886]
Raw formula value 4.4627
Probability from raw value 0.9886
```

In [59]:
```
# was not able to import shap libraries


# import shap as

# shap.initjs()
# shap.force_plot(expected_value, shap_values[1,:], X_validation.iloc[1,:])
```

In [60]:
```
proba = model.predict_proba([X.iloc[91,:]])[0]
raw = model.predict([X.iloc[91,:]], prediction_type='RawFormulaVal')[0]
print('Probabilities', proba)
print('Raw formula value %.4f' % raw)
print('Probability from raw value %.4f' % sigmoid(raw))
```

```
Probabilities [0.375 0.625]
Raw formula value 0.5109
Probability from raw value 0.6250
```

In [ ]:
```
# import shap
# shap.initjs()
# shap.force_plot(expected_value, shap_values[91,:], X_validation.iloc[91,:
```

In [62]:
```
# shap.summary_plot(shap_values, X_validation)
```

Snapshotting

If you have long training it is good idea to use snapshotting. So we can proceed from the point where everything stopped

In [66]:
```
# !rm 'catboost_info/snapshot.bkp'

model = CatBoostClassifier(
    iterations=100,
    save_snapshot=True,
    snapshot_file='snapshot.bkp',
    snapshot_interval=1
)

model.fit(train_pool, eval_set=validation_pool, verbose=10);
```

```
Learning rate set to 0.294577

bestTest = 0.1559378784
bestIteration = 97

Shrink model to first 98 iterations.
```

Saving the model

We can save model as a binary and than can use for production.

In [67]:
```
model = CatBoostClassifier(iterations=10)
model.fit(train_pool, eval_set=validation_pool, verbose=False)
model.save_model('catboost_model.bin')
model.save_model('catboost_model.json', format='json')
```

In [69]:
```
model.load_model('catboost_model.bin')
print('The parameters of the model: ', model.get_params(), '\n')
print(model.learning_rate_)
```

```
The parameters of the model:  {'iterations': 10, 'loss_function': 'Loglos
s', 'logging_level': 'Silent', 'verbose': 0}

0.5
```

Hyperparameter tunning

The most important parameter is learning rate (in pair with iterations). So we want our model to overfit a little bit.

The other important thing is treeth depth. The default value of the trees is 6, but we may also try the value 10. In some cases it just works better. When choosing the number of leaves we should remember that catboost uses full binary trees and the number of leaves equals to 2 to the power of deprth. If we choose depth = 16 we get a lot of leaves.

In [72]:
```
tunned_model = CatBoostClassifier(
    iterations=4000,
    learning_rate=0.03,
    depth=6,
    l2_leaf_reg=3,
    random_strength=1,
    bagging_temperature=1
)

tunned_model.fit(
    X_train, y_train,
    cat_features=cat_features,
    verbose=False,
    eval_set=(X_validation, y_validation),
    plot=True
);
```

--- Learn — Eval

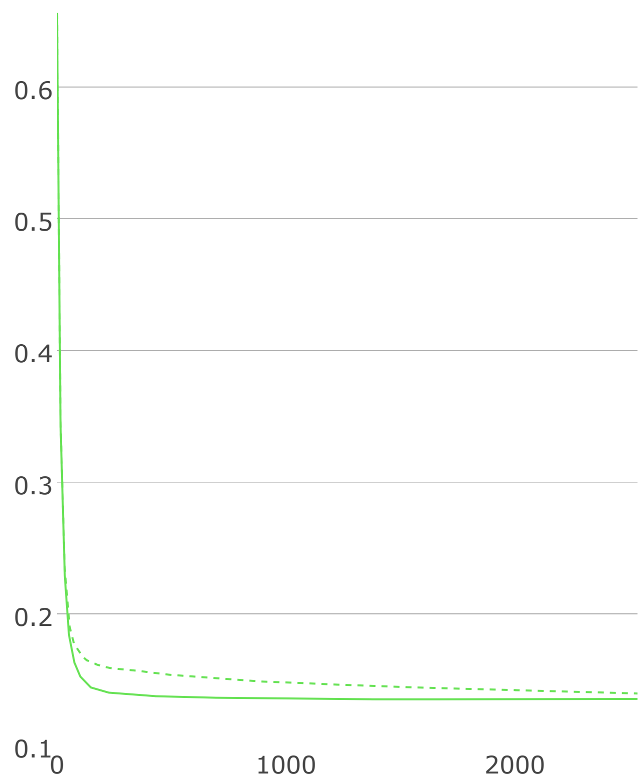
catboost_info 41s 722ms

	--- learn	— test	
curr	--- 0.1483741…	— 0.135854889	942
best		0.1350011…	2667

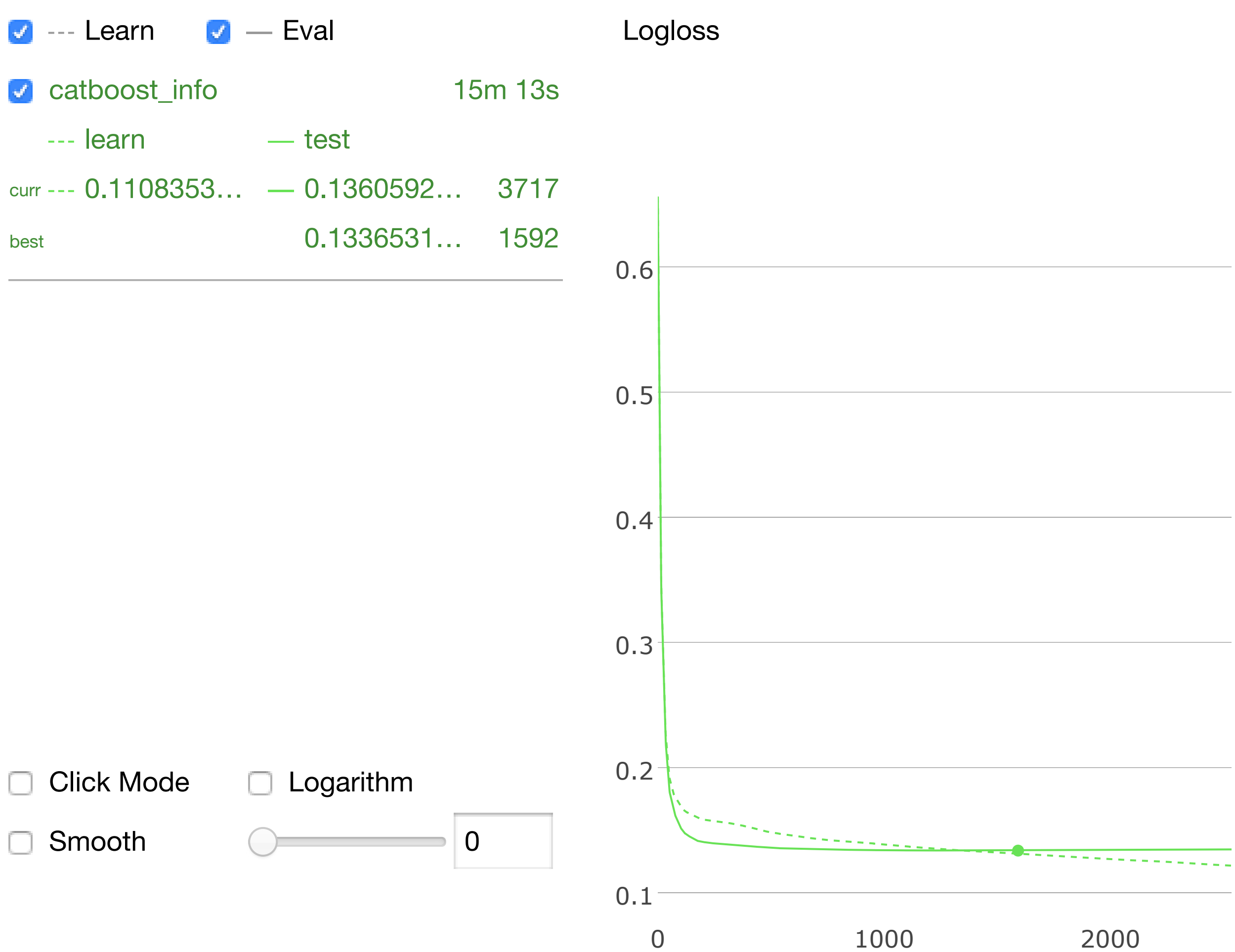
Click Mode Logarithm

Smooth 0

Logloss



In [73]:
```
tunned_model = CatBoostClassifier(
    iterations=4000,
    learning_rate=0.03,
    depth=10,
    l2_leaf_reg=3,
    random_strength=1,
    bagging_temperature=1
)

tunned_model.fit(
    X_train, y_train,
    cat_features=cat_features,
    verbose=False,
    eval_set=(X_validation, y_validation),
    plot=True
);
```

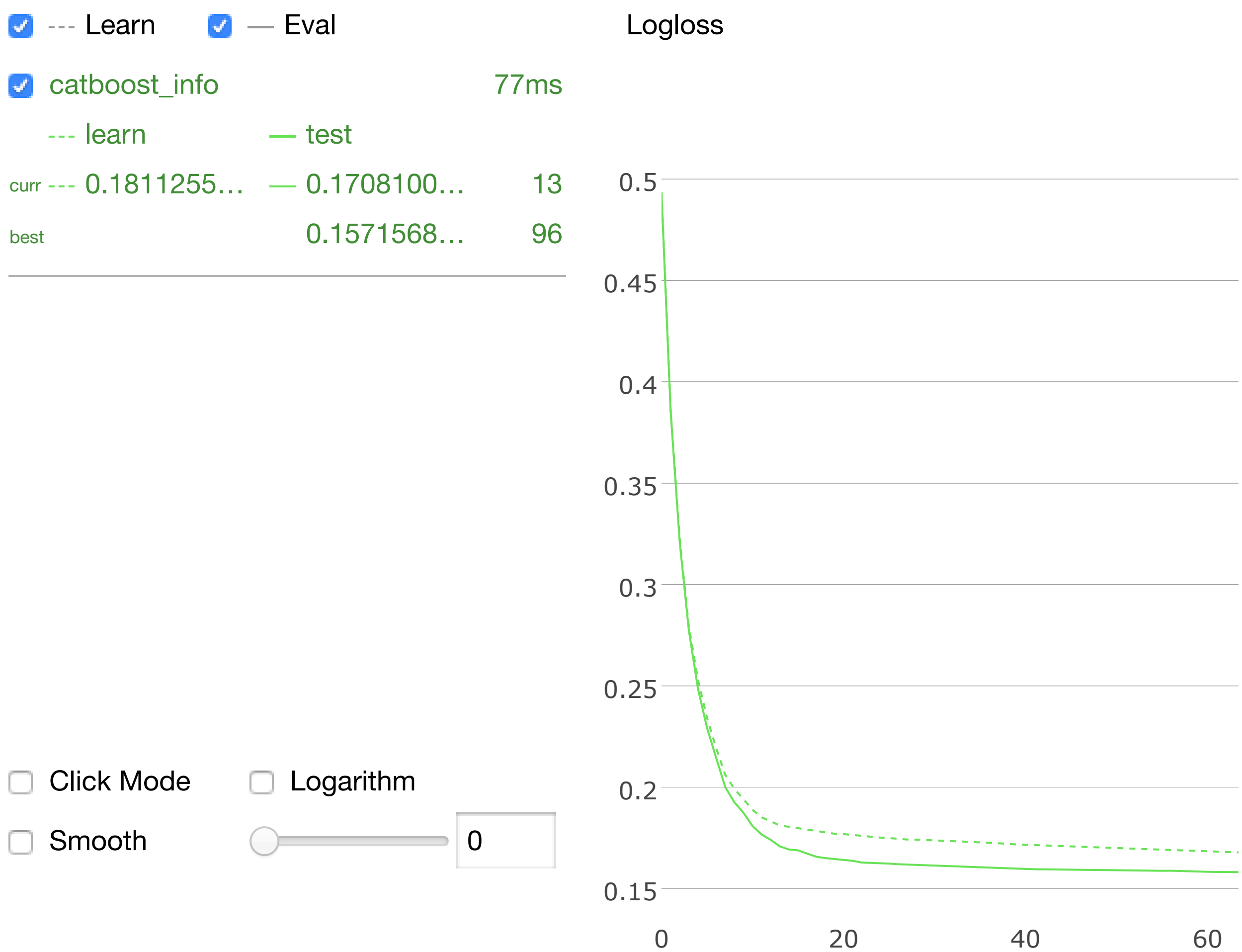


There are several strategies for tree growth in the CatBoost.

Speeding up the training

In [74]:

```
fast_model = CatBoostClassifier(
    boosting_type='Plain',
    rsm=0.5,
    one_hot_max_size=50,
    leaf_estimation_iterations=1,
    max_ctr_complexity=1,
    iterations=100,
    learning_rate=0.3,
    bootstrap_type='Bernoulli',
    subsample=0.5
)
fast_model.fit(
    X_train, y_train,
    cat_features=cat_features,
    verbose=False,
    eval_set=(X_validation, y_validation),
    plot=True
);
```



Reducing model size

In [75]:
```
small_model = CatBoostClassifier(
    learning_rate=0.03,
    iterations=500,
    model_size_reg=50,
    max_ctr_complexity=1,
    ctr_leaf_count_limit=100
)
small_model.fit(
    X_train, y_train,
    cat_features=cat_features,
    verbose=False,
    eval_set=(X_validation, y_validation),
    plot=True
);
```

--- Learn — Eval

catboost_info 1s 332ms

	--- learn	— test	
curr	--- 0.2255052...	— 0.2221050...	58
best		0.2076840...	351

Click Mode Logarithm

Smooth 0