Fast Gradient Boosting with Numba

Nicolas Hug, Columbia University -- @hug_nicolas

Olivier Grisel, INRIA Paris -- @ogrisel

pygbm: fast gradient boosting in Python with `Numba`

Outline

1. Gradient Boosting is a gradient descent
2. Making it fast with histograms
3. `Numba`: field report (and a tiny bit of `Cython`)

slides at https://bit.ly/2YFaIPO

Gradient Boosting

$$\hat{y}_i = \sum_{m=1}^{\text{n\_iter}} h_m(\mathbf{x}_i)$$

h_m: "weak" estimator that predicts **gradients** (not the target)

This is a gradient descent!

Linear regression

$$L = \sum_i (y_i - \hat{y}_i)^2$$

$\hat{y}_i = x_i^\top \theta$

Gradient boosting

$$L = \sum_i (y_i - \hat{y}_i)^2$$

Linear regression

$$L = \sum_i (y_i - \hat{y}_i)^2$$

$\hat{y}_i = x_i^\top \theta$

Optimize $\theta \in \mathbb{R}^d$

Gradient boosting

$$L = \sum_i (y_i - \hat{y}_i)^2$$

Optimize $\hat{y} = (\hat{y}_1, \dots \hat{y}_n) \in \mathbb{R}^n$

Linear regression

$$L = \sum_i (y_i - \hat{y}_i)^2$$

$\hat{y}_i = x_i^\top \theta$

Optimize $\theta \in \mathbb{R}^d$

$$\theta^{(m+1)} = \theta^{(m)} - \alpha \frac{\partial L}{\partial \theta^{(m)}}$$

Gradient boosting

$$L = \sum_i (y_i - \hat{y}_i)^2$$

Optimize $\hat{y} = (\hat{y}_1, \dots \hat{y}_n) \in \mathbb{R}^n$

$$\hat{y}^{(m+1)} = \hat{y}^{(m)} - \alpha \frac{\partial L}{\partial \hat{y}^{(m)}}$$

Gradient Descent

Hack # 1

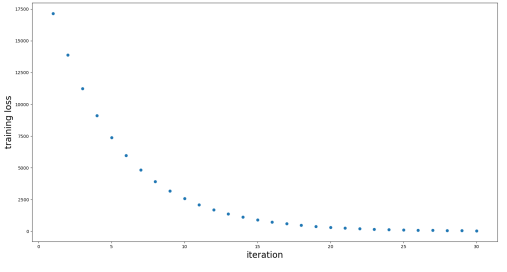
Optimize w.r.t. predictions: $\hat{\mathbf{y}}^{(m+1)} = \hat{\mathbf{y}}^{(m)} - \alpha \frac{\partial \mathcal{L}}{\partial \hat{\mathbf{y}}^{(m)}}$

```
def compute_gradient(y_true, y_pred):
    return -2 * (y_true - y_pred)  # Least squares loss gradient

y_pred = np.zeros(shape=n_samples)

for m in range(n_iterations):
    negative_gradient = -compute_gradient(y_true, y_pred)
    y_pred += alpha * negative_gradient

    # save training loss value for plotting
    loss_values.append(np.mean((y_true - y_pred)**2))
```



Hack # 1

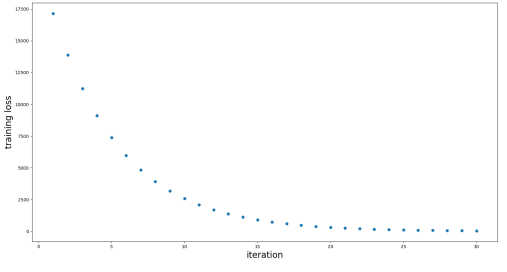
Optimize w.r.t. predictions: $\hat{\mathbf{y}}^{(m+1)} = \hat{\mathbf{y}}^{(m)} - \alpha \frac{\partial \mathcal{L}}{\partial \hat{\mathbf{y}}^{(m)}}$

```
def compute_gradient(y_true, y_pred):
    return -2 * (y_true - y_pred)  # Least squares loss gradient

y_pred = np.zeros(shape=n_samples)

for m in range(n_iterations):
    negative_gradient = -compute_gradient(y_true, y_pred)
    y_pred += alpha * negative_gradient

    # save training loss value for plotting
    loss_values.append(np.mean((y_true - y_pred)**2))
```



Prediction for new data???

Hack # 2

Train weak predictor h to predict the gradients!

```
for m in range(n_iterations):
    negative_gradient = -compute_gradient(y_true, y_pred)
    h = DecisionTree().fit(X, y=negative_gradient)
    y_pred += alpha * h.predict(X)
    self.predictors.append(h)  # save predictor for later
```

We can now use the h_m for new data

```
def predict(X):
    pred = np.zeros(...)
    for h in self.predictors:
        pred += alpha * h.predict(X)
    return pred
```

$$\hat{y}_i = \sum_{m=1}^{\text{n\_iter}} h_m(\mathbf{x}_i)$$

That's our gradient descent!

Recipe for Gradient Boosting

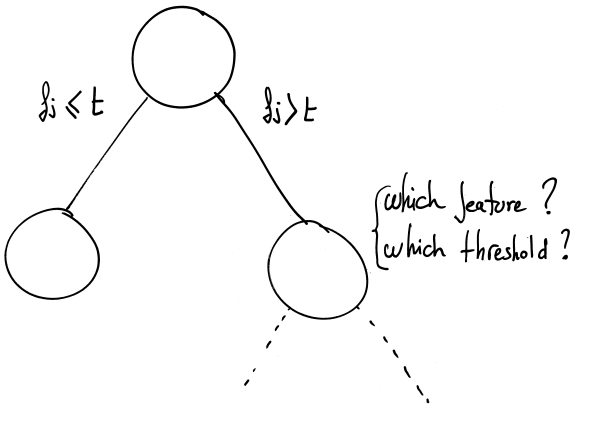
1. Choose a (differentiable) loss
2. Optimize loss w.r.t **predictions** instead of a model parameter
3. Predict the GD step with a base estimator (a regressor)
4. Use these base estimators to predict unseen data

Making it fast

Taking inspiration from LightGBM

What is slow: split finding

```
for m in range(n_iterations):
    ...
    h = DecisionTree().fit(X, y=negative_gradient)
    ...
```



Repeated for each tree, at every node

What is slow: split finding

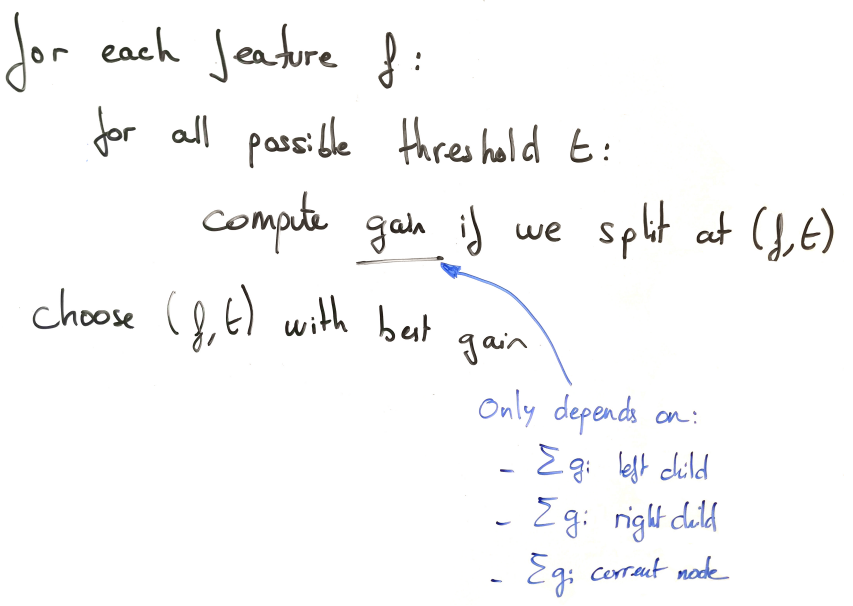
for each feature f:

  for all possible threshold t:

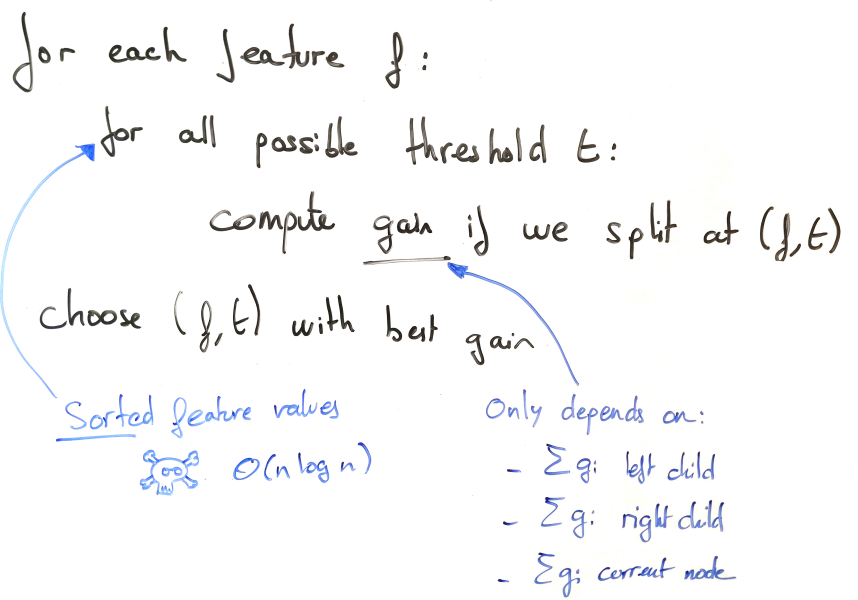
    compute gain if we split at (f, t)

choose (f, t) with best gain

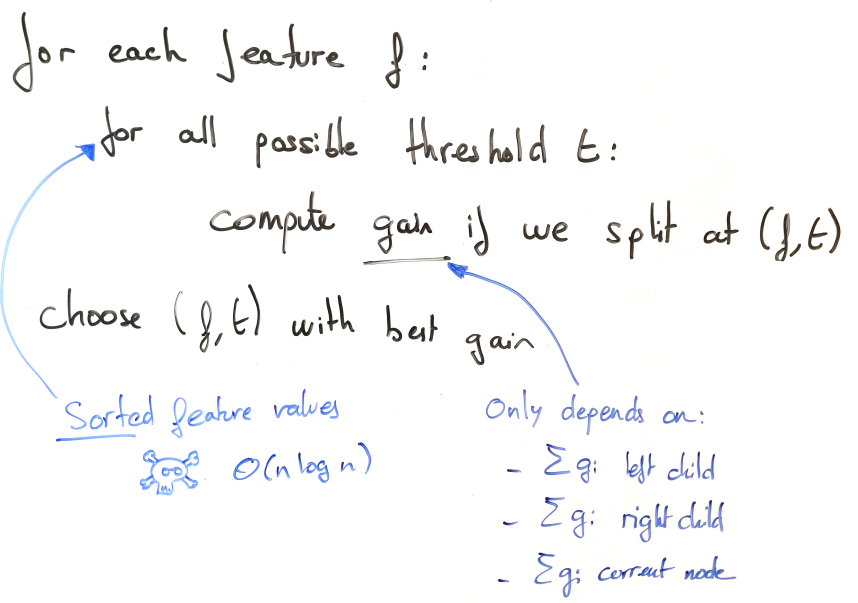
What is slow: split finding



What is slow: split finding

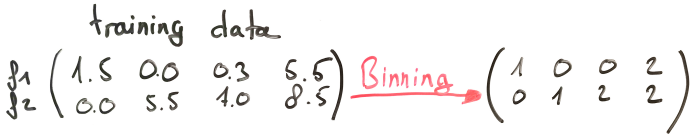


What is slow: split finding



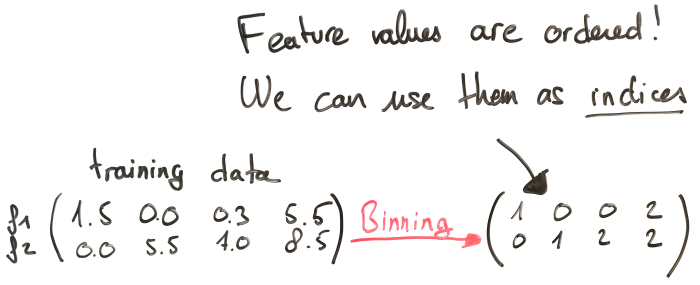
From $\mathcal{O}(n \log n)$ to $\mathcal{O}(n)$: Histograms

Bin the data!



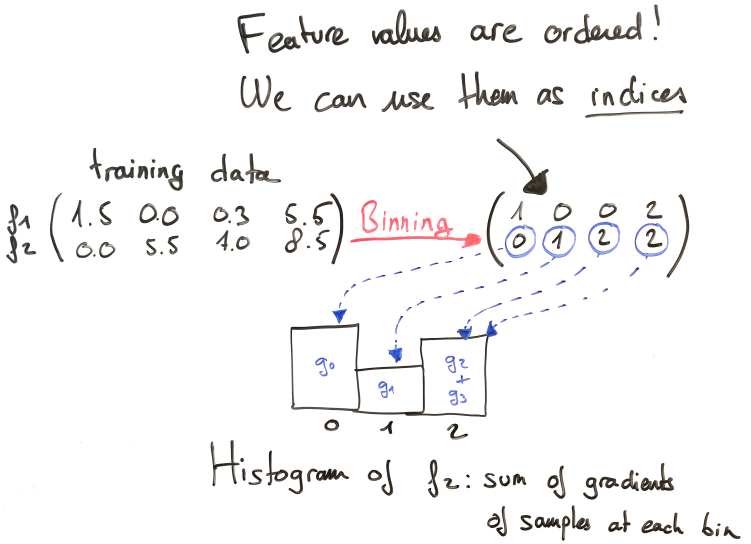
We bin once and for all at the start of the fitting process

Bin the data!



We bin once and for all at the start of the fitting process

Bin the data!



We bin once and for all at the start of the fitting process

Histograms are then computed at each node, for each feature

Split finding now

for each feature f:

    build histogram

    for each bin with threshold t:

        Compute gain

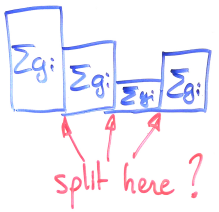
Split finding now

for each feature f:

build histogram

for each bin with threshold t:

Compute gain



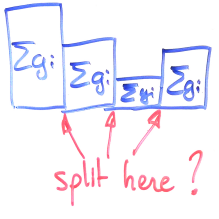
depends on Σg_i in | left / right / current

Split finding now

for each feature f:

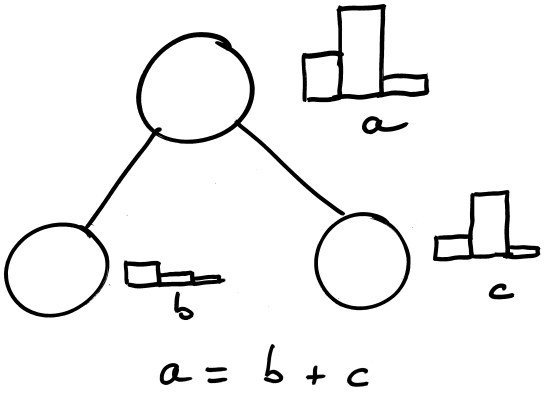
was $\mathcal{O}(n \log n)$

- build histogram $\mathcal{O}(n)$
- for each bin with threshold t: $\mathcal{O}(n\_bins)$
  - Compute gain



depends on Σg_i in | left / right / current

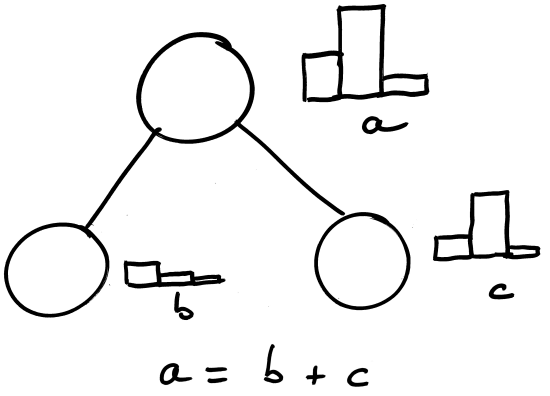
Subtraction trick



Samples are mapped either to left or right child

Hist(parent) = Hist(left) + Hist(right)

Subtraction trick



Samples are mapped either to left or right child

Hist(parent) = Hist(left) + Hist(right)

For half of the nodes:
$\mathcal{O}(n)$ --> $\mathcal{O}$(n_bins)

Numba makes Python code fast

Numba is an open source JIT compiler that translates a subset of Python and NumPy code into fast machine code.

GBDT implementation with `Numba`:

github.com/ogrisel/pygbm

Support for custom dtypes (structured arrays)

```
HISTOGRAM_DTYPE = np.dtype([
    ('g', np.float32),  # sum gradients
    ('h', np.float32),  # sum hessians
])

@njit
def subtract_histograms(hist_a, hist_b):
    out = np.zeros(n_bins, dtype=HISTOGRAM_DTYPE)

    for i in range(n_bins):
        out[i]['g'] = hist_a[i]['g'] - hist_b[i]['g']
        out[i]['h'] = hist_a[i]['h'] - hist_b[i]['h']

    return out
```

Support for custom dtypes (structured arrays)

```
HISTOGRAM_DTYPE = np.dtype([
    ('g', np.float32),  # sum gradients
    ('h', np.float32),  # sum hessians
])

@njit
def subtract_histograms(hist_a, hist_b):
    out = np.zeros(n_bins, dtype=HISTOGRAM_DTYPE)

    for i in range(n_bins):
        out[i]['g'] = hist_a[i]['g'] - hist_b[i]['g']
        out[i]['h'] = hist_a[i]['h'] - hist_b[i]['h']

    return out
```

Support for parallelization (OpenMP or Intel TBB)

```
for feature in prange(n_features):  # parallel loop
    subtract_histograms(...)
```

Support for custom dtypes (structured arrays)

```
HISTOGRAM_DTYPE = np.dtype([
    ('g', np.float32),  # sum gradients
    ('h', np.float32),  # sum hessians
])

@njit
def subtract_histograms(hist_a, hist_b):
    out = np.zeros(n_bins, dtype=HISTOGRAM_DTYPE)

    for i in range(n_bins):
        out[i]['g'] = hist_a[i]['g'] - hist_b[i]['g']
        out[i]['h'] = hist_a[i]['h'] - hist_b[i]['h']

    return out
```

Support for parallelization (OpenMP or Intel TBB)

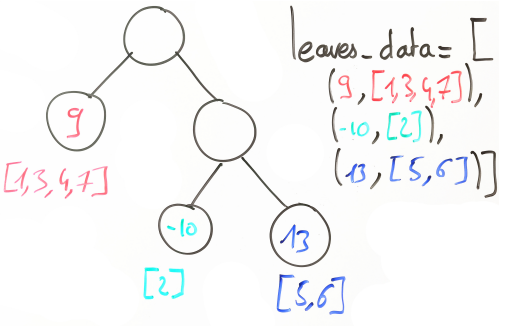
```
for feature in prange(n_features):  # parallel loop
    subtract_histograms(...)
```

`Cython` equivalent:

- need to declare C-struct equivalent to DTYPE
- cannot allocate numpy array in function
  - parallel code needs GIL to be released
  - allocate outside and pass it as a parameter

Lists of different length? No problem

```
for m in range(n_iterations):  # main GB loop
    ...
    y_pred += h.predict(X)  # can be optimized for training data!
    ...
```




```
@njit(parallel=True)
def update_predictions(leaves_data, y_pred):
    for leaf in prange(len(leaves_data)):
        leaf_value, sample_indices = leaves_data[leaf]
        for i in sample_indices:
            y_pred[i] += leaf_value
```

It's fast

Higgs boson dataset (10E7 samples), 20 trees, 4 cores

```
pygbm (ignoring compile time!)
done in 42.851s, ROC AUC: 0.7229, ACC: 0.7245

Lightgbm
done in 42.400s, ROC AUC: 0.7232, ACC: 0.7250
```

Overall experience with Numba

- the doc is good ♥♥♥
- low barrier to entry
- from slow code to fast code: easy process
- code looks like Python (with for loops): stay Pythonic
- it works

Overall experience with Numba

- the doc is good ♥♥♥
- low barrier to entry
- from slow code to fast code: easy process
- code looks like Python (with for loops): stay Pythonic
- it works

Things I wish would improve:

- error messages

Overall experience with Numba

- the doc is good ♥♥♥
- low barrier to entry
- from slow code to fast code: easy process
- code looks like Python (with for loops): stay Pythonic
- it works

Things I wish would improve:

- error messages
- compilation caching (takes 10s for `pygbm`)

Overall experience with Numba

- the doc is good ♥♥♥
- low barrier to entry
- from slow code to fast code: easy process
- code looks like Python (with for loops): stay Pythonic
- it works

Things I wish would improve:

- error messages
- compilation caching (takes 10s for `pygbm`)
- a few bugs still

Overall experience with Numba

- the doc is good ♥♥♥
- low barrier to entry
- from slow code to fast code: easy process
- code looks like Python (with for loops): stay Pythonic
- it works

Things I wish would improve:

- error messages
- compilation caching (takes 10s for `pygbm`)
- a few bugs still
- missing features
  - use parallelization in methods
  - parallel loop over tuples
  - control OpenMP thread scheduling
  - profiling?
  - ...

Overall experience with Numba

- the doc is good ♥♥♥
- low barrier to entry
- from slow code to fast code: easy process
- code looks like Python (with for loops): stay Pythonic
- it works

Things I wish would improve:

- error messages
- compilation caching (takes 10s for `pygbm`)
- a few bugs still
- missing features
  - use parallelization in methods
  - parallel loop over tuples
  - control OpenMP thread scheduling
  - profiling?
  - ...

It's still WIP though

Use it!

Thanks!

(BTW, check out our new GBDTs in scikit-learn!)