Accelerating Random Forests in Scikit-Learn

Gilles Louppe

Université de Liège, Belgium

August 29, 2014
Motivation

... and many more applications!

<table>
<thead>
<tr>
<th>Title</th>
<th>Classification and Regression Trees</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total citations</td>
<td>Cited by 25255</td>
</tr>
<tr>
<td>Citations per year</td>
<td></td>
</tr>
</tbody>
</table>
About

Scikit-Learn

- **Machine learning** library for Python
- Classical and **well-established algorithms**
- Emphasis on **code quality** and **usability**

Myself

- [@glouppe](https://www.glouppe.com)
- PhD student (Liège, Belgium)
- Core developer on Scikit-Learn since 2011
  *Chief tree hugger*
Outline

1 Basics

2 Scikit-Learn implementation

3 Python improvements
Machine Learning 101

• Data comes as...
  ■ A set of **samples** \( \mathcal{L} = \{(x_i, y_i)| i = 0, \ldots, N - 1\} \), with
  ■ **Feature vector** \( x \in \mathbb{R}^p \) (= input), and
  ■ **Response** \( y \in \mathbb{R} \) (regression) or \( y \in \{0, 1\} \) (classification) (= output)

• Goal is to...
  ■ Find a function \( \hat{y} = \varphi(x) \)
  ■ Such that error \( L(y, \hat{y}) \) on new (unseen) \( x \) is minimal
Decision Trees

\[ t \in \varphi : \text{nodes of the tree } \varphi \]
\[ X_t : \text{split variable at } t \]
\[ v_t \in \mathbb{R} : \text{split threshold at } t \]
\[ \varphi(x) = \arg \max_{c \in Y} p(Y = c | X = x) \]
Random Forests

\[ p_{\varphi_1}(Y = c|X = x) \]
\[ p_{\varphi_m}(Y = c|X = x) \]
\[ p_{\psi}(Y = c|X = x) \]

Ensemble of \( M \) randomized decision trees \( \varphi_m \)

\[ \psi(x) = \arg \max_{c \in Y} \frac{1}{M} \sum_{m=1}^{M} p_{\varphi_m}(Y = c|X = x) \]
function BUILDDECISIONTREE(\(\mathcal{L}\))
Create node \(t\)
if the stopping criterion is met for \(t\) then
\(\hat{y}_t =\) some constant value
else
Find the best partition \(\mathcal{L} = \mathcal{L}_L \cup \mathcal{L}_R\)
\(t_L = \text{BUILDDECISIONTREE}(\mathcal{L}_L)\)
\(t_R = \text{BUILDDECISIONTREE}(\mathcal{L}_R)\)
end if
return \(t\)
end function
Outline

1 Basics

2 Scikit-Learn implementation

3 Python improvements
History

Time for building a Random Forest (relative to version 0.10)

0.10: January 2012
- First sketch at sklearn.tree and sklearn.ensemble
- Random Forests and Extremely Randomized Trees modules
History

Time for building a Random Forest (relative to version 0.10)

0.11 : May 2012
- Gradient Boosted Regression Trees module
- Out-of-bag estimates in Random Forests
**Time for building a Random Forest (relative to version 0.10)**

- **1.00** (0.99, 0.98, 0.33, 0.11, 0.04)
  - **0.10**
  - **0.11**
  - **0.12**
  - **0.13**
  - **0.14**
  - **0.15**

- **0.12**: October 2012
  - Multi-output decision trees
History

Time for building a Random Forest (relative to version 0.10)

0.13: February 2013
- Speed improvements
  - Rewriting from Python to Cython
- Support of sample weights
- Totally randomized trees embedding
History

Time for building a Random Forest (relative to version 0.10)

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>0.99</th>
<th>0.98</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.33</td>
<td>0.11</td>
<td>0.04</td>
</tr>
</tbody>
</table>

0.14: August 2013
- Complete rewrite of sklearn.tree
  - Refactoring
  - Cython enhancements
- AdaBoost module
**History**

*Time for building a Random Forest (relative to version 0.10)*

- **0.10**
- **0.11**
- **0.12**
- **0.13**
- **0.14**
- **0.15**

**0.15 : August 2014**

- Further speed and memory improvements
  - Better algorithms
  - Cython enhancements
- Better parallelism
- Bagging module
Implementation overview

- Modular implementation, designed with a strict separation of concerns
  - Builders: for building and connecting nodes into a tree
  - Splitters: for finding a split
  - Criteria: for evaluating the goodness of a split
  - Tree: dedicated data structure

- Efficient algorithmic formulation [See Louppe, 2014]

Tips. An efficient algorithm is better than a bad one, even if the implementation of the latter is strongly optimized.
  - Dedicated sorting procedure
  - Efficient evaluation of consecutive splits

- Close to the metal, carefully coded, implementation
  - 2300+ lines of Python, 3000+ lines of Cython, 1700+ lines of tests

# But we kept it stupid simple for users!
clf = RandomForestClassifier()
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)
Development cycle

Implementation → User feedback

Benchmarks → Profiling

Algorithmic and code improvements → Peer review
Continuous benchmarks

- During code review, changes in the tree codebase are monitored with **benchmarks**.
- **Ensure performance** and code quality.
- **Avoid code complexification** if it is not worth it.

@arjoly does your last bugfix have an impact on the outcome of the regression benchmark?
Outline

1. Basics
2. Scikit-Learn implementation
3. Python improvements
Disclaimer. Early optimization is the root of all evil.

(This took us several years to get it right.)
Use profiling tools for **identifying bottlenecks**.

In [1]: clf = DecisionTreeClassifier()

# Timer
In [2]: %timeit clf.fit(X, y)
1000 loops, best of 3: 394 µs per loop

# memory_profiler
In [3]: %memit clf.fit(X, y)
peak memory: 48.98 MiB, increment: 0.00 MiB

# cProfile
In [4]: %prun clf.fit(X, y)
```text
   ncalls  tottime  percall  cumtime  percall   filename:lineno(function)
     390/32  0.003    0.000   0.004    0.000   _tree.pyx:1257(introsort)
     4719  0.001    0.000   0.001    0.000   _tree.pyx:1229(swap)
         8  0.001    0.000   0.006    0.001   _tree.pyx:1041(node_split)
     405  0.000    0.000   0.000    0.000   _tree.pyx:123(impurity_improvement)
         1  0.000    0.000   0.007    0.007   tree.py:93(fit)
         2  0.000    0.000   0.000    0.000   {method 'argsort' of 'numpy.ndarray'}
     405  0.000    0.000   0.000    0.000   _tree.pyx:294(update)
```
Profiling (cont.)

```python
# line_profiler
In [5]: %lprun -f DecisionTreeClassifier.fit clf.fit(X, y)

Line % Time Line Contents
=================================
... 
256 4.5 self.tree_ = Tree(self.n_features_, self.n_classes_, self.n_outputs_)
257
258 # Use BestFirst if max_leaf_nodes given; use DepthFirst otherwise
259 0.4 if max_leaf_nodes < 0:
260 0.5 builder = DepthFirstTreeBuilder(splitter, min_samples_split,
261 self.min_samples_leaf, max_depth)
262 else:
263 0.6 builder = BestFirstTreeBuilder(splitter, min_samples_split,
264 self.min_samples_leaf, max_leaf_nodes)
265
266 22.4 builder.build(self.tree_, X, y, sample_weight)
... 
```
python -m cProfile -o profile.prof script.py
gprof2dot -f pstats profile.prof -o graph.dot
Python is slow :-(

- Python overhead is **too large for high-performance code**.
- Whenever feasible, **use high-level operations** (e.g., SciPy or NumPy operations on arrays) to **limit Python calls** and rely on highly-optimized code.

```python
def dot_python(a, b):
    # Pure Python (2.09 ms)
    s = 0
    for i in range(a.shape[0]):
        s += a[i] * b[i]
    return s
```

```python
np.dot(a, b)  # NumPy (5.97 us)
```

- Otherwise (and only then!), **write compiled C extensions** (e.g., using Cython) for critical parts.

```python
cpdef dot_mv(double[:,:] a, double[:,:] b):
    # Cython (7.06 us)
    cdef double s = 0
    cdef int i
    for i in range(a.shape[0]):
        s += a[i] * b[i]
    return s
```
Stay close to the metal

- Use the right data type for the right operation.
- Avoid repeated access (if at all) to Python objects.
  - Trees are represented by single arrays.

**Tips.** *In Cython, check for hidden Python overhead. Limit yellow lines as much as possible!*

cython -a _tree.pyx
• Take care of **data locality and contiguity**.
  - Make data contiguous to leverage CPU prefetching and cache mechanisms.
  - Access data in the same way it is stored in memory.

**Tips.** *If accessing values row-wise (resp. column-wise), make sure the array is C-ordered (resp. Fortran-ordered).*

```python
cdef int[::1, :] X = np.asfortranarray(X, dtype=np.int)
cdef int i, j = 42

cdef s = 0
for i in range(...):
    s += X[i, j]  # Fast
    s += X[j, i]  # Slow
```

- If not feasible, use pre-buffering.
Stay close to the metal (cont.)

- Arrays accessed with **bare pointers** remain the fastest solution we have found (sadly).
  - NumPy arrays or MemoryViews are slightly slower
  - Require some **pointer kung-fu**

```python
18: cpdef dot_mv(double[:1] a, double[:1] b):
19:     cdef double s = 0
20:     cdef int i
21:     for i in range(a.shape[0]):
22:         s += a[i] * b[i]

23:     return s
```

```python
26: cpdef dot_ptr(np.ndarray[double, ndim=1] a, np.
27:     cdef double* a_ptr = <double> a.data
28:     cdef double* b_ptr = <double> b.data
29:     cdef double s = 0
30:     cdef int i
31:     for i in range(a.shape[0]):
32:         s += a_ptr[i] * b_ptr[i]
33:     return s
```

# 7.06 us

# 6.35 us
Efficient parallelism in Python is possible!

Andreas Mueller
@t3kcit

Just a quick reminder what sklearn random forests look like on EC2. want? aws.amazon.com/grants/
Scikit-Learn implementation of Random Forests relies on joblib for **building trees in parallel.**

- Multi-processing backend
- Multi-threading backend
  - Require C extensions to be GIL-free
  - **Tips.** *Use nogil declarations whenever possible.*
  - Avoid memory duplication

```python
trees = Parallel(n_jobs=self.n_jobs)(
    delayed(_parallel_build_trees)(
        tree, X, y, ...)
    for i, tree in enumerate(trees))
```
A winning strategy

Scikit-Learn implementation proves to be **one of the fastest** among all libraries and programming languages.
Summary

• The open source development cycle really empowered the Scikit-Learn implementation of Random Forests.

• Combine algorithmic improvements with code optimization.
• Make use of profiling tools to identify bottlenecks.
• Optimize only critical code!