Exploiting CPU SIMD Extensions to Speed-up Document Scoring with Tree Ensembles

Claudio Lucchese
ISTI–CNR, Italy and Istella Srl
c.lucchese@isti.cnr.it

Franco Maria Nardini
ISTI–CNR, Italy and Istella Srl
f.nardini@isti.cnr.it

Salvatore Orlando
U. of Venice, Italy
orlando@unive.it

Raffaele Perego
ISTI–CNR, Italy and Istella Srl
r.perego@isti.cnr.it

Nicola Tonellotto
ISTI–CNR, Italy and Istella Srl
n.tonellotto@isti.cnr.it

Rossano Venturini
U. of Pisa, Italy and Istella Srl
rossano@di.unipi.it

ABSTRACT

Scoring documents with *learning-to-rank* (LtR) models based on large ensembles of regression trees is currently deemed one of the best solutions to effectively rank query results to be returned by large scale Information Retrieval systems.

This paper investigates the opportunities given by SIMD capabilities of modern CPUs to the end of efficiently evaluating regression trees ensembles. We propose V-QuickScorer (vQS), which exploits SIMD extensions to vectorize the document scoring, i.e., to perform the ensemble traversal by evaluating multiple documents simultaneously. We provide a comprehensive evaluation of vQS against the state of the art on three publicly available datasets. Experiments show that vQS provides speed-ups up to a factor of 3.2x.

1. INTRODUCTION

Additive ensembles of regression trees, such as GBRT [3] and λ-MART [6], are nowadays considered among the most advanced LtR models for ranking documents in IR systems, although these require very efficient scoring algorithms for processing queries by strict time budgets [1]. The state-of-the-art algorithm for efficient scoring via additive ensemble of regression trees is QuickScorer (QS) [4]. The main novelty of QS is given by the novel traversal strategy of a tree ensemble $\mathcal{T}$: QS does not traverse $\mathcal{T}$ one tree at a time, but rather evaluates the branching nodes of the trees in a feature-wise order. This strategy was proven to be very efficient for a number of reasons: *(i)* a very small number of nodes is actually visited, *(ii)* the exploited data structures have a cache-friendly access pattern, and *(iii)* fast bitwise operations with few and predictable branch instructions are performed.

In this paper we discuss how QS can be parallelized by exploiting the advanced SIMD capabilities of mainstream CPUs [5]. Streaming SIMD Extensions (SSE) and Advanced Vector Extensions (AVX) are sets of instructions exploiting wide registers of 128 and 256 bits. A single SIMD instruction performs parallel operations on simple data types, e.g., a 128 bit register can manage four single precision or two double precision floats simultaneously. We use in the following a notation similar to the one of Intel Intrinsics[1], where a SIMD instruction codes in its name: (i) a prefix _mm or _mm256 stating if 128 or 256 bits registers are used; (ii) the name of actual operation; (iii) a suffix indicating the types of the operands, e.g., _ps or _pd for 32 and 64 bit floats, respectively. For instance,

$$\overrightarrow{c} = \texttt{\_mm\_cmpgt\_ps}(\overrightarrow{a}, \overrightarrow{b})$$

is a SIMD instruction that works on two registers $\overrightarrow{a}$ and $\overrightarrow{b}$ of 128 bits, each storing a sequence of four single precision floats, and performing four *greater than* comparisons in parallel. The result is stored in a 128-bit register $\overrightarrow{c}$, which will contain four 32-bits sequences of 1s or 0s, depending on the test outcome of the four comparisons. In the following, we will use the notation $\overrightarrow{c} \equiv \langle c_3, c_2, c_1, c_0 \rangle$ to refer to the four elements of a SIMD register.

In this work we propose V-QuickScorer (vQS)[2], a version of QS that exploits CPU vector extensions to boost the efficient traversing of additive ensembles of regression trees. Since exploiting SIMD instructions requires to identify data parallelism opportunities in the code, in QS the most natural source of data parallelism derives from the need of scoring multiple documents for a given query. In particular, we discuss the use of both SSE-4.2 and AVX-2, providing different register widths as well as different SIMD instruction sets, to score up to 8 documents in parallel. Experiments show that vQS provides up to 3.2x speedup compared to the state-of-the-art sequential QS algorithm.

2. QUICKSCORER

Given a query-document pair (q, d_i), represented by a feature vector $\mathbf{x}$, a LtR model based on an additive ensemble of regression trees predicts a relevance score $s(\mathbf{x})$ used for ranking a set of documents. Typically, a tree ensemble encompasses several binary decision trees, denoted by $\mathcal{T} = \{T_0, T_1, \ldots\}$. Each interal (or branching) node $n \in T_h$ is associated with a Boolean test over a specific feature $f_\phi \in \mathcal{F}$, and a constant threshold $\gamma \in \mathbb{R}$. Tests are of the form $\mathbf{x}[\phi] \leq \gamma$, and, during the visit, the left branch is taken

SIGIR '16, July 17 - 21, 2016, Pisa, Italy

ISBN 978-1-4503-4069-4/16/07...$15.00

DOI: http://dx.doi.org/10.1145/2911451.2914758

[1] https://software.intel.com/sites/landingpage/IntrinsicsGuide

[2] Source code: https://github.com/hpclab/vectorized-quickscorer

Algorithm 1: The QUICKSCORER Algorithm

```
   QuickScorer(x,T):
1  |  foreach T_h ∈ T do
2  |  |  leafindexes[h] ← 11...11
3  |  foreach f_φ ∈ F do                    // Mask Computation Step
4  |  |  foreach (γ,h,n) ∈ N_φ in ascending order do
5  |  |  |  if x[φ] > γ then
6  |  |  |  |  leafindexes[h] ← leafindexes[h] ∧ nodemasks[n]
   |  |  |  else
   |  |  |  |  break
7  |  score ← 0
8  |  foreach T_h ∈ T do                    // Score Computation Step
9  |  |  j ← index of leftmost bit set to 1 of leafindexes[h]
10 |  |  l ← h · Λ + j
11 |  |  score ← score + leafvalues[l]
12 |  return score
```

iff the test succeeds. Each leaf node stores the tree prediction, representing the potential contribution of the tree to the final document score. The scoring of $\mathbf{x}$ requires the traversal of all the ensemble's trees and it is computed as a *weighted sum* of tree predictions.

Algorithm 1 illustrates QS [4]. One important result of QS is that to compute $s(\mathbf{x})$ it needs to identify only the branching nodes whose tests evaluate to false, called *false nodes*. To do so, QS maintains for each tree $T_h \in \mathcal{T}$ a bitvector `leafindexes[`h`]`, made of Λ bits, one per leaf. Initially, every bit in `leafindexes[`h`]` is set to 1. Moreover, each branching node n is associated with a binary mask `nodemasks[`n`]` idendifying the set of unreachable leaves in case the corresponding test evaluates to false. Whenever a false node is visited, the set of unreachable leaves `leafindexes[`h`]` is updated through a *logical and* with `nodemasks[`n`]`. Eventually, the leftmost bit set in `leafindexes[`h`]` identifies the leaf corresponding to the score contribution of T_h, stored in the lookup table `leafvalues`.

To efficiently identify all the *false nodes* in the ensemble, QS processes the branching nodes of all the trees *feature by feature* and in ascending order of their predicate thresholds. Specifically, for each feature f_ϕ, QS builds a list $\mathcal{N}_\phi$ of tuples (γ, h, n), where γ is the predicate threshold of node n occurring in tree T_h. When processing $\mathcal{N}_\phi$ in ascending order by γ, as soon as a test evaluates to true, i.e., $\mathbf{x}[\phi] \leq \gamma$, the remaining occurrences surely evaluate to true as well, and their evaluation is thus safely skipped.

We call *mask computation* the first step of the algorithm during which all the bitvectors `leafindexes[`h`]` are updated, and *score computation* the second step where such bitvectors are used to retrieve tree predictions.

3. VECTORIZED QUICKSCORER

SIMD extension of modern CPUs provide powerful fine-grained parallelism which can be exploited to score multiple documents simultaneously. Note that the QS paper [4] already investigated multiple documents scoring as a strategy to improve cache performance. In this work, we propose V-QUICKSCORER (vQS), a SIMD-based algorithm exploiting the natural *data parallelism* deriving from this strategy.

Both the *mask computation* and *score computation* steps of QS can be engineered to take advantage of SIMD registers. During the first step, multiple documents can be tested against a given node predicate and their `leafindexes[`h`]` updated in parallel. Similarly, the score of multiple documents can be computed simultaneously during the second step. The data structure `leafindexes` used to encode the exit leaves must be replicated to accomodate the documents scored simultaneously.

The specific optimizations used by vQS depend on the SIMD register width available and on the maximum number of leaves Λ in the ensemble. We first discuss how vQS exploits SSE-4.2 or AVX-2 instructions when $\Lambda = 32$, then we highlight the differences when $\Lambda = 64$.

Algorithm 2: The vQS Algorithm (SSE-4.2, $\Lambda = 32$)

```
   V-QuickScorer( {x_i}_{i=0,1,2,3}, T, scores_{3:0}):
1  |  foreach T_h ∈ T do
2  |  |  leafindexes[h] ← 11...11
3  |  foreach f_φ ∈ F do                    // Mask Computation Step
4  |  |  foreach (γ,h,n) ∈ N_φ in ascending order do
5  |  |  |  γ⃗ ← _mm_set1_ps(γ)
6  |  |  |  x⃗ ← _mm_set_ps(x_3[φ], x_2[φ], x_1[φ], x_0[φ])
7  |  |  |  c⃗ ← _mm_cmpgt_ps(x⃗, γ⃗)
8  |  |  |  if ¬(_mm_test_all_zeros(c⃗, c⃗)) then
9  |  |  |  |  b⃗ ← _mm_load_ps(leafindexes_{3:0}[h])
10 |  |  |  |  m⃗ ← _mm_set1_ps(nodemask[n])
11 |  |  |  |  y⃗ ← _mm_andnot_ps(m⃗, c⃗)
12 |  |  |  |  y⃗ ← _mm_andnot_ps(y⃗, b⃗)
13 |  |  |  |  _mm_store_ps(leafindex_{3:0}[h], y⃗)
   |  |  |  else
14 |  |  |  |  break
15 |  s⃗_{1:0} ← _mm_set1_pd(0)              // Score Computation Step
16 |  s⃗_{3:2} ← _mm_set1_pd(0)
17 |  foreach T_h ∈ T do
18 |  |  ∀i = 3:0 : j_i ← index leftmost 1 bit of leafindex_i[h]
19 |  |  ∀i = 3:0 : l_i ← h · Λ + j_i
20 |  |  v⃗_{1:0} ← _mm_set_pd(leafvalues[l_1], leafvalues[l_0])
21 |  |  v⃗_{3:2} ← _mm_set_pd(leafvalues[l_3], leafvalues[l_2])
22 |  |  s⃗_{1:0} ← _mm_add_pd(s⃗_{1:0}, v⃗_{1:0})
23 |  |  s⃗_{3:2} ← _mm_add_pd(s⃗_{3:2}, v⃗_{3:2})
24 |  _mm_store_pd(s⃗_{1:0}, scores_{1:0})
25 |  _mm_store_pd(s⃗_{3:2}, scores_{3:2})
```

V-QuickScorer with Λ=32 and SSE-4.2

SSE-4.2 registers are 128 bits wide, and permit processing 4 documents simultaneously during the *mask computation* step. Therefore, as shown in Alg. 2, vQS is given in input a set of four instances $\{\mathbf{x}_i\}_{i=0,1,2,3}$, and a vector $\texttt{scores}_{3:0}$ of four double precision floats where the scores are stored upon completion.

During the *mask computation* step, one 128-bit SIMD register $\overrightarrow{\gamma}$ is used to store 4 copies of the same test threshold γ, and another register $\overrightarrow{x}$ to store the feature $\mathbf{x}_i[\phi]$, $i = 0, ..., 3$ of the 4 input instances (lines 5-6). A single instruction is used to compare the feature values of these four documents against γ (line 7). If all tests evaluate to true, i.e., we do not have any *false node*, then the next feature is processed, otherwise `leafindexes` is updated. Note that unlike the QS sequential implementation, the need of verifying the true condition for all 4 documents rather than for a single one, may lead to some overheads in the strategy aimed at identifying false nodes only in the tree ensemble.

The update of `leafindexes` involves potentially four documents and should occur only for those where $\mathbf{x}_i[\phi] > \gamma$. Since SSE-4.2 does not support masked/predicated SIMD instructions, to avoid conditional branches vQS implements the update with two bitwise operations. Let $\texttt{leafindexes}_i[h]$ be a 32-bit vector ($\Lambda = 32$), relative to tree T_h and associated with document $\mathbf{x}_i$. Let variable c_i store a string of 32 bits of 1s or 0s, depending on the outcome of the test $\mathbf{x}_i[\phi] > \gamma$. We can rewrite the update as:

$$\texttt{leafindexes}_i[h] \leftarrow (\texttt{nodemask}[n] \vee \neg c_i) \wedge \texttt{leafindexes}_i[h]$$

where the *bitwise logical or* has the effect of leaving $\texttt{nodemask}[n]$ unaltered when $\mathbf{x}_i[\phi] > \gamma$, or making it useless otherwise. We can re-write the expression as follows by applying the De Morgan law:

$$\texttt{leafindexes}_i[h] \leftarrow \neg\,(\neg\,\texttt{nodemask}[n] \wedge c_i) \wedge \texttt{leafindexes}_i[h]$$

which we can straightforwardly implement by a repeated application of the SIMD function $\texttt{andnot}(x, y) = \neg x \wedge y$.

The layout of `leafindexes` is tree-wise, i.e., given a tree T_h the bitvectors $\texttt{leafindexes}_i[h]$ of the four $\mathbf{x}_i$ are stored contiguously in memory. As shown in Alg. 2 (lines 9–13), this allows loading the four bitvectors with a single 128-bit load instruction, and to apply them the two SIMD andnot instructions. Indeed, first the four $\texttt{leafindexes}_{3:0}[h]$ are loaded into the register $\overrightarrow{b}$ and $\texttt{nodemask}[n]$ is replicated into $\overrightarrow{m}$. After composing $\overrightarrow{m}$, $\overrightarrow{c}$ and $\overrightarrow{b}$, the resulting mask is finally copied back to memory.

The *score computation* is also parallelized (see Alg. 2, from line 15). To provide the required precision, tree predictions are stored as double precision float values (64 bits), which means that only 2 document scores can be processed simultaneously using 128-bit registers. Thus, vQS uses two (pd) SIMD variables, namely $\overrightarrow{s_{1:0}}$ and $\overrightarrow{s_{3:2}}$, to maintain the score of our 4 documents. For each tree, the predicted partial scores related to the 4 input instances are similarly stored in $\overrightarrow{v_{1:0}}$ and $\overrightarrow{v_{3:2}}$, and added up to update the final document scores. Eventually, the computed scores for the 4 documents are copied to vector $\texttt{scores}_{3:0}$.

V-QuickScorer with Λ=32 and AVX-2

When AVX-2 is supported, it is possible to increase the parallelism degree of vQS. Trivially, 8 document features tests can be performed simultaneously instead of 4, and 4 document scores updated instead of 2. In this case, vQS scores 8 documents at each invocation. We do not discuss the pseudocode for the document feature testing and document scores calculation, as it simply requires to adopt the 256-bit versions of the respective instructions illustrated above.

More interestingly, AVX-2 also provides additional instructions, such as _mm256_maskstore_ps: this copies a 256-bit register to memory according to a given mask enabling/disabling sub-groups of 32-bits. This makes it possible to *conditionally* update each of the 8 elements of $\texttt{leafindexes}_{7:0}$ (or to leave it unchanged) depending on the output of the 8 node predicates, which is stored in $\overrightarrow{c}$. Lines 11–13 of Alg. 2 are replaced as follows, where the vector variables involved are now 256 bit registers:

$$\overrightarrow{y} \leftarrow \texttt{\_mm256\_and\_ps}(\overrightarrow{m}, \overrightarrow{b})$$
$$\texttt{\_mm256\_maskstore\_ps}(\texttt{leafindexs}_{7:0}[h], \overrightarrow{c}, \overrightarrow{y})$$

V-QuickScorer with Λ=64

Increasing Λ (the maximum number of tree leaves) impacts on the size of the bitwise structures `leafindexes` and `nodemask`, as each bitvector they store is Λ bits wide. As a consequence, the number of bitvectors (either `leafindexes` or `nodemask`) that can be processed simultaneously in a SIMD register decreases. Recall that, when $\Lambda = 32$, the number of tests and the number of bitvectors fitting in a SIMD register are the same, either 4 or 8 for SSE-4.2 or AVX-2, respectively. Therefore, the variables $\overrightarrow{m}$, $\overrightarrow{c}$ and $\overrightarrow{b}$ are seamlessly processed together. When $\Lambda = 64$, there is a mismatch between the 32 bits returned by each predicate test, and the 64 bits of the `leafindexes` and `nodemask` bitvectors.

For 128-bits registers, let $\overrightarrow{c} \equiv \langle c_3, c_2, c_1, c_0\rangle$ be the outcome of the four comparisons against a threshold γ. For the subsequent update of the 64-bits masks, vQS requires to process $\overrightarrow{c}$ in order to obtain two variables, storing only two comparison outcomes of 64 bits each. To this end, we use the following two instructions, working on the low and high half of $\overrightarrow{c}$, respectively.

$\langle c_1, c_1, c_0, c_0\rangle \equiv$ _mm_unpacklo_ps($\overrightarrow{c}, \overrightarrow{c}$)
$\langle c_3, c_3, c_2, c_2\rangle \equiv$ _mm_unpacklhi_ps($\overrightarrow{c}, \overrightarrow{c}$)

Once prepared these two result variables, they are used in the subsequent andnot operations, similarly to Alg. 2. The only difference is that the code from line 11 to 13 must be repeated twice, one for updating the two copies of leafindexes associated with the first pair of documents, and the other for the second pair.

By using 256-bits registers on AVX-2, vQS performs 8 tests in parallel, while we update the 8 copies of leafindexes by exploiting two blocks of SIMD instruction, each performing 4 operations in parallel on the 64-bit bitwise data structures. As before, from $\overrightarrow{c} \equiv \langle c_7, c_6, \ldots, c_0\rangle$, we would like to extract two vectors with the following layout $\langle c_3, c_3, c_2, c_2, c_1, c_1, c_0, c_0\rangle$ and $\langle c_7, c_7, c_6, c_6, c_5, c_5, c_4, c_4\rangle$. Unfortunately, the AVX-2 instructions for unpacking these bitmasks, namely _mm256_unpacklo_ps and _mm256_unpackhi_ps, work by considering each 256-bits registers as two 128-bit lanes, and thus pick the least/most significant 64 bits from each of these lanes. The solution adopted by vQS is to set a different layout for $\overrightarrow{c}$ in order to be able to apply the above unpacking instructions. To this end, we load the 8 features of $\overrightarrow{x}$, to be compared with the threshold γ, in the following suitable order:

$$\overrightarrow{x} \leftarrow \texttt{\_mm256\_set\_ps}(\ \mathbf{x}_7[\phi], \mathbf{x}_6[\phi], \mathbf{x}_3[\phi], \mathbf{x}_2[\phi], \mathbf{x}_5[\phi], \mathbf{x}_4[\phi], \mathbf{x}_1[\phi], \mathbf{x}_0[\phi]\ )$$

This new 256 bit instruction replaces line 6 of Alg. 2.

4. EXPERIMENTS

Datasets. We used three publicly available datasets: Microsoft LETOR (MSN-1[3]) and Yahoo LETOR (Y!S1[4]), commonly used in the scientific community for LtR experiments, and a new larger one called Istella LETOR (istella[5]). istella is composed of 33,018 queries and 10,454,629 query-document pairs, where each pair is represented by a vector of 220 features. This is split in training and test sets with a 70%-30%

[3] http://research.microsoft.com/en-us/projects/mslr/
[4] http://learningtorankchallenge.yahoo.com
[5] http://blog.istella.it/istella-learning-to-rank-dataset/

Table 1: Per-document scoring time in μs of QS, vQS (SSE 4.2), vQS (AVX-2) on MSN-1, Y!S1, and istella datasets. Speedups over the baseline QS are reported in parentheses.

Λ	Method	Number of trees/dataset					
		1,000			10,000		
		MSN-1	Y!S1	istella	MSN-1	Y!S1	istella
32	QS	6.3 (–)	12.5 (–)	8.9 (–)	73.7 (–)	88.7 (–)	69.9 (–)
	vQS (SSE 4.2)	3.2 (2.0x)	5.2 (2.4x)	4.2 (2.1x)	46.2 (1.6x)	53.7 (1.7x)	38.6 (1.8x)
	vQS (AVX-2)	**2.6 (2.4x)**	**3.9 (3.2x)**	**3.1 (2.9x)**	**39.6 (1.9x)**	**43.7 (2.0x)**	**30.7 (2.3x)**
64	QS	11.9 (-)	18.8 (-)	14.3 (-)	183.7 (-)	182.7 (-)	162.2 (-)
	vQS (SSE 4.2)	10.2 (1.2x)	13.9 (1.4x)	11.0 (1.3x)	173.1 (1.1x)	164.3 (1.1x)	132.2 (1.2x)
	vQS (AVX-2)	**7.9 (1.5x)**	**10.5 (1.8x)**	**8.0 (1.8x)**	**138.2 (1.3x)**	**140.0 (1.3x)**	**104.2 (1.6x)**

partitioning. To the best of our knowledge, this is the largest publicly available LtR dataset, particularly useful for large-scale experiments on the efficiency and scalability of LtR solutions. In all the three datasets, feature vectors are labeled with judgments ranging from 0 (irrelevant) to 4 (perfectly relevant).

Experimental methodology. We trained λ-MART [6] models optimizing NDCG@10 on the three datasets, and generated models with $\Lambda = 32$ or $\Lambda = 64$ leaves and with $|\mathcal{T}|$=1,000 or $|\mathcal{T}|$=10,000 trees. We used the open source implementation of λ-MART by [2], however it is worth noting that the results reported in this work are independent of the training algorithm implementation. To provide a fair comparison, vQS was implemented by engineering the source code of QS. In the following we reported the average per-document scoring time averaged over 10 runs. The tests were performed on a machine equipped with an Intel Xeon CPU E5-2630 v3 clocked at 2.40GHz with 20 MB of cache L3 and 64GB RAM.

Efficiency evaluation. Table 1 reports the average time (in μs) for scoring a single document across the three datasets, when varying both the number of trees and leaves in the ensemble. The best improvements are achived with $\Lambda = 32$, as vQS can use either 4- or 8-way parallelism for both feature predicate testing and bitvectors updating. When using AVX-2, speed-ups range from 1.9x (for MSN-1 with 10,000 trees) to 3.2x (for Y!S1 with 1,000 trees). These are greatly reduced woth SSE 4.2, with a maximum speedup of 2.4x for the 1,000 trees model over Y!S1. As expected, performance worsen with $\Lambda = 64$, with a maximum speed-up of 1.8x. The lower improvement is due to inefficiencies deriving from additional processing required to align the 4-/8-way comparisons to the 2-/4-way conditional mask updates.

A final note regards the overheads of the vectorized code during the scan of the ordered list of feature thresholds $\mathcal{N}_\phi$. While QS stops as soon as the single document feature is greater of the current threshold, vQS must continue as long as at least one among the 4 or 8 documents evaluated simultaneously does not match the exit criterion. We instrumented the code to measure this difference. The tests conducted on MSN-1, with 10K trees and $\Lambda = 64$, confirmed the hypothesis: to score a single document QS executes in average 15.76 tests per tree, while this number increases to 22.80 and 26.68 for the SSE 4.2 and AVX-2 versions of vQS, respectively. In fact, we observed that while the *score computation* step benefits significantly of the increased parallelism provided by AVX-2, the *mask computation* step exhibits only a limited improvement, due to the additional comparison costs mentioned above.

5. CONCLUSION

We discussed in depth the vectorization of QS, the state-of-the-art algorithm for scoring documents with LtR tree ensembles. Using SIMD capabilities of mainstream CPUs, namely SSE 4.2 and AVX 2, vQS can process up to 8 documents in parallel, although there is a tradeoff due to the possible increase in the number of operations carried out. We also highlighted some features of these SIMD coprocessors, which force to re-design algorithms in non trivial ways.

The upcoming AVX-512 extension, due to wider registers, would allow to further increase the parallelism degree up to 16 documents. Wider registers are not the only benefit, since many new instructions will be available. One example is _mm512_lzcnt_epi32, which counts the number of leading zeros, i.e., the index of the first bit set to 1, in each of the 16 sub-groups of 32 bits. This would allow to parallelize the code at lines 18-19 of Alg. 2, where the indexes of 16 exit leaves in `leafvalues` would be computed simultaneously. Moreover, masked/predicated instructions would allow to more easily pipeline comparision, update and store operations.

Acknowledgements. This work was partially supported by the EC H2020 Program INFRAIA-1-2014-2015 SoBigData: Social Mining & Big Data Ecosystem (654024).

6. REFERENCES

[1] N. Asadi, J. Lin, and A. P. de Vries. Runtime optimizations for tree-based machine learning models. *IEEE Transactions on Knowledge and Data Engineering*, 26(9):2281–2292, 2014.

[2] G. Capannini, C. Lucchese, F. M. Nardini, S. Orlando, R. Perego, and N. Tonellotto. Quality versus Efficiency in Document Scoring with Learning-to-Rank Models. *Information Processing and Management*, 2016.

[3] J. H. Friedman. Greedy function approximation: a gradient boosting machine. *Annals of Statistics*, 2001.

[4] C. Lucchese, F. M. Nardini, S. Orlando, R. Perego, N. Tonellotto, and R. Venturini. Quickscorer: A fast algorithm to rank documents with additive ensembles of regression trees. In *Proc. of the 38th ACM SIGIR Conference*, pages 73–82, 2015.

[5] O. Polychroniou, A. Raghavan, and K. A. Ross. Rethinking simd vectorization for in-memory databases. In *Proc. of the 2015 ACM SIGMOD Conference*, pages 1493–1508, New York, NY, USA, 2015.

[6] Q. Wu, C. J. Burges, K. M. Svore, and J. Gao. Adapting boosting for information retrieval measures. *Information Retrieval*, 2010.