Machine Learning for Embedded Systems: A Case Study

Karen Zita Haigh, Allan M. Mackay, Michael R. Cook, Li G. Lin

Raytheon BBN Technologies
10 Moulton Street, Cambridge, MA 02138
khaigh@bbn.com

Abstract—We describe our application’s need for Machine Learning on a General Purpose Processor of an embedded device. Existing ML toolkits tend to be slow and consume memory, making them incompatible with real-time systems, limited hardware resources, or the rapid timing requirements of most embedded systems. We present our ML application, and the suite of optimizations we performed to create a system that can operate effectively on an embedded platform. We perform an ablation study to analyze the impact of each optimization, and demonstrate over 20x improvement in runtimes over the original implementation, over a suite of 19 benchmark datasets. We present our results on two embedded systems.

I. INTRODUCTION

Mobile ad hoc networks (MANETs) operate in highly dynamic, potentially hostile environments. Current approaches to network configuration tend to be static, and therefore perform poorly. It is instead desirable to adaptively configure the radio and network stack to maintain consistent communications. A human is unable to perform this dynamic configuration partly because of the rapid timescales involved, and partly because there are an exponential number of configurations [7].

Machine Learning is a suite of techniques that learn from their experience, by analyzing their observations, updating models of how previous actions performed, and using those insights to make better decisions in the future. The system can then learn how current conditions affect communications quality, and automatically select a configuration to improve performance, even in highly-dynamic missions. The domain requires the ability for the decision maker to select a configuration in real-time, within the decision-making loop of the radio and IP stack.

This paper presents our effort to place Support Vector Machines (SVMs) [21], [22] onto the general purpose processors of two communications networks. Existing SVM libraries are slow and memory intensive. This paper describes how we optimized an existing SVM library to obtain a 20x runtime improvement and controlled the memory footprint of the system. This paper describes the optimizations that either had the most effect on results, or were the most surprising to us as developers.

II. EMBEDDED COMMUNICATIONS DOMAIN

Our target domain is a communications controller that automatically learns the relationships among configuration parameters of a mobile ad hoc network (MANET) to maintain near-optimal configurations automatically in highly dynamic environments. Consider a MANET with $N$ nodes; each node has a set of observable parameters $o$ that describe the environment, a set of controllable parameters $c$ that it can use to change its behavior, and a metric $m$ that provide feedback on how well it is doing. Each control parameter has a known set of discrete values. If all $n$ controllables are binary on/off, then there are $2^n$ strategies, well beyond the ability of a human to manage. The goal is to have each node choose a combination of controllables $c$, to maximize performance of the metric $m$, by learning a model $f$ that predicts performance of the metric from the observables $o$ and controllables $c: m = f(o, c)$. The mathematics of this domain is described in more detail elsewhere [8], [9].

Target Platforms: Our target platforms are two existing embedded systems for communications, each with pre-established hardware and runtime environments. These are legacy systems on which we are deploying upgraded software capabilities. Both platforms have general-purpose CPUs with no specialized hardware acceleration. We consider this an embedded system because it is dedicated to a specific set of capabilities, has limited hardware resources, limited operating system capabilities, and requires an external device to build and download its runtime software. Our embedded platforms are:

**ARMv7:** ARMv7 rev 2 (v7l) at 800 MHz, 256 kbyte cache, 256MB RAM, vintage 2005. Linux 2.6.38.8, G++ 4.3.3, 2009.

**PPC440:** IBM PPC440GX [1] at 533MHz, 256 kbyte cache, 128MB RAM, vintage 1999. Green Hills Integrity RTOS, version 5.0.6. We use the Green Hills Multi compiler, version 4.2.3, which (usually) follows the ISO 14882:1998 standard for C++.

For comparison, we also show timing results on a modern Linux server:

**Linux:** 16 processor Intel Xeon CPU E5-2665 at 2.40GHz, 20480 kbyte cache, vintage 2012. Ubuntu Linux, version 3.5.0-54-generic. g++ Ubuntu/Linaro 4.6.3-1ubuntu5, running with -std=c++98.

Operating Environment: At runtime, the learner builds a model from available training data, which is presented as a set of vectors of observables and controllables, each with an associated performance metric. To make control decisions, the system receives a vector of observables describing the current environment, uses the model to estimate expected performance for each combination of controllables.

The operating system controls available CPU, shared between the learner and the communications IP stack. The learner
operates asynchronously, returning a decision when finished. Notably, the PPC440 platform’s real-time operating system explicitly allows us to directly control how much CPU the learner can use. The speed of the decision maker therefore directly affects which controllable parameters to capture in the learned model: any controllables that must be chosen more rapidly than the learner can handle are not candidates.

**Development Team:** Our development team had one Machine Learning expert, one hard-real time embedded expert, and one C++ algorithms expert. We also received code reviews and consulting from the individuals most familiar with the platforms (hardware and software).

### III. MACHINE LEARNING AND REGRESSION

Support Vector Machines [21], [22] are ideally suited to learning this regression function from attributes to performance. The regression function is commonly written as:

\[ m = f(x) = < w, x > + b \]

where \( x \) are the attributes (combined \( o \) and \( c \)), where \( w \) is a set of weights (similar to a slope) and \( b \) is the offset of the regression function.

When the original problem is not linear, we transform the feature space into a high-dimensional space that allows us to solve the new problem linearly. In general, the non-linear mapping is unknown beforehand, and therefore we perform the transformation using a *kernel*, \( \phi(x_i, x) \), where \( x_i \) is an instance in the training data that was selected as a support vector, \( x \) is the instance we are trying to predict, and where \( \phi \) is a vector representing the actual non-linear mapping. In this work, we use the *Pearson Universal Kernel* [21] because it has been shown to work well in a wide variety of domains, and was consistently the most accurate in our target communications domain:

\[
\phi(x_i, x) = \frac{1}{1 + \left( \frac{2}{\sigma} \sqrt{||x_i - x||^2 \left(2(1/\omega) - 1\right)} \right)^\omega} \tag{1}
\]

\( \omega \) describes the shape of the curve; as \( \omega = 1 \), it resembles a Lorentzian, and as it approaches infinity, it becomes equal to a Gaussian peak. \( \sigma \) controls the half-width at half-maxima.

The regression function then becomes:

\[ m = f(x) = \sum_{i=1}^{n} (\alpha_i - \alpha_i^*) \phi(x_i, x) + b \tag{2} \]

where \( n \) is the number of training instances that become support vectors, \( \alpha_i \) and \( \alpha_i^* \) are Lagrange multipliers computed when the model is built (see Üstün et al. [21]).

There are many available implementations of SVMs, e.g., [12], [24]. We considered language (preference for C++), licenses, memory usage, and compilation effort. Table 1 lists some of the options that are available in C++ with licenses appropriate for our work. We did not consider Java implementations because of its higher memory requirements and because Java is not compatible with other software components on our target platform. We also eliminated several packages that rely heavily on malloc() calls, as dynamic memory usage is both slow and likely to cause runtime errors on our RAM-limited hardware. None of the remaining libraries directly compiled on our embedded target platforms, so we conducted the initial tests on a modern Linux server. We only tested those packages that compiled within approximately an hour after download: if the software wouldn’t compile easily on a modern platform, it would be extremely painful to migrate it to the older platforms.

To select the specific package from which to continue development, we ran the suite of benchmark datasets listed in Table 2. Table 3 shows the timing results; Weka’s SMOReg is the fastest in all but a few cases. LibSVM is 2.0x slower than Weka on average.† Dlib is 10.4x slower than Weka on average; moreover DLib gets worse the bigger the dataset (e.g. 4.1x when fewer than 1000 instances, and 15.6x when more than 1000 instances).

Given these results, we decided to rely on the SVM implementation found in Weka [10], with Üstün’s Pearson VII Universal Kernel (Puk) [21] and Platt’s Sequential Minimal Optimization algorithm [17] to compute the maximum-margin hyperplanes.

### IV. OPTIMIZATIONS

Our first working C++ implementation was based on SMOReg in WekaC++ [18], with a translation of WekaJava [6] items that were not already in the C++ version. We refer to this version as *Baseline*.

This paper describes the optimizations that either had the most effect on our results, or were the most surprising to us as developers. These include numerical representations (double vs float vs integer) algorithmic constructs (kernel, memory vs compute), data structures (vectors), and compiler tricks (flattening object structure, inlining, exceptions). We also

<table>
<thead>
<tr>
<th>Test Name</th>
<th>Description</th>
<th>Instances</th>
<th>Attributes</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>apple</td>
<td>Airfoil noise</td>
<td>1503</td>
<td>6</td>
<td>UCI [2]</td>
</tr>
<tr>
<td>AutoPrice</td>
<td>Automobile prices</td>
<td>159</td>
<td>16</td>
<td>Weka [10]</td>
</tr>
<tr>
<td>bodyfat</td>
<td>Percentage bodyfat</td>
<td>252</td>
<td>15</td>
<td>Weka [10]</td>
</tr>
<tr>
<td>concrete</td>
<td>Concrete compressive strength</td>
<td>1030</td>
<td>9</td>
<td>UCI [25],[2]</td>
</tr>
<tr>
<td>cpu</td>
<td>CPU relative performance</td>
<td>227</td>
<td>7</td>
<td>Weka [10]</td>
</tr>
<tr>
<td>fishcatch</td>
<td>Fish weight</td>
<td>159</td>
<td>8</td>
<td>Weka [10]</td>
</tr>
<tr>
<td>housing</td>
<td>Boston housing values</td>
<td>506</td>
<td>14</td>
<td>Weka [10]</td>
</tr>
<tr>
<td>pole</td>
<td>Telecommunications</td>
<td>14998</td>
<td>26</td>
<td>LIACC [23]</td>
</tr>
<tr>
<td>wine red</td>
<td>Wine quality, red</td>
<td>1599</td>
<td>12</td>
<td>UCI [4],[2]</td>
</tr>
<tr>
<td>wine white</td>
<td>Wine quality, white</td>
<td>4898</td>
<td>12</td>
<td>UCI [4],[2]</td>
</tr>
<tr>
<td>comm-A-TP</td>
<td>Communications throughput</td>
<td>1078</td>
<td>59</td>
<td>Custom</td>
</tr>
<tr>
<td>comm-A-Lat</td>
<td>Communications latency</td>
<td>1078</td>
<td>59</td>
<td>Custom</td>
</tr>
<tr>
<td>comm-B-TP</td>
<td>Communications throughput</td>
<td>820</td>
<td>59</td>
<td>Custom</td>
</tr>
<tr>
<td>comm-B-Lat</td>
<td>Communications latency</td>
<td>820</td>
<td>59</td>
<td>Custom</td>
</tr>
<tr>
<td>comm-C-TP</td>
<td>Communications throughput</td>
<td>378</td>
<td>59</td>
<td>Custom</td>
</tr>
<tr>
<td>comm-C-Lat</td>
<td>Communications latency</td>
<td>378</td>
<td>59</td>
<td>Custom</td>
</tr>
<tr>
<td>comm-D-TP</td>
<td>Communications throughput</td>
<td>1732</td>
<td>44</td>
<td>Custom</td>
</tr>
<tr>
<td>comm-D-Lat</td>
<td>Communications latency</td>
<td>1732</td>
<td>44</td>
<td>Custom</td>
</tr>
<tr>
<td>comm-D-BER</td>
<td>Communications hit rate</td>
<td>1732</td>
<td>44</td>
<td>Custom</td>
</tr>
</tbody>
</table>

† Due to memory restrictions on our embedded platforms, we used 2186 instances on the target platforms.

---

1 An implementation of LibSVM is also available as part of Weka; we used the direct download [3].
examined cache use and loop unrolling; these had no effect on compiler times on either platform.

A. Collapsing Object Structure and Inlining

Table 4 shows that many functions in Baseline are called many times, and are thus good candidates for inlining. Across all 19 datasets, Baseline Weka makes over 1 billion calls to the eight most common functions.

Our first step was to collapse the Object hierarchy of the original Weka code, to reduce indirection in the call stack and to improve readability of the code. Baseline had 21 CPP files (plus corresponding H files), of which 14 corresponded to Weka functionality and 7 to data import. We flattened to 8 CPP files (2 importers). For example, Baseline’s Puk, Kernel, and CachedKernel became a single class in our new code, Puk. Similarly, Baseline’s RegSMOImproved, RegOptimizer, SMOreg, RegSMO and Classifier became a single class in our new code, SVMOptimizer. We incorporated the abilities of Baseline’s ReplaceMissingValues and Normalize into Attribute.

We then repeated the profile analysis of the codebase, and forced small, frequently-called functions to be inlined, including those in SMOset, Puk, Attribute, and SVMOptimizer. (We left longer functions alone.) Notably, the compiler on PPC440 is unable to inline any of the six SMOset functions, despite the fact that each are only a few lines long. We first placed the function in the HPP file (rather than in the CPP file), and then verified that the compiler was able to successfully inline the call. The compiler could not inline three key functions, so we created #define macros to replace these; each of these were only called in one or two places in the code, and thus we did not suffer from a bloated assembly.

B. Numerical Representations

The SVM code relies on many floating point mathematics operations. To build a SVM, the code repeatedly computes a predicted value and its corresponding error, and stops the algorithm when error is sufficiently low. It is therefore critical to find a numerical representation that can be computed quickly and still meet accuracy requirements.

We performed the unit tests of Table 5 on our target hardware for 64-bit double, 32-bit float, 32-bit integer, and a fixed point representation [5], [20]. Neither platform has a floating point co-processor. The results in Table 6 indicate that integer computations take only 1% of double precision and 3% of float operations on PPC440 (3% and 8% on ARMv7 respectively). Fixed Point was extremely slow: Table 7 and Table 8 show the assembly for the Fixed Point multiply-accumulate operation on ARMv7 and PPC440 respectively. The PPC440 requires eight instruction for each load, add, multiply, divide, and store on a fixed point value, explaining the extremely slow timing results.

We eliminated fixed point representation because the timing results were so poor, and then updated the Weka code to support side-by-side testing of the other representations. We also developed an intermediate mixed int+float version intended to take advantage of the integer speed improvements without impacting accuracy.

1) Double: All numbers in Weka are 64-bit double, using a type definition, InstData.

2) Float: All numbers in Weka are 32-bit float, using a type definition, InstData.

3) Integer: All numbers in Weka are 32-bit integer. Our approach was to scale all of the values $\alpha$ and kernel parameters by a scaling factor $F$, and scale the data (or target values) and error by $F^2$.

4) Mixed Float and Integer: To leverage the potential timing improvement from integer math as indicated by Table 6, while not losing as much accuracy as for a fully integer representation, we focussed on converting the two key inner loops: dotProd() and SVMOutput(), both of which are multiply-accumulate loops. dotProd() computes the dot product between two instances; it is called $2n^2$ times, for $n$ instances in the dataset. SVMOutput() calculates the predicted value for a given instance, per Equation 2. It is a function the Lagrangian multipliers $\alpha$, and the kernel evaluation $k$. SVMOutput() is called approximately $10n$ times when building a model, depending on the complexity of the underlying data.

In both cases, we use a scaling (or normalizing) factor $F$ outside the loop. The loop itself operates on scaled integer values. For example, Table 9 shows pseudocode from the original Weka implementation of SVMOutput(). Table 10 shows how we scale the $\alpha$ and kernel value $k$ separately, and then de-scale the sum outside the loop. This single floating-point division outside the loop is much cheaper than many inner-loop floating-point multiplies.

Note that evalKernel() of Table 10 contains a floating point multiply and round. This multiply is computed relatively infrequently due to caching of the kernel values (approximately 90% of all calls are cache hits).
TABLE 5. To measure the performance of each numerical type, we timed the three loops Creation, Division, and Multiply-Accumulate on vectors of 250,000 elements.

```cpp
template <typename T>
void measure()
{
    size_t i, j;
    static T values[250000];
    for (i = 0; i < 250000; ++i) {
        values[i] = T((i & 0x7fff) + 1);
    }
    static T results[250000];
    for (i=0; i < 250000; ++i)
    {
        results[i] = values[i];
    }
    const T numerator(127);
    for (i=0; i < 250000; ++i)
    {
        results[i] = numerator / values[i];
    }
    T sum(0);
    for (i=0; i < 250000; ++i)
    {
        for (j=0; j < 250000; ++j)
        {
            results[i] += values[i] * values[j];
        }
    }
}
```

TABLE 6. Integer computations are significantly faster than float or double; Fixed point representation is inappropriate for software on PPC440.

<table>
<thead>
<tr>
<th></th>
<th>PPC440 (µs)</th>
<th>ARMv7 (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>5,962</td>
<td>5,247</td>
</tr>
<tr>
<td>float</td>
<td>20,436</td>
<td>9,004</td>
</tr>
<tr>
<td>double</td>
<td>5,196</td>
<td>6,355</td>
</tr>
<tr>
<td>fixed</td>
<td>187,095</td>
<td>84,000</td>
</tr>
<tr>
<td></td>
<td>395,314</td>
<td>141,249</td>
</tr>
<tr>
<td></td>
<td>409,291</td>
<td>22,586</td>
</tr>
</tbody>
</table>

TABLE 7. Assembly code on ARMv7 is very efficient.

```
mov r7, #0 set r7 to be "j"
mov s1, r7 set s1 to be "i"
.L16:
    ldr r3, [r8, sl] r3 = values[i]
    mov lr, #0
    mov r5, r3
    mov r6, r5, asr #31
    .L17:
    ldr r1, [r8, lr] r1 = values[j]
    add lr, lr, #4
    mov r2, r1, asr #31
    mul ip, r5, r2
    umull r3, r4, r5, r1
    mla r0, r1, r6, ip
    mov r2, r3, lar #16
    add r4, r0, r4
   orr r2, r2, r4, asl #16
    cmp lr, #4000
    add r7, r7, r2
    bne .L17 branch back to .L17
    add sl, sl, #4
    cmp sl, #4000
    bne .L16 branch back to .L16
```

TABLE 8. Assembly code on PPC440 requires eight instructions per value.

```
#128: /* multiply accumulate */
#129: const size_t dim = 1000;
#130: T sum(0);
li r19, 0
#132: for (size_t i = 0; i != dim; ++i) # .bs
.LD263:
    mr r18, r19
    li r20, 500
    li r21, 4
    subi r4, r31, 8
.L12047:
    # .bs
.LD363:
    lwzu r3, 8(r4)
    li r5, 125
    subi r6, r31, 32
    mtctr r5
.L12055:
    lwzu r7, 32(r6)
    lwz r10, 20(r6)
    lwz r8, 16(r6)
    lwz r9, 4(r6)
    lwz r5, 12(r6)
    lwz r11, 8(r6)
    lwz r12, 24(r6)
    lwz r0, 28(r6)
    mulw r16, r3, r7
    mulw r7, r3, r8
    mulw r17, r3, r11
    mulw r11, r3, r12
    mulw r12, r3, r9
    mulw r9, r3, r10
    mulw r15, r3, r5
    mulw r5, r3, r0
    add r19, r19, r16
    add r19, r19, r12
    add r19, r19, r17
    add r19, r19, r15
    add r19, r19, r17
    add r19, r19, r19
    add r19, r19, r19
    add r19, r19, r5
    bdnz .L12055
    # .es
.LD463:
    lwz r16, 4(r4)
    li r5, 125
    subi r17, r31, 32
    mtctr r5
    .L12066:
    # (8 lwz then 8 mulw then 8 add)
    bdnz .L12066
    subic. r20, r20, 1
    addi r21, r21, 8
    addi r18, r18, 2
    bne .L12047
    # .es
.LD563:
```
C. Kernel Implementation

We rigorously examined the code for correctness according to the algorithm definition for SVMs, and discovered several small inefficiencies. For example, the PuK kernel of Equation 1 squares the value of a square root: \((\sqrt{\text{sqDist}})^2\). This function requires calls to \(\text{pow}()\) and \(\text{sqrt}()\) in the code, both of which are expensive floating point computations.

We can simplify Equation 1 by removing the call to square root and hence the subsequent floating point multiply:

\[
\phi(x_i, x_j) = \frac{1}{1 + \frac{1}{\omega} (||x_i - x_j||^2 (2(1/\omega) - 1))}\]

We can also insert a simple test for when \(\omega\) is 1.0, allowing us to avoid calling \(\text{pow}()\). Table 11 shows the original pseudo code for Equation 1, while the Table 12 shows the modified pseudo code for the simpler Equation 3.

D. Memory vs Computation

We also explored the tradeoff between computing values every time, versus caching them in memory. Two key items are

The original function squares a square root, and always calls \(\text{pow}()\) even when the exponent is 1.0.

\[
\phi(x_i, x_j) = \frac{1}{1 + \frac{1}{\omega} (||x_i - x_j||^2 (2(1/\omega) - 1))}
\]

We can also insert a simple test for when \(\omega\) is 1.0, allowing us to avoid calling \(\text{pow}()\). Table 11 shows the original pseudo code for Equation 1, while the Table 12 shows the modified pseudo code for the simpler Equation 3.

We ensured that our vector and matrix accesses are efficient, by eliminating function calls and using fixed-size memory, per Section IV-F.
TABLE 14. Weka computes all dot products every time they are needed.

```c
typedef InstData float or double
// x: index of instance 1
// y: index of instance 2
// a: number of attributes
InstData dotProd( int x, int y )
InstData result = 0
for (i=0; i < a; ++i)
   result += inst[x].attr[i] * inst[y].attr[i]
return result
```

TABLE 15. We precompute the dot products and cache them; we also use pointer addition to move along the vectors.

```c
typedef InstData float or double
// n: number of instances
// a: number of attributes
void computeDotProds()
   for (x = 0; x < n; ++x)
      InstData* xp = &inst[x].attr[0]
      InstData* yp = &inst[y].attr[0]
      for (j = 0; j < a; ++j)
         dpM[x, y] = result
```

TABLE 16. Assembly generated by GHS compiler for std::vector includes a function call.

```c
8942 # for (size_t j = 0; j < 8192)
8943 # bs
8944 .L15251:
8945 lwz r4, 4(r23)
8946 addi r25, sp, 60
8947 mr r24, r31
8948 .L7635:
8949 # bs
8950 .L15252:
8951 li r19, 0
8952 li r20, 512
8953 mr r22, r23
8954 mr r23, r24
8955 .L7643:
8956 lwz r4, 4(r23)
8957 mr r3, r22
8958 bl _std_Vector_iterator
8959 lwz r4, 4(r23)
8960 slwi r12, r19, 2
8961 mr r3, r22
8962 add r27, r27, r12
8963 bl _std_Vector_iterator
8964 mr r3, r22
8965 add r27, r27, r12
8966 add r27, r27, r12
8967 add r19, r19, 8
8968 bne .L7643:
8969 move r19, r27,
8970 addi r25, sp, 60
8971 mr r24, r31
8972 .L7635:
8973 li r19, 0
8974 li r20, 512
8975 mr r22, r23
8976 mr r23, r24
8977 .L7643:
8978 lwz r4, 4(r23)
8979 mr r3, r22
8980 bl _std_Vector_iterator
8981 lwz r4, 4(r23)
8982 slwi r12, r19, 2
8983 mr r3, r22
8984 add r27, r27, r12
8985 bl _std_Vector_iterator
8986 mr r3, r22
8987 add r27, r27, r12
8988 add r27, r27, r12
8989 add r19, r19, 8
8990 bne .L7643:
```

E. Removing exceptions

In examining the assembly code for the PPC440 platform, we discovered that GHS adds overhead (4 instructions) to every function that declares a non-trivial object on the stack. To reduce the impact of this overhead we

- Inlined as many functions as we could (Section IV-A)
- Disabled exceptions
- Removed all throw, try and catch statements. We achieved this by changing throw to a #define macro BBN_THROW. When exceptions are disabled, BBN_THROW results in an abort. We used #ifdef to eliminate try and catch statements.

F. Vector and Matrix Optimization

This section describes optimizations for vectors (containers of values of the same type stored contiguously) and matrices (multi-dimensional containers of values of the same type). We consider these together because their similar underlying representations and interfaces led to similar optimization approaches.

We make extensive use of the C++ vector container, especially for maintaining Instance and Support Vector data. In analyzing the assembly code produced by GHS on PPC440, we observed that every indexing operation (`std::vector<T>::operator[]`) generates a function call. Table 16 shows the assembly generated by GHS compiler for `std::vector`. The compiler automatically unrolls the loop of Table 17, but produces a function call in the listing (the `bl` instruction).

To eliminate this overhead, we developed a minimal implementation of the `std::vector` interface. Our implementation uses a single, fixed size allocation; i.e. the vector size is specified during template instantiation. By having the vector size established at compile time, we were able to eliminate all but the initial heap allocation/deallocation operations. We also implemented bounds and safety checks as `assert()` calls, which we could eliminate or enable entirely at compile time. Moreover, the simpler indexing operations allowed the GHS compiler to successfully inline our implementation of `operator[]` (Table 18).

On our PPC440 platform, running a focussed unit test to determine the effect of our vector library, we initially observed a 17x speed improvement with safety checks enabled and a 69x improvement with safety checks disabled. Table 19 shows the timing results when we first integrated our module into the overall system, showing an 6.7x runtime improvement.

Weka’s kernel matrices are triangular. We also use a triangular matrix for caching instance dot products to avoid recomputing them. These matrices are large (n²) and have dynamic size; we add a row to the matrix for each new instance.

Representing a triangular matrix with a square matrix would needlessly allocate unused memory. In our initial implementation, we used a `std::vector` for rows, each of which is a `std::vector` of columns. For dynamic resizing, we simply called `std::vector::resize()`. For triangular matrices, we developed a new triangular matrix implementation based on a fixed-size, one-dimensional array that uses index arithmetic. The `bbn::vector` class allocates only half of a square matrix. The index of `x[i, j]` in the one-dimensional array is \(\frac{1}{2}i(i + 1) + j\).

V. Results

We performed a series of ablation experiments, in which we independently removed each optimization from the final version of the code, leaving other optimizations in place. This approach allows us to evaluate the impact of each optimization independently. We explicitly measured both timing and accuracy, and measured memory through analysis.
We tag the original Weka code as the *Baseline*; *Baseline* is the first working C++ implementation we had. *Mixed* is the version containing all of our intended optimizations: Mixed integer+float, Sqrt, Pow, Disabling exceptions, etc. Section V-B describes why we chose the *Mixed* representation as the basis for additional trials.

We tag each ablation trial as the name of the relevant optimization(s) subtracted from *Mixed*. Thus *Double* means that all numerical representations in Weka use double representation, but the other optimizations (Disabled exceptions, Customized vectors, etc) are the same as in *Mixed*. Similarly, *Pow+Sqrt* means that the *sqrt()* and *pow()* functions are always called per Section IV-C, but it uses mixed int+float and other optimizations as in *Mixed*.

Because we have not discussed all of the optimizations we made to the *Baseline*, we created a version without any of the optimizations discussed in this paper, *Reference*, which allows the reader to estimate how much of the change is due to the presented optimizations. *Reference* removes all of the pre-supplied optimizations: it uses double, it enables exceptions, it calls *sqrt()* and *pow()*, it uses std::vector, it always computes dot products, and it does not inline functions.

Table 20 shows the tests we performed. Each row indicates an optimization, each column indicates an ablation trial, and an optimization, each column indicates an ablation trial, and each cell indicates the setting for that capability in that trial. *Baseline* is the most similar version of the optimizations discussed in this paper, *Reference*, which allows the reader to estimate how much of the change is due to the presented optimizations. *Reference* removes all of the pre-supplied optimizations: it uses double, it enables exceptions, it calls *sqrt()* and *pow()*, it uses std::vector, it always computes dot products, and it does not inline functions.

Table 21 and 22 show the results of these experiments, in terms of time and accuracy of each trial. Fig. 1 and Fig. 2 plot the PPC440 results visually, where the ablation trials on the x-axis are sorted by average time.

- Each row represents a unit test from a benchmark dataset of Table 2. Not all tests could be run on both platforms.
TABLE 22. PPC440. Baseline cannot run on PPC440. Mixed int+float is better than other numerical representations, per Section V-B. Disabling exceptions makes a big difference, see Section V-E. sqrt () (Section V-C), dotprod (Section V-D) and vectors (Section V-F) depend on platform and dataset. Time (ms) in top, NormRMSE in bottom. Matches Table 21 on ARMv7. Bold is the best item. Fig. 1 and Fig. 2 plot these results visually, where the ablation trials on the x-axis are sorted by average time.

<table>
<thead>
<tr>
<th>Function</th>
<th>Time (ms)</th>
<th>NormRMSE</th>
<th>ref</th>
<th>float</th>
<th>double</th>
<th>inline</th>
<th>exc+vec+tri</th>
<th>dotprod+x[i]</th>
<th>pow+sqrt</th>
<th>vec+tri</th>
<th>dotprod+srk</th>
</tr>
</thead>
<tbody>
<tr>
<td>AutoPrice</td>
<td>475</td>
<td>2.16</td>
<td>0.72</td>
<td>1.14</td>
<td>0.50</td>
<td>0.59</td>
<td>0.58</td>
<td>0.44</td>
<td>0.51</td>
<td>0.61</td>
<td>1.11</td>
</tr>
<tr>
<td>bodyfat</td>
<td>531</td>
<td>3.01</td>
<td>1.41</td>
<td>0.62</td>
<td>0.86</td>
<td>1.00</td>
<td>0.80</td>
<td>0.72</td>
<td>0.86</td>
<td>1.06</td>
<td>1.91</td>
</tr>
<tr>
<td>concrete</td>
<td>3,650</td>
<td>5.610</td>
<td>1.209</td>
<td>1.103</td>
<td>0.604</td>
<td>0.767</td>
<td>1.064</td>
<td>0.504</td>
<td>0.604</td>
<td>0.908</td>
<td>1.524</td>
</tr>
<tr>
<td>cpu</td>
<td>603</td>
<td>0.911</td>
<td>0.11</td>
<td>0.13</td>
<td>0.135</td>
<td>0.154</td>
<td>0.57</td>
<td>0.112</td>
<td>0.127</td>
<td>0.156</td>
<td>0.295</td>
</tr>
<tr>
<td>fishcatch</td>
<td>830</td>
<td>0.469</td>
<td>0.62</td>
<td>0.230</td>
<td>0.114</td>
<td>0.128</td>
<td>0.56</td>
<td>0.103</td>
<td>0.114</td>
<td>0.131</td>
<td>0.227</td>
</tr>
<tr>
<td>housing</td>
<td>1,719</td>
<td>1.333</td>
<td>3.663</td>
<td>0.160</td>
<td>0.189</td>
<td>0.169</td>
<td>0.189</td>
<td>0.189</td>
<td>0.189</td>
<td>0.189</td>
<td>0.189</td>
</tr>
<tr>
<td>wine red</td>
<td>15,538</td>
<td>5.634</td>
<td>3.292</td>
<td>2.122</td>
<td>2.610</td>
<td>2.326</td>
<td>2.635</td>
<td>2.195</td>
<td>2.608</td>
<td>3.810</td>
<td>6.121</td>
</tr>
<tr>
<td>wine white</td>
<td>13,360</td>
<td>5.926</td>
<td>2.474</td>
<td>4.461</td>
<td>2.306</td>
<td>2.652</td>
<td>2.822</td>
<td>1.985</td>
<td>2.301</td>
<td>2.910</td>
<td>4.959</td>
</tr>
<tr>
<td>pole</td>
<td>11,518</td>
<td>6.065</td>
<td>2.109</td>
<td>1.588</td>
<td>2.624</td>
<td>3.056</td>
<td>2.218</td>
<td>2.269</td>
<td>2.621</td>
<td>3.404</td>
<td>5.715</td>
</tr>
<tr>
<td>commasA-TP</td>
<td>6,790</td>
<td>5.645</td>
<td>1.875</td>
<td>3.655</td>
<td>2.082</td>
<td>2.335</td>
<td>1.557</td>
<td>1.869</td>
<td>2.081</td>
<td>2.481</td>
<td>4.376</td>
</tr>
<tr>
<td>commasB-Lat</td>
<td>9,539</td>
<td>5.054</td>
<td>2.480</td>
<td>2.560</td>
<td>1.990</td>
<td>2.233</td>
<td>2.243</td>
<td>1.760</td>
<td>1.989</td>
<td>2.373</td>
<td>4.154</td>
</tr>
<tr>
<td>commasB-TP</td>
<td>6,389</td>
<td>1.948</td>
<td>1.088</td>
<td>1.024</td>
<td>0.753</td>
<td>0.854</td>
<td>1.401</td>
<td>0.684</td>
<td>0.754</td>
<td>1.832</td>
<td>1.623</td>
</tr>
<tr>
<td>commasC-Lat</td>
<td>2,629</td>
<td>3.347</td>
<td>1.371</td>
<td>1.109</td>
<td>0.756</td>
<td>0.903</td>
<td>0.881</td>
<td>0.667</td>
<td>0.755</td>
<td>1.013</td>
<td>1.786</td>
</tr>
<tr>
<td>commasC-TP</td>
<td>1,459</td>
<td>7.88</td>
<td>0.330</td>
<td>0.159</td>
<td>0.437</td>
<td>0.465</td>
<td>0.327</td>
<td>0.393</td>
<td>0.420</td>
<td>0.495</td>
<td>0.876</td>
</tr>
<tr>
<td>commasC-Lat</td>
<td>1,125</td>
<td>6.21</td>
<td>0.328</td>
<td>0.425</td>
<td>0.423</td>
<td>0.471</td>
<td>0.381</td>
<td>0.405</td>
<td>0.416</td>
<td>0.497</td>
<td>0.917</td>
</tr>
</tbody>
</table>

Each column represents an ablation trial. The Baseline could not run on the PPC440, and thus Reference is the closest representation we have of Baseline on this platform.

The top half of each table shows the time to build the model, in milliseconds. Boldface indicates the fastest time.

The bottom half of each table shows the NormRMSE of the model, as defined in Table 23. Boldface indicates the lowest NormRMSE.

Several time results below are explained in terms of the convergence to build the model. Convergence in an SVM is defined as how many examples violate optimality conditions [19], and is measured by the number of examples that cause the model to change. The numerical representation (Section V-B) enforces most functions in Weka as functions with overhead related to the kernel. Column Puk enforces most functions in SMOset, and some functions in the SMOset+SVMOptimizer concepts by Weka module. Column SMOset+SVMOptimizer removes inlining for all functions in SMOset, and some functions in the SVMOptimizer. Column Puk removes inlining for functions related to the kernel. Column Inline inlines minimal inlining; it enforces most functions in Weka as functions with overhead incuding SMOset, SVMOptimizer, Puk, and Attribute.

A. Collapsing Object Structure and Inlining

Column Inline of Table 21 and Table 22 shows the timing and NormRMSE results when most Weka functions are function calls, while column Mixed shows the results when these are inlined. Inlining function calls reduces runtime by about 20%.

Table 24 shows that inlining the functions reduces function-call overhead from 1 billion calls in Baseline, to 20 million calls in Mixed. Differences between Baseline and Inline are partially due to flattening, and partially due to the total number of calls to converge because we move from Double to Mixed (Section V-B).

TABLE 23. NormRMSE measures the accuracy of the learned model.

Normalized Root-Mean-Squared Error (NormRMSE) is a measure of how well the learned model performs. It measures the difference between values predicted by the SVM model and the values observed in the training data. We compute it as the Root-Mean-Squared error divided by the standard deviation of the data:

\[
\text{NormRMSE} = \sqrt{\frac{1}{n} \sum_{i=1}^{n} (\hat{y}_i - x_i)^2}
\]

where \( n \) is the number of instances in the data, \( \mu \) is the average target value, \( \hat{y}_i \) is the observed value of instance \( i \), and \( x_i \) is the predicted value of instance \( i \) using Equation 2.

The standard deviation represents the performance of a learner that uses the mean as the prediction for all instances. Our goal is to achieve as low a NormRMSE as possible; 0.0 indicates that every instance is predicted with no error, while 1.0 indicates that no “fancy” model is needed because the mean value is just as good.

Table 25 shows the average results for the two platforms and groups of datasets, and also breaks down the inlining concepts by Weka module. Column Mixed inlines most Weka functions. Column SMOset+SVMOptimizer removes inlining for all functions in SMOset, and some functions in the SVMOptimizer. Column Puk removes inlining for functions related to the kernel. Column Inline has minimal inlining; it enforces most functions in Weka as functions with overhead incuding SMOset, SVMOptimizer, Puk, and Attribute. The
Fig. 1. PPC440, communications datasets. While ablation trials tend to yield a similar performance improvement across datasets, characteristics of the dataset can cause varied results.

Fig. 2. PPC440, non-communications datasets. While ablation trials tend to yield a similar performance improvement across datasets, characteristics of the dataset can cause varied results.

<table>
<thead>
<tr>
<th>TABLE 24</th>
<th>By inlining functions, We reduce 1 billion function calls from Baseline to 20 million calls in Mixed.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SMOset</strong></td>
<td><strong>Baseline</strong></td>
</tr>
<tr>
<td><strong>airfoil</strong></td>
<td>26,339,476</td>
</tr>
<tr>
<td><strong>autoPrice</strong></td>
<td>551,778</td>
</tr>
<tr>
<td><strong>bodyfat</strong></td>
<td>777,743</td>
</tr>
<tr>
<td><strong>concrete</strong></td>
<td>16,689,819</td>
</tr>
<tr>
<td><strong>housing</strong></td>
<td>6,996,474</td>
</tr>
<tr>
<td><strong>wine red</strong></td>
<td>24,184,029</td>
</tr>
<tr>
<td><strong>wine white</strong></td>
<td>54,699,503</td>
</tr>
<tr>
<td><strong>pole</strong></td>
<td>68,449,393</td>
</tr>
<tr>
<td><strong>commsA-TP</strong></td>
<td>11,413,877</td>
</tr>
<tr>
<td><strong>commsA-Lat</strong></td>
<td>10,454,653</td>
</tr>
<tr>
<td><strong>commsB-TP</strong></td>
<td>10,488,652</td>
</tr>
<tr>
<td><strong>commsB-Lat</strong></td>
<td>8,282,091</td>
</tr>
<tr>
<td><strong>commsC-TP</strong></td>
<td>1,958,177</td>
</tr>
<tr>
<td><strong>commsC-Lat</strong></td>
<td>1,765,778</td>
</tr>
<tr>
<td><strong>commsD-TP</strong></td>
<td>56,020,056</td>
</tr>
<tr>
<td><strong>commsD-Lat</strong></td>
<td>26,854,672</td>
</tr>
<tr>
<td><strong>commsD-BER</strong></td>
<td>39,976,688</td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td>367,987,492</td>
</tr>
<tr>
<td><strong>compare to inline</strong></td>
<td>0.0%</td>
</tr>
<tr>
<td><strong>compare to baseline</strong></td>
<td>0.0%</td>
</tr>
</tbody>
</table>

B. Numerical Representations

The columns `Double`, `Float`, `Integer`, and `Mixed` in Table 21 and Table 22 show the results for each of the four numerical representations. (Note that the other columns to the right of `Mixed` all use the mixed integer+float representation.)

<table>
<thead>
<tr>
<th>TABLE 25</th>
<th>Inlining functions cuts runtime to about 80%. This table summarizes data in Table 21 and Table 22.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Average Time (ms)</strong></td>
<td><strong>Minimal</strong></td>
</tr>
<tr>
<td><strong>Platform (dataset)</strong></td>
<td><strong>Baseline</strong></td>
</tr>
<tr>
<td>PPC440 (comms only)</td>
<td>1,295</td>
</tr>
<tr>
<td>PPC440 (all datasets)</td>
<td>1,476</td>
</tr>
<tr>
<td>ARMv7 (comms only)</td>
<td>1,149</td>
</tr>
<tr>
<td>ARMv7 (all datasets)</td>
<td>1,107</td>
</tr>
<tr>
<td><strong>Average NormRMSE</strong></td>
<td>0.315</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TABLE 26</th>
<th>On average, Mixed is consistently faster than the other representations, for a small impact on NormRMSE. This table summarizes data in Table 21 and Table 22.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Platform (dataset)</strong></td>
<td><strong>Average Time (ms)</strong></td>
</tr>
<tr>
<td>PPC440 (comms only)</td>
<td>4,464</td>
</tr>
<tr>
<td>PPC440 (all datasets)</td>
<td>5,192</td>
</tr>
<tr>
<td>ARMv7 (comms only)</td>
<td>19,905</td>
</tr>
<tr>
<td>ARMv7 (all datasets)</td>
<td>20,659</td>
</tr>
<tr>
<td><strong>Average NormRMSE</strong></td>
<td>0.310</td>
</tr>
</tbody>
</table>

As expected, `Double` representation is always slower than the other representations. Table 26 summarizes the raw data of Table 21 and Table 22, showing the average runtimes for the two platforms, for all of the datasets and for the subset of datasets we are most interested in. On average, Mixed is consistently faster than either Float or Integer, although there are specific cases in which Float or Integer performs faster than Mixed.

majority of accessor functions like `getValue()` are inlined, but functions that perform meaningful computation are not inlined.

B. Numerical Representations

The columns `Double`, `Float`, `Integer`, and `Mixed` in Table 21 and Table 22 show the results for each of the four numerical representations. (Note that the other columns to the right of `Mixed` all use the mixed integer+float representation.)
While some might expect Integer to perform consistently fastest because there are never any floating point operations, building a strictly integer SVR model may require more overall computations to converge. The last two rows of Table 27 indicate how well correlated these results are to the runtime results of Table 22. The primary conclusion is that the properties of the data drive the runtime: the harder it is to build a model that fits the data, the longer it takes to build the model.

When choosing the final representation for our target platform, our secondary factor is the NormRMSE: we do not want to sacrifice too much accuracy. While some might expect Bonk to perform consistently best, there are never any floating point operations, building a strictly integer SVR model may require more overall computations to converge. The last two rows of Table 27 indicate how well correlated these results are to the runtime results of Table 22. The primary conclusion is that the properties of the data drive the runtime: the harder it is to build a model that fits the data, the longer it takes to build the model.

Overall Mixed runs at 22% of the runtime of the Reference on PPC440, 30% of the runtime of Reference on ARMv7, and 6% of the runtime of Baseline on ARMv7.

Given that our primary interest is for communications datasets, we have selected the Mixed numerical representation for all further tests. Thus, columns Except, Pow+sqrt, etc. can all be directly compared to the Mixed column.

C. Kernel Implementation

Column pow+sqrt in Table 21 and Table 22 shows the performance results when we enable the sqrt() and pow() functions from the original kernel computation for PuK (Table 11). Table 28 shows the average runtimes for our two platforms, for all of the datasets and for our communications datasets. In all cases, pow() is slightly cheaper than Mixed because our tests used $\omega = 1.0$, and our code from Table 12 incurs an if call; we have since removed the if from our code.

On both platforms, sqrt() is extremely expensive, particularly for the double representation. On PPC440, for example, a single call to the sqrtf() function for float is 0.129\$\mu s$, sqrt() for double is 0.455\$\mu s$. With over a million calls to this function, we can quickly account for a large percentage over overall compute time. Despite this standalone performance, when we remove the functions from the system, we get mixed performance results: on average removing these two functions yields a runtime that is 118% of Mixed, and almost identical on ARMv7.

On PPC440, removing these two functions can improve runtime dramatically compared to Mixed (54% for commsB-TP), or it make runtime significantly worse (236% for cpu). On ARMv7, the range is 38% (for fishcatch) to 135% (for commsB-Lat). When we remove the sqrt() call, we increase overall floating point accuracy, and this may actually cause Weka to take longer to converge. Table 29 shows the runtime and total number of changed examples for Mixed and pow+sqrt on PPC440; the number of changed examples is 96% correlated to the runtime across all testcases.

Just as for the numerical representations (Section V-B), the results for sqrt() and pow() depend on the specific platform and dataset. While we expected that removing these functions would reduce computation time, the increased convergence time might yield slower overall performance. This was yet another example where our intuitive expectation did not bear out.

D. Memory vs Computation

Recall that caching the kernel results saved 600% of runtime, and thus it is clear that caching the kernel computations is usually worth the memory cost of a fixed-size $n \times n$ triangular matrix.

The timing results for caching dot products, however, show that caching dot products are probably not worth the memory
TABLE 30. Computing (dotprod-x[i] and dotprod-*x++) or caching (Mixed) the dot products has little impact on run times when exceptions are disabled. There is no impact on NormRMSE. This table summarizes data in Table 21 and Table 22. (Table 31 shows that caching dot products is useful when exceptions are enabled.)

<table>
<thead>
<tr>
<th>Platform (dataset)</th>
<th>Average Time (ms)</th>
<th>dotprod-x[i]</th>
<th>dotprod-*x++</th>
<th>Mixed</th>
</tr>
</thead>
<tbody>
<tr>
<td>PPC400 (comms only)</td>
<td>1.069</td>
<td>1.065</td>
<td>1.073</td>
<td></td>
</tr>
<tr>
<td>PPC400 (all datasets)</td>
<td>1.125</td>
<td>1.124</td>
<td>1.127</td>
<td></td>
</tr>
<tr>
<td>ARM7 (comms only)</td>
<td>963</td>
<td>955</td>
<td>938</td>
<td></td>
</tr>
<tr>
<td>ARM7 (all datasets)</td>
<td>1.048</td>
<td>1.039</td>
<td>1.012</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Platform (dataset)</th>
<th>Average NormRMSE</th>
<th>dotprod-x[i]</th>
<th>dotprod-*x++</th>
<th>Mixed</th>
</tr>
</thead>
<tbody>
<tr>
<td>PPC400 (comms only)</td>
<td>0.315</td>
<td>0.315</td>
<td>0.315</td>
<td></td>
</tr>
<tr>
<td>PPC400 (all datasets)</td>
<td>2.289</td>
<td>2.289</td>
<td>2.107</td>
<td></td>
</tr>
<tr>
<td>ARM7 (comms only)</td>
<td>0.307</td>
<td>0.307</td>
<td>0.307</td>
<td></td>
</tr>
<tr>
<td>ARM7 (all datasets)</td>
<td>0.272</td>
<td>0.272</td>
<td>0.272</td>
<td></td>
</tr>
<tr>
<td>ARM7 (all datasets)</td>
<td>0.308</td>
<td>0.308</td>
<td>0.308</td>
<td></td>
</tr>
</tbody>
</table>

Table 31. When exceptions are enabled, runtimes are slower on PPC440, and have little impact on ARMv7. There is no impact on NormRMSE. This table summarizes data in Table 21 and Table 22.

<table>
<thead>
<tr>
<th>Platform (dataset)</th>
<th>Average Time (ms)</th>
<th>exc+</th>
<th>vec+tri</th>
<th>exc+</th>
<th>vec+tri</th>
<th>exc+</th>
<th>vec+tri</th>
<th>dp+</th>
<th>except</th>
<th>Mixed</th>
</tr>
</thead>
<tbody>
<tr>
<td>PPC400 (comms only)</td>
<td>2.328</td>
<td>2.289</td>
<td>2.122</td>
<td>1.210</td>
<td>1.073</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PPC400 (all datasets)</td>
<td>2.508</td>
<td>2.492</td>
<td>1.317</td>
<td>1.322</td>
<td>1.127</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARM7 (comms only)</td>
<td>950</td>
<td>1.016</td>
<td>0.950</td>
<td>0.938</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARM7 (all datasets)</td>
<td>1.061</td>
<td>1.064</td>
<td>1.030</td>
<td>1.012</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

F. Vector and Matrix Optimization

Column vec+tri of Table 21 and Table 22 shows the timing and NormRMSE results when we use std::vector, while column Mixed shows the results with our custom vector and matrix modules. Table 32 shows the average results for our two platforms and dataset groups.

To our surprise, std::vector is faster than our custom bbn::vector. To validate our initial results (Table 19), we enabled exceptions in a variety of trials for dot products and vectors, corresponding to the columns exc+ in Table 31 and Table 32. Ablation trials specifically tied to the vectors and matrices are:

Table 32. Unexpectedly, our custom vector and matrix modules performed better than the Mixed when exceptions are disabled. When we enable exceptions, std::vector (column exc+vec+tri) performs significantly more poorly than Mixed. There is no impact on NormRMSE. This table summarizes data in Table 21 and Table 22.

<table>
<thead>
<tr>
<th>Platform (dataset)</th>
<th>Average Time (ms)</th>
<th>exc+vec+tri</th>
<th>vec+tri</th>
<th>triangle</th>
<th>vector</th>
<th>Mixed</th>
</tr>
</thead>
<tbody>
<tr>
<td>PPC440 (comms only)</td>
<td>2.328</td>
<td>2.289</td>
<td>1.222</td>
<td>2.120</td>
<td>2.107</td>
<td></td>
</tr>
<tr>
<td>PPC440 (all datasets)</td>
<td>2.492</td>
<td>2.492</td>
<td>2.492</td>
<td>2.492</td>
<td>1.127</td>
<td></td>
</tr>
<tr>
<td>ARMv7 (comms only)</td>
<td>950</td>
<td>950</td>
<td>950</td>
<td>950</td>
<td>0.307</td>
<td></td>
</tr>
<tr>
<td>ARMv7 (all datasets)</td>
<td>1.061</td>
<td>1.064</td>
<td>1.030</td>
<td>1.012</td>
<td>0.307</td>
<td></td>
</tr>
</tbody>
</table>

Therefore exc+vec+tri therefore corresponds to the performance results before we created our custom modules, and except corresponds to the results immediately after creating them.

The code relies heavily on triangular matrices for caching dot products and kernel results. As we expected initially, when exceptions are enabled our bbn::vector modules of except saved over 50% of compute time compared to std::vector of exc+vec+tri on PPC440. ARMv7 does not add the exception overhead, and thus there is no time savings. However, when exceptions are disabled, the std::vector of vec+tri is faster than our custom modules Mixed on PPC440.

VI. CONCLUSION

Table 33 summarizes the impact of our modifications to the Weka code. We obtained a final runtime that is on average 5% the runtime of Baseline on ARMv7. We were unable to get the original code to run on the PPC440 platform. The ablation trials revealed the independent effects of the optimizations. This paper presented the optimizations that either had the most effect, or were the most surprising. The following list summarizes the results, in approximate order of impact:

- Removing double saved at least 50% of runtime on both platforms. Improvements for Float, integer, and mixed int+float depend on the dataset and the platform. The SVM model is extremely sensitive to numerical representation, as all four representations can yield different NormRMSE results.

- Removing std::vector saved over 50% of compute time compared to bbn::vector on PPC440. ARMv7 does not add the exception overhead, and thus there is no time savings. However, when exceptions are disabled, the bbn::vector of exc+vec+tri is faster than our custom modules Mixed on PPC440.

- Removing std::vector saved over 50% of compute time compared to bbn::vector on PPC440. ARMv7 does not add the exception overhead, and thus there is no time savings. However, when exceptions are disabled, the bbn::vector of exc+vec+tri is faster than our custom modules Mixed on PPC440.

- Removing std::vector saved over 50% of compute time compared to bbn::vector on PPC440. ARMv7 does not add the exception overhead, and thus there is no time savings. However, when exceptions are disabled, the bbn::vector of exc+vec+tri is faster than our custom modules Mixed on PPC440.
TABLE 33. Our optimized version of Weka requires about 5% of the original runtimes. This table summarizes data in Table 21 and Table 22.

<table>
<thead>
<tr>
<th>Platform (dataset)</th>
<th>Compared to Baseline (%)</th>
<th>Compared to Reference (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PPC440 (comms only)</td>
<td>27.7%</td>
<td></td>
</tr>
<tr>
<td>PPC440 (all datasets)</td>
<td>11.7%</td>
<td></td>
</tr>
<tr>
<td>ARMv7 (comms only)</td>
<td>5.5%</td>
<td>37.4%</td>
</tr>
<tr>
<td>ARMv7 (all datasets)</td>
<td>6.1%</td>
<td>32.1%</td>
</tr>
</tbody>
</table>

- Disabling exceptions saves at least 50% of runtime (with std::vector) on PPC440, but has a smaller impact with (a) our custom vector module and (b) on ARMv7.
- Inlining functions saves about 20% of runtime, even on the ARMv7 platform whose more modern compiler can automatically inline more functions.
- Fixed-size vectors improves memory usage, but may not affect runtime.
- Removing sqrt() and pow() does not necessarily reduce runtime or improve accuracy, due to characteristics of the dataset.
- Adding a cache of the dot product calculations can improve performance when there are many attributes in the data.
- Creating a custom vector module (and not using std::vector) may be valuable on some platforms if exceptions are enabled.

The main lesson learned is to not make assumptions about how the hardware will respond to code structures. This was most noticeable for the surprising results when removing expensive function calls in the kernel implementation. We constructed our unit tests to monitor both time and accuracy; if either of these changed significantly, we inspected the assembly to find an explanation. Some changes we adopted, others we reverted, and others we redesigned to leverage positive effects and reduce negative effects.

REFERENCES


