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 embeddded 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 kB cache, 256MB RAM, vintage 2005. Linux 2.6.38.8, G++ 4.3.3, 2009.
PPC440:	IBM PPC440GX [1] at 533MHz, 256 kB 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 kB 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 rapidy 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 ϕ 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:

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

ω 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. σ controls the half-width at half-maxima.

The regression function then becomes:

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

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

There are many available implementations 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.

TABLE 1. *Several SVM packages are available in C++.*

Software	Language	License
DLib ML [15]	C++	Unlimited with copyright
LibSVM [3]	C++, Java	Unlimited with copyright
Shark [11]	C++	GNU Lesser GPL
SVM^{Light} [13], [14]	C	Free for science
TinySVM [16]	C++	GNU Lesser GPL
Weka [6], [10], [18]	Java, C++	GNU GPL

TABLE 2. *We measured performance against 19 regression datasets, of which 10 are standard benchmarks, and 9 represent different scenarios in our target communications domain.*

Test Name	Description	Instances	Attribs	Source
airfoil	Airfoil noise	1503	6	UCI [2]
AutoPrice	Automobile prices	159	16	Weka [10]
bodyfat	Percentage bodyfat	252	15	Weka [10]
concrete	Concrete compressive strength	1030	9	UCI [25],[2]
cpu	CPU relative performance	227	7	Weka [10]
fishcatch	Fish weight	159	8	Weka [10]
housing	Boston housing values	506	14	Weka [10]
pole	Telecommunications	14998†	26	LIAAC [23]
wine red	Wine quality, red	1599	12	UCI [4],[2]
wine white	Wine quality, white	4898†	12	UCI [4],[2]
commsA-TP	Communications throughput	1078	59	Custom
commsA-Lat	Communications latency	1078	59	Custom
commsB-TP	Communications throughput	820	59	Custom
commsB-Lat	Communications latency	820	59	Custom
commsC-TP	Communications throughput	378	59	Custom
commsC-Lat	Communications latency	378	59	Custom
commsD-TP	Communications throughput	1732	44	Custom
commsD-Lat	Communications latency	1732	44	Custom
commsD-BER	Communications bit error rate	1732	44	Custom

† *Due to memory restrictions on our embedded platforms, we used 2186 instances on the target 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.[1] 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 Ũstün'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

---

[1] An implementation of LibSVM is also available as part of Weka; we used the direct download [3].

TABLE 3. *Weka performs faster than other SVM libraries on our modern Linux server.*

Testcase	Weka	DLib C++	LibSVM
airfoil	350ms	385ms	9,270ms
autoPrice	8ms	6ms	23ms
bodyfat	12ms	15ms	66ms
Concrete	424ms	185ms	4,344ms
cpu	15ms	8ms	60ms
fishcatch	12ms	5ms	29ms
housing	103ms	63ms	305ms
wine red	433ms	1,066ms	9,127ms
wine white	2,626ms	5,257ms	68,006ms
pole	22,571ms	186,928ms	979,275ms
commsA-TP	276ms	427ms	1,948ms
commsA-Lat	264ms	735ms	2,217ms
commsB-TP	207ms	399ms	846ms
commsB-Lat	176ms	460ms	1,266ms
commsC-TP	39ms	135ms	94ms
commsC-Lat	35ms	150ms	100ms
commsD-TP	1,025ms	645ms	7,479ms
commsD-Lat	574ms	746ms	7,788ms
commsD-BER	790ms	954ms	7,600ms

TABLE 4. *Linux. For the 1503 instances of airfoil, at least five functions are called over a million times (`gprof`).*

% time	Num calls	Function name
0	122,409	SMOset::contains
1.27	2	SMOset::∼SMOset
1.27	1	RegOptimizer::init
2.53	1,503	RegOptimizer::SVMOutput
2.53	1	CachedKernel::numCacheHits
5.06	3,371,229	Puk::evaluate
6.33	4,279,744	RegSMOImproved::updateBoundaries
8.86	26,212,709	SMOset::getNext
11.39	90,074	RegSMOImproved::takeStep
16.46	11,057	RegOptimizer::SVMOutput
17.72	5,612,202	CachedKernel::dotProd
26.58	26,354,158	CachedKernel::eval

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 α and kernel parameters by a scaling factor $\mathcal{F}$, and scale the data (or target values) and error by $\mathcal{F}^2$.

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 α, 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 $\mathcal{F}$ 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 α 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.*

```
template <T>
void measure()
  size_t i,j;
  static T values[250000];
  for (i = 0; i < 250000; ++i) {// Base Data
    values[i] = T((i & 0x7fff) + 1);
  }
  static T results[250000];        // Creation
  for (i=0; i<250000; ++i) {
    results[i] = values[i];
  }
  const T numerator(127);          // Division
  for (i=0; i<250000; ++i) {
    results[i] = numerator / values[i];
  }
  T sum(0);                        // Mult-Accum
  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.*

	PPC440 (μs)		
	creation	**division**	**multiply-accumulate**
int	5,962	20,436	5,196
float	5,772	41,046	187,095
double	14,500	94,227	395,314
fixed	10,249	514,533	409,291
	ARMv7 (μs)		
	creation	**division**	**multiply-accumulate**
int	5,247	9,004	6,355
float	4,854	30,909	84,000
double	9,927	154,848	141,249
fixed	5,100	81,408	22,586

TABLE 7. *Assembly code on ARMv7 is very efficient.*

```
    mov     r7, #0                  set r7 to be "j"
    mov     sl, r7                  set s1 to be "i"
.L16:                               for (i=0; i!= dim; ++i)
    ldr     r3, [r8, sl]            r3 = values[i]
    mov     lr, #0
    mov     r5, r3
    mov     r6, r5, asr #31
.L17:                               for (j=0; j!=dim; ++j)
    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, lsr #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
.LDW263:
  mr r18, r19
  li r20, 500
  li r21, 4
  subi r4, r31, 8
.L12047:
#      .bs
.LDW363:
  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)
  mullw r16, r3, r7
  mullw r7, r3, r8
  mullw r17, r3, r11
  mullw r11, r3, r12
  mullw r12, r3, r9
  mullw r9, r3, r10
  mullw r15, r3, r5
  mullw r5, r3, r0
  add r19, r19, r16
  add r19, r19, r12
  add r19, r19, r17
  add r19, r19, r15
  add r19, r19, r7
  add r19, r19, r9
  add r19, r19, r11
  add r19, r19, r5
  bdnz .L12055
#      .es
.LDW463:
  lwz r16, 4(r4)
  li r5, 125
  subi r17, r31, 32
  mtctr r5
.L12066:

  # (8 lwz then 8 mullw then 8 add)

  bdnz .L12066
  subic. r20, r20, 1
  addi r21, r21, 8
  addi r18, r18, 2
  bne .L12047
#      .es
.LDW563:
```

TABLE 9. *The original Weka implementation does floating point multiplies.*

```
// typedef InstData: float or double
// id1: index of instance 1
// id2: index of instance 2

InstData evalKernel(int id1, int id2)
   Return cache(id1,id2) if possible
   InstData dp11 = dotProd(id1, id1)
   InstData dp12 = dotProd(id1, id2)
   InstData dp22 = dotProd(id2, id2)

   InstData sqDist = dp11 - 2.0*dp12
                        + dp22
   InstData fk = evaluatePuK( sqDist )
   cache(id1,id2) = fk
   return fk

// typedef InstData: float or double
// SV: Support Vectors
// index: instance to predict


InstData SVMOutput(int index)
   InstData result = -b
   for each SV$i$
      InstData $ad = \alpha[i] - \alpha^*[i]$
      InstData $k$ = evalKernel(index, $i$)
      result += $ad \times k$
   return result
```

TABLE 10. *The modified algorithm uses integer multiplies and scaling.*

```
// typedef InstData: float or double
// id1: index of instance 1
// id2: index of instance 2
// $\mathcal{F}$: scale factor
int evalKernel(int id1, int id2)
   Return cache(id1,id2) if possible
   InstData dp11 = dotProd(id1, id1)
   InstData dp12 = dotProd(id1, id2)
   InstData dp22 = dotProd(id2, id2)

   InstData sqDist = dp11 - 2.0*dp12
                        + dp22
   InstData fk = evaluatePuK( sqDist )
   int ik = round( $\mathcal{F}$ * fk )
   cache(id1,id2) = ik
   return ik

// typedef InstData: float or double
// SV: Support Vectors
// index: instance to predict
// $\mathcal{F}$: scale factor
// $\alpha^\delta[i] : \mathcal{F} * (\alpha[i] - \alpha^*[i])$
InstData SVMOutput(int index)
   int iResult = 0
   for each SV$_i$
      int $ad = \alpha^\delta[i]$
      int $k$ = evalKernel(index, $i$)
      iResult += $ad \times k$
   InstData fResult = iResult / $\mathcal{F}^2$
   return (fResult - b)
```

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: $(\sqrt{sqDist})^2$. This function requires calls to `pow()` and `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:

$$\phi(x_i, x) \quad = \quad \frac{1}{\left[1 + \frac{4}{\sigma^2}\left(||x_i - x||^2\right)\left(2^{(1/\omega)} - 1\right)\right]^{\omega}} \qquad (3)$$

We can also insert a simple test for when ω is 1.0, allowing us to avoid calling `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 dot products of all pairs instances (dotProd), and the kernel computations (eval). *Baseline* Weka caches kernel values but does not cache dot products. Table 4 shows that Weka calls these two functions 31 million times to build a model for the 1503 instances in the airfoil dataset.

TABLE 11. *The original function squares a square root, and always calls* `pow()` *even when the exponent is 1.0.*

```
// typedef InstData: float or double
// factor: $\frac{2}{\sigma} * \sqrt{2^{(1/\omega)} - 1}$
// sqDist: $||x - y||^2$
InstData evaluatePuK(InstData sqDist)
   InstData inter = factor * sqrt(sqDist);
   InstData sqInter = inter * inter;
   result = 1.0 / pow(1.0 + sqInter, $\omega$);
   return result;
```

TABLE 12. *The simplified function removes* `sqrt()`*, a floating point multiply, and* `pow()` *when possible.*

```
// typedef InstData: float or double
// sqFactor: $\frac{4}{\sigma^2} * (2^{(1/\omega)} - 1)$
// sqDist: $||x - y||^2$
InstData evaluatePuK(InstData sqDist)
   InstData sqInter = sqFactor * sqDist;
   if ($\omega$ == 1) {
      result = 1.0 / (1.0 + sqInter);
   } else {
      result = 1.0 / pow(1.0 + sqInter, $\omega$);
   }
   return result;
```

TABLE 13. *Linux. The* `gprof` *results show that* Baseline *Weka's three cached kernel functions take 75% of compute time on average, and are therefore excellent candidates for optimization.*

Testcase	**CachedKernel:: dotProd**	**CachedKernel:: eval**	**Puk:: evaluate**
airfoil	17.72	26.58	5.06
autoPrice	0	100.01	0
bodyfat	50	0	0
concrete	8.93	32.15	7.14
cpu	0	0	0
fishcatch	100.01	0	0
housing	38.89	15.28	11.11
wine red	36.25	31.67	13.33
wine white	49.87	24.79	9.33
pole	65.52	16.1	4.6
commsA-TP	120.5	19.28	8.43
commsA-Lat	55.32	6.38	4.26
commsB-TP	63.42	9.76	7.32
commsB-Lat	62.51	12.5	12.5
commsC-TP	54.55	0	9.09
commsC-Lat	46.52	23.72	3.72
commsD-TP	59.5	17.72	8.23
commsD-Lat	54.77	22.03	3.57
commsD-BER	60.56	8.26	6.42
Average	**49.7**	**19.3**	**6.0**

Table 13 shows the profile results for all of the datasets, in the original *Baseline* Weka code. On average, 75% of runtime is spent in the kernel (Puk and CachedKernel), computing dot products and the kernel values. `evaluate()` is the top-level function, corresponding to `evalKernel()` in Table 9. Despite the fact that approximately 90% of all calls hit the cache, and thus only 10% computes new dot products and kernel values, these computations still require 69% of runtime (49.7% and 19.3% respectively), suggesting a possible inefficiency in the way Weka caches the data. (We also tested the effect of removing caching of the kernel values entirely from *Baseline* Weka; *Baseline* runtimes increase by approximately 600%.)

Given that such a large majority of compute time is spent in these two functions, we made these significant changes:

- We optimized how Weka caches the kernel values. *Baseline* Weka relies on a least-recently-used cache, and moves items in the cache around with slow `memcpy` routines. In *Baseline* Weka, the default cache size is 250,007 elements times 4 slots per entry, storing a `double` for the kernel value and a `long` for the key. In our optimized code, we use a fixed-size $n \times n$ triangular matrix (n being the maximum number of instances), and thus dramatically reducing both total memory and all the management overhead.
- We added caching of dot product values. We first precompute all of the dot products, allowing the compiler to optimize the loop constructs. We also increment pointers to the array of values, thus making dereferencing faster. Pseudocode for the original Weka construct is in Table 14; our optimizations are in Table 15. The memory cost is an additional $n \times n$ triangular matrix.
- 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.*

```
// 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[$i$]
                  $\times$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.*

```
// 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])
      for ($y = 0$; $y \leq x$; ++$y$)
         InstData result = 0
         InstData* yp = &(inst[$y$].attr[0])
         for ($j$=0; $j < a$; ++$j$, ++xp, ++yp)
            result += (*xp) $\times$ (*yp);
         dpM[$x, y$] = result
```

```
// $x$: index of instance 1
// $y$: index of instance 2
InstData dotProd( int $x$, int $y$ )
   $x > y$ ? return dpM[$x, y$] :
            return dpM[$y, x$]
```

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

```
# for (size_t j = 0; j != 8192
#  .bs
.LDW642:
   li r21, 4096
   addi r25, sp, 60
   mr r24, r31
.L7635:
#  .bs
.LDW742:
   li r19, 0
   li r20, 512
   mr r22, r25
   mr r23, r24
.L7643:
   lwz r4, 4(r23)
   mr r3, r22
   bl __std_Vector_iterator
   lwz r4, 4(r23)
   slwi r12, r19, 2
   mr r3, r22
   add r27, r27, r12
   bl __std_Vector_iterator
...
   subic. r20, r20, 1
   addi r7, r19, 7
   slwi r12, r7, 2
   add r27, r27, r12
   addi r19, r19, 8
   bne .L7643
```

TABLE 17. *This C code generates the assembly in Table 16 and Table 18.*

```
for (size_t j = 0; j != 8192; ++j)
   for (size_t i = 0; i != SIZE; ++i)
      sum += w[i];
```

TABLE 18. *Assembly generated by GHS compiler for bbn::vector is extremely simple and involves no function calls.*

```
# for (size_t j = 0; j != 8192
#  .bs
.LDW532:
   li r6, 4096
.L15251:
#  .bs
.LDW632:
#  .es
.LDW732:
   subic. r6, r6, 1
   bne .L15251
# .es
.LDW832:
```

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.

TABLE 19. *bbn::vector is significantly faster than std::vector on our PPC440 platform.*

Time (ms)	std::vector	bbn::vector	Speedup
AutoPrice	114	19	6.0x
bodyfat	258	62	4.2x
cpu	115	7	16.4x
fishcatch	138	17	8.1x
housing	564	73	7.8x

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 ($\frac{n^2}{2}$) 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::triangle 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.

TABLE 20. *Each ablation trial removes an optimization from the* Mixed *version.* Baseline *is the true original Weka code;* Reference *is the most similar version that worked on the PPC440 platform.*

	Baseline	*Reference*	**double**	**float**	**int**	**Mixed int+float**	**except**	**pow+ sqrt**	**vector+ triangle**	**dotprod-x[i]**	**inline**
Representation	Double	Double	Double	Float	Integer	Int+Float	Int+Float	Int+Float	Int+Float	Int+Float	Int+Float
except	Enabled	Enabled	Disabled	Disabled	Disabled	Disabled	Enabled	Disabled	Disabled	Disabled	Disabled
pow+sqrt	Call	Call	Eliminate	Eliminate	Eliminate	Eliminate	Eliminate	Call	Eliminate	Eliminate	Eliminate
vector+triangle	std::vector	std::vector	bbn::vector	bbn::vector	bbn::vector	bbn::vector	bbn::vector	bbn::vector	std::vector	bbn::vector	bbn::vector
dotprod-x[i]	Compute	Compute	Cache	Cache	Cache	Cache	Cache	Cache	Cache	Compute	Cache
inline	No inlining	No inlining	Inlining	Inlining	Inlining	Inlining	Inlining	Inlining	Inlining	Inlining	No inlining

TABLE 21. *ARMv7.* Mixed int+float *is better than other numerical representations, per Section V-B. Disabling exceptions does not affect ARMv7, 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 22 on PPC440. Bold is the best item.*

Time (ms)	*Baseline*	*Reference*	**double**	**float**	**integer**	*Mixed*	**except**	**pow+sqrt**	**vec+tri**	**dotprod-x[i]**	**inline**	**exc+vec+tri**	**exc+dp**
airfoil	11,663	3,186	3,186	**848**	1,581	954	916	923	1,017	895	1,259	1,013	892
AutoPrice	286	98	89	60	37	12	14	16	11	12	15	**11**	12
bodyfat	570	125	126	91	**19**	30	31	36	29	31	37	29	31
concrete	11,180	916	1,357	649	**401**	511	548	754	511	506	615	513	624
cpu	612	143	331	**48**	52	66	68	58	64	67	161	63	66
fishcatch	453	183	216	26	69	31	21	86	**19**	20	24	19	20
housing	4,754	393	744	219	205	113	115	166	191	113	130	**109**	113
wine red	22,695	6,321	4,483	1,657	**1,170**	1,216	1,245	1,514	1,174	1,279	1,544	1,290	1,281
wine white	138,123	17,365	6,264	2,848	4,626	**1,938**	1,977	3,322	2,047	2,098	2,589	2,045	1,984
pole		14,937	7,831	3,268	**2,514**	3,236	3,298	2,849	3,370	3,360	4,317	3,357	3,350
commsA-TP	16,326	2,485	2,070	1,026	1,405	703	660	915	749	645	820	**629**	643
commsA-Lat	15,920	1,813	2,375	**794**	972	908	871	971	956	850	947	834	1,128
commsB-TP	10,184	1,040	766	472	**358**	363	387	545	366	364	407	361	485
commsB-Lat	9,016	992	997	373	461	364	373	**291**	365	388	418	367	484
commsC-TP	2,141	357	359	134	**51**	127	131	129	125	128	145	126	128
commsC-Lat	1,939	265	263	153	**127**	198	202	145	194	198	334	194	198
commsD-TP	48,939	7,667	6,119	3,379	3,014	**1,601**	1,646	1,794	1,646	1,667	1,876	1,638	1,665
commsD-Lat	32,850	7,439	4,490	2,134	**1,627**	2,488	2,543	1,990	2,588	2,675	3,432	2,694	2,665
commsD-BER	41,827	7,481	6,516	3,308	**1,465**	1,692	1,736	1,522	1,719	1,755	1,964	1,707	1,751

NormRMSE	*Baseline*	*Reference*	**double**	**float**	**integer**	*Mixed*	**except**	**pow+sqrt**	**vec+tri**	**dotprod-x[i]**	**inline**	**exc+vec+tri**	**exc+dp**
airfoil	0.448	**0.398**	0.409	0.435	0.425	0.537	0.537	0.518	0.537	0.537	0.537	0.537	0.537
AutoPrice	0.125	0.118	**0.114**	0.116	0.116	0.207	0.207	0.300	0.207	0.207	0.207	0.207	0.207
bodyfat	**0.047**	0.048	0.048	0.049	0.054	0.067	0.067	0.055	0.067	0.067	0.067	0.067	0.067
concrete	**0.251**	0.271	0.276	0.290	0.301	0.291	0.291	0.260	0.291	0.291	0.291	0.291	0.291
cpu	0.151	0.187	**0.148**	0.189	0.160	0.260	0.260	0.188	0.260	0.260	0.260	0.260	0.260
fishcatch	**0.066**	0.068	0.067	0.080	0.090	0.123	0.123	0.090	0.123	0.123	0.123	0.123	0.123
housing	**0.146**	0.172	0.150	0.164	0.164	0.180	0.180	0.165	0.180	0.180	0.180	0.180	0.180
wine red	0.778	**0.626**	0.714	0.672	0.666	0.793	0.793	0.731	0.793	0.793	0.793	0.793	0.793
wine white	1.016	0.731	0.730	0.771	0.790	0.846	0.846	**0.706**	0.846	0.846	0.846	0.846	0.846
pole	**0.000**	0.293	0.214	0.240	0.240	0.205	0.205	0.222	0.205	0.205	0.205	0.205	0.205
commsA-TP	1.103	0.873	0.870	0.855	0.854	**0.851**	**0.851**	0.881	**0.851**	**0.851**	**0.851**	**0.851**	**0.851**
commsA-Lat	1.102	0.878	0.863	0.882	0.887	**0.844**	**0.844**	0.882	**0.844**	**0.844**	**0.844**	**0.844**	**0.844**
commsB-TP	**0.003**	0.047	0.047	0.019	0.013	0.020	0.020	0.018	0.020	0.020	0.020	0.020	0.020
commsB-Lat	**0.003**	0.028	0.028	0.028	0.036	0.021	0.021	0.048	0.021	0.021	0.021	0.021	0.021
commsC-TP	**0.050**	0.053	0.053	0.060	0.226	0.062	0.062	0.062	0.062	0.062	0.062	0.062	0.062
commsC-Lat	**0.002**	0.017	0.017	0.030	0.034	0.028	0.028	0.025	0.028	0.028	0.028	0.028	0.028
commsD-TP	**0.098**	0.152	0.160	0.314	0.254	0.172	0.172	0.212	0.172	0.172	0.172	0.172	0.172
commsD-Lat	**0.060**	0.136	0.192	0.467	0.169	0.272	0.272	0.527	0.272	0.272	0.272	0.272	0.272
commsD-BER	**0.052**	0.180	0.138	0.205	0.208	0.174	0.174	0.224	0.174	0.174	0.174	0.174	0.174

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 the presented optimizations. *Reference* removes all of the presented 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 each cell indicates the setting for that capability in that trial.

Table 21 and Table 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.*

Time (ms)	***Reference***	**double**	**float**	**integer**	***Mixed***	**except**	**pow+sqrt**	**vec+tri**	**dotprod-x[i]**	**inline**	**exc+vec+tri**	**exc+dp[i]**
airfoil	11,950	7,115	**1,982**	4,396	2,787	3,409	2,391	2,332	2,784	4,019	6,374	3,393
AutoPrice	475	216	72	114	50	59	58	**44**	51	61	111	59
bodyfat	531	301	141	**62**	86	100	80	72	86	106	191	100
concrete	3,650	5,610	1,209	1,103	604	767	1,064	**504**	604	908	1,524	765
cpu	603	911	111	113	135	154	**57**	112	127	156	295	153
fishcatch	830	469	62	230	114	128	**56**	103	114	131	227	128
housing	1,719	1,333	363	610	282	328	**209**	256	282	344	623	326
wine red	15,538	5,634	3,292	3,222	2,610	3,236	2,635	**2,195**	2,608	3,810	6,121	3,220
wine white	13,360	5,926	2,474	4,461	2,306	2,652	2,822	**1,985**	2,301	2,910	4,959	2,637
pole	11,518	8,605	2,109	**1,588**	2,624	3,056	2,218	2,269	2,621	3,404	5,715	3,042
commsA-TP	6,790	5,645	1,875	3,655	2,082	2,335	**1,557**	1,866	2,081	2,481	4,376	2,328
commsA-Lat	9,539	5,054	2,480	2,560	1,990	2,233	2,243	**1,760**	1,989	2,373	4,154	2,227
commsB-TP	6,389	1,948	1,088	1,024	753	854	1,401	**684**	754	913	1,623	853
commsB-Lat	2,629	3,347	1,371	1,109	756	903	881	**667**	755	1,013	1,786	902
commsC-TP	1,459	788	330	**159**	437	465	327	393	420	495	876	465
commsC-Lat	1,125	621	**328**	425	423	471	381	405	416	497	917	469

NormRMSE	***Reference***	**double**	**float**	**integer**	***Mixed***	**except**	**pow+sqrt**	**vec+tri**	**dotprod-x[i]**	**inline**	**exc+vec+tri**	**exc+dp[i]**
airfoil	**0.397**	0.401	0.434	0.425	0.513	0.513	0.444	0.513	0.513	0.513	0.513	0.513
AutoPrice	0.118	**0.114**	0.123	0.116	0.129	0.129	0.126	0.129	0.129	0.129	0.129	0.129
bodyfat	**0.048**	**0.048**	0.049	0.054	0.058	0.058	0.055	0.058	0.058	0.058	0.058	0.058
concrete	**0.271**	0.294	0.304	0.301	0.344	0.344	0.310	0.344	0.344	0.344	0.344	0.344
cpu	0.187	**0.148**	0.206	0.160	0.189	0.189	0.232	0.189	0.189	0.189	0.189	0.189
fishcatch	0.067	**0.066**	0.090	0.090	0.093	0.093	0.093	0.093	0.093	0.093	0.093	0.093
housing	0.173	**0.151**	0.213	0.164	0.179	0.179	0.225	0.179	0.179	0.179	0.179	0.179
wine red	**0.630**	0.634	0.665	0.701	0.680	0.680	0.679	0.680	0.680	0.680	0.680	0.680
wine white	0.761	0.690	0.622	**0.578**	0.632	0.632	0.619	0.632	0.632	0.632	0.632	0.632
pole	0.266	0.212	0.234	0.229	**0.205**	**0.205**	0.234	**0.205**	**0.205**	**0.205**	**0.205**	**0.205**
commsA-TP	0.874	0.858	0.872	**0.854**	0.877	0.877	0.887	0.877	0.877	0.877	0.877	0.877
commsA-Lat	0.868	0.873	0.884	0.887	0.878	0.878	**0.864**	0.878	0.878	0.878	0.878	0.878
commsB-TP	0.015	**0.009**	0.117	0.013	0.026	0.026	0.015	0.026	0.026	0.026	0.026	0.026
commsB-Lat	0.031	0.025	0.036	0.036	0.040	0.040	**0.021**	0.040	0.040	0.040	0.040	0.040
commsC-TP	0.053	0.053	**0.052**	0.226	0.053	0.053	0.054	0.053	0.053	0.053	0.053	0.053
commsC-Lat	0.017	0.017	0.035	0.034	0.018	0.018	**0.016**	0.018	0.018	0.018	0.018	0.018

- 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 and kernel implementation (Section V-C) directly impact the number of computations required to converge.

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:

$$\text{NormRMSE} = \frac{\sqrt{\frac{1}{n}\sum_i^n(\hat{x}_i - x_i)^2)}}{\sqrt{\frac{1}{n}\sum_i^n(\hat{x}_i - \mu)^2)}}$$

where n is the number of instances in the data, μ is the average target value, $\hat{x}_i$ 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

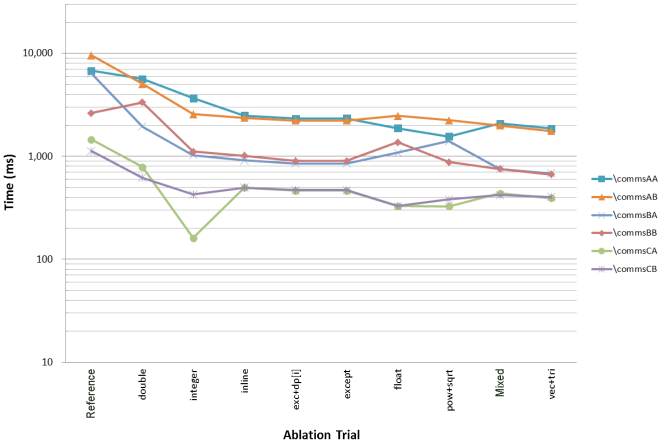

Fig. 1. *PPC440, communications datasets. While ablation trials tend to yield a similar performance improvement across datasets, characteristics of the dataset can cause varied results.*

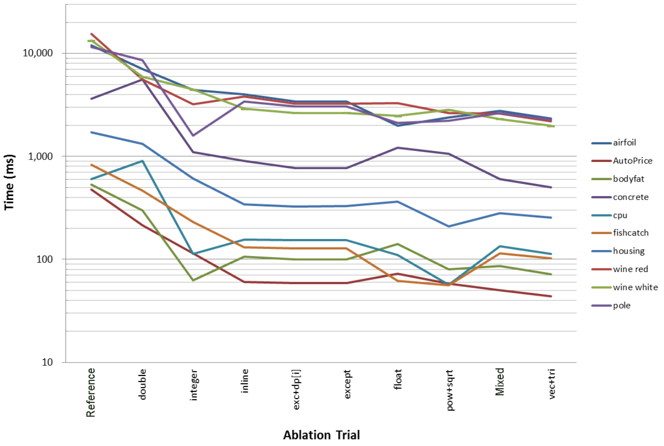

Fig. 2. *PPC440, non-communications datasets. While ablation trials tend to yield a similar performance improvement across datasets, characteristics of the dataset can cause varied results.*

TABLE 24. *By inlining functions, We reduce 1 billion function calls from* Baseline *to 20 million calls in* Mixed.

	SMOset::			**Puk::**			**SVMOptimizer::**		
	Baseline	**Inline**	***Mixed***	***Baseline***	**Inline**	***Mixed***	***Baseline***	**Inline**	***Mixed***
airfoil	26,339,476	17,116,905	0	35,337,590	17,038,926	1,130,258	4,581,774	2,253,452	108,533
autoPrice	551,778	137,754	0	632,078	130,113	12,720	251,661	51,009	5,634
bodyfat	777,743	383,622	0	1,004,410	370,659	31,878	339,330	131,205	12,779
concrete	16,689,819	8,256,954	0	20,883,175	8,193,441	530,965	6,299,215	2,307,960	76,740
cpu	1,152,233	802,530	0	1,279,303	776,187	21,945	508,734	1,435,242	16,633
fishcatch	932,400	239,583	0	988,402	228,270	12,561	439,098	6,360,099	6,702
housing	6,996,474	1,494,210	0	7,978,030	1,468,386	128,271	3,221,846	4,240,755	18,082
wine red	24,184,029	41,527,128	0	34,296,581	21,289,674	3,966,338	8,696,277	7,303,671	137,899
wine white	54,699,503	21,372,672	0	86,356,027	40,295,430	1,279,201	10,215,449	2,845,732	133,553
pole	68,449,393	40,393,032	0	100,076,035	41,393,729	3,966,339	20,873,528	3,730,674	147,865
commsA-TP	11,413,877	7,783,980	0	27,999,242	7,721,310	662,977	4,685,919	1,525,365	61,716
commsA-Lat	10,454,653	9,507,624	0	29,390,957	9,437,436	662,976	4,233,701	1,388,763	62,992
commsB-TP	10,488,652	4,146,540	0	12,742,813	4,101,445	336,611	4,701,337	607,110	41,439
commsB-Lat	8,282,091	4,668,729	0	10,846,363	4,615,512	336,610	3,626,289	966,129	77,973
commsC-TP	1,958,177	1,440,888	0	2,527,100	1,408,203	71,631	922,243	1,526,424	39,384
commsC-Lat	1,765,778	2,200,104	0	2,282,637	2,146,260	71,631	781,643	335,535	51,225
commsD-TP	56,020,056	19,023,933	0	67,722,573	18,924,544	1,500,779	25,203,021	100,617	120,792
commsD-Lat	26,854,672	51,665,469	0	38,531,625	51,444,433	1,500,778	10,496,839	538,074	663,993
commsD-BER	39,976,688	19,672,197	0	52,055,883	19,573,639	1,500,778	17,416,322	10,352,839	91,716
TOTAL	**367,987,492**	**251,833,854**	**0**	**532,930,824**	**250,557,597**	**17,725,247**	**127,494,226**	**48,000,655**	**1,875,650**
compare to inline			0.0%			7.1%			3.9%
compare to baseline			0.0%			3.3%			1.5%

TABLE 25. *Inlining functions cuts runtime to about 80%. This table summarizes data in Table 21 and Table 22.*

	Average Time (ms)			
Platform (dataset)	**Minimal Inline**	**Puk**	**SMOset+ SVMOptimizer**	**All Inlined**
PPC440 (comms only)	1,295	1,251	1,168	1,073
PPC440 (all datasets)	1,476	1,401	1,236	1,127
ARMv7 (comms only)	1,149	1,131	978	957
ARMv7 (all datasets)	1,107	1,074	901	890
	Average NormRMSE			
PPC440 (comms only)	0.315	0.315	0.315	0.315
PPC440 (all datasets)	0.307	0.307	0.307	0.307
ARMv7 (comms only)	0.272	0.272	0.272	0.272
ARMv7 (all datasets)	0.313	0.313	0.313	0.313

TABLE 26. *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.*

	Average Time (ms)					
Platform (dataset)	***Baseline***	***Reference***	**double**	**float**	**integer**	***Mixed***
PPC440 (comms only)		4,464	2,901	1,246	1,488	1,073
PPC440 (all datasets)		5,192	3,345	1,205	1,552	1,127
ARMv7 (comms only)	19,905	3,282	2,662	1,308	1,053	938
ARMv7 (all datasets)	20,659	4,016	2,567	1,244	1,098	1,012
	Average NormRMSE					
Platform (dataset)	***Baseline***	***Reference***	**double**	**float**	**integer**	***Mixed***
PPC440 (comms only)		0.310	0.306	0.333	0.342	0.315
PPC440 (all datasets)		0.299	0.287	0.309	0.304	0.307
ARMv7 (comms only)	0.275	0.263	0.263	0.318	0.298	0.272
ARMv7 (all datasets)	0.268	0.265	0.258	0.303	0.285	0.308

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.)

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*.

TABLE 27. *PPC440. The number of changed examples correlates highly with the runtime (ms) of Table 22.*

Test case	*Reference*	double	float	integer	*Mixed*
airfoil	8,712	8,961	4,734	11,412	4,800
AutoPrice	2,688	2,268	1,293	5,568	945
bodyfat	1,989	1,989	1,464	1,491	1,008
concrete	3,870	8,790	3,576	5,661	2,103
cpu	2,577	10,140	1,392	3,579	1,740
fishcatch	5,145	4,926	1,083	11,715	1,602
housing	3,237	4,449	2,163	7,140	1,950
wine red	9,537	6,471	4,350	8,454	4,140
wine white	8,148	5,709	5,250	11,889	5,529
pole	6,582	8,331	4,158	4,380	5,475
commsA-TP	5,907	7,737	4,761	15,105	5,343
commsA-Lat	7,803	7,044	5,790	10,773	5,184
commsB-TP	6,894	3,987	3,606	7,410	3,027
commsB-Lat	3,099	6,825	3,174	7,140	2,655
commsC-TP	3,357	3,306	2,379	2,628	2,772
commsC-Lat	2,988	2,988	2,682	7,773	3,420
Correlation (comms only)	**96.8%**	**96.5%**	**95.7%**	**93.7%**	**93.9%**
Correlation (all datasets)	**93.8%**	**68.0%**	**90.8%**	**71.7%**	**91.4%**

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 the 64-bit `double` representation tends to have the lowest NormRMSE, its very high runtime does not justify the small gain. Moreover, note that in 4 of the 12 PPC440 cases and 7 of the 19 ARMv7 cases, one of the other representations yields lower error. The *Mixed* representation has lower error than *Float* or *Integer* for the communications datasets, while the *Integer* representation does better across all the datasets.

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 incurrs 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()` for float is 0.129μs, `sqrt()` for double is 0.455μs. 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.

TABLE 28. *On average, removing* `sqrt()` *and* `pow()` *improves runtimes on PPC440, but specific cases may improve or worsen. Similarly, NormRMSE may increase or decrease because removing the unnecessary* `sqrt()` *avoids incorrect floating point errors on the processor. This table summarizes data in Table 21 and Table 22.*

Average Time (ms)				
Platform (dataset)	**pow+sqrt**	**sqrt**	**pow**	***Mixed***
PPC440 (comms only)	1,132	1,066	1,132	1,073
PPC440 (all datasets)	1,149	1,122	1,148	1,127
ARMv7 (comms only)	922	923	929	938
ARMv7 (all datasets)	1,033	1,035	1,002	1,012
Average NormRMSE				
Platform (dataset)	**pow+sqrt**	**sqrt**	**pow**	***Mixed***
PPC440 (comms only)	0.310	0.310	0.315	0.315
PPC440 (all datasets)	0.305	0.305	0.307	0.307
ARMv7 (comms only)	0.320	0.320	0.272	0.272
ARMv7 (all datasets)	0.317	0.317	0.308	0.308

TABLE 29. *PPC440. The number of changed examples is 96% correlated to the runtime. Ratio is (pow+sqrt /* Mixed*).* Table 27 *presents similar results.*

	Runtime (ms)			**Optimize count**		
Test name	**pow+sqrt**	***Mixed***	**Ratio**	**pow+sqrt**	***Mixed***	**Ratio**
cpu	57	135	42%	855	1,740	49%
fishcatch	56	114	49%	1,026	1,602	64%
housing	209	283	74%	1,602	1,950	82%
commsC-TP	326	435	75%	2,400	2,772	87%
commsA-TP	1,560	2,083	75%	4,434	5,343	83%
pole	2,221	2,625	85%	5,115	5,475	93%
airfoil	2,393	2,787	86%	5,199	4,800	108%
commsC-Lat	380	422	90%	3,168	3,420	93%
bodyfat	80	86	93%	993	1,008	99%
wine red	2,637	2,610	101%	4,266	4,140	103%
commsA-Lat	2,247	1,991	113%	5,643	5,184	109%
AutoPrice	58	50	116%	1,050	945	111%
commsB-Lat	882	756	117%	3,336	2,655	126%
wine white	2,826	2,307	123%	5,997	5,529	108%
concrete	1,064	604	176%	3,018	2,103	144%
commsB-TP	1,403	753	186%	4,776	3,027	158%

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 commsC-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.)*

Average Time (ms)			
Platform (dataset)	**dotprod-x[i]**	**dotprod-*x++**	***Mixed***
PPC440 (comms only)	1,069	1,065	1,073
PPC440 (all datasets)	1,125	1,124	1,127
ARMv7 (comms only)	963	955	938
ARMv7 (all datasets)	1,048	1,039	1,012
Average NormRMSE			
Platform (dataset)	**dotprod-x[i]**	**dotprod-*x++**	***Mixed***
PPC440 (comms only)	0.315	0.315	0.315
PPC440 (all datasets)	0.307	0.307	0.307
ARMv7 (comms only)	0.272	0.272	0.272
ARMv7 (all datasets)	0.308	0.308	0.308

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.*

Average Time (ms)						
Platform (dataset)	**exc+vec+ tri+dp[i]**	**exc+ vec+tri**	**exc+ dp[i]**	**exc+ dp***	**except**	***Mixed***
PPC440 (comms only)	2,328	2,289	1,207	1,212	1,210	1,073
PPC440 (all datasets)	2,508	2,492	1,317	1,322	1,322	1,127
ARMv7 (comms only)		950		1,016	950	938
ARMv7 (all datasets)		1,061		1,064	1,030	1,012
Average NormRMSE						
PPC440 (comms only)	0.315	0.315	0.315	0.315	0.315	0.315
PPC440 (all datasets)	0.307	0.307	0.307	0.307	0.307	0.307
ARMv7 (comms only)	0.272	0.272	0.272	0.272	0.272	0.272
ARMv7 (all datasets)	0.308	0.308	0.308	0.308	0.308	0.308

cost. Column *dotprod-x[i]* in Table 21 and Table 22 shows the timing and NormRMSE results when we do not cache the dot product results. Column *Mixed* shows the results when we do cache the dot products. Table 30 shows the average results for the two platforms and groups of datasets. There is no significant difference in the timing results between these two columns.

Table 30 also shows the effect of using pointer math, by breaking out the dot product results into *dotprod-x[i]*, which refers to dot product computations similar to Table 14, and *dotprod-*x++*, which refers to computations similar to Table 15. Note again that only *Mixed* caches the results of the dot product computations.

When we take into account the number of attributes or instances in the data, there is a slight performance improvement; that is, when there are more computations to make, caching the dot product results can reduce overall compute time. However the cost to memory is probably not worth the small timing improvements.

E. Removing exceptions

Each of the columns *exc* in Table 21 and Table 22 show the timing and NormRMSE results when exceptions are enabled, while column *Mixed* shows the results when they are disabled. Table 31 shows the average results by platform and dataset group.

Disabling exceptions speeds up execution by about 15% on PPC440, and 1% on ARMv7.

The more interesting (unexpected) result is the effect of exceptions on our custom vector and matrix structures. When we disable exceptions, `std::vector` becomes more efficient than our custom `bbn::vector` (Section IV-F and V-F).

TABLE 32. *Unexpectedly, our custom vector and matrix modules performed* better *than the* Mixed *when exceptions are disabled. When we enable exceptions,* `std` *modules (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.*

Average Time (ms)					
Platform (dataset)	**exc+vec+tri**	**vec+tri**	**triangle**	**vector**	***Mixed***
PPC440 (comms only)	2,289	962	966	1,053	1,073
PPC440 (all datasets)	2,492	978	990	1,100	1,127
ARMv7 (comms only)	950	994	967	890	938
ARMv7 (all datasets)	1,061	1,081	1,068	978	1,012
Average NormRMSE					
Platform (dataset)	**exc+vec+tri**	**vec+tri**	**triangle**	**vector**	***Mixed***
PPC440 (comms only)	0.315	0.315	0.315	0.315	0.315
PPC440 (all datasets)	0.307	0.307	0.307	0.307	0.307
ARMv7 (comms only)	0.272	0.272	0.272	0.272	0.272
ARMv7 (all datasets)	0.308	0.308	0.308	0.308	0.308

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:

Ablation version	Exceptions	Vectors	Triangles
exc+vec+tri	Enabled	std::vector	std::vector
vec+tri	Disabled	std::vector	std::vector
triangle	Disabled	bbn::vector	std::vector
vector	Disabled	std::vector	bbn::vector
except	Enabled	bbn::vector	bbn::vector
Mixed	Disabled	bbn::vector	bbn::vector

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.

TABLE 33. *Our optimized version of Weka requires about 5% of the original runtimes. This table summarizes data in Table 21 and Table 22.*

Platform (dataset)	**Compared to *Baseline***	**Compared to *Reference***
PPC440 (comms only)		27.7%
PPC440 (all datasets)		11.7%
ARMv7 (comms only)	5.5%	37.4%
ARMv7 (all datasets)	6.1%	32.1%

- 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

[1] Applied Micro Circuits Corporation. *PPC440 (PPC440G5) Processor User's Manual*, 1.0 edition, August 2011. http://www.amcc.com.

[2] K. Bache and M. Lichman. UCI machine learning repository, 2013.

[3] C.-C. Chang and C.-J. Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011. Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm.

[4] P. Cortez, A. Cerdeira, F. Almeida, T. Matos, and J. Reis. Modeling wine preferences by data mining from physicochemical properties. *Decision Support Systems, Elsevier*, 47:547–553, 2009.

[5] B. Dawes, D. Abrahams, and R. Rivera. Boost C++ libraries, 2014. http://www.boost.org/.

[6] E. Frank, M. Hall, P. Reutemann, and L. Trigg. Weka java 3.7.9, February 2013. http://grepcode.com/snapshot/repo1.maven.org/maven2/nz.ac.waikato.cms.weka/weka-dev/3.7.9/.

[7] K. Z. Haigh. AI technologies for tactical edge networks. In *MobiHoc 2011 Workshop on Tactical Mobile Ad Hoc Networking*, Paris, France, 2011. (New York, NY: ACM Press). Keynote address. http://www.cs.cmu.edu/~khaigh/papers/Haigh-MobiHoc2011.pdf.

[8] K. Z. Haigh, O. Olofinboba, and C. Y. Tang. Designing an implementable user-oriented objective function for MANETs. In *IEEE International Conference On Networking, Sensing and Control*, pages 693–698, London, U.K., April 2007. (New York, NY: IEEE Press). http://www.cs.cmu.edu/~khaigh/papers/Haigh07-ICNSC.pdf.

[9] K. Z. Haigh, S. Varadarajan, and C. Y. Tang. Automatic learning-based MANET cross-layer parameter configuration. In *Workshop on Wireless Ad hoc and Sensor Networks (WWASN2006)*, Lisbon, Portugal, 2006. (New York, NY: ACM Press). http://www.cs.cmu.edu/~khaigh/papers/haigh06a-configuration.pdf.

[10] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The WEKA data mining software: An update. *SIGKDD Explorations*, 11(1), 2009. http://www.cs.waikato.ac.nz/~eibe/pubs/weka_update.pdf.

[11] C. Igel, V. Heidrich-Meisner, and T. Glasmachers. Shark. *Journal of Machine Learning Research*, 9:993–996, 2008. http://image.diku.dk/shark/.

[12] O. Ivanciuc. Support vector machine software, 2005. http://www.support-vector-machines.org/SVM_soft.html.

[13] T. Joachims. Making large-scale SVM learning practical. In B. Schölkopf, C. Burges, and A. Smola, editors, *Advances in Kernel Methods - Support Vector Learning*, chapter 11, pages 169–184. (Cambridge, MA: MIT Press), 1999.

[14] T. Joachims. Svm^{light}, 2008. http://svmlight.joachims.org/, version 6.02.

[15] D. E. King. Dlib-ml: A machine learning toolkit. *Journal of Machine Learning Research*, 10:1755–1758, 2009. http://dlib.net/ml.html.

[16] T. Kudoh. TinySVM, 2002. http://chasen.org/~taku/software/TinySVM/.

[17] J. C. Platt. Fast training of Support Vector Machines using Sequential Minimal Optimization. In B. Schoelkopf, C. Burges, and A. Smola, editors, *Advances in Kernel Methods - Support Vector Learning*. (Cambridge, MA: MIT Press), 1998. http://research.microsoft.com/en-us/um/people/jplatt/smo-book.pdf.

[18] A. Purtell and S. Schwab. Weka C++, 2013. http://wekacpp.sourceforge.net/.

[19] S. K. Shevade, S. S. Keerthi, C. Bhattacharyya, and K. R. K. Murthy. Improvements to the smo algorithm for svm regression. *IEEE Transactions on Neural Networks*, 11(5), Sept 2000. http://www.keerthis.com/smoreg_ieee_shevade_00.pdf.

[20] E. Teran. Fixed point header, 2008. http://www.codef00.com/code/Fixed.h.

[21] B. Üstün, W. J. Melssen, and L. M. C. Buydens. Facilitating the application of Support Vector Regression by using a universal Pearson VII function based kernel. *Chemometrics and Intelligent Laboratory Systems*, 81(1):29–40, March 2006. http://www.sciencedirect.com/science/article/pii/S0169743905001474.

[22] V. N. Vapnik. *The Nature of Statistical Learning Theory*. (New York, NY: Springer), 1995.

[23] S. M. Weiss and N. Indurkhya. Rule-based machine learning methods for functional prediction. *Journal of Artificial Intelligence Research*, 3:383–403, 1995. Dataset from: http://www.dcc.fc.up.pt/~ltorgo/Regression/DataSets.html.

[24] Wikipedia. Support vector machine, 2014. http://en.wikipedia.org/wiki/Support_vector_machine, downloaded October 2014.

[25] I.-C. Yeh. Modeling of strength of high performance concrete using artificial neural networks. *Cement and Concrete Research*, 28:1797–1808, 1998.