Tuning Machine-Learning Algorithms for Battery-Operated Portable Devices

Ziheng Lin[1,⋆], Yan Gu[2], and Samarjit Chakraborty[3]

[1] Department of Computer Science, National University of Singapore, Singapore
linzihen@comp.nus.edu.sg
[2] Continental Automotive Singapore Pte Ltd, Singapore
Yan.Gu@continental-corporation.com
[3] Institute for Real-time Computer Systems, TU Munich, Germany
samarjit@tum.de

Abstract. Machine learning algorithms in various forms are now increasingly being used on a variety of portable devices, starting from cell phones to PDAs. They often form a part of standard applications (e.g. for grammar-checking in email clients) that run on these devices and occupy a significant fraction of processor and memory bandwidth. However, most of the research within the machine learning community has ignored issues like memory usage and power consumption of processors running these algorithms. In this paper we investigate how machine learned models can be developed in a power-aware manner for deployment on resource-constrained portable devices. We show that by tolerating a small loss in accuracy, it is possible to dramatically improve the energy consumption and data cache behavior of these algorithms. More specifically, we explore a typical sequential labeling problem of *part-of-speech tagging* in natural language processing and show that a power-aware design can achieve up to 50% reduction in power consumption, trading off a minimal decrease in tagging accuracy of 3%.

Keywords: Low-power Machine Learned Models, Part-of-speech Tagging, Mobile Machine Learning Applications, Power-aware Design.

1 Introduction

While mobile devices are already ubiquitous, in the near future such devices are likely to become a dominant computing platform. Since battery-life is a major design concern for these devices, applications currently designed for the desktop need to be redesigned to allow the operating system or human operator to control the application's power consumption. While general methods do exist to regulate the underlying processor's frequency and voltage without detailed knowledge of the applications, we may be able to achieve even better energy savings when we are informed of the specifics of the application domain.

Take the case of email on a portable device. In the near future, portable email clients will gain the capability to correct grammatical errors and recognize names of contacts, products and companies and link them with appropriate information (e.g., address

⋆ This work was partially supported by a National Research Foundation grant "Interactive Media Search" (grant # R-252-000-325-279).

P.-J. Cheng et al. (Eds.): AIRS 2010, LNCS 6458, pp. 502–513, 2010.

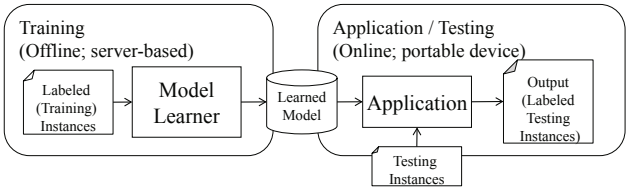


Fig. 1. Machine learning application on portable platforms

book contact information, Wikipedia entries). These nascent functions require the email client to understand the text of emails, requiring the use of natural language processing (NLP) tools. Up to now, applications requiring NLP have been largely restricted to the desktop or server platforms. What software design changes are a consequence of moving to portable, battery-powered platforms?

To answer this question, we explore one such application, namely that of applying a large-scale machine learned model. Diagnosing a patient's illness from symptoms [4], recognizing faces [3], delimiting addresses and place names in an email [8] are all sample problems where an approach of applying machine-learned model is a possible solution. While these applications have traditionally been done solely on desktops, we see a shift where portable devices are also used to perform these tasks: already, web browsers are used on PDAs and digital cameras attempt to recognize faces.

To our knowledge, implementations of such large-scale machine learned models have only concerned themselves with time (and to a much lesser extent, space) efficiency. We introduce the concept of *power-aware machine learning model application*, in which power consumption and prediction accuracy can be traded off, allowing us to choose a suitable performance level based on context. We concretize our investigation on this general model by examining the sample application of *part-of-speech tagging*, a key NLP task that underlies the grammar checking and name recognition functionality (among others) in the email client scenario. We examine how this task can be tackled using a power-sensitive adaptation of the Naïve Bayes machine learning algorithm [6]. We show simulated results using the SimpleScalar instruction set simulator [1] over different hardware configurations that exhibit a power-accuracy tradeoff and confirm this with observed voltage and current measurements on an instrumented PDA. A key contribution of our work is in designing the Naïve Bayes model to trade off accuracy for power conservation by varying the generated learned model size. Such work enables efficient tagging of the parts-of-speech of words in a sentence on mobile platforms.

We first give a general introduction to the machine learning setting and relate how part of speech tagging constitutes an instance of this setting. We discuss how the basic Naïve Bayes algorithm can be designed to make power consumption versus accuracy tradeoffs in Section 3 by selecting a suitable model size. In Section 4, we then further explore how to optimize the algorithm's data access to further reduce power consumption through the use of early termination and batching feature and problem instance computation. We end by discussing both our simulated and real-world experiments and conclude with possible extensions of this work.

2 Machine Learning and Sequence Labeling Tasks

Machine learning is a broad area of research and a subdiscipline in its own right within computer science [6]. The basic task is to predict an output $f(x)$ given inputs x, where x could be a vector of different individual *features*: $\{x_1, x_2, ..., x_n\}$. One method to do this is known as *supervised learning*, in which we are given some *training instances* where the output $f(x)$ is known, which is used as evidence to build a prediction model. While different features may have different utility in predicting the output, in most machine learning scenarios, we typically provide as many potentially useful features as possible, and relegate the task of deciding which features are actually useful to the learning software. Linear regression is an example of a learning method, in which the given examples determine the coefficients (i.e., the model) of the predicted plane. Once a model is computed, new *testing instances* (where the output $f(x)$ is not known) can be run through the model to predict an output.

Building the model (also called *training*) and applying it on unseen instances (or *testing*) are logically distinct steps. As we often want to apply a trained model repeatedly to many sets of unseen instances, we may require that testing is computationally efficient (i.e., fast) to apply, but the computational efficiency of training is not an issue (we could compute the model on a server, for example). In cases where we want to apply a machine learned model on a power-constrained device, we can simply store a precomputed model (often much more compact than the set of training instances itself) on the device and apply it to instances on-line. This scenario is illustrated in Figure 1.

The machine learning paradigm is currently used to obtain state-of-the-art performance on many problems in computer vision and NLP. In this paper, we use the *part-of-speech (POS) tagging* task and the Naïve Bayes (NB) learning algorithm as a motivating running example of how we can adapt the application of machine learned models for a power-aware environment.

Note that our solution is not limited to this task nor to this learning algorithm; we adopt this example as both the task and the learning algorithm are easy to explain. We now explain these two aspects.

Part of speech tagging. The POS tagging task is set as follows: given a text, tag each word with its proper part of speech [7]. For example, given the English sentences "`The parachute jump was spectacular`" and "`IBM and Apple stocks jump`", the tagged output would be[1]:

"`The`/article `parachute`/noun `jump`/noun `was`/verb `spectacular`/adjective",
"`IBM`/proper_noun `and`/conjunction `Apple`/proper_noun `stocks`/noun `jump`/verb".

Note that the word "jump" plays different roles of noun and verb in the two sentences respectively. As many words take on different parts of speech depending on context, it is not trivial to predict the tag of the word. Accurate POS tagging forms the basis

[1] English POS taggers generally tag not only for word class but inflections to a word, including number, tense and gender, but we ignore these issues here for a simplified presentation. See [5] for a comprehensive treatment of the standard inventory of tags.

for many NLP applications, including suggesting spelling corrections, grammar checking and locating place and person names in documents, all of which may need to be performed on power-constrained devices.

Determining a word's POS can thus be cast as a machine learning problem, where relevant contextual information are represented as features that help to choose between the possible 45 standard POS tags [5]. Note here that the predicted output is discrete rather than continuous; i.e., $f(x) \in C : \{noun, verb, ...\}$. For example, knowing the word's identity (e.g., "jump" should usually be a verb or noun), the previous word's identity (e.g., words following "the" should be nouns), or whether the word is capitalized (e.g., "Apple" (the company) versus "apple" (the fruit)) are all classes of features that hint at a word's POS tag. Some feature classes may lead to thousands of individual features (e.g., the feature class of word identity needs a feature for every possible English word), where other feature classes may simply be represented by single features (e.g., capitalization). These features are calculated for each word to be tagged, forming the feature matrix in Figure 2. The output of a learning algorithm $f(x)$ is a POS tag, such as noun or verb, selected out of a tag inventory C (where $|C| = 45$).

Naïve Bayes. Naïve Bayes (NB) is a simple machine learning algorithm that uses probability to predict the output function. NB casts the output of calculating $P(f(x)|x)$ as $\frac{P(f(x))\times P(x|f(x))}{P(x)}$ by Bayes' theorem. As x is given (it is the given testing vector), its probability is constant over all of the $P(f(x)|x)$ calculations and can be dropped, resulting in $P(f(x)|x) = P(f(x)) \times P(x|f(x))$. As x may consist of many (say n) individual features in a feature vector, NB makes a naïve (hence the name) assumption that each feature is independent of others and estimates the true joint probability $P(x|f(x))$ as $\hat{P}(x|f(x)) \approx P(x_1|f(x)) \times P(x_2|f(x)) \times ... \times P(x_n|f(x))$. During testing, NB predicts the output that has the highest estimate over all possible outputs, that is:

$$\hat{c} \approx \underset{f(x)\in C}{\operatorname{argmax}} P(f(x)) \prod_{j=1}^{n} P(x_j|f(x)) \tag{1}$$

NB does its prediction based on the stored probabilities of $P(f(x))$ and $P(x_j|f(x))$. These probabilities form the model that NB computes in training. Note that for POS tagging, storing $P(f(x))$ requires only storing 45 floating point (or fixed precision) probabilities for each of the possible POS tags, but storing $P(x_j|f(x))$ requires storing 45 probabilities for each feature. As typical large-scale machine learning employs tens of thousands of features, the storage of the second part of the model is clearly the bottleneck.

The basic Naïve Bayes classification algorithm is shown as pseudocode in Algorithm 1. For each word to be tagged, we first initialize the scores for each output tag as the unconditional probabilities $P(f(x))$ from the NB model as scores for $P(f(x)|x)$. We process each feature by first calculating the feature value x_j and then updating our score for $P(f(x)|x)$ by multiplying the current score by the conditional probability $P(x_j|f(x))$. We process each of $|x|$ features in turn, where each feature calculation refines the score values for each possible output class.

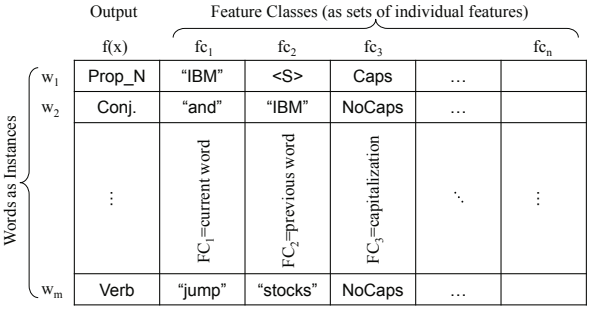


Fig. 2. Sample feature matrix for part of speech tagging for training instances

Algorithm 1. Standard Naïve Bayes model application for a single problem, where features are applied in ranked order

/* Initialize probabilities for each output class $f(x) \in C$ */
  for all $c \in C$ **do**
    $P(f(x) == c) \leftarrow P(c)$
/* Use each of the n features, in turn: */
  for all $j \in \{1, 2, ..., n\}$ **do**
    Calcuate feature x_j
    for all $c \in C$ **do**
      Update $P(f(x) == c) \leftarrow P(f(x) == c) \times f(x_j|c)$
  Assign $f(x) \leftarrow \text{argmax}\, P(f(x) == c)$

3 Impact of Varying the Model Size

In general, the more evidence we provide the learner with, the more accurate the predictions will be. However, we observe that acquiring features can be potentially expensive: given an input text we need to compute these features from an input text, and we also have to allocate space in memory to store the learned model's probabilities and then apply them at runtime. In the standard desktop environment, these space concerns are unimportant; we typically give the learner as many features as possible, as space and power are abundant.

On mobile devices, we have a more constrained environment, in which power and memory limitation creates a more complex scenario. A further complication is that there are usually (implicit) deadlines for when certain processing should be completed by. A user of a mobile device may be willing to wait a few seconds for a background process to highlight names or do grammar checking, but not if it takes over ten seconds and not if it constantly drains the device's batteries.

A key observation is that by using fewer features in the model, we can limit the model size and time and power needed in calculating features at the expense of prediction accuracy. To do this, we need to decide which features to retain and which can be left out. This is the problem of *feature selection* which has attracted much attention

in the machine learning community (see [2] for a survey). Work on feature selection is used primarily to filter out features that provide no information or which only add noise. Usually the objective function is to optimize accuracy unconditionally, whereas we wish to optimize accuracy per unit feature, to reduce model size. A simple method for selection uses the *information gain* to rank features by how well they discriminate between the output classes, which we employ here.

$$IG(Ex, f_i) = H(Ex) - H(Ex|f_i). \tag{2}$$

where Ex denotes all the training examples and $H()$ denotes entropy, which is defined as $\sum_{c=1}^{|C|} -P(c) \log P(c)$. Using this selection criteria, we can rank all the features by their information gain and build a pseudo-optimal[2] model of any size using the first k highest-ranking features.

While a smaller model reduces prediction accuracy, it would allow the model to fit in memory and reduce the execution time to compute the output. We can thus compute various models of different sizes in an off-line training (perhaps done on a server) and download the appropriate-sized model on to a portable device.

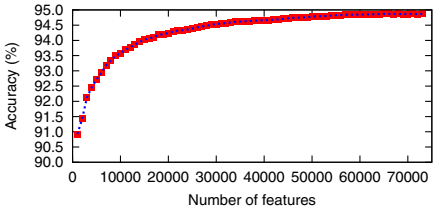
3.1 Experimental Results

We performed two sets of experiments to verify that our approach works as expected. First, we experimented with different feature set sizes and evaluated how the size of the feature set (i.e., the size of the model) affects the accuracy of the learning algorithm. For each feature set size, we computed the resulting workload using a cycle-accurate instruction set simulator. Towards this, we used the SimpleScalar simulator with the *sim-outorder* configuration [1] for collecting various execution statistics resulting from running our algorithm. As one might expect, the workload increases in an approximately linear fashion with the size of the feature set (see Figure 3(b)). However, the rate at which the accuracy improves, exhibits an exponential decay as the number of features is increased. We can see from Figure 3(a) that beyond 10000 features, the return on accuracy diminishes. Hence, given a battery-operated portable device with a power budget, it would be prudent to settle for a lower-than-optimal accuracy by using less features as input to the learning algorithm. Given that the tagging algorithm has to complete within a pre-specified deadline, settling for a lower accuracy results in a corresponding decrease in the workload (see Figure 3(c)). If we factor in the deadline, the reduced workload can be completed within the same amount of time by operating the processor at a lower frequency. Since most portable devices currently have a voltage/frequency-scalable processor (e.g., an Intel XScale processor), a lower operating frequency (and voltage) immediately translates into lower energy consumption and hence improved battery life.

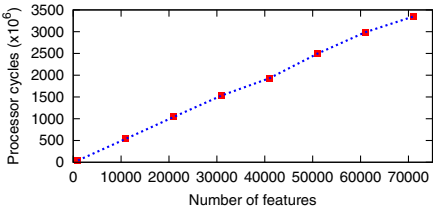
Our second set of experiments consist of using the SimpleScalar statistics and the power consumption characteristics of a PDAdevelopment board to estimate the energy savings resulting from different choices of the feature set size. We used a PDA development board from iWave Systems[3] and connected it to a National Instruments

[2] Optimal given the assumption that features are linearly independent, which is usually not true in general; but useful as a simplifying assumption.

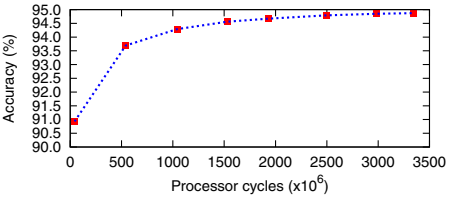
[3] http://www.iwavesystems.com/



(a) Accuracy correlates to model size (in number of features)



(b) Processor cycles correlate to model size



(c) Derived correlation between processor cycles and accuracy

Fig. 3. Processor workload versus model size and accuracy

PXI-4071 $7\frac{1}{2}$-digit Digital Multimeter to measure the power consumption of the CPU core in the iWave board for various operating frequencies. The CPU on the board is an Intel XScale PXA270, a typical commercial CPU found in many off-the-shelf PDAs. The energy consumption estimates obtained using this process accurately reflect those that are obtained by implementing our tagging algorithm on a XScale PXA270 processor.

For our target setup — PDAs, cell phones, etc. running email clients and word-processing applications — it is safe to assume that each screen can display at most 200 words to be tagged for use in downstream applications (i.e., grammar correction). We also assume that once the user triggers such an application, the tagging algorithm for these 200 words should have a 1.5 second latency at most (i.e., an execution deadline of 1.5 seconds).

Why is 1.5 seconds an acceptable deadline? Note that tagging is an intermediate task, a means to an end application such as grammar checking or name recognition. Here we make an assumption that both the tagging and the downstream application are both instances of machine learned model applications and that the models are of approximately the same size. This means that the full NLP pipeline should complete within 3 seconds, which we feel is a reasonable response time that a user is willing to tolerate on a portable device when it is actively being used. In the future we plan to conduct tests with subjects and get Mean Opinion Scores on how delay values affect user perception. Here, we also only examine the tagging execution time; initialization costs in loading the model are not examined, as these are fixed costs that do not vary. The processor frequencies and resulting power consumptions for different feature set sizes that we report in the rest of this section below are based on these assumptions.

Figure 4 gives the CPU-core power consumptions for the six different discrete frequency settings provided by the XScale PXA270 CPU. Note that the power consumption of the CPU-core varies between 0.4 to 0.13 watt, corresponding to the frequency range 520 MHz (maximum) to 104 MHz (minimum). Therefore, the maximum possible reduction in power consumption is upper bound by 68%.

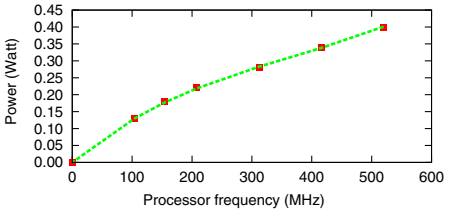
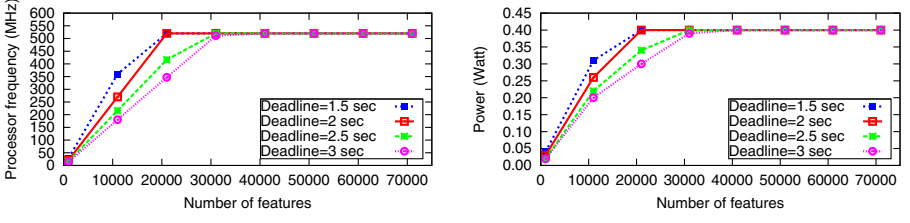


Fig. 4. Processor frequency versus CPU-core power consumption of the PDA



(a) Minimum processor frequencies corresponding specified deadlines

(b) Corresponding power consumptions

Fig. 5. Minimum processor frequencies and power consumptions on the PDA, while tagging 200 words within specified deadlines

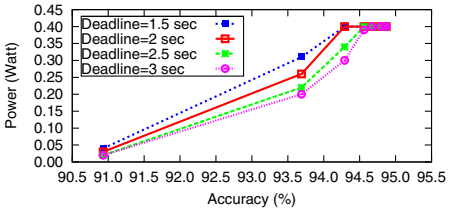


Fig. 6. Power consumption associated with different tagging accuracy values

In our experiments, we designed a simulator where the measured CPU-core power consumption of the iWave board (Figure 4) is used to estimate the power characteristics of the processor. We assume the processor frequency is continuously scalable in the simulator. We set the *deadlines* of the tagging operation to 1.5, 2, 2.5 and 3 seconds respectively, and show the processor frequencies and power consumptions associated with a variable *feature window* size in Figure 5. We investigate these latter, more relaxed deadlines to illustrate additional power savings that can be achieved if the user

requirements are relaxed (such as when the tagging task is done concurrently with other processes in the background of a foreground client application).

Note that Figure 5(a) shows that the frequency is scaled within the range of 0 – 520 MHz on the PDA's processor, to match the processing workload with variable model sizes, as specified in number of features. Figure 5(b) shows the corresponding power consumptions estimated by the power characteristics of the PDA. Notice that in the case when the model size is set to 11,000 features, the NB algorithm completes tagging 200 words with 93.6% accuracy in 1.5 seconds and 3 seconds by scaling the processor frequency, thereby achieving a power savings of 23% and 50% respectively (see Figure 6). Note that the accuracy is independent of the delay and depends on the model size.

How does this result compare with typical state-of-the-art taggers? A standard comparison point in the NLP literature is Adwait Ratnaparkhi's tagger [7], which employs a maximum entropy learning technique, similar in spirit to NB, but which is optimizing on conditional probabilities rather than generative ones. This tagger is reported to tag at 96.6% accuracy using over 100K features. In contrast, our tagger adds 3% more absolute error (a very insignificant difference) but reduces the number of features (and power consumption) by a magnitude. This adjustment makes POS tagging plausible on PDA-like devices with realistic deadlines, whereas the original NB and state-of-the-art algorithms would stall, incurring a much longer running time and possibly inducing thrashing, as the memory footprint for the model greatly exceeds the capacity of the cache and onboard memory of the device.

4 Improving Data Caching

Up to now, we have not made any changes to the NB algorithm itself. For prediction, we deal with each problem instance individually (here in POS tagging, each word is a separate problem instance): we calculate features individually in an outer loop and update each conditional probabilities for each possible output prediction in an inner loop, one at a time, as shown in Algorithm 1. While this doubly-nested loop execution pattern may be fine for single problem instances, in POS tagging (and many other NLP tasks) we need to perform testing on many instances. In POS tagging, predicting each word's tag constitutes a separate problem instance.

We can optimize data access to the probabilities stored in the model by batch processing several problem instances in one go (i.e., tagging several words in one go), making better use of parts of the model that have already been loaded in memory while processing one instance. Although this type of processing is typical in the compiler optimization and computer architecture literature, to our knowledge, this type of optimization has not been explored within the machine learning community. We believe this is because such algorithms have (up to now) only been deployed on server computing platforms, where memory is plentiful and bus speeds are fast. With such large memory footprints, optimizing for data access may only result in marginal improvements.

However, on mobile platforms we do not have this luxury as devices need to minimize space, cost and power consumption, resulting in designs with small cache sizes. For example on a cell phone, a typical shared cache size may be only 4 KB. With such small cache sizes, dealing with cache misses become a significant proportion of total

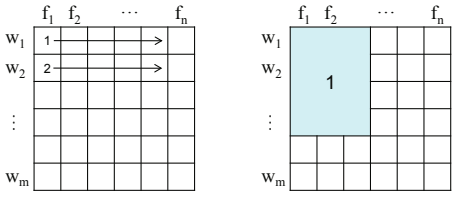


Fig. 7. (1) Naïve Bayes standard computation sequence. (2) Proposed windowed NB computation, where a word window of $WWIN = 4$ and a feature window of $FWIN = 3$ is used.

cycles incurred during the model application. As such, in this scenario, it is worthwhile examining how to lower cache misses by optimizing the use of cached information.

We introduce the notion of batch processing through the introduction of a *word (instance) window*. We predict tags for an entire sequence of words $w_1, w_2, ..., w_m$ rather just for an individual word w_j. In the basic Naïve Bayes algorithm, we loop over all features for each individual instance, computing each feature for a particular instance. In Figure 7(1) this corresponds to calculating values row by row, where each row is a new instance that needs to be tagged. This may be wasteful as the method to compute a feature needs to be reloaded each time a new instance's output class needs to be predicted. Instead, we can compute a feature for a window of words at one time, more effectively utilizing cached information. In our schematic representation, this corresponse to tagging in vertical columns.

We can batch process not only multiple problem instances but also multiple features, computing a set of n features for m words, in one go. In theory, this best utilizes the memory when the necessary memory footprint for computing $m \times n$ is roughly the cache size. This corresponds to processing in vertical tiles, as shown in Figure 7(2). When we finish with one tile, we proceed to the next tile underneath it, looping back to the next set of features. We define this setting through these two parameters: a *feature window* (FWIN) and *word window* (WWIN).

We thus experimented with different cache sizes typical of smaller, portable devices such as cell phones. Note here that such devices are typically more compact and their displays are smaller, allowing an average of 20 words to be displayed. Also, users are used to having significant longer latency on such devices, so we can relax our processing deadline further to 5 seconds. For these devices, we explored different *word* and *feature window* sizes to exploit maximum data cache usage.

The key results we obtained suggest (1) that for any given cache size, there are optimal *word* and *feature window* sizes that maximize data cache reuse, and (2) that not using such windows at all signficantly increases processing load (see Figure 8). The improved data cache reuse immediately translate into fewer cache misses, and hence lower execution time/workload, leading to lower power consumption.

Figure 8 shows the number of cache misses on the 4 KB cache, when tagging 20 words with our optimized algorithm. This plot shows a clear inflection point for the word window axis, illustrating the significance of batching problem instance predictions. Varying the feature window has a much less noticeable effect. We observe that for the cache of 4 KB, when the feature window size is 50 and the word window size is

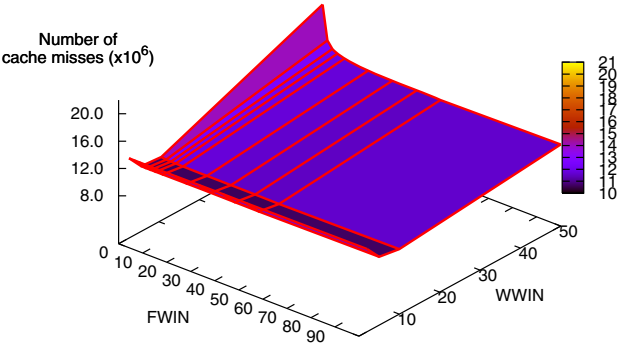


Fig. 8. Cache misses associated with different feature window size and word window size

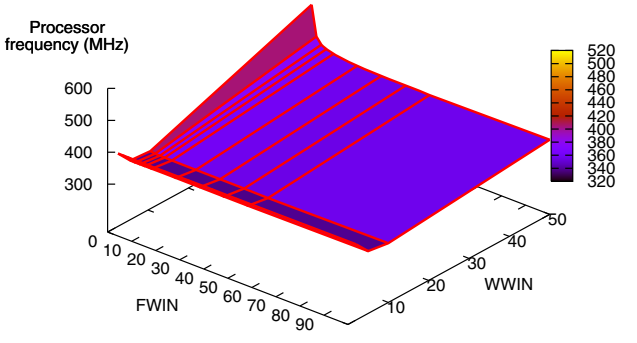


Fig. 9. Processor frequencies to tag 20 words within 5 seconds, associated with different feature window size and word window size

10, the number of cache misses goes down to its minimal point of 10.3×10^6. This is a 48% reduction in cache misses, compared with the case of 20.0×10^6 misses when the feature window and word window are arbitrarily set to 1 and 50 respectively.

We assume that the computational cost to process models (without accounting for cache misses) is linearly proportional to the model size. With a cache penalty of 100 cycles, we derived the total number of processor cycles incurred while tagging 20 words with different feature and word window sizes. With a deadline of 5 sec for tagging 20 words, we then derived the minimum clock frequency at which the processor needs to be clocked to meet this deadline. These results are shown in Figure 9. When the deadline to tag 20 words as 5 seconds, the minimal processor frequency to accomplish tagging is 317.12 MHz, when the feature window is set to 50 and the word window is set to 10; while the maximal processor frequency of the same task goes up to 501.49 MHz, when the feature window is set to 1 and the word window is set to 50. Hence, with the optimal selection of feature window and word window sizes, we were able to

reduce the processing frequency required by 36.7%, which immediately translates to significant energy savings on a voltage/frequency-scalable processor.

5 Concluding Remarks

We have introduced the domain of machine learning model applications, where embedded software design methods may be employed to create lightweight, power-sensitive applications. We believe that this domain is critically important in the coming years as portable battery-powered computing becomes a dominant platform, and more applications need to apply machine learning techniques to acquire intelligent behavior.

Towards this, we examined a typical scenario of part-of-speech tagging using the Naïve Bayes algorithm. We showed that with proper use of feature selection, we can tune the model size to fit the memory specifications of a mobile device and reduced the power consumption when applying the model. We also validated these findings with physical voltage and current measurements on an iWave board, as well as via simulation using SimpleScalar.

Further reductions in power consumption can be observed when we redesign how the model application process computes and re-uses the model data. This process is controlled by our newly introduced parameters of *word window* and *feature window*, enabling the batch processing of several words and features together. We showed that the batching of several problems leads to more significant savings, demonstrating a 48% reduction in cache misses when we optimally tune these windows to fit an example cache of 4 KB.

Acknowledgment

We would like to thank Prof. Min-Yen Kan for initiating this work, for numerous discussions, and for his help with the writing of this paper.

References

1. Austin, T., Larson, E., Ernst, D.: SimpleScalar: An infrastructure for computer system modeling. IEEE Computer 35(2), 59–67 (2002)
2. Blum, A.L., Langley, P.: Selection of relevant features and examples in machine learning. Artificial Intelligence 97(1–2), 245–271 (1997)
3. Heisele, B., Ho, P., Poggio, T.: Face recognition with support vector machines: Global versus component-based approach. In: Proceedings of ICCV 2001, Vancouver, Canada, pp. 688–694 (2001)
4. Kononenko, I.: Machine learning for medical diagnosis: History, state of the art and perspective. Artificial Intelligence in Medicine 23(1), 89–109 (2001) (invited paper)
5. Marcus, M.P., Santorini, B., Marcinkiewicz, M.A.: Building a large annotated corpus of english: The Penn Treebank. Computational Linguistics 19(2), 313–330 (1994)
6. Mitchell, T.M.: Machine Learning. McGraw-Hill, New York (1997)
7. Ratnaparkhi, A.: Trainable methods for surface natural language generation. In: Proceedings of ANLP-NAACL 2002, Seattle, Washington, USA, pp. 194–201 (2000)
8. Zhou, G., Su, J.: Named entity recognition using an HMM-based chunk tagger. In: Proceedings of ACL 2002, Philadelphia, PA, USA, pp. 473–480 (2002)