Parallel Active Learning: Eliminating Wait Time with Minimal Staleness

Robbie Haertel, Paul Felt, Eric Ringger, Kevin Seppi
Department of Computer Science
Brigham Young University
Provo, Utah 84602, USA
rah67@cs.byu.edu, pablofelt@gmail.com,
ringger@cs.byu.edu, kseppi@cs.byu.edu
http://nlp.cs.byu.edu/

Abstract

A practical concern for Active Learning (AL) is the amount of time human experts must wait for the next instance to label. We propose a method for eliminating this wait time independent of specific learning and scoring algorithms by making scores always available for all instances, using old (stale) scores when necessary. The time during which the expert is annotating is used to train models and score instances–in parallel–to maximize the recency of the scores. Our method can be seen as a parameterless, dynamic batch AL algorithm. We analyze the amount of staleness introduced by various AL schemes and then examine the effect of the staleness on performance on a part-of-speech tagging task on the Wall Street Journal. Empirically, the parallel AL algorithm effectively has a batch size of one and a large candidate set size but eliminates the time an annotator would have to wait for a similarly parameterized batch scheme to select instances. The exact performance of our method on other tasks will depend on the relative ratios of time spent annotating, training, and scoring, but in general we expect our parameterless method to perform favorably compared to batch when accounting for wait time.

1 Introduction

Recent emphasis has been placed on evaluating the effectiveness of active learning (AL) based on realistic cost estimates (Haertel et al., 2008; Settles et al., 2008; Arora et al., 2009). However, to our knowledge, no previous work has included in the cost measure the amount of time that an expert annotator must wait for the active learner to provide instances. In fact, according to the standard approach to cost measurement, there is no reason not to use the theoretically optimal (w.r.t. a model, training procedure, and utility function) (but intractable) approach (see Haertel et al., 2008).

In order to more fairly compare complex and time-consuming (but presumably superior) selection algorithms with simpler (but presumably inferior) algorithms, we describe "best-case" (minimum, from the standpoint of the payer) and "worst-case" (maximum) cost scenarios for each algorithm. In the best-case cost scenario, annotators are paid only for the time they spend actively annotating. The worst-case cost scenario additionally assumes that annotators are always on-the-clock, either annotating or waiting for the AL framework to provide them with instances. In reality, human annotators work on a schedule and are not always annotating or waiting, but in general they expect to be paid for the time they spend waiting for the next instance. In some cases, the annotator is not paid directly for waiting, but there are always opportunity costs associated with time-consuming algorithms, such as time to complete a project. In reality, the true cost usually lies between the two extremes.

However, simply analyzing only the best-case cost, as is the current practice, can be misleading, as illustrated in Figure 1. When excluding waiting time for a particular selection algorithm[1] ("AL Annotation Cost Only"), the performance is much bet-

[1]We use the ROI-based scoring algorithm (Haertel et al., 2008) and the zero-staleness technique, both described below.

Proceedings of the NAACL HLT 2010 Workshop on Active Learning for Natural Language Processing, pages 33–41,
Los Angeles, California, June 2010.

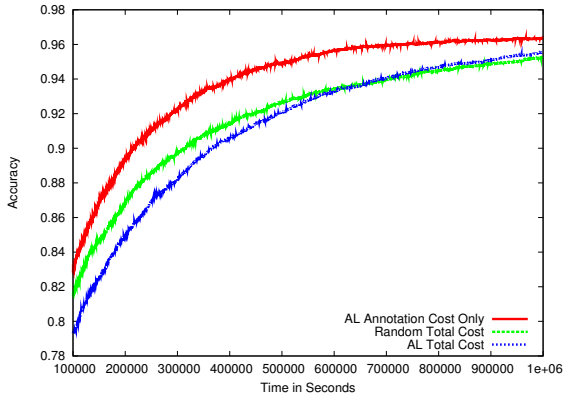

Figure 1: Accuracy as a function of cost (time). Side-by-side comparison of best-case and worst-case cost measurement scenarios reveals that not accounting for the time required by AL to select instances affects the evaluation of an AL algorithm.

ter than the cost of random selection ("Random Total Cost"), but once waiting time is accounted for ("AL Total cost"), the AL approach can be worse than random. Given only the best-case cost, this algorithm would appear to be very desirable. Yet, practitioners would be much less inclined to adopt this algorithm knowing that the worst-case cost is potentially no better than random. In a sense, waiting time serves as a natural penalty for expensive selection algorithms. Therefore, conclusions about the usefulness of AL selection algorithms should take both best-case and worst-case costs into consideration.

Although it is current practice to measure only best-case costs, Tomanek et al. (2007) mention as a desideratum for practical AL algorithms the need for what they call fast selection time cycles, i.e., algorithms that minimize the amount of time annotators wait for instances. They address this by employing the batch selection technique of Engleson and Dagan (1996). In fact, most AL practitioners and researchers implicitly acknowledge the importance of wait time by employing batch selection.

However, batch selection is not a perfect solution. First, using the tradtional implementation, a "good" batch size must be specified beforehand. In research, it is easy to try multiple batch sizes, but in practice where there is only one chance with live annotators, specifying a batch size is a much more difficult problem; ideally, the batch size would be set during the process of AL. Second, traditional methods use the same batch size throughout the entire learning process. However, in the beginning stages of AL, models have access to very little training data and retraining is often much less costly (in terms of time) than in the latter stages of AL in which models are trained on large amounts of data. Intuitively, small batch sizes are acceptable in the beginning stages, whereas large batch sizes are desirable in the latter stages in order to mitigate the time cost of training. In fact, Haertel et al. (2008) mention the use of an increasing batch size to speed up their simulations, but details are scant and the choice of parameters for their approach is task- and dataset-dependent. Also, the use of batch AL causes instances to be chosen without the benefit of all of the most recently annotated instances, a phenomenon we call *staleness* and formally define in Section 2. Finally, in batch AL, the computer is left idle while the annotator is working and vice-verse.

We present a parallel, parameterless solution that can eliminate wait time irrespective of the scoring alogrithm and training method. Our approach is based on the observation that instances can always be available for annotation if we are willing to serve instances that may have been selected without the benefit of the most recent annotations. By having the computer learner do work while the annotator is busy annotating, we are able to mitigate the effects of using these older annotations.

The rest of this paper will proceed as follows: Section 2 defines staleness and presents a progression of four AL algorithms that strike different balances between staleness and wait time, culminating in our parallelized algorithm. We explain our methodology and experimental parameters in Section 3 and then present experimental results and compare the four AL algorithms in Section 4. Conclusions and future work are presented in Section 5.

2 From Zero Staleness to Zero Wait

We work within a pool- and score-based AL setting in which the active learner selects the next instance from an unlabeled pool of data $\mathcal{U}$. A scoring function σ (aka scorer) assigns instances a score using a model θ trained on the labeled data $\mathcal{A}$; the scores serve to rank the instances. Lastly, we assume that

Input: A seed set of annotated instances $\mathcal{A}$, a set of pairs of unannotated instances and their initial scores $\mathcal{S}$, scoring function σ, the candidate set size N, and the batch size B
Result: $\mathcal{A}$ is updated with the instances chosen by the AL process as annotated by the oracle

```
1  while S ≠ ∅ do
2      θ ← TrainModel(A)
3      stamp ← |A|
4      C ← ChooseCandidates(S,N)
5      K ← {(c[inst], σ(c[inst], θ)) | c ∈ C}
6      S ← S − C ∪ K
7      T ← pairs from K with c[score] in the top B
       scores
8      for t ∈ T do
9          S ← S − t
10         staleness ← |A| − stamp ;  // unused
11         A ← A ∪ Annotate(t)
12     end
13 end
```

Algorithm 1: Pool- and score-based active learner.

an unerring oracle provides the annotations. These concepts are demonstrated in Algorithm 1.

In this section, we explore the trade-off between staleness and wait time. In order to do so, it is beneficial to quantitatively define staleness, which we do in the context of Algorithm 1. After each model θ is trained, a stamp is associated with that θ that indicates the number of annotated instances used to train it (see line 3). The staleness of an item is defined to be the difference between the current number of items in the annotated set and the stamp of the scorer that assigned the instance a score. This concept can be applied to any instance, but it is particularly informative to speak of the staleness of instances at the time they are actually annotated (we will simply refer to this as staleness, disambiguating when necessary; see line 10). Intuitively, an AL scheme that chooses instances having less stale scores will tend to produce a more accurate ranking of instances.

2.1 Zero Staleness

There is a natural trade-off between staleness and the amount of time an annotator must wait for an instance. Consider Algorithm 1 when $B = 1$ and $N = \infty$ (we refer to this parameterization as **zerostale**). In line 8, a single instance is selected for annotation ($|\mathcal{T}| = B = 1$); the staleness of this instance is zero since no other annotations were provided between the time it was scored and the time it was removed. Therefore, this algorithm will never select stale instances and is the only way to guarantee that no selected instances are stale.

However, the zero staleness property comes with a price. Between every instance served to the annotator, a new model must be trained and every instance scored using this model, inducing potentially large waiting periods. Therefore, the following options exist for reducing the wait time:

1. Optimize the learner and scoring function (including possible parallelization)
2. Use a different learner or scoring function
3. Parallelize the scoring process
4. Allow for staleness

The first two options are specific to the learning and scoring algorithms, whereas we are interested in reducing wait time independent of these in the general AL framework. We describe option 3 in section 2.4; however, it is important to note that when training time dominates scoring, the reduction in waiting time will be minimal with this option. This is typically the case in the latter stages of AL when models are trained on larger amounts of data.

We therefore turn our attention to option 4: in this context, there are at least three ways to decrease the wait time: (A) train less often, (B) score fewer items, or (C) allow old scores to be used when newer ones are unavailable. Strategies A and B are the batch selection scheme of Engelson and Dagan (1996); an algorithm that allows for these is presented as Algorithm 1, which we refer to as "traditional" batch, or simply **batch**. We address the traditional batch strategy first and then address strategy C.

2.2 Traditional Batch

In order to train fewer models, Algorithm 1 can provide the annotator with several instances scored using the same scorer (controlled by parameter B); consequently, staleness is introduced. The first item annotated on line 11 has zero staleness, having been scored using a scorer trained on all available annotated instances. However, since a model is not retrained before the next item is sent to the annotator,

the next items have staleness $1, 2, \cdots, B-1$. By introducing this staleness, the time the annotator must wait is amortized across all B instances in the batch, reducing the wait time by approximately a factor of B. The exact effect of staleness on the *quality* of instances selected is scorer- and data-dependent.

The parameter N, which we call the candidate set size, specifies the number of instances to score. Typically, candidates are chosen in round-robin fashion or with uniform probability (without replacement) from $\mathcal{U}$. If scoring is expensive (e.g., if it involves parsing, translating, summarizing, or some other time-consuming task), then reducing the candidate set size will reduce the amount of time spent scoring by the same factor. Interestingly, this parameter does not affect staleness; instead, it affects the probability of choosing the same B items to include in the batch when compared to scoring all items. Intuitively, it affects the probability of choosing B "good" items. As N approaches B, this probability approaches uniform random and performance approaches that of random selection.

2.3 Allowing Old Scores

One interesting property of Algorithm 1 is that line 7 guarantees that the only items included in a batch are those that have been scored in line 5. However, if the candidate set size is small (because scoring is expensive), we could compensate by reusing scores from previous iterations when choosing the best items. Specifically, we change line 7 to instead be:

$\mathcal{T} \leftarrow$ pairs from $\mathcal{S}$ with $c[score]$ in the top B scores

We call this **allowold**, and to our knowledge, it is a novel approach. Because selected items may have been scored many "batches" ago, the expected staleness will never be less than in **batch**. However, if scores do not change much from iteration to iteration, then old scores will be good approximations of the actual score and therefore not all items necessarily need to be rescored every iteration. Consequently, we would expect the quality of instances selected to approach that of **zerostale** with less waiting time. It is important to note that, unlike **batch**, the candidate set size does directly affect staleness; smaller N will increase the likelihood of selecting an instance scored with an old model.

2.4 Eliminating Wait Time

There are portions of Algorithm 1 that are trivially parallelizable. For instance, we could easily split the candidate set into equal-sized portions across P processors to be scored (see line 5). Furthermore, it is not necessary to wait for the scorer to finish training before selecting the candidates. And, as previously mentioned, it is possible to use parallelized training and/or scoring algorithms. Clearly, wait time will decrease as the speed and number of processors increase. However, we are interested in parallelization that can guarantee zero wait time independent of the training and scoring algorithms without precluding these other forms of parallelization.

All other major operations of Algorithm 1 have serial dependencies, namely, we cannot score until we have trained the model and chosen the candidates, we cannot select the instances for the batch until the candidate set is scored, and we cannot start annotating until the batch is prepared. These dependencies ultimately lead to waiting.

The key to eliminating this wait time is to ensure that all instances have scores at all times, as in **allowold**. In this way, the instance that currently has the highest score can be served to the annotator without having to wait for any training or scoring. If the scored instances are stored in a priority queue with a constant time *extract-max* operation (e.g., a sorted list), then the wait time will be negligible. Even a heap (e.g., binary or Fibonacci) will often provide negligible overhead. Of course, eliminating wait time comes at the expense of added staleness as explained in the context of **allowold**.

This additional staleness can be reduced by allowing the computer to do work while the oracle is busy annotating. If models can retrain and score most instances in the amount of time it takes the oracle to annotate an item, then there will be little staleness.[2]

Rather than waiting for training to complete before beginning to score instances, the old scorer can be used until a new one is available. This allows us to train models and score instances in parallel. Fast training and scoring procedures result in more instances having up-to-date scores. Hence, the stale-

[2]Since the annotator requests the next instance immediately after annotating the current instance, the next instance is virtually guaranteed to have a staleness factor of at least 1.

ness (and therefore quality) of selected instances depends on the relative time required to train and score models, thereby encouraging efficient training and scoring algorithms. In fact, the other forms of parallelization previously mentioned can be leveraged to reduce staleness rather than attempting to directly reduce wait time.

These principles lead to Algorithm 2, which we call **parallel** (for clarity, we have omitted steps related to concurrency). `AnnotateLoop` represents the tireless oracle who constantly requests instances. The call to `Annotate` is a surrogate for the actual annotation process and most importantly, the time spent in this method is the time required to provide annotations. Once an annotation is obtained, it is placed on a shared buffer $\mathcal{B}$ where it becomes available for training. While the annotator is, in effect, a producer of annotations, `TrainLoop` is the consumer which simply retrains models as annotated instances become available on the buffer. This buffer is analagous to the batch used for training in Algorithm 1. However, the size of the buffer changes dynamically based on the relative amounts of time spent annotating and training. Finally, `ScoreLoop` endlessly scores instances, using new models as soon as they are trained. The set of instances scored with a given model is analagous to the candidate set in Algorithm 1.

Input: A seed set of annotated instances $\mathcal{A}$, a set of pairs of unannotated instances and their initial scores $\mathcal{S}$, and a scoring function σ
Result: $\mathcal{A}$ is updated with the instances chosen by the AL process as annotated by the oracle

$\mathcal{B} \leftarrow \emptyset$, $\theta \leftarrow$ null
`Start(AnnotateLoop)`
`Start(TrainLoop)`
`Start(ScoreLoop)`

procedure `AnnotateLoop()`
  while $\mathcal{S} \neq \emptyset$ **do**
    $t \leftarrow c$ from $\mathcal{S}$ having max $c[score]$
    $\mathcal{S} \leftarrow \mathcal{S} - t$
    $\mathcal{B} \leftarrow \mathcal{B} \cup$ `Annotate(`t`)`
  end
end

procedure `TrainLoop()`
  while $\mathcal{S} \neq \emptyset$ **do**
    $\theta \leftarrow$ `TrainModel(`$\mathcal{A}$`)`
    $\mathcal{A} \leftarrow \mathcal{A} \cup \mathcal{B}$
    $\mathcal{B} \leftarrow \emptyset$
  end
end

procedure `ScoreLoop()`
  while $\mathcal{S} \neq \emptyset$ **do**
    $c \leftarrow$ `ChooseCandidate(`$\mathcal{S}$`)`
    $\mathcal{S} \leftarrow$
    $\mathcal{S} - \{c\} \cup \{(c[inst], \sigma(c[inst], \theta)) | c \in \mathcal{S}\}$
  end
end

Algorithm 2: **parallel**

3 Experimental Design

Because the performance of the parallel algorithm and the "worst-case" cost analysis depend on wait time, we hold computing resources constant, running all experiments on a cluster of Dell PowerEdge M610 servers equipped with two 2.8 GHz quad-core Intel Nehalem processors and 24 GB of memory.

All experiments were on English part of speech (POS) tagging on the POS-tagged Wall Street Journal text in the Penn Treebank (PTB) version 3 (Marcus et al., 1994). We use sections 2-21 as initially unannotated data and randomly select 100 sentences to seed the models. We employ section 24 as the set on which tag accuracy is computed, but do not count evaluation as part of the wait time. We simulate annotation costs using the cost model from Ringger et al. (2008): $cost(\mathbf{s}) = (3.80 \cdot l + 5.39 \cdot c + 12.57)$, where l is the number of tokens in the sentence, and c is the number of pre-annotated tags that need correction, which can be estimated using the current model. We use the same model for pre-annotation as for scoring.

We employ the return on investment (ROI) AL framework introduced by Haertel et. al (2008). This framework requires that one define both a cost and benefit estimate and selects instances that maximize $\frac{benefit(x) - cost(x)}{cost(x)}$. For simplicity, we estimate cost as the length of a sentence. Our benefit model estimates the utility of each sentence as follows: $benefit(\mathbf{s}) = -\log\left(\max_{\mathbf{t}} p(\mathbf{t}|\mathbf{s})\right)$ where $p(\mathbf{t}|\mathbf{s})$ is the probability of a tagging given a sentence. Thus, sentences having low average (in the geometric mean sense) per-tag probability are favored. We use a maximum entropy Markov model to estimate these probabilities, to pre-annotate instances, and to evaluate accuracy.

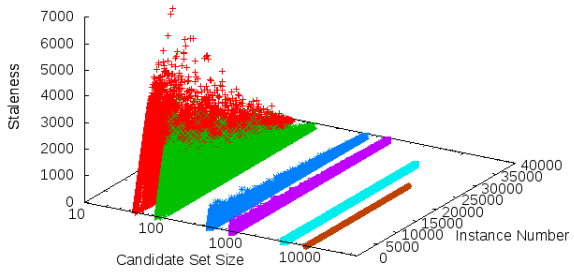

Figure 2: Staleness of the **allowold** algorithm over time for different candidate set sizes

4 Results

Two questions are pertinent regarding staleness: how much staleness does an algorithm introduce? and how detrimental is that staleness? For **zerostale** and **batch**, the first question was answered analytically in a previous section. We proceed by addressing the answer empirically for **allowold** and **parallel** after which we examine the second question.

Figure 2 shows the observed staleness of instances selected for annotation over time and for varying candidate set sizes for **allowold**. As expected, small candidate sets induce more staleness, in this case in very high amounts. Also, for any given candidate set size, staleness decreases over time (after the beginning stages), since the effective candidate set includes an increasingly larger percentage of the data.

Since **parallel** is based on the same allow-old-scores principle, it too could potentially see highly stale instances. However, we found the average per-instance staleness of **parallel** to be very low: **1.10**; it was never greater than **4** in the range of data that we were able to collect. This means that for our task and hardware, the amount of time that the oracle takes to annotate an instance is high enough to allow new models to retrain quickly and score a high percentage of the data before the next instance is requested.

We now examine effect that staleness has on AL performance, starting with **batch**. As we have shown, higher batch sizes guarantee more staleness so we compare the performance of several batch sizes (with a candidate set size of the full data) to **zerostale** and **random**. In order to tease out the effects that the staleness has on performance from the effects that the batches have on wait time (an element of performance), we purposely ignore wait time. The results are shown in Figure 3. Not surprisingly, **zerostale** is slightly superior to the batch methods, and all are superior to random selection. Furthermore, **batch** is not affected much by the amount of staleness introduced by reasonable batch sizes: for $B < 100$ the increase in cost of attaining 95% accuracy compared to **zerostale** is 3% or less.

Recall that **allowold** introduces more staleness than **batch** by maintaining old scores for each instance. Figure 4 shows the effect of different candidate set sizes on this approach while fixing batch size at 1 (wait time is excluded as before). Larger candidate set sizes have less staleness, so not surprisingly performance approaches **zerostale**. Smaller candidate set sizes, having more staleness, perform similarly to random during the early stages when the model is changing more drastically each instance. In these circumstances, scores produced from earlier models are not good approximations to the actual scores so allowing old scores is detrimental. However, once models stabilize and old scores become better approximations, performance begins to approach that of **zerostale**.

Figure 6 compares the performance of **allowold** for varying batch sizes for a fixed candidate set size (5000; results are similiar for other settings). As before, performance suffers primarily in the early stages and for the same reasons. However, a batch excerbates the problem since multiple instances with poor scores are selected simultaneously. Nevertheless, the performance appears to mostly recover once the scorers become more accurate. We note that batch sizes of 5 and 10 increase the cost of acheiving 95% accuracy by 3% and 10%, respectively, compared to **zerostale**. The implications for **parallel** are that stalness may not be detrimental, especially if batch sizes are small and candidate set sizes are large in the beginning stages of AL.

Figure 5 compares the effect of staleness on all four algorithms when excluding wait time ($B = 20$, $N = 5000$ for the batch algorithms). After achieving around 85% accuracy, **batch** and **parallel** are virtually indistinguishable from **zerostale**, implying that the staleness in these algorithms is mostly ignorable. Interestingly, **allowold** costs around 5% more than **zerostale** to acheive an accuracy of 95%. We attribute this to increased levels of staleness which **parallel** combats by avoiding idle time.

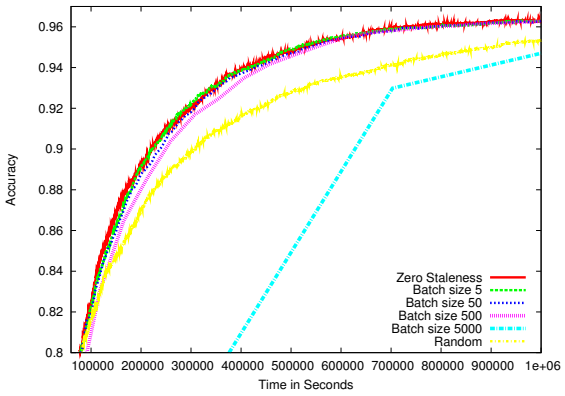

Figure 3: Effect of staleness due to batch size for **batch**, $N = \infty$

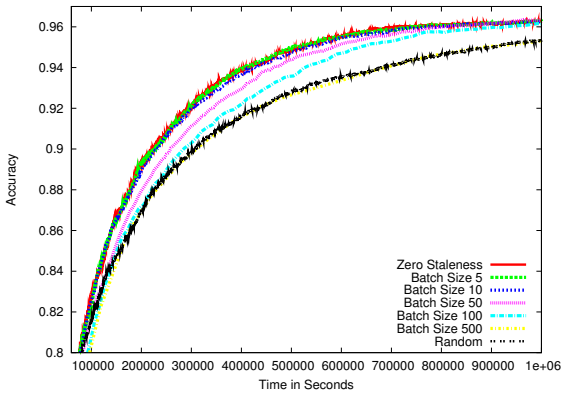

Figure 6: Effect of staleness due to batch size for **allowold**, $N = 5000$

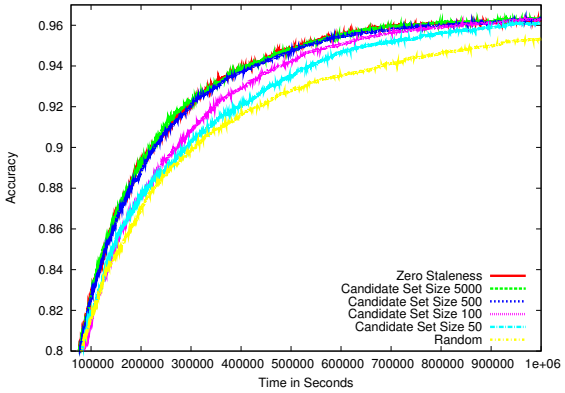

Figure 4: Effect of staleness due to candidate set size for **allowold**, $B = 1$

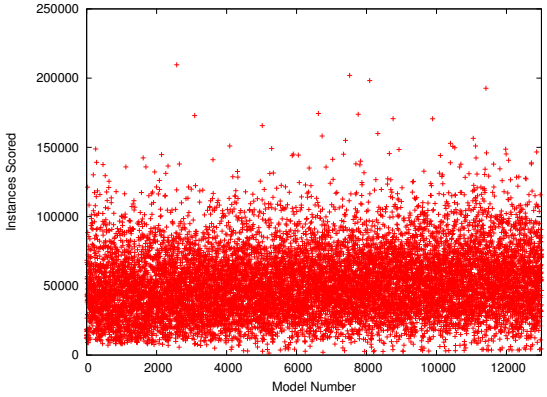

Figure 7: Effective candidate set size of **parallel** over time

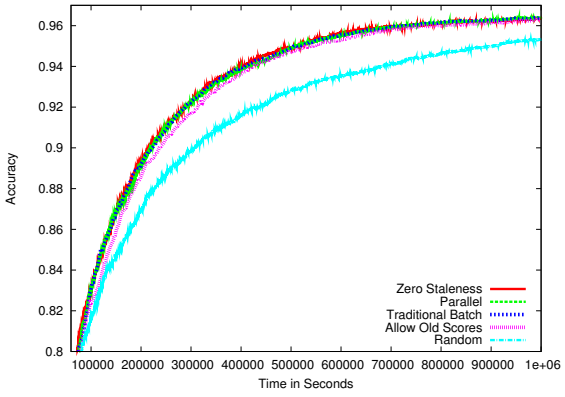

Figure 5: Comparison of algorithms (not including wait time)

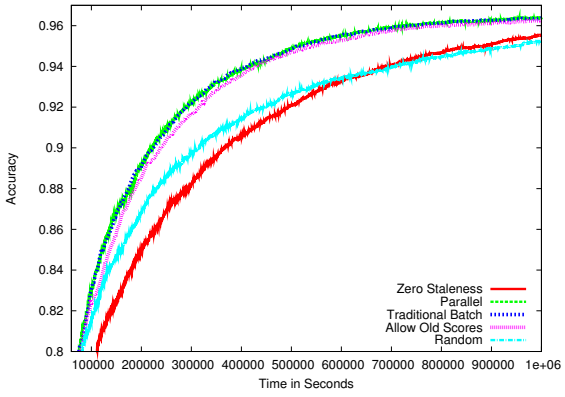

Figure 8: Comparison of algorithms (including wait time)

Since the amount of data **parallel** uses to train models and score instances depends on the amount of time instances take to annotate, the "effective" candidate set sizes and batch sizes over time is of interest. We found that the models were always trained after receiving exactly one instance, within the data we were able to collect. Figure 7 shows the number of instances scored by each successive scorer, which appears to be very large on average: over 75% of the time the scorer was able to score the entire dataset. For this task, the human annotation time is much greater than the amount of time it takes to train new models (at least, for the first 13,000 instances). The net effect is that under these conditions, **parallel** is parameterized similar to **batch** with $B = 1$ and N very high, i.e., approaching **zerostale**, and therefore has very low staleness, yet does so without incurring the waiting cost.

Finally, we compare the performance of the four algorithms using the same settings as before, but include wait time as part of the cost. The results are in Figure 8. Importantly, **parallel** readily outperforms **zerostale**, costing 40% less to reach 95% accuracy. **parallel** also appears to have a slight edge over **batch**, reducing the cost to acheive 95% accuracy by a modest 2%; however, had the simulation continued, we we may have seen greater gains given the increasing training time that occurs later on. It is important to recognize in this comparison that the purpose of **parallel** is not necessarily to significantly outperform a well-tuned batch algorithm. Instead, we aim to eliminate wait time without requiring parameters, while hopefully maintaining performance. These results suggest that our approach successfully meets these criteria.

Taken as a whole, our results appear to indicate that the net effect of staleness is to make selection more random. Models trained on little data tend to produce scores that are not reflective of the actual utility of instances and essentially produce a random ranking of instances. As more data is collected, scores become more accurate and performance begins to improve relative to random selection. However, stale scores are by definition produced using models trained with less data than is currently available, hence more staleness leads to more random-like behavior. This explains why batch selection tends to perform well in practice for "reasonable" batch sizes: the amount of staleness introduced by batch ($\frac{B-1}{2}$ on average for a batch of size B) introduces relatively little randomness, yet cuts the wait time by approximately a factor of B.

This also has implications for our **parallel** method of AL. If a given learning algorithm and scoring function outperform random selection when using **zerostale** and excluding wait time, then any added staleness should cause performance to more closely resemble random selection. However, once waiting time is accounted for, performance could actually degrade below that of random. In **parallel**, more expensive training and scoring algorithms are likely to introduce larger amounts of staleness, and would cause performance to approach random selection. However, **parallel** has no wait time, and hence our approach should always perform at least as well as random in these circumstances. In contrast, poor choices of parameters in **batch** could perform worse than random selection.

5 Conclusions and Future Work

Minimizing the amount of time an annotator must wait for the active learner to provide instances is an important concern for practical AL. We presented a method that can eliminate wait time by allowing instances to be selected on the basis of the most recently assigned score. We reduce the amount of staleness this introduces by allowing training and scoring to occur in parallel while the annotator is busy annotating. We found that on PTB data using a MEMM and a ROI-based scorer that our parameterless method performed slightly better than a hand-tuned traditional batch algorithm, without requiring any parameters. Our approach's parallel nature, elimination of wait time, ability to dynamically adapt the batch size, lack of parameters, and avoidance of worse-than-random behavior, make it an attractive alternative to **batch** for practical AL.

Since the performance of our approach depends on the relative time spent annotating, training, and scoring, we wish to apply our technique in future work to more complex problems and models that have differing ratios of time spent in these areas. Future work could also draw on the *continual computation* framework (Horvitz, 2001) to utilize idle time in other ways, e.g., to predict annotators' responses.

References

S. Arora, E. Nyberg, and C. P. Rosé. 2009. Estimating annotation cost for active learning in a multi-annotator environment. In *Proceedings of the NAACL HLT 2009 Workshop on Active Learning for Natural Language Processing*, pages 18–26.

S. P. Engelson and I. Dagan. 1996. Minimizing manual annotation cost in supervised training from corpora. In *Proceedings of the 34th annual meeting on Association for Computational Linguistics*, pages 319–326.

R. A. Haertel, K. D. Seppi, E. K. Ringger, and J. L. Carroll. 2008. Return on investment for active learning. In *NIPS Workshop on Cost Sensitive Learning*.

E. Horvitz. 2001. Principles and applications of continual computation. *Artificial Intelligence Journal*, 126:159–96.

M. P. Marcus, B. Santorini, and M. A. Marcinkiewicz. 1994. Building a large annotated corpus of english: The penn treebank. *Computational Linguistics*, 19:313–330.

E. Ringger, M. Carmen, R. Haertel, K. Seppi, D. Lonsdale, P. McClanahan, J. Carroll, and N. Ellison. 2008. Assessing the costs of machine-assisted corpus annotation through a user study. In *Proc. of LREC*.

B. Settles, M. Craven, and L. Friedland. 2008. Active learning with real annotation costs. In *Proceedings of the NIPS Workshop on Cost-Sensitive Learning*, pages 1069–1078.

K. Tomanek, J. Wermter, and U. Hahn. 2007. An approach to text corpus construction which cuts annotation costs and maintains reusability of annotated data. *Proc. of EMNLP-CoNLL*, pages 486–495.