Keeping Things that Matter: An Exploration on Delayed Feedback Online Learning

Marco Tulio Ribeiro and Ignacio Cano
Department of Computer Science & Engineering
University of Washington
Seattle, WA 98195
marcotcr, icano@cs.washington.edu

Abstract

Online learning algorithms operate on a single instance at a time, allowing for updates that are fast, simple and perform well in a wide range of practical settings [1]. In this paper, we focus on online learning algorithms under delayed feedback, where the true labels arrive minutes, hours or even days later. Current solutions to this problem assume that all the instances are kept until the corresponding labels arrive [2]. In this project, we aim to examine this problem in a scenario where limited resources are available, so that it becomes infeasible to keep all of the instances in memory until the correct labels arrive. Herein we analyze different strategies for keeping only the data that matters: either by dropping certain training instances, features or both. We borrow techniques from the Active Learning literature, modified to fit this scenario. We also propose new techniques and baselines.

1 Introduction

Traditional online learning algorithms operate in rounds. In round $t$ the algorithm receives an instance $x^t \in \mathbb{R}^d$ and applies its current prediction rule to make a prediction $\hat{y}^t \in Y$. It then receives the true label $y^t \in Y$ and suffers a loss $l(y^t, \hat{y}^t)$. Finally, the algorithm updates its model using $(x^t, y^t)$ and proceeds to the next round [3].

In some applications the true label is revealed long after the prediction is made. An example of this is anomaly detection in network traffic. In such systems, an attack can be noticed hours after it starts. Pay per purchase is another case, where advertisers pay the publisher (typically the website owner where the ad is placed) only if an item is clicked, and then bought. A user might perform the purchase after a week she/he has clicked on the ad. Wikipedia vandalism is another example. In this case, the perpetrator adds, removes or changes the content of a page in a deliberate attempt to compromise the integrity of the system. Wiki administrators may discover the vandalizing edits later on and revert them.

In these type of scenarios, it may be intractable to keep all instances or features in memory until the feedback arrives, especially when the feedback delay is long. In order to get the best out of the instances that are kept, one could think of multiple strategies to handle the available space: (1) dropping instances, (2) dropping features, (3) reducing the dimensionality of instances [4], (4) combinations of the previous 3. Dropping can be thought as moving instances/features to some slower stable storage (such as disk) or actually throwing them away. In this project, we considered the second option.

It should be noted that while the delayed feedback scenario has been studied before [2,5], previous work assume that all instances can be kept in memory. As far as we know, the problem we are
studying here is an original one, and thus we provide no literature review (as there seems to be no
literature yet).

There are similarities between the problem of selecting which instances to keep and Active Learning
[6], in the sense that in general we would want to select the most informative instances to keep.
However, there are key constraints: in the delayed feedback setting, we are not guaranteed to get the
label for the examples we select. The limitation is in terms of memory size, and not in terms of
labels provided - and thus dimensionality reduction or compression can be good strategies. Finally,
in this scenario we have to select a set of examples / features instead of a single one, and thus we
may want to preserve some form of “diversity” of examples, such that we do not keep redundant
information.

For the first solution (dropping instances), we evaluate several policies. Some baselines are: drop-
ning random instances and dropping the oldest instance (similar to a LRU policy in a cache). We
borrowed uncertainty sampling from the active learning literature. Another idea we had was to try
to determine which features would help the model learn more, assuming varying step sizes [1][3][7].

For the second solution (dropping features), we evaluate policies such as dropping the least infor-
mative features, random features from an example and features that are very frequent in our current
examples in memory.

For the third solution (dimensionality reduction), we try hashing kernels [4] and combine it with the
previously mentioned policies.

In sum, we propose and provide an empirical analysis of different techniques for handling online
learning under delayed feedback. We evaluate our techniques in a click-prediction dataset from
KDD Cup 2012 (task 2).

1.1 Big Data specifics

The “Big Data” challenge of our project is two-fold: streaming N and very large N. The efficient
approaches for coping with the challenges we used were: Stochastic Gradient Descent, Adaptative
Gradient (AdaGrad), Hashing Kernels and our own proposed solutions to this particular problem.
As for computational systems, we implemented everything we used in C++ (no machine learning
libraries were used) and ran the experiments on our laptops.

2 Problem definition

Just like the Online Learning scenario, examples arrive in a streaming fashion. When an example
\( x^i \) arrives, the algorithm must make a prediction. The label \( y^i \) for example \( x^i \) arrives \( d^i \) steps after
\( x^i \), meaning that the algorithm will have to make \( d^i \) predictions before updating the model with \( y^i \).
As an example, let \( d^i = 4 \) be the label delay for example \( i \), and let \( d^i+1, d^i+2, d^i+3, d^i+4 \) be greater
than \( d^i \). Then the streaming data arrives in the following order (from left to right):

\[
x^i \rightarrow x^{i+1} \rightarrow x^{i+2} \rightarrow x^{i+3} \rightarrow x^{i+4} \rightarrow y^i \ldots
\]

Let \( M \) be the memory limit, in terms of number of features that can be kept in memory. If we have
3 examples in memory, with dimensionality 10, 20 and 30, the number of features in memory is 60.
We are assuming here that all features take up the same space in memory - a reasonable assumption
when dealing with datasets that only have real features. When a label \( y^i \) arrives, the algorithm
updates its model if \( x^i \) (or a subset of \( x^i \)) is still in memory. The instance \( x^i \) can be safely discarded
after \( y^i \) arrives.

The focus of this project is defining a policy \( P \) for handling the memory, such that the memory limit
\( M \) is maintained and the expected loss of an online learning algorithm is minimized. \( P \) can discard
or modify any example \( x^i \), in order to maintain the memory limit.

3 Policies that drop examples

In this section, we propose different strategies that handle memory by keeping a fixed number of
examples in memory and dropping the rest.
3.1 Baselines

As baselines, we present two strategies: **Random drop**, which drops an example from memory at random when the memory is full, and **LRU drop**, which drops the oldest example in memory - that is, drops \( x^i \) such that \( i < j \) \( \forall x^j \) in memory.

3.2 Uncertainty drop

The simplest and most commonly used query framework in active learning is *Uncertainty Sampling* ([6]). The intuition behind this framework is that instances for which the classifier is most unsure about provide the most amount of information, and thus they should be queried. This approach is often straightforward for probabilistic learning models: it usually just means sampling the instance whose posterior probability of being positive is nearest to 0.5.

In the delayed feedback model with constrained memory, this is equivalent to evicting the instance \( x^i \) that we are most certain about (and thereby keep the instances we are most uncertain about). This corresponds to evicting the instance \( x^i \) that solves the following optimization problem:

\[
\arg \max_{x^i} |P(Y = 1|x^i, w) - 0.5|
\]  

(1)

In order to optimize for predictions (instead of optimizing for learning), whenever we update our model (when we get a label for an example \( x^i \) that is in memory), we make predictions for all examples in memory and keep \( P(Y = 1|x^i, w) \) for all of them in a max-heap. With this, selecting the examples to drop when memory is full or adding new examples to the heap becomes \( O(\log(K)) \), where \( K \) is the number of examples in memory. Learning has an additional cost of \( O(K) \) predictions.

3.3 Ada drop

AdaGrad is an adaptative gradient method proposed by Ducchi et al ([7]). In AdaGrad, the step size (learning rate) is different for different features, such that frequently occurring features in the gradients get small step sizes and infrequent features get higher ones. The intuition is simple: the learner takes small steps for features that it has already received a lot of information about, but large steps for features that are rare and informative. One of the ways of implementing AdaGrad is setting the per-feature step size for feature \( i \) at each time step \( t \) to:

\[
\eta_{t,i} = \frac{\eta}{\sqrt{G_{t,ii}}}
\]  

(2)

Where each \( G_t \) is a diagonal matrix and \( G_{t,ii} = \sum_{t' = 1}^{t} g_{t',i}^2 \) - that is, the sum of the squares of the \( i \)th dimension of all the historical gradients.

One could extend the intuition of AdaGrad to examples, by saying that examples that have rare and informative features should get higher priority than examples that have common features. Formally, the utility of an example \( x^i \) is represented as:

\[
U(x^i) = \sum_{j \in x^i} \log(\eta_{t,j})
\]  

(3)

*Ada drop* is a policy that drops the example in memory with the least utility. Similar to what we do in Uncertainty drop, we keep a heap with the utility of each example in order to optimize for predictions and update the heap whenever the model is updated, by updating the appropriate \( G_t \)s.

4 Policies that drop features

Instead of dropping entire examples from memory, one could think of dropping features in order to reduce the size of the current instances in memory. In this section we present different strategies for this.
4.1 Baselines

We have two baselines for feature dropping. FR1 is a policy that selects an example at random, then selects a feature from that example at random and removes it. Whenever the memory is full, this process is repeated until there is enough memory. FR2 selects a feature at random and removes it from all examples. Once again, this process is repeated until there is enough memory.

4.2 Adaptive feature drop

Building on the intuition laid out in Ada drop, we devised two strategies for dropping features. Ada feature drop drops the feature $j$ with the smallest adaptative step size from all examples in memory. Random ada feature drop chooses a random example $x^t$ and then drops the feature $j$ from $x^t$ with the smallest step size. Both of these repeat until there is enough space in memory.

We keep a heap with the feature step sizes in memory, so that removing or adding a feature becomes $O(log(K))$.

4.3 Max frequency feature drop

Some of the previous feature dropping strategies drop features across every example in memory. However, it is intuitive that we would want to keep a diverse set of features in memory (in different examples), so that we would learn appropriate weights for each of the features. With this intuition in mind, Max frequency feature drop removes the feature that appears the most in memory from a random example that contains it. This preserves diversity of features in the examples in memory. Ideally, we would want to incorporate the notion of diversity with the adaptive feature drops, but it is a route that we did not pursue in this project.

5 Dimensionality Reduction

By reducing the dimensionality of the examples, we can keep more examples in memory. We implemented the unbiased hashing kernel proposed in [4] and set the hashing size $m$ to be smaller than the mean number of features in each example. The original feature space $X$ is transformed into a space $F$ with dimension $m$, by using two hash functions: $h \rightarrow \{0, ..., m - 1\}$, and $\xi : I \rightarrow \{+1, -1\}$. The hashed example becomes m-dimensional, where $\phi(x^t) = \sum_{j: h(j) = i} \xi(j)x^t_j$.

We defined the hash functions $h$ and $\xi$ as $h(j) = j \mod m$ and $\xi(j) = -I[j \mod 2 == 0] + I[j \mod 2 == 1]$, where $I[.]$ is an indicator function.

The strategies for dropping examples and features can also be applied to the hashed examples. The intuition is that the feature dropping strategies would work better with the hashed representations, since they ensure (with a small $m$) that most examples will be dense (as opposed to the sparse original examples), enabling feature dropping to free a lot of space in memory.

6 Experimental Results

We ran tests over a “delayed” version of the click-through rate dataset from 2012 KDD Cup Track 2. In order to include the notion of delay $d$ in the labeled examples, we drawn non-negative samples from a Normal distribution with $\mu = 0$ and $\sigma^2 = 1000$, i.e. $d \sim N(0, 1000)$. We induced different delays, but the results were very similar. We denote $M$ as the memory limit (number of instances that can be kept in memory).

We implemented Logistic Regression with Stochastic Gradient Descent and delayed regularization. We included three step size policies. (1), constant step size $\eta$. (2), $\eta/t$, following [8]. (3), AdaGrad [7]. The results for the $\eta/t$ step size were worse than the other two on all experiments, so we decided to omit them. Each example had, in average, 30 features. We used the hash size $m = 15$, such that we doubled the amount of examples that can be kept in memory (in average).

We split the training data into training and validation sets, in order to pick the step sizes for the various policies described above. We did not tune any parameters on the test data. Finally, we train
the model in the whole training set (one pass) and test on the test set. Our evaluation measure is the area under the ROC curve (AUC), which was the metric used in KDD Cup. It should be noted that we did not focus in optimizing the ROC results for the no-delay case - we did no feature engineering, and used only logistic regression. The point of our experiment is comparing the different memory management policies, and not being competitive in KDD cup track 2.

<table>
<thead>
<tr>
<th>NO DELAY</th>
<th>AUC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step Size Policy</td>
<td>No Hashing</td>
</tr>
<tr>
<td>Constant</td>
<td>0.668481254</td>
</tr>
<tr>
<td>AdaGrad</td>
<td>0.676014587</td>
</tr>
</tbody>
</table>

Figure 1: Results without delay

Figure 1 shows the results for the original dataset, under no delay, just for reference. In this case, no dropping occurs, as each example is always followed by its label. In this setting, AdaGrad performs slightly better than the constant step size policy and hashing with such a small $m$ (half the average of the features in each example) decreases the performance.

Figure 2: AUC vs. Memory Size without Hashing
In Figure 2, we present the results for the different strategies using the original data representation (no hashing). We vary the memory limit $M$. Figure 2(a) presents the results for the constant step size policy, while Figure 2(b) presents the results for the adaptive step size (AdaGrad). It is clear from the figure that increasing the memory limit leads to better accuracy. This is expected, as more examples / features can be kept in memory until the feedback arrives, and thus more learning happens. The different step size policies did not influence the results much, so both figures 2(a) and 2(b) look very similar.

As for the memory management strategies that drop examples, the first thing to note is that the only strategy that consistently outperforms the example dropping baselines is Ada drop. This is expected, as we are keeping in memory the examples we would learn the most from. Uncertainty drop does not work well. A possible reason is that it keeps a lot of redundant examples in memory. In contrast to the traditional active learning scenario, where the examples are sampled one at a time, here we would have to worry about keeping a diverse set of uncertain examples in memory, such that learning the label of one example does not render the other examples useless.

The feature dropping strategies did not work well, with the exception of Random Ada feature drop for small $M$. In retrospect, we saw that dropping features across all of the examples in memory led to dropping many infrequent features, such that the number of features that needed to be dropped in order to maintain the memory limit was high. This meant losing a lot of information. Random ada
feature drop did not suffer from this limitation. However, in hindsight, we think we need to figure out a way to compensate for the dropped features when learning.

Figure 3 shows the results for the different strategies applied to the hashed representation of the data. Figure 3(a) presents the results for the constant step size policy, while Figure 3(b) presents the results for the adaptive step size (AdaGrad). This time, it seems that AdaGrad yields slightly better results for most strategies - although the graphs are still very similar.

The example dropping strategies for the hashing case performs similarly (in comparison to its corresponding counterparts without hashing), but worse overall compared to the original representation. It is interesting to note that Ada feature drop, the worst strategy in the original representation is the best strategy in the hashed case, for $M < 1000$. By inspecting the results, our intuition for this behaviour is that under the hashed representation, the examples are mostly dense - such that when a feature is dropped across all examples in memory, a lot of space is freed. In the original representation, as mentioned before, a lot of features had to be dropped before enough space was freed from memory, and thus a lot of information was lost. It is also interesting to note that the results under the hashed representation were consistently worse than the results under the original representation, except for smaller values of $M$ (up to around 1000). We believe that we will be able to improve on the feature dropping strategies after we work on the theory behind it (i.e. further reasoning about the effect of dropping features in the gradients).

6.1 Work in progress

We consider it’s worth mentioning some extra work that has not yet ended up in any useful insight. We explored the 2010 PAN Wikipedia Vandalism dataset, one of the first applications we thought of, detecting vandalism in Wikipedia edits. We preprocessed the corpus and did some feature engineering work.

Table 1 enumerates some of the features we came out with. Others, such as Language (e.g. pronoun frequency and impact, biased and bad words), Reputation (e.g. user reputation, user’s country reputation) and Text (e.g. distance between the edited page and its previous version) based features are still work in progress. Once we compute them, we think we will be able to present thoughtful results on this dataset.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EditorRegistered</td>
<td>Whether the editor is a registered user or not</td>
</tr>
<tr>
<td>EditLength</td>
<td>Edit’s length</td>
</tr>
<tr>
<td>EditDigitRatio</td>
<td>Number of digits divided by the edit’s length</td>
</tr>
<tr>
<td>EditAlphaNumRatio</td>
<td>Number of alphanumeric characters divided by the edit’s length</td>
</tr>
<tr>
<td>EditUpperRatio</td>
<td>Number of capital characters divided by the edit’s length</td>
</tr>
<tr>
<td>EditLongestWordLength</td>
<td>Length of the longest word in the edit</td>
</tr>
<tr>
<td>EditAgreement</td>
<td>Reviewers that annotated the edit as a vandalism over the total reviewers</td>
</tr>
<tr>
<td>EditAvgDecisionTime</td>
<td>Average time that took the reviewers to make the decision on the edit</td>
</tr>
<tr>
<td>ReviewerId</td>
<td>Id of the reviewer. Each review can have more than one reviewer</td>
</tr>
<tr>
<td>ReviewerAge</td>
<td>Integer value indicating different ranges of a reviewer’s age</td>
</tr>
<tr>
<td>ReviewerReading</td>
<td>Integer representing how often the reviewer reads Wikipedia</td>
</tr>
<tr>
<td>ReviewerEditing</td>
<td>Integer representing how often the reviewer edits Wikipedia</td>
</tr>
<tr>
<td>ReviewerVandalizing</td>
<td>Indicates whether or not the reviewer has vandalized Wikipedia</td>
</tr>
<tr>
<td>ReviewerNoticing</td>
<td>Represents how often the reviewer notices vandalism on Wikipedia</td>
</tr>
</tbody>
</table>

Table 1: Wikipedia vandalism features

7 Conclusion and Future Work

Online learning has received increasing attention in the recent years. In particular, we focus on settings where the true label is discovered long after the prediction is made. To the best of our knowledge, we are the first to study the problem under memory constraints. We propose and evaluate some solutions for it, which consists of different mechanisms for dropping instances / features,
and reducing the dimensionality of the feature space. It seems that dropping instances based on their utility is the most promising strategy, unless the memory is heavily constrained. In such scenario, reducing the dimensionality and dropping features that have very small step sizes according to AdaGrad seems to be the best.

Currently, we are assuming data will actually be thrown off when dropped, but another line of future work would be exploring the scenario where data would be stored in disk, such that keeping data in memory just allows for faster learning.

Further exploration on different datasets, such as the Wikipedia vandalism one, will allow us to confirm our intuitions on the different policies.

We also considered putting together the different feature dropping strategies, instead of using them in isolation. Finally, an interesting line of research would be to explore how the theoretical bounds that already exist for online learning under delayed feedback [2] are changed when data is dropped.

Acknowledgments

Thanks to Tianqi Chen, Sameer Singh and Carlos Guestrin for their key contributions to the project. Finally, thanks to Emily Fox and the CSE547 teaching staff for their guidance and support during the quarter.

References


