Real-time Ranking with Concept Drift Using Expert Advice [*]

Hila Becker
hila@cs.columbia.edu

Marta Arias
marta@ccls.columbia.edu

Center for Computational Learning Systems
Columbia University
New York, NY 10027

ABSTRACT

In many practical applications, one is interested in generating a ranked list of items using information mined from continuous streams of data. For example, in the context of computer networks, one might want to generate lists of nodes ranked according to their susceptibility to attack. In addition, real-world data streams often exhibit concept drift, making the learning task even more challenging. We present an online learning approach to ranking with concept drift, using weighted majority techniques. By continuously modeling different snapshots of the data and tuning our measure of belief in these models over time, we capture changes in the underlying concept and adapt our predictions accordingly. We measure the performance of our algorithm on real electricity data as well as a synthetic data stream, and demonstrate that our approach to ranking from stream data outperforms previously known batch-learning methods and other online methods that do not account for concept drift.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; I.2.6 [**Artificial Intelligence**]: Learning

General Terms

Algorithms, Experimentation, Performance

Keywords

Concept drift, data streams, online learning, ranking

1. INTRODUCTION

In this paper we present a solution to the problem of generating ranked lists of a given set of items in a dynamic, real-time setting. Each item in our system is represented by a collection of attribute-value pairs, where the values can change over time. We apply inductive learning techniques to past data to construct predictive models so that when new data arrives, we can generate a new ranking that is accurate with respect to some phenomenon of interest. This application was inspired by the problem of generating real-time rankings of the components of an electrical system according to their susceptibility to impending failure based on the current status of the network and on information gathered during past failures. Specifically, in this application, we want components that are about to fail to appear toward the top of the current ranking so that engineers and operators can focus their attention on the most-at-risk components and can thus take appropriate measures to prevent failures or mitigate their consequences. We believe that our proposed solution can be useful for other applications and therefore present the problem and our solution in a generic form.

This problem presents several challenges. First, in order to build predictive models we use supervised machine learning algorithms. Most successful algorithms in practice do not handle dynamic data, hence we need to convert the incoming stream of data to a batch representation. We do this by using fairly standard window-based aggregations of the time-dependent data. We will not include any details on this data-assembly process since this is not the main focus of this paper. More importantly, in real applications one cannot assume that the underlying system is static. For example, the electricity system changes quite dramatically depending on the environmental conditions, and a computer network may behave differently at different times of the day or the week due to different usage patterns. Thus, models trained at a given time may perform well for a while but could cease to do so if there is a change of context in the underlying system. We assume that these changes of context happen and alter the behavior of the underlying system that we are trying to model, but when or why this happens is not known to the algorithm. This phenomenon is generally referred to in the literature as learning in the presence of concept drift [20, 16, 37, 22, 35, 42, 38, 8, 41, 43] and is the focus of this paper.

Our solution is based on the weighted majority algorithm [28], an online learning algorithm that given a set of "experts" (predictors that are used as black boxes) uses a combination of the experts' predictions with strong theoretical worst-case performance bounds. The weighted majority algorithm and its multiple variants [7, 13, 4, 27, 40] keep a weight or score for each expert, updating it as new feedback of the experts' performance becomes available. All these al-

[*] This work has been partly supported by a research contract from Consolidated Edison Company of New York

KDD'07, August 12–15, 2007, San Jose, California, USA.

gorithms are based on the plausible assumption that experts that have performed well recently are more likely to perform well in the future than experts that have had poor performance. Clearly, this assumption breaks down in presence of concept drift. In order to account for concept drift, we have extended this algorithm in several ways. First, the original formulation in [28] uses a fixed set of experts available in the beginning of the learning process. In our case, we periodically add new experts as we gather new training data over time. Therefore, we need to drop experts to avoid having to store and monitor an infinitely growing set of experts. Secondly, our algorithm uses rankers as underlying predictors, whereas all algorithms in the learning from experts' framework are designed for classification tasks. Finally, we have included several ad-hoc parameters to control when and how new experts are added, as it will be explained in Section 4.

The paper is organized as follows. Section 2 we briefly survey related work and situate our work in context; Section 3 presents the problem formally and introduces some notation and concepts; Section 4 describes OnlineRank, our proposed solution to the problem; Section 5 presents experimental results in the context of the feeder ranking application, and Section 6 concludes with a discussion of the system and some directions for future work.

2. RELATED WORK

As sensor, communication, computing and storage devices improve, the need of algorithms that can keep up with, make sense of and exploit the tremendous flow of information grows. It is therefore not surprising that the problem of dealing with concept drift in the context of learning from data streams is receiving so much attention recently, see e.g. [20, 16, 37, 22, 35, 42, 38, 8, 41, 43].

When one learns from a data stream, examples are presented one at a time. Hence, online learning algorithms are ideal for this task, as they are able to process examples incrementally as they arrive. Therefore a straightforward solution to learning from data streams is to use online learning algorithms directly (e.g. perceptron, winnow, etc. [2]). Unfortunately, many successful machine learning algorithms are *batch* in the sense that all examples need to be available as soon as the learning process starts. In order to use these algorithms in the context of learning from data streams one needs to transform them into online learning algorithms. This sometimes results in complicated methods that require $\Omega(1)$ time per example, although oftentimes speed-up heuristics and approximations exist that alleviate this problem. Examples of typically batch algorithms with online versions are decision trees [11, 20, 39], SVMs [10, 14, 30], and bagging and boosting [31, 32].

An alternative approach is what we refer to as the *wrapper* approach. In the wrapper approach, the batch machine learning algorithms are not modified but are used within a meta-learning layer that applies them to different subsets of the input data and combines them in multiple ways. The most common strategy in wrapper algorithms is to divide the input stream of data into subsets of sequential data (or "data windows"), and to repeatedly train models with the batch machine learning algorithm using only one or several contiguous windows of data at a time. The algorithms proposed to date within this approach differ in how a single or a combination of window-specific models are used to make future predictions. Within this group of algorithms we distinguish two broad categories: the ones that maintain a single model and replace it with newer ones to cope with the concept drift, and ensemble-based methods that keep a set of models and use combinations of the existing models to make predictions. An example of the former type is the work of Gama et al. [16, 36, 15, 17], where a process monitors the error rate of the current predictive model and whenever the performance significantly degrades, a new model is generated. Most of the existing work, however, falls into the category of ensemble-based wrappers. An early example of this is the system of [42], where in addition to dealing with concept drifting using a weighted ensemble of classifiers, they also introduce the notion of *hidden context*: assuming that the concept drift is due to a change of a hidden context, they store representations successful under different contexts so that their algorithm can take advantage of the already learned concepts if some previous context reappears. This meta-learning level should be clearly beneficial if the concept drift is cyclic and the same contexts re-appear. This meta-level idea of hidden context is exploited also in the system of [43]. Other ensemble-based algorithms that use averages or weighted averages for future predictions include [37, 38, 41, 8]. All these algorithms are similar in that they use heuristics to estimate the predictive accuracy of the ensemble models and use these to weigh models' predictions. Additionally, the system in [38] favors ensembles with diverse members since it has been empirically shown that having a diverse set of models benefits predictive accuracy. Additionally, the work of Klinkenberg et al. [24, 22, 23, 25, 35] describes and compares several strategies for dealing with concept drift such as selecting base learners adaptively, selecting window size adaptively, selecting examples adaptively, etc.

Our solution falls into the category of ensemble-based wrappers. The main difference is that instead of using heuristics or boosting-like combinations of underlying models we follow the framework of *learning from expert advice*. This framework has been thoroughly studied in the theory community, and it offers very strong performance guarantees. After the seminal paper of [28] several variants and extensions have been proposed [7, 13, 4, 27, 26, 40]. For example, [13] can handle the fact that experts may choose not to make predictions for a while. The algorithm in [4] works well if the concept drift is only among a small number of experts. Although this algorithm has been developed in the theory community, it has been applied to several domains successfully [3, 19, 9].

Our solution extends the existing algorithms in several ways: (1) it handles concept drift by continually adding new experts to the ensemble, (2) it has been adapted to the problem of ranking, and (3) it uses several ad-hoc parameters to control various aspects of the learning and meta-learning. The algorithm of [26] uses a similar idea of adding and dropping experts throughout the execution of the algorithm but differs from our approach in the type of base learners they use, in the set of tunable parameters, and in the fact that we are performing ranking instead of classification or regression.

3. THE REAL-TIME RANKING PROBLEM

Consider a network of interconnected nodes, where each node generates a stream of data representing its state information. We want to rank these nodes according to some event of interest, for instance susceptibility to failure in electrical grids or vulnerability to attack in computer networks. We assume that these events are observable and that we have access to past observations of events as well as state information of the nodes during the events, and so we are dealing with a supervised learning problem.

Any real-time ranking system should generate rankings that change over time in order to reflect the changing state of the network. Hence, we assume that $\textsc{Ranking}(t)$ is a ranked list of nodes in the network at time t, and $\textsc{Rank}(n, t)$ is the rank of node n at time t. How quickly the system should update its rankings will be driven by how quickly new data becomes available or by the requirements of the actual application. The system is effectively guessing that nodes at or near the top of the ranking are more likely to be involved in the next occurrence of the event of interest than nodes ranked low in the list. Naturally, we want these guesses to be accurate and therefore we use the following natural performance metric to measure the quality of the rankings over time.

We use pairs (n, t) to represent the fact that a node n has been involved in an event at time t. Let $\textsc{Events}(start, end)$ be the set of event pairs $\{(n_i, t_i)\}_i$ listing all the events observed between *start* and *end* in a network with N nodes. Assume there are K events between the start and end dates. To evaluate the performance of a system generating time-varying rankings $\textsc{Ranking}(t)$ we use the formula

$$\textsc{performance}(\{(n_i, t_i)\}_i) = 1 - \frac{1}{K} \sum_{i=1}^{K} \frac{\textsc{Rank}(n_i, t_i)}{N} \quad (1)$$

For example, in the context of electrical component failures in an electricity grid, suppose that the network consists of 943 components and that 3 of them fail while ranked at positions 100, 231, and 51. The corresponding performance is $1 - \frac{1}{3}\left(\frac{100+231+51}{943}\right) \approx 0.865$. Notice that the higher up in the ranking a failure is, the better (higher) the performance is; 0.5 is the expected value if rankings are random.

In order to produce time-varying rankings of the nodes, we must capture snapshots of the continuously streaming data and assemble training and test sets. We assume that the set of attributes for each node consists of static data, which changes rarely, and dynamic data that continuously changes over time. It is the dynamic data that requires us to constantly adapt our models to reflect the latest state of the system.

Training data and ranking models. Training datasets are assembled with respect to date intervals, capturing the state of (selected) nodes in the network during the specified period. We assume the system has access to the procedure $\textsc{TrainingDataset}(start, end)$ that operates as follows: for each event within the given time interval, it samples nodes involved in the event and nodes that were not involved, collects their data from the data stream at the time of the event and labels them according to whether the nodes were involved in the event or not. [1] The ranking models are obtained by applying machine learning ranking algorithms to training data; these models attempt to make generalizations about how the static and dynamic attributes relate to the labels seen in the training datasets.

Test data and rankings. Test or *snapshot* data at a given time t is compiled by obtaining the readings of all dynamic attributes at time t for each node, together with the known static values. The procedure $\textsc{Snapshot}(t)$ assembles test datasets. Given a ranking model m_i, the ranking constructed by applying m_i to the snapshot at time t is denoted by $\textsc{Ranking}_i(t) = m_i(\textsc{Snapshot}(t))$.

The problem. Given an interconnected set of nodes in a network with associated data streams reflecting the state of each node over time, produce a ranking of the nodes in real-time such that the performance over time as given by Formula (1) with respect to some observable event is maximized.

4. DESCRIPTION OF THE ALGORITHM

Our online ranking approach consists of multiple batch-trained models that we refer to as experts, and a meta-model which determines how to combine the expert predictions. We use training datasets from different time intervals and a diverse base of learning algorithms to create a diverse base of experts. To assess the quality of experts' predictions we use Formula (1) on each expert's ranking and use this as feedback to the meta-model. While the experts are used to model different states of the system, the meta-model is used to determine which captured states are better at predicting the current state of the system. The meta-model is an adaptive (online) learning model, meaning that the training and testing is performed by feeding in each example, receiving the model's prediction, and altering the model based on the loss of the experts with respect to the example. The set of labeled examples for the meta-model in our case consists of the nodes that participate in the event of interest. For example, if we are interested in generating a ranking of electrical components according to their failure susceptibility, our labeled examples would consist of the electrical components that failed. The final ranking output of the meta-model is a weighted average of experts' rankings, where the weight of each expert translates to the meta-model's measure of belief in that expert.

Our online ranking algorithm is based on the principle of learning from expert advice, and draws on ideas from the Continuous Weighted Majority algorithm [28]. To cope with concept drift, new models, trained with the most recent data, are periodically added to the existing ensemble. In order to avoid growing an infinitely large ensemble, models are removed according to a function of their age and performance; the age of a model is the number of days since its creation.

Periodically, we train and add new models to the current ensemble. A parameter f determines how frequently this should happen, i.e., new models will be added every f iterations. When new models are created, we assign each of them a weight to be used as an individual performance measure. We add these models to the ensemble of experts used by the algorithm in making its predictions. The expert ensemble is then presented with a set of items to rank and each expert makes a separate prediction. The algorithm combines these predictions by ranking the items according to their weighted average rank. It then receives the true ranking of the items and updates the weights of the experts in the ensemble. The weight update function is similar to the one discussed in [28], where the weight is multiplied by a function of the loss. The loss of each expert in the ensemble is

[1] To be precise, we have experimented with alternative ways of assembling datasets but for brevity we will omit an explanation of these in this paper.

a measure of its performance, relative to the other experts.

There are several input parameters that can be used to tune the performance of the algorithm. The learning rate $\beta \in [0, 1)$ is used in the weight update function to adjust the fraction by which the weights are reduced. A larger value of β corresponds to a slower learning rate, making the algorithm more forgiving to experts that make a mistake by reducing their influence by a smaller fraction. We also use a parameter B (or *budget*) to limit the number of models that the algorithm can keep track of at each iteration. Since we do not use a static set of experts as in the traditional weighted majority approach, we have to make sure that our ensemble does not grow infinitely when we add new models. We can also adjust the number of models that the algorithm uses for prediction. In the traditional approach, the advice of all experts in the ensemble is combined to make the final prediction. By using a parameter E for the number of predictors, we can try to further enhance the performance, combining the advice of top performing experts only.

Since we add and remove models from our expert ensemble throughout the algorithm, additional parameters are introduced. Let n be the number of new models added to the ensemble. This parameter n depends on how many machine learning algorithms we use (2 in our experiments) and on how many training sets we assemble (we vary the training data windows, currently set to 7 and 14 days). When these new models are added, they are assigned an initial weight w_{new}. This weight can be also adjusted to reflect our trust in these new models, and should be relative to the weights of the existing models in the ensemble. We use a parameter p that determines what weight to assign new models as a percentile in the range $[w_{min}, w_{max}]$ for the minimum and maximum weights of the existing models. We also need to decide what models to drop when the ensemble size grows larger than the budget B. We order the experts according to a function of their performance and age, where $\alpha \in (0, 1)$ is a parameter used to set the exponential decay by age. Pseudocode of our online ranking algorithm can be found in Figure 1. In the algorithm, we keep a set of current models $\mathcal{M}$. Every f iterations, we add n new models and if necessary remove the worst models that exceed our budget B. The weights of models w_i are computed according to the update formula $w_i = w_i * \beta^{l_i}$, where l_i is the *loss* of model m_i in the last event. The loss of model m_i is computed according to its relative performance with respect to the other models in $\mathcal{M}$ using the formula $l_i = \frac{s_{best} - s_i}{s_{best} - s_{worst}}$. This loss function normalizes the losses of experts to the range $[0, 1]$ and could be used with any (even unbounded) performance metric, which in our case is given by formula (1).

5. RESULTS

In this section, we present various experiments with the goal of studying and evaluating the online ranking algorithm. We perform our experiments on real electricity data and a synthetically generated data stream. As a baseline, we compare our performance to that of a commonly used online learning algorithm as well as a batch-trained model.

In our experiments we use as “experts” two types of learning algorithms: *SVMs* and *MartiRank* [18], a ranking algorithm based on the boosting framework in [29]. To obtain a ranking from the outputs of the *SVMs* we simply use the margin scores to sort the examples appropriately. Since rankers are used as black boxes we could use any other ranking algorithm e.g. [12, 6, 34].

```
OnlineRank($B, E, \beta, \alpha, p, n, f$)
$\mathcal{M} \leftarrow \{\}$
while ($true$)
do at time $t$
    Ranking$_i(t) \leftarrow m_i($Snapshot$(t))$ for $m_i \in \mathcal{M}$
    $\mathcal{E} \leftarrow$ E top-scoring models according to weights $w_i$
    Ranking$(t) \leftarrow$ WAvg$(\{w_i \times$ Ranking$_i(t) | m_i \in \mathcal{E}\})$
    if new event$(x, t)$
      then $s_i =$ PERFORMANCE$_{m_i}(x, t)$ for $m_i \in \mathcal{M}$
           $s_{best} \leftarrow min(s_1, \ldots, s_{|\mathcal{M}|})$
           $s_{worst} \leftarrow max(s_1, \ldots, s_{|\mathcal{M}|})$
           for $m_i \in \mathcal{M}$
           do $l_i = \frac{s_{best} - s_i}{s_{best} - s_{worst}}$
              $w_i = w_i * \beta^{l_i}$
    if no models generated in $f$ iterations
      then train $n$ models $m_{|\mathcal{M}|+1}, \ldots, m_{|\mathcal{M}|+n}$
           $w_{new} \leftarrow$ PERCENTILE$(p, \{w_1, \ldots, w_S\})$
           $w_{S+1}, \ldots, w_{S+n} \leftarrow w_{new}$
           $\mathcal{M} \leftarrow \mathcal{M} \cup \{m_{|\mathcal{M}|+1}, \ldots, m_{|\mathcal{M}|+n}\}$
           if $|\mathcal{M}| > B$
             then remove $|\mathcal{M}| - B$ worst
                  models according to
                  $q_i = w_i * \alpha^{age_i}$
    normalize weights $w_i$
```

Figure 1: Pseudocode for our online ranking algorithm.

5.1 Experiments with Electricity Data

Our first set of experiments are performed on data collected from an electricity distribution system. In particular, we examine attributes of electrical feeders[2] with the intent to rank them according to their failure susceptibility. The electricity data is very diverse, not only in nature but also in location, type, format, etc. A significant amount of work has been devoted to understanding, processing and merging this data into attribute-value vector datasets that can be used by standard machine learning algorithms. The data used for these experiments ranges from June 2005 to December 2006. The main input data sources are:

- **Static Data:** attributes comprising this category are mainly physical characteristics of the feeders such as age and composition of each feeder section as well as attributes reflecting connectivity and topology of the network. These values rarely change.
- **Dynamic Data:** attributes in this category do change over time. We distinguish two types:
  - **Outage data:** lists all the failures happened starting in 2001 up to date. This data is updated daily.
  - **Stress data:** reflects the state of a feeder and its transformers. We obtain new readings of this data in intervals of roughly 20 minutes, accumulating at a rate of several hundred megabytes per day in real-time.

[2] Feeders are cables, usually 10-20 km long, over which electricity is sent at mid-level voltage (tens of thousands of volts) from substations to local transformers.

We compare the performance of our online ranking algorithm against two separate baseline experiments. The first one uses the perceptron algorithm [33]. We chose the perceptron particularly because it is an online method that learns directly from the labeled feeder examples, as opposed to using a meta-learning approach like ours. To generate a ranking from the perceptron classification predictions, we sort the feeders according to their (signed) distance from the boundary in descending order.

The second experiment involves using a single batch-trained model throughout the whole run. The batch model is trained using a ranking algorithm based on the boosting framework in [29], which proved to be the best performing batch learning method on this specific dataset [18]. The performance of these baseline approaches can be seen alongside our online ranking method in Figure 2 for the summer of 2005 and Figure 3 for the winter of 2006. For both the summer and the winter months our online ranking approach outperforms both the perceptron and the batch model.

The default parameters used in our system are: learning rate $\beta = 0.9$, budget $B = 50$, ensemble size $E = 10$, new model frequency $f = 7$, age decay $a = 0.99$, and new weight percentile $p = 0.7$.

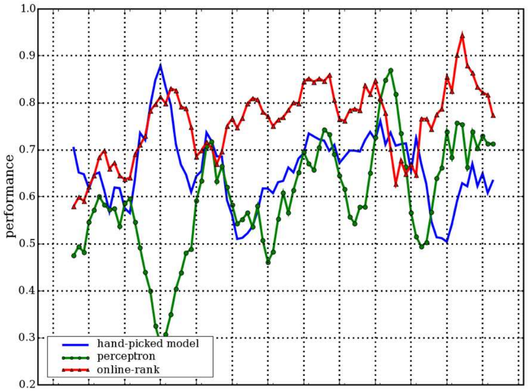

Figure 2: OnlineRank daily performance over baseline methods - Summer 2005 (6/1/2005-8/31/2005)

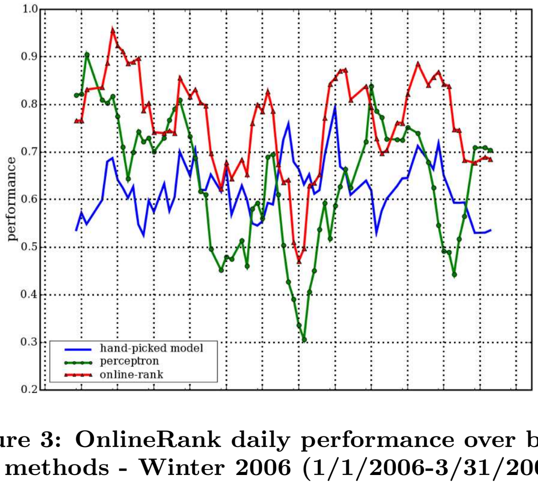

Figure 3: OnlineRank daily performance over baseline methods - Winter 2006 (1/1/2006-3/31/2006)

In order to optimize the online algorithm's performance, we examine the effects of varying the value of a single parameter while keeping the rest constant. In doing so, we can observe the change in performance (or lack thereof) associated with each parameter setting, and determine the optimal input values to the algorithm.

Figure 4 shows the performance of the system during the summer of 2006 for different values of the budget B, which corresponds to the maximum number of experts that the algorithm can select from to make its prediction. We can observe that the performance of the system is directly correlated with the number of existing experts. Intuitively, when there is a larger pool of models to select from, we have a higher chance of selecting the top performing models amongst them, especially if the predicting ensemble size E is small.

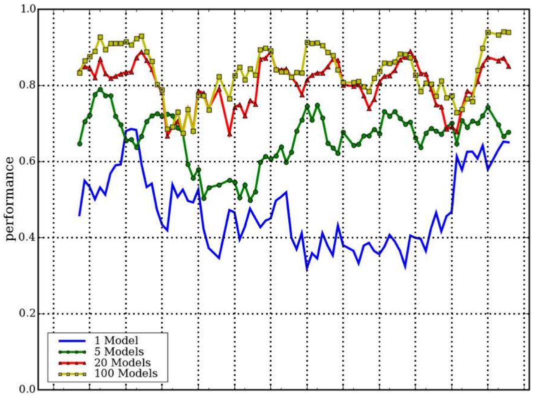

Figure 4: OnlineRank daily performance with varying budget size (B), here $E = 1$ - Summer 2006 (6/1/2006-8/31/2006)

Another parameter that we are interested in observing is the new models' weight percentile p, which determines our degree of belief in the incoming models. The performance of the online ranking system with varying weight percentile during the summer of 2006 can be seen in Figure 5. Although the difference in performance is more subtle than that of the budget variations, it is still clear that there are values that lead to better performance than others. Assigning new models the lowest weight in the range is understandably a poor choice since new models are trained with the latest snapshot of the data, thus carrying an up to date information about the system which should help increase the accuracy of the prediction. On the other hand, assigning too high a weight may force the system to use the newest models always, which may not be a good choice if an older model has been found to work the best. Notice that the performance of the algorithm for the 90th and 70th new weight percentile setting is almost identical.

To understand the reason for this results, we performed a weight analysis on the experts used for prediction. We found that more than 80% of the total weight is taken up by 10 equally weighted models. This was not the case for the algorithm's run with new models' weight in the 10th, 30th and 50th percentiles where the models were more diverse in terms of their weight and duration of stay in the top E. We can attribute the prevalence and consistency of the top 10 models in the case of 90th and 70th new weight percentile values to the fact that assigning low weights to the incoming models prevented them from rising to the top of the ensemble, making them more likely to be dropped. The weight similarity among the top E models may imply that although these models are different in the time and method of training they may be similar in their predictions. We plan to explore a measure of model similarity in future work in order to gain insight into the system's behavior.

From our experiments, we found that assigning new models a weight value in the middle of the range generally yields better performance. This can be seen by observing the performance of the 50th percentile new weight value in Figure 5, relative to the other percentile values.

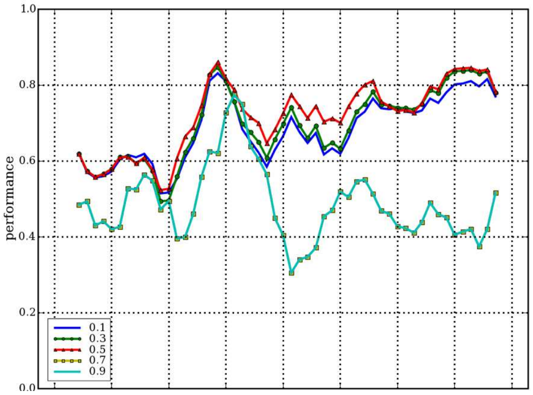

Figure 5: OnlineRank daily performance with varying new weight percentile (p) - Summer 2006 (6/1/2006-8/31/2006)

The maximum ensemble size E is the parameter that controls how many models out of the available pool contribute to the final prediction. While in theory we should combine the advice of all available experts, in practice it may be useful to ignore some of the lower-weighted experts for the purpose of prediction but still keep them in the pool in case they redeem themselves. For instance, in cases where at any given time there are only a few experts that have an accurate model of the system, it may be useful to combine the advice of these top E without the majority of the experts that would weigh down the performance. Figure 6 shows the performance of the online ranking system during the summer of 2005, using different values for the predicting ensemble size E.

Finally, we have experimented with the learning rate β of the ranking algorithm. Figure 7 shows the performances obtained for different settings of this parameter. Theoretically, the learning rate should affect the performance by controlling the fraction by which weights are reduced. However, our experiments show that varying the learning rate has a minor impact to the overall performance change. To understand why this is the case, we examine the weights of the top soring models for each setting of β. We plot the weights of the top E experts per day as a stacked bar, where the total y-value corresponds to the fraction of the weights of models in the budget B that are also in the predicting ensemble E. Each color uniquely identifies a particular model so we can observe which models are dominant and how long they stay in the ensemble. Figure 8 shows the weight distribution for learning rate setting $\beta = 0.1$ and Figure 9 shows the weight distribution for $\beta = 0.9$. The weight contribution of the two most dominant ensemble models in both plots explains the similarity of the results.

Please note that we chose this particular example since it is easier to interpret and render. In most of our weight analysis plots there is a lot more diversity of models and movement, although similar patterns still exist between the plots for the algorithm's runs with different learning rate settings. This results is somewhat surprising and we plan to experiment with different weight update functions in future work.

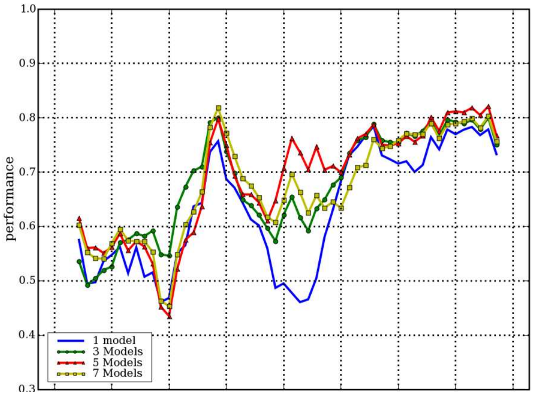

Figure 6: OnlineRank daily performance with varying ensemble size (E) - Summer 2005 (6/1/2005-8/31/2005)

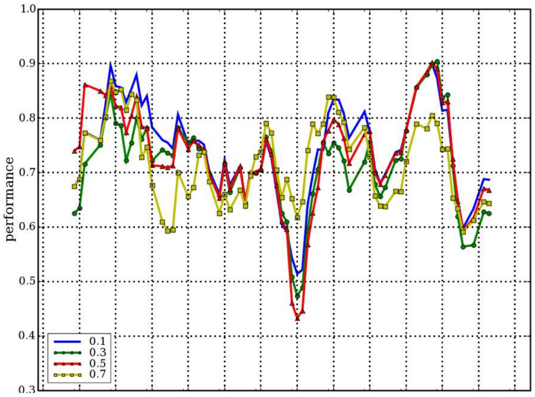

Figure 7: OnlineRank daily performance with varying learning rate (β) - Winter 2006 (1/1/2006-3/31/2006)

5.2 Experiments with Synthetic Data

In order to gain a deeper understanding of the behavior of our system, we have "manually" generated datasets for which the concept drift is controlled. The idea is that we pick models at random and use them to generate events. A parameter $c \in [0, 1]$ controls the rate of change of the models used to generate the events. That is, with probability c we will substitute the event-generating model, otherwise we will continue to use the current one. The data streams used in these experiments are the same as in the experiments with electricity data, the only difference being in the events that occur over time. As a consequence, the labels of training and test sets will differ from the experiments in Section 5.1.

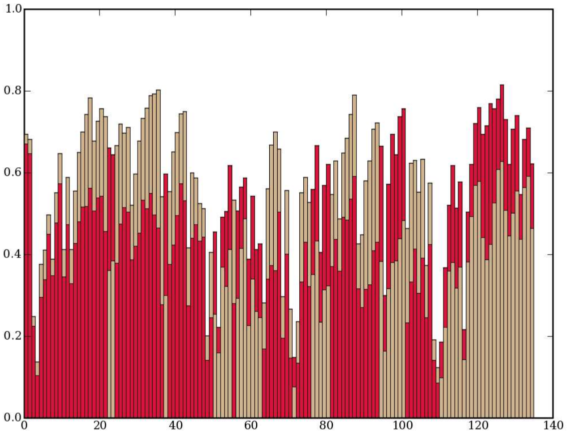

Figure 8: Ensemble models' weights using learning rate $\beta = 0.1$ for Winter 2006 (1/1/2006-3/31/2006)

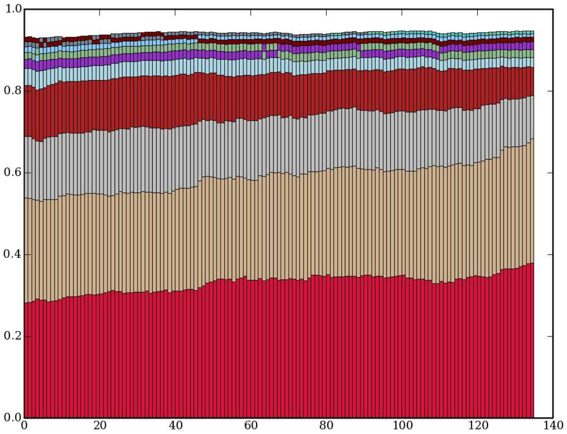

Figure 9: Ensemble models' weights using learning rate $\beta = 0.9$ for Winter 2006 (1/1/2006-3/31/2006)

To generate an event for a given ranking model m at time t, we generate snapshot data and use the model to produce a ranking $\textsc{Ranking}_m(t)$. Another parameter d (for *depth percentile*) introduces noise by picking the node event uniformly at random among the top-ranked $d * |\textsc{Ranking}_m(t)|$ nodes of $\textsc{Ranking}_m(t)$.

Figures 10 and 11 show the performance of our system vs. the online learning algorithm perceptron under different settings of the concept change rate c and depth percentile noise d. Our system dominates although the difference is not as overwhelming as in the previous section.

We have also experimented with an alternative data generation approach. We generate random hyperplanes and random examples in euclidean space and label them according to the hyperplane. In each iteration we tweak the hyperplane to simulate concept drift. Using this approach we have observed that perceptron consistently outperforms our system. In hindsight, this is not surprising since we are using hyperplanes as concepts, namely the hypotheses used by perceptron whereas our system explores a completely different hypothesis space, thus favoring perceptron. The reader should note that in practice it is unlikely that real-world data is linearly separable.

6. CONCLUSIONS AND FUTURE WORK

We have presented a real-time algorithm that generates time-varying rankings of nodes in a network based on data streams associated with each node. We have demonstrated its validity in the real-world scenario where one wants to generate rankings of components based on their susceptibility to impending failure. In fact, this is the context in which the algorithm was originally developed. Our system outperforms a standard online learning algorithm using synthetic data although results are not as dramatic. We are still investigating alternative ways of generating data in realistic ways to further understand the behavior of our system. Additionally, we are devising visualization methods for the internal state of our algorithm such as the weight plots of Figures 8 and 9. Such visualization methods should not only give us a better understanding of the system's behavior and performance but also give us insights into the underlying system itself (for example, factors or causes of failure in the electricity grid).

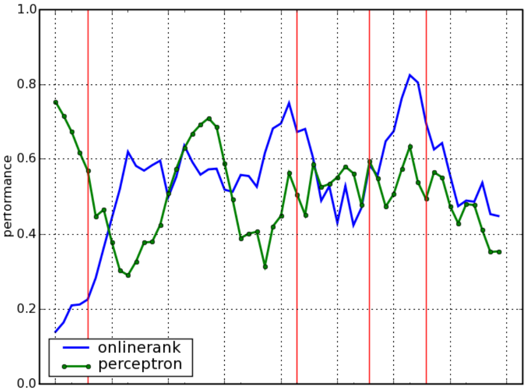

Figure 10: OnlineRank vs. perceptron on synthetic data for concept drift rate $c = 0.1$ and depth percentile noise $d = 0.01$. Vertical bars mark a change of concept.

There are several ways in which we could improve the performance of our system. We are planning to include an engine for the detection of concept drift, so that instead of periodically adding new experts we will only add when a change is detected [16, 21]. We also want to control for the diversity of the ensemble. Currently we do not check whether we have multiple copies of the same model in the ensemble. If we are about to add a new model, we should make sure it is not already present in the ensemble. This should also ensure that the ensemble is diverse, which has been found to help predictive performance [38].

Another direction of future work is to exploit re-occurring scenarios. If there are cyclical patterns in the behavior of the system under consideration, information learned during one phase can be re-used when the system re-enters that phase again. For example, in the electricity data applica-

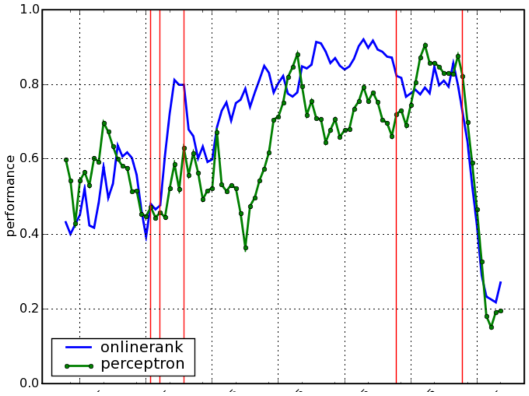

Figure 11: OnlineRank vs. perceptron on synthetic data for concept drift rate $c = 0.05$ and depth percentile noise $d = 0.1$. Vertical bars mark a change of concept.

tion of section 5.1, load demand increases greatly when the temperature is high, and hence the stresses on the grid in the summer differ greatly from those in the winter. It is natural to think that experts that have performed well in the summer are likely to perform well in the upcoming summer. We want to extend our online algorithm to include a meta-learning layer that is able to discover such correlations so that the weight update can be further informed by the information of results in previous years. In other words, the *hidden context* discussed in [42] would correspond to a different environmentally-driven condition. Since these different contexts re-appear with periodicity of one year, we should be able to take advantage of what has been learned in the past. We have accumulated enough data to start to look at such patterns. Examples of other systems that exploit this meta-layer are [42, 43].

Finally, even though in our application the items we need to rank are interconnected and form a graph, we do not make use of this structure and use a feature-vector representation instead (although some of our static features model the underlying topology of the network). The area of relational learning deals with problems where examples are interconnected and we should generalize our algorithm by using learning methods that can exploit the network structure [5, 1]. An obvious way to incorporate this style of learning into our algorithm would be to use such algorithms as experts, although a more challenging problem would be to incorporate this information directly at the meta-learning level in the online learning algorithm.

Acknowledgments

We gratefully acknowledge the support provided by Consolidated Edison Company of New York. We wish to thank all of those who gave so generously in time, advice, and effort. Additionally, a number of Columbia faculty, students, staff, and researchers contributed towards this effort: Phil Gross, Albert Boulanger, Chris Murphy, Luis Alonso, Joey Fortuna, Gail Kaiser, Roger Anderson, and Dave Waltz.

7. REFERENCES

[1] A. Agarwal, S. Chakrabarti, and S. Aggarwal. Learning to rank networked entities. In *KDD*, pages 14–23, 2006.

[2] A. Blum. On-line algorithms in machine learning. In *Online Algorithms*, pages 306–325, 1996.

[3] A. Blum. Empirical support for winnow and weighted-majority algorithms: Results on a calendar scheduling domain. *Machine Learning*, 26(1):5–23, 1997.

[4] O. Bousquet and M. K. Warmuth. Tracking a small set of experts by mixing past posteriors. *Journal of Machine Learning Research*, 3:363–396, 2002.

[5] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks*, 30(1-7):107–117, 1998.

[6] C. J. C. Burges, T. Shaked, E. Renshaw, A. Lazier, M. Deeds, N. Hamilton, and G. N. Hullender. Learning to rank using gradient descent. In *ICML*, pages 89–96, 2005.

[7] N. Cesa-Bianchi, Y. Freund, D. Haussler, D. P. Helmbold, R. E. Schapire, and M. K. Warmuth. How to use expert advice. *J. ACM*, 44(3):427–485, 1997.

[8] F. Chu and C. Zaniolo. Fast and light boosting for adaptive mining of data streams. In *Proceedings of the Pacific-Asia Knowledge Discovery and Data Mining Conference*, pages 282–292, 2004.

[9] V. Dani, O. Madani, D. Pennock, S. Sanghai, and B. Galebach. An empirical comparison of algorithms for aggregating expert predictions. In *UAI*, 2006.

[10] C. Domeniconi and D. Gunopulos. Incremental support vector machine construction. In *ICDM*, pages 589–592, 2001.

[11] P. Domingos and G. Hulten. Mining high-speed data streams. In *Proceedings of ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 71–80, 2000.

[12] Y. Freund, R. D. Iyer, R. E. Schapire, and Y. Singer. An efficient boosting algorithm for combining preferences. *Journal of Machine Learning Research*, 4:933–969, 2003.

[13] Y. Freund, R. E. Schapire, Y. Singer, and M. K. Warmuth. Using and combining predictors that specialize. In *STOC*, pages 334–343, 1997.

[14] G. Fung and O. L. Mangasarian. Data selection for support vector machine classifiers. In *Proceedings of ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 64–70, 2000.

[15] J. Gama and G. Castillo. Learning with local drift detection. In *ADMA*, pages 42–55, 2006.

[16] J. Gama, P. Medas, G. Castillo, and P. P. Rodrigues. Learning with drift detection. In *SBIA*, pages 286–295, 2004.

[17] J. Gama, P. Medas, and P. P. Rodrigues. Learning decision trees from dynamic data streams. In *SAC*, pages 573–577, 2005.

[18] P. Gross, A. Boulanger, M. Arias, D. L. Waltz, P. M. Long, C. Lawson, R. Anderson, M. Koenig, M. Mastrocinque, W. Fairechio, J. A. Johnson, S. Lee, F. Doherty, and A. Kressner. Predicting electricity distribution feeder failures using machine learning

susceptibility analysis. In *AAAI*, 2006.

[19] D. P. Helmbold, D. D. E. Long, T. L. Sconyers, and B. Sherrod. Adaptive disk spin-down for mobile computers. *MONET*, 5(4):285–297, 2000.

[20] G. Hulten, L. Spencer, and P. Domingos. Mining time-changing data streams. In *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 97–106, San Francisco, CA, 2001. ACM Press.

[21] D. Kifer, S. Ben-David, and J. Gehrke. Detecting change in data streams. In *VLDB*, pages 180–191, 2004.

[22] R. Klinkenberg. Meta-learning, model selection, and example selection in machine learning domains with concept drift. In *LWA*, pages 164–171, 2005.

[23] R. Klinkenberg and T. Joachims. Detecting concept drift with support vector machines. In P. Langley, editor, *Proceedings of ICML-00, 17th International Conference on Machine Learning*, pages 487–494, Stanford, US, 2000. Morgan Kaufmann Publishers, San Francisco, US.

[24] R. Klinkenberg and I. Renz. Adaptive information filtering: Learning in the presence of concept drifts, 1998.

[25] R. Klinkenberg and S. Rüping. Concept drift and the importance of example. In *Text Mining*, pages 55–78. Jürgen Franke, Gholamreza Nakhaeizadeh, and Ingrid Renz, 2003.

[26] J. Z. Kolter and M. A. Maloof. Using additive expert ensembles to cope with concept drift. In *ICML*, pages 449–456, 2005.

[27] Y. Li and P. M. Long. The relaxed online maximum margin algorithm. *Machine Learning*, 46(1-3):361–387, 2002.

[28] N. Littlestone and M. K. Warmuth. The weighted majority algorithm. In *IEEE Symposium on Foundations of Computer Science*, pages 256–261, 1989.

[29] P. M. Long and R. A. Servedio. Martingale boosting. In *COLT*, pages 79–94, 2005.

[30] O. L. Mangasarian and D. R. Musicant. Large scale kernel regression via linear programming. *Machine Learning*, 46(1-3):255–269, 2002.

[31] N. Oza and S. Russell. Online bagging and boosting. In *Artificial Intelligence and Statistics 2001*, pages 105–112. Morgan Kaufmann, 2001.

[32] N. C. Oza and S. J. Russell. Experimental comparisons of online and batch versions of bagging and boosting. In *Proceedings of ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 359–364, 2001.

[33] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(1):386–407, 1958.

[34] C. Rudin. Ranking with a p-norm push. In *COLT*, pages 589–604, 2006.

[35] M. Scholz and R. Klinkenberg. An ensemble classifier for drifting concepts. In *Proceedings of the Second International Workshop on Knowledge Discovery in Data Streams*, pages 53–64, 2005.

[36] M. Severo and J. Gama. Change detection with kalman filter and cusum. In *Discovery Science*, pages 243–254, 2006.

[37] K. Stanley. Learning concept drift with a committee of decision trees, 2001.

[38] W. N. Street and Y. Kim. A streaming ensemble algorithm (sea) for large-scale classification. In *Proceedings of ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 377–382, 2001.

[39] P. E. Utgoff, N. C. Berkman, and J. A. Clouse. Decision tree induction based on efficient tree restructuring. *Machine Learning*, 29(1):5–44, 1997.

[40] V. G. Vovk. Aggregating strategies. In *Proceedings of the Conference on Computational Learning Theory*, pages 371–386, 1990.

[41] H. Wang, W. Fan, P. S. Yu, and J. Han. Mining concept-drifting data streams using ensemble classifiers. In *Proceedings of ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 226–235, 2003.

[42] G. Widmer and M. Kubat. Learning in the presence of concept drift and hidden contexts. *Machine Learning*, 23(1):69–101, 1996.

[43] Y. Yang, X. Wu, and X. Zhu. Combining proactive and reactive predictions for data streams. *Data Mining and Knowledge Discovery*, 13:261–289, 2006.