

---

# Incremental Learning in SwiftFile

---

Richard B. Segal  
Jeffrey O. Kephart

RSEGAL@WATSON.IBM.COM  
KEPHART@WATSON.IBM.COM

IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598

## Abstract

SwiftFile is an intelligent assistant that helps users organize their e-mail into folders. SwiftFile uses a text classifier to predict where each new message is likely to be filed by the user and provides shortcut buttons to quickly file messages into one of its predicted folders. One of the challenges faced by SwiftFile is that the user's mail-filing habits are constantly changing — users are frequently creating, deleting and rearranging folders to meet their current filing needs. In this paper, we discuss the importance of incremental learning in SwiftFile. We present several criteria for judging how well incremental learning algorithms adapt to quickly changing data and evaluate SwiftFile's classifier using these criteria. We find that SwiftFile's classifier is surprisingly responsive and does not require the extensive training that is often assumed in most learning systems.

## 1. Introduction

SwiftFile is an intelligent assistant that helps users organize their e-mail into folders (Segal & Kephart, 1999). Using a text classifier that dynamically adjusts to the user's mail-filing habits, SwiftFile predicts for each incoming message the three folders that it deems most likely to be chosen by the user as destinations. The user may then file a message into one of these predicted folders by clicking on a shortcut button representing the desired folder. If none of SwiftFile's predictions are correct, the user can simply resort to the usual method for filing messages.

Previously, SwiftFile's classification accuracy was assessed using standard static evaluation techniques. However, static evaluation is not entirely satisfactory. The e-mail environment is highly dynamic, with folders and messages constantly being created, destroyed,

and reorganized. A classifier that is useful today may be much less so a month from now.

Static analysis fails to address several key questions about how well a dynamic learner like SwiftFile is likely to perform in a dynamic environment.

1. How quickly can SwiftFile adapt to new users?

SwiftFile can bootstrap itself by learning from a user's previously-filed messages when it is first installed. However, if the user is new to e-mail and does not have any previously-filed messages, there is no data available to initialize the classifier. How many messages does the user have to file before SwiftFile can make useful suggestions?

2. How well does SwiftFile track the changing environment of established users?

An important and frequent source of change is the creation of a new folder. A new folder may reflect either a change in the stream of mail received by a user or a change in that user's filing habits. Initially, new folders contain only a handful of messages. How many messages will SwiftFile have to see in a new folder before it can start suggesting which messages should be placed there?

3. How important is incremental learning?

Some systems designed to help users prioritize or organize their e-mail do not use incremental learning (Maes, 1994; Payne & Edwards, 1997). The authors of these systems address the need for adaptation by recommending retraining the system from scratch, perhaps on a daily basis. Is retraining a classifier on a daily basis sufficient in a highly-dynamic environment like e-mail?

After a brief description of SwiftFile and a brief review of results obtained from a standard static evaluation, we develop a variety of dynamic evaluation measures inspired by these questions and use them to assess the performance of SwiftFile's incremental learning algo-

rithm. We then conclude with a brief discussion, including a comparison to related work.

## 2. SwiftFile

SwiftFile is an add-on to Lotus Notes that helps users file their e-mail into folders. Figure 1 shows SwiftFile in action. SwiftFile places three *MoveToFolder* shortcut buttons above each message. The shortcut buttons allow the user to quickly move the message into one of the three folders that SwiftFile predicts to be the message’s most-likely destinations. The buttons are ordered from top to bottom, with the topmost representing SwiftFile’s best guess and the bottommost representing its third-best guess. When one of the three buttons is clicked, the message is immediately moved to the indicated folder.

SwiftFile’s predictions are made using a text classifier. Classifiers often require a large number of training messages before they yield accurate predictions, but SwiftFile circumvents this potential problem. During installation, it treats previously-filed messages as a corpus of labeled documents and uses them to train its text classifier. After its initial training, SwiftFile is immediately ready to make accurate predictions.

SwiftFile adapts to changing conditions by using incremental learning. Once the classifier has been trained, the classifier’s model is continuously updated by presenting to the classifier the messages that have been added to or deleted from each folder. The cost of the update is only linear in the length of the message. After updating, the classifier’s predictions are identical to those that would be obtained by training the classifier from scratch on the entire mail database.

## 3. AIM

SwiftFile uses a modified version of AIM to classify messages (Barrett & Selker, 1995). AIM is a TF-IDF style text classifier. We have modified the original AIM implementation to support incremental learning.

AIM represents each message  $\mathcal{M}$  as a word-frequency vector  $F(\mathcal{M})$ , in which each component  $F(\mathcal{M}, w)$  represents the total number of times the word  $w$  appears in  $\mathcal{M}$ . Each folder  $\mathcal{F}$  is represented using a *weighted* word-frequency vector  $W(\mathcal{F}, w)$ . Several steps are involved in computing  $W(\mathcal{F}, w)$ . First, the folder  $\mathcal{F}$ ’s centroid vector  $F(\mathcal{F}, w)$  is computed by summing the word-frequency vectors for each message in the folder:

$$F(\mathcal{F}, w) = \sum_{\mathcal{M} \in \mathcal{F}} F(\mathcal{M}, w). \quad (1)$$

This folder centroid vector is then converted to a

weighted word-frequency vector using the TF-IDF principle: the weight assigned to a word is proportional to its frequency in the folder and inversely proportional to its frequency in other folders. We define  $FF(\mathcal{F}, w)$  to be the fractional frequency of word  $w$  among messages contained within folder  $\mathcal{F}$ , or the number of times word  $w$  occurs in folder  $\mathcal{F}$  divided by the total number of words in  $\mathcal{F}$ :

$$FF(\mathcal{F}, w) = \frac{F(\mathcal{F}, w)}{\sum_{w' \in \mathcal{F}} F(\mathcal{F}, w')}. \quad (2)$$

The definition of term frequency  $TF(\mathcal{F}, w)$  used by AIM is

$$TF(\mathcal{F}, w) = FF(\mathcal{F}, w) / FF(\mathcal{A}, w), \quad (3)$$

where  $\mathcal{A}$  represents the set of all messages (the entire database of organized mail). We define the document frequency  $DF(w)$  to be the fraction of folders in which the word  $w$  appears at least once. The definition of inverse document frequency  $IDF(w)$  used by AIM is

$$IDF(w) = \frac{1}{DF(w)^2}. \quad (4)$$

Finally, AIM combines these two formulas to define the weight for word  $w$  in folder  $\mathcal{F}$ :

$$W(\mathcal{F}, w) = TF(\mathcal{F}, w) \times IDF(w). \quad (5)$$

The weight vectors for each folder are used to classify each new message. When a message  $\mathcal{M}$  arrives to be classified, it is first converted into a word-frequency vector  $F(\mathcal{M})$ . Then, AIM computes the similarity between  $\mathcal{M}$  and the weighted word-frequency vectors for each folder,  $W(\mathcal{F})$ . AIM computes the similarity between the message vector and the weighted folder vectors using a variation of cosine distance called  $SIM_4$  (Salton & McGill, 1983):

$$SIM_4(\mathcal{M}, \mathcal{F}) = \frac{\sum_{w \in \mathcal{M}} F(\mathcal{M}, w) W(\mathcal{F}, w)}{\min(\sum_{w \in \mathcal{M}} F(\mathcal{M}, w), \sum_{w \in \mathcal{M}} W(\mathcal{F}, w))}. \quad (6)$$

Here the sums are taken only over the words that are contained within  $\mathcal{M}$ . Finally, AIM takes the three folders with greatest similarity as its predictions.

The original AIM classifier implemented this classification technique as a batch algorithm. We modified AIM’s implementation to support incremental

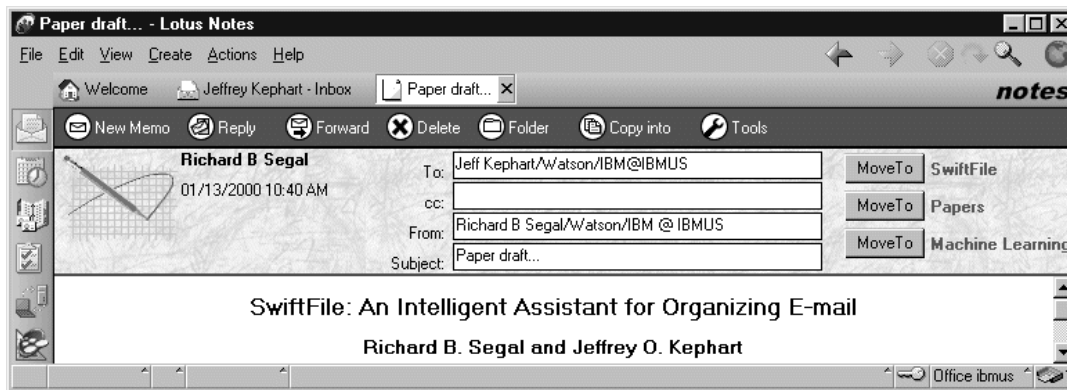


Figure 1. SwiftFile creates shortcut buttons for the three folders in which it predicts the user is most likely to place each message. When its predictions are correct, the buttons enable the user to quickly file messages into one of these folders.

learning. Our new AIM implementation maintains a database containing the centroids  $F(\mathcal{F})$  for each folder  $\mathcal{F}$ . Whenever a new message  $\mathcal{M}$  is added to a folder  $\mathcal{F}$ , each centroid is updated by adding to it the word-frequency vector for that message:

$$F(\mathcal{F}, w) \leftarrow F(\mathcal{F}, w) + F(\mathcal{M}, w). \quad (7)$$

Similarly, when a message is removed from a folder, the word-frequency vector for that message is subtracted from the folder’s centroid. The centroids of each folder are stored in an inverted index for quick access.

To classify a message  $\mathcal{M}$ , AIM computes the *SIM4* similarity between  $\mathcal{M}$  and each folder’s weight vector. The weight terms  $W(\mathcal{F}, w)$  required to compute *SIM4* can be computed on the fly from the folder centroids. Once the *SIM4* similarity score has been computed for each folder, AIM predicts the message as belonging to the three folders with the highest score.

## 4. Static Experiments

We first analyzed the performance of SwiftFile using standard, static evaluation techniques. We applied SwiftFile’s text classifier to the mailboxes of five users at our local institution. The experiments were performed using five of the original six users that participated in our previous study. The experiments presented in this paper are all new and use more recent versions of each user’s mail database. User #5 from the previous study was omitted because he was not available for the second round of experimentation.

Table 1 presents the characteristics of our test databases. The databases range from 636 to 7,411 messages filed in anywhere from 19 to 98 folders. The number of folders is important because the more folders in a database, the more difficult the classification problem. The experiment was conducted using each user’s previously-filed messages as data. We randomly

Table 1. Mail databases used to test SwiftFile.

Database	# Folders	# Messages
User #1	98	1,370
User #2	72	2,415
User #3	59	4,675
User #4	38	636
User #6	19	7,411

sampled 70% of each user’s previously-filed messages for training and used the remaining 30% of the messages for testing. We repeated the experiment ten times and averaged the results.

Figure 2 shows the results of this experiment. The graph shows SwiftFile’s accuracy for each user when SwiftFile provides from one to five shortcut buttons. The accuracy with  $N$  buttons is defined as the frequency that one of the first  $N$  buttons will move the message into the correct folder.

SwiftFile’s performance with just one button varies from 52% to 76%. A performance level of 76% is likely to be deemed helpful by most users, but a performance level of 52% is borderline. While the user may still save time using SwiftFile, the perception that SwiftFile is frequently incorrect may lead to user dissatisfaction. With three shortcut buttons, SwiftFile’s accuracy improves to 73% to 90%. The two extra buttons substantially improve performance without adversely affecting the user. While SwiftFile would be slightly more accurate by using up to five buttons, the extra performance seems not to merit the increase in screen real estate.

If SwiftFile were automatically filing messages, this level of performance would be unacceptable. However, since SwiftFile’s incorrect predictions are easily overridden, a 10% to 27% error rate is only a minor annoyance that is more than compensated by the benefit of receiving the right suggestion 73% to 90% of the time.

Figure 2 also compares the performance of SwiftFile to the naive strategy of providing shortcut buttons for

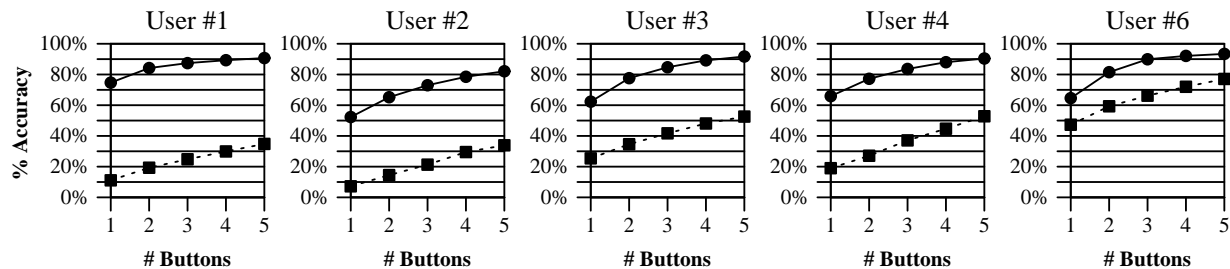


Figure 2. Static performance of SwiftFile. Solid lines show the accuracy of SwiftFile with one to five buttons. Dotted lines show the accuracy of the naive strategy of providing shortcut buttons for the  $N$  most-frequently-used folders.

the  $N$  most-frequently-used folders. SwiftFile provides a substantial improvement over the naive strategy on all five databases. The naive strategy performs well for User #6 since this user actively uses only a few folders. SwiftFile is still advantageous to User #6 since it cuts the perceived error rate with three buttons in half.

## 5. Dynamic Experiments

Static evaluation of SwiftFile does not necessarily reveal much about its performance in a dynamic environment. The goal of dynamic evaluation is to understand how SwiftFile performs over time. This analysis will allow us to answer key questions such as how well SwiftFile performs for new users, how well it reacts to new information and the creation of new folders, and how important is incremental learning.

The most accurate way to evaluate SwiftFile’s dynamic behavior would be to give SwiftFile to several users and chart its performance over a year or two. While this method is the most accurate, it is very time consuming and makes it difficult to experiment with algorithmic variations. Instead, we use a static mail database to simulate a dynamic environment. The basic idea is to use the previously-filed messages in a mailbox to simulate a continuous stream of messages. The previously-filed messages are presented to the SwiftFile simulator in date order. As each message arrives, it is classified by SwiftFile using its latest classifier. A running score is maintained by comparing SwiftFile’s classifications to the known destination of each folder. The message is then filed into the destination folder, and the classifier’s model of the destination folder is updated. The simulator assumes there are no folders at the beginning of the simulation and assumes a new folder is created when the simulator processes the first message for each folder. The above procedure is repeated for each message in the mail database.

In the experiments that follow, since there is not sufficient space to show the learning curves for each experiment on each database tested, we present the learning

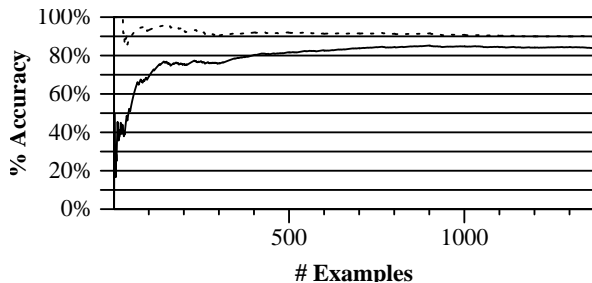


Figure 3. Simulation results for the mailbox of User #1. The solid line graphs accuracy across all messages seen. The dotted line is otherwise identical, but SwiftFile is not penalized for missing messages destined for new folders.

curve for User #1 and summarize the results for the other databases in a table.

The first question we would like to answer is how many messages does a TF-IDF style classifier need to see before it can make useful predictions. Figure 3 helps answer this question by plotting SwiftFile’s learning curve when simulating its behavior on User #1’s mailbox. The messages span a period of three years. The solid line graphs the accuracy of SwiftFile using three buttons. At first, the results appear disappointing. After missing most of the first few messages, the classifier stabilizes at 40% accuracy. The classifier continues to struggle at the 40% level until it has processed about 30 messages, after which point the classifier displays a classic learning curve with steady improvement. The classifier does not start performing well until it has processed over 100 messages.

While these results suggest that the classifier needs hundreds of examples before it can effectively learn the user’s mail-filing habits, a closer evaluation reveals that something more subtle is happening. In the first twenty-five messages, there are fourteen messages that are placed in new folders. SwiftFile cannot make correct predictions for these messages because their destination folders do not exist at classification time. SwiftFile can hardly be blamed for not classifying a message into a category that does not exist. The dotted line in

Table 2. Performance of SwiftFile on the first fifty training examples. The results show that SwiftFile does not need many examples to achieve good performance.

Database	Cumulative Accuracy
User #1	90.0
User #2	87.5
User #3	97.6
User #4	95.0
User #6	100.0

Figure 3 shows SwiftFile’s performance if we exclude messages that prompt the creation of new folders. The results are dramatically different. SwiftFile’s classifier performs exceptionally well right from the start. SwiftFile is perfect over the first thirty messages, sixteen of which go to new folders and fourteen of which are classified correctly. SwiftFile’s performance stays high throughout the experiment, only briefly dropping below the 90% level near the beginning.

Figure 3 also demonstrates an interesting phenomenon that is common in SwiftFile — an inverted learning curve in which accuracy starts out high and slowly decreases until plateauing. The decreasing learning curve is the result of the classification task constantly changing. SwiftFile’s initial classification task is easy because the user has very few folders. In fact, before the user creates three folders, SwiftFile’s three shortcut buttons can provide quick access to all of the user’s folders. The small number of folders that exists during the first part of the experiment explains why SwiftFile performs so well with limited training data. The classification task gets more difficult as time progresses because the number of folders from which it must choose is gradually increasing. Moreover, the messages placed in any folder become less homogeneous over time as the content of the messages placed in that folder shifts. However, as the learning problem gets harder, the learning algorithm has more data from which to learn. Therefore, there are two competing forces — one that forces the curve downward and one that forces the curve upward. In this case, the result is a downward-sloping learning curve.

Table 2 shows the performance of SwiftFile’s text classifier on the first fifty messages of each mail database when the classifier is not penalized for messages destined for new folders. The results clearly demonstrate that because of the nature of this domain very few training examples are needed to successfully train a classifier to make good predictions.

While our goal was to investigate the learning rate of TF-IDF, what we learned instead is that e-mail classification often does not require a quick learner. The

e-mail classification task often starts out easy and does not become difficult until a substantial number of messages are available for training.

## 6. Classifier Adaptation

We now investigate how well the classifier adapts to changes in the user’s mail-filing habits. The graphs shown in the previous section do not display short-term dynamics well because the performance measure being graphed is averaged over the entire history of the experiment. Instead, we graph the same data using moving averages. The moving average tells us how well the classifier performs over the last dozen or so messages and gives a better idea of how its performance varies over time.

Figure 4 displays the same data as were presented in Figure 3, except that the performance is displayed using a moving average. The moving average was calculated using a dampening function that discounts the weight of earlier predictions. The dampening function has a half-life of ten messages. That is, it assigns half the weight to the last ten messages, a quarter of the weight to the previous ten messages, etc. The graph shows the accuracy of SwiftFile when the classifier is not penalized for messages placed into new folders.

The graph demonstrates that classifier accuracy does vary considerably but usually maintains at least an 80% accuracy level. There are five dips in performance at about 30, 200, 250, 900, and 1,350 messages. The right side of the figure shows the same graph with vertical lines indicating when folders were created. Each dip is associated with substantial folder-creation activity. The system usually recovers quickly after each dip, with the dip at 250 messages taking the longest — 56 messages from when the classifier first drops below 90% until it returns to over 90%. While these results show that the creation of new folders can cause classifier performance to degrade, the amount of degradation is acceptable given that the classifier stays mostly above the 80% level and never drops below 67%.

Table 3 shows the results of this experiment for all six databases. The table summarizes how the moving average for each database varies over time by calculating the minimum, maximum, mean, and standard deviation of the moving average during the length of the experiment. The first fifty examples are excluded from the calculation to filter out early variations. The results show that the mean of the moving average is often very close to the three-button accuracy reported in the static experiments — suggesting that incremental learning is in fact successful. Furthermore, while

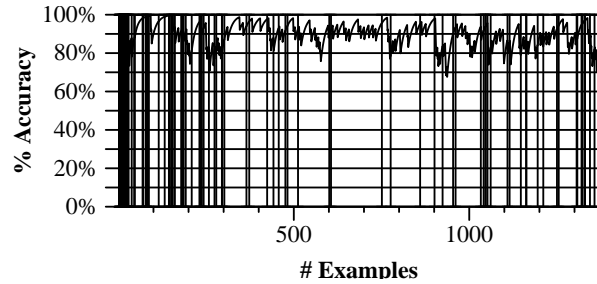
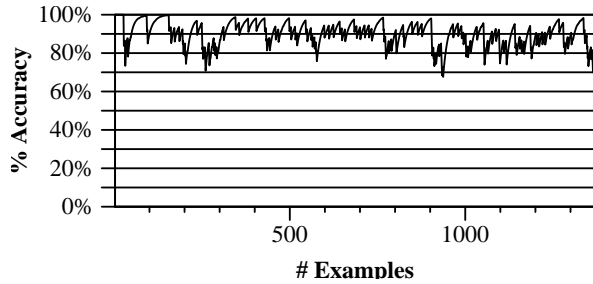


Figure 4. Simulation results for User #1 graphed using a moving average. The two graphs are identical except that the right shows when new folders were created. While performance varies considerably over time, it rarely drops below 75%.

Table 3. Performance of incremental learning. Cumulative accuracy and the mean of the moving average are indicative of absolute performance, while the minimum, maximum and standard deviation of the moving average suggest how much performance varies over time.

Database	Cumulative Accuracy	Moving Average			
		Min	Max	Mean	Std. Dev.
User #1	89.7	67.6	99.7	90.0	5.5
User #2	74.5	42.4	97.5	74.4	10.1
User #3	86.3	57.3	99.9	86.3	7.1
User #4	85.0	52.0	98.7	84.5	8.6
User #6	92.9	59.2	100.0	92.9	6.5

the standard deviations do indicate some volatility, the variations are usually not large enough for the user to experience very low accuracies.

Interestingly, many of the folder creations shown in Figure 4 are not accompanied with drops in classifier performance. SwiftFile’s classifier maintains roughly-similar performance despite being faced with a harder classification problem after new folders are created and not having much data with which to learn about the new folders. This suggests that SwiftFile’s classifier can often successfully learn about new folders from very few examples.

Figure 5 analyzes how well SwiftFile classifies User #1’s messages into a folder after seeing only a single message for that folder. The graph shows the cumulative accuracy of SwiftFile in filing the second message that is to be placed in each folder. That is, the graph is identical to that of Figure 3, except that only the second message to be placed in each folder is scored. The graph shows that SwiftFile’s classifier does quite well on the second message placed in each folder, especially considering the limited training data upon which it must base its decision. SwiftFile is perfect on the first six folders and only misses two of the first sixteen. More importantly, SwiftFile continues to perform well even as the number of folders gets large. For the last thirty folders, SwiftFile is 73% accurate in placing the

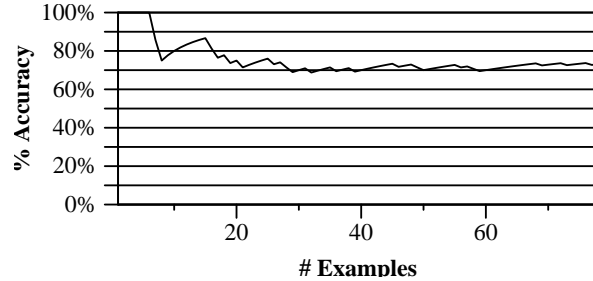


Figure 5. SwiftFile’s accuracy in filing the second message that is to be placed into each folder. While there is only a single training example, performance is still good.

Table 4. Performance of incremental learning on newly created folders. The results are very good given that there is only one training example for each new folder.

Database	Cumulative Accuracy
User #1	73.1
User #2	59.4
User #3	73.2
User #4	72.7
User #6	89.5

second message into a new folder. Table 4 shows that SwiftFile’s accuracy on newly-created folders is above 59% on each of our test databases and is above 72% for most of them.

The ability of SwiftFile’s classifier to learn from a single example is unexpected. Usually, learning algorithms require a substantial number of examples before they can make any predictions. The success of TF-IDF in learning from a single example is probably partially due to TF-IDF using all the features present in a message to make its classification rather than selecting a few salient features as is common in decision-tree and decision-rule algorithms. However, we leave testing this hypothesis to future work.

## 7. Incremental Learning

Incremental learning allows SwiftFile to adapt to changes in how each user files e-mail. SwiftFile up-

dates its classifier whenever a user files a message, moves a message between folders, or deletes a message. While incremental learning ensures that the classifier is as up-to-date as possible, it comes at a cost. First, the use of incremental learning reduces one’s choices of learning algorithms, given that many good algorithms for text classification are not incremental (Cohen, 1995; Apte et al., 1994). Second, it requires that the classifier and the e-mail client be closely integrated in order for the classifier to be notified of all important events. Given the costs of incremental learning, the decision to use it must be based on empirical evidence that doing so confers some advantage.

We evaluate the importance of incremental learning by comparing its performance to that of periodic learning in which the classifier is only updated every thirty messages. The intent is to compare incremental learning against the common strategy in batch systems to update the classifier overnight (Maes, 1994; Payne & Edwards, 1997; Mitchell et al., 1994). We chose a period of thirty messages because we estimate that the average active e-mail user receives about thirty messages per day. Second, we believe many users do not file their e-mail daily. Instead, they wait until their inbox starts to get too large, and then they file many messages in one sitting. As a result, users who receive fewer messages per day may still present to the classifier on the order of thirty new examples per session.

The left side of Figure 6 compares the performance of incremental learning (dotted line) and periodic learning (solid line). While the performance of periodic learning is only slightly less than that of incremental learning throughout the experiment, there are many places where the performance of periodic learning dips considerably below that of incremental learning. Periodic learning drops to 65% or below several times and falls to as low as 33%, while incremental learning never drops below 67% and rarely drops below 80%. The cumulative accuracy of periodic learning across the entire database is 84.2%, which is 5.5% points lower than the 89.7% cumulative accuracy of incremental learning.

Table 5 shows the performance of periodic learning on all databases. When compared with the results of incremental learning in Table 3, we see that periodic learning substantially reduces the overall accuracy of the classifier and simultaneously increases volatility.

The right side of Figure 6 compares the performance of incremental and periodic learning on the task of predicting the second message to be placed in each folder. Again, the solid line represents periodic learning, and the dotted line represents incremental learning. Periodic learning struggles with this task, averaging about

Table 5. Performance of periodic learning. Periodic learning results in lower accuracy and increased variability as compared with incremental learning.

Database	Cumulative Accuracy	Moving Average			
		Min	Max	Mean	Std. Dev.
User #1	84.2	40.1	99.4	84.8	9.6
User #2	64.2	18.1	97.2	64.3	13.7
User #3	82.6	35.0	99.6	82.5	8.8
User #4	66.2	16.9	95.0	67.8	14.0
User #6	91.4	55.3	100.0	91.3	7.4

Table 6. Performance of periodic learning on newly created folders. Periodic learning cannot match incremental learning’s success because of its infrequent updates.

Database	Cumulative Accuracy
User #1	46.1
User #2	32.8
User #3	37.5
User #4	36.4
User #6	73.7

50% accuracy. Similar results occur on all databases, as shown in Table 6.

These results suggest that incremental learning does offer a substantial improvement for e-mail classification tasks. Incremental learning improves accuracy, reduces the amount of time the user perceives the classifier as performing poorly, and enables the classifier to perform well on newly-created folders.

## 8. Related Work

Mitchell et al. (1994) performs a similar analysis of the dynamic behavior of CAP, a calendar-scheduling assistant that learns user preferences to assist the user in scheduling meetings. CAP uses a batch learning algorithm which is retrained nightly. Interestingly, since it was tested in an academic environment, CAP demonstrates cyclical learning curves that fluctuate with changes in the academic calendar.

There is a substantial body of work investigating the use of text classification to help users handle their e-mail (Lashkari et al., 1994; Payne & Edwards, 1997; Boone, 1998; Cohen, 1996). Much of the work has focused on accurately predicting which actions should be performed based on a static training corpus, while ignoring equally-important issues regarding how well the classifier adapts to continuous change. There are a few notable exceptions. Lashkari et al. (1994) presents a small learning curve to demonstrate that her agents learn to collaborate. Payne and Edwards (1997) do consider the possibility of incremental learning, but very few details are provided.

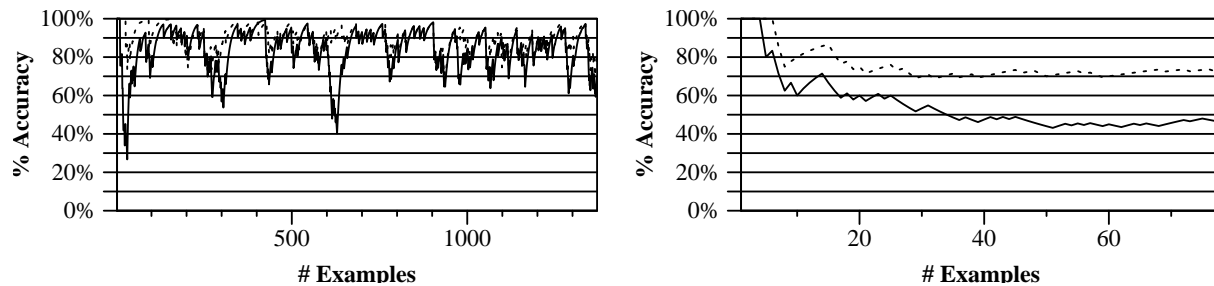


Figure 6. Comparison of incremental learning and periodic learning in which learning occurs every thirty messages. The solid line shows periodic learning, and the dotted line shows incremental learning. The left graph compares the two using a moving average, and the right graph compares them on the learning of the second example placed in a new folder. The accuracy of periodic learning is always slightly behind incremental learning and displays some large performance drops.

## 9. Conclusions

While many learning systems are designed to operate in a dynamic environment, they are often evaluated using static analysis techniques. In this paper, we have argued the importance of studying the dynamic properties of a learning algorithm. By evaluating the dynamic properties of SwiftFile, we have learned valuable lessons about the nature of the e-mail classification problem and the properties of TF-IDF text classifiers. In particular, we were able to dispel two myths about learning an e-mail classifier. First, it does not take many messages for an e-mail classification system to learn about the user's mail-filing habits because, by the time the user creates enough folders for the problem to be difficult, the learning algorithm will have ample training data to make good predictions. Second, updating a classifier overnight is not sufficient because overnight updates cannot respond quickly enough to frequent changes in the user's mail-filing habits. We also learned that TF-IDF classifiers are very responsive and can make useful predictions in a category even after seeing just a single message. In future work, we hope to compare the responsiveness of TF-IDF classifiers to other types of text classifiers.

## References

- Apte, C., Damerau, F., & Weiss, S. M. (1994). Towards language independent automated learning of text categorization models. *Proceedings of the Seventeenth Annual ACM SIGIR Conference* (pp. 23–30). New York: ACM Press.
- Barrett, R., & Selker, T. (1995). *AIM: A new approach for meeting information needs* (Technical Report). IBM Almaden Research Center, Almaden, CA.
- Boone, G. (1998). Concept features in Re:Agent, an intelligent email agent. *Proceedings of the Second International Conference on Autonomous Agents* (pp. 141–148). New York: ACM Press.
- Cohen, W. W. (1995). Fast effective rule induction. *Proceedings of the Twelfth International Conference on Machine Learning* (pp. 115–123). San Francisco, CA: Morgan Kaufmann.
- Cohen, W. W. (1996). Learning rules that classify e-mail. *Proceedings of the 1996 AAAI Spring Symposium on Machine Learning and Information Access* (pp. 18–25). Menlo Park, CA: AAAI Press.
- Lashkari, Y., Metral, M., & Maes, P. (1994). Collaborative interface agents. *Proceedings of the Twelfth National Conference on Artificial Intelligence* (pp. 444–449). Menlo Park, CA: AAAI Press.
- Maes, P. (1994). Agents that reduce work and information overload. *Communications of the ACM*, 37, 31–40.
- Mitchell, T., Caruana, R., Freitag, D., McDermott, J., & Zabowski, D. (1994). Experience with a learning personal assistant. *Communications of the ACM*, 37, 80–91.
- Payne, T. R., & Edwards, P. (1997). Interface agents that learn: An investigation of learning issues in a mail agent interface. *Applied Artificial Intelligence*, 11, 1–32.
- Salton, G., & McGill, M. J. (1983). *Introduction to modern information retrieval*. New York: McGraw-Hill Book Company.
- Segal, R. B., & Kephart, J. O. (1999). MailCat: An intelligent assistant for organizing e-mail. *Proceedings of the Third International Conference on Autonomous Agents* (pp. 276–282). New York: ACM Press.