Learning from evolving data streams: online triage of bug reports

Grzegorz Chrupala
Spoken Language Systems
Saarland University
gchrupala@lsv.uni-saarland.de

Abstract

Open issue trackers are a type of social media that has received relatively little attention from the text-mining community. We investigate the problems inherent in learning to triage bug reports from time-varying data. We demonstrate that concept drift is an important consideration. We show the effectiveness of online learning algorithms by evaluating them on several bug report datasets collected from open issue trackers associated with large open-source projects. We make this collection of data publicly available.

1 Introduction

There has been relatively little research to date on applying machine learning and Natural Language Processing techniques to automate software project workflows. In this paper we address the problem of bug report triage.

1.1 Issue tracking

Large software projects typically track defect reports, feature requests and other issue reports using an issue tracker system. Open source projects tend to use trackers which are open to both developers and users. If the product has many users its tracker can receive an overwhelming number of issue reports: Mozilla was receiving almost 300 reports per day in 2006 (Anvik et al. 2006). Someone has to monitor those reports and **triage** them, that is decide which component they affect and which developer or team of developers should be responsible for analyzing them and fixing the reported defects. An automated agent assisting the staff responsible for such triage has the potential to substantially reduce the time and cost of this task.

1.2 Issue trackers as social media

In a large software project with a loose, not strictly hierarchical organization, standards and practices are not exclusively imposed top-down but also tend to spontaneously arise in a bottom-up fashion, arrived at through interaction of individual developers, testers and users. The individuals involved may negotiate practices explicitly, but may also imitate and influence each other via implicitly acquired reputation and status. This process has a strong emergent component: an informal taxonomy may arise and evolve in an issue tracker via the use of free-form tags or labels. Developers, testers and users can attach tags to their issue reports in order to informally classify them. The issue tracking software may give users feedback by informing them which tags were frequently used in the past, or suggest tags based on the content of the report or other information. Through this collaborative, feedback driven process involving both human and machine participants, an evolving consensus on the label inventory and semantics typically arises, without much top-down control (Halpin et al. 2007).

This kind of emergent taxonomy is known as a **folksonomy** or **collaborative tagging** and is very common in the context of social web applications. Large software projects, especially those with open policies and little hierarchical structures, tend to exhibit many of the same emergent social properties as the more prototypical social applications. While this is a useful phenomenon, it presents a special challenge from the machine-learning point of view.

Proceedings of the 13th Conference of the European Chapter of the Association for Computational Linguistics, pages 613–622,
Avignon, France, April 23 - 27 2012.

1.3 Concept drift

Many standard supervised approaches in machine-learning assume a stationary distribution from which training examples are independently drawn. The set of training examples is processed as a *batch*, and the resulting learned decision function (such as a classifier) is then used on test items, which are assumed to be drawn from the same stationary distribution.

If we need an automated agent which uses human labels to learn to tag objects the batch learning approach is inadequate. Examples arrive one-by-one in a stream, not as a batch. Even more importantly, both the output (label) distribution and the input distribution from which the examples come are emphatically **not** stationary. As a software project progresses and matures, the type of issues reported is going to change. As project members and users come and go, the vocabulary they use to describe the issues will vary. As the consensus tag folksonomy emerges, the label and training example distribution will evolve. This phenomenon is sometimes referred to as **concept drift** (Widmer and Kubat 1996, Tsymbal 2004).

Early research on learning to triage tended to either not notice the problem (Čubranić and Murphy 2004), or acknowledge but not address it (Anvik et al. 2006): the evaluation these authors used assigned bug reports randomly to training and evaluation sets, discarding the temporal sequencing of the data stream.

Bhattacharya and Neamtiu (2010) explicitly address the issue of online training and evaluation. In their setup, the system predicts the output for an item based only on items preceding it in time. However, their approach to incremental learning is simplistic: they use a batch classifier, but retrain it from scratch after receiving each training example. A fully retrained batch classifier will adapt only slowly to changing data stream, as more recent example have no more influence on the decision function that less recent ones.

Tamrawi et al. (2011) propose an incremental approach to bug triage: the classes are ranked according to a fuzzy set membership function, which is based on incrementally updated feature/class co-occurrence counts. The model is efficient in online classification, but also adapts only slowly.

1.4 Online learning

This paucity of research on online learning from issue tracker streams is rather surprising, given that truly incremental learners have been well-known for many years. In fact one of the first learning algorithms proposed was Rosenblatt's perceptron, a simple mistake-driven discriminative classification algorithm (Rosenblatt 1958). In the current paper we address this situation and show that by using simple, standard online learning methods we can improve on batch or pseudo-online learning. We also show that when using a sophisticated state-of-the-art stochastic gradient descent technique the performance gains can be quite large.

1.5 Contributions

Our main contributions are the following: Firstly, we explicitly show that concept-drift is pervasive and serious in real bug report streams. We then address this problem by leveraging state-of-the-art online learning techniques which automatically track the evolving data stream and incrementally update the model after each data item. We also adopt the continuous evaluation paradigm, where the learner predicts the output for each example before using it to update the model. Secondly, we address the important issue of reproducibility in research in bug triage automation by making available the data sets which we collected and used, in both their raw and preprocessed forms.

2 Open issue-tracker data

Open source software repositories and their associated issue trackers are a naturally occurring source of large amounts of (partially) labeled data. There seems to be growing interest in exploiting this rich resource as evidenced by existing publications as well as the appearance of a dedicated workshop (Working Conference on Mining Software Repositories).

In spite of the fact that the data is publicly available in open repositories, it is not possible to directly compare the results of the research conducted on bug triage so far: authors use non-trivial project-specific filtering, re-labeling and pre-processing heuristics; these steps are usually not specified in enough detail that they could be easily reproduced.

Field	Meaning
Identifier	Issue ID
Title	Short description of issue
Description	Content of issue report, which may include steps to reproduce, error messages, stack traces etc.
Author	ID of report submitter
CCS	List of IDs of people CC'd on the issue report
Labels	List of tags associated with issue
Status	Label describing the current status of the issue (e.g. Invalid, Fixed, Won't Fix)
Assigned To	ID of person who has been assigned to deal with the issue
Published	Date on which issue report was submitted

Table 1: Issue report record

To help remedy this situation we decided to collect data from several open issue trackers, use the minimal amount of simple preprocessing and filter heuristics to get useful input data, and publicly share both the raw and preprocessed data.

We designed a simple record type which acts as a common denominator for several tracker formats. Thus we can use a common representation for issue reports from various trackers. The fields in our record are shown in Table 1.

Below we describe the issue trackers used and the datasets we build from them. As discussed above (and in more detail in Section 4.1), we use progressive validation rather than a split into training and test set. However, in order to avoid developing on the test data, we split each data stream into two substreams, by assigning odd-numbered examples to the test stream and the even-numbered ones to the development stream. We can use the development stream for exploratory data analysis and feature and parameter tuning, and then use progressive validation to evaluate on entirely unseen test data. Below we specify the size and number of unique labels in the development sets; the test sets are very similar in size.

Chromium Chromium is the open source-project behind Google's Chrome browser (`http://code.google.com/p/chromium/`). We retrieved all the bugs from the issue tracker, of which 66,704 have one of the closed statuses. We generated two data sets from the Chromium issues:

- Chromium SUBCOMPONENT. Chromium uses special tags to help triage the bug reports. Tags prefixed with `Area-` specify which subcomponent of the project the bug should be routed to. In some cases more than one `Area-` tag is present. Since this affects less than 1% of reports, for simplicity we treat these as single, compound labels. The development set contains 31,953 items, and 75 unique output labels.
- Chromium ASSIGNED. In this dataset the output is the value of the `assignedTo` field. We discarded issues where the field was left empty, as well as the ones which contained the placeholder value `all-bugs-test.chromium.org`. The development set contains 16,154 items and 591 unique output labels.

Android Android is a mobile operating system project (`http://code.google.com/p/android/`). We retrieved all the bugs reports, of which 6,341 had a closed status. We generated two datasets:

- Android SUBCOMPONENT. The reports which are labeled with tags prefixed with `Component-`. The development set contains 888 items and 12 unique output labels.
- Android ASSIGNED. The output label is the value of the `assignedTo` field. We discarded issues with the field left empty. The development set contains 718 items and 72 unique output labels.

Firefox Firefox is the well-known web-browser project (`https://bugzilla.mozilla.org`).

We obtained a total of 81,987 issues with a closed status.

- Firefox ASSIGNED. We discarded issues where the field was left empty, as well as the ones which contained a placeholder value (`nobody`). The development set contains 12,733 items and 503 unique output labels.

Launchpad Launchpad is an issue tracker run by Canonical Ltd for mostly Ubuntu-related projects (`https://bugs.launchpad.`

net/). We obtained a total of 99,380 issues with a closed status.

- Launchpad ASSIGNED. We discarded issues where the field was left empty. The development set contains 18,634 items and 1,970 unique output labels.

3 Analysis of concept drift

In the introduction we have hypothesized that in issue tracker streams concept drift would be an especially acute problem. In this section we show how class distributions evolve over time in the data we collected.

A time-varying distribution is difficult to summarize with a single number, but it is easy to appreciate in a graph. Figures 1 and 2 show concept drift for several of our data streams. The horizontal axis indexes the position in the data stream. The vertical axis shows the class proportions at each position, averaged over a window containing 7% of all the examples in the stream, i.e. in each thin vertical bar the proportion of colors used corresponds to the smoothed class distribution at a particular position in the stream.

Consider the plot for Chromium SUBCOMPONENT. We can see that a bit before the middle point in the stream class proportions change quite dramatically: The orange BROWSERUI and violet MISC almost disappears, while blue INTERNALS, pink UI and dark red UNDEFINED take over. This likely corresponds to an overhaul in the label inventory and/or recommended best practice for triage in this project. There are also more gradual and smaller scale changes throughout the data stream.

The Android SUBCOMPONENT stream contains much less data so the plot is less smooth, but there are clear transitions in this image also. We see that light blue GOOGLE all but disappears after about two thirds point and the proportion of violet TOOLS and light-green DALVIK dramatically increases.

In Figure 2 we see the evolution of class proportions in the ASSIGNED datasets. Each plot's idiosyncratic shape illustrates that there is wide variation in the amount and nature of concept drift in different software project issue trackers.

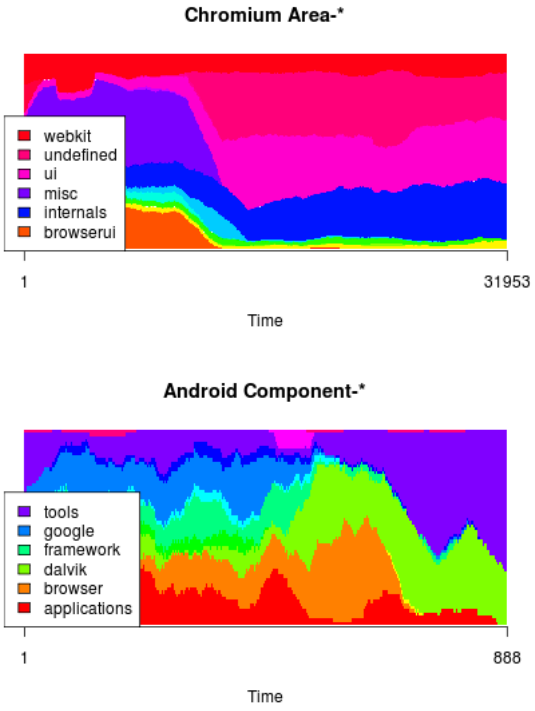

Figure 1: SUBCOMPONENT class distribution change over time

4 Experimental results

In an online setting it is important to use an evaluation regime which closely mimics the continuous use of the system in a real-life situation.

4.1 Progressive validation

When learning from data streams the standard evaluation methodology where data is split into a separate training and test set is not applicable. An evaluation regime know as progressive validation has been used to accurately measure the generalization performance of online algorithms (Blum et al. 1999). Under progressive evaluation, an input example from a temporally ordered sequence is sent to the learner, which returns the prediction. The error incurred on this example is recorded, and the true output is only then sent to the learner which may update its model based on it. The final error is the mean of the per-example errors. Thus even though there is no separate test set, the prediction for each input is generated based on a model trained on examples which do not include it.

In previous work on bug report triage, Bhattacharya and Neamtiu (2010) and Tamrawi et al. (2011) used an evaluation scheme (close to) pro-

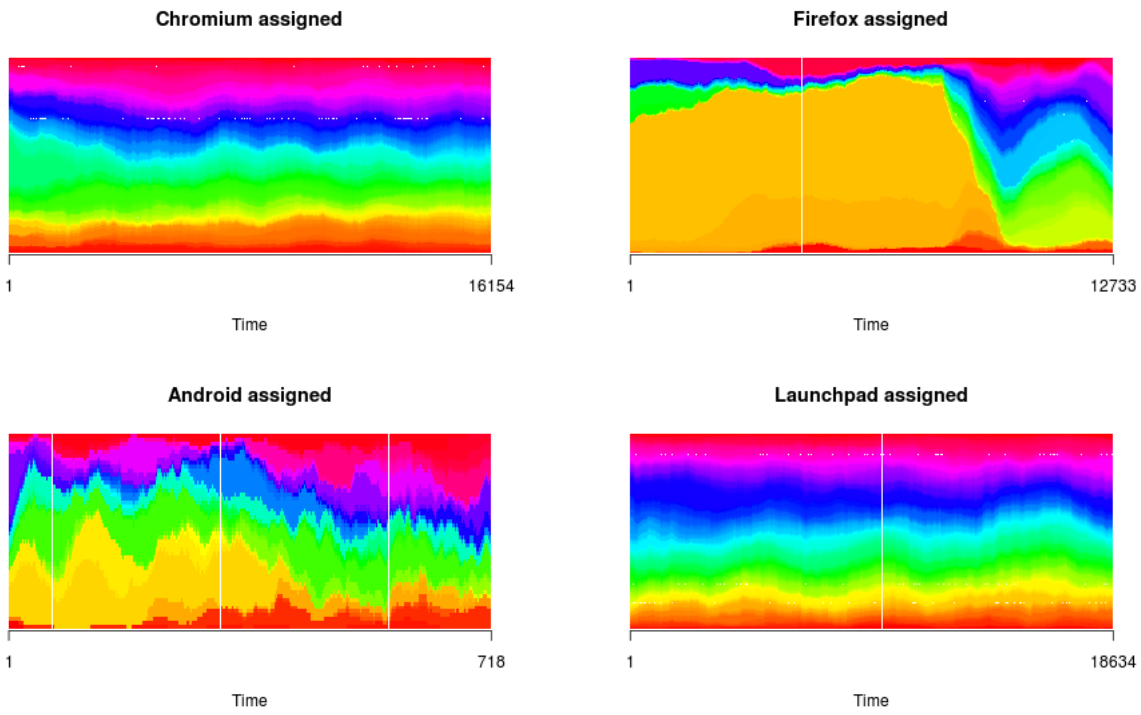

Figure 2: ASSIGNED class distribution change over time

gressive validation. They omit the initial $\frac{1}{11}^{\text{th}}$ of the examples from the mean.

4.2 Mean reciprocal rank

A bug report triaging agent is most likely to be used in a semi-automatic workflow, where a human triager is presented with a ranked list of possible outputs (component labels or developer IDs). As such it is important to evaluate not only accuracy of the top ranking suggesting, but rather the quality of the whole ranked list.

Previous research (Bhattacharya and Neamtiu 2010, Tamrawi et al. 2011) made an attempt at approximating this criterion by reporting scores which indicate whether the true output is present in the top n elements of the ranking, for several values of n. Here we suggest borrowing the mean reciprocal rank (MRR) metric from the information retrieval domain (Voorhees 2000). It is defined as the mean of the reciprocals of the rank at which the true output is found:

$$\text{MRR} = \frac{1}{N}\sum_{i=1}^{N}\text{rank}(i)^{-1}$$

where $\text{rank}(i)$ indicates the rank of the i^{th} true output. MRR has the advantage of providing a single number which summarizes the quality of whole rankings for all the examples. MRR is also a special case of Mean Average Precision when there is only one true output per item.

4.3 Input representation

Since in this paper we focus on the issues related to concept drift and online learning, we kept the feature set relatively simple. We preprocess the text in the issue report title and description fields by removing HTML markup, tokenizing, lowercasing and removing most punctuation. We then extracted the following feature types:

- Title unigram and bigram counts
- Description unigram and bigram counts
- Author ID (binary indicator feature)
- Year, month and day of submission (binary indicator features)

4.4 Models

We tested a simple online baseline, a pseudo-online algorithm which uses a batch model and repeatedly retrains it, an online model used in previous research on bug triage and two generic online learning algorithms.

Window Frequency Baseline This baseline does not use any input features. It outputs the

ranked list of labels for the current item based on the relative frequencies of output labels in the window of k previous items. We tested windows of size 100 and 1000 and report the better result.

SVM Minibatch This model uses the multiclass linear Support Vector Machine model (Crammer and Singer 2002) as implemented in SVM Light (Joachims 1999). SVM is known as a state-of-the-art batch model in classification in general and in text categorization in particular. The output classes for an input example are ranked according to the value of the discriminant values returned by the SVM classifier. In order to adapt the model to an online setting we retrain it every n examples on the window of k previous examples. The parameters n and k can have large influence on the prediction, but it is not clear how to set them when learning from streams. Here we chose the values (100,1000) based on how feasible the run time was and on the performance during exploratory experiments on Chromium SUBCOMPONENT. Interestingly, keeping the window parameter relatively small helps performance: a window of 1,000 works better than a window of 5,000.

Perceptron We implemented a single-pass online multiclass Perceptron with a constant learning rate. It maintains a weight vector for each output seen so far: the prediction function ranks outputs according to the inner product of the current example with the corresponding weight vector. The update function takes the true output and the predicted output. If they are not equal, the current input is subtracted from the weight vector corresponding to the predicted output and added to the weight vector corresponding to the true output (see Algorithm 1). We hash each feature to an integer value and use it as the feature's index in the weight vectors in order to bound memory usage in an online setting (Weinberger et al. 2009). The Perceptron is a simple but strong baseline for online learning.

Bugzie This is the model described in Tamrawi et al. (2011). The output classes are ranked according to the fuzzy set membership function defined as follows:

$$\mu(y, X) = 1 - \prod_{x \in X} \left(1 - \frac{n(y,x)}{n(y) + n(x) - n(y,x)}\right)$$

Algorithm 1 Multiclass online perceptron

function PREDICT($Y, \mathbf{W}, \mathbf{x}$)
  return $\{(y, \mathbf{W}_y^T \mathbf{x}) \mid y \in Y\}$

procedure UPDATE($\mathbf{W}, \mathbf{x}, \hat{y}, y$)
  if $\hat{y} \neq y$ **then**
    $\mathbf{W}_{\hat{y}} \leftarrow \mathbf{W}_{\hat{y}} - \mathbf{x}$
    $\mathbf{W}_y \leftarrow \mathbf{W}_y + \mathbf{x}$

where y is the output label, X the set of features in the input issue report, $n(y, x)$ the number of examples labeled as y which contain feature x, $n(y)$ number of examples labeled y and $n(x)$ number of examples containing feature x. The counts are updated online. Tamrawi et al. (2011) also use two so called caches: the label cache keeps the $j\%$ most recent labels and the term cache the k most significant features for each label. Since in Tamrawi et al. (2011)'s experiments the label cache did not affect the results significantly, here we always set j to 100%. We select the optimal k parameter from $\{100, 1000, 5000\}$ based on the development set.

Regression with Stochastic Gradient Descent This model performs online multiclass learning by means of a reduction to regression. The regressor is a linear model trained using Stochastic Gradient Descent (Zhang 2004). SGD updates the current parameter vector $\mathbf{w}^{(t)}$ based on the gradient of the loss incurred by the regressor on the current example $(\mathbf{x}^{(t)}, y^{(t)})$:

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta(t) \nabla L(y^{(t)}, \mathbf{w}^{(t)^T} \mathbf{x}^{(t)})$$

The parameter $\eta(t)$ is the learning rate at time t, and L is the loss function. We use the squared loss:

$$L(y, \hat{y}) = (y - \hat{y})^2$$

We reduce multiclass learning to regression using a one-vs-all-type scheme, by effectively transforming an example $(\mathbf{x}, y) \in X \times Y$ into $|Y|$ $(\mathbf{x}', y') \in X' \times \{0, 1\}$ examples, where Y is the set of labels seen so far. The transform T is defined as follows:

$$T(\mathbf{x}, y) = \{(\mathbf{x}', I(y = y')) \mid y' \in Y,\ x'_{h(i,y')} = x_i\}$$

where $h(i, y')$ composes the index i with the label y' (by hashing).

For a new input $\mathbf{x}$ the ranking of the outputs $y \in Y$ is obtained according to the value of the

prediction of the base regressor on the binary example corresponding to each class label.

As our basic regression learner we use the efficient implementation of regression via SGD, Vowpal Wabbit (VW) (Langford et al. 2011). VW implements setting adaptive individual learning rates for each feature as proposed by Duchi et al. (2010), McMahan and Streeter (2010).

This is appropriate when there are many sparse features, and is especially useful in learning from text from fast evolving data. The features such as unigram and bigram counts that we rely on are notoriously sparse, and this is exacerbated by the change over time in bug report streams.

4.5 Results

Figures 3 and 4 show the progressive validation results on all the development data streams. The horizontal lines indicate the mean MRR scores for the whole stream. The curves show a moving average of MRR in a window comprised of 7% of the total number of items. In most of the plots it is evident how the prediction performance depends on the concept drift illustrated in the plots in Section 3: for example on Chromium SUBCOMPONENT the performance of all the models drops a bit before the midpoint in the stream while the learners adapt to the change in label distribution that is happening at this time. This is especially pronounced for Bugzie, since it is not able to learn from mistakes and adapt rapidly, but simply accumulates counts.

For five out of the six datasets, Regression SGD gives the best overall performance. On Launchpad ASSIGNED, Bugzie scores higher – we investigate this anomaly below.

Another observation is that the window-based frequency baseline can be quite hard to beat: In three out of the six cases, the minibatch SVM model is no better than the baseline. Bugzie sometimes performs quite well, but for Chromium SUBCOMPONENT and Firefox ASSIGNED it scores below the baseline.

Regarding the quality of the different datasets, an interesting indicator is the relative error reduction by the best model over the baseline (see Table 2). It is especially hard to extract meaningful information about the labeling from the inputs on the Firefox ASSIGNED dataset. One possible cause of this can be that the assignment labeling practices in this project are not consistent: this impression seems to be born out by informal inspection.

Dataset		RER
Chromium	SUB	0.36
Android	SUB	0.38
Chromium	AS	0.21
Android	AS	0.19
Firefox	AS	0.16
Launchpad	AS	0.49

Table 2: Best model's error relative to baseline on the development set

Task	**Model**	**MRR**	**Acc**
Chromium	Window	0.5747	0.3467
	SVM	0.5766	0.4535
	Perceptron	0.5793	0.4393
	Bugzie	0.4971	0.2638
	Regression	**0.7271**	**0.5672**
Android	Window	0.5209	0.3080
	SVM	0.5459	0.4255
	Perceptron	0.5892	0.4390
	Bugzie	0.6281	0.4614
	Regression	**0.7012**	**0.5610**

Table 3: SUBCOMPONENT evaluation results on test set.

On the other hand as the scores in Table 2 indicate, Chromium SUBCOMPONENT, Android SUBCOMPOMENT and Launchpad ASSIGNED contain enough high-quality signal for the best model to substantially outperform the label frequency baseline.

On Launchpad ASSIGNED Regression SGD performs worse than Bugzie. The concept drift plot for these data suggests one reason: there is very little change in class distribution over time as compared to the other datasets. In fact, even though the issue reports in Launchpad range from year 2005 to 2011, the more recent ones are heavily overrepresented: 84% of the items in the development data are from 2011. Thus fast adaptation is less important in this case and Bugzie is able to perform well.

On the other hand, the reason for the less than stellar score achieved with Regression SGD is due to another special feature of this dataset: it has by far the largest number of labels, almost 2,000. This degrades the performance for the one-vs-all scheme we use with SGD Regression. Preliminary investigation indicates that the problem is mostly caused by our application of the "hash-

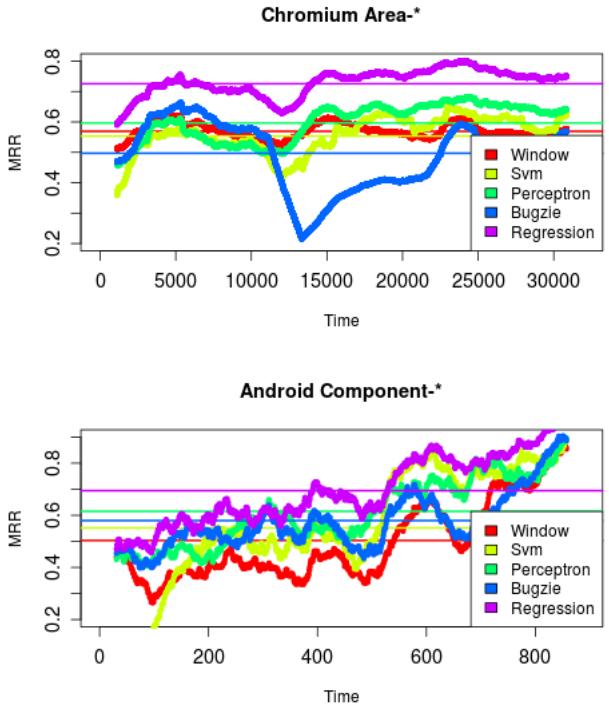

Figure 3: SUBCOMPONENT evaluation results on the development set

Task	Model	MRR	Acc
Chromium	Window	0.0999	0.0472
	SVM	0.0908	0.0550
	Perceptron	0.1817	0.1128
	Bugzie	0.2063	0.0960
	Regression	**0.3074**	**0.2157**
Android	Window	0.3198	0.1684
	SVM	0.2541	0.1684
	Perceptron	0.3225	0.2057
	Bugzie	0.3690	0.2086
	Regression	**0.4446**	**0.2951**
Firefox	Window	0.5695	0.4426
	SVM	0.4604	0.4166
	Perceptron	0.5191	0.4306
	Bugzie	0.5402	0.4100
	Regression	**0.6367**	**0.5245**
Launchpad	Window	0.0725	0.0337
	SVM	0.1006	0.0704
	Perceptron	0.3323	0.2607
	Bugzie	**0.5271**	**0.4339**
	Regression	0.4702	0.3879

Table 4: ASSIGNED evaluation results on test set

ing trick" to feature-label pairs (see section 4.4), which leads to excessive collisions with very large label sets. Our current implementation can use at most 29 bit-sized hashes which is insufficient for datasets like Launchpad ASSIGNED. We are currently removing this limitation and we expect it will lead to substantial gains on massively multi-class problems.

In Tables 3 and 4 we present the overall MRR results on the test data streams. The picture is similar to the development data discussed above.

5 Discussion and related work

Our results show that by choosing the appropriate learner for the scenario of learning from data streams, we can achieve much better results than by attempting to twist batch algorithm to fit the online learning setting. Even a simple and well-know algorithm such as Perceptron can be effective, but by using recent advances in research on SGD algorithms we can obtain substantial improvements on the best previously used approach. Below we review the research on bug report triage most relevant to our work.

Čubranić and Murphy (2004) seems to be the first attempt to automate bug triage. The authors cast bug triage as a text classification task and use the data representation (bag of words) and learning algorithm (Naive Bayes) typical for text classification at the time. They collect over 15,000 bug reports from the Eclipse project. The maximum accuracy they report is 30% which was achieved by using 90% of the data for training.

In Anvik et al. (2006) the authors experiment with three learning algorithms: Naive Bayes, SVM and Decision Tree: SVM performs best in their experiments. They evaluate using precision and recall rather than accuracy. They report results on the Eclipse and Firefox projects, with precision 57% and 64% respectively, but very low recall (7% and 2%).

Matter et al. (2009) adopt a different approach to bug triage. In addition to the project's issue tracker data, they use also the source-code version control data. They build an *expertise model* for each developer which is a word count vector of the source code changes committed. They also build a word count vector for each bug report, and use the cosine between the report and the expertise model to rank developers. Using this approach (with a heuristic term weighting scheme) they report 33.6% accuracy on Eclipse.

Bhattacharya and Neamtiu (2010) acknowledge the evolving nature of bug report streams and attempt to apply incremental learning methods to bug triage. They use a two-step approach:

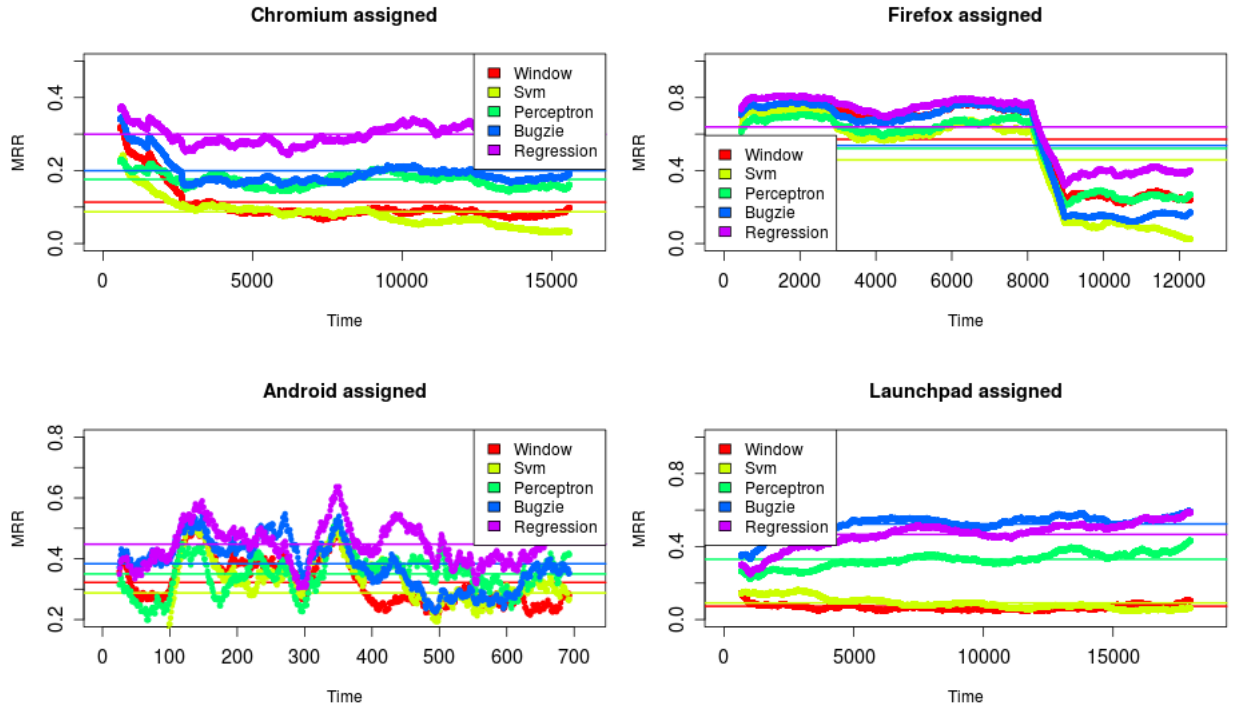

Figure 4: ASSIGNED evaluation results on the development set

first they predict the most likely developer to assign to a bug using a classifier. In a second step they rank candidate developers according to how likely they were to take over a bug from the developer predicted in the first step. Their approach to incremental learning simply involves fully retraining a batch classifier after each item in the data stream. They test their approach on fixed bugs in Mozilla and Eclipse, reporting accuracies of 27.5% and 38.2% respectively.

Tamrawi et al. (2011) propose the Bugzie model where developers are ranked according to the fuzzy set membership function as defined in section 4.4. They also use the label (developer) cache and term cache to speed up processing and make the model adapt better to the evolving data stream. They evaluate Bugzie and compare its performance to the models used in Bhattacharya and Neamtiu (2010) on seven issue trackers: Bugzie has superior performance on all of them ranging from 29.9% to 45.7% for top-1 output. They do not use separate validation sets for system development and parameter tuning.

In comparison to Bhattacharya and Neamtiu (2010) and Tamrawi et al. (2011), here we focus much more on the analysis of concept drift in data streams and on the evaluation of learning under its constraints. We also show that for evolving issue tracker data, in a large majority of cases SGD Regression handily outperforms Bugzie.

6 Conclusion

We demonstrate that concept drift is a real, pervasive issue for learning from issue tracker streams. We show how to adapt to it by leveraging recent research in online learning algorithms. We also make our dataset collection publicly available to enable direct comparisons between different bug triage systems.[1]

We have identified a good learning framework for mining bug reports: in future we would like to explore smarter ways of extracting useful signals from the data by using more linguistically informed preprocessing and higher-level features such as word classes.

Acknowledgments

This work was carried out in the context of the Software-Cluster project EMERGENT and was partially funded by BMBF under grant number 01IC10S01O.

[1] Available from http://goo.gl/ZquBe

References

Anvik, J., Hiew, L., and Murphy, G. (2006). Who should fix this bug? In *Proceedings of the 28th international conference on Software engineering*, pages 361–370. ACM.

Bhattacharya, P. and Neamtiu, I. (2010). Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging. In *International Conference on Software Maintenance (ICSM)*, pages 1–10. IEEE.

Blum, A., Kalai, A., and Langford, J. (1999). Beating the hold-out: Bounds for k-fold and progressive cross-validation. In *Proceedings of the twelfth annual conference on Computational learning theory*, pages 203–208. ACM.

Crammer, K. and Singer, Y. (2002). On the algorithmic implementation of multiclass kernel-based vector machines. *The Journal of Machine Learning Research*, 2:265–292.

Duchi, J., Hazan, E., and Singer, Y. (2010). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*.

Halpin, H., Robu, V., and Shepherd, H. (2007). The complex dynamics of collaborative tagging. In *Proceedings of the 16th international conference on World Wide Web*, pages 211–220. ACM.

Joachims, T. (1999). Making large-scale svm learning practical. In Schölkopf, B., Burges, C., and Smola, A., editors, *Advances in Kernel Methods-Support Vector Learning*. MIT-Press.

Langford, J., Hsu, D., Karampatziakis, N., Chapelle, O., Mineiro, P., Hoffman, M., Hofman, J., Lamkhede, S., Chopra, S., Faigon, A., Li, L., Rios, G., and Strehl, A. (2011). Vowpal wabbit. https://github.com/JohnLangford/vowpal_wabbit/wiki.

Matter, D., Kuhn, A., and Nierstrasz, O. (2009). Assigning bug reports using a vocabulary-based expertise model of developers. In *Sixth IEEE Working Conference on Mining Software Repositories*.

McMahan, H. and Streeter, M. (2010). Adaptive bound optimization for online convex optimization. *Arxiv preprint arXiv:1002.4908*.

Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386.

Tamrawi, A., Nguyen, T., Al-Kofahi, J., and Nguyen, T. (2011). Fuzzy set and cache-based approach for bug triaging. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 365–375. ACM.

Tsymbal, A. (2004). The problem of concept drift: definitions and related work. *Computer Science Department, Trinity College Dublin*.

Voorhees, E. (2000). The TREC-8 question answering track report. *NIST Special Publication*, pages 77–82.

Weinberger, K., Dasgupta, A., Langford, J., Smola, A., and Attenberg, J. (2009). Feature hashing for large scale multitask learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 1113–1120. ACM.

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

Zhang, T. (2004). Solving large scale linear prediction problems using stochastic gradient descent algorithms. In *Proceedings of the twenty-first international conference on Machine learning*, page 116. ACM.

Čubranić, D. and Murphy, G. C. (2004). Automatic bug triage using text categorization. In *In SEKE 2004: Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering*, pages 92–97. KSI Press.