The Thirty-Second AAAI Conference
on Artificial Intelligence (AAAI-18)

Batchwise Patching of Classifiers

Sebastian Kauschke,[1,2] **Johannes Fürnkranz**[1]
[1] Knowledge Engineering Group, [2] Telecooperation Group
TU Darmstadt, Germany
{kauschke, fuernkranz}@ke.tu-darmstadt.de

Abstract

In this work we present classifier patching, an approach for adapting an existing black-box classification model to new data. Instead of creating a new model, patching infers regions in the instance space where the existing model is error-prone by training a classifier on the previously misclassified data. It then learns a specific model to determine the error regions, which allows to patch the old model's predictions for them. Patching relies on a strong, albeit unchangeable, existing base classifier, and the idea that the true labels of seen instances will be available in batches at some point in time after the original classification. We experimentally evaluate our approach, and show that it meets the original design goals. Moreover, we compare our approach to existing methods from the domain of ensemble stream classification in both concept drift and transfer learning situations. Patching adapts quickly and achieves high classification accuracy, outperforming state-of-the-art competitors in either adaptation speed or accuracy in many scenarios.

1 Introduction

In practical classifications, one occasionally faces a scenario where a given classification model needs to be adapted to a changing environment, but neither can the model itself be modified nor can it be re-trained because the necessary expert knowledge or training data are not available. For example, consider the need for *adaptation of legacy or expert models*, which have often been developed and successfully deployed over decades, so that the required expertise for re-programming them is no longer available. Moreover, they may be implemented in hardware, so that it may be infeasible and impractical to reprogram the entire system.

Another application scenario is the *specialization of universal models*. Often, classification models have been developed for a general setting, but need to be refined to a particular context. A typical case is the personalization of a general user model to an individual user. The general model is trained on historical data gathered from many users, and therefore gives a good approximation of the average person's behavior. However, when considering the specific preferences, needs or abilities of a single person, it is possible to improve the model's predictions. Of course, the adaptation should only occur where necessary and helpful, and should in general not hamper the performance of the original model. A last example scenario is *efficient model re-use*. In deep learning, substantial amounts of data and computational power is needed to train a model. For pattern recognition, such models are then often re-used in slightly different contexts by taking the first layers of a generally trained image classification network and re-training only the final layer for a new classification task. However, this technique is specific to a neural network architecture, whereas we aim for a generally applicable method.

In this paper, we propose classifier patching, a technique which allows to adapt general black-box classification models to a new context. The key idea behind this approach is to train classifiers that identify the regions of the instance space in which adaptations are needed, and then train local classifiers for these regions. We assume a setting where batchwise labels of incoming examples are available. The adaptation is triggered when a decay in the performance of the base model is detected, with the goal of finding local patches to the global classification model that act in a flexible and efficient way without having to re-train the model from scratch.

The structure of this document is as follows. In Section 2, we formalize the problem, explain the patching approach, and relate it to previous work in concept drift and transfer learning in Section 3. We then describe our evaluation scenario and briefly explain the datasets and algorithms we compare our approach against (Section 4). We give a summary of the experimental results in Section 5, and conclude our work in Section 6.

2 Patching Classifiers

In the following, we introduce the *Patching* framework, starting with a formal problem description in Section 2.1.

2.1 Problem Description

We assume a general instance space D of instances x with labels $l(x)$, and a black-box classifier C_0. C_0 is immutable and inscrutable and is able to classify the examples of D well, i.e., with a high probability $\Pr(C_0(x) = l(x))$. We now receive new batches of examples $D_i, i > 0$, for which C_0 makes imperfect predictions, presumably because the la-

Copyright © 2018, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

beling function l_i underlying D_i is slightly different from the function l, which C_0 is approximating.
Formally, the assumption is that

$$\Pr(C_0(x) = l_i(x)) \leq \Pr(C_0(x) = l(x)).$$

Our goal is to learn classifiers C_i that approximate l_i as close as possible.

2.2 The Patching Approach

A straight-forward way of addressing this problem is to directly train a classifier $C_i = f(D_i)$ from the instances of D_i. However, this approach is quite wasteful in terms of training examples because it requires to re-train a complete classifier from a sufficient number of samples. If $|D_i|$ is rather small, we can expect the classification performance of C_i to be inferior to the performance of a suitable combination P_i of the classifiers C_0 and C_i, because C_0 is trained on a much larger set of instances. Formally, we expect that there is

$$\Pr(P_i(x) = l_i(x)) \geq \Pr(C_i(x) = l_i(x)).$$

For constructing such a classifier P_i, our approach aims at combining C_0 and C_i in such a way, that the resulting ensemble improves the performance over C_0 and C_i alone. This is achieved in two steps:

(i) We train a binary classifier E_i that is able to identify one or more *error regions* $R_{i,j}$, in which C_0 misclassifies data of D_i (illustrative example shown in Figure 1).

(ii) We train a new classifier $C_{i,j} = f(R_{i,j} \mid C_0)$, a so-called *patch*, for each such region.

These two steps are further explained below. Optionally, the original prediction of C_0 can be added as an additional attribute to both the error region and the patch learning steps. At classification time for batch D_i, the patching classifier P_i first uses E_i to determine whether x lies in one of the error regions $R_{i,j}$, and then uses the corresponding classifier $C_{i,j}$ for classification. If x does not lie in any error region (i.e., if $E_i(x) = 0$), the classifier uses C_0 for classifying the example. More formally,

$$P_i(x) = \begin{cases} C_{i,j}(x) & \textbf{if } E_i(x) = 1 \land x \in R_{i,j}(x), \\ C_0(x) & \textbf{if } E_i(x) = 0. \end{cases}$$

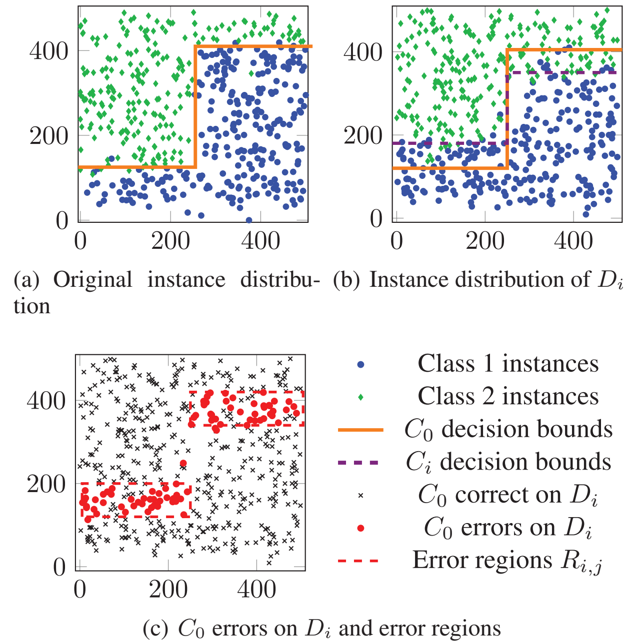
2.3 Learning Error Regions and Patches

After receiving a new batch of examples D_i, we train a classifier that learns in which part of the instance space the base classifier is likely to err, based on the sample D_i. Therefore, we define a new training set consisting of all examples $x \in D_i$, which are labeled with $l_i(x)$ in D_i, and which are now re-labeled as

$$e_i(x) = \mathbb{1}(C_0(x) \neq l_i(x)).$$

In principle, any classifier E_i can be trained on this dataset to predict the errors of C_0 on D_i. However, we assume a tree-based or rule-based classifier, which divides the instance space into smaller regions $R_{i,j}$.

Each *error region* $R_{i,j}$ corresponds to a single rule (or the leaf of a decision tree) of E_i that predicts 1, i.e., predicts that



(a) Original instance distribution (b) Instance distribution of D_i

(c) C_0 errors on D_i and error regions

Figure 1: Learning error regions on a 2-dimensional dataset with a base classifier C_0 and instances from a different distribution D_i

all examples of D_i covered by this rule will be misclassified by C_0. Rules that predict 0 are ignored, as they identify regions where C_0 still operates well.

In order to learn *patches* for the error regions $R_{i,j}$, we train one new classifier $C_{i,j}$ for each region using all training examples $(x, l_i(x)) \in D_i$ where $x \in R_{i,j}$. This classifier now serves as the predominant classifier for the decision space region determined by $R_{i,j}$.

3 Related Work

The idea of patching is related to several well-known concepts in machine learning, in particular the work on transfer learning and concept drift. In the following sections we will experimentally compare it to previous work in these areas. Unlike these works, we assume a fixed, immutable base classifier, and our goal is not to re-learn or adapt this classifier, but to track and model changes in the data *relative* to this base classifier.

Overviews of work in transfer learning can be found in (Torrey and Shavlik 2009; Pan and Yang 2010). The main goal of transfer learning is to apply the knowledge of a source task that has already been learned to a new target task (Torrey and Shavlik 2009). The knowledge from the source task is usually combined with additional information—e.g. from the target task or a third, related task—to improve learning in the target task.

In this work, we relate to specific experiments that were conducted in (Dai et al. 2007; Gao et al. 2008; Pan and Yang 2010) for comparison. Those experiments are located in the domain of inductive transfer learning with source and target tasks being different, but related.

A seminal paper on concept drift is authored by Widmer and Kubat (1996), introducing the FLORA family of algorithms to deal with various types of changing concepts in a stream of data.

An overview of concept drift adaptation, its assumptions, and its goals can be found in Gama et al. (2014). Concept drift can be classified in multiple dimensions: duration, type of transition, re-occurrence, etc. (Webb et al. 2016). The general assumption to deal with concept drift affected classification is that the true label will be available after some time, and can then be used to enhance the classifier for future predictions. Recent work on concept drift is centered on classification of massive amounts of stream data, where fast processing and reduction of overall resources is paramount (Aggarwal 2014).

The concept of *Patching* is quite similar to ensemble methods such as stacking (Wolpert 1992; Ting and Witten 1999), where the idea is to collectively correct the predictions of multiple classifiers by training a meta classifier that combines their predictions. Most related to our technique are arbitrating (Ortega, Koppel, and Argamon 2001) and grading (Seewald and Fürnkranz 2001), which both already feature the idea of training separate classifiers that indicate where the base classifiers in an ensemble err. However, these classifiers are then employed for filtering the predictions in an ensemble, whereas we train a separate classifier on the error regions. In this respect it is also related to boosting or additive logistic regression (Friedman, Hastie, and Tibshirani 2000), but the setting differs in that there, multiple iterations are performed on the same data, whereas the goal here is the adaptation to new data. It is also similar to reframing (Hernández-Orallo et al. 2016), where the goal is to have a unspecific general model that can be specialized for a particular task. Finally, patching may also be viewed as an instance of exceptional model mining (Duivesteijn, Feelders, and Knobbe 2016), in that it also focuses on recognizing and modeling differences to a base model.

Due to the fact that our method uses an ensemble of classifiers for data that occurs in batches, we rely on the extensive survey by Gomes et al. (2017) to choose appropriate algorithms to compare against.

4 Experimental Setup

In this section, we elaborate on the experiment setup. We conduct experiments in three scenarios: We use *Patching* as intended, in a scenario with a given classifier that is patched based on the knowledge gained from new instances. Additionally, we apply it on *concept drift* and *transfer learning* tasks.

We use the Massive Online Analysis (MOA) framework[1] (Bifet et al. 2010) and the WEKA toolkit[2] (Witten and Frank 2005) for machine learning, thereby simulating a real-world scenario, where instances arrive one by one and labels are available in batches some time after. From a data stream, we preserve multiple batches of examples D_i, together with their respective labels $l_i(x)$ for the learning steps. We allow the *Patching* algorithm to store the most recent n batches, all of which will be used to learn the error regions and respective patches. Different sizes of n yield different purposes: small n are required for quicker adaptation, but larger n usually result in a better accuracy in the long run. In our experiments we used a compromise value of $n = 8$ which yielded overall adequate results in preliminary runs.

[1] http://moa.cms.waikato.ac.nz

[2] http://www.cs.waikato.ac.nz/ml/weka/

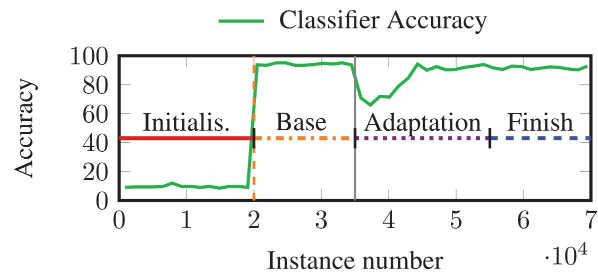


Figure 2: Phases during the course of the stream. Change points are shown as gray vertical lines (here at point 3.5), the initialization phase ends at the dashed vertical line.

We use a chunk/batch-based evaluation method to retrieve the performance of the classifier every m instances, for which we split each dataset into a certain amount of batches (Tab. 1). This way, we can use the whole dataset for training and evaluation without having to rely on a separate hold-out set. The classifiers can use previously evaluated instances to incrementally expand their learning base.

In order to achieve this type of evaluation, we rely on the *EvaluateInterleavedChunks* method provided by MOA. A typical run of patching consists of four phases (illustrated in Fig. 2):

- *Initialization (Init):* In the first phase, we create the base classifier. In the evaluation part we will not show this phase, because it is irrelevant to our findings.
- *Base:* In this phase, the dataset remains the same as in the *Init* phase. The patching algorithm will start to collect batches of instances, learn errors and update regularly.
- *Adaptation:* This phase starts with the first *change point* (CP), where one or multiple changes in the data occur. It ends when the performance has stabilized after the changes.
- *Finish:* In the final phase, the concepts remain stable. It is later used to calculate the final performance estimates that the classifiers can achieve.

This setup represents a real-world usage scenario of classification on batches of instances. For the *transfer-learning* experiments we aim at getting insight into how many instances are required to reach a certain performance level compared to the original data distribution. The *Patching* scenario skips the initialization altogether, and starts at the change point. In this scenario quick adaptation is the key performance indicator. Here, the benchmark algorithms we compare against start from scratch, whereas *Patching* has the benefit of a given classifier (constructed from the data in the Initialization phase).

4.1 The Datasets

We evaluate the patching approach in a variety of different scenarios, encompassing it's intended use as well as *concept drift* and *transfer learning*.

For the *Patching* scenario, we create experiments based on *MNIST*[3], to which we introduce various changes. We also use the *20 newsgroups*[4] dataset, which originates from the domain of *transfer learning* (Dai et al. 2007; Gao et al. 2008). In the *transfer learning* and *concept drift* experiments, we also rely on modified *MNIST* variants. Additionally, the *SEA concepts* (Street and Kim 2001) and a dataset created from a *rotating hyperplane* as described in Hulten, Spencer, and Domingos (2001) are used for the *concept drift* domain.

All datasets are explained below and described in Table 1 in detail. It shows a summary of all studied datasets, the type of drift that occurs (Webb et al. 2016), and the number of change points. The *Init* value specifies the number of instances used for the *Initialization* phase, *Total* specifies the total number of instances, and *Chunks* is the number of batches the dataset is divided into.

The MNIST Dataset The first dataset is the *MNIST* dataset of handwritten digits. It consists of a total of 70,000 digits. We are randomizing the order, use it as a stream and introduce four different variants of change. For the first scenario, $MNIST_{merge}$, the labels of the stream are changed such that classes 4,5,7 and 9 are all labeled as 9 at the change point. For the second scenario $MNIST_{appear}$, the labels 3,5,7,9 do not exist during the initialization, and appear at the change point. In $MNIST_{flip}$ the pixel values are changed, so that the written digits are flipped both horizontally and vertically. Finally, in $MNIST_{switch}$ the classes of digits are switched such that 2 becomes 4 and 3 becomes 5 and vice versa.

[3]http://yann.lecun.com/exdb/mnist/

[4]http://www.iesl.cs.umass.edu/data

Table 1: Summary of the datasets used in the experiments

Dataset	Type	Init	Total	Chunks	CPs
Patching Domain					
$MNIST_{split}$	—	—	1000	100	—
$MNIST_{switch}$	—	—	1000	100	—
$MNIST_{appear}$	—	—	1000	100	—
$20NG_{R/T}$	—	—	1000	100	—
$20NG_{R/S}$	—	—	1000	100	—
Transfer Domain					
$MNIST_{flip}$	incr.	20k	140k	200	1
$20NG_{R/T}$	a, p	1600	4340	22	1
$20NG_{S/T}$	a, p	1600	4324	22	1
$20NG_{R/S}$	a, p	1600	4762	24	1
Concept Drift Domain					
SEA_{lin}	a, i	100k	500k	200	2
SEA_{multi}	a, i	100k	500k	200	2
RotHyp	g, t	100k	500k	100	—
$MNIST_{merge}$	a, p	20k	70k	100	1
$MNIST_{appear}$	a, p	20k	70k	100	1
$MNIST_{switch}$	a, p	20k	70k	100	1
Drift Duration: a = abrupt, g = gradual					
Transition: i = incremental, t = transient, p = permanent					

20 Newsgroups The second dataset that we will use in our evaluation is *20 Newsgroups*. This dataset consists of newsgroup entries and contains two hierarchical category attributes, one being the top-category and the second being the subcategory. We derive a transfer learning task from this dataset, such that the goal is to classify the top-category. Therefore, we vectorize the text part of the dataset and split it based on the subcategories. This means, that if A and B are top-categories, and each has two subcategories A_1, A_2 and B_1, B_2, we split it such that A_1 and B_1 will be in the training set, whereas A_2 and B_2 are put in the test set. Therefore, the training and test sets are made up from different subcategories and will show a varying distribution. We derived three datasets $20NG_{R/S}$ (Top category REC vs SCI), $20NG_{R/T}$ (REC vs TALK), $20NG_{S/T}$ (SCI vs TALK).

The SEA Dataset The SEA concepts dataset is a dataset with abrupt concept drift (Street and Kim 2001) consisting of three attributes, where the attributes are used to generate the resulting binary label. All three attributes are real-valued and have random values between 0 and 10.

We generate two variants of this dataset: SEA_{lin} with a linear and SEA_{multi} with a multiplicative relation between attributes and class label. A threshold value θ will be changed during the course of the instance stream to introduce concept change. We generate a stream of 500,000 instances with 2 change points per dataset. Furthermore, 10% class noise is added.

Rotating Hyperplane The last dataset we use is based on a rotating hyperplane in a d-dimensional space as proposed in (Hulten, Spencer, and Domingos 2001), with which a binary stream classification problem can be constructed. The hyperplane is defined by the set of points x that satisfy $\sum_{i=1}^{d} w_i x_i = w_0$ where x_i is the i-th attribute of x. The class for each instance is determined as such: If $\sum_{i=1}^{d} w_i x_i \geq w_0$, the class is *positive*, otherwise it is *negative*. By changing the weights w_i, the orientation and position of the hyperplane can be changed. We generate a stream *RotHyp* of 500,000 instances with 10 numeric attributes and introduce a slow movement of the hyperplane. Thereby we simulate a continuous gradual shift in the problem space.

4.2 The Patching Environment

The *Patching* algorithm builds an ensemble of classifiers, consisting of three classification steps: (i) the base classifier C_0, (ii) the error region classifiers E_i and (iii) the patches $C_{i,j}$. Our implementation allows any WEKA classifier to be used for each of the steps. In our experiments, we primarily use random forests (RF) with 100 random trees, mostly because this is a fast algorithm and gives good results without requiring extensive parameter optimization.

As a base classifier, we train a random forest on all instances from the initialization phase.

In order to learn the error region classifiers, random forests only give binary information, which does not allow us to identify multiple error regions. For this reason, we us a version of the RIPPER (Cohen 1995) rule learning algorithm, specifically modified to determine by which of its rules an instance has been classified, allowing us to specify precise error regions by using each triggered rule as a region. For some problems, this works remarkably well, outperforming the binary error regions. However, in general scenarios the binary variant performs equally good, most likely a result of the random forests behavior.

The entire patching framework is implemented in MOA and publicly available on GitHub [5].

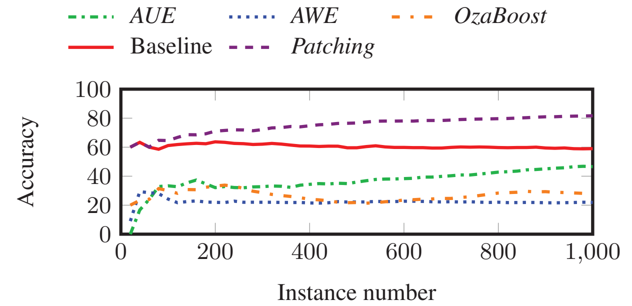
4.3 Benchmark Algorithms

In this work, we compare our algorithm against some state-of-the-art ensemble learning algorithms. In order to ensure comparable results between the algorithms, we rely on random decision trees as the general underlying classifier. In general, *Patching* can use any given classifier as base, error region detector and for the patches. Our goal of parameterizing the algorithms is that they have a maximum of 500 (random) trees in their ensemble to achieve comparable results. When we use RIPPER as error region detector, we can not guarantee a maximum number of error regions and hence patches. Therefore we decided to use random forests (with 100 trees each) for error regions and the patch. *Patching* can be configured to keep multiple batches in a FIFO queue. We keep the most recent 8 batches for the *concept drift* and *transfer* problems, and all batches for the *Patching* scenario, where the batch size is very small and all gathered information is crucial. The base classifier for *Patching* is trained as a random forest with 100 random trees.

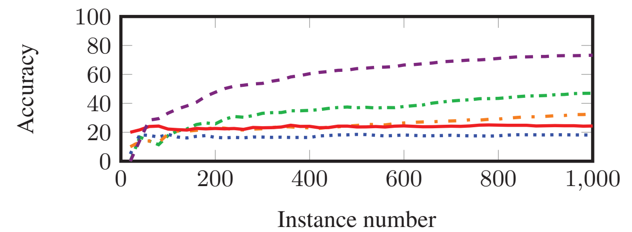
The accuracy-weighted ensemble (*AWE*) by (Wang et al. 2003) constructs an ensemble of weighted classifiers based on their accuracy w.r.t. a time-evolving environment. The accuracy-updated ensemble (*AUE*) by (Brzeziński and Stefanowski 2011) extends this idea such that updates based on the current distribution are possible as well as it fixes some problems with *AWE*. Furthermore, we compare our Algorithm with *OzaBoost* (Oza 2005), which is essentially an online-version of boosting for data-streams that can also be used in a batch-scenario. We also looked into *DACC* (Jaber, Cornuéjols, and Tarroux 2013), but appropriate parametrization was difficult and therefore we excluded it from the experimental comparison. For *AWE*, we set the ensemble size to 5 random forests, each consisting of 100 random trees. *AUE* is configured similarly, with a maximum of 500 random trees. *OzaBoost* is allowed to use 500 random Hoeffding Trees (Domingos and Hulten 2000), since it does not support generic random trees.

Since there is—to our knowledge—no transfer learning algorithm that deals with the transfer learning problem as a streaming situation, we will also apply said algorithms for the transfer learning experiments. In our experiments, we choose the baseline performance to be a classifier that is

[5]https://github.com/Shademan/Patching/releases/tag/AAAI2018



(a) Accuracy of $MNIST_{appear}$



(b) Accuracy of $MNIST_{flip}$

Figure 3: Sample results of stream classification accuracy for problems that resemble the intended use of patching

trained on the data of the initialization phase and has no capabilities to adapt during the course of the stream. It is also a random forest consisting of 100 random trees.

4.4 Evaluation Measures

For the comparison of the algorithms we use the following metrics:

- *Final Accuracy (F.Acc.):* Classification accuracy, measured in the second half of the *Finish* phase
- *Recovery Speed (R.Spd):* Number of instances that a classifier requires during the *Adaptation* phase to achieve 95% of its final accuracy.
- *Adaptation Rank (Ad.Rk):* Average Rank of the classifier during the *Adaptation* phase.
- *Final Rank (F.Rk):* Average Rank of the classifier during the *Finish* phase.

We ran each algorithm 10 times on every dataset with different random seeds and averaged over the results for the 10 runs. The standard deviation of accuracy in those 10 runs was smaller than 2%.

5 Results

In this section, we give an overview of our experimental results for the datasets described in the previous section. We treat all the problems as stream problems, which allows us to measure how quickly a classifier reacts to concept change, or how much additional information it needs to adapt to a transfer situation. Furthermore, we show exemplary graphs of the adaptation behavior for selected datasets.

Table 2: Experiment results for the patching scenario

Classifier	F.Acc	R.Spd	Ad.Rk	F.Rk
	***MNIST**$_{appear}$*			
Baseline	59.41	—	1.96	2.00
AUE	43.83	$> 20k$	3.24	3.00
AWE	25.56	20	4.60	5.00
OzaBoost	37.18	360	4.16	4.00
Patching$_{RF\text{-}RF\text{-}all}$	**81.85**	220	**1.04**	**1.00**
	***MNIST**$_{merge}$*			
Baseline	70.71	—	1.96	2.00
AUE	48.12	100	4.24	3.00
AWE	42.51	20	3.48	5.00
OzaBoost	44.37	80	4.28	4.00
Patching$_{RF\text{-}RF\text{-}all}$	**86.66**	20	**1.04**	**1.00**
	***MNIST**$_{flip}$*			
Baseline	24.55	—	3.48	4.00
AUE	45.03	$> 20k$	2.68	2.00
AWE	22.28	20	4.68	5.00
OzaBoost	28.65	140	3.00	3.00
Patching$_{RF\text{-}RF\text{-}all}$	**74.19**	$> 20k$	**1.16**	**1.00**
	***MNIST**$_{switch}$*			
Baseline	59.73	—	1.84	2.00
AUE	43.68	$> 20k$	3.20	3.00
AWE	25.35	20	4.64	5.00
OzaBoost	37.17	360	4.16	4.00
Patching$_{RF\text{-}RF\text{-}all}$	**83.35**	320	**1.16**	**1.00**
	20***NG***$_{R/S}$			
Baseline	67.18	—	2.08	2.00
AUE	61.26	260	3.88	4.00
AWE	56.04	0	4.48	5.00
OzaBoost	65.08	440	3.48	3.00
Patching$_{RF\text{-}RF\text{-}all}$	**73.09**	0	**1.08**	**1.00**
	20***NG***$_{R/T}$			
Baseline	63.65	—	4.64	5.00
AUE	68.75	0	3.08	3.45
AWE	68.56	0	3.28	3.55
OzaBoost	**82.01**	160	**1.52**	**1.00**
Patching$_{RF\text{-}RF\text{-}all}$	77.83	240	2.48	2.00

5.1 Patching Scenario

In Table 2 we show the results of the datasets which put *Patching* to it's intended use: Leveraging a given model and adapt to changes relative to it. As we can see, *Patching* excels in almost all scenarios w.r.t. the adaptation rank (Ad.Rk) and the final rank (F.Rk). The recovery speed (R.Spd) values may be misleading, since the final accuracy of *Patching* is higher for most datasets. The examples in Figure 3 demonstrate the behavior over the course of the instances. As can be seen, *Patching* shows both a better start and quicker adaptation, since it can leverage the given classifier. Although in some settings, it does not improve significantly from it. Overall, *Patching* achieves the highest rank for adaptation and final accuracy in five of six datasets.

5.2 Transfer Learning Datasets

The results for transfer learning are shown in Table 3. Each of the applied algorithms shows strengths and weaknesses regarding both adaptation speed and final performance. *Patching* performs well overall, but not significantly better (cf. Fig. 4), the only exception being 20*NG*$_{S/T}$.

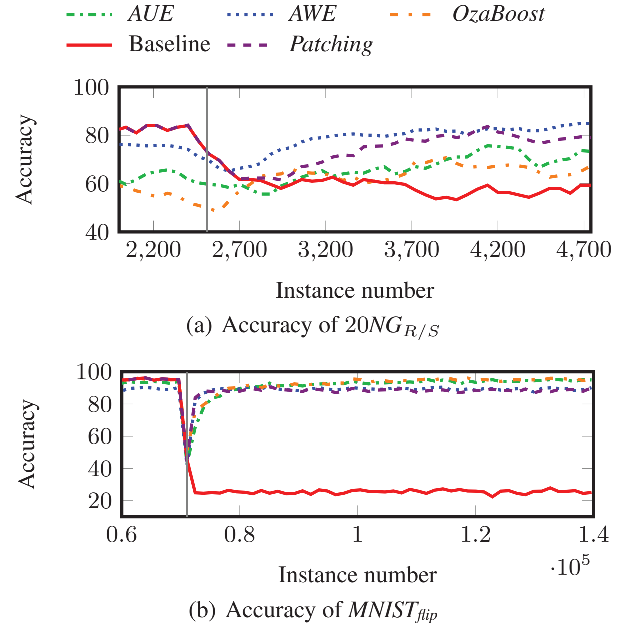


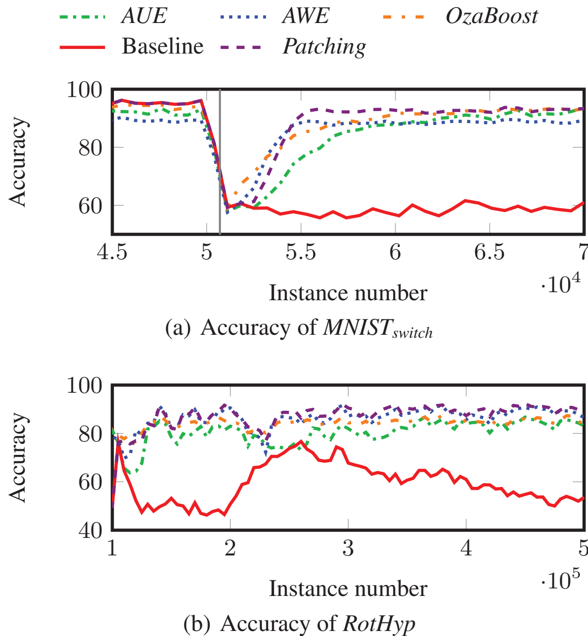
Figure 4: Exemplary results of stream classification accuracy for transfer problems

Table 3: Experiment results in the transfer domain

Classifier	F.Acc	R.Spd	Ad.Rk	F.Rk
	***MNIST**$_{flip}$*			
Baseline	23.79	—	4.60	5.00
AUE	**93.62**	3500	3.00	**1.19**
AWE	91.24	1400	**1.60**	2.29
OzaBoost	89.28	5600	3.53	4.00
Patching$_{RF\text{-}RF\text{-}8}$	91.39	2100	2.27	2.52
	20***NG***$_{R/S}$			
Baseline	64.36	—	3.33	5.00
AUE	71.70	360	3.78	3.10
AWE	**91.79**	240	**1.37**	**1.00**
OzaBoost	67.73	300	4.44	3.90
Patching$_{RF\text{-}RF\text{-}8}$	85.03	720	2.07	2.00
	20***NG***$_{S/T}$			
Baseline	48.75	—	4.52	5.00
AUE	79.22	300	2.38	3.50
AWE	78.81	180	2.62	3.50
OzaBoost	86.78	540	**2.19**	2.00
Patching$_{RF\text{-}RF\text{-}8}$	**91.07**	780	3.29	**1.00**

5.3 Concept Drift Datasets

Table 4 shows the results for all concept drift-related datasets. As we can see, *Patching* performs well in adaptation rank where it achieves the highest rank 4 out of 6 times. In final rank, however, *Patching*, *OzaBoost* and *AUE* perform very similar with no significant differences. It has to be mentioned that the other ensemble methods are continuously improving algorithms and might start to outperform *Patching* at some point, given more instances (Fig. 5).



(a) Accuracy of $MNIST_{switch}$

(b) Accuracy of *RotHyp*

Figure 5: Exemplary results of stream classification accuracy for concept drift problems

5.4 Statistical Evaluation

In order to show that *Patching* has significant advantages, we applied the non-parametrical Friedmann test (Friedmann 1937) to our average ranks of both adaptation and final accuracy, followed by the Nemenyi post-hoc test (Nemenyi 1963) as shown in (Demšar 2006) to assure pairwise significance and calculate critical distances. For this calculation we removed the *Baseline* from the ranking, since it distorts the Friedmann test on the null-hypothesis that all algorithms are equally good. For our rankings, the critical distance is 1.22. Regarding the final ranks, *Patching* is significantly better than *AWE* (distance 1.77) and *OzaBoost* (distance 1.73) and in the same group as *AUE* (distance 0.9). For the adaptation rank the distances are similar (*AUE*:1.2, *AWE*:1.23, *OzaBoost*:1.43), which means that *Patching* performs significantly better compared to *AWE* and *OzaBoost* while *AUE* is just within the range of non-significance. The statistical evaluation was performed on all ranked results.

Table 4: Experiment results in the concept drift domain

Classifier	F.Acc	R.Spd	Ad.Rk	F.Rk
	***MNIST**$_{appear}$*			
Baseline	59.23	—	4.47	5.00
AUE	90.87	2100	3.00	2.67
AWE	90.94	2800	2.80	2.33
OzaBoost	87.23	1400	3.67	4.00
Patching$_{RF\text{-}RF\text{-}8}$	**93.21**	1400	**1.07**	**1.00**
	***MNIST**$_{switch}$*			
Baseline	59.54	—	3.89	5.00
AUE	91.42	4200	3.44	2.47
AWE	91.23	2100	2.33	2.53
OzaBoost	88.46	4200	3.33	4.00
Patching$_{RF\text{-}RF\text{-}8}$	**93.77**	3500	**2.00**	**1.00**
	RotHyp			
Baseline	53.02	—	4.90	5.00
AUE	84.78	10000	3.90	4.00
AWE	89.96	5000	2.24	1.64
OzaBoost	88.11	0	**1.62**	3.00
Patching$_{RF\text{-}RF\text{-}8}$	**89.98**	15000	2.33	**1.36**
	SEA$_{multi}$			
Baseline	85.59	—	4.84	5.00
AUE	**99.38**	0	2.35	**1.00**
AWE	98.20	0	2.32	3.14
OzaBoost	97.71	0	3.94	3.81
Patching$_{RF\text{-}RF\text{-}8}$	98.92	0	**1.55**	2.05
	SEA$_{lin}$			
Baseline	67.35	—	4.90	5.00
AUE	**99.81**	5000	2.20	1.62
AWE	99.54	0	2.56	2.81
OzaBoost	95.20	0	3.83	4.00
Patching$_{RF\text{-}RF\text{-}8}$	99.78	5000	**1.51**	**1.57**

6 Conclusion

In this work, we have introduced *Patching*, i.e., the idea of learning local corrections to existing classifiers, thereby eliminating the need for re-learning an existing classifier. We have shown experimentally that classifier patching works well for its intended use: scenarios where the classifier can leverage a pre-existing model for new situations, where (partial) adaptation is required and regions of erroneous instances can be determined and patched. Because of its design, it adapts faster in these scenarios than the benchmark algorithms. *Patching* can also be applied to scenarios in *transfer learning* or situations of *concept drift*, where it manages to rival state-of-the-art competitors in either adaptation speed or accuracy in many scenarios. However, for massive online analysis, where classification is learned from zero, we recommend the state-of-the-art ensemble algorithms that are mentioned in this paper for their focus on computationally efficient behavior.

In the future, we will improve *Patching* in order to achieve better computational performance and improve upon some issues we encountered. For example, a major issue is the parameter for the number of kept batches. Choosing this parameter optimally affects both the adaptation speed and the final performance. By adaptive windowing techniques and batch weighting we can probably improve the speed while simultaneously increasing the final accuracy. We are also looking into other methods for learning the error regions, more specifically clustering algorithms. One of the main current constraints on computational efficiency is the re-learning of error regions and patches, which we want to address by adaptive error region learning and updateable patch classifiers.

Acknowledgements. We gratefully acknowledge the use of the Lichtenberg high performance computer of the TU Darmstadt for our experiments.

References

Aggarwal, C. C. 2014. A Survey of Stream Classification Algorithms. In *Data Classification: Algorithms and Applications*. CRC Press. 245–273.

Bifet, A.; Holmes, G.; Kirkby, R.; and Pfahringer, B. 2010. MOA Massive Online Analysis. *Journal of Machine Learning Research* 11:1601–1604.

Brzeziński, D., and Stefanowski, J. 2011. Accuracy Updated Ensemble for Data Streams With Concept Drift. *Hybrid Artificial Intelligent Systems* 155–163.

Cohen, W. W. 1995. Fast Effective Rule Induction. In *Proceedings of the 12th International Conference on Machine Learning – ICML '95*, 115–123.

Dai, W.; Xue, G.-R.; Yang, Q.; and Yu, Y. 2007. Transferring Naive Bayes Classifiers for Text Classification. In *Proceedings of the 22nd AAAI Conference on Artificial Intelligence – AAAI '07*, 540–545.

Demšar, J. 2006. Statistical Comparisons of Classifiers Over Multiple Data Sets. *Journal of Machine Learning Research* 7:1–30.

Domingos, P., and Hulten, G. 2000. Mining High-Speed Data Streams. In *Proceedings of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining – KDD '00*, 71–80. ACM.

Duivesteijn, W.; Feelders, A.; and Knobbe, A. J. 2016. Exceptional Model Mining – Supervised Descriptive Local Pattern Mining with Complex Target Concepts. *Data Mining and Knowledge Discovery* 30(1):47–98.

Friedman, J. H.; Hastie, T.; and Tibshirani, R. 2000. Additive Logistic Regression: A Statistical View of Boosting. *The Annals of Statistics* 38(2):337–407.

Friedmann, M. 1937. The Use of Ranks to Avoid the Assumption of Normality Implicit in the Analysis of Variance. *Journal of the American Statistical Association* 22:675–701.

Gama, J.; Žliobaitė, I.; Bifet, A.; Pechenizkiy, M.; and Bouchachia, A. 2014. A Survey on Concept Drift Adaptation. *ACM Computing Surveys* 46(4):44.

Gao, J.; Fan, W.; Jiang, J.; and Han, J. 2008. Knowledge Transfer via Multiple Model Local Structure Mapping. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining – KDD '08*, 283–291.

Gomes, H. M.; Barddal, J. P.; Enembreck, F.; and Bifet, A. 2017. A Survey on Ensemble Learning for Data Stream Classification. *ACM Computing Surveys* 50(2):23:1–23:36.

Hernández-Orallo, J.; Martínez-Usó, A.; Prudêncio, R. B.; Kull, M.; Flach, P.; Ahmed, C. F.; and Lachiche, N. 2016. Reframing in Context: A Systematic Approach for Model Reuse in Machine Learning. *AI Communications* 29(5):551–566.

Hulten, G.; Spencer, L.; and Domingos, P. 2001. Mining Time-Changing Data Streams. In *Proceedings of the 7th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining – KDD '01*, 97–106.

Jaber, G.; Cornuéjols, A.; and Tarroux, P. 2013. Online Learning: Searching for the Best Forgetting Strategy Under Concept Drift. In *Proceedings of the 20th International Conference on Neural Information Processing – ICONIP '13*, 400–408. Springer.

Nemenyi, P. 1963. *Distribution-Free Multiple Comparisons*. Ph.D. Dissertation, Princeton University.

Ortega, J.; Koppel, M.; and Argamon, S. 2001. Arbitrating Among Competing Classifiers Using Learned Referees. *Knowledge and Information Systems* 3(4):470–490.

Oza, N. C. 2005. Online Bagging and Boosting. In *Proceedings of the 2005 IEEE International Conference on Systems, Man and Cybernetics – SMC '05*, 2340–2345.

Pan, S. J., and Yang, Q. 2010. A Survey on Transfer Learning. *IEEE Transactions on Knowledge and Data Engineering* 22(10):1345–1359.

Seewald, A. K., and Fürnkranz, J. 2001. An Evaluation of Grading Classifiers. In *Advances in Intelligent Data Analysis: Proceedings of the 4th International Conference – IDA '01*, 115–124.

Street, W. N., and Kim, Y. 2001. A Streaming Ensemble Algorithm (SEA) for Large-Scale Classification. In *Proceedings of the 7th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining – KDD '01*, 377–382. ACM.

Ting, K. M., and Witten, I. H. 1999. Issues in Stacked Generalization. *Journal of Artificial Intelligence Research* 10:271–289.

Torrey, L., and Shavlik, J. 2009. Transfer Learning. In *Handbook of Research on Machine Learning Applications and Trends: Algorithms, Methods, and Techniques*, volume 1. Information Science Reference. 242–264.

Wang, H.; Fan, W.; Yu, P. S.; and Han, J. 2003. Mining Concept-Drifting Data Streams Using Ensemble Classifiers. In *Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining – KDD '03*, 226–235. AcM.

Webb, G. I.; Hyde, R.; Cao, H.; Nguyen, H. L.; and Petitjean, F. 2016. Characterizing Concept Drift. *Data Mining and Knowledge Discovery* 30(4):964–994.

Widmer, G., and Kubat, M. 1996. Learning in the Presence of Concept Drift and Hidden Contexts. *Machine Learning* 23(1):69–101.

Witten, I. H., and Frank, E. 2005. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann.

Wolpert, D. H. 1992. Stacked Generalization. *Neural Networks* 5(2):241–260.