UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

ALISTER MACHADO DOS REIS

Incremental Learning Applied to Streaming Environments

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação

Trabalho realizado no Institut National Polytechnique de Grenoble - Ensimag dentro do acordo de dupla diplomação UFRGS - INP Grenoble

Orientador: Prof. Dr. Bruno Silva
Co-orientador: Prof. Dr. Jérôme Malick

Porto Alegre
2017

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Rui Vicente Oppermann
Vice-Reitora: Profª. Jane Fraga Tutikian
Pró-Reitor de Graduação: Prof. Wladimir Pinheiro do Nascimento
Diretora do Instituto de Informática: Profª. Carla Maria Dal Sasso Freitas
Coordenador do Curso de Ciência de Computação: Prof. Raul Fernando Weber
Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

RESUMO

A necessidade de extrair conhecimento a partir de dados está presente em muitos campos e sob variadas formas. Um exemplo é quando precisa-se tratar fluxos de dados, sejam eles provenientes de sensores, redes sociais, ou do mercado financeiro, para citar alguns exemplos. SAP, uma multinacional alemã cuja sede principal localiza-se em Walldorf, na Alemanhã, com mais de 300 mil clientes, possui uma solução de software para tratar fluxos de dados chamada *SAP HANA Smart Data Streaming*. Esta solução oferece atualmente somente uma possibilidade de algoritmo para efetuar classificação sobre fluxos de dados. Neste trabalho, investigamos o potencial de técnicas alternativas baseadas no conceito de Aprendizado Incremental. Isso significa que os algoritmos estudados retêm informação adaptando ou expandindo um único modelo através do tempo, ao invés de reconstruir modelos a partir do zero e treiná-los em novos dados. Com base nesses algoritmos alternativos, podemos avaliar a performance dos modelos, bem como *trade-offs* entre acurácia e tempo de treinamento. Critérios de comparação são propostos e instanciados para os exemplos estudados, provendo uma visão geral sobre a aplicabilidade destes métodos para o problema.

Palavras-chave: Inteligência artificial. fluxos de dados. aprendizado incremental. algoritmos de *ensemble*.

Resumo Estendido

Este é um resumo estendido (páginas 5 a 12) em português para a Universidade Federal do Rio Grande do Sul do trabalho original que segue. O trabalho de conclusão original, em inglês, foi apresentado no Institut National Polytechnique de Grenoble, através do programa de dupla diplomação BRAFITEC - EcoSud entre as duas universidades.

1 INTRODUÇÃO

Este trabalho explora a aplicabilidade de algoritmos que usam aprendizado incremental para lidar com problemas de classificação em ambientes de *streaming*. Seu desenvolvimento foi realizado no SAP Labs LLC, na cidade de San Ramon, Califórnia, Estados Unidos da América.

A empresa possui uma ferramenta para analisar fluxos de dados chamada SAP HANA Smart Data Streaming (SAP HANA SDS). Ela fornece ao usuário diferentes adaptadores, com o objetivo de ler dados de múltiplas fontes, como arquivos Hadoop, CSV e *Web Services*. O usuário pode então aplicar diversas transformações sobre os dados lidos, desde consultas no estilo SQL (e.g. junções, filtros, projeções) até algoritmos de Machine Learning.

Um dos problemas clássicos de Machine Learning é o de realizar *classificação* (também chamado de *aprendizado supervisionado*) sobre dados. Este trabalho foca neste problema em específico. Ele se faz presente em variados ambientes, como reconhecimento de escrita, de fala, ou tradução automática. Empresas também usam técnicas de classificação para melhorar ou expandir suas capacidades. Por exemplo, sistemas de recomendação podem ser construídos com estas técnicas (como por exemplo no caso do Netflix). Além disso, estes métodos podem ser usados para detecção de fraudes por bancos, ou também para o desenvolvimento de assistentes digitais.

Entretanto, as técnicas mais conhecidas de classificação geralmente supõem que os dados de treino — que nesse caso, são pares vetor-rótulo, onde o primeiro elemento nos dá as informações que podem ser usadas na predição do segundo — estão organizados em um conjunto de dados fixo. Em algumas aplicações, nosso sistema pode estar recebendo dados incrementalmente através do tempo (ou nosso conjunto de dados pode ser muito grande para caber em memória). Este é o caso quando queremos extrair informações de dados provenientes de leituras de sensores ou do mercado de ações, por exemplo. Uma consequência direta deste fato é que o algoritmo fazendo predições sobre os dados precisa se adaptar automaticamente através do tempo para incluir novas informações, o que conflita com a hipótese de um conjunto de dados fixo. Isso motiva a necessidade de técnicas de Aprendizado Incremental, uma categoria de métodos capazes de fazer justamente isso: aprender através do tempo, mantendo uma versão atualizada do conhecimento contido no fluxo de dados observado.

Atualmente, o produto da SAP oferece duas opções de algoritmos incrementais:

uma árvore de decisão de Hoeffding para classificação, e uma implementação do algoritmo DenStream para *clustering*. Este trabalho se propõe a explorar opções de novos algoritmos de classificação e estudar a possibilidade de sua integração na plataforma. Diferentemente do caso de *clustering*, onde o objetivo é extrair grupos coesos de dados sem informação de classe, em classificação quer-se um algoritmo — também chamado de classificador — que aprende padrões a partir de dados que contém informação de classe. Posteriormente, usa-se este conhecimento para prever a classe de novos exemplos.

Neste trabalho, nós restringimos a classe de algoritmos a serem explorados à dos que utilizam técnicas de Aprendizado Incremental. Esta técnica também já foi aplicada a problemas de aprendizado semi-supervisionado, onde apenas alguns dados possuem informação de classe. Nestes casos, os dados sem classe são usados para inferir a estrutura das classes, enquanto os que possuem classe conhecida nos permitem generalizar esta informação para novos exemplos.

2 APRENDIZADO INCREMENTAL

A definição de Aprendizado Incremental varia na literatura. Apesar disso, podemos usar os pontos principais e mais frequentes para caracterizar este grupo de técnicas: ele compreende algoritmos nos quais aprende-se informação através da expansão ou da adaptação de modelos; além disso, supõe-se que os dados de treinamento não podem ser armazenados para uso posterior; por último, deseja-se que o modelo aprendido desta forma seja similar ao obtido via treinamento por *batches* de dados.

2.1 Aplicação a Streaming

Esta família de técnicas, a princípio, parece adequada para ambientes de streaming, o que motiva nossa escolha em explorá-la. Nesses ambientes, não há a noção de um conjunto de dados fixo, pois dados são gerados ao longo do tempo. Isso complementa a premissa de que não podemos armazenar dados, e também faz com que adaptar ou expandir modelos seja uma alternativa valiosa quando comparada ao treino repetitivo de modelos. E como ajustes incrementais a um modelo seriam intuitivamente menos computacionalemente intensivos do que treinar novos modelos a partir do zero, seria possível efetuar a classificação de novos dados sempre que necessário.

Em contraponto, lidar com fluxos de dados introduz problemas que não estão presentes no cenário mais comum de aprendizado supervisionado. Os que mais se destacam são relacionados a variação de conceito (*concept drift*) ou ao desbalanceamento entre classes. A presença de variação de conceito invalida a hipótese fundamental de vários modelos usados em cenários clássicos de aprendizado de máquina: a de que dados são amostras independentes de uma mesma distribuição de probabilidade. O desbalanceamento entre classes – quando uma classe é muito mais frequente que a outra no conjunto de dados – faz com que classificadores que não generalizam sejam criados com mais facilidade.

Por consequência, começamos avaliando o estado da arte e investigando as técnicas usadas e as medidas por elas tomadas para lidar com estes problemas. Como a literatura em Aprendizado Incremental é diversa, e nosso tempo limitado, escolhemos explorar o que acreditamos ser a técnica de aprendizado por *ensemble* de mais destaque, assim como técnicas baseadas em Máquinas de Vetores de Suporte (*Support Vector Machines*, SVM). Também exploramos o uso de Redes Neurais Recorrentes para este

problema, mais especificamente utilizando unidades LSTM (*Long Short-Term Memory*).

2.2 Estado da Arte

Nossa pesquisa apontou dois tipos de métodos que possuem uma quantidade razoável de literatura os descrevendo: métodos de *ensemble* e SVMs Incrementais. Exploramos as duas opções neste trabalho.

Como representante de algoritmo de ensemble, estudamos o uso de uma família de métodos chamada Learn++, cujo funcionamento é baseado na técnica de *boosting*. Neste caso, isto quer dizer que o algoritmo foca o aprendizado nos pontos que são determinados como sendo mais difíceis de aprender, através da criação de vários classificadores fracos — i.e., que possuem acurácia melhor do que um chute aleatório — que são treinados em diferentes conjuntos de dados. Existem diversas variantes deste algoritmo, destinadas a tratar problemas comuns em cenários de streaming, como variação de conceito, datasets desbalanceados e introdução de novas classes.

No contexto de SVMs Incrementais, nossa pesquisa mostrou a existência de duas técnicas capazes de adaptar incrementalmente a solução de uma SVM. A primeira explora a localidade de certos *kernels*, o que permite que o problema de otimização associado ao classificador seja resolvido por mudanças em um conjunto limitado de variáveis. A segunda técnica usa a noção de incrementos adiabáticos para adicionar um vetor novo ao classificador. Neste caso, tal incremento é definido como um que não viole as condições Karush-Kuhn-Tucker (KKT) de optimalidade.

Um método que ao nosso ver não é explorado na literatura para este problema em específico, é o uso de Redes Neurais Recorrentes. Dedicamos parte do nosso tempo a explorar este domínio, com a ressalva de que devido à falta de literatura, os resultados obtidos deverão ser mais intensamente analisados.

3 DESENVOLVIMENTO

Devido à limitação de tempo para realização do projeto, tendo em vista os objetivos que queríamos atingir, e também pela presença de bibliotecas focadas em aprendizado de máquina, escolhemos Python 3.5 como linguagem principal de implementação. Além disso, usamos R para efetuar algumas das análises sobre os conjuntos de dados e para geração de alguns gráficos.

Dos três eixos principais de exploração – algoritmos de *ensemble*, SVMs e redes neurais recorrents –, nem todos puderam ser completamente explorados quanto ao seu uso para o nosso problema.

O mais problemático dentre eles foi o uso de SVMs Incrementais. A implementação envolve múltiplos cálculos para realizar expansão e compressão de matrizes, juntamente com adaptações a multiplicadores de Lagrange, que causaram variados problemas. O uso do método foi descartado por estas razões.

Exploramos redes neurais recorrentes com células LSTM devido a uma expectativa de que seu funcionamento fosse bom, visto que estas redes tratam sequências de dados naturalmente. Não fomos capazes de encontrar literatura adequada que corrobore o uso dessa técnica para nosso problema especificamente, então os resultados por nós obtidos são majoritariamente exploratórios.

Em contraponto, pudemos explorar em profundidade o uso de algoritmos de *ensemble*, em particular a família Learn++ de algoritmos. Há muitas variações dentro desta mesma família, com o objetivo de remediar diversos dos problemas citados anteriormente que surgem ao tratar fluxos de dados.

4 METODOLOGIA EXPERIMENTAL

Definir experimentos relevantes e válidos pode ser complicado quando lidamos com algoritmos incrmentais. Primeiramente, não há uma equivalência óbvia entre um modelo incremental e um modelo treinado em *batch*. Isso faz com que comparações entre métodos incrementais e em *batch* sejam facilmente tendenciosas.

Para podermos avaliar as diferenças de comportamento entre essas classes de algoritmos, escolhemos basear as comparações em métricas concretas relacionadas a aplicações práticas. Por exemplo, um teste relevante para a escolha ou não de um método é o comportamento da acurácia através do tempo, relacionado ao tempo gasto treinando modelos. Isto é de interesse prático para uma empresa como a SAP, pois proporcionar a clientes métodos que usem menos poder de processamento para atingir resultados que são comparáveis aos atuais possui grande valor.

Outro teste interessante que pode auxiliar a colocar as técnicas exploradas em perspectiva é uma comparação com a atual implementação de árvore de decisão de Hoeffding presente no SAP HANA SDS.

Por último, comparações dentro da família de algoritmos incrementais também são valiosas: podemos comparar as diferentes técnicas entre si, mas também variar os parâmetros de cada técnica individualmente. Um exemplo é ativar ou desativar o uso de um buffer de replay e comparar os resultados obtidos. Estes testes podem ser usados para determinar se uma ideia faz sentido dentro do contexto de algoritmos incrementais, sem a necessidade de estabelecer equivalências com modelos em *batch*.

5 RESULTADOS EXPERIMENTAIS E CONCLUSÕES

De um modo geral, os algoritmos incrementais por nós estudados parecem ser uma escolha viável para implementação no sistema da SAP. O uso de SVMs Incrementais foi problemático, e nossa implementação não pode ser completamente terminada: mesmo que as técnicas estudadas pareçam logicamente corretas, por vezes detalhes de suas implementação não são descritos ou não estão claros.

Em suma, as conclusões e contribuições deste trabalho são:

- Proporcionamos uma visão geral do campo de Aprendizado Incremental, mostrando exemplos e o funcionamento interno de algoritmos, bem comos problemas por eles enfrentados;
- Investigamos os problemas de variação de conceito e desbalanceamento de classes, e como algoritmos se adaptam frente a eles, combinando ideias para combatê-los;
- Adaptamos uma ideia do campo de Aprendizado por Reforço para que ela fosse usada em algoritmos de Aprendizado Incremental;
- Comparamos as técnicas propostas com a que é usada atualmente pela SAP e pudemos ver que nossos objetos de estudo têm performance ao menos tão boa quanto a da técnica já implementada no produto;
- Desenvolvemos uma visualização com o objetivo de comparar algoritmos em batch e incrementais quanto a seus comportamentos e o tempo de treinamento lado a lado, o que pode ser usado para reforçar vantagens do uso de técnicas incrementais.

Ainda existem mais coisas que podem ser exploradas; por exemplo, a expansão destes resultados para mais conjuntos de dados, tanto sintéticos quanto de casos reais, pode nos dar uma compreensão melhor do comportamento destas técnicas, e evidências para apoiar nossas hipóteses. Outro ponto deixado como trabalho futuro é o desenvolvimento fim-a-fim de um caso de uso para as técnicas propostas, juntamente a dados que elicitem as vantagens por elas introduzidas. O objetivo final desta tarefa seria de dar razões concretas para que a SAP implemente técnicas de Aprendizado Incremental em seus produtos, possibilitando que clientes criem seus próprios casos de uso.

Acreditamos que as conclusões tiradas de nosso trabalho não são limitadas ao contexto da SAP. Em particular, técnicas de Aprendizado Incremental podem introduzir vantagens em diferentes contextos, não somente de streaming — e.g para conjuntos de dados que não cabem completamente em memória, usar um algoritmo incremental torna

o problema tratável — que não são explorados neste trabalho.

UNIVERSITÉ Grenoble Alpes

Grenoble INP

Master of Science in Informatics at Grenoble
Master Informatique
Specialization Data Science

Incremental Learning Applied to Streaming Environments

Alister Machado dos Reis

June 19th, 2017

Research project performed at SAP Labs, LLC

Under the supervision of:
Nanda Kaushik, Jérôme Malick

June 2017

Abstract

The need to extract knowledge from data is present in many fields and under many forms. One of the cases where it is needed is when reasoning over streaming data, be it from a sensor feed, a social network, or market data, to name a few examples. SAP, a German multinational headquartered in Walldorf, Germany, with over 300 thousand customers, has a software solution for treating data streams called *SAP HANA Smart Data Streaming*. It currently offers only one possibility of algorithm for performing classification on streaming data. We investigate the potential applicability of alternative algorithms based on the concept of Incremental Learning. This means that the algorithms work by adapting or expanding a single model over time, instead of rebuilding new models from scratch and learning on new data. Based on those alternative algorithms, we can evaluate model performance as well as trade-offs between accuracy and training time. Comparison criteria are proposed and instantiated for the studied examples, providing an overview on how suitable these methods are for the problem at hand. We propose the inclusion of Learn++ in SAP HANA SDS after comparing its performance over study cases, since it can be better than the technique that is currently available in the platform depending on properties of the stream such as the presence or absence of concept drift.

Résumé

La nécessité d'extraire des connaissances à partir de données existe dans plusieurs domaines, et sous des formes variées. Un de ces cas, c'est quand il est nécéssaire de raisonner sur des flux de données, que ce soit des lectures d'un capteur, des informations d'un réseau social ou des tendances de marché, par exemple. SAP a une solution pour analyser des flux de données appelée SAP HANA Smart Data Streaming. Aujourd'hui, cette plate-forme n'a qu'un algorithme pour résoudre des problèmes de classification de données. Ce travail explore l'applicabilité potentielle d'algorithmes alternatifs basés sur le concept de l'Apprentissage Incrémentale. Cela signifie que ces modèles s'adaptent aux nouvelles données, au lieu de créer un nouveau modèle et le faire apprendre l'information la plus courante. En utilisant ces modèles, il est possible d'évaluer la performance ainsi que les *trade-offs* entre précision et temps d'entraînement, par exemple. Des critères de comparaison sont proposés et explorés pour les cas étudiés, ce qui nous possibilite déterminer si ces méthodes sont adéquats pour résoudre le problème.

Contents

Abstract **i**

Résumé **i**

1 Introduction **1**
1.1 Incremental Learning: Overview . . . 2
1.2 State of the Art . . . 2
1.3 Summary of Contributions and Results . . . 3
1.3.1 Experiments and Results . . . 3
1.3.2 Conclusions and Pointers for Future Work . . . 4

2 Problem Statement and Context **5**
2.1 The Company: SAP . . . 5
2.2 SAP Streaming Analytics . . . 5
2.3 Machine Learning on Data Streams . . . 6
No fixed dataset . . . 6
Stream intensity . . . 6
Availability . . . 6
2.4 Particular Challenges . . . 6
Unbalanced classes . . . 7
Unseen classes . . . 7
Concept drift . . . 7
2.5 Naive Approaches . . . 7
2.5.1 Static Batch Learning . . . 7
2.5.2 Iterated Batch Learning . . . 7
2.5.3 Sliding Windows . . . 8
2.6 Incremental Learning . . . 8
2.6.1 Concept Drift . . . 9
2.6.2 Learning on Chunks of Data . . . 10
2.6.3 Memory Usage . . . 10
2.6.4 The Stability-Plasticity Dilemma . . . 10

3 State of the Art **11**
3.1 Ensemble Techniques . . . 11

3.1.1 Learn++ and Boosting 11
3.1.2 Learn++.NSE 12
3.1.3 Learn++.NIE 13
3.1.4 Learn++.SMOTE 15
3.1.5 Learn++.NC 16
3.2 Incremental Support Vector Machines 16
3.2.1 Introduction to Support Vector Machines 16
3.2.2 Incremental SVM: A Locality-Based Approach 18
3.2.3 Incremental and Decremental SVM 18
3.3 Neural Networks 20
3.3.1 Recurrent Networks on Streams 21

4 Implementation and Experiments 23
4.1 Design Choices 23
4.2 Implemented Algorithms 24
4.2.1 Learn++ 24
4.2.2 Incremental and Decremental SVM 24
4.2.3 LSTM Neural Network 25
4.3 Experiments 25
Parameters 26
4.4 Datasets 26
SEA Concepts 26
KDDCup Invasion Detection 27
4.5 Comparison to Batch Models 27
4.6 Comparisons Amongst Incremental Models 28
4.6.1 Replay Memory 28
4.6.2 Window Size 29
4.6.3 Accuracy of Incremental Models 31
4.7 Comparison to SAP HANA SDS 31
4.7.1 Building the Project 31
4.7.2 Importing the Data 32
4.7.3 Results 32
SEA Concepts Dataset 32
KDDCup Dataset 33

5 Conclusions and Future Work 35

A Appendix 37
A.1 Changing the Sigmoid in Learn++.NSE 37

Bibliography 41

— 1 —
Introduction

This work explores the applicability of algorithms that employ incremental learning to classification problems in streaming environments. It has been developed at SAP Labs LLC in San Ramon, California, USA.

SAP has a tool for dealing with streams of data called SAP HANA Smart Data Streaming. It provides the user with various tools, allowing them to read data from different sources such as Hadoop files, CSV files and Web Services, for example. On top of the read data, the user has the possibility of applying SQL-like queries (e.g. joins, filters, projections), and also of feeding data into Machine Learning models.

One of the problems that fall under the Machine Learning umbrella – and the one we will focus on in this thesis – is that of performing *classification* (also called *supervised learning*) over data. Such problems arise in a variety of scenarios. Digit recognition, speech recognition and machine translation are all examples of classification problems. Some companies now use classification algorithms to increase or improve their capabilities. These range from recommender systems in the case of Netflix, to fraud detection in banks or the development of digital assistants, to name a few instances.

Unfortunately, the most well-known algorithms for classification assume that training data – in this case composed by many pairs, for which the first element is a vector of variables and the second is the *class* or *label* information – are stored in a fixed, predefined dataset. In many other applications, our system might be receiving data incrementally instead (or our dataset might be too big to fit in memory). This is the case for when we perform learning over sensor feeds or the stock market, to name a few examples. An immediate consequence is that the algorithm performing predictions on this data should then adapt on-the-fly upon seeing new information, which is in conflict with the usual assumption of a static dataset. This motivates the need for Incremental Learning, a category of techniques that are capable of doing just that: learning over time and keeping up with the data.

While there are several techniques to attack this problem, different algorithms might better suit different use cases. Therefore, it is helpful to have a suite of methods from which we can pick the one that is best suited to each problem at hand.

Currently, SAP's product offers only two options of algorithms that can be applied on streams of data: a decision tree for classification, and a clustering algorithm. The goal of this work is, then, to explore, implement, and evaluate different options of new classification algorithms to be integrated in the platform. To reach this goal, we will conduct experimental analyses to evaluate the performance of these new methods in different datasets, both synthetic and not. Ultimately, we aim not only to build a broad and generic framework to deal with

streams of data, but also to obtain guidelines that suggest which of the studied methods is applicable with more potential for success with each type of data that might be analyzed.

1.1 Incremental Learning: Overview

In this section, we give a high-level description of Incremental Learning, as well as some basic notions needed to understand the setting of the problem we are handling. This is further discussed in Chapter 2.

Incremental Learning [17], while appearing under different definitions in the literature, can be succinctly defined by its main points: in this setting, we learn by adapting or expanding a *model*. A model is essentially a parameterized function that maps features to classes, and its parameters can be tuned so that, for a given dataset of pairs of features and classes, the model is capable of correctly performing this mapping. Tuning these parameters is the task of a *learning algorithm*. Another assumption of incremental learning algorithms is that training data cannot be stored for later use; a last defining point is that the learned model should be similar to one learned using a batch method.

Such an approach sounds fitting to a streaming environment, which is why we chose to explore it in the first place. In streaming, we do not have a fixed dataset, since data is generated over time. This complements the assumption that we cannot store data, and also makes adapting one existing model instead of re-training a single one from scratch a valuable alternative. And since incremental corrections to a model are expected to be less computationally intensive than training a new one from scratch, it is possible to perform classification whenever it is needed and based on whatever data the system had access to, up to that point in time.

However, dealing with data streams introduces some problems. The ones that stand out the most are related to concept drift and class imbalance. Concept drift invalidates the fundamental assumption of many models used in classic Machine Learning scenarios: the i.i.d. assumption, where we assume data points are sampled independently and from a fixed, yet unknown distribution. Class imbalance creates a problem for the process of learning a classifier since we can have highly biased predictors – e.g., ones that are very accurate when predicting data points associated with one class, for which we have many examples, but not so much for rarer classes – if we do not account for it.

In this next section we evaluate the state of the art in streaming algorithms and discuss the learning techniques used by each so algorithm, as well as the measures they take to mitigate the problems mentioned above . Since the literature on Incremental Learning is diverse, we choose to explore what seemed to us like the most prominent ensemble technique (the one that had reasonable presence in the literature), as well as approaches based on Incremental Support Vector Machines. We also evaluated the use of Recurrent Neural Networks in this setting. The experiments and obtained results are discussed in Chapter 4.

1.2 State of the Art

This section is dedicated to summarizing the main state-of-the-art techniques identified during our literature review. Existing methods will be further discussed in Chapter 3.

Our literature research brought up mainly two types of methods that have a reasonable amount of scientific literature to support them: ensemble methods and Incremental SVMs. We explore both of these domains in this thesis.

The ensemble technique we investigated as part of our state-of-the-art review is called Learn++ [24], and it is based on a technique called *boosting*, using multiple weak classifiers to provide strong predictive power through voting schemes. *Boosting* is a technique based on the idea of learning to predict and correct a classifier's errors. The mentioned algorithm performs boosting by focusing learning on the examples deemed most difficult to classify. This is done through the training of many *weak learners*, which can be any model that achieves better accuracy than that of a random guess. During the years, many variants of this algorithm were developed, with the goal of dealing with concept drift, imbalanced datasets, and unseen classes. We explored these variants, combined some of their ideas and even borrowed others from the Reinforcement Learning literature.

As for Incremental SVMs, two approaches were explored. The first one explores the locality of certain kernel functions, solving the SVM dual problem again while optimizing only a limited number of variables. The second one uses the notion of adiabatic increments to include a new vector in the solution. An adiabatic increment is defined as one that keeps the Karush-Kuhn-Tucker conditions valid while introducing a new example from the stream to the current SVM.

One approach that, to the best of our knowledge, is not explored in the literature, is the use of Recurrent Neural Networks to address stream classification problems. This is also explored in this work, but the results need to be thoroughly analyzed since there is no theoretical support for their application to this specific problem.

1.3 Summary of Contributions and Results

In the following sub-sections we summarize the main experimental results and conclusions achieved by this thesis; these will be further detailed in Chapters 4 and 5, respectively.

1.3.1 Experiments and Results

Defining relevant and informative experiments can be a tricky task when dealing with incremental algorithm. First and foremost, there is not a clear, obvious equivalence between an incremental model and a batch model[1].This makes it harder to interpret experimental results when directly comparing incremental and batch models.

In this work we address this difficulty by basing our comparisons on concrete measures that have practical applications. For example, a relevant test is to compare the behavior of the algorithm's accuracy over time while also keeping track of the time spent training them. This is of practical interest for a company such as SAP, where providing clients with a method that uses less processing power to achieve results that are at least comparable to the current approach is a valuable asset.

Another interesting test and comparison that helps identifying promising learning techniques is to compare the implemented algorithms to the currently available decision tree implementation that is a part of SAP HANA SDS.

[1]perhaps one based on VC-dimension would be suitable, but this needs further investigation

Finally, since each evaluated algorithm has their own parameter settings, it is interesting to compare them with each other, and with themselves while varying their parameter settings; for example, by tuning a replay memory parameter (which will be discussed in more details later) and comparing the resulting model accuracies. These tests can be used to detect if an implemented idea makes sense within the incremental learning context, without the need to establish an equivalence with a batch model.

1.3.2 Conclusions and Pointers for Future Work

Overall, incremental algorithms seem to be a viable choice to be implemented inside within SAP's system so far. The use of Incremental SVMs was problematic, and our implementation could not be properly finished. While the explored techniques seem sound, details of their implementation are at times not well described, or left unexplained in the texts that describe them.

To summarize the conclusions and contributions of this work:

- An overview on the Incremental Learning field was provided, showing examples and inner workings of algorithms and the problems they face;
- Investigating concept drift and class imbalance and the ways algorithms adapt to them, and combining ideas to tackle those problems;
- Adapting an idea from Reinforcement Learning to be used in Incremental Learning;
- Comparing the proposed approaches to the one currently used by SAP and making sure that they perform at least on par with it;
- Developing a visualization to compare batch and incremental behavior and training time side by side, which can be used to make a concrete case for the use of incremental techniques.

There are still things to explore; for example, expanding these results to more datasets, both synthetic and real, in order to give us a better grasp on our insights and evidence regarding our hypotheses. Another point left as future work is developing a full end-to-end use case of the proposed techniques, together with data that supports the advantages they introduce. The final goal of this task would be to give reason for SAP to implement these Incremental Techniques in their product and let clients come up with their own tailored use cases.

We believe the conclusions drawn from our work may be extrapolated to contexts other than SAP's. In particular, Incremental Learning techniques can introduce gains in different streaming and non-streaming contexts – e.g, for datasets that do not fit in memory, an incremental approach enables learning – that are not covered here.

— 2 —
Problem Statement and Context

In this thesis, we explore the application of Incremental Learning algorithms to streaming environments. This is of particular importance to SAP, the company where this work was developed, in the context of their streaming analytics product.

2.1 The Company: SAP

SAP SE is a multinational software corporation that makes enterprise software to manage business operations and customer relations. SAP is headquartered in Walldorf, Baden-Württemberg, Germany, with regional offices in 130 countries. The company has over 293,500 customers in 190 countries. It is also a component of the Euro Stoxx 50 stock market index. In 2014, SAP had over 75,000 employees, a yearly revenue of around € 20 billion and € 3 billion profit. This work was developed at the SAP Labs LLC in San Ramon, California, United States of America.

2.2 SAP Streaming Analytics

One of SAP's products is SAP HANA Smart Data Streaming (SAP HANA SDS), which is part of the SAP HANA platform. This product focuses on the processing of data streams that may come from multiple and heterogeneous sources. Examples of those include but are not limited to sensor data, external databases and data feeds*.

The above-mentioned product offers the capability of performing SQL-like queries on data from the input streams. This means we can join streams, filter and aggregate data, for example. More than that, the possibility of using Machine Learning algorithms that are streaming-specific is also available. Currently, the product implements two of them, being one for classification problems and the other for data clustering.

For *classification*, SAP HANA SDS uses a Hoeffding decision tree, an algorithm described in [9] and [18]. For clustering, it uses the DenStream algorithm, described in [2].

*https://help.sap.com/viewer/52acc1f6b1d7428caab280d193c820f6/2.0.01/en-US/e7a0423e6f0f10148f58c70473b6787b.html

2.3 Machine Learning on Data Streams

In this work, we focus on classification problems and explore the different options that exist to deal with them. This task can be described as trying to estimate a function $h : \mathcal{X} \to \mathcal{Y}$ where $\mathcal{X}$ is an input space, typically $\mathbb{R}^n$ and $\mathcal{Y}$ is a label space. In the case of binary classification, $\mathcal{Y} = \{-1,+1\}$ or $\{0,1\}$; in the multiclass case, $\mathcal{Y} = \{1,\ldots,K\}$. This function h should minimize some error criterion with respect to a dataset of examples for which we know the desired output. One possible criterion to be minimized is the 0-1 Error, defined as:

$$\mathcal{L}_{0-1}(h,D) = \frac{1}{m}\sum_{i=1}^{m} [\![y_i \neq h(\mathbf{x}_i)]\!]; \text{ where } D = \{(\mathbf{x}_i, y_i) | i \in \{1,\ldots,m\}\}$$

where $[\![\cdot]\!]$ is the indicator function, which takes the value 1 if the condition inside it is true, and 0 otherwise.

Since this error measure is not convex, we do not directly minimize it but typically work with an upper bound for it. Different algorithms employ different error measures: for example, Support Vector Machines originally use the Hinge Loss, and Neural Networks can use the Cross-Entropy Loss.

The fact that we are dealing with *streams* modifies the classical Machine Learning setting in some ways, since it implies that the dataset D is not fixed, but in fact built incrementally over time. These impact the set of solutions that can be soundly applied to the problem. Some of the main differences between the classical Machine Learning setting – here called the "static batch" setting – and the streaming setting are listed below.

No fixed dataset When dealing with streaming, the notion of a stable, fixed dataset over which we run machine learning algorithms does not exist. New data is assumed to be arriving constantly. Therefore, we cannot use multiple pass algorithms, for instance, since the dataset is always evolving. A way to theoretically represent this is to say that we are dealing with datasets of *infinite* size. To deal with this, at each point in time we will look at only a *chunk* or a *window* of the whole dataset.

Stream intensity Data is assumed to arrive at an unknown rate. This poses a problem in case this rate is too high, because storing data in that setting can be infeasible, or not practical. Even if we could store all the arriving data, an algorithm that would use all of it would probably take a long time to run, and as an example lead us to an obsolete classifier.

Availability In streaming environments, we want to be able to classify data with the most up-to-date information at all times. This means classifiers should be always ready to predict classes of data points.

2.4 Particular Challenges

Beyond the changes to the machine learning setting itself when we move to streaming environments, some new issues arise. These are related to changes in the data generating process and to the fact that we do not have a fixed dataset.

Unbalanced classes Inside a portion of the dataset, there is no guarantee that we'll have a uniform distribution over classes: examples from one class might come in an arbitrarily high proportion. This is potentially problematic because in cases where, for example, one class accounts for 99% of the data points, learning a classifier that defaults to the majority class will give us 99% accuracy while having no generalization power.

Unseen classes There is no guarantee that we will have examples from all possible classes in a given time window. There is no bound on the first time that we will have seen all possible classes either. This means we can spend an unknown and unbounded amount of time seeing examples of only the positive class when doing binary classification, for instance. Whenever a new class appears in the dataset, or classifier needs to recognize it as new and avoid erroneously classifying it as one of the classes it had seen so far.

Concept drift The data generating process will most probably change over time when we're mining a data stream. This is called *concept drift* in the Incremental Learning literature [17]. The main consequence of dealing with datasets that have drifting concepts is that one of the very first assumptions we have when using Machine Learning algorithms is not valid: we cannot say our data is independent and identically distributed (i.i.d. for short). Concept drift will be more thoroughly discussed in Section 2.6.1.

2.5 Naive Approaches

There are simples alternatives to using Incremental Learning when dealing with streaming data. Naively, we can think of applying batch algorithms several times along the stream, for example. While possible, this entails both technical and theoretical problems to learning. Three possible naive approaches are discussed below.

2.5.1 Static Batch Learning

We have the option of training one classifier using data that we might have in storage for some reason, and using that same classifier on the stream. Suppose we kept some data points from the past of the stream in a database, then trained any classifier using them. Some of the problems with this approach are:

- The classifier will become obsolete if the data generating process changes, since we do not train it again and just use it for prediction;
- If the data we have are from a slice of the stream where the data generating process changed, the i.i.d. assumption does not hold and we have no classic theoretical guarantees on the generalization error of the classifier, for example.

2.5.2 Iterated Batch Learning

We call *iterated batch learning* the process that repeatedly trains a batch model over chunks of data either periodically or whenever its accuracy falls below a given threshold. While straightforward to implement and intuitively better than static batch learning, this approach is not without its flaws:

- It might be computationally intensive to train a model several times over different data, and this can be aggravated when the stream has a high input rate.
- When retraining periodically, we might either over- or underestimate the rate with which data changes. If we overestimate it, we will be potentially wasting computational resources; conversely, when we underestimate it, the classifier's performance might be unacceptably low for a long time before we train a new one with the most current data.
- Throwing away old classifiers leads to what is called *catastrophic forgetting*: we forget everything we knew and keep only the knowledge that can be extracted from the current portion of the data. This is myopic in the sense that previous knowledge can still be useful – or might become useful again in the future. Ideally, we would have previous data help us interpret present data.

2.5.3 Sliding Windows

Another alternative to mitigate the catastrophic forgetting problem would be to use sliding windows on the stream, discarding the oldest data point when the window fills up. In this setting, we train a model when the window fills up for the first time and whenever a new example arrives. Some problems with this approach are:

- It is undeniably computationally intensive since we are training a new model whenever new data points arrive after the first time we fill the window.
- Determining the window size is not as simple as it seems, and should be related to the rate with which data changes. In moments where the concept is changing, we should use a small window to make sure we quickly adapt to the new concept. Conversely, when the concept is stable, the window can grow to consolidate the learned concept.

2.6 Incremental Learning

As an alternative to these naive approaches, in this work we investigate the use of Incremental Learning algorithms. The main idea behind these is avoiding the retraining of classifiers by expanding or adapting a single classifier. The definition of Incremental Learning varies from author to author, but the most frequent and important aspects are:

- We do not keep a stored version of the data. This makes multiple passes algorithms such as naive k-means clustering not directly applicable.
- Each example is only fed to the learning algorithm *once*. We assume that once we see an example and use it to improve our classifier, it is gone and we no longer have access to it.
- Ideally, previous knowledge from the classifier should help us in dealing with new data. We should not learn everything from scratch every time.
- Our classifier should adapt to changes in data distribution – concept drift. There are many ways in which concept drift can present itself, and our algorithms should be aware of that.

- The learned model should be similar to the one learned using a batch algorithm.

The aforementioned aspects of Incremental Learning are seen as a good fit for performing Machine Learning over streaming data, since our classifier keeps up with the data, modifying itself to stay up-to-date throughout the classification process.

It is worth noting that nothing advises against the use of incremental techniques in settings. In fact, we will talk about algorithms whose characteristics make their applicability limited to certain stream types, while matching perfectly classical Machine Learning settings. The main improvement introduced by these techniques would then be saving processing power and perhaps improving the ease of interpretation of the learned model.

When dealing with streams, here formalized as infinite datasets, we can choose different strategies to handle them: using sliding windows, chunking or even treating a single example at a time – which is called *online learning*.

Incremental Learning does not solve all of our problems without introducing some of its own, but they tend to be manageable. Some of the most notable issues with learning incrementally are listed below.

2.6.1 Concept Drift

In a typical machine learning setting, we have a dataset $D = \{(\mathbf{x}_i, y_i) \,|\, i \in \{1, \ldots, m\}\}$, where $\mathbf{x}_i \in \mathbb{R}^n$ are the vectors that represent the examples, and $y_i \in \{-1, +1\}$ is the example's label ($y_i \in \{1, \ldots, K\}$ for multiclass problems). This dataset is assumed to be composed of independent draws from a fixed, yet unknown distribution: $(\mathbf{x}_i, y_i) \sim P(x, y), \forall i \in \{1, \ldots, m\}$. This assumption does *not* generally hold for streaming environments.

Take, for example, the case of weather data. It is simple to see that the distribution of temperatures, rain and even the occurrence of extreme weather conditions such as hurricanes is correlated to the *seasons* of the year. A direct implication of such fact is that the probability of seeing examples for certain classes will change over time. This makes our data not identically distributed. Data points are also not independent: the probability of tomorrow being a sunny day is different if today was a sunny day, for example.

We can classify concept drift according to some of its properties. For example, concepts may be recurring or not. Concepts may also drift abruptly or gradually. There is also a distinction between virtual and real concept drift.

Since we're trying to predict an example's class from its attributes, we can use Bayes' rule to analyze this problem from a probabilistic point of view:

$$P(Y_i = k \,|\, \mathbf{X}_i = \mathbf{x}) = \frac{P(\mathbf{X}_i = \mathbf{x} \,|\, Y_i = k) P(Y_i = k)}{P(\mathbf{X}_i = \mathbf{x})}$$

Since $P(X_i = x)$ is constant across all classes, we may experience concept drift in any of the three major variables that compose Bayes' theorem [19].

When the class priors $P(Y_i = k)$ or the distributions of the classes $P(\mathbf{X}_i = \mathbf{x} \,|\, Y_i = k)$ change, we say we experience *virtual* drift. The only type of drift that matters, however, is *real* drift. This one latter occurs when we have changes in the posterior distributions of class membership $P(Y_i = k \,|\, \mathbf{X}_i = \mathbf{x})$. This means that the shape of the boundaries between the classes has changed.

2.6.2 Learning on Chunks of Data

The fact that we do not have a fixed dataset over which we train our algorithm creates a few issues that we need to be aware of. For example, as cited before, we do not have any guarantee that we will have examples from every possible class inside a given chunk of our stream. Still, we need to make sure that our classifier generalizes well once data from new classes start coming in.

Another issue is that we learn using a limited amount of data every time, and the most recent data chunk might not reflect the behavior of the whole stream – or even of the current concept of the stream.

2.6.3 Memory Usage

This is a crucial constraint on Incremental Learning: we need our algorithms to have a constant memory use, or one that is bounded by a constant. In big-O notation, we need our procedures to have $O(1)$ space complexity.

This is because the stream is considered to be infinite, so if the memory use grows with the number of examples seen, we will deplete our memory at a given point in time.

2.6.4 The Stability-Plasticity Dilemma

When learning incrementally, we need to be wary of how much we forget about the past in favor of adapting to the present. This problem is called the *stability-plasticity dilemma* [15]. A classifier that is completely stable will not change according to the current concept, possibly becoming obsolete at a certain point in the lifetime of the stream. Conversely, a completely plastic classifier will choose to forget what it knew previously and adapt to the most recent data only. This is dangerous behavior, since as cited before, learning on chunks of data gives us a narrow view on what the trend of the stream is as a whole.

— 3 —
State of the Art

The most current literature on Incremental Learning presents mainly two types of methods: ensemble techniques and purely incremental approaches. The most prominent algorithm that uses an ensemble for classification was found to be Learn++ [24]. Among the purely incremental approaches, we learned of the existence of techniques that learn Support Vector Machines in an incremental fashion. Also, algorithms that handle sequences in general might be of interest; as a representative of that class of methods, our search brought up the Long Short-Term Memory (LSTM) Recurrent Neural Networks.

In the following sections, we discuss these techniques and their variations. We also introduce some details related to their implementation. This is done mostly because there are some expansion points in the techniques, and we chose to explore them too. The discussion is done in parallel with brief evaluations on the techniques' advantages and disadvantages, with the goal of selecting algorithms that will be implemented (and evaluated in Chapter 4).

3.1 Ensemble Techniques

Ensemble techniques do not use a single classifier to predict the class of an unseen example. Instead, they build a *set* of classifiers that works as a whole to determine the label of the example at hand. This is typically done through a majority voting scheme, where the class that receives the most votes is the one finally predicted. The idea of using many classifiers to provide a better prediction power remounts to the theory of Boosting. In fact, the algorithm we explore in this section is strongly inspired by AdaBoost [12, 27].

3.1.1 Learn++ and Boosting

Learn++ [24] works with the concept of combining many *weak classifiers* to provide a strong classification. This was notably used in the AdaBoost algorithm, which was applied to a batch setting.AdaBoost maintains a distribution over the training samples that focuses on examples that are deemed hard to classify. These are then fed into a weak learning algorithm that needs to have an accuracy better than 0.5 with respect to the current distribution of the training samples.

The idea of improving classification by "fitting your errors" is the base of the Boosting theory. In AdaBoost, the intuition is that, since a classifier will most probably receive the hard examples when it is trained, it will learn these examples and improve the ensemble's overall performance. The pseudocode for this algorithm is shown in Algorithm 1. The built classifier

Algorithm 1 The AdaBoost Algorithm

Require: Dataset $D = \{(\mathbf{x}_i, y_i) \,|\, i \in \{1, \ldots, m\}\}$
Require: Weak learning algorithm **WeakLearner**
Require: Integer T defining the number of iterations

```
 1: function AdaBoost(D, WeakLearner, T)
 2:     P_1(i) = 1/m   ∀i ∈ {1,...,m}
 3:     for t = 1,...,T do
 4:         repeat
 5:             D_t ← sample from D according to P_t
 6:             h_t(x) ← WeakLearner(D_t)  ▷ h_t : ℝ^n → {−1,+1} is the current hypothesis
 7:             ε_t ← Σ_{i=1}^{m} P_t(i)⟦h_t(x_i) ≠ y_i⟧
 8:         until ε_t < 0.5
 9:         β_t ← 0.5 ln((1−ε_t)/ε_t)
10:         P_{t+1}(i) ← P_t(i) exp(−β_t y_i h_t(x_i))   ∀i ∈ {1,...,m}
11:         P_{t+1}(i) ← P_{t+1}(i) / Σ_{i'=1}^{m} P_{t+1}(i')   ∀i ∈ {1,...,m}   ▷ normalize P_{t+1}
12:     end for
13:     return The final hypothesis H(x) = sign(Σ_{i=1}^{T} β_t h_t(x))
14: end function
```

can be interpreted as a majority voting scheme, where every sub-classifier created votes for the class it predicts – the output of $h_t(\mathbf{x})$ – with weight inversely proportional to its error on the data ε_t.

Learn++ borrows from this idea and adapts it to deal with streaming data. Just like AdaBoost, it uses an ensemble of weak classifiers to provide a strong classification performance. One major difference is that we train along the stream: every new data chunk that arrives prompts us to create a given number of new classifiers. These are then combined for prediction using a rule similar to the one used in AdaBoost.

The pseudocode for Learn++ in its basic version is given in Algorithm 2. For clarity, in this algorithm we suppose we have an infinite source of data D_∞, the stream, and feed chunks from this stream into our learning algorithm. Every chunk is supposed to have size m for brevity, but in practice nothing constrains us to using equally sized chunks. The variable k indexes over the many chunks of data our algorithm will see during its execution.

This version of the algorithm does not account for many of the problems that can arise when learning from streaming data. For example, there is no resilience to concept drift, neither there is a mechanism to handle unbalanced classes. These problems are tackled by variants of the Learn++ procedure, some of which we describe in the following sections. We cite this basic version of the algorithm because it is the base from which all the others derive, as well as because it can be applied to non-streaming data.

3.1.2 Learn++.NSE

Learn++.NSE [11] (**N**on-**S**tationary **E**nvironments) was developed to handle the problem of concept drift in stream data. It does this by decaying the influence of past classifiers over time, while also re-evaluating them in the most recent data chunk. This is due to the fact that an old concept might become relevant again in the future – in the case it is a recurrent concept – and then we can give a higher weight to an old classifier if it performs well in a new data chunk.

Algorithm 2 The Learn++ Algorithm

Require: Data stream $D_\infty = \{(\mathbf{x}_i, y_i) \mid i \in \{1, 2, \dots\}\}$ presented as *chunks* D_k
Require: Weak learning algorithm **WeakLearner**
Require: Integer T_k defining the number of classifiers to build per data chunk
1: **for** chunk D_k of data from D_∞ **do**
2: LEARNPP(D_k, **WeakLearner**, T_k)
3: **end for**
4: **function** LEARNPP(D_k, WeakLearner, T_k)
5: $w_1(i) \leftarrow 1/m \quad \forall i \in \{1, \dots, m\}$
6: **for** $t = 1, \dots, T_k$ **do**
7: $P_t(i) \leftarrow w_t(i) / \sum_j w_t(j)$
8: **repeat**
9: $S_t \leftarrow$ sample from D_k using P_t
10: $h_t^k(\mathbf{x}) \leftarrow$ WEAKLEARNER(S_t)
11: $\varepsilon_t^k \leftarrow \sum_i P_t(i) [\![h_t^k(\mathbf{x}_i) \neq y_i]\!]$
12: **until** $\varepsilon_t^k < 0.5$
13: $\beta_t^k \leftarrow \varepsilon_t^k / (1 - \varepsilon_t^k)$
14: $H_t^k(\mathbf{x}) \leftarrow \arg\max_{y \in \mathcal{Y}} \sum_{s=1}^t [\![h_s^k(\mathbf{x}) = y]\!] \log\left(1/\beta_t^k\right)$
15: $E_t^k \leftarrow \sum_i P_t(i) [\![H_t^k(\mathbf{x}_i) \neq y_i]\!]$
16: **if** $E_t^k \geq 0.5$ **then**
17: discard H_t^k, set $t \leftarrow t - 1$, go back to line 8
18: **end if**
19: $B_t^k \leftarrow E_t^k / (1 - E_t^k)$
20: $w_{t+1}(i) \leftarrow w_t(i) \times (B_t^k)^{[\![H_t^k(\mathbf{x}_i) = y_i]\!]} \quad \forall i \in \{1, \dots, m\}$
21: **end for**
22: **return** The final hypothesis $H(\mathbf{x}) = \arg\max_{y \in \mathcal{Y}} \sum_k \sum_{t : H_t^k(x) = y} \log\left(1/B_t^k\right)$
23: **end function**

This algorithm, in its definition, builds only one new classifier per data chunk received. We then re-evaluate every classifier created in step k again at the current time t, as well as the most recently created classifier. The errors of each classifier are then averaged using weights given by a sigmoid, so that the most recent errors are more important to determine the strength of a classifier's vote. The requirements for this algorithm are the same as for Algorithm 2, and the main function is replaced with the contents of Algorithm 3.

3.1.3 Learn++.NIE

When dealing with streams, one of the problems that can cause an incremental classifier to perform poorly is when classes are imbalanced.Learning in such environments is complicated in the sense that a classifier that defaults its predictions to the majority class can have a high accuracy rate without learning how to generalize. This also means that such a classifier would have a high voting weight when used inside Learn++, even though it always predicts the same class.

As a way to mitigate this, Learn++.NIE (for **N**on-stationary and **I**mbalanced **E**nvironments) uses a different error measure when calculating the classifiers' ε_t^k. Instead of using the accuracy

Algorithm 3 The Learn++.NSE[†] Algorithm

Require: Sigmoid slope a, inflection point b
1: **function** LEARN++NSE(D_t, WeakLearner)
2: **if** $t = 1$ **then**
3: $P_1(i) = w_1(i) = 1/m \quad \forall i \in \{1, \ldots, m\}$
4: **else**
5: $E_t \leftarrow \sum_{i=1}^{m} (1/m) [\![H_{t-1}(x_i) \neq y_i]\!]$ ▷ H_{t-1} is the current composite hypothesis
6: $w_t(i) \leftarrow (1/m) E_t^{[\![H_{t-1}(x_i) = y_i]\!]}$; $P_t(i) \leftarrow w_t(i) / \sum_j w_t(j)$
7: **end if**
8: **repeat**
9: $h_t(\mathbf{x}) \leftarrow$ WEAKLEARNER(D_t)
10: $\varepsilon_t^t = \sum_{i=1}^{m} P_t(i) [\![h_t(\mathbf{x}_i) \neq y_i]\!]$
11: **until** $\varepsilon_t^t < 0.5$
12: $\varepsilon_k^t \leftarrow \min\{\sum_{i=1}^{m} P_t(i) [\![h_k(\mathbf{x}_i) \neq y_i]\!], 0.5\} \quad \forall k \in \{1, \ldots, t-1\}$
13: $\beta_k^t \leftarrow \varepsilon_k^t / (1 - \varepsilon_k^t)$
14: $\omega_k^t \leftarrow 1/(1 + e^{-a(t-k-b)})$; $\omega_k^t \leftarrow \omega_k^t / \sum_{j=k}^{t} \omega_k^j \quad \forall k \in \{1, \ldots, t\}$
15: $\bar{\beta}_k \leftarrow \sum_{j=k}^{t} \omega_k^j \beta_k^j \quad \forall k \in \{1, \ldots, t\}$
16: **return** The final hypothesis $H_t(\mathbf{x}) = \arg\max_{y \in \mathcal{Y}} \sum_{k=1}^{t} \log\left(1/\bar{\beta}_k\right) [\![h_k(\mathbf{x}) = y]\!]$
17: **end function**

weighted by the current probability distribution associated to the data, the following rule is used (for binary classification problems):

$$\varepsilon_t^k = \eta(1 - r_{k,+}^t) + (1 - \eta)(1 - r_{k,-}^t) \tag{3.1}$$

where $r_{k,+}^t$ and $r_{k,-}^t$ are the measures of *recall* from the minority and the majority classes, respectively. The η parameter determines how much we should prioritize being able to correctly classify the minority examples over the majority.

This algorithm still uses a sigmoid to decay the importance given to older classifiers, just as in Learn++.NSE. The parameter η needs to receive special attention since performing better on the minority class can make classification on the majority class become worse. There is a trade-off between being able to successfully handle the minority without lowering the overall accuracy of the ensemble too much.

In the next chapters, we will describe our experiments. When the actual code was written, we chose to leave the particular error function to be used by the algorithm as a free parameter. This was done to allow for some flexibility in our exploration of error functions that might better suit the problems we were facing. Before we were aware of Learn++.NIE, for instance, the problem of learning under class imbalance was attacked by using Matthew's correlation coefficient as an error measure . For the binary classification case, the MCC is given by

$$\text{MCC} = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}$$

[†]In practice, we implemented a different version of the sigmoid function. In our code, we use $\omega_k^t \leftarrow 1/(1 + \exp(-a(k - t + b)))$ (see Appendix A.1). The original sigmoid function was found to overvalue the vote of the most recent classifier, which led to myopic behavior. This has been reported to the original authors.

where TP and TN are the number of true positives and true negatives, and FP and FN are the false positives and false negatives, respectively. We use $(1 - \text{MCC})/2$ to have an error measure between 0 and 1. The intuition is that, if any of the elements in the denominator of this measure is zero, the MCC can be set to zero. Therefore, if we were defaulting to the positive class, for instance, the element TN + FN would be zero, which indicates a classifier that does not generalize. Our measure would then output 0.5, which would not pass the inner loop's test. This did not show promising results , so we chose to stick with the measure presented in Learn++.NIE's paper.

3.1.4 Learn++.SMOTE

Another option to deal with imbalanced classes in the stream is using a *supersampling*, or *over-sampling* technique. The idea behind that is to artificially create examples from the minority class in order to have more balanced proportions in the dataset.

One of the algorithms for doing that is called SMOTE [4], which stands for Synthetic Minority Over-sampling Technique. This algorithm creates new artificial examples that lie on the line connecting two existent examples from the minority class.

For every example that we want to use to expand our training set, its k nearest neighbors are determined and used for the creation of a new, synthetic example, that is added to our dataset and used for training. A simplified pseudocode for the SMOTE procedure is shown in Algorithm 4.

Algorithm 4 The SMOTE over-sampling technique

Require: i: the index of the example to create artificial samples from
Require: k: the number of nearest neighbors to use
Require: D: the dataset of vectors with dimension d
1: **function** SMOTE(i, k, D)
2: $\quad \mathbf{x} \leftarrow D[i]$
3: $\quad \mathcal{N} \leftarrow$ the k nearest neighbors of $\mathbf{x}$ in D
4: $\quad \mathbf{x}' \leftarrow$ randomly select a member of $\mathcal{N}$
5: $\quad [\delta]_i \leftarrow \mathcal{U}(0,1) \quad \forall i \in \{1, \ldots, d\}$ ▷ δ is a vector with uniform random values
6: $\quad$ **return** $\mathbf{x} + \delta \odot (\mathbf{x}' - \mathbf{x})$ ▷ $\odot$ is the element-wise product
7: **end function**

In Learn++.SMOTE [8], we use this over-sampling technique to extend our training set with more examples from the minority class. While this can handle well some imbalanced class cases, the more extreme ones still require other solutions. When there are only very few examples from the minority class (less than 5, for instance), trying to make the dataset more balanced means we have to create many examples from very little “real information”. This can negatively impact performance.

On top of that, if we have only one example from the minority class in a chunk, it is impossible to apply this technique. We should then tune our error measure towards something that penalizes errors in the minority class severely to avoid creating a classifier that defaults to the majority class.

3.1.5 Learn++.NC

In a data stream, as mentioned in Section 2.6, it might take us an unknown and unbounded time to see examples from all classes. We also have no guarantee that examples from every class will be present in every chunk of data from our stream. When using algorithms that employ voting mechanisms, this might lead to the *out-voting* problem.

This happens when classifiers that performed too well on the datasets they were trained on — and thus have high voting weights — have never seen the correct class, and will be inevitably mistaken. All of these will vote for wrong classes, overruling the vote of the classifier that has been trained on the new class, and is therefore more capable of determining examples that belong to it.

To address this problem, the algorithm Learn++.NC [23] was proposed. In this variant, a different voting scheme is used. It is called Dynamically Weighted Consult and Vote. In this scheme, classifiers cross-reference their predicted classes with the classes they were trained on. This allows them to adjust their voting weights if, for example, they have not been trained on a class that was predicted by another classifier. The snippet that performs DW-CAV is shown in Algorithm 5.

Algorithm 5 The DW-CAV weight correction technique

Require: The regular voting weights $W_k^t = \log\left(1/\beta_k^t\right), \quad \forall k \forall t$
Require: The sets of classes on which each classifier was trained $C_k^t, \quad \forall k \forall t$
  function DW-CAV
    $Z_c \leftarrow \sum_k \sum_{t:c \in C_k^t} W_k^t \quad \forall c \in \mathcal{Y}$
    $P_c(i) \leftarrow \sum_k \sum_{t:h_k^t(\mathbf{x}_i)=c} W_k^t / Z_c \quad \forall c \in \mathcal{Y} \quad \forall i \in \{1, \dots, m\}$
    if $P_a(i) = P_b(i) \wedge a \neq b \wedge \{h_t^k | \{a,b\} \subseteq C_k^t\} = \emptyset$ **then**
      ▷ degenerate case: two classes with max confidence, no classifier trained on both
      $P_a(i) \leftarrow P_b(i) \leftarrow 0$
    end if
    $W_k^t(i) \leftarrow W_k^t \prod_{c:c \notin C_k^t} (1 - P_c(i))$
    return The final hypothesis $H_k^t(\mathbf{x}_i) = \arg\max_{c \in \mathcal{Y}} \sum_k \sum_{t:h_t^k(\mathbf{x}_i)=c} W_k^t(i)$
  end function

3.2 Incremental Support Vector Machines

Instead of building a set of classifiers, we can also investigate the possibility of updating a single model in an incremental fashion. Thus, we would incorporate new training samples in this model, refining its predictive power.

A type of model that can be updated incrementally is the Support Vector Machine. There exist several approaches to do this, using different strategies to update the model upon the arrival of a new training example.

3.2.1 Introduction to Support Vector Machines

The Support Vector Machine [5] is a model that classifies data by creating a hyperplane with maximum margin (smallest distance between a training sample and the separating hyperplane).

This means that the form of the classifying function is $f(\mathbf{x}) = \text{sign}(\mathbf{w}^T\mathbf{x} + b)$, for a weight vector $\mathbf{w}$ and an offset b. Since it can be shown that the margin is inversely proportional to $\|\mathbf{w}\|_2^2$, this problem is usually formulated as:

$$\text{minimize}_{\mathbf{w},\xi} \ \frac{1}{2}\|w\|_2^2 + C\sum_{i=1}^{m}\xi_i \tag{3.2}$$

$$\text{s.t. } y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1 - \xi_i \qquad \forall i \in \{1,\dots,m\} \tag{3.3}$$

$$\xi_i \geq 0 \qquad \forall i \in \{1,\dots,m\} \tag{3.4}$$

where C is a parameter determining the misclassification penalty. This formulation is also called the *soft-margin* C-SVM, because it allows for misclassifications and penalizes them. We typically focus on the dual representation of this problem, obtained from its Lagrangian (we omit the full derivation for brevity):

$$\text{maximize}_{\alpha} \ W = \sum_{i=1}^{m}\alpha_i - \frac{1}{2}\sum_{i,j} y_i y_j \alpha_i \alpha_j \langle \mathbf{x}_i, \mathbf{x}_j \rangle \tag{3.5}$$

$$\text{s.t. } 0 \leq \alpha_i \leq C \qquad \forall i \in \{1,\dots,m\} \tag{3.6}$$

$$\sum_{i=1}^{m} y_i \alpha_i = 0 \tag{3.7}$$

where the classifying function becomes $f(\mathbf{x}) = \text{sign}(\sum_{i=1}^{m}\alpha_i y_i \langle \mathbf{x}, \mathbf{x}_i \rangle)$ and $\langle \cdot, \cdot \rangle$ denotes the inner product between vectors.

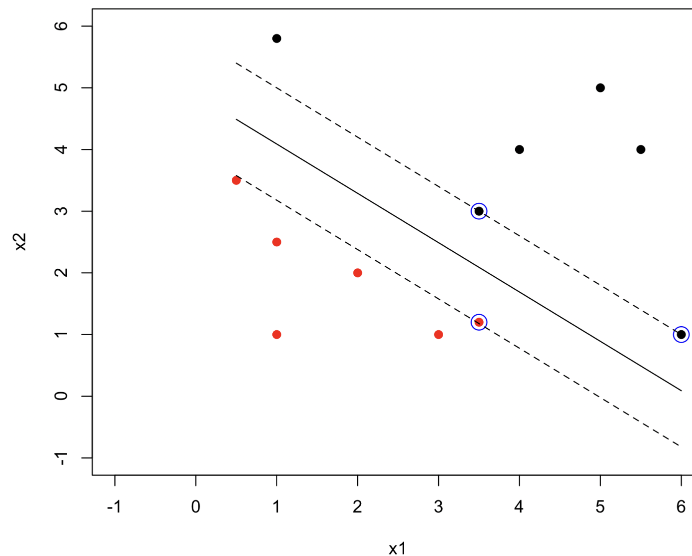


Figure 3.1 – The hyperplane created by an example of Support Vector Machine, using a linear kernel. The margin support vectors ($0 < \alpha < C$) are circled.

Training samples are often not linearly separable in their original space. To alleviate that problem, we can map the samples to a space with higher dimension. Thus, we can think of a

transformation function $\phi : \mathcal{X} \rightarrow \mathcal{F}$ that maps examples to a feature space. Noticing that we only need to know the value of inner products between vectors, not their representation in this new feature space, allows us to use the *kernel trick*.

This is done by replacing the inner product in Equation 3.5 and in the decision function by a function $k(x,x')$ which has to respect Mercer's conditions [21] to be equivalent to an inner product in some feature space. In simple words, this kernel function needs to be *symmetric* – $k(x,x') = k(x',x)$ – and *positive definite* – $\sum_{i,j} a_i a_j k(x_i,x_j) \geq 0$.

From the form of the decision function and the Karush-Kuhn-Tucker conditions for this problem, we see that it is only necessary to keep a few vectors from the training data to perform classification. These are the ones for which α_i is not 0, which happens for the *support vectors* $(0 < \alpha_i < C)$ and for the *error vectors* $(\alpha_i = C)$.

With those characteristics, we can update such a model upon seeing a new training sample $(\mathbf{x}_c, y_c)$ in different ways. Some works suggest training a new model on the previous model's support and error vectors and the new sample [29]. Since this gives only approximate results [3], we direct our attention to two alternatives, one of which we chose to implement.

3.2.2 Incremental SVM: A Locality-Based Approach

In the work described in [25], they explore the inherent *locality* that some kernel functions have. For example, the Radial Basis Function kernel is given by $k(x,x') = e^{-\gamma\|x-x'\|^2}$, where γ is a free parameter.

The parameter γ can be related to the inverse of the *radius* of this kernel. The bigger it is, the faster the value of $k(x,\cdot)$ decays when we move away from x. This implies that, when introducing a new example in our dataset, the kernel function will only have a non-negligible value in a neighborhood around the new point.

The algorithm works by recomputing the solution to the dual C-SVM problem, but optimizing only over a subset of the Lagrange multipliers α_i. This subset is determined by selecting the vectors in a neighborhood around the new example.

The generalization error of the new classifier is then estimated, and the neighborhood is grown if this error estimate is not under some threshold. We then solve the dual C-SVM problem again, but this time, optimizing over a bigger subset of the Lagrange multipliers.

This is a first approach that shows us that recomputing the solution to an SVM problem is feasible incrementally and might save us processing time.

However, this algorithm does not handle concept drift – in fact, it was one of the ideas for future work in the original paper. Since that is a property we find desirable in an algorithm that will be applied to a data stream, this counts negatively towards picking this technique. Perhaps using an aging function to discard old examples and recompute the solution to the SVM problem would be a suitable solution, but we chose not to explore this for time reasons.

In the end, we chose to skip implementing this technique, and cite it here for the ideas it introduces and for its highly comprehensible "local perturbations" scheme.

3.2.3 Incremental and Decremental SVM

Another possibility for updating a Support Vector Machine upon the arrival of a new training sample arises from the analysis of the Karush-Kuhn-Tucker conditions for optimality. In the work presented in [3], new examples are added to the current SVM in a way that keeps these optimality conditions valid.

The KKT conditions for the dual C-SVM problem are (using $Q_{ij} = y_i y_j k(\mathbf{x}_i, \mathbf{x}_j)$):

$$g_i = \frac{\partial W}{\partial \alpha_i} = \sum_j Q_{ij}\alpha_j + y_i b - 1 = y_i f(\mathbf{x}_i) - 1 \begin{cases} > 0; & \alpha_i = 0 \\ = 0; & 0 < \alpha_i < C \\ < 0; & \alpha_i = C \end{cases} \tag{3.8}$$

$$\frac{\partial W}{\partial b} = \sum_j y_j \alpha_j = 0 \tag{3.9}$$

which partition the training dataset $\mathcal{D}$ in three sets: the set $\mathcal{S}$ of on-margin *support* vectors; the set $\mathcal{E}$ of *error support* vectors (which may or may not be misclassified); and $\mathcal{R}$, the *reserve* vectors.

When adding a new example $(\mathbf{x}_c, y_c)$ to the current SVM, it helps to see these conditions from the differential point of view:

$$\Delta g_i = Q_{ic}\Delta\alpha_c + \sum_{j\in\mathcal{S}} Q_{ij}\Delta\alpha_j + y_i \Delta b, \quad \forall i \in \mathcal{D} \cup \{c\} \tag{3.10}$$

$$0 = y_c \Delta\alpha_c + \sum_{j\in\mathcal{S}} y_j \Delta\alpha_j \tag{3.11}$$

where α_c is the Lagrange multiplier of the example being added to the training set and is initially zero.

From these we can determine how much a change in α_c causes changes in the existing Lagrange multipliers $\alpha_j, j \in \mathcal{S}$, the bias b and the distance of other vectors to the margin. Using β to refer to how much the bias changes, and $\beta_j, j \in \mathcal{S}$ for the sensitivity of the Lagrange multipliers and defining $\mathcal{Q}$ as the extended kernel matrix for the support vectors:

$$\mathcal{Q} = \begin{bmatrix} 0 & y_{s_1} & \cdots & y_{s_{l_S}} \\ y_{s_1} & Q_{s_1 s_1} & \cdots & Q_{s_1 s_{l_S}} \\ \vdots & \vdots & \ddots & \vdots \\ y_{s_{l_S}} & Q_{s_{l_S} s_1} & \cdots & Q_{s_{l_S} s_{l_S}} \end{bmatrix} \tag{3.12}$$

$$\Delta b = \beta \Delta\alpha_c \tag{3.13}$$

$$\Delta\alpha_j = \beta_j \Delta\alpha_c \tag{3.14}$$

$$\begin{bmatrix} \beta \\ \beta_{s_1} \\ \vdots \\ \beta_{s_{l_S}} \end{bmatrix} = -\mathcal{Q}^{-1} \begin{bmatrix} y_c \\ Q_{s_1 c} \\ \vdots \\ Q_{s_{l_S} c} \end{bmatrix} \tag{3.15}$$

For non-support vectors, the β_j sensitivities are all zero. In this case, what we care about is how their distance to the margin changes, which we call *margin sensitivities*.

$$\Delta g_i = \gamma_i \Delta\alpha_c \tag{3.16}$$

$$\gamma_i = Q_{ic} + \sum_{j\in\mathcal{S}} Q_{ij}\beta_j + y_i \beta \tag{3.17}$$

One advantage of this procedure is that whenever a new support vector is added to the $\mathcal{S}$ set, we can update the matrix $\mathcal{R} = \mathcal{Q}^{-1}$ incrementally, instead of recomputing the inverse. Also,

this approach supports *unlearning* of vectors, which gives it its name and allows for simple computation of leave-one-out estimates.

The algorithm works by testing if the new example is well classified and respects the margin of the current classifier. In case it does, it is stored as a reserve vector and the procedure terminates.

Otherwise, the vector must be either a support or an error support vector. Successive increments are then applied to its corresponding Lagrange multiplier. Since changes to α_c cause changes in margins and in the other Lagrange multipliers, we need to keep track of moments where vectors change status. We stop incrementing α_c when the value of g_c hits zero – in this case, the new vector goes into the set $\mathcal{S}$ of support vector – or when α_c itself hits C – and the new vector is an error support vector.

Practical ways of determining the increments to be used for α_c are described in [13, 7]. The latter also introduces a way of learning batches of new examples at once instead of only a single vector.

This procedure does not account for concept drift. Also, we need at least one example from each class to create an initial solution, otherwise the SVM problem is ill-posed. This has direct implications to the way we chose to implement this solution.

3.3 Neural Networks

A Neural Network is a model that uses compositions of differentiable transformations on data to successively map it to spaces where it can be more easily classified. The history of this model starts with Rosenblatt's Perceptron [26], but the model only started being intensely used after the development of the *backpropagation* algorithm. This procedure is an efficient way of computing derivatives inside functions that can be represented as graphs. Backpropagation is a smart way of applying the chain rule. It enables us to use methods that rely on gradient information, such as Stochastic Gradient Descent, to minimize some cost function.

For example, a Multilayer Perceptron is a feedforward, fully-connected neural network. In the case where we have only one hidden layer, we can look at it from a completely mathematical standpoint (here, the binary classification case is shown):

$$a_j = \sum_i W_{ij}^{(1)} x_i + b_1 \quad \forall j \in \{1, \dots, l\} \tag{3.18}$$

$$z_j = \sigma(a_j) \quad \forall j \in \{1, \dots, l\} \tag{3.19}$$

$$o = \sum_j W_j^{(2)} z_j + b_2 \tag{3.20}$$

$$\hat{y} = \sigma(o) \tag{3.21}$$

where l is the number of neurons in the hidden layer and $\sigma(x) = 1/(1+e^{-x})$ is the sigmoid function, used as activation function in this case. The output $\hat{y}$ encodes the probability of the positive class, in this case. A suitable error measure for training an MLP would then be

$$\mathcal{L}_{CE}(h, D) = \frac{1}{m} \sum_{i=1}^{m} \left(y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i)\right)$$

which is the cross-entropy loss.

A Neural Network is originally an algorithm made to operate in *batch* mode. It is, however, applicable to streaming environments if we are aware of how its training happens. When training a neural network, it is common practice to use mini-batches of data — smaller sets of training examples — as opposed to using the whole dataset at once. We can, under some adaptations, look at chunks of data from our stream as mini-batches, and proceed to train a neural network over time.

3.3.1 Recurrent Networks on Streams

For problems where the goal is to learn about sequences, such as in speech recognition [14], translation [28], image captioning [6] and time series forecasting [1], a commonly used model is the Recurrent Neural Network (RNN). Many architectures for those exist, but what sets them apart from other types of neural networks is the fact that they can have *loops*.

These loops propagate the previous state of the network across time, and can be used to retain information from the past to help make predictions about the future. In the case of translation, this means that when classifying the second word, the network's output will be influenced by the first word too. We always expand the recursive neural network long enough to match the length of the sequence we are using as input. A consequence of this is that we need to form sequences of data both for training and for testing the network.

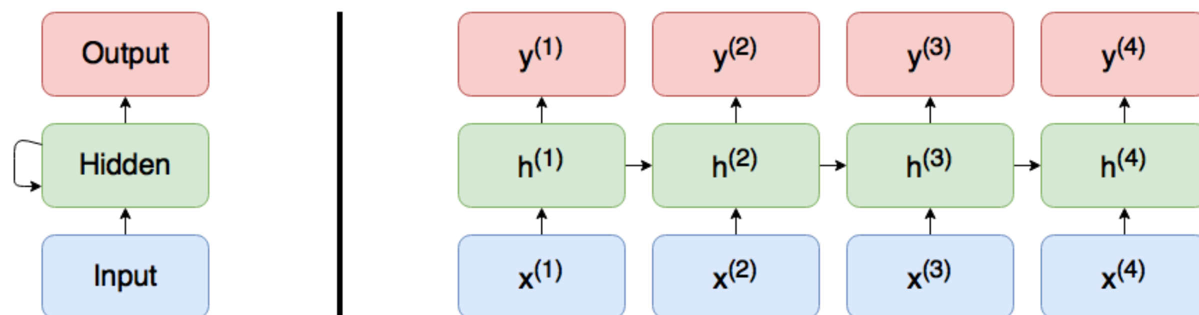


Figure 3.2 – The concept of a Recurrent Neural Network (left) and its "unrolled" form (right), where each element of the sequence feeds into a different node.

There are some issues with training RNNs, notably the *vanishing gradient* and the *exploding gradient* problems. This happens because, inside an RNN, a linear transformation is repeatedly applied to the hidden state (in the longest path inside the RNN, it is applied l times, where l is the length of the input sequence). Since this linear transformation also affects the gradient, when the transformation matrix has spectral radius ρ different from 1, the gradient either vanishes ($\rho < 1$) or explodes ($\rho > 1$).

More precisely, the hidden states $h^{(t)}$ are calculated as $h^{(t)} = Wh^{(t-1)} + Ux^{(t)}$. When propagating errors back through time, the linear transformation $x \mapsto Wx$ is also applied to the error. This means we will have an error with a component of the form $\varepsilon = W^k E$, where E is the final error of the network. A little Linear Algebra shows us the result stated above, involving the biggest eigenvalue in absolute value of the matrix W – in other words, its spectral radius ρ.

A reformulation of the RNN was given in [16] and is called Long Short-Term Memory unit. This type of cell does not suffer from the problems elicited above because it uses more complex mechanisms to reformulate the propagation of the hidden state. It has also proven to

be effective in different types of problems. Here, we explore its application on learning the behavior of a stream.

— 4 —
Implementation and Experiments

From the techniques described in the previous chapter, we chose to implement and evaluate Learn++, the Incremental and Decremental SVM algorithms, and to also explore the use of neural networks with LSTM gates—all in the setting of incremental learning.

In the experiments that follow we used both synthetic and "real-world" datasets that are known to have issues that incremental algorithms need to account for.

4.1 Design Choices

The language of choice was Python 3.5, and the implementations use the libraries Numpy and Scikit-Learn. Preprocessing of the datasets was done in both Python – with the Pandas library – and R. The algorithms were implemented in separate source files and imported from Jupyter Notebooks to intersperse explanations, texts and formulas with code. The API for the classifiers follows the one used by Scikit-Learn: every classifier possesses a `fit` or `partial_fit` function and a `predict` function, at least.

The `fit` functions are called to make the classifier *learn* from new data. Both the examples and their labels must be given. The `predict` function receives only the examples from the input space and returns the predicted classes for them.

Also, when employing incremental algorithms, we need to transform datasets to streams. Learn++ expects to receive chunks of data, while our version of the Incremental and Decremental SVM learns *online* (i.e. using individual samples). For the LSTM Network, we chose to use *jumping windows*, where we slide a window using a stride bigger than 1. This was employed so that the neurons would be allowed to learn over intersecting windows of data, possibly causing them to detect moments of concept drift.

Since none of the implemented algorithms are tailored to deal with categorical variables in the data, we use the technique of *one-hot encoding* in those cases. For example, if a variable x_j can take any one of the values in a set $\mathcal{V}$ with size v, we encode this as a v-dimensional vector is the following way:

$$[E(x_j)]_l = \begin{cases} 1 & \text{if } \mathcal{V}_l = x_j \\ 0 & \text{otherwise} \end{cases} \quad , \forall l \in \{1, \ldots, v\}$$

4.2 Implemented Algorithms

Since some changes were made to the original algorithms when implementing them, we describe the reasons for those changes and the problems encountered when testing the algorithms.

4.2.1 Learn++

We implemented the NSE and NIE variants of Learn++, as well as modified versions of the NC and SMOTE variants. In all of those, special measures were taken to make sure the space complexity of the algorithms is bounded by a constant. Since all of these methods build many classifiers during their execution, we need to limit the number of classifiers that can be kept in memory. This calls for a way to choose which classifier will be discarded whenever we hit this limit.

One simple idea is to drop the classifier that has the smallest voting weight. Because every variant of the procedure uses some sort of voting weight, it is a uniform way of dealing with the problem. Using the most recent error of a classifier or its age are also valid options, and are suggested by the authors, although no theoretical reasons for them are given [10].

Since Learn++ relies on weak learners to work, we had to choose the base model. In all of the papers that introduced Learn++ and its variants, it was reported that the choice of weak learner does not have a strong influence on the performance of the algorithm. Therefore, we chose to use Scikit-Learn's implementation of Multilayer Perceptrons. We explored the use of a C-SVM as weak learner, but they took longer to train without a noticeable payoff.

In the particular case of Learn++.SMOTE, we explored the use of a *replay memory*, an idea borrowed from the Reinforcement Learning literature [22]. It consists of keeping "past experience" in memory and learning from that. In our setting, this means we keep a window of data when dealing with highly imbalanced datasets. This window contains the last w examples of each class, and these examples are used whenever we need to over-sample a class, but do not have enough data for it. An obvious drawback of that approach is its vulnerability to concept drift: since we are using old data to learn under a possibly new concept, our weak learner might not be able to provide us with good classifications.

4.2.2 Incremental and Decremental SVM

When implementing the Incremental and Decremental SVM, special attention should be given to numerical instability problems, where by manipulating the $\mathcal{R}$ matrix repeatedly, we might ruin its invertibility. To counter this problem, we can add a small number ε to new diagonal elements when expanding the matrix.

Also, if a redundant vector is added to the support vectors set – where the definition of redundant is explained in [13] – this could make the original matrix $\mathcal{Q}$ have two identical rows, and therefore be non-invertible.

On top of that, when elements migrate across error, support and reserve sets, we need to determine their associated sensitivities and/or Lagrange multipliers. None of the explored papers explain how to do this. Attempts to use the constraints of the problem yielded inconsistent results so far (e.g. negative Lagrange multipliers).

4.2.3 LSTM Neural Network

Using the Keras library for Python, building a network for classification of sequences is fairly straightforward. Since it is not the goal of this work to investigate the best architecture possible, but whether it makes sense or not to use such an algorithm for this problem, we use the same hidden architecture for all the examined datasets.

Our network is composed of 2 hidden layers. The first one is an LSTM layer with 100 neurons – this is the recurrent part which connects the sequence over time. For each one of the unrolled cells, there is a fully connected layer with 5 neurons, and the output layer.

4.3 Experiments

To give an idea on the applicability of incremental models to a streaming environment, many types of experiments can be done. They do need to be carefully designed, so that they test consistent hypotheses. For example, comparing batch and incremental models is something that is not easily done without introducing some bias towards one or the other. We conducted the following experiments:

- Comparison of the implemented techniques with SAP HANA SDS's current stream classification algorithm [9, 18], with the goal of evaluating how the performance of the proposed methods fares with respect to the current method;

- Comparison between batch model and the proposed incremental models. To make this comparison fair, it is necessary to eliminate as many variables that can affect accuracy as possible;

  - Training time for batch: since batch algorithms typically take longer to train, we should explore what happens when we train them for a small number of iterations (comparable to the time taken to incrementally update the proposed model).
  - It is difficult to determine a batch model that is "equivalent" to an incremental learner, so we cannot base our evaluations on that.

- Evaluation of the way all algorithms respond to concept drift, class imbalance and the introduction of new classes.

The experiments need to account for the fact that, in a real streaming environment, training data and test data can be provided in parallel. This makes the classification process more complex, for example in the case where data to be classified arrives while we are training a new model. If such thing happens, our old classifier should be used for classification while the new one is being created.

When testing the proposed algorithms, we need to simulate this real world scenario where both streams happen in parallel, even though the code runs sequentially. In practice, we measure the time it takes to train a new classifier, and then based on the stream intensity, we determine how many test examples were "missed" by it. We then use a copy of the old classifier to predict the classes of those ones, avoiding accidental "cheating" (using a future classifier to predict past events).

Parameters Since the variants of Learn++ have many parameters, here are the ones we used and why:

- The *weak learner* was a Multilayer Perceptron, with 2 hidden layers: one with 100 neurons, and the other with 5.
- For the algorithms who use the weighted recall error measure (Equation 3.1), η was set to 0.65.
- Learn++ has an internal loop that tries to build a weak classifier, but when it performs too poorly, another one needs to be trained. In degenerate cases, this was seen to cause infinite loops. To mitigate this situation, we limit the maximum number of *tries* for this loop. It was set to 2 in our tests.
- In order to learn weak classifiers, we subclass Scikit-Learn's Multilayer Perceptron implementation and allow for the manual setting of an *error goal*. When this goal is met, the training stops, providing us with a sufficiently weak hypothesis. We set the error goal to be 0.2 in tests.
- For SMOTE super-sampling, we create as many artificial examples as needed to make the minority class have at least 20% of the size of the majority class.

4.4 Datasets

Two datasets were explored so far in this work, one of which is synthetic.

SEA Concepts This is a synthetic dataset containing 60000 samples. The input vectors are three dimensional, and only two of those dimensions are important in determining their classes. The class is determined by thresholding the value of the sum of the last two components of each vector. This dataset has four different concepts (here, determined by the thresholds used to assign the classes) and about 10% noise. These aspects are shown in Figure 4.1.

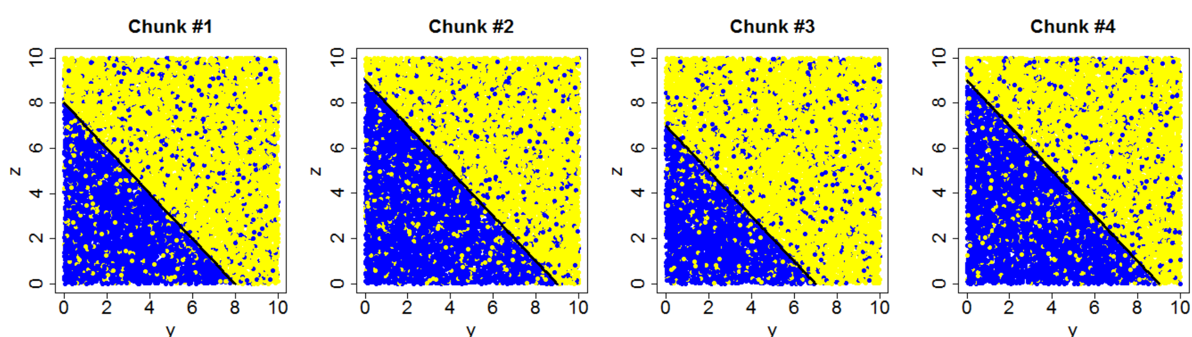


Figure 4.1 – A visualization of concept drift in the SEA dataset. Yellow and blue represent the negative and positive classes, respectively. The two relevant variables used to determine the class of the examples are on the x and y axes. It is visible that their sum determines an example's class – except for noisy examples –, and the threshold for each chunk is shown by a solid line. The thresholds of each of the four chunks are, respectively, 8, 9, 7 and 9.

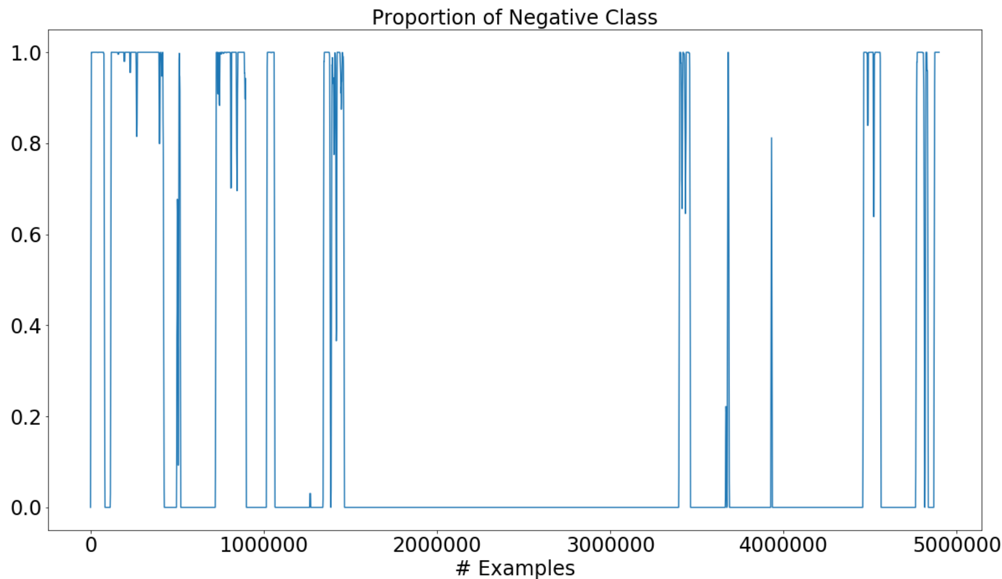


Figure 4.2 – This graph shows how many of the examples come from the negative class in the KDDCup dataset, using blocks of 5000 elements. It can be seen that this dataset is highly imbalanced, with one of the classes dominating the block most of the time. Also, there are abrupt changes in proportions. We would like our algorithms to be resilient to such behavior.

KDDCup Invasion Detection This is a real world dataset containing around 5 million data points, with 122 dimensions each (after preprocessing). The goal in this dataset is to detect malicious connections. In the original data, those appear under different types, which we replace by a single "abnormal" label. This dataset presents intense class imbalance, as can be seen in Figure 4.2, which shows the prevalence of the negative class on chunks of 5 thousand elements from this dataset.

4.5 Comparison to Batch Models

When comparing to Batch Models, a browser-based visualization was developed using Python and the Bokeh library. The objective behind it was to show the accuracy of both algorithms side by side, while giving notions on time spent training in both batch and incremental settings.

This is of practical importance to SAP since using less processing power to keep a model up to date could be a reason for customers to shift from retraining batch models to using incremental ones.

Figure 4.3 is a screen capture of the developed visualization for this data. The training time is shown by the shaded region in the graphs, while the accuracy is represented by the blue line. The average, minimum and maximum accuracy values are also reported.

The example shown in Figure 4.3 is the comparison between a Multilayer Perceptron learned in batch mode and Learn++.NSE. The main takeaways from these graphs are:

- Both algorithms perform comparably well: they show mostly the same trends of rise and fall in accuracy, and their average accuracies only diverge by 0.01;

Figure 4.3 – A screen capture of the visualization developed to compare the behavior of batch and incremental algorithms. The shaded regions (which are very narrow and close to each other in the bottom half) correspond to time spent training a model. The overall proportion of time spent training is reported on the right side of the window.

- The incremental approach reduces the time used to train the model to one sixth of the time used in the batch model, without incurring any grave loss in accuracy.

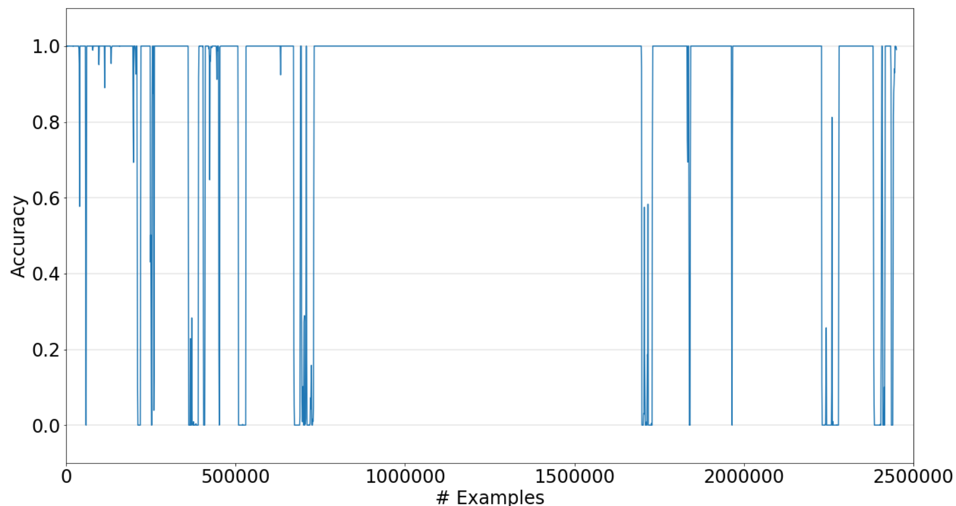
4.6 Comparisons Amongst Incremental Models

Varying parameters of our algorithms might give us insight on their behavior and sensitivity. One example of parameter that can be easily modified is the chunk size we use when processing the stream. Specific algorithms such as Learn++.NSE also have parameters for the used sigmoid – the slope and the inflection point. For that specific case, we default to the values recommended by the authors, but some other variables such as the η factor in Learn++.NIE should be tuned depending on the problem at hand.

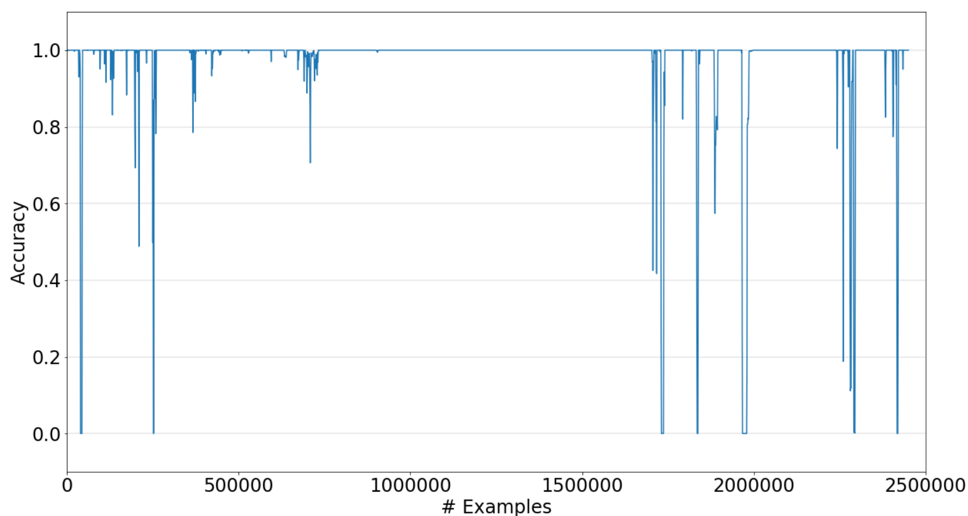
4.6.1 Replay Memory

While exploring imbalanced datasets, the idea of using a sort of replay memory was tested. This idea was inspired from the "experience replay" technique used in [22].The intention behind its use was to be able to generate examples from the minority class from more "real" minority class data, instead of using only current examples, that could be scarce.

The change in the behavior of the overall accuracy is noticeable from Figure 4.4. The only parameter that was changed between the two examples shown in the figure was the use of a replay memory that keeps the last 1000 examples in memory. This is still compatible with the incremental learning scenario since we use a constant amount of memory for this structure. Also, we are not storing this permanently and performing multiple passes over it: data in the replay memory will be lost when it becomes too old.



(a) Behavior of accuracy over time, without employing a replay memory.



(b) Behavior of accuracy over time, while employing a replay memory.

Figure 4.4 – Influence of the use of a "replay memory" in Learn++.NIE+SMOTE. While still presenting cases where the accuracy falls to zero, its variations are less frequent – overall, the graph looks less noisy.

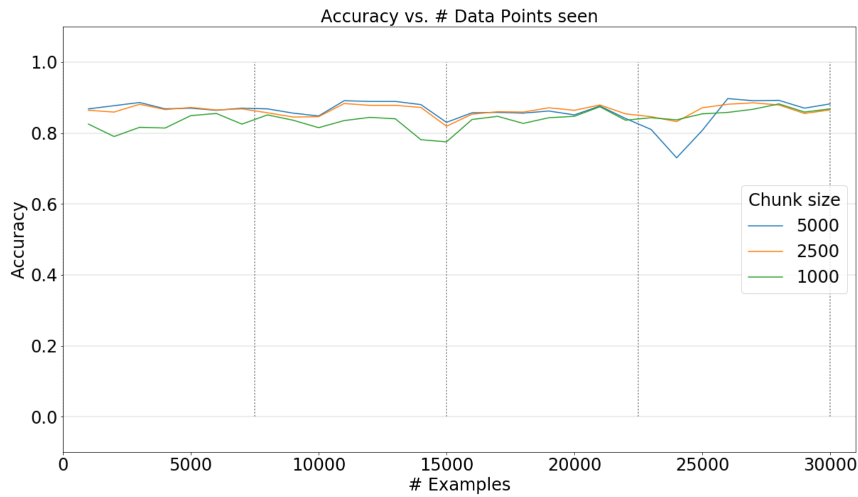
This result points us in the direction that, even if using old data might be harmful under concept drift, it can still be better than extrapolating from too little current data.

4.6.2 Window Size

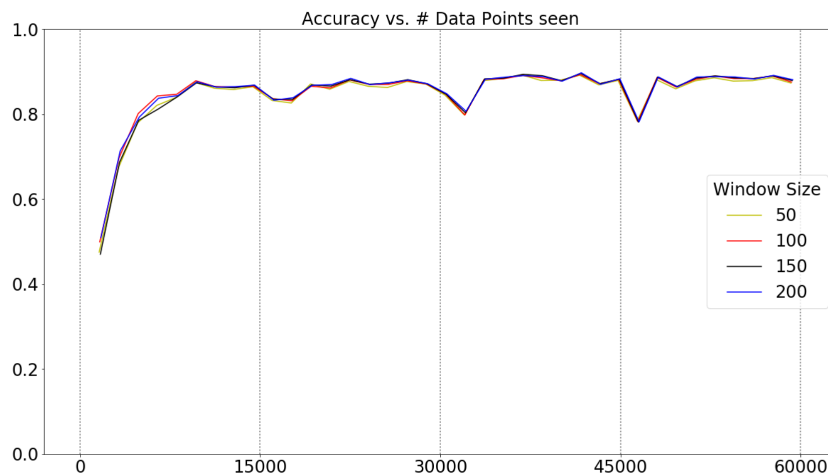
Changing the window/chunk size when training incremental algorithms might have an impact on the time taken to respond to a change in concept, or to stabilize a new concept [20].

This happens intuitively because, when using short windows, examples age faster, since our notion of "time" inside the algorithm is given by the creation of new classifiers. An algorithm that uses small chunks of data would then adapt to new concepts swiftly.

Analogously, using larger windows gives us more data to train classifiers on. When learning under a stable concept, or on a stream that does not present concept drift, our classifier would



(a) For Learn++, we see a difference around the 22500 examples mark, which is one of the times when concept drift occurs. In this occasion, using a bigger chunk size causes a sharper drop in accuracy, and takes longer to recover.



(b) For the LSTM model, it does not seem to affect significantly the behavior of accuracy over time. Each measurement was averaged over 10 runs. The stride used for the windows was 50 elements. This means that the next window used for training intersects the old one for window sizes of 100, 150 and 200 elements.

Figure 4.5 – The behavior of accuracy over time when varying the chunk/window size. The dataset used for this test was the SEA Concepts dataset, that has concept drift – shown here by the vertical, dotted lines.

then have access to a lot of cohesive data. Intuitively, this would raise its predictive power. Figure 4.5a shows the results obtained when varying the chunk size for Learn++.NSE, while Figure 4.5b shows the same for an LSTM network.

The window sizes for the LSTM network tests are considerably smaller than the ones used

to test Learn++. This is due to the fact that we noticed no evolution in performance for using windows in the order of thousands of elements, and they took considerably longer to train.

4.6.3 Accuracy of Incremental Models

For completeness, we show how the explored models' accuracy behaves over time. This is where the results of the LSTM network are presented in comparison to those of Learn++.

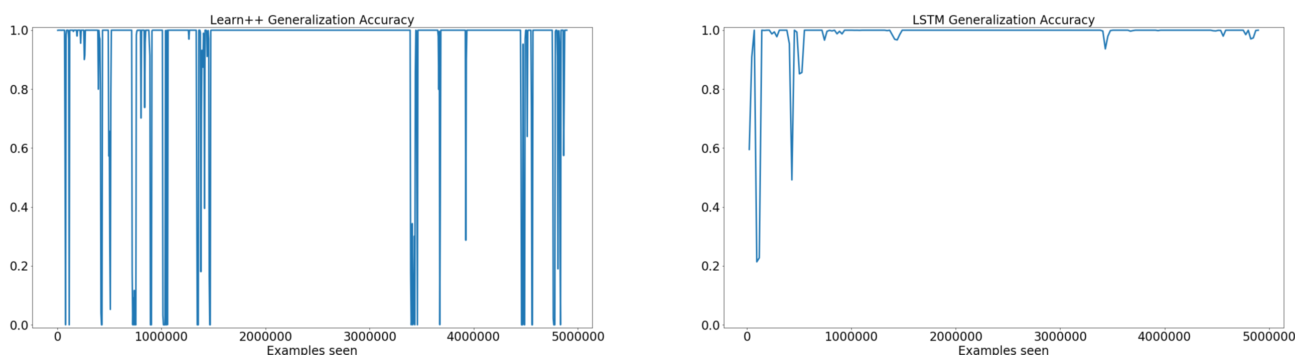


Figure 4.6 – These are the accuracy results for both Learn++.NSE and LSTM networks. The accuracy value reported is obtained by evaluating the current classifier on the next chunk or window of data before using it for training. The version of Learn++ presented does not use a Replay Memory.

The results shown in Figure 4.6 surprised us in the sense that the LSTM network seems to be resilient to sudden changes in the proportions of the classes. While Learn++ falls to zero accuracy whenever the dominating class changes, the LSTM network shows smaller variations.

Again, since we found no literature backing up the use of LSTM networks for classification of streams, it is hard to determine the precise cause of this behavior. This issue is one that should be studied in the future.

4.7 Comparison to SAP HANA SDS

To compare the performance of the implemented algorithms to what is currently available in SAP's streaming platform, we need to adapt the data to its playback file format. We use this file type because it allows us to use timestamps and simulate the training stream in parallel with the test stream.

4.7.1 Building the Project

To use SAP HANA SDS we need to first create a project. In our case, it will be composed of two *input streams*, one for training and one for testing. The relevant information from both streams is projected through an SQL-like query component. The diagram corresponding to the project is shown in Figure 4.7

We also need to declare the models composing our project in a Data Service. When doing so, it is necessary to specify the input and output schemas for the models. Even if we are using

only one model to do prediction, we need to build two in HANA: one explicitly for training, and another one explicitly for predicting – or, in SAP HANA SDS terms, "scoring". The latter *references* the former, in this case, and uses it to make its predictions.

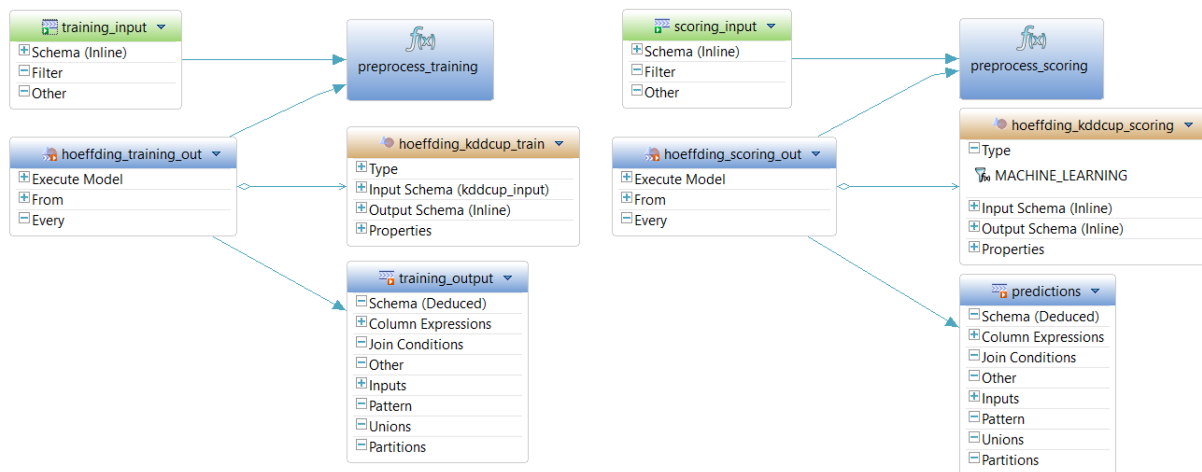


Figure 4.7 – An example of project used to train a decision tree and use it for scoring.

For training, the model receives the feature values and the correct class info. For prediction the model receives an identifier for the data point, and the feature values for it. The training model outputs an accuracy measure, while the scoring model outputs, for each data point fed into it, its identifier, the predicted class and the degree of confidence in this prediction. All of this information is collected in an output stream for later inspection.

4.7.2 Importing the Data

To import data into the streams, we use playback files. The playback file specifies, in each row, three main things: the *stream name*, which is the identifier of the stream that should receive this row's data; the *operation code*, which in our case is always `insert`; and the *data*. We also introduce a timestamp column that was artificially generated as a way to order the data.

The fact that the timestamp information is contained in the playback file is what allows us to simulate parallel streams inside SAP HANA SDS. For every training point that arrives to adapt our model, a test point will be fed into our prediction function.

Since the algorithm used in SAP HANA SDS allows for categorical values in its columns, we do not need to perform one-hot encoding as is the case with the alternative algorithms that were implemented.

4.7.3 Results

SEA Concepts Dataset Learn++ was trained using blocks of 1000 elements each in this case. The proposed algorithms perform at least on par with the one currently implemented in HANA SDS most of the time in terms of accuracy. No big difference between the variants of Learn++ is expected in this case, but the NIE+SMOTE and NC versions seem to perform a little better than the pure NSE. We attribute this behavior to the use of the weighted recall error measure.

It is also valuable to point out that these results are dependent on the order in which we see the data when analyzing the stream. Reordering examples *within* the concepts should not change how accuracy behaves over time, but mixing up the four different concepts will essentially destroy them, and concept drift will therefore be nullified.

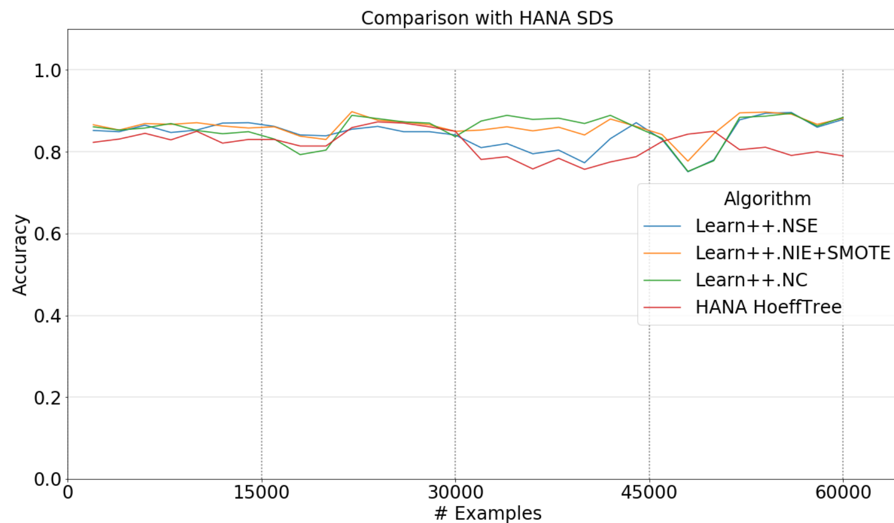


Figure 4.8 – Accuracy comparison between variants of Learn++ and the algorithm currently implemented in SAP HANA SDS. The vertical dotted lines show the points where the concept drifts in the dataset.

KDDCup Dataset We used a subset of the KDDCup dataset, containing around 250000 elements. This was enough to notice how both HANA's and our alternative algorithm behave (see Figure 4.9). Their behavior was comparable, with sharp drops in accuracy happening for both of them at times.

In this case, too, shuffling the data would turn the learning problem into a much easier one, with respect to concept drift and class imbalance. The adversarial nature of invasions over time — where safety measures are taken and later broken — makes the order of the examples very relevant for the outcome of a learning algorithm in this scenario.

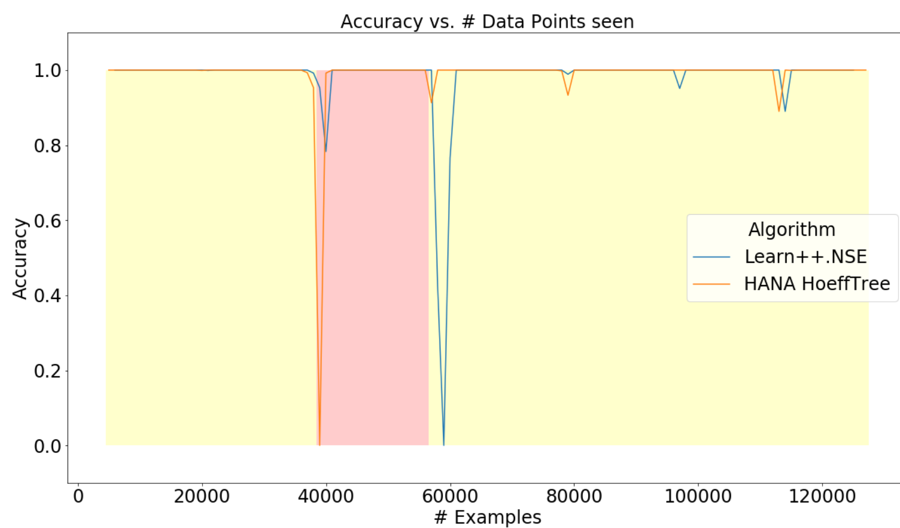


Figure 4.9 – Both algorithms' accuracies collapse to zero at distinct points, but present otherwise comparable values. The background color indicates the current majority class (yellow = negative, red = positive). This lets us have an idea of moments where concept drifts and relates to the drops in accuracy.

— 5 —

Conclusions and Future Work

The work started in this thesis is still in progress, and there is still ground to cover. Some initial conclusions can, however, be drawn at this point.

We presented an overview of the field of Incremental Learning. While not exhaustive, it provided us with a reasonable idea on how these algorithms work, the problems they solve and the ones they introduce.

The investigation of these methods under circumstances of concept drift and class imbalance – which are cases that can happen in the real world – led us to a finer understanding of use cases for these techniques. Building on this, we could also explore models that are, to the best of our knowledge, not present or sufficiently explored in the Incremental Learning literature – e.g. LSTM Neural Networks. These seem to perform well, being resilient to both class imbalance and concept drift, which indicates that they should be more thoroughly investigated and tested. The goal of this testing would then be to find the cases where this approach does not work and what affects its behavior negatively.

Using a replay memory in scenarios where there is an evident minority class showed to be a valuable technique. We attribute that to both the facts that less over-sampling is needed and that when we do over-sample, we have more "real data" from which we can derive artificial data. It is worth mentioning once more that this technique might backfire under concept drift if we use samples from an obsolete concept to expand the current chunk.

The algorithms that were implemented and tested so far perform at least on par with the Hoeffding Decision Trees that are currently implemented in SAP HANA SDS. Therefore, their integration into SAP's system would make sense and could give users more options on the type of model they want to use. Another positive effect of that, would be to use less processing power while keeping the classifier up-to-date with more frequency, and thus intuitively "missing" less data and being able to make more informed predictions.

Creating a visualization for the accuracy and training time of both batch and incremental algorithms side by side provided us with a more tangible idea on the latter's potential. The fact that we can train more frequently, with less intensity, and still achieve comparable accuracy values to batch models is something that motivates the implementation of Incremental Methods inside SAP's streaming analytics platform.

As for future work, we need to complete the implementation of the Incremental and Decremental SVM. This proved to be more difficult than expected, since the papers describing it do not explain in detail how the migration between sets occurs.

Another point is coming up with a full end-to-end use case for a business analytics scenario. This would be easier for SAP to grasp and build a case on, potentially investing time and human

resources into the development of such approaches and improving their streaming analytics product.

While built inside SAP and with focus on SAP's Data Streaming platform, the results obtained by this work are not limited to it. We believe that the techniques explored here may generalize to both streaming and non-streaming problems — e.g., scenarios where datasets do not fit in memory or in which we might want to force learning to happen over time for analysis —, inside and outside SAP's context.

— A —

Appendix

A.1 Changing the Sigmoid in Learn++.NSE

In this Appendix we discuss how the weight decay scheme – in particular, the sigmoid function – used by Learn++.NSE affects the behavior of the weights given to classifiers over time. We started by analyzing the partial derivatives of the weighting of errors with respect to time. In other words, for a given, fixed classifier, we investigated how the weighting of its errors would change during the course of the stream. This a simple derivative to analyze:

$$\frac{\partial[(1+e^{-a(t-k-b)})^{-1}]}{\partial t} = -(1+e^{-a(t-k-b)})^{-2}e^{-a(t-k-b)}(-a) = \frac{a \times e^{-a(t-k-b)}}{(1+e^{-a(t-k-b)})^2} > 0 \quad \text{(A.1)}$$

The inequality holds since $a > 0$, $x^2 > 0, \forall x \in \mathbb{R}$, $e^x > 0, \forall x \in \mathbb{R}$. This means the value of the weighting factors of a single classifier *grows* over time. Such a result caught our attention. We then verified this by plotting the value of the weights over time for a classifier:

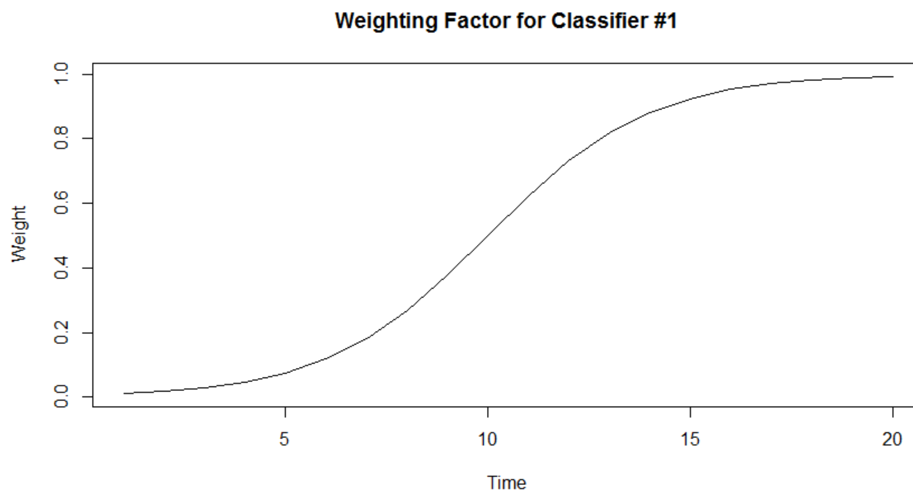


Figure A.1 – Behavior of voting weight over time for a single classifier. The weight grows over time according to a sigmoid function.

When all the weights are on either side of this curve, this does not pose any problem since we normalize them. However, this backfires tremendously once we get closer to the inflection

point, making the previous classifiers have voting weights that are too small and degenerating the current composite hypothesis to a myopic one.

For example, $t = 10$ timesteps after the creation of the first classifier, the normalized voting weights present the following values:

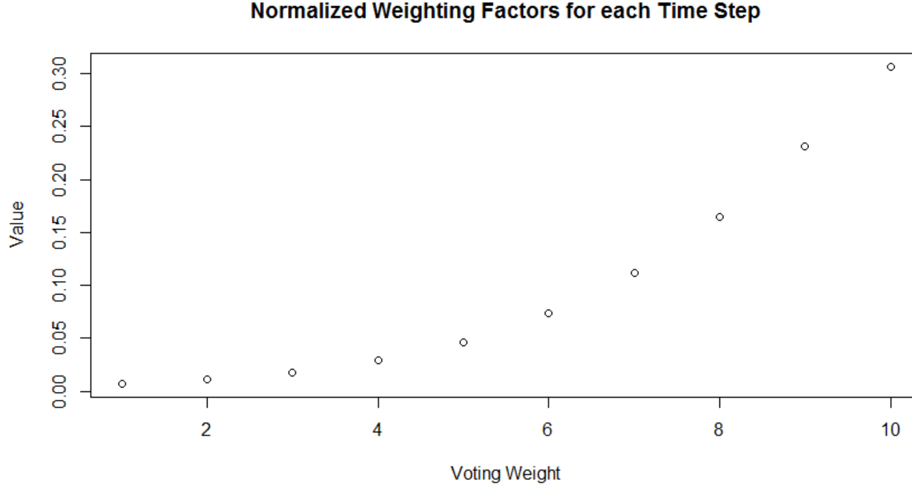


Figure A.2 – A comparison between the normalized voting weights for each classifier after 10 time steps. Recent classifiers are given too much strength because of the normalization applied on top of an inappropriate sigmoid function.

Notice how, even after little time, the oldest classifier already has a weight close to zero. Another indicator that this situation is not desired is the fact that, in this case, the sum of the 7 first voting weights is smaller than the last weight alone. This means that the errors of the 7 first classifiers together are given less importance than the most recent error.

We propose the use of a different weighting function, which has a behavior that we believe is more fitting to the problem. In our implementation, we use $1/(1+e^{-a(k-t+b)})$ to weight the classifiers' errors. We show why it is a suitable weighting factor below.

$$\frac{\partial[(1+e^{-a(k-t+b)})^{-1}]}{\partial t} = -(1+e^{-a(k-t+b)})^{-2}e^{-a(k-t+b)}a = \frac{-a \times e^{-a(k-t+b)}}{(1+e^{-a(k-t+b)})^2} < 0 \quad \text{(A.2)}$$

Its value *decreases* over time, which is already a good indicator. Plotting the values of the weighting factors over times confirms our hypothesis. In practice, the weighting factors now start "on top" of the sigmoid, and "roll downwards" over time, as shown in Figure A.3.

By doing this, we believe to have come up with a weight decay schedule that is more mathematically sound than the one proposed originally by the authors of Learn++.NSE. We employ this revised schedule in our implementation and consequently in our experiments.

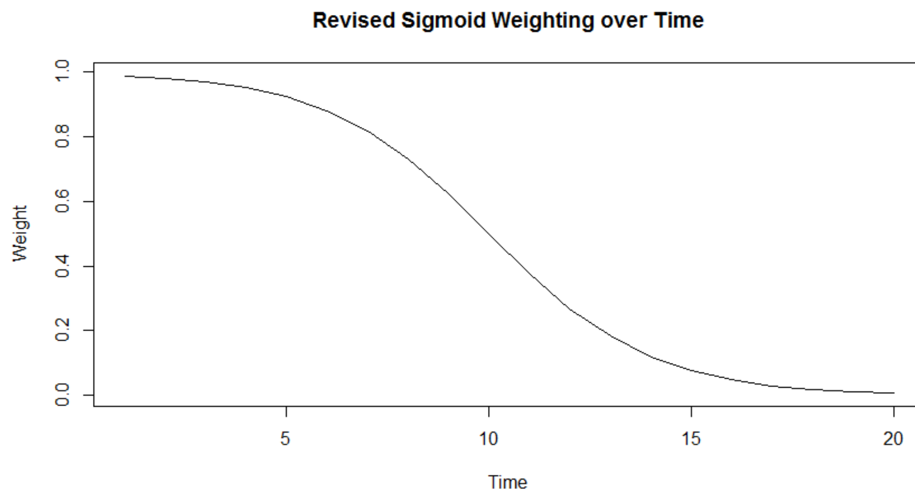


Figure A.3 – The revised weight decay schedule, where weights for each classifier decrease over time.

Bibliography

[1] Mohammad Assaad, Romuald Boné, and Hubert Cardot. A new boosting algorithm for improved time-series forecasting with recurrent neural networks. *Information Fusion*, 9(1):41–55, 2008.

[2] Feng Cao, Martin Estert, Weining Qian, and Aoying Zhou. Density-based clustering over an evolving data stream with noise. In *Proceedings of the 2006 SIAM international conference on data mining*, pages 328–339. SIAM, 2006.

[3] Gert Cauwenberghs and Tomaso Poggio. Incremental and decremental support vector machine learning. In *NIPS*, volume 13, 2000.

[4] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. SMOTE: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002.

[5] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[6] Jacob Devlin, Hao Cheng, Hao Fang, Saurabh Gupta, Li Deng, Xiaodong He, Geoffrey Zweig, and Margaret Mitchell. Language models for image captioning: The quirks and what works. *CoRR*, abs/1505.01809, 2015.

[7] Christopher P Diehl and Gert Cauwenberghs. Svm incremental learning, adaptation and optimization. In *Neural Networks, 2003. Proceedings of the International Joint Conference on*, volume 4, pages 2685–2690. IEEE, 2003.

[8] Gregory Ditzler, Robi Polikar, and Nitesh Chawla. An incremental learning algorithm for non-stationary environments and class imbalance. In *Pattern Recognition (ICPR), 2010 20th International Conference on*, pages 2997–3000. IEEE, 2010.

[9] Pedro Domingos and Geoff Hulten. Mining high-speed data streams. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 71–80. ACM, 2000.

[10] Ryan Elwell and Robi Polikar. Incremental learning in nonstationary environments with controlled forgetting. In *Neural Networks, 2009. IJCNN 2009. International Joint Conference on*, pages 771–778. IEEE, 2009.

[11] Ryan Elwell and Robi Polikar. Incremental learning of concept drift in nonstationary environments. *IEEE Transactions on Neural Networks*, 22(10):1517–1531, 2011.

[12] Yoav Freund and Robert E Schapire. A desicion-theoretic generalization of on-line learning and an application to boosting. In *European conference on computational learning theory*, pages 23–37. Springer, 1995.

[13] Honorius Gâlmeanu and Răzvan Andonie. Implementation issues of an incremental and decremental svm. *Artificial Neural Networks-ICANN 2008*, pages 325–335, 2008.

[14] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *Acoustics, speech and signal processing (icassp), 2013 ieee international conference on*, pages 6645–6649. IEEE, 2013.

[15] Stephen Grossberg. Nonlinear neural networks: Principles, mechanisms, and architectures. *Neural networks*, 1(1):17–61, 1988.

[16] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[17] T Ryan Hoens, Robi Polikar, and Nitesh V Chawla. Learning from streaming data with concept drift and imbalance: an overview. *Progress in Artificial Intelligence*, 1(1):89–101, 2012.

[18] Geoff Hulten, Laurie Spencer, and Pedro Domingos. Mining time-changing data streams. In *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 97–106. ACM, 2001.

[19] Mark G Kelly, David J Hand, and Niall M Adams. The impact of changing populations on classifier performance. In *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 367–371. ACM, 1999.

[20] Ralf Klinkenberg and Thorsten Joachims. Detecting concept drift with support vector machines. In *ICML*, pages 487–494, 2000.

[21] James Mercer. Functions of positive and negative type, and their connection with the theory of integral equations. *Philosophical transactions of the royal society of London. Series A, containing papers of a mathematical or physical character*, 209:415–446, 1909.

[22] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[23] Michael D Muhlbaier, Apostolos Topalis, and Robi Polikar. Learn++.NC: Combining ensemble of classifiers with dynamically weighted consult-and-vote for efficient incremental learning of new classes. *IEEE transactions on neural networks*, 20(1):152–168, 2009.

[24] Robi Polikar, Lalita Upda, Satish S Upda, and Vasant Honavar. Learn++: An incremental learning algorithm for supervised neural networks. *IEEE transactions on systems, man, and cybernetics, part C (applications and reviews)*, 31(4):497–508, 2001.

[25] Liva Ralaivola and Florence d'Alché Buc. Incremental support vector machine learning: A local approach. *Artificial Neural Networks—ICANN 2001*, pages 322–330, 2001.

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

[27] Robert E Schapire. Theoretical views of boosting and applications. In *International Conference on Algorithmic Learning Theory*, pages 13–25. Springer, 1999.

[28] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 3104–3112. Curran Associates, Inc., 2014.

[29] Nadeem Ahmed Syed, Syed Huan, Liu Kah, and Kay Sung. Incremental learning with support vector machines. 1999.