Online to Batch Conversions

Mika Huttunen

October 29, 2017

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta/Osasto – Fakultet/Sektion – Faculty/Section	Laitos – Institution – Department
Faculty of Science	Department of Computer Science

Tekijä – Författare – Author
Mika Huttunen

Työn nimi – Arbetets titel – Title
Online to Batch Conversions

Oppiaine – Läroämne – Subject
Computer Science

Työn laji – Arbetets art – Level	Aika – Datum – Month and year	Sivumäärä – Sidoantal – Number of pages
M. Sc. Thesis	29.10.2017	56

Online to batch conversions have been studied by only a handful of people. Online learning algorithms are usually efficient in time and memory usage and they are easy to implement. Batch learning algorithms, on the other hand, are more complex both in resource requirements and implementation. It turns out that there are ways to convert the hypothesis sequence made by online algorithm into a single batch hypothesis that can be used on batch problems.

The basic idea of online to batch conversions is as follows. First, we get some samples of the data i.e. instances and their respective targets. These samples are called the training set. Then, we consider the training set as an ordered sequence and give it to some online algorithm A that goes through data one element at the time. A returns a sequence of hypotheses and we perform the conversion. We study both *single methods*, where we pick one hypothesis from this sequence and use it as a batch hypothesis, and *ensemble methods*, where we combine a subset of the hypothesis sequence and use all the hypotheses in this set to construct the batch hypothesis.

In this thesis we introduce most of the known conversions. The greatest emphasis is on *data-driven* conversions, where we assume that some of the hypotheses generated by A are better than others, and try to extract those to generate the batch hypothesis. For data-driven conversions we prove a connection between risk bounds in batch setting and loss bounds of the conversion techniques. These conversions are also evaluated using well-known test data, MNIST.

Studying 7 conversions, we reproduce some previous results. We evaluate each conversion with various partitions of a single data set to the training and test sets. The conversions based on single methods are mainly fast, but not as accurate as the slower conversions based on ensemble methods. Most of these conversions are more accurate than the common practice of picking the last hypothesis of the sequence.

There seems to be lot to study in online to batch conversions. Some of the best performing conversions are rather slow, even quadratic with respect to the size of the hypothesis sequence. Most of the time is spent on choosing an optimal subset of the hypothesis sequence. One could study heuristics for picking this set. Optimizing the trade-off between accuracy and the time spent is another problem left unanswered.

Avainsanat – Nyckelord – Keywords
Machine Learning

Säilytyspaikka – Förvaringställe – Where deposited
Kumpula Science Library, serial number C-

Muita tietoja – Övriga uppgifter – Additional information

Contents

1 Introduction **1**

2 Machine Learning **4**
2.1 Basic Concepts in Machine Learning 6
2.2 Batch Learning 7
2.3 Online Learning 8
2.4 Perceptron Algorithm 9
2.5 Support Vector Machine 11
2.6 Passive-Aggressive Algorithm 11
2.7 Conversions 13
2.8 Loss Functions 14
2.9 Risk and Regret 15
2.10 Practical Issues in Machine Learning 16

3 Conversion Algorithms **19**
3.1 Data-Independent Conversions 19
3.2 Data-Driven Conversions 21
3.3 Finding a Good Hypothesis Set 22

4 Martingales **26**

5 Data-Driven Conversion Algorithms **30**
5.1 Suffix Conversion 30
5.2 Interval Conversion 30
5.3 Tree-Based Conversion 31
5.4 Cutoff-Averaging 32
5.5 Wrap-up 36

6 Experiments **37**
6.1 Previous Experiments by Dekel and Singer 38
6.2 Results 39

7 Conclusion **48**

8 Appendix **53**

1 Introduction

The idea of machine learning is to develop a procedure that can automatically learn and improve its performance. This is done by finding patterns and making wise decisions based on the given data. Algorithms are often based on fundamental statistical principles. In this thesis, we will consider two learning frameworks: *online* and *batch learning*, and the connection between them.

Online learning algorithms are usually efficient in time and memory usage and they are easy to implement. The data is presented to the online learning algorithm as a sequence, and often no assumption of any data-generating distribution is made. The algorithm updates a built-in predictor after each data point. In this thesis, we are interested of that predictor sequence, also called *hypothesis sequence*.

Batch learning algorithms, on the other hand, are more complex both in resource requirements and implementation. In the batch learning setting, we often assume the data is independently and identically drawn (*i.i.d.*) from some distribution, and we try to learn the data generating rule by analyzing given data. The goal is to find a hypothesis that minimizes the *expected loss*, also called *risk*.

In the online setting we cannot define the risk of a hypothesis, since we don't make any assumptions of a data generating distribution. We use *cumulative loss* instead, and compare our performance to the best performing hypothesis, if we had picked it in hindsight.

It turns out that there are ways to convert the hypothesis sequence made by online algorithm into a single batch hypothesis that can be used on batch problems. There is also a connection between risk bounds in batch setting and loss bounds of the conversion techniques that can theoretically justify these techniques.

The last hypothesis of the hypothesis sequence generated by online algorithm is often chosen as the batch hypothesis. This makes sense, as the predictor tends to improve its performance over time. Nevertheless, using the last hypothesis of the sequence does not give an optimal result. There are more sophisticated and reliable ways to do a conversion.

Online-to-Batch conversion was pioneered by Stephen Gallant in 1986 [16] and Nick Littlestone in 1989 [26]. The conversion by Gallant is known as the *Pocket algorithm* and it returns the hypothesis that made the most correct predictions. He used the same idea later in 1990 to the *Perceptron algorithm* to develop the famous Pocket algorithm with ratchet [17].

The idea of Littlestone is as follows: after the online algorithm has generated the sequence of hypotheses, the conversion is made by choosing the

hypothesis among them that performs the best in a subsequent hypothesis testing phase. These two are examples of so called *single hypothesis conversions*.

Some time later, new techniques were developed. These techniques form the batch hypothesis by using all of the hypotheses of the sequence, and they are called *ensemble conversions*. We introduce three ensemble conversions: Sampling conversion by Helmbold and Warmuth in 1995 [19], Voting conversion by Freund and Shapire in 1999 [15] and Averaging conversion by Cesa-Bianchi, Conconi and Gentile in 2004 [5].

The Sampling conversion is most general of these, and it can be used in any input space. After generating the sequence of hypotheses we simply pick one hypothesis randomly from this sequence for each instance individually and use it as our predictor. The randomness of this conversion may give poor predictions at times, but the performance averages well.

Voting conversion technique assumes the output space is discrete. After the online algorithm has generated a sequence of hypotheses from the training set, all those hypotheses give a vote for an unseen data and we pick the label that receives the highest number of votes. The conversion is reliable and accurate, but the downside is that we need to keep the whole hypothesis sequence in memory.

In Averaging conversion, the output space is assumed to be *convex*. Using the sequence of hypotheses, the prediction is made by taking the average over predictions given by all the hypotheses. This conversion works especially well with linear hypotheses and does not utilize as much memory as voting conversion.

None of these conversions use the actual data when constructing the conversion, only the hypothesis sequence produced by the online algorithm. Hence they are called *data-independent conversions*.

Data-driven conversion techniques were introduced by Ofer Dekel and Yoram Singer in 2005 [12]. They consider three different conversions: Suffix conversion, Interval conversion and Tree-based conversion. In all these conversion techniques, additional information of the data is used in finding the optimal batch hypothesis. The basic idea of all these conversions is to find an optimal subset of hypotheses from the hypothesis sequence and then use the conversions above to the hypotheses in this subset.

These conversions yield more accurate results than the data-independent ones. The drawback is that some of these conversions are rather slow and they utilize plenty of memory, so they need to be tuned to be usable in big datasets.

A few years later, Dekel introduced a new data-driven conversion technique called Cutoff-Averaging [11]. This conversion is based on the Pocket

algorithm, where the best hypothesis so far is kept on memory until we find a better one. In Cutoff-Averaging, we are interested in hypotheses that have predicted correctly for some predefined amount of times. We do not have to keep them all in memory, as we can calculate the averages on the fly.

The conversion does not require to see the data again when choosing the hypotheses since that is done on the go. This is a key aspect in making the algorithm faster, as going through the data is usually slow. The Cutoff-Averaging conversion performs usually better than the aforementioned ones and it is relatively fast, thus capturing the idea of online to batch conversions.

2 Machine Learning

One of the earliest definitions of machine learning is by Arthur Samuel in 1959: "Field of study that gives computers the ability to learn without being explicitly programmed". In his article *Some studies in machine learning using the game of checkers* [29] he phrase it as: "Programming computers to learn from experience should eventually eliminate the need for much of this detailed programming effort."

In traditional programming, we have a data set and a program, which is run on a computer to produce an output. In machine learning, we change this order: we have the data and the output, from which a program is produced. Essentially, machine learning means what the name suggests: the machine learns from the given data.

The ability to learn without being explicitly programmed has become the key to handle huge data masses collected everyday. This is why machine learning is becoming more and more important and utilized in all areas.

The field is nowadays very wide and expands all the time. We will give a simple example why we want computers to learn by themselves.

Let us consider the game backgammon, where machine learning algorithms have surpassed humans long time ago as discussed in the paper by Tesauro in 2002 [33]. It all began in 1992, when Gerald Tesauro developed the program called TD-Gammon, which became widely known when he published the article *TD-Gammon, A Self-Teaching Backgammon Program, Achieves Master-Level Play* in 1994 [31].

TD-Gammon is based on a *neural network* that teach itself by playing games of backgammon against itself. Then it learns from the outcomes of the games. Even back in 1992, with randomized initial weights and no knowledge built in at the start of the learning, the TD-Gammon was able to achieve a strong intermediate level in backgammon. When it is provided with a set of features that are hand-crafted by humans it was able to play at a strong master level, which is almost on par to the best human players.

This algorithm is an example of *reinforcement learning*, where the algorithm receives less information in the training process than in the *supervised* setting. The algorithm is only given a *reward* (often called *reinforcement*) that is often delayed, after a sequence of predictions. We will not discuss reinforcement learning further in this thesis, but rather focus on supervised learning.

For many traditional board game there is a machine learning algorithm that can challenge the human masters. In the complex game of chess the computer *Deep Blue* was able to win the reigning champion Garry Kasparov in 1997 [4]. This was a project of many years that ultimately led to the

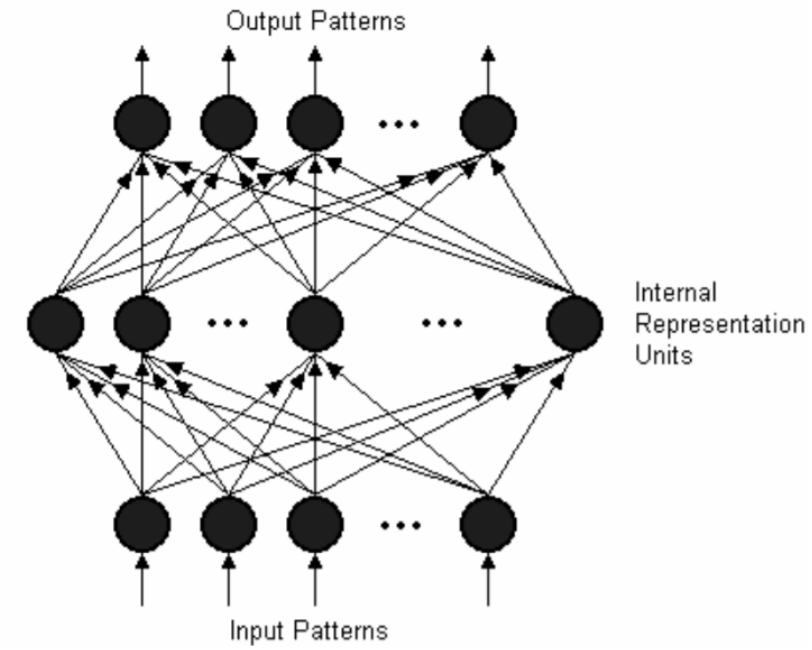

Figure 1: Multilayer architecture used in the TD-Gammon neural network. The figure is adapted from Gerald Tesauros article published in 1995 [32].

victory.

Developing a clever algorithm or a whole machine like Deep Blue to play chess is not an easy task. Big number of different moves on each rounds and their outcomes make the game tree large very fast. Therefore it is hard to develop an intelligent algorithm that uses *explicit* rules for playing chess.

It turns out that we actually don't have to give the algorithm explicit rules what to do in each possible situation, we can let the algorithm learn that by itself, like in backgammon. The algorithm can learn good moves and game strategies from a data set of games and their outcomes.

An example of this kind of learning is the chess engine called *Giraffe* by Matthew Lai [22]. This engine was made by a computer science student and it has proved to be very successful and tested in international chess tournaments. The algorithm also relies on neural networks and learned to play chess all by itself at a high international level.

In addition to popular games, machine learning is used in many tasks and fields such as face recognition [36], stopping malware [27], blocking spam messages [18], preventing money laundering [30] and preventing credit card frauds [6].

One popular field is text analysis, where machines try to learn what the

given text means. For example the machine can be provided with the text of a news article and categorize it automatically. One such machine learning algorithm is the fastText by Joulin *et al.*[21].

Machine learning is really everywhere as the amount of available data is expanding. Whenever we have huge amounts of data or a need for something that is too hard for a human programmer to code explicit rules, machine learning is used to tackle the problem.

2.1 Basic Concepts in Machine Learning

Machine learning is a wide area of study and there are many ways to do it. It is closely related to computational statistics, mathematical optimization and data mining. In this chapter, we will go shortly through some very basic concepts of machine learning.

Everything begins with data. Any prior knowledge we have of the data will affect what we can do with it. Having some insight of the domain of the problem can help with designing the machine learning algorithm to solve the problem. In practice, the domain knowledge is combined with the statistical principles to achieve the optimal result.

If the data includes the desired output, we can try to learn the rule that generates the output from the data. This kind of learning is called supervised. If we have just the data without any output, we have a more restricted setting. In this setting we can try to learn the structure of the data or relationships between different inputs. This kind of learning is called unsupervised. A well known problem in unsupervised setting is clustering, in which we generate clusters where each input is then assigned. In this thesis we focus on the supervised setting.

The next question is: do we have access to some batch of the data, or is the data fed to the algorithm one example at the time. The former is called *batch learning* and the latter *online learning*. First we introduce the basic components of both learning methods and then we will go through the differences.

The basic components for both online and batch learning models are:

- *Input space* X, whose elements $x \in X$ are called *instances*
- *Output space* Y, whose elements $y \in Y$ are called *response*, or in the case where Y is discrete, *labels*
- *Target function* $f(x) = y$ is some pattern between the input and output values. It is reasonable to assume that there is some kind of functional and deterministic relationship between the two

- *Hypotheses*, a family of functions $H \subset \{h : X \to Y\}$. The target function that generates the outputs is unknown, so we try to find a function that approximates it.

The goal is to find a hypothesis h that makes as few mistakes as possible when predicting the target y of a given unseen data element x, *i.e.* it is as close to the target function as possible.

Example. Linear classifiers. Here is a concrete example of the listed components above. Consider the case where $X \subset \mathbb{R}^n$ and $Y = \{-1, 1\}$. Let $H = \{h^w : X \to Y : h^w(x) = sign(w \cdot x)$ and $w \in \mathbb{R}^n\}$. Here, w is called a *weight vector* and $sign(z)$ is -1 if $z < 0$ and otherwise 1.

Now the data is a vector in $\mathbb{R}^n$ and w is a weight vector in the same space. The prediction generated by h^w is the sign of the dot product of the weight vector and a given data element. The learning part is then to *update* this weight vector according to the result of the prediction made by it.

In batch learning, the accuracy of the learner is evaluated after *all* the training examples have been seen. In online learning, the performance is evaluated during the process, and is called *regret*. Any online algorithm with small regret can be converted into a batch algorithm with good accuracy. This will be proved in Section 3 and it justifies the conversions theoretically.

Example. We have 500 distinct pictures of handwritten digits 1 and 2. Each number is written by a different person, and the appearance of the numbers vary. Our goal is to use these 500 pictures to learn an algorithm that can distinguish number one from number two written by any person in the world.

The set of pictures is the input space X and each picture is an instance $x \in X$. A label associated with each picture is either one or two, so $Y = \{1, 2\}$. Hypothesis h is now a function trained by some algorithm that will predict either one or two when it receives any picture from X. Note that the set X can contain pictures that the algorithm has never seen before, not just the 500 used in training.

2.2 Batch Learning

The basic idea of batch learning is that the algorithm receives a set of m pairs $(x_i, f(x_i)) = (x_i, y_i) \in X \times Y$, $i = 1, 2, \ldots, m$. The examples (x_i, y_i) are assumed to be drawn independently from distribution D. Then, the algorithm generates a hypothesis $h \in H$ by analyzing the m examples. This is called the *training phase*. After training, the hypothesis h is used to

unseen data and it should give rather good results, as we assume that all the examples and unseen data are drawn from the same distribution.

The ultimate goal is to find a hypothesis that generalizes well and gives reliable prediction on unseen data that is from the same distribution as the training data. To achieve this, the hypothesis has to perform well for the training data while not being too specific to that data alone. After all, it is just a sample of the data and may contain bias and noise.

Example. We continue the previous example with the handwritten numbers one and two. The algorithm used in this example is called the k-nearest neighbours.

The idea is simple: we already have 500 examples of pictures with their true labels. When we obtain a new sample we will compare it to all other samples and find k samples that are closest to it. Then we check the majority in these k samples and predict the label of the new example with the majority vote.

One question arises: how do we define what 'close' means? This has to be defined with machine learning algorithms and there is no single simple answer. The handwritten pictures are represented by matrices with numbers 0 to 255 corresponding to each pixel's darkness, we can for example use the Frobenius norm (Euclidean norm of matrices) as the closeness measure. This approach has many flaws but it works fairly well on this setting.

In general, the most suitable measure for closeness is often derived by trial and error and it is one of the key elements of a machine learning algorithm. Minimizing or maximizing this measure is an essential step in many machine learning algorithms.

2.3 Online Learning

In this setting, the learning is done as a sequence of trials. On any round t the online learning algorithm receives $x_t \in X$ and predicts the target value by using hypothesis h_{t-1}, which was generated at the previous round. The initial hypothesis h_0 is arbitrary. Next, the correct label $y_t \in Y$ is revealed and the algorithm updates the hypothesis h_{t-1} into h_t. Note that in online setting we usually do not make any assumptions of how the data was generated. After k iterations, we have obtained a sequence of hypotheses $h_0, h_1, \ldots, h_{k-2}, h_{k-1}$. These hypotheses are not necessarily distinct.

This has many advantages in practice. First of all, the examples can be processed one by one, which means that we do not have to save the training examples in memory, thus reducing the need for memory. The algorithm is

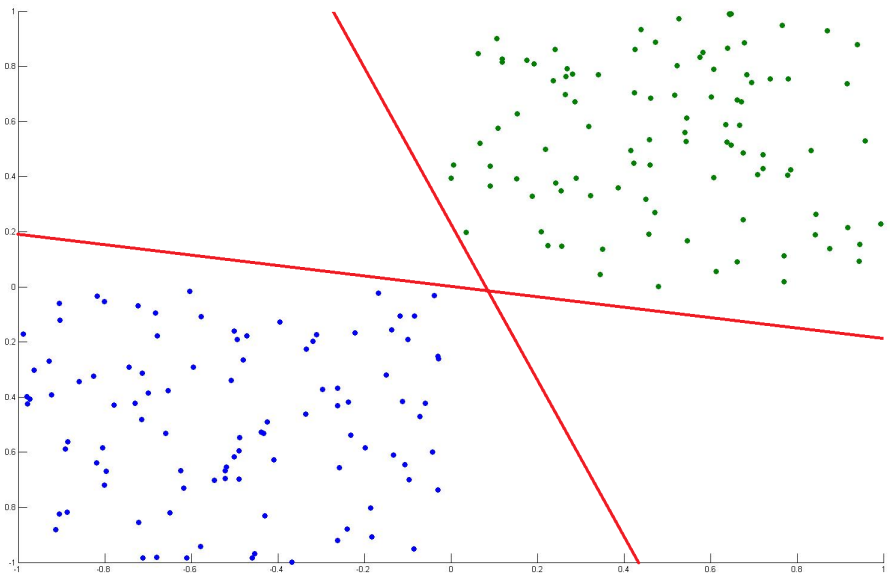

Figure 2: Example of a linearly separable data set and two linear boundaries. There are infinitely many boundaries that separate the sets from each other and the Perceptron algorithm might converge to any of them.

also faster as we do not run many iterations over the training set.

Both online and batch algorithms are fairly common in many fields. In the next chapter, we will give a few concrete examples of online and batch algorithms. These algorithms are well-known and are often used as a reference for new machine learning algorithms. We start with the famous *Perceptron* algorithm.

2.4 Perceptron Algorithm

Introduced by Rosenblatt [28] in 1958, the Perceptron algorithm is one of the best known online algorithms. It is easy to implement, fast and gives fairly good results in many situations.

The original algorithm is intended to be used when each data point belong to one of two classes and the classes are linearly separable. This means that if the data is n-dimensional, we can separate the two sets from each other with an $n-1$-dimensional hyperplane. After a finite number of iterations the algorithm will converge to a set of weights that classifies each example correctly.

If the data is not linearly separable, the Perceptron algorithm does not converge to any set of weights. There are many ways to tune the algorithm

Initialise $\mathbf{w}_1 \leftarrow \mathbf{0}$.
Until all predictions are correct do
    For $t = 1, \ldots, n$ do
        receive $\mathbf{x}_t \in \mathbb{R}^m$,
        predict $y'_t = sign(\mathbf{w}_t \cdot \mathbf{x}_t) \in \{-1, 1\}$,
        receive the correct answer $y_t \in \{-1, 1\}$,
        if $y'_t \neq y_t$ do
            set $\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t + y_t \mathbf{x}_t$.
        end
    end
end

Algorithm 1: The Perceptron algorithm by Rosenblatt [28]. The algorithm is *conservative* as it does not update the weight vector when the prediction is correct.

to stop at a reasonably good solution. One such way is called the Pocket algorithm by Gallant [17], [16] that stores the best weight vectors so far. So even though we cannot separate the data linearly, we can try to find the best separation to get the largest amount of correct classifications.

The idea of the Perceptron algorithm is as follows. We assume that the classes used are $Y = \{-1, 1\}$. The algorithm starts with an initial prediction vector $w = 0$. Then it predicts the label of a new instance x to be $y' = sign(w \cdot x)$. After the prediction, the true label y is revealed to the algorithm. If the prediction was right, we will continue to the next instance without changes. If the prediction differs from the true label y, then the prediction vector w is updated to be $w = w + yx$. This means that the hyperplane that eventually separates the two classes moves after every mistake.

This process is repeated until the prediction vector converges to a vector that linearly separates the classes. The pseudo-code of the Perceptron algorithm is found in Algorithm 1.

There are ways to convert Perceptron to multiclass problems. Easiest to implement are methods called *one vs. all* and *all vs. all*. Both methods have their pros and cons and it is not always clear which of them one should use. In one vs. all we first create a two-class classification problem for each class y and then we predict if our data sample belongs to that class or not. In all vs. all we make a modified training set for each pair of classes. This means that for a multiclass problem with k classes we have to train $\sum_{i=1}^{k-1} i = k(k-1)/2$ classifiers.

Despite the fact that the algorithm is almost 60 years old, it serves as a base for many machine learning algorithms used today. It is used for example in natural language processing [8].

2.5 Support Vector Machine

Support Vector Machine (SVM) is one of the best known batch learning algorithms. We again consider a classification problem with two classes that are linearly separable. We can find infinitely many decision boundaries, so which one should we choose? The answer is the one that is as far as possible from the both classes. This gap between two classes is called the margin (m) and we want to maximize it.

This becomes a constrained optimization problem that can be solved with a Support Vector Machine (SVM, originally invented by Chervonenkis and Vapnik in 1964 [35], and popularized in the 1990s by Boser, Guyon and Vapnik [3] and by Cortes and Vapnik [9]).

To put it short, the idea of SVM is to first map the instances to a very high-dimensional space, where the two classes become linearly separable with large margin. Note that this is always possible as we can add as many dimensions as needed. Next, we use a special method called *quadratic programming* to find a vector that classifies all the data without mistakes with a maximum margin. This separator is a hyperplane in a high-dimensional space.

The support vector machines are a good example of so called *kernel methods*. The idea of a kernel method is to operate in a high-dimensional space with *kernel functions*. A kernel function reduces the calculation of coordinates of the data to calculating the inner products of *feature maps*. This technique is called the *kernel trick*.

By this remapping we can transform a non-linearly separable data to be linearly separable in a high dimensional space. The computation of feature mappings themselves can be very expensive, but with kernel functions it is enough to calculate only the inner products of the feature map, and producing the output in an inner product space.

More thorough explanation of SVMs can be found in the article by Cortes and Vapnik [9].

2.6 Passive-Aggressive Algorithm

The next online algorithm we are going to introduce is the Passive-Aggressive Algorithm (PA), developed by Crammer *et al.*[10]. It is a generalization of the SVM to the online setting. The PA algorithm is used in the original

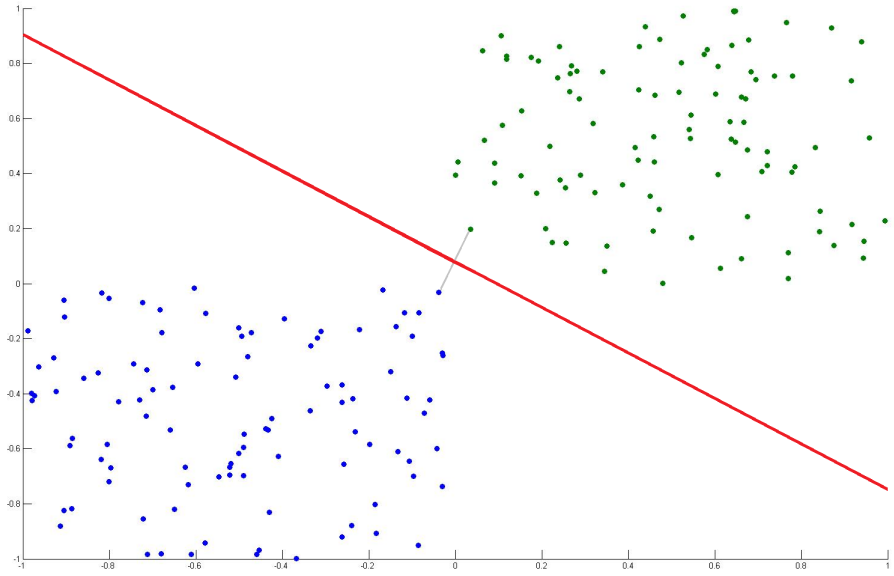

Figure 3: The SVM separates the two classes of the data set with maximal margin. The data sets are the same as in Figure 1.

data-driven conversion paper by Dekel and Singer [12] for experiments on their conversions.

We will only discuss the binary classification case. The idea is to have a weight vector that is updated according to the seen data. The weight vector is initialized to a zero vector and on round t the new weight vector w_{t+1} is the solution to the following constrained optimization problem:

$$w_{t+1} = \underset{w \in \mathbb{R}^n}{\operatorname{argmin}} \frac{1}{2}\|w - w_t\|^2 \quad \text{s.t.} \quad l(w; (x_t, y_t)) = 0. \tag{1}$$

Loss function used is the *Hinge loss* defined in Section 2.8 The algorithm is *passive* whenever the Hinge loss is zero: we do not change the weight vector. On the other hand, the algorithm is *aggressive* whenever the loss is positive, as it forces the new weight vector to satisfy the constraint $l(w; (x_t, y_t)) = 0$. Therefore the algorithm is called *Passive-Aggressive* (*PA*). The pseudo-code of the algorithm is presented in Algorithm 2.

The idea of the update rule originates from Helmbold *et al.* [20]. The update requires the current weight vector w_{t+1} to predict with high margin while staying as close to w_t as possible. This way the algorithm makes confident decision while keeping as much information learned from previous rounds as possible.

Initialize: $\mathbf{w}_1 = (0, \ldots, 0)$
For $t = 1, 2, \ldots$ do
    receive instance $\mathbf{x}_t \in \mathbb{R}^n$
    predict $y'_t = sign(w_t \cdot x_t)$
    receive correct label $y_t \in \{-1, 1\}$
    suffer loss $l_t = max\{0, 1 - y_t(w_t \cdot x_t)\}$
    update $w_{t+1} = w_t + \frac{l_t}{\|x_t\|^2} y_t x_t$
end

Algorithm 2: The passive-aggressive algorithm. The algorithm is passive whenever the loss is zero, but it aggressively updates the weight vector whenever a mistake occurs.

Equation (1) has a closed form solution

$$w_{t+1} = w_t + \frac{l_t}{\|x_t\|^2} y_t x_t.$$

We will not go through the derivation, which can be found in [10].

There are many versions of this algorithm with a different weight update strategy. Some modification is needed for a multiclass case. We will not, however, go more in to details of online passive-aggressive algorithms. A detailed explanation of different variants of the above presented can be found from the article *Online Passive-Aggressive Algorithms* [10].

2.7 Conversions

Now we are ready to describe conversions of the hypothesis sequence constructed by an online algorithm into a single batch hypothesis. Conversion is not a single method, there are many different conversions to be used, all with their pros and cons.

The basic idea of online to batch conversions is as follows. First, we get some samples of the data *i.e.* instances and their respective targets. These samples are called the *training set*. Then, we consider the training set as an ordered sequence and give it to some online algorithm A that goes through data one element at the time. A returns a sequence of hypotheses H_i and we perform the conversion.

The goal is to find a good subset of hypotheses from the sequence A produced and use only those hypotheses to generate the batch hypothesis. The batch hypothesis in this thesis will usually be a collection of the best

performing hypotheses of the hypothesis sequence.

One of the oldest conversions is to pick the last hypothesis and use it as a batch hypothesis. The technique makes sense as the hypothesis sequence tends to converge to a reasonably good hypothesis and sometimes this is the best possible result, for example when the Perceptron algorithm is used on linearly separable data.

This method can also result in a bad batch hypothesis. If the end of the data sequence seen by A contains bad examples or complete outliers, the end of the hypothesis sequence generated by A may be worse than the hypotheses before the outliers. We will discuss this problem and define more reliable conversions in Chapter 3.

2.8 Loss Functions

We use loss functions to measure the performance of a hypothesis. We denote the loss function by l. The value $l(y, y')$ measures the penalty suffered for predicting y' when correct target was y.

Choosing a suitable loss function for a problem is important. In some learning problems, there is no distinction between different incorrect predictions. In this case our loss function should always punish the hypothesis by same amount of loss when it makes a mistake, no matter what the prediction was. The loss function called *0-1-loss* fits this purpose well.

In general, this is not the case, and we need to use more complex loss functions. For example, we can learn a spam filter using machine learning algorithms. Putting an important message to spam folder is a worse mistake than letting a spam message through and thus the loss function should punish the algorithm more for filtering real messages out than for letting spam messages through.

The loss functions used in this thesis are:

0-1 -loss:

$$l(y, y') = \begin{cases} 1 & \text{if } y \neq y' \\ 0 & \text{otherwise} \end{cases}$$

Absolute loss:

$$l(y, y') = |y - y'|$$

Square loss:

$$l(y, y') = (y - y')^2$$

Logarithmic loss: for $y \in \{0, 1\}$ and $0 \leq y' \leq 1$,

$$l(y, y') = \begin{cases} -\ln(1 - y') & \text{if } y = 0 \\ -\ln(y') & \text{otherwise} \end{cases}$$

Note that this is not defined if either $y = 0$ and $y' = 1$ or $y = 1$ and $y' = 0$.

Hinge loss:
for any margin parameter $\mu \geq 0$ and weight vector w:

$$l(w, (x, y)) = \begin{cases} 0 & \text{if } y(w \cdot x) \geq \mu \\ \mu - y(w \cdot x) & \text{otherwise} \end{cases}$$

The common use of this loss function is with support vector machine.

The right loss function is not always obvious for a given problem. Machine learning tasks often include a metric we want to optimize and we need to choose the loss function carefully to succeed in this. Finding a good loss function might get tricky and may require some background knowledge of the data.

2.9 Risk and Regret

The risk and regret are used in batch and online setting to measure how well a given hypothesis performs on a given task.

In the batch learning setup, we assume there exists a probability distribution D over the product space $X \times Y$. Thus, we can define the average performance of a hypothesis h over the entire domain, called the *risk*

$$R(h) = E_{(x,y)\sim D}[l(y, h(x))].$$

This is the expected loss of a hypothesis h with respect to the loss function l, when a sample (x, y) is independently and identically distributed (*i.i.d.*) and drawn from the distribution D. Naturally, we want to find a hypothesis that minimizes the risk.

In the online setting, we cannot define the risk because we often do not make any assumptions of a data generating distribution. Therefore, we measure the performance of an algorithm by *regret*. It is the difference between the losses of our hypotheses and the best possible hypothesis in H on the given example sequence. This can be measured only after all the examples

Representation	Evaluation	Optimization
Instances	Accuracy/Error rate	Combinatorial optimization
K-nearest neighbor	Precision and recall	Greedy search
Support vector machines	Squared error	Beam search
Hyperplanes	Likelihood	Branch-and-bound
Naive Bayes	Posterior probability	Continuous optimization
Logistic regression	Information gain	Unconstrained
Decision trees	K-L divergence	Gradient descent
Sets of rules	Cost/Utility	Conjugate gradient
Propositional rules	Margin	Quasi-Newton methods
Logic programs		Constrained
Neural networks		Linear programming
Graphical models		Quadratic programming
Bayesian networks		
Conditional random fields		

Figure 4: The three components of learning algorithms. This figure is adapted from Pedro Domingos [13].

have been seen, because only then we can know for sure which hypothesis performed the best.

We compare the performance of our hypothesis sequence to the best performing *single* hypothesis. This is different from minimizing the accumulated loss. The reason is that in online learning the target function is not fixed, as the length of the data stream is not known. In online learning we can never be sure what happens next, whatever we have learned now might be obsolete in future.

Mathematically, the regret of an online algorithm is

$$\frac{1}{n}\sum_{t=1}^{n} l(h_{t-1}, (x_t, y_t)) - \min_{h' \in H} \frac{1}{n}\sum_{t=1}^{n} l(h', (x_t, y_t)).$$

We want to make the regret of an online algorithm as small as possible. It turns out that by minimizing the regret of an online algorithm we can have a loss bound for the risk of the batch algorithm generated by the following online to batch conversions.

2.10 Practical Issues in Machine Learning

The ability to generalize beyond training data examples is the key in machine learning. It is all about statistics and learning the data generating distribution. We will next go through a few practical issues that arise when designing and implementing machine learning algorithms.

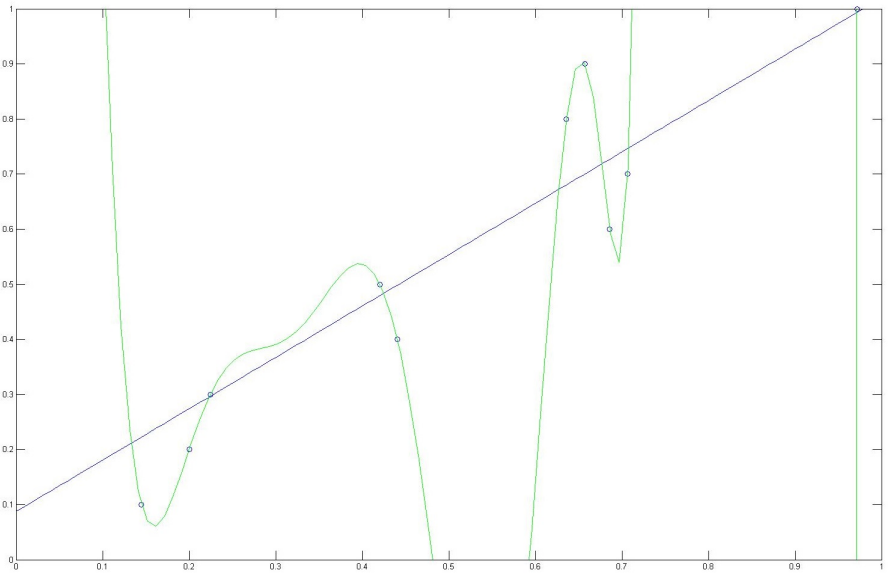

Figure 5: An example of overfitting. The datapoints (circles) are values of a linear function with some added noise. A first degree polynomial (blue line) has larger training error than the 10-degree polynomial (green line), but the simpler predictor will generalize better.

Most of this chapter is adapted from the article by Pedro Domingos: *A few useful things to know about machine learning* [13]. The first problem with machine learning is to choose a suitable algorithm for a given problem. As there are thousands of algorithms, this is not a trivial task. Domingos give few hints how to start this search.

He states that learning is equal to representation, evaluation and optimization: a classifier must be represented in some formal language a computer can understand. Choosing the representation of a learner defines the *hypothesis space*, from which a classifier can be learned. With evaluation we mean a scoring function that help us distinguish good classifiers from bad. The last part is to optimize the algorithm so that it can efficiently traverse the hypothesis space to find the best classifier. In Figure 4 are listed some examples for each of these components.

One of a more concrete problems is *overfitting*, which means that our algorithm learns the training set well, but does not generalize to fit the underlying distribution. For example a higher degree polynomial will always fit the data at least as well as a lower degree polynomial. This is seen in the Figure 5.

This problem is often tackled by folding the training set in k distinct

sets. One of these sets is left out of training the algorithm, and is used for validation. Then, we vary the training and validation sets, and finally pick the hypothesis that worked best out of all the different hypotheses we made during the verification phase.

Also a additional term called *regularization* is often added to the loss function. The regularization term punishes the more complex models and enables a trade-off between accuracy on the training set and model complexity. The term *Occam's razor* is often used with overfitting. It is a principle by William of Ockham from 14th century and it states that one should choose among competing models the model that has least assumptions.

The *curse of dimensionality*, an expression by Richard E. Bellman, means that many algorithms that perform well in low dimensions become useless in large dimensions. Also the intuition and the ability to visualize the data becomes harder when dimensionality of the data increases, often to hundreds or even thousands of dimensions. The common theme of these problems is that the *volume* of the space increases so fast that the available data becomes sparse.

Fortunately the examples in most applications are not spread uniformly in the high dimensional space, but rather occur in clusters on lower-dimensional subspaces.

The theoretical guarantees are not what they seem to be at the first glance. The main thing in machine learning is generalization and many published machine learning articles contain theorems that assure generalization of the algorithm, if we just have enough training data.

This is usually misleading, as first of all the bounds are usually very loose and probabilistic. Secondly, the hypothesis spaces tend to be large and it is not always true that a good approximation of the underlying function is found in this space and the theorem does not say how to find a good hypothesis space. What these bounds usually say is just that if we have large enough training corpus, we have high probability that the learner either returns a hypothesis that generalizes well, or fails to find one.

3 Conversion Algorithms

Finding a good hypothesis with a batch learning algorithm is usually time consuming. Programming takes time, and training and validation have to be done with care and precision. On the other hand, online algorithms and conversions are usually very fast to implement and run. As a result we obtain a batch hypothesis that performs well and it is more easily obtained than the ones generated by batch algorithms.

What choices do we have for conversion? The conversions are often divided to two classes: *single* and *ensemble methods*. In the single method, we pick one hypothesis from the hypothesis sequence. In ensemble method, we try to combine some set of the hypotheses in the hypothesis sequence to generate a batch hypothesis.

In this chapter, we will go through the basic ideas of online to batch conversions. We will start with the earliest and simplest conversions called data-independent conversions. The word "data-independent" is from Dekel and Singer [12]. They do not consider the data when constructing the batch hypothesis. Then, we will move to more powerful data-driven conversions that exploit also the given examples in generating the batch hypothesis.

3.1 Data-Independent Conversions

We will consider three different data-independent algorithms for the online-to-batch conversion: Voting, Averaging and Sampling conversions. The first two are ensemble methods and the Sampling conversion is a single method. They serve as a basis for data-driven conversions.

Voting Conversion

Voting conversion was developed by Freund and Schapire in 1999 [15]. It is a good example of an ensemble method.

We assume that the target space Y is discrete and we call $y \in Y$ a *labelling*. First some online algorithm A has generated a sequence of hypotheses $h_0, \ldots, h_n$ from the training set $(x_1, y_1), \ldots, (x_n, y_n)$. Given an input $x \in X$, each online hypothesis outputs a vote $h_i(x)$ and our batch hypothesis h^V outputs the class that received the most votes. If two or more classes receive the same number of votes, we pick one of them at random.

The Voting conversion gives fairly good results but it is very slow. Where it wins 'pick last hypothesis'-method in accuracy it loses in time consumption by big marginal. This is because with every unseen data element we have to go through our whole hypothesis sequence to collect all the votes before we

can output the prediction. The size of the hypothesis sequence depends on the size of the training set used and there is a trade-off between the accuracy of the predictor and the time consumed in predicting.

Back in 1999 the authors of this algorithm experimented it with the Perceptron algorithm against SVM on the task of hand-written digit recognition. The end result was that SVM gives slightly better results, but this conversion is easier to implement and it is much faster.

Averaging Conversion

The next one is called Averaging conversion by Cesa-Bianchi *et al.*published in 2004 [5]. It applies to problems where the target space Y is convex. Using the sequence of hypotheses generated by A we predict by taking the average over predictions given by all the hypotheses. Mathematically, our batch hypothesis h^A is defined as

$$h^A = \frac{1}{n+1} \sum_{i=0}^{n} h_i(x)$$

This conversion is also time consuming if the hypothesis sequence is long, as we have to calculate the prediction of every hypothesis in the sequence. Then again, the conversion is robust, as it is the average over all the hypotheses gathered in the training phase. This conversion is also used as a base to the more advanced conversion *Cutoff-Averaging* which we will go through in detail in Section 5.4.

Sampling Conversion

Sampling conversion published in 1995 by Helmbold and Warmuth [19] is the most general of these conversions: it applies to any learning problem. Generated batch hypothesis h^S is a stochastic function that is obtained by choosing for each unseen data element a random hypothesis h_i from the hypothesis sequence generated by A. This means that if we apply it twice to a same instance we might get a different result on each.

This conversion is very fast but it does not guarantee a good result on any given data element. On the other hand, it does not get stuck with a hypothesis that predicts often incorrectly. This is because we pick a different hypothesis after every prediction, so the ability to predict correctly averages over the data set.

In the experiments section we will see that this conversion does not provide as accurate results as the other conversions. It usually loses to the pick-last method except if the last hypothesis happens to be exceptionally bad.

The experiments section shows that this conversion should be used if we need a very fast conversion and we just cannot risk and pick the last hypothesis which can be very inaccurate even though the probability for generating one is relatively small.

The performance of this conversion improves by using the techniques introduced in the next section.

3.2 Data-Driven Conversions

This section follows Section 2 of the presentation of Dekel and Singer [12]. We use the three data-independent conversions introduced in Section 3.1. as a basis for the data-driven conversions. The main idea behind the data-driven conversions is to assume that some of the hypotheses generated by A are better than others. This is a reasonable assumption, because before seeing any data, our online algorithm can only guess.

After the first sample our algorithm can update the hypothesis only by using the information obtained from the first data element, hence the first part of the hypothesis sequence is generally worse than the last part. After an anomalous example or two, we might get into a wrong direction for a while, so there tends to be some bad hypotheses in the middle of the hypothesis sequence. We can try to separate the best hypotheses from the rest, and only use those to generate the batch hypothesis.

We start the search of the optimal set of hypotheses by introducing some notation. First, we denote the set $\{0, \ldots, n\}$ by $[n]$. Let $I \subseteq [n]$. Now we only use the hypotheses h_i where, $i \in I$, in Voting, Averaging and Sampling conversions. We want this set to contain only hypotheses that perform well. Since the set $[n]$ has an exponential amount of subsets, we will restrict our search of I to some family of subsets $\mathcal{I}$ of the set $[n]$.

We use Voting, Averaging and Sampling conversions as before, but only apply each to hypotheses $\{h_i : i \in I\}$.

The idea is to use the training data to find the set I that leads to batch hypotheses with smallest risk. Finding the optimal set plays a crucial role. There are two advantages in this approach. First, the hypotheses in this set I have better performance on average than those that are left out of this set. Secondly, this set is smaller than the whole hypothesis sequence, so for example Voting conversion is faster as there are not so many hypotheses voting in the first place.

Table 1: Conversions

Method	**Target space**	**Time complexity for one classification**	**Type**
Pick Last Hypothesis	Any	Constant	Single hypothesis
Voting Conversion	Discrete	Linear w.r.t. the size of hypothesis sequence	Ensemble
Averaging Conversion	Convex	Linear w.r.t. the size of hypothesis sequence	Ensemble
Sampling Conversion	Any	Constant	Single hypothesis
Data-Driven Voting	Discrete	Linear w.r.t the size of a subset of hypothesis sequence	Ensemble
Data-Driven Averaging	Convex	Linear w.r.t. the size of a subset of hypothesis sequence	Ensemble
Data-Driven Sampling	Any	Constant	Single hypothesis

Note that we get the original data-independent conversions by choosing $I = [n]$.

3.3 Finding a Good Hypothesis Set

How can we find a good set I? If the set contains only a few hypotheses, even one poorly predicting hypothesis in this set can have a devastating effect on the performance of the resulting batch hypothesis. This is why we want the set to be reasonably large to average out any biased hypotheses.

The hypotheses in the set have to predict well, so we try to pick only the hypotheses that made as few mistakes in the training phase as possible.

We start our search for a good set by defining for any set $J \subset [n-1]$ the *empirical loss*

$$L(J) = (1/|J|) \sum_{j \in J} l(y_{j+1}, h_j(x_{j+1})). \tag{2}$$

It is the average loss of each hypothesis h_j on the next example (x_{j+1}, y_{j+1}). We expect that a small empirical loss indicates a low batch risk, so we want to find a set J for which $L(J)$ is small. We also want that our set J is reasonably large so that a few bad hypotheses would not effect the resulting hypothesis too much. This trade-off can be formalized as follows. Let C be a positive constant. For a hypothesis set J we define

$$\beta(J) = L(J) + C|J|^{-\frac{1}{2}}. \tag{3}$$

The function β decreases when empirical risk decreases and also when the size of the set increases. Therefore, it is a good measure for our purposes. Now, we can set $I = arg\min_{J \subset [n-1]} \beta(J)$. This β-function and its minimization

is the main point in the article by Dekel and Singer and it is also used throughout this thesis.

We aim to theoretically justify the Voting, Averaging and Sampling conversion. The following lemma (Lemma 1 from Dekel and Singer [12]) relates the risk of h_J^V, h_J^A and h_J^S with the average risk of the hypotheses indexed by J.

Lemma 1. *Let $(x_1, y_1), \ldots, (x_m, y_m)$ be a sequence of examples which is presented to the online algorithm A and let $h_0, \ldots, h_m$ be the resulting sequence of online hypotheses. Let $J \subset [m-1]$ be non-empty and let $l : Y \times Y \to \mathbb{R}_+$ be a loss function.*

1. *If l is the $0-1$ loss then*
   $Risk_D(h_J^V) \leq (2/|J|) \sum_{i \in J} Risk_D(h_i(x))$.
2. *If l is convex in its second argument then*
   $Risk_D(h_J^A) \leq (1/|J|) \sum_{i \in J} Risk_D(h_i(x))$.
3. *For any hypothesis h_J it holds that*
   $Risk_D(h_J^S) \leq (1/|J|) \sum_{i \in J} Risk_D(h_i(x))$.

Note that we leave the last hypothesis of the sequence out of the set J. This leads to simpler proofs.

Proof. Recall that $Risk(h) = E_{(x,y)\sim D}[l(y, h(x))]$

1. Let l be $0-1$-loss. We have two cases; either we predict correctly and we suffer no loss or we predict wrong so $l(y, h_J^V(x)) = 1$. Let's consider first the case when we do not make a mistake. Then at least half of the hypotheses in $\{h_j\}_{j \in J}$ predict correctly, so also h_J^V predicts correctly, which means that $l(y, h_J^V(x)) = 0$.

   If we make a mistake, then at least half of the hypotheses in $\{h_j\}_{j \in J}$ make an incorrect prediction. Since $0-1$ loss calculates the number of mistakes, we get $(|J|/2) \leq \sum_{i \in J} l(y, h_i(x))$. We solve that

   $$l(y, h_J^V(x)) = 1 \leq \frac{2}{|J|} \sum_{i \in J} l(y, h_i(x))$$

   Now we can take expectation over the examples and use linearity of expectation to obtain

$$E[l(y, h_J^V(x))] \leq E\left[\frac{2}{|J|}\sum_{i\in J} l(y, h_i(x))\right] = \frac{2}{|J|}\sum_{i\in J} E[l(y, h_i(x))].$$

By the definition of Risk we get

$$Risk_D(h_J^V) \leq (2/|J|)\sum_{i\in J} Risk_D(h_i(x))$$

as was to be proved.

2. Let l be convex in its second argument. We use Jensen's inequality (Theorem 4, Appendix):

$$\begin{aligned}
Risk_D(h_J^A) &= E_{(x,y,)\sim D}[l(y, h_J^A(x))] \\
&= E_{(x,y,)\sim D}\left[l\left(y, \frac{1}{|J|}\sum_{i\in J} h_i(x)\right)\right] \\
&= E_{(x,y,)\sim D}\left[l\left(y, \sum_{i\in J} \frac{1}{|J|} h_i(x)\right)\right] \\
&\leq E_{(x,y,)\sim D}\left[\frac{1}{|J|}\sum_{i\in J} l(y, h_i(x))\right] \\
&= \frac{1}{|J|}\sum_{i\in J} E_{(x,y,)\sim D}[l(y, h_i(x)] \\
&= \frac{1}{|J|}\sum_{i\in J} Risk_D(h_i(x)).
\end{aligned}$$

3. Let l be any loss function. The hypothesis h_J^S chooses a random hypothesis from $\{h_i : i \in J\}$ and uses the selected hypothesis to output a prediction. The probability of choosing any given hypothesis from this set equals $1/|J|$. This means that the expected loss suffered by h_J^S on any example (x, y) is

$$(1/|J|)\sum_{i\in J} l(y, h_i(x)),$$

i.e. the average loss of all the hypotheses in J. Now,

$$
\begin{aligned}
Risk_D(h_J^S) &= E_{(x,y,)\sim D}[l(y, h_J^S(x))] \\
&= E_{(x,y,)\sim D}\frac{1}{|J|}\sum_{i\in J} l(y, h_i(x)) \\
&= \frac{1}{|J|}\sum_{i\in J} E_{(x,y,)\sim D}[l(y, h_i(x))] \\
&= \frac{1}{|J|}\sum_{i\in J} Risk_D(h_i(x)),
\end{aligned}
$$

which proves our claim. □

The above lemma means these three conversions give fairly good results in theory. This is verified in the experiments section and we will actually see that these upper bounds are loose. We will get back to this in Chapter 6.

Next, we need some definitions and theorems before we are ready to justify the choice of the β-function in equation (3). We begin by introducing a concept called *martingales*.

4 Martingales

Martingales arise naturally in the context of online to batch conversions, as we will see in lemma 2. First we introduce them and then we prove an important theorem by Azuma [1], which plays an essential role in proving the error bound of the following conversions.

Definition 1. A discrete time *martingale* is a sequence of random variables $X_1, X_2, X_3, \ldots$, also known as a stochastic process, that satisfies for any time n,

1. $E(|X_n|) < \infty$
2. $E(X_{n+1} \mid X_1, \ldots, X_n) = X_n$

i.e. the expected value of a random variable is always finite and the expectation of the next value is the same as the currently observed value, even if we know all the previous values. Martingales express a fair game where any prior knowledge does not give information on the next outcome.

Example. A gambler is playing a game of coin tossing. He will win one dollar if the result is heads and lose one dollar if the result is tails. Let X_n be the gamblers wealth after n coin tosses. The gamblers expected wealth after the next trial conditioned with the whole previous coin toss sequence is the same as his fortune at the moment. Thus, this sequence is a martingale.

Given a sequence $X_1, X_2, X_3, \ldots$ of random variables, a *martingale with respect to the sequence* (X_i) is another sequence of random variables $Z_1, Z_2, Z_3, \ldots$ if each Z_i satisfies

1. $E(|Z_i|) < \infty$
2. there exist functions g_i such that $Z_i = g_i(X_1, X_2, \ldots, X_i)$, and
3. $E(Z_i \mid X_1, \ldots, X_{i-1}) = Z_{i-1}$.

We will later see that the difference of the risk and the penalty by a loss function turns out to be a martingale. This is a very important observation in proving the necessary loss bounds for our conversion techniques. Now, we are ready to prove Azuma's inequality which plays a central role to bound our method.

Theorem 1. ***(Azuma's inequality [1])*** *Let $X_0, X_1, \ldots$ be a martingale such that for each i, $|X_i - X_{i-1}| \leq c_i$ for some $c_i > 0$. Then for all $\lambda > 0$ it holds that*

$$P(X_n - X_0 \geq \lambda) \leq exp\left(-\frac{\lambda^2}{2\sum_i c_i^2}\right).$$

Proof. For $i \geq 1$, let $Y_i = X_i - X_{i-1}$. Now $E[Y_i \mid X_0, \ldots, X_{i-1}] = 0$ since the sequence (X_i) is a martingale. Let $Z_n = \sum_{i=0}^{n} Y_i = X_n - X_0$. Now for any $t > 0$ we have

$$\begin{aligned}
P(Z_n \geq \lambda) &= P(e^{tZ_n} \geq e^{\lambda t}) \\
&\leq e^{-\lambda t} E[e^{tZ_n}] \\
&= e^{-\lambda t} E[E[e^{tZ_n} \mid X_0, \ldots, X_{n-1}]] \\
&= e^{-\lambda t} E[E[e^{tZ_{n-1}} \cdot e^{tY_n} \mid X_0, \ldots, X_{n-1}]] \\
&= e^{-\lambda t} E[e^{tZ_{n-1}} \cdot E[e^{tY_n} \mid X_0, \ldots, X_{n-1}]] \\
&\leq e^{-\lambda t} E[e^{tZ_{n-1}} \cdot e^{t^2 (c_n)^2/2}] \\
&= e^{-\lambda t} e^{t^2 (c_n)^2/2} \cdot E[e^{tZ_{n-1}}] \\
&\leq exp\left(\frac{t^2 \sum_i (c_i)^2}{2} - \lambda t\right)
\end{aligned}$$

where we used Lemma 4 (Appendix) to the random variable Y_n/c_n. It has 0 mean and takes values in $[-1, 1]$. We also solved $E[e^{tZ_{n-1}}]$ recursively.

As this holds for any $t > 0$, we choose $t = \frac{\lambda}{\sum_i (c_i)^2}$ and get

$$P[X_n - X_0 \geq \lambda] \leq exp\left(-\frac{\lambda^2}{2\sum_i (c_i)^2}\right).$$

□

The next lemma relates the empirical performance and average risk of hypotheses indexed by J. This guarantees that our conversions work in the sense that if we minimize the regret, we will get a batch hypothesis with a low risk. We will use Azuma's inequality in the proof.

Lemma 2. *Let $(X_1, Y_1), \ldots, (X_m, Y_m)$ be a sequence of examples independently drawn from D. Let $H_0, \ldots, H_m$ be a sequence of online hypotheses generated by some online algorithm A while observing this sequence of examples. Assume that the loss function l is bounded from above by some $R > 0$. Then for any $J \subset [m-1]$,*

$$P\left(\frac{1}{|J|}\sum_{i \in J} Risk_D(H_i) > \beta(J)\right) < exp\left(-\frac{C^2}{2R^2}\right)$$

where C is the constant used in the definition of β (Equation 3).

Proof. We begin by manipulating the expression:

$$\begin{aligned}
&P\left(\frac{1}{|J|}\sum_{i\in J} Risk_D(H_i) > \beta(J)\right) \\
&= P\left(\frac{1}{|J|}\sum_{i\in J} Risk_D(H_i) > L(J) + C|J|^{-\frac{1}{2}}\right) \\
&= P\left(\frac{1}{|J|}\sum_{i\in J} (Risk_D(H_i) - l(Y_{i+1}, h_i(x_{i+1})) > C|J|^{-\frac{1}{2}}\right) \\
&= P\left(\sum_{i\in J} (Risk_D(H_i) - l(Y_{i+1}, H_i(X_{i+1})) > C\sqrt{|J|}\right).
\end{aligned}$$

Next, we denote $Risk_D(H_i) - l(H_i(X_{i+1}), Y_{i+1})$ by V_i. Since l takes values in $(0, R)$ we get that $-R < V_i < R$ for all i:

$$\begin{aligned}
&E[V_i \mid (X_1, Y_1), \ldots, (X_i, Y_i)] \\
&= Risk_D(H_i) - E[l(H_i(X_{i+1}), Y_{i+1}) \mid (X_0, Y_0), \ldots, (X_i, Y_i)] = 0.
\end{aligned}$$

This means that the random variable sequence (V_i) is a martingale with respect to (X_i) by definition 1. First denote $\{j_0, \ldots, j_n\} = J$, which is possible because $J \subset [m-1]$. We can use Azuma's inequality 1 to $V_i's$ to get the desired bound:

$$\begin{aligned}
P\left(\sum_{i=0}^{n} Risk_D(H_{j_i}) - l(H_{j_i}(X_{j_{i+1}}), Y_{j_{i+1}}) > C\sqrt{|J|}\right) &= P\left(V_n - V_0 > C\sqrt{|J|}\right) \\
&< exp\left(-\frac{(C\sqrt{|J|})^2}{2\sum_{i\in J} R^2}\right) \\
&= exp\left(-\frac{C^2|J|}{2|J|R^2}\right) \\
&= exp\left(-\frac{C^2}{2R^2}\right),
\end{aligned}$$

which proves the lemma. □

The following theorem justifies the choice of β in Equation 3. This is a very important result as all the conversions rely on finding the optimal set of hypotheses.

Theorem 2. *Let h_J^* denote any of the three considered conversion strategies (Voting, Averaging or Stochastic). If the loss function l meets the requirements of Lemmas 1 and 2, then for all $J \in \mathcal{I}$ it holds that $Risk_D(h_J^*) \leq \beta(J)$ with a probability greater than or equal to*

$$1 - |\mathcal{I}| exp\left(-\frac{C^2}{2R^2}\right).$$

Proof. We prove this theorem for Averaging technique only, as the proofs for the other techniques are almost identical.

Lemma 1 says that $Risk_D(h_J^A) \leq \frac{1}{|J|}\sum_{i \in J} Risk_D(h_i(x))$. From lemma 2, we get that $Risk_D(h_J^A) < \beta(J)$ holds with probability at least $1 - exp(-C^2/2R^2)$ for any J. Now we can use union bound to get that $Risk_D(h_J^A) < \beta(J)$ with all $J \in \mathcal{I}$ simultaneously with probability at least

$$1 - \sum_{J \in \mathcal{I}} P(Risk_D(h_J^A) \geq \beta(J)) = 1 - |\mathcal{I}| exp\left(-\frac{C^2}{2R^2}\right).$$

□

Now when we increase the value of C, the size of the set J influences β on a greater degree. Increasing the value of C also increases the probability that β upper bounds the risk of hypotheses indexed by any $J \in \mathcal{I}$. The choice of β is now also justified theoretically.

After these theorems and lemmas, we have proved that the conversions given in the earlier section do work fairly well in theory. Now we improve these methods and move on to experimenting.

5 Data-Driven Conversion Algorithms

The idea of data-driven conversions is to use the provided data to find a set with as small $\beta-$value as possible. This set is then used with algorithms provided in Section 3 to get a batch hypothesis.

We discuss four different online to batch conversions. The first three conversions are by Dekel and Singer [12] and the fourth is by Dekel [11].

5.1 Suffix Conversion

Many online algorithms generate bad hypotheses during the first few rounds of learning. This sounds natural, as we start from a guess and the more examples we have seen the more we have learned. Therefore, we consider subsets of the form $\{a, \cdots, n\}$ for some $1 \leq a < n$. In this setting, we set $\mathcal{I}$ to be the set of all suffixes of $[n-1]$.

After the algorithm has generated the hypothesis sequence $h_0, \ldots, h_n$, we set $I = \operatorname{argmin}_{J \in \mathcal{I}} \beta(J)$. A downside of this conversion is that the memory requirement grows linearly with n. Also going through all the suffixes is very time consuming. This is seen in the experiments section: this conversion is the slowest one in our experiments.

Some heuristics for choosing the optimal value of a have been suggested by Li [25], but we use the β-function minimization approach, like Dekel and Singer did.

5.2 Interval Conversion

The idea of the Interval conversion is only sketched here, as introducing all the necessary background properly is out of the scope of this thesis. Interested reader may take a look at the original paper by Dekel and Singer [12] for a more thorough explanation.

A function h is a kernel-based hypothesis, if it is defined by

$$h(x) = \sum_{j=1}^{n} \alpha_j K(z_j, x),$$

where K is a Mercer Kernel *i.e.* positive semi-definite and z_j's are instances (also called support patterns) and α_j's are real valued weights.

Support Vector Machine (SVM) is an example of a batch algorithm that produces kernel-based hypotheses. An interesting learning problem is to learn a kernel-based hypothesis h that is defined by at most B support patterns,

where B is a predefined constant. This constant is often called the *budget* of the support patterns.

The less support patterns needed for representing a kernel-based hypothesis, the faster it is to calculate.

In online learning setting, a similar problem arises when we want to construct an online algorithm where each hypothesis h_i is a kernel-based function that is defined by at most B support patterns.

To convert a budget-constrained online algorithm A into a budget-constrained batch algorithm, we need to make an additional assumption of the update strategy used by A. We have to assume that whenever the algorithm A updates the hypothesis, it adds a new support pattern to the set that represents the kernel-hypothesis. Now we can choose I to be the set $\{a, a+1, \ldots, b\}$ for some integers $0 \leq a < b < n$.

Let A update its hypothesis k times during rounds $a+1$ through b. Then the resulting hypothesis is defined by at most $B + k$ support patterns. So we define $\mathcal{I}$ to be the set of all non-empty intervals in $[m-1]$. Now $\beta(J)$ bounds the risk $Risk_D(h_J^A)$ for every $J \in \mathcal{I}$ with high probability.

Next we generate the hypothesis sequence by running the algorithm with budget parameter $B/2$. Then we choose I to be the set in $\mathcal{I}$ which contains at most $B/2$ updates and minimizes β. By this construction, the resulting hypothesis h_I^A is defined by at most B support patterns.

5.3 Tree-Based Conversion

In this conversion, we build the subset $\mathcal{I}$ recursively. This way, we do not have to store the hypothesis sequence in memory like with suffix conversion. First, we assume without loss of generality that n is a power of 2, as we can just add more hypotheses with infinite loss, if needed. We define $J_{a,a} = \{a\}$ for all $0 \leq a \leq n-1$.

Next, we assume that we have already constructed the sets $J_{a,b}$ and $J_{c,d}$, where a, b, c and d are integers such that $a < d$, $b = (a+d-1)/2$ and $c = b+1$ *i.e.* we divide the interval in half. Now we define

$$J_{a,d} = \begin{cases} J_{a,b} & \text{if } \beta(J_{a,b}) \leq \beta(J_{c,d}) \text{ and } \beta(J_{a,b}) \leq \beta(J_{a,b} \cup J_{c,d}) \\ J_{c,d} & \text{if } \beta(J_{c,d}) \leq \beta(J_{a,b}) \text{ and } \beta(J_{c,d}) \leq \beta(J_{a,b} \cup J_{c,d}) \\ J_{a,b} \cup J_{c,d} & \text{otherwise} \end{cases}$$

Finally, we define $\mathcal{I} = J_{0,m-1}$ and output the batch hypothesis h_I^A. To put it in words, we start by looking at each individual hypothesis and their beta-value and we will compare every hypothesis h_i with its neighbor h_{i+1}

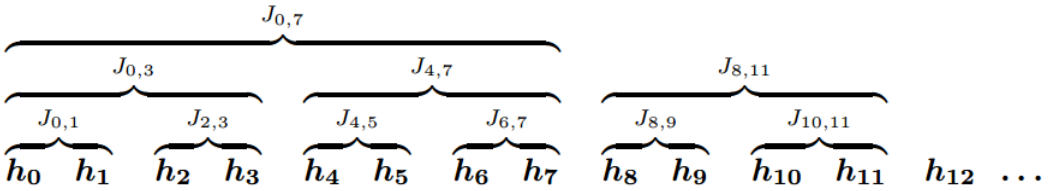

Figure 6: An example of the tree based conversion. The picture is from Dekel and Singer [12]

.

and then move to the next hypothesis pair that has not been compared yet. Then we will choose either of the hypotheses or merge them, depending which of these options get the smallest beta-value.

The benefit of this conversion is that it can also be done in parallel with the online rounds.

When the instances are elements in $\mathbb{R}^n$ and Averaging conversion is used with linear hypotheses, the memory utilization with this conversion is only $\mathcal{O}(\log n)$ compared to memory utilization of suffix conversion that is $\mathcal{O}(n)$.

The reason for this is that in this special case the hypotheses are weight vectors and we can discard all the online hypotheses $h_a, \ldots, h_b$ after we have calculated $J_{a,b}$. We also don't have to save the set $J_{a,b}$, we only need to know the loss $L(J_{a,b})$ and the size of this set. Storing these require only constant amount of memory so the overall memory utilization is indeed $\mathcal{O}(\log n)$.

5.4 Cutoff-Averaging

This algorithm is from Dekel, published in 2009 [11] and it is considered an improvement to the above three conversions.

Cutoff-Averaging conversion can be used for any *conservative* online learning algorithm that utilizes a *convex* hypothesis set. Conservative learning algorithms change the hypothesis only when a mistake is made. If the prediction is correct, the weight vector stays the same for predicting the next example. The convexity constraint is needed, as in the earlier Averaging conversion algorithm.

The idea of the algorithm is to calculate the survival time for the hypotheses. The survival time of a hypothesis is the number of consecutive rounds it is used in predicting before it makes a mistake and is replaced with a new hypothesis. After calculating the survival time for all the hypotheses

we keep only the hypotheses with survival time greater than some predefined cutoff parameter k.

The conversion outputs the weighted average over all the survived hypotheses. Weight of a hypothesis that survived m times is $m-k$. Ultimately, the optimal cutoff parameter k is chosen by the algorithm itself; if there are few outstanding hypotheses we take only them and set k high. If no clear winners are found, the cutoff parameter is set small.

This conversion is fast because we can collect all the survived hypotheses during the online phase. Together with good accuracy, this conversion captures the idea of online to batch conversion as it is easy to implement and significantly faster than the suffix conversion.

Statistical Analysis

The analysis presented earlier does not apply to the Cutoff-Averaging-algorithm. Therefore, we are going to go through some statistical analysis and justify the correctness of the conversion before going further. The analysis is mostly adapted from Dekel [11]. To refresh memory on conditional expectation, please see the Appendix.

We represent the examples as a sequence of random variables $((X_i, Y_i))_{i=1}^m$. From online algorithm A, we get the online hypothesis sequence $(H_i)_{i=1}^n$. This is a sequence of random functions. Each of these functions are defined deterministically by $((X_i, Y_i))_{i=1}^m$. Since the examples are independent we get that

$$l(H_i, D) = E(l(H_i, (X_{i+1}, Y_{i+1})) \mid ((X_j, Y_j))_{j=1}^i)$$

It means that the risk of H_i is the same as the expectation of the online loss on round $i+1$ conditioned on all the previous examples. With this observation we can again convert the regret bound into risk bound. We define a sequence of binary random variables as follows

$$B_i = \begin{cases} 1 & \text{if } i = 0 \text{ or if } i \geq k \text{ and } H_{i-k} = H_{i-k+1} = \cdots = H_i \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

Now we can define the output hypothesis to be

$$H_k^* = \left(\sum_{i=0}^{m-1} B_i \right)^{-1} \sum_{i=0}^{m-1} B_i H_i. \tag{5}$$

Setting the survival time parameter k to 0 signifies that every hypothesis in the sequence is qualified and the procedure reduces to standard Averaging

conversion technique. Setting the value k larger and larger we will achieve the longest survivor technique i.e. we will use only the hypothesis that survived the longest and discard others.

The next theorem bounds the risk of H_k^* in case where the used loss function is convex and bounded. This is the case with Hinge loss and the Perceptron algorithm we are going to use in experiments. The theorem and the related lemma are adapted from [11]. To prove the theorem we need the following lemma which is a corollary of Freedman's tail bound for martingales [14].

Lemma 3. *Let $(L_i)_{i=1}^m$ be a sequence of real-valued random variables and let $(Z_i)_{i=1}^m$ be a sequence of arbitrary random variables such that $L_i = E[L_i|(Z_j)_{j=1}^i]$ and $L_i \in [0, C]$ for all i.*

Define $U_i = E[L_i|(Z_j)_{j=1}^{i-1}]$ for all i, and define $\bar{L}_t = \sum_{i=1}^t L_i$ and $\bar{U}_t = \sum_{i=1}^t U_i$ for all t. For any $m \geq 4$ and for any $\delta \in (0, 1)$, with probability at least $1 - \delta$, it holds that

$$\forall t \in \{1, \ldots, m\} \quad \bar{U}_t < \bar{L}_t + \sqrt{2C \ln(\frac{m}{\delta})\bar{L}_t} + 7C \ln(\frac{m}{\delta}).$$

The proof of this lemma is found in the technical appendix of Dekel's article [11].
Now we are ready to prove the main theorem related to Cutoff-Averaging.

Theorem 3. *Let k be a non-negative constant and let l be a convex loss function such that $l(h \mid (x, y)) \in [0, C]$. An online algorithm is given $m \geq 4$ independent samples from distribution D and it constructs the online hypothesis sequence $(H_i)_{i=0}^m$.*

Define B_i and H_k^* as in equations (4) and (5), let $L_i = B_{i-1} l(H_{i-1}; (X_i, Y_i))$ for all i and let $\bar{L} = (\sum B_i)^{-1} \sum L_i$. For any $\delta \in (0, 1)$, with probability at least $1 - \delta$, it holds that

$$l(H_k^*; D) < \bar{L} + \sqrt{\frac{2C \ln(\frac{m}{\delta})\bar{L}}{\sum B_i}} + \frac{7C \ln(\frac{m}{\delta})}{\sum B_i}.$$

Proof. Define $U_i = E[L_i|(X_j, Y_j)_{j=1}^{i-1}]$ for all $i \in \{1, \ldots, m\}$, and define $\bar{U} = \sum_{i=1}^m U_i$. Using lemma 3, we have that with probability at least $1 - \delta$

$$\bar{U}_t < \bar{L}_t + \sqrt{2C \ln(\frac{m}{\delta})\bar{L}_t} + 7C \ln(\frac{m}{\delta}).$$

By definition we have

$$U_i = E[B_{i-1} l(H_{i-1}; (X_i, Y_i)) | ((X_j, Y_j))_{j=1}^{i-1}].$$

Since B_i is deterministically defined by $((X_j, Y_j))_{j=1}^{i-1}$, it can be taken out of the conditional expectation above. Now we have $U_i = B_{i-1} l(H_{i-1}; D)$. Overall, we have shown that

$$\sum_{i=1}^{m} B_{i-1} l(H_{i-1}; D) < \overline{L} + \sqrt{2C \ln(\frac{m}{\delta}) \overline{L}_t} + 7C \ln(\frac{m}{\delta}).$$

Using Jensen's inequality, the left-hand side above is at least

$$\left(\sum_{i=1}^{m} B_{i-1} \right) l(H_k^*; D).$$

□

The Cutoff-Averaging technique sets the output hypothesis H^* to be the hypothesis with smallest bound obtained from the Theorem 3 from the set $\{H_0^*, \ldots, H_{m-1}^*\}$. So the number k is chosen automatically. If a small number of online hypotheses have long survival times, k will be large. In the case where most hypotheses have small survival times, k will be small and more hypotheses will be taken to the ensemble.

This conversion is quite similar to the Averaging conversion technique described earlier in this section. As discussed before, the suffix conversion requires $\mathcal{O}(m)$ space whereas the Cutoff-Averaging has smaller memory utilization. The memory requirement of $\mathcal{O}(\sqrt{m})$ is achieved with the following observation.

The technique cannot choose the optimal value k before it has seen the whole dataset, but it can group the seen online hypotheses based on their survival times and store the average hypothesis and total loss in each group. Then calculating the optimal k and outputting the hypothesis is straightforward. In a sequence of length m, the maximum number of distinct survival times x is calculated by solving the following inequality:

$$\sum_{i=1}^{x} i = \frac{x(x+1)}{2} < m,$$

where the maximum for x is proportional to the square root of m. Hence the memory requirement for this conversion is $\mathcal{O}(\sqrt{m})$.

5.5 Wrap-up

We considered four different conversion: Suffix, Interval, Tree-Based and Cutoff-Averaging. In the article by Dekel and Singer where the first three were introduced, the authors claimed that all the three perform roughly equally. So when choosing a conversion for a given problem, the important difference between these conversions is the target space and memory requirement.

The most intuitive and easiest to implement is the Suffix conversion. But calculating the β-value for each possible suffix is very time consuming. But we have to pay this price only once, and when we are done, the resulting suffix of the hypothesis sequence tend to perform well in practice. This is more emphasized in the next section.

The Tree-Based conversion saves memory compared to the Suffix conversion and is easier to both understand and implement than the Interval conversion. It is also relatively fast and gives reliable results as seen in the *Experiments* section.

The Cutoff-Averaging is more sophisticated than the other conversions. It is both fast and yields good results which is verified in the next section. The theoretical analysis of this conversion is somewhat complicated, but the implementation is straightforward.

6 Experiments

Online to batch conversions have been studied by only a handful of people and there are not many papers of the subject. The latest I am aware of is from 2009 by Dekel [11]. In this paper, he proposes the Cutoff-Averaging algorithm, which seems to be still the state-of-the-art conversion from online to batch.

Some well-performing algorithms studied by Dekel and Singer in 2005 [12] require finding optimal suffixes of the hypothesis sequence H. Their algorithm searches through and analyzes the whole set of suffixes. One of our goals is to find sophisticated ways to prune the set of suffixes before further analysis. One way to achieve this could be by observing the behavior of the β-value as a function of the suffix length to find useful properties.

We will use MNIST [24] handwritten digits, by Lecun and Cortes, as our database, because it is well-known set used for classification. Also Dekel and Singer [12] used MNIST in some of their experiments. The online algorithm they used is the Passive-aggressive algorithm [10]. In addition to this, we use the Perceptron algorithm [28].

The MNIST data set consists of 60,000 images of the size 786 pixels. Some of the conversions have to run through the data several times, which is quite time consuming. This is especially emphasized with both the Suffix conversion and the Voting method.

The experiments can be divided into three phases. First, one of the two online algorithms is applied to the training set consisting of images and correct labels. As a result, we get the hypothesis sequence H the algorithm generated.

In the second phase, we run the conversions. This is the task we are mostly interested in. Picking the last hypothesis serves as the baseline conversion. We apply the suffix method to H in order to obtain the best hypothesis set J with respect to small β-value. We experiment with different values of C when calculating the β-value in data-driven conversions. This means that we will have to run through the data multiple times. We study the Voting, Averaging and stochastic methods applied to the pruned sequence J.

The last conversion we run is Cutoff-Averaging. We also experimented something we call *Cutoff-Voting* with this algorithm: we use the Voting conversion to the hypothesis set constructed by Cutoff-Averaging instead of the Averaging conversion.

In the last phase we record the error rate and time consumed of the conversions. We represent the results visually. We use the k-fold cross-validation, *i.e.* divide the training set into k parts and use one of them in testing and others for training. The value of k will vary from 3 to 10 as in the

article by Dekel and Singer [12]. We will then study the standard deviation of errors of every algorithm used on MNIST.

Next we will go through the results of the previous experiments by Dekel and Singer.

6.1 Previous Experiments by Dekel and Singer

In the article by Dekel and Singer [12] the Voting and Averaging conversions are tested. The algorithm used is multiclass version of the Passive-Aggressive algorithm. The datasets used are *MNIST*[24], *USPS* [23], *ISOLET* [7] and *LETTER* [2]. The sizes are 70000, 7291, 7792 and 20000, respectively.

The MNIST and USPS are databases of handwritten digits. The LETTER is a database of handwritten letters and ISOLET is a database of spoken letters. So each of these datasets have a discrete output space.

The experiments are done with cross validation where the training set is split into a k distinct parts and each algorithm is trained on using each of these parts and tested on remaining $k-1$ parts. The value of k varied from three to ten

The data was applied to (data-independent) Averaging and Voting conversion and to the three data-driven variants. The number of different data-driven conversions is six. The parameter C in the definition of β-function was set to three. The interval conversion was set to choose an interval containing 500 updates. Then test error of the conversions was calculated and compared especially to the "pick last hypothesis" conversion.

The results by Dekel and Singer are convincing. The suffix and tree-based conversions constantly beat their respective data-independent variants. The interval conversion loses occasionally to the data-independent variants. In general, the Averaging technique achieves best results while the Voting conversion is close second. Pick last hypothesis conversion is almost always inferior.

The Cutoff-Averaging algorithm by Dekel is tested in his article [11] with the *Reuters Corpus Vol. 1 (RCV1)* dataset. This set has over 800k news articles. Each article is associated with one or more high level label, which are: Markets (MCAT), Government/Social (GCAT), Economics (ECAT), Corporate/industrial (CCAT), and Other (OTHER). Roughly 20% of the articles has more than one high level label. After removing those Dekel was left with over 600k articles. The results are shown in the Figure 7.

The algorithm was used with regular Perceptron and *Margin-based* Perceptron. Here margin-based means that the algorithm is minimizing the hinge-loss instead of zero-one loss. Each of the experiments was performed

	last	average	average-sfx	voting	voting-sfx
LETTER 5-fold	29.9 ± 1.8	21.2 ± 0.5	$\mathbf{20.5} \pm 0.6$	23.4 ± 0.8	21.5 ± 0.8
LETTER 10-fold	37.3 ± 2.1	26.9 ± 0.7	$\mathbf{26.5} \pm 0.6$	30.2 ± 1.0	27.9 ± 0.6
MNIST 5-fold	7.2 ± 0.5	5.9 ± 0.4	$\mathbf{5.3} \pm 0.6$	7.0 ± 0.5	6.5 ± 0.5
MNIST 10-fold	13.8 ± 2.3	9.5 ± 0.8	9.1 ± 0.8	8.7 ± 0.5	$\mathbf{8.0} \pm 0.5$
USPS 5-fold	9.7 ± 1.0	7.5 ± 0.4	$\mathbf{7.1} \pm 0.4$	9.4 ± 0.4	8.8 ± 0.3
USPS 10-fold	12.7 ± 4.7	10.1 ± 0.7	$\mathbf{9.5} \pm 0.8$	12.5 ± 1.0	11.3 ± 0.6
ISOLET 5-fold	20.1 ± 3.8	17.6 ± 4.1	$\mathbf{16.7} \pm 3.3$	20.6 ± 3.4	18.3 ± 3.9
ISOLET 10-fold	28.6 ± 3.6	25.8 ± 2.8	$\mathbf{22.7} \pm 3.3$	29.3 ± 3.1	26.7 ± 4.0

Table 2: The results of pick last conversion, average conversion, voting conversion and the suffix method testing by Dekel and Singer [12]. The best performance is bolded. The figure is adapted from Dekel and Singer [12].

10 times with new permutation of the data and a new split into a training set and a test set. For a more detailed explanation of the experiments performed by Dekel, please see the article [11].

In the Figure 7, the pick last hypothesis conversion gives unstable results and is not as accurate as the Cutoff-Averaging technique when the standard Perceptron is used.

With the Margin-Based Perceptron, the pick last hypothesis gives better results and the simple Averaging technique does not perform as well. The Cutoff-Averaging starts to perform better than the pick last hypothesis after some online rounds.

This is expected as the performance of the Cutoff-Averaging relies on tail-bounds with enough samples. When the training set is big enough, the Cutoff-Averaging algorithm will see that a small set will be best and adjust its parameters automatically. More detailed experimentation can be found from Dekel's article and my following experiments.

6.2 Results

We have performed experiments on the MNIST database of handwritten digits by Lecun and Cortes [24]. The dataset consists of a training set of 60,000 examples and a test set of 10,000 examples. Each example is a 28x28 pixel picture of a handwritten digit. Each pixel has a value between 0 and 255 to describe the intensity of the picture in that pixel. An example of 100 handwritten digits from this dataset is shown in Figure 8.

The conversions only need the hypothesis sequence, and they do not have any prior knowledge of the algorithms used. We chose the Perceptron and Passive Aggressive algorithms for two reasons. First of all, they are easy to implement and they give fairly good results on the MNIST dataset. Secondly,

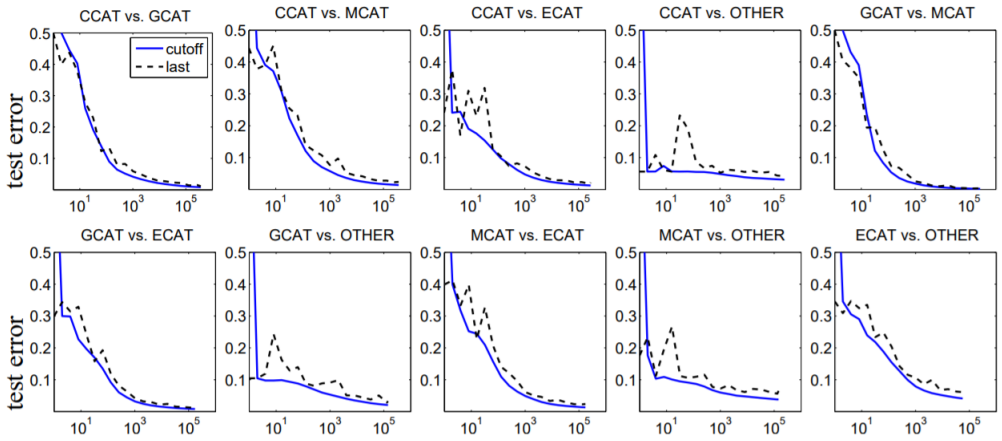

Figure 7: The test error of the Cutoff-Averaging compared to the pick last hypothesis method. These two conversions are applied to the standard Perceptron on ten binary classification from the *Reuters Corpus Vol. 1 (RCV1)* dataset. The x-axis represents the training set size on log-scale. Each plot shows the average over 10 random training and testing splits. The picture is from Dekel [11]

.

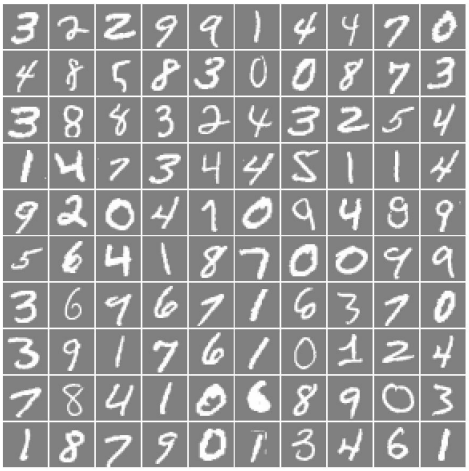

Figure 8: A visualization of 100 handwritten digits from the MNIST dataset.

the PA algorithm is used in the experiments in the original article by Dekel and Singer [12], so comparison of the results are easier.

Note that the error percentage of our implementation might be larger than those reported by Dekel and Singer. This is probably because we have implemented only very simple versions of the Perceptron and PA and no optimization is done for these. Tuning these algorithms was not relevant as we are interested of the actual conversions, not online learning algorithms themselves.

We focus on the *difference* between the error percentages of the various conversions rather than the actual error percentage. There was no noticeable difference in terms of error between the hypothesis sequences generated by the Perceptron algorithm and the PA algorithm. In the following 4 test scenarios we have constructed each of the hypothesis sequence with the Perceptron algorithm, so that the scenarios are comparable with each other.

Our experiments are done with Matlab by Mathworks [34]. First, the data is loaded and the pictures and corresponding labels are extracted to separate matrices. We shuffle the data between each iteration, but save the seed of random so we can rerun any experiment if needed. Then, we choose the size of the training and testing sets and pass the pictures and labels to the Perceptron algorithm, which will return a hypothesis sequence.

This sequence is then used with various conversions. Of data-independent conversions, we have implemented the *pick last hypothesis* (from now on *PLH*), Sampling, Averaging and Voting conversions. These conversions use the whole hypothesis sequence in the conversion, as discussed earlier. The batch hypotheses generated with these conversions are then tested with unseen data and the error percentages are recorded.

After these conversions, we introduce the β-function and extract a subset of the hypothesis sequence that minimizes the function. The suffix and the tree-based conversion are used and the subsets are saved. Then we run the Sampling, Averaging and Voting conversions again, but only using the chosen set of hypotheses.

Error percentages of the conversions are saved and the process is ran multiple times with different training and test sets. The sizes of these sets vary. Last, we compare all error percentages and variance of these multiple conversions. The results are presented in boxplots.

The last conversion we test is the Cutoff-Averaging. At first we implemented the idea behind the algorithm as follows. We chose the survival time k by hand and tried it with different values. Then we chose the hypotheses that survived k or more times and used these with Voting conversion. The amount of votes a hypothesis have in this model is proportional to the time

it survived. We call this conversion Cutoff-Voting.

This conversion is accurate, but it leads to bigger memory consumption than the Cutoff-Averaging conversion. The upside is that it was easy to implement. Implementing the Cutoff-Averaging conversion as described in the article by Dekel was more laborious.

The following notes were taken during multiple iterations of this test framework: Loading the data, shuffling, and running one epoch of the Perceptron algorithm take only seconds even on a big training set. The Voting method is noticeably slower than other methods as each hypothesis in the sequence casts a vote for each picture in the test set. This means that the running time is proportional to the size of training set multiplied by the size of the test set. In most cases, this is quadratic.

The suffix conversion is an other slow part of one iteration, as we have to calculate the β-value for each suffix. This is also quadratic. The data driven Voting and Averaging are faster than their data-independent counterparts as the hypothesis set is smaller. In a sense, this mimics the training with batch methods as the training takes time, but once we have the batch hypothesis the testing is very fast.

We will illustrate four test cases. The sizes of the training sets are relatively small, as we are especially interested of the performance of the conversions when training is hard. Other reason is that running the whole test case with multiple iterations takes time and the resulting batch hypotheses perform close to each other.

In the following figures we have used boxplot to illustrate the accuracy of the resulting batch hypothesis, constructed by different conversions. In the y-axis we plot the success percentage of predicting a label of a given conversion, when presented a batch of unseen pictures of handwritten digits.

We start by looking at a rather small case, where the size of the training set is 600 and the size of the test set is 900. We have run this test scenario 30 times, every time with a different training and test sets while keeping their size the same.

As seen in the boxplot in Figure 9, the Sampling conversion is the worst in terms of accuracy. The PLH conversion gives better results, but still loses to other conversions. The most notable thing in this setting is that the PLH conversions performance varies noticeably: the worst performing classifier produced by PLH got under 60% of the labels right in the test set, whereas for example all the 30 classifiers produced by Voting conversion got over 75% of the labels right.

Other conversions are quite stable with good accuracy. It is worth to note that all the other conversions give better results than PLH even in this

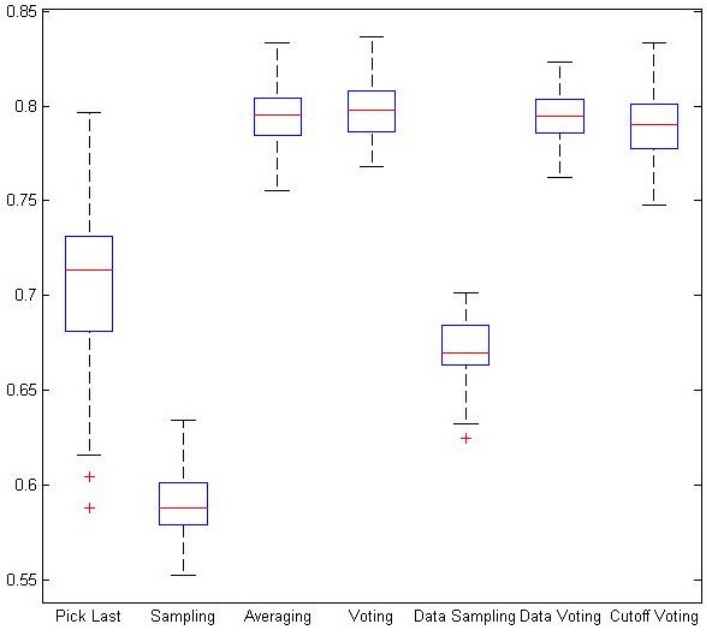

Figure 9: The success percentages of seven different conversions over 30 iterations. Data-driven conversions are done with suffix conversion. The size of the training set is 600 and the size of the test set is 900.

small test scenario except the Sampling conversion, both data-independent and data-driven one. But the performance of the Sampling conversion is more predictable than the performance of the PLH conversion.

With this small training set of only 600 examples, the data-driven conversions do not perform any better than the data-independent ones, except for Sampling conversion. This is because the conversion chooses the hypothesis randomly, but when we have constructed the set by minimizing the β-function, all the hypotheses in that set are fairly accurate.

In Figure 10, we have a test scenario with more training data, so the hypothesis sequence generated by online algorithm is longer. The size of the training set is 1000 and the size of the test set is 2000. This is iterated 10 times. In table 3 the actual success percentages of each conversion per iteration are shown. The best performing conversion per iteration is shown in bold.

With this setting, the Cutoff-Voting is gaining edge. This accuracy is not much more than with the data-independent ones though. Picking the last hypothesis has again lots of variance and does not give as accurate results as the other conversions excluding Sampling.

The performance of data-independent Voting seems to be better than

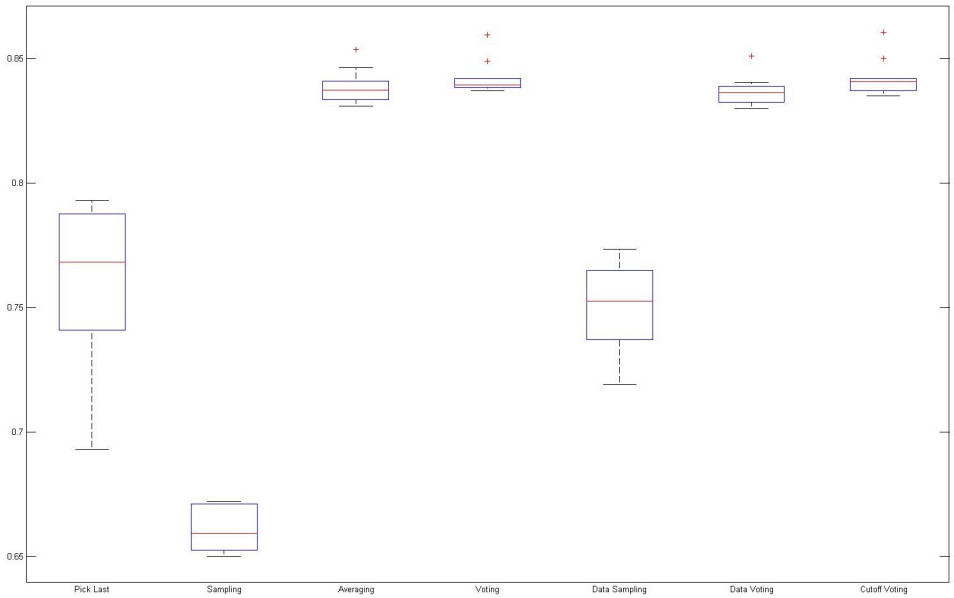

Figure 10: The size of the training set is 1000 and the test set is 2000. The Cutoff-Averaging is most accurate and data-driven conversions give better results than their data-independent counterparts.

that of data-driven one. This is curious, as the data-driven sampling is clearly better than data-independent sampling. We have not studied this phenomenon further in this thesis.

The boxplot in Figure 11 is generated from a small training set of size 100, and the size of the test set is 200. We repeated this 100 times. This scenario emphasized the observation that the performance of the PLH conversion varies lot from run to run. The best result was approximately 0.71 of right predictions whereas in the worst run it managed to predict only 0.29 of the test samples correctly. This highlights the fact that picking the last hypothesis as the batch hypothesis is risky and can give inferior results.

With this small training set, it is not really feasible to implement the data-driven conversions, as they perform roughly the same, or even worse, than their data-independent counterparts.

When the size of the training set is increased, the suffix conversion becomes slow. This is expected as it is quadratic. Also the Voting conversion is noticeably slow with a training set size of 9000 as in Figure 12. The Averaging and Voting, both data-independent and data-driven versions are performing well compared to other conversions. The free parameter C should be tuned to achieve better results with data-driven conversions in this setting.

Table 3: The success percentage of conversions in Figure 10

PLH	**Sample**	**Average**	**Voting**	**D-sample**	**D-Voting**	**Cut-Off**
0.7845	0.65	**0.8405**	0.8395	0.7365	0.831	**0.8405**
0.7795	0.672	0.835	**0.8385**	0.737	0.83	0.836
0.7445	0.657	0.8335	0.837	0.7385	**0.838**	0.835
0.7875	0.6525	0.8465	0.849	0.765	0.8405	**0.85**
0.741	0.6605	0.831	**0.8385**	0.755	0.8325	0.837
0.693	0.672	**0.841**	0.837	0.762	0.8355	0.84
0.793	0.658	0.839	**0.842**	0.75	0.839	0.8415
0.729	0.663	0.8535	0.8595	0.77	0.851	**0.8605**
0.757	0.652	0.836	0.8405	0.719	0.834	**0.841**
0.7925	0.671	0.8335	0.8395	0.7735	0.837	**0.842**

The Voting conversion is faster with data-driven version. Even though calculating the β-value for all suffixes takes time, after we have the set of best suffixes the Voting is faster, as there are fewer hypotheses casting vote. We expect that this speed-up would be even noticeable if we used larger test set.

On very big sample sizes the same behaviour continues: the PLH conversion has still plenty of variance compared to other conversions. The ensemble methods give reliability and confidence on both big and small training sets.

All of these experiments highlight the fact that the PLH conversion can give inferior results from time to time. The advantage of the data-driven versions compared to their data-independent counterparts is not so clear. Sometimes they seem to perform a bit better and sometime there is no real difference between the data-independent and the data-driven ones.

I am, however, quite sure that the implementation of the tree and suffix conversion work. Even if it is not so clear with the Voting and Averaging conversion, we can see it from the performance of the Sampling conversion. The data-driven Sampling conversion is always noticeably better than the data-indepent counterpart. This is because some of the hypotheses at the start of the hypothesis sequence are basically just guessing, or they recognize only a class or two, but have not seen any example of most of the classes.

The poor performance of the data-driven Voting and Averaging conversions could be because of the MNIST dataset itself. Some of the hand-written digits are easily recognized, whereas others are hard. For example, the digit 7 get often confused with digit 9. Nevertheless, the speed-up in predicting is often so significant that one should consider the data-driven versions.

In Table 4, we have listed both the running times of some of the conversions as well as the size of the suffix conversion from the test scenario

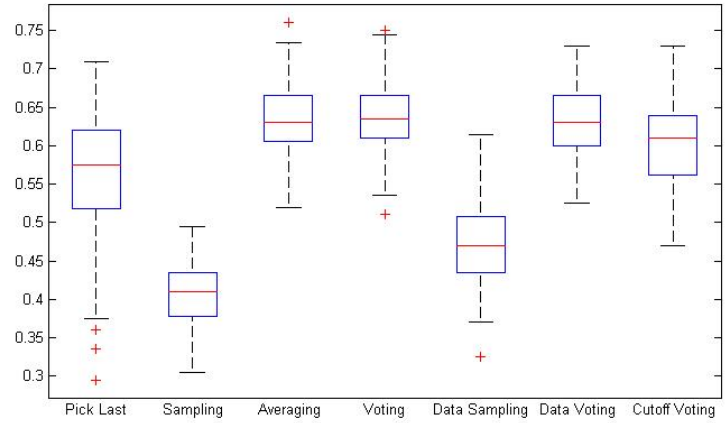

Figure 11: Here the size of the training set is only 100 and the test set size is 200. The performance is tracked over 100 iterations. The PLH methods unstability is clearly seen and we can also see that the Cutoff-Voting does not perform very well as there are not enough samples. Data-driven Voting performs roughly as well as the data-independent version, which is expected with a small training set.

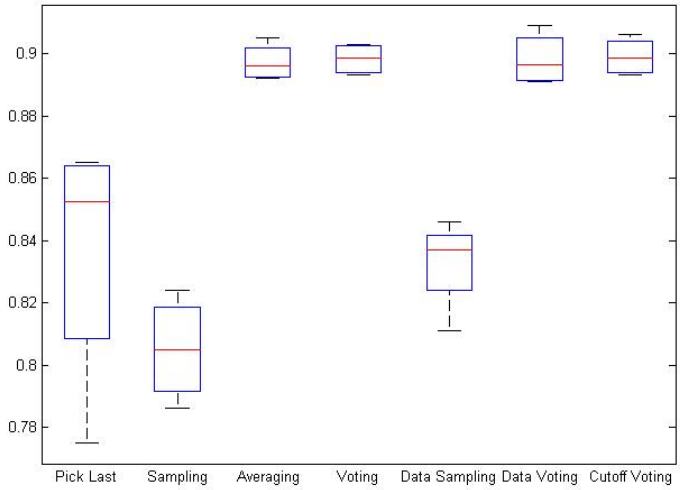

Figure 12: The size of the training set is 9000 and the size of the test set is 1000. The number of iterations is 4. With this large size of the training set, the Voting method and suffix conversion are infeasible. The other conversions run fast enough.

Table 4: Runtimes

	Iter. 1	Iter. 2	Iter. 3	Iter. 4
Perceptron (seconds)	1.41	1.39	1.42	1.44
PLH (seconds)	0.088	0.092	0.087	0.087
Sampling (seconds)	0.21	0.22	0.21	0.22
Voting (seconds)	705	684	677	675
Averaging (seconds)	581	505	548	536
Suffix Size	4891	2808	4719	4374
Suffix (seconds)	3580	3540	3500	3510
Data Driven Sampling (seconds)	0.119	0.119	0.12	0.122
Data Driven Voting (seconds)	368	206	357	353

illustrated in Figure 12. We chose this extreme scenario, as it shows best how long some of these conversions take. The online algorithm itself takes mere seconds to run, and the PLH and Sampling are both run in under a second. Finding the optimal suffix is the most time consuming part, and it took almost an hour.

After finding the optimal suffix, the Data Driven Voting is faster than the data-independent version. The speed-up is directly proportional to the size of the suffix.

7 Conclusion

We have studied a few conversions, both data-independent and data-driven ones, so we have some insight of the bottlenecks in implementation and running speed as well as some suggestions for improvement.

As online to batch conversion is little studied, there are many aspects where we can improve. The practice of picking the last online hypothesis to a batch hypothesis has one advantage and that is speed. Many of the conversions described in this thesis lack the speed and are even unusable on big datasets.

The high variance of the error with PLH is still something that one can not ignore. One possibility could be to just choose a relatively small constant suffix of the hypothesis sequence and use some ensemble method on that set. This way, the probability that the resulting batch hypothesis is performing poor should should be lower than with PLH.

One could also study heuristics for picking an optimal subset of hypotheses from the hypothesis sequence. Especially the suffix conversion is very slow when the hypothesis sequence is long and our experiments suggest that calculating the β-value for a small set of suffixes provides a set of hypotheses that work roughly as good as the slow suffix conversion. So far this is seen only in practice and we do not present any theoretical guarantees of this phenomenon.

The Voting conversion is also impractical when there are many hypotheses casting a vote. This is seen clearly with the data-independent Voting conversion as every hypothesis in the sequence casts a vote. Nevertheless, this conversion gives good results in our experiments even when the voting-set is rather small and it is accurate with the MNIST dataset. The smaller the set of hypotheses giving the vote, the faster the resulting predictor is. If we can extract only few hypotheses to this set, we expect it to be suitable even for big data masses.

The Cutoff-Averaging algorithm seems to capture the idea of online to batch conversions. It gives accurate results and can adapt by choosing a large or small set of hypotheses to be used in constructing the batch hypothesis. It is also considerably faster than the other data-driven conversions in this thesis. The idea of using hypotheses with large enough survival time can also be used with for example Voting conversion, but then the algorithm is not as fast as with Averaging.

The downside of Cutoff-Averaging is that it does not perform as well as other conversions when the training set is very small. This makes sense as bad hypotheses may survive a few rounds in the beginning just by guessing right. The performance of this conversion grows rapidly with the training

set size. Also certain assumptions of the data have to be fulfilled, so this conversion cannot be used for all learning tasks.

The idea of using the actual data in the conversion makes sense. This should be studied more, whilst keeping in mind that the big idea behind the conversions is simplicity. The algorithms should be both easy to implement and fast to run. If the conversion lacks either of these two, picking the last hypothesis is still tempting.

References

[1] Azuma, Kazuoki. "Weighted sums of certain dependent random variables." *Tohoku Mathematical Journal, Second Series 19.3, 357-367.* 1967

[2] Blake, Catherine L., and Christopher J. Merz. "UCI repository of machine learning databases." . 1998

[3] Boser, Bernhard E., Isabelle M. Guyon, and Vladimir N. Vapnik. "A training algorithm for optimal margin classifiers." *Proceedings of the fifth annual workshop on Computational learning theory. ACM.* 1992

[4] Campbell, Murray, A. Joseph Hoane, and Feng-hsiung Hsu. "Deep blue." *Artificial intelligence 134.1-2: 57-83.* 2002

[5] Cesa-Bianchi, Nicolo, Alex Conconi, Claudio Gentile. "On the Generalization Ability of On-Line Learning Algorithms." *IEEE TRANSACTIONS ON INFORMATION THEORY, VOL. 50, NO. 9.* September 2004

[6] Chan, Philip K., and Salvatore J. Stolfo. "Toward Scalable Learning with Non-Uniform Class and Cost Distributions: A Case Study in Credit Card Fraud Detection." *KDD (pp. 164-168).* 1998

[7] Cole, Ron. "The ISOLET spoken letter database." *CSETech. 205. http://digitalcommons.ohsu.edu/csetech/205.* 1990

[8] Collins, Michael. "Discriminative training methods for hidden markov models: Theory and experiments with perceptron algorithms." *Proceedings of the ACL-02 conference on Empirical methods in natural language processing-Volume 10. Association for Computational Linguistics.* 2002

[9] Cortes, Corinna, and Vladimir Vapnik. "Support-vector networks." *Machine learning 20.3: 273-297.* 1995

[10] Crammer, Koby *et al.*. "Online passive-aggressive algorithms." *The Journal of Machine Learning Research 7: 551-585.* 2006

[11] Dekel, Ofer. "From online to batch learning with cutoff-averaging." *Advances in Neural Information Processing Systems.* 2009

[12] Dekel, Ofer, and Yoram Singer. "Data-driven online to batch conversions." *Advances in Neural Information Processing Systems.* 2005

[13] Domingos, Pedro. "A few useful things to know about machine learning." *Communications of the ACM 55.10: 78-87.* 2012

[14] Freedman, David A. "On tail probabilities for martingales." *the Annals of Probability: 100-118.* 1975

[15] Freund Yoav, and Robert E. Schapire. "Large margin classification using the perceptron algorithm." *Machine Learning, 37(3):277–296.* 1999

[16] Gallant, Stephen I. "Optimal linear discriminants." *Eighth International Conference on Pattern Recognition.* 1986

[17] Gallant, Stephen I. "Perceptron-based learning algorithms." *Neural Networks, IEEE Transactions on 1.2, 179-191.* 1990

[18] Guzella, Thiago S., and Walmir M. Caminhas. "A review of machine learning approaches to spam filtering." *Expert Systems with Applications 36.7: 10206-10222.* 2009

[19] Helmbold, David P., and Manfred K. Warmuth. "On weak learning." *Journal of Computer and System Sciences 50.3, 551-573.* 1995

[20] Helmbold, David P., Jyrki Kivinen, and Manfred K. Warmuth. "Relative loss bounds for single neurons." *IEEE Transactions on Neural Networks 10.6: 1291-1304.* 1999

[21] Joulin, Armand, *et al.*. "Bag of Tricks for Efficient Text Classification." *arXiv preprint arXiv:1607.01759.* 2016

[22] Lai, Matthew. "Giraffe: Using deep reinforcement learning to play chess." *arXiv preprint arXiv:1509.01549.* 2015

[23] LeCun, Yann, *et al.*. "Handwritten digit recognition with a back-propagation network." *In Advances in neural information processing systems (pp. 396-404).* 1990

[24] Lecun, Yann, and Corinna Cortes. The MNIST database of handwritten digits.

[25] Li, Yi. "Selective voting for perceptron-like online learning." *ICML 17.* 2000

[26] Littlestone, Nick. "From on-line to batch learning." *COLT 2, pages 269–284.* July 1989

[27] Rieck, Konrad, *et al.*. "Learning and classification of malware behavior." *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (pp. 108-125). Springer Berlin Heidelberg.* 2008

[28] Rosenblatt, Frank. "The perceptron: a probabilistic model for information storage and organization in the brain." *Psychological review 65.6:386.* 1958

[29] Samuel, Arthur L. "Some studies in machine learning using the game of checkers." *IBM Journal of research and development 3.3:210-229.* 1959

[30] Senator, Ted E., *et al.*. "Financial Crimes Enforcement Network AI System (FAIS) Identifying Potential Money Laundering from Reports of Large Cash Transactions." *AI magazine 16.4: 21.* 1995

[31] Tesauro, Gerald. "TD-Gammon, a self-teaching backgammon program, achieves master-level play." *Neural computation 6.2: 215-219.* 1994

[32] Tesauro, Gerald. "Temporal difference learning and TD-Gammon." *Communications of the ACM 38.3: 58-68.* 1995

[33] Tesauro, Gerald Programming backgammon using self-teaching neural nets Artificial Intelligence 134.1: 181-1992002

[34] The MathWorks, Inc., Natick, Massachusetts, United States. "Matlab." . 2009

[35] Vapnik, Vladimir, and Alexey Chervonenkis. "A note on one class of perceptrons." *Automation and remote control 25.1: 103.* 1964

[36] Viola, Paul, and Michael Jones. "Rapid object detection using a boosted cascade of simple features." *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on. Vol. 1. IEEE.* 2001

8 Appendix

In this appendix, we present some definitions, lemmas and theorems that are used in the thesis.

Theorem 4. ***(Jensen's inequality)*** *Let $(\Omega, \mathcal{F}, \mathbb{P})$ be a probability space. Let $X : \Omega \to \{x_i \in \mathbb{R} | 1 \leq i \leq n\}$ be a real valued random variable. Note that this means that X can take n different values. Let $D \subset \mathbb{R}$ be any interval that contains $\{x_i : i \leq n\}$ and let $f : D \to \mathbb{R}$ be convex. Then*

$$f(E[X]) \leq E[f(X)].$$

Proof. By the convexity of f

$$f(\alpha x_i + (1-\alpha)x_j) \leq \alpha f(x_i) + (1-\alpha)f(x_j) \tag{6}$$

for all x_i, x_j and $0 \leq \alpha \leq 1$.

Denote $p_i := P(X = x_i)$. Note that $\sum_i p_i = 1$ We first prove the case where $n = 2$ and then we prove the general case by induction over n. In the first case $p_2 = 1 - p_1$ and the inequality follows directly from the definition of convexity:

$$f(E[X]) = f(p_1 x_1 + p_2 x_2) \leq p_1 f(x_1) + p_2 f(x_2) = E[f(X)].$$

Next we assume that the theorem holds when $n = k - 1$ and we prove the theorem for $n = k$. Now

$$
\begin{aligned}
f(E[X]) &= f\left(\sum_{i=1}^{k} p_i x_i\right) \\
&= f\left(\sum_{i=1}^{k-1} p_i x_i + p_k x_k\right) \\
&= f\left(p_k x_k + (1-p_k)\sum_{i=1}^{k-1} \frac{p_i x_i}{1-p_k}\right) \\
&\leq p_k f(x_k) + (1-p_k) f\left(\sum_{i=1}^{k-1} \frac{p_i x_i}{1-p_k}\right) \\
&= p_k f(x_k) + (1-p_k) f\left(\sum_{i=1}^{k-1} \frac{p_i}{1-p_k} x_i\right). \\
&\leq p_k f(x_k) + (1-p_k) \sum_{i=1}^{k-1} \frac{p_i}{1-p_k} f(x_i) \\
&= p_k f(x_k) + \sum_{i=1}^{k-1} p_i f(x_i) \\
&= \sum_{i=1}^{k} p_i f(x_i) \\
&= E[f(X)].
\end{aligned}
$$

□

Theorem 5. ***(Markov's inequality)*** *If X is a nonnegative random variable, then for all $a > 0$,*

$$P(X \geq a) \leq \frac{E[X]}{a}.$$

Proof. Let a be arbitrary real number**.** We can calculate and estimate the expectation of X as follows

$$\begin{aligned} E[X] &= \int_0^\infty xp(x)dx \\ &= \int_0^a xp(x)dx + \int_a^\infty xp(x)dx \\ &\geq \int_a^\infty xp(x)dx \\ &\geq a\int_a^\infty p(x)dx \\ &= aP(X \geq a), \end{aligned}$$

and dividing the equation by a yields the result. □

Lemma 4. *Let $Y \in [-1, 1]$ be a random variable and $E[Y] = 0$. Then for all $t \geq 0$ holds*

$$E[e^{tY}] \leq e^{\frac{t^2}{2}}.$$

Proof. By convexity for all $y \in [-1, 1]$ holds $e^{ty} \leq \frac{1}{2}(1+y)e^t + \frac{1}{2}(1-y)e^{-t}$**.** Now we can take expectations and use power series to obtain our result

$$\begin{aligned} E[e^{tY}] &\leq \frac{1}{2}(e^t + e^{-t}) \\ &= \frac{1}{2}\left[\left(\sum_{i=0}^{\infty} \frac{t^i + (-t)^i}{i!}\right)\right] \\ &= \sum_{i=0}^{\infty} \frac{t^{2i}}{(2i)!} \\ &\leq \sum_{i=0}^{\infty} \frac{t^{2i}}{2^i i!} \\ &= \sum_{i=0}^{\infty} \frac{(t^2/2)^i}{i!} = e^{\frac{t^2}{2}}. \end{aligned}$$

□

Definition 2. *Conditional expectation* of a discrete random variable Y with respect to an event Ψ is defined by

$$E[Y \mid \Psi] = \sum_y yP(Y = y \mid \Psi).$$

Lemma 5. *Let X and Y be random variables. Let f and g be functions. Then we have*

1. $E[X] = E[E[X \mid Y]]$.
2. $E[E[f(X) \cdot g(X,Y) \mid X = x]] = E[f(X) \cdot E[g(X,Y) \mid X = x]]$.

Proof. 1. We note that $E[X \mid Y]$ is a function of Y and that $E[X] = \sum_y E[X \mid Y = y]P(Y = y)$. Now we can calculate

$$E[E[X \mid Y]] = \sum_y E[X \mid Y = y]P(Y = y) = E[X].$$

2. Conditioning on any $X = a$, the function $f(X) = f(a)$ is a constant and we can take it outside the expectation. So for any value $X = a$ we get

$$\begin{aligned} E[E[f(X)g(X,Y) \mid X = a]] &= E[E[f(a) \cdot g(X,Y) \mid X = a]] \\ &= E[f(a) \cdot E[g(X,Y) \mid X = a]]. \end{aligned}$$

this holds for any value a.

□