Thesis

The Multiplicative Weights Method in Quantum Computing

Freek Henstra

Supervised by:
Ronald de Wolf,
Erik Jan van Leeuwen,
Tristan van Leeuwen

February, 2023

Utrecht University

Contents

Chapter 1

Introduction

Meta-algorithms like dynamic programming and divide-and-conquer provide a general template for a large variety of algorithms. A lesser-known meta-algorithm is the multiplicative weights method, in which a set of weights is maintained that are updated iteratively by the multiplicative update rule. Algorithms with this general structure have been independently developed across many fields and for many applications. Even the analyses tend to follow a similar pattern. A survey by Arora, Hazan and Kale [6] compiles many such instances.

As the development of quantum computers progresses, the question of what improvements quantum computing can offer over classical computing becomes ever more relevant. Many quantum algorithms have been developed that significantly outperform the best possible classical algorithms. The most impressive example is Shor's algorithm [20], which provides an exponential speed-up for integer factorization. This threatens the security of RSA encryption, which relies on the complexity of integer factorization.

A few recent results [16, 4, 5] suggest that algorithms that follow the multiplicative weights method are well-suited for quantization. These results use quantum subroutines to speed up classical multiplicative weights algorithms. This space of quantum multiplicative weights algorithms is still fairly unexplored. The goal of this thesis is to explore this space further, in hopes of finding new quantum speed-ups.

In Chapter 2, we will explain the multiplicative weights method and several variants of it. Chapter 3 will give a brief overview of quantum computing and some of the subroutines we will come across in this thesis. In Chapters 4 and 5, we explore how the multiplicative weights method and quantum computing are used in specific applications, including some new results. In Chapter 4 we will take a look at various boosting algorithms, which are used in the field of machine learning. In Chapter 5, we take a look at solvers for linear programs and the more general semidefinite programs. Chapter 6 concludes the thesis.

Chapter 2

The multiplicative weights method

2.1 Stock market example

In order to best illustrate the setting for the multiplicative weights method, we will consider the example of a simplified stock market from the survey by Arora et al. [6]. Suppose that we are following a single stock of interest. At the end of each day, the price of the stock either increases or decreases. At the start of the trading day, we can make a prediction of which of the two will happen. If we predict this correctly, we make a profit. If our prediction is wrong, however, we make a loss.

Without any information, we can only guess what will happen. Luckily, there are n experts, each of which publicly makes their own prediction of whether the stock price will increase or decrease on that day. We might be able to use the predictions of these experts to make a more informed prediction ourselves.

Each day goes as follows. First, the n experts make their predictions. Next, we make a prediction. Finally, it is revealed what happened to the stock price. Based on our prediction and the behavior of the stock, we either make or lose money. We repeat this process for T days. Our goal is to maximize our profit. In other words, we want to maximize the number of correct predictions over all T days.

A simple strategy would be to always follow the prediction made by the majority of experts on that day, but this does not necessarily lead to correct predictions. There are no guarantees on the accuracy of the experts' predictions. In fact, it is perfectly possible for an expert to get every single prediction wrong. If this holds for the majority of experts, the above strategy also gets every prediction wrong.

As the days go on, we learn more about the experts. For each expert, we know exactly how many of their predictions were correct. In order to find a better strategy, we need to use this knowledge for future predictions. It makes sense to believe more in the experts that have been the most successful thus far: we give their predictions more weight.

This is the concept at the heart of the multiplicative weights method. We assign a weight to each of the experts. At first, each of the experts has the same weight. At the end of each day, when we find out what happened to the stock price, we increase the weight of each expert that predicted it correctly, while decreasing the weight of each expert that was incorrect. We can then base our prediction on the weighted majority.

This approach seems like an improvement over the previous strategy: even when a majority of experts has bad predictions, we gravitate towards the "good" experts. However, this approach also has weaknesses. The "good" experts are in no way guaranteed to continue giving good predictions. We can really only know which experts are good when all T days have passed. Until then, the experts' performance can be misleading. As such, this strategy will not necessarily point us in the right direction.

As a matter of fact, both the experts' predictions and the change in stock price can be chosen adversarially. Although this should not occur in an actual stock market, an adversary could manipulate the stock in such a way that we never get any prediction correct. It seems impossible to have any guarantees on our performance when such scenarios are possible.

Despite these apparent weaknesses, the multiplicative weights method is surprisingly effective. The catch is that we do not measure the performance in a vacuum, but we compare it to that of the best expert. As we will see in the next section, the multiplicative weights method does not perform much worse than the best expert in hindsight. This means that an adversary cannot ensure we perform badly, without also making every expert perform nearly as badly. This is a surprisingly strong result, considering how easy it is for an adversary to just make all our predictions wrong.

2.2 General setting

In the general setting, it is not entirely accurate to use the term "expert". As we will see in the applications discussed later on, the notion of having access to various expert opinions is usually missing entirely. Instead, we will use the term "constraint", which generalizes much better to other applications. In the stock market example, the analysis will show that we do not perform much worse than the best expert. Equivalently, we can say that it holds for every expert that our performance is not much worse than theirs. This can be seen as a set of n constraints that we would like to satisfy, one for each expert.

As in the stock market example, we have n constraints and we run the algorithm for T iterations. In the stock market example, the result of each iteration was binary: we either predict the stock correctly or incorrectly. In the general setting, however, the result of one iteration t is given by a cost vector $m^{(t)} \in [-1,1]^n$, which assigns a cost $m_i^{(t)}$ to each constraint i. We keep track of the weights in a weight vector $w^{(t)} \in \mathbb{R}^n_{\geq 0}$. These weights are used to compute the probability distribution $p^{(t)} = w^{(t)}/\Phi^{(t)}$, where $\Phi^{(t)} = \sum_{i=1}^n w_i^{(t)}$.

Since each constraint has its own cost, we do not have a binary choice like before. Instead, we choose a constraint i and incur the same cost $m_i^{(t)}$ in iteration t. This choice is made at random based on the distribution $p^{(t)}$. In order to still fit this into a stock market setting, we can assume there are multiple stocks of interest, and the experts each recommend a certain investment strategy. The cost $m_i^{(t)}$ is then the loss made from the investment if the strategy of expert i is followed, which is negative if profit is made.

Unfortunately, even this is not general enough. In many applications, the distribution $p^{(t)}$ is not used to choose one of the constraints at random, but it is instead fed into an "oracle" once per iteration. An oracle is a subroutine that usually outputs some intermediate solution. The type of solution depends on the application. The purpose of the distribution $p^{(t)}$ is generally to guide the oracle towards certain constraints, particularly those with higher weights. As magical as it sounds, this oracle is usually a relatively simple subroutine that solves a much easier problem than the full algorithm is intended to solve.

In the stock market example, all the "oracle" has to do is pick one of the experts. Note that this choice does not impact the cost vector $m^{(t)}$ in any way, only the cost you incur. In the general setting, however, the result of the oracle can, and usually does, impact the cost vector $m^{(t)}$.

2.3 Algorithm & analysis

Algorithm 2.1: The multiplicative weights method

```
1 Fix $\eta \le \frac{1}{2}$. Set weight $w_i^{(1)} := 1, \forall i \in [n]$.
2 for $t = 1, 2, \dots, T$ do
3     Feed into the oracle the distribution $p^{(t)} = w^{(t)}/\Phi^{(t)}$, where
      $\Phi^{(t)} = \sum_{i=1}^n w_i^{(t)}$.
4     Observe cost vector $m^{(t)} \in [-1, 1]^n$.
5     Update weights: $w_i^{(t+1)} := w_i^{(t)} \exp(-\eta m_i^{(t)}), \forall i \in [n]$.
```

Algorithm 2.1 is our generalization of the multiplicative weights method. In particular, it is the Hedge algorithm by Freund and Schapire [13]. The standard multiplicative weights method in [6] uses a different update rule, namely $w_i^{(t+1)} := w_i^{(t)}(1 - \eta m_i^{(t)})$. For small η, the difference between the two update rules is small, but it does lead to a different analysis and a slightly different theorem. Most applications we will look at, use the update rule of the Hedge algorithm, so we will view that as the standard form.

One change we have made to the formulation of the algorithm compared to [6] is that we refer the oracle instead of sticking to the stock market example where a decision is made in each iteration. This allows most of the applications to more closely fit this framework.

In step 5 of the algorithm, we see the multiplicative update rule, which is where the "multiplicative" in the name comes from. Each weight $w_i^{(t)}$ is multiplied by a factor $\exp(-\eta m_i^{(t)})$. The parameter $\eta \leq \frac{1}{2}$ determines by how much the weights change each iteration. We will see later how it can be chosen optimally. As expected, a positive loss decreases the weight while a negative loss increases the weight.

While the algorithm does not specify a final output, the weight vector $w^{(t)}$ can be seen as the output of iteration t, often alongside an output from the oracle. Usually, the final output is some combination of the intermediate solutions returned by the oracle over all T iterations. In the stock market example, we have a separate output for every iteration, namely the expert we choose.

Theorem 2.1 gives us the n constraints that are satisfied after all T iterations of the multiplicative weights method, with regards to the cost vectors $m^{(t)}$ and distributions $p^{(t)}$.

Theorem 2.1 (Theorem 2.3 in [6])**.** *The multiplicative weights method guarantees that after T rounds, for every $i \in [n]$, we have*

$$\sum_{t=1}^{T} m^{(t)} \cdot p^{(t)} \leq \sum_{t=1}^{T} m_i^{(t)} + \eta \sum_{t=1}^{T} (m^{(t)})^2 \cdot p^{(t)} + \frac{\ln n}{\eta},$$

where $(m^{(t)})^2$ is the vector obtained by taking the coordinate-wise square of $m^{(t)}$.

Proof. The proof follows from analyzing the sum of weights $\Phi^{(t)}$. In particular, we will give an upper bound and n lower bounds for $\Phi^{(T+1)}$. Combining these bounds will result in the inequalities in the theorem. We use the fact that $\exp(-\eta x) \leq 1 - \eta x + \eta^2 x^2$ if $|\eta x| \leq 1$. We obtain

$$\begin{aligned}
\Phi^{(t+1)} &= \sum_{i=1}^{n} w_i^{(t+1)} \\
&= \sum_{i=1}^{n} w_i^{(t)} \exp(-\eta m_i^{(t)}) \\
&\leq \Phi^{(t)} \sum_{i=1}^{n} p_i^{(t)} (1 - \eta m_i^{(t)} - \eta^2 (m_i^{(t)})^2) \\
&= \Phi^{(t)} \left(1 - \eta m^{(t)} \cdot p^{(t)} + \eta^2 (m^{(t)}) \cdot p^{(t)}\right) \\
&\leq \Phi^{(t)} \exp\left(-\eta m^{(t)} \cdot p^{(t)} + \eta^2 (m^{(t)}) \cdot p^{(t)}\right).
\end{aligned}$$

For the last inequality, we use the fact that $1 + x \leq e^x = \exp(x)$ for all x. By induction, after T iterations, we have

$$\begin{aligned}\Phi^{(T+1)} &\leq \Phi^{(1)} \prod_{t=1}^{T} \exp\left(-\eta m^{(t)} \cdot p^{(t)} + \eta^2 (m^{(t)}) \cdot p^{(t)}\right) \\ &= n \cdot \exp\left(-\eta \sum_{t=1}^{T} m^{(t)} \cdot p^{(t)} + \eta^2 \sum_{t=1}^{T} (m^{(t)}) \cdot p^{(t)}\right),\end{aligned}$$

which is the upper bound. For the lower bounds, we have for all constraints $i \in [n]$,

$$\Phi^{(T+1)} \geq w_i^{(T+1)} = \prod_{t=1}^{T} \exp(-\eta m_i^{(t)}) = \exp\left(-\eta \sum_{t=1}^{T} m_i^{(t)}\right).$$

Taking the logarithm and dividing by η in the upper bound and the lower bounds, we get, for all $i \in [n]$,

$$-\sum_{t=1}^{T} m_i^{(t)} \leq \frac{\ln n}{\eta} - \sum_{t=1}^{T} m^{(t)} \cdot p^{(t)} + \eta \sum_{t=1}^{T} (m^{(t)})^2 \cdot p^{(t)},$$

and thus

$$\sum_{t=1}^{T} m^{(t)} \cdot p^{(t)} \leq \sum_{t=1}^{T} m_i^{(t)} + \eta \sum_{t=1}^{T} (m^{(t)})^2 \cdot p^{(t)} + \frac{\ln n}{\eta}.$$

□

The interpretation of this theorem changes drastically depending on the setting. For now, we will assume the stock market example, which allows for a fairly simple interpretation.

Note that for all iterations t, the expected cost is $m^{(t)} \cdot p^{(t)}$. This means the left-hand side is simply the total expected cost over all T iterations. The theorem therefore gives n upper bounds for the total cost, which is great since minimizing it was our goal.

On the right-hand side, the first term is the total cost of expert i. The last two terms together form the loss, the amount by which we do worse than the i-th expert. Note that we can pick η such that this loss is minimal. For example, in the case of binary costs $m^{(t)} \in \{-1, 1\}^n$ for all iterations t, we minimize the loss by picking $\eta = \sqrt{\ln(n)/T}$, after which the loss becomes $2\sqrt{T \ln n}$.

Note that the inequality holds for all experts i, so this even holds for the best expert: the one with the lowest total cost. The loss grows sublinearly with respect to the number of iterations T. This means that the loss per iteration goes to 0 with T.

All in all, this theorem implies that you will not perform much worse than even the best expert, despite not knowing who the best expert is until most (if not all) iterations have passed and despite the fact that an adversary can choose the cost vectors. With such an oddly strong result, it is not too surprising that this meta-algorithm has been rediscovered as many times as it has, especially considering its simplicity.

2.4 Alternative update rule

As we mentioned previously, the standard multiplicative weights method in [6] uses a different update rule.

Algorithm 2.2: The multiplicative weights method

```
1 Fix $\eta \leq \frac{1}{2}$. Set weight $w_i^{(1)} := 1, \forall i \in [n]$.
2 for $t = 1, 2, \ldots, T$ do
3     Feed into the oracle the distribution $p^{(t)} = w^{(t)}/\Phi^{(t)}$, where
      $\Phi^{(t)} = \sum_{i=1}^n w_i^{(t)}$.
4     Observe cost vector $m^{(t)} \in [-1, 1]^n$.
5     Update weights: $w_i^{(t+1)} := w_i^{(t)}(1 - \eta m_i^{(t)}), \forall i \in [n]$.
```

We see the different update rule in line 5, where instead of multiplying the previous weight by the exponential factor $\exp(-\eta m_i^{(t)})$, we multiply it by the additive factor $(1-\eta m_i^{(t)})$. While this difference is small for small η, the resulting theorem does differ slightly. There is one instance later on where we need this specific theorem.

Theorem 2.2 (Theorem 2.1 in [6])**.** *The multiplicative weights method guarantees that after T rounds, for every $i \in [n]$, we have*

$$\sum_{t=1}^T m^{(t)} \cdot p^{(t)} \leq \sum_{t=1}^T m_i^{(t)} + \eta \sum_{t=1}^T |m_i^{(t)}| + \frac{\ln n}{\eta}.$$

Proof. The proof follows from analyzing the sum of weights $\Phi^{(t)}$. In particular, we will give an upper bound and n lower bounds for $\Phi^{(T+1)}$. Combining these bounds will result in the inequalities in the theorem. First, we have

$$\begin{aligned}
\Phi^{(t+1)} &= \sum_{i=1}^n w_i^{(t+1)} \\
&= \sum_{i=1}^n w_i^{(t)}(1 - \eta m_i^{(t)}) \\
&= \sum_{i=1}^n w_i^{(t)} - \sum_{i=1}^n \eta w_i^{(t)} m_i^{(t)} \\
&= \Phi^{(t)} - \sum_{i=1}^n \eta p_i^{(t)} \Phi^{(t)} m_i^{(t)} \\
&= \Phi^{(t)} - \eta \Phi^{(t)} \sum_{i=1}^n m_i^{(t)} p_i^{(t)} \\
&= \Phi^{(t)}(1 - \eta m^{(t)} \cdot p^{(t)}) \\
&\leq \Phi^{(t)} \exp(-\eta m^{(t)} \cdot p^{(t)}).
\end{aligned}$$

For the last inequality, we use the fact that $1 + x \leq e^x = \exp(x)$ for all x. By induction, after T iterations, we have

$$\Phi^{(T+1)} \leq \Phi^{(1)} \exp\left(-\eta \sum_{t=1}^{T} m^{(t)} \cdot p^{(t)}\right) = n \cdot \exp\left(-\eta \sum_{t=1}^{T} m^{(t)} \cdot p^{(t)}\right),$$

which is the upper bound. Next, we use the following facts, which follow from the convexity of the exponential function:

$$\begin{aligned}(1-\eta)^x &\leq (1-\eta x) \text{ if } x \in [0,1],\\ (1+\eta)^{-x} &\leq (1-\eta x) \text{ if } x \in [-1,0].\end{aligned}$$

Since the costs $m_i^{(t)} \in [-1,1]$ are bounded, we have for all constraints $i \in [n]$,

$$\Phi^{(T+1)} \geq w_i^{(T+1)} = \prod_{t=1}^{T}(1-\eta m_i^{(t)}) \geq (1-\eta)^{\sum_{t:m_i^{(t)}\geq 0} m_i^{(t)}} \cdot (1+\eta)^{-\sum_{t:m_i^{(t)}<0} m_i^{(t)}}.$$

These are the n lower bounds. Taking logarithms in the upper bound and the lower bounds, we get, for all $i \in [n]$,

$$\begin{aligned}\ln n - \eta \sum_{t=1}^{T} m^{(t)} \cdot p^{(t)} &\geq \sum_{t:m_i^{(t)}\geq 0} m_i^{(t)} \ln(1-\eta) - \sum_{t:m_i^{(t)}<0} m_i^{(t)} \ln(1+\eta),\\ -\eta \sum_{t=1}^{T} m^{(t)} \cdot p^{(t)} &\geq \sum_{t:m_i^{(t)}\geq 0} m_i^{(t)} \ln(1-\eta) - \sum_{t:m_i^{(t)}<0} m_i^{(t)} \ln(1+\eta) - \ln n,\\ \sum_{t=1}^{T} m^{(t)} \cdot p^{(t)} &\leq \frac{1}{\eta} \sum_{t:m_i^{(t)}\geq 0} m_i^{(t)} \ln \frac{1}{1-\eta} + \frac{1}{\eta} \sum_{t:m_i^{(t)}<0} m_i^{(t)} \ln(1+\eta) + \frac{\ln n}{\eta}.\end{aligned}$$

Next, we use the facts that

$$\begin{aligned}\ln\left(\frac{1}{1-\eta}\right) &\leq \eta + \eta^2,\\ \ln(1+\eta) &\geq \eta - \eta^2\end{aligned}$$

for $\eta \leq \frac{1}{2}$. We get, for all $i \in [n]$,

$$\begin{aligned}
\sum_{t=1}^{T} m^{(t)} \cdot p^{(t)} &\leq \frac{1}{\eta} \sum_{t: m_i^{(t)} \geq 0} m_i^{(t)} \ln \frac{1}{1-\eta} + \frac{1}{\eta} \sum_{t: m_i^{(t)} < 0} m_i^{(t)} \ln(1+\eta) + \frac{\ln n}{\eta} \\
&\leq \frac{1}{\eta} \sum_{t: m_i^{(t)} \geq 0} m_i^{(t)} (\eta + \eta^2) + \frac{1}{\eta} \sum_{t: m_i^{(t)} < 0} m_i^{(t)} (\eta - \eta^2) + \frac{\ln n}{\eta} \\
&= \sum_{t=1}^{T} m_i^{(t)} + \eta \sum_{t: m_i^{(t)} \geq 0} m_i^{(t)} - \eta \sum_{t: m_i^{(t)} < 0} m_i^{(t)} + \frac{\ln n}{\eta} \\
&= \sum_{t=1}^{T} m_i^{(t)} + \eta \sum_{t=1}^{T} |m_i^{(t)}| + \frac{\ln n}{\eta}.
\end{aligned}$$

□

2.5 Matrix multiplicative weights

Another notable variation is the matrix multiplicative weights method. In this variant, we keep track of a weight matrix instead of simply a vector of weights.

Algorithm 2.3: The matrix multiplicative weights algorithm

```
1 Fix $\eta \leq \frac{1}{2}$. Set weight matrix $W^{(1)} := I_n$.
2 for $t = 1, 2, \ldots, T$ do
3     Feed into the oracle the density matrix $\rho^{(t)} = W^{(t)}/\Phi^{(t)}$, where
        $\Phi^{(t)} = \mathrm{Tr}\left(W^{(t)}\right)$.
4     Observe cost matrix $M^{(t)}$.
5     Update weight matrix: $W^{(t+1)} := \exp(-\eta \sum_{\tau=1}^{t} M^{(\tau)})$.
```

A real symmetric matrix X is *positive semidefinite* if all its eigenvalues are non-negative, which implies that $v^\top X v \geq 0$ for any real vector v. Note that we can extend this definition to complex Hermitian matrices, but for the remainder of this thesis we will consider only real matrices. A *density matrix* is a positive semidefinite, symmetric matrix of trace one.

We assume all cost matrices $M^{(t)} \in \mathbb{R}^{n \times n}$ are symmetric and have eigenvalues in $[-1, 1]$. In the multiplicative weights method, the cost vector assigned a cost to each constraint. In the matrix multiplicative weights method, a unit vector $v \in \mathbb{S}^{n-1}$ is assigned cost $v^\top M^{(t)} v \in [-1, 1]$ in iteration t. The exponent in the update rule is the matrix exponential $e^X = \sum_{k=0}^{\infty} \frac{1}{k!} X^k$. While it shares many properties with the regular exponential function, it does not generally hold that $\exp(X + Y) = \exp(X)\exp(Y)$. This is also why the update rule is not written recursively. The following result does hold.

Theorem 2.3 (Golden-Thompson inequality [12]). *For symmetric matrices X, Y, it holds that*

$$Tr\left(\exp(X+Y)\right) \leq Tr\left(\exp(X)\exp(Y)\right)$$

The matrix exponential is always positive semidefinite if the matrix in the exponent is symmetric. Since $\rho^{(t)}$ is the normalized weight matrix $W^{(t)}$, it follows that $\rho^{(t)}$ is indeed a density matrix.

Theorem 2.4 (Theorem 5.1 in [6]). *The matrix multiplicative weights method guarantees that after T rounds, for all $v \in \mathbb{S}^{n-1}$, we have*

$$\sum_{t=1}^{T} Tr\left(M^{(t)}\rho^{(t)}\right) \leq \sum_{t=1}^{T} v^\top M^{(t)} v + \eta \sum_{t=1}^{T} Tr\left((M^{(t)})^2\rho^{(t)}\right) + \frac{\ln n}{\eta}.$$

Proof. The proof follows from analyzing the sum of weights $\Phi^{(t)}$. In particular, we will give an upper bound and n lower bounds for $\Phi^{(T+1)}$. Combining these bounds will result in the inequalities in the theorem. Using the Golden-Thompson inequality, we obtain

$$\begin{aligned}
\Phi^{(t+1)} &= \mathrm{Tr}\left(W^{(t+1)}\right) \\
&= \mathrm{Tr}\left(\exp\left(-\eta\sum_{\tau=1}^{t} M^{(\tau)}\right)\right) \\
&= \mathrm{Tr}\left(\exp\left(-\eta\sum_{\tau=1}^{t-1} M^{(\tau)} - \eta M^{(t)}\right)\right) \\
&\leq \mathrm{Tr}\left(W^{(t)}\exp(-\eta M^{(t)})\right) \\
&= \mathrm{Tr}\left(W^{(t)} - W^{(t)}(I - \exp(-\eta M^{(t)}))\right) \\
&= \Phi^{(t)} - \Phi^{(t)}\mathrm{Tr}\left(\rho^{(t)}(I - \exp(\eta M^{(t)}))\right) \\
&= \Phi^{(t)}\left(1 - \mathrm{Tr}\left(\rho^{(t)}(I - \exp(\eta M^{(t)}))\right)\right) \\
&\leq \Phi^{(t)}\exp\left(-\mathrm{Tr}\left(\rho^{(t)}(I - \exp(\eta M^{(t)}))\right)\right).
\end{aligned}$$

For the last inequality, we used the fact that $1 + x \leq e^x = \exp(x)$ for all x. We define a rest matrix R via

$$I - \exp(-\eta M^{(t)}) = \eta M^{(t)} - \eta^2(M^{(t)})^2 + R,$$

which means that $R = I - \exp(-\eta M^{(t)}) - \eta M^{(t)} + \eta^2(M^{(t)})^2$. We use the fact that $\exp(-\eta X) \preceq I - \eta X + \eta^2 X^2$ if $||\eta X|| \leq 1$, which implies that $R \succeq 0$.

Substituting this and using induction, after T iterations, we have

$$\begin{aligned}\Phi^{(T+1)} &\leq \Phi^{(1)} \prod_{t=1}^{T} \exp\left(-\mathrm{Tr}\left(\rho^{(t)}(\eta M^{(t)} - \eta^2 (M^{(t)})^2)\right) - \mathrm{Tr}\left(\rho^{(t)} R\right)\right) \\ &\leq n \cdot \exp\left(-\sum_{t=1}^{T} \mathrm{Tr}\left(\rho^{(t)}(\eta M^{(t)} - \eta^2 (M^{(t)})^2)\right)\right).\end{aligned}$$

For the last inequality, we removed the term $\mathrm{Tr}\left(p^{(t)} R\right)$ using the fact that $\mathrm{Tr}\,(XY) \geq 0$ if $X, Y \succeq 0$. We now have the upper bound and move on to the lower bounds. Let $\lambda_1(X) \geq \ldots \geq \lambda_n(X)$ denote the eigenvalues of X in decreasing order. Note that $\mathrm{Tr}\,(\exp(X)) = \sum_{k=1}^{n} \exp(\lambda_k(X)) \geq \exp(\lambda_1(X))$. We have

$$\begin{aligned}\Phi^{(T+1)} &= \mathrm{Tr}\left(W^{(T+1)}\right) \\ &= \mathrm{Tr}\left(\exp\left(-\eta \sum_{t=1}^{T} M^{(t)}\right)\right) \\ &\geq \exp\left(\lambda_1\left(-\eta \sum_{t=1}^{T} M^{(t)}\right)\right).\end{aligned}$$

Now, for all unit vectors $v \in \mathbb{S}^{n-1}$, it holds that

$$\lambda_1\left(-\eta \sum_{t=1}^{T} M^{(t)}\right) \geq v^\top \left(-\eta \sum_{t=1}^{T} M^{(t)}\right) v = -\eta \sum_{t=1}^{T} v^\top M^{(t)} v.$$

Hence, for all unit vectors $v \in \mathbb{S}^{n-1}$, we obtain the lower bound

$$\Phi^{(T+1)} \geq \exp\left(-\eta \sum_{t=1}^{T} v^\top M^{(t)} v\right).$$

Taking the logarithm and dividing by η in the upper bound and the lower bounds, we get, for all unit vectors $v \in \mathbb{S}^{n-1}$,

$$-\sum_{t=1}^{T} v^\top M^{(t)} v \leq \frac{\ln n}{\eta} - \sum_{t=1}^{T} \mathrm{Tr}\left(\rho^{(t)}(M^{(t)} - \eta (M^{(t)})^2)\right),$$

and thus

$$\sum_{t=1}^{T} \mathrm{Tr}\left(M^{(t)} \rho^{(t)}\right) \leq \sum_{t=1}^{T} v^\top M^{(t)} v + \eta \sum_{t=1}^{T} \mathrm{Tr}\left((M^{(t)})^2 \rho^{(t)}\right) + \frac{\ln n}{\eta}.$$

□

Chapter 3

Quantum Computing

There are many quantum algorithms with better running times than possible with classical computers. By using such algorithms as subroutines within multiplicative weights, as well as some other techniques, quantum speed-ups can be achieved. This section aims to give a brief overview of quantum computing, starting with preliminaries and followed up by various relevant subroutines. We will mostly follow the content of the course Quantum Computing by de Wolf [22]. For a more complete overview, we refer to these lecture notes.

3.1 Preliminaries

In a classical computer, a system can be in one state at a time. Suppose we have two bits, then we have four possible states: $|00\rangle, |01\rangle, |10\rangle, |11\rangle$. We can view these as integers and write them as $|0\rangle, |1\rangle, |2\rangle, |3\rangle$ instead, respectively. In a quantum computer, a *pure quantum state* $|\phi\rangle$ that uses two qubits (quantum bits) can assume a superposition of these classical states:

$$|\phi\rangle = \alpha_0 |0\rangle + \alpha_1 |1\rangle + \alpha_2 |2\rangle + \alpha_3 |3\rangle ,$$

where each α_i is a complex number known as the amplitude of $|i\rangle$. Due to this, even a pure quantum state of only one qubit could assume infinitely many different superpositions. This sounds very promising, but there is one major drawback: when a quantum state is measured, it collapses back into a classical state. If a measurement is done in the computational basis, it collapses into state $|i\rangle$ with probability $|\alpha_i|^2$.

If we have two quantum systems with Hilbert spaces $\mathcal{H}_1$ and $\mathcal{H}_2$, then we can combine the quantum systems by taking the tensor product of the two spaces: $\mathcal{H}_1 \otimes \mathcal{H}_2$. For states, the tensor symbol can often be left out. For example, the classical states $|1\rangle \otimes |1\rangle$, $|1\rangle |1\rangle$, $|11\rangle$ and $|3\rangle$ are all equal.

If a quantum state $|\phi\rangle$ can be written as the tensor product of two states $|\phi\rangle = |\phi_1\rangle \otimes |\phi_2\rangle$, then it is *separable*. Otherwise, it is called *entangled*. An example of an entangled state is $|\phi\rangle = \frac{1}{\sqrt{2}} |0\rangle |0\rangle + \frac{1}{\sqrt{2}} |1\rangle |1\rangle$. If we measure the qubit in the first register, the outcome could be 0 or 1, each with probability 1/2. If the outcome is 0, then the state collapses to $|0\rangle|0\rangle$. Hence, this guarantees that measuring the qubit in the second register will also have outcome 0. If a state is separable, then we can measure one half without affecting the other half in any way.

We can apply various unitary operations without the state collapsing. If we view a pure quantum state as an N-dimensional vector of amplitudes, then the unitary operation can be seen as a unitary $N \times N$ matrix that is multiplied with the vector. Smaller unitaries are known as *quantum gates*, which can be combined into a larger *quantum circuit*. The most common 1-qubit unitaries are

$$I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \quad Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}, \quad H = \tfrac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}.$$

The gate H is known as the *Hadamard transform*, and does the following.

$$H|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle),$$
$$H|1\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle).$$

Both of the above resulting states obtain outcomes 0 and 1 with equal probability when measured. However, when we apply H to the superposition $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$, we obtain

$$H\left(\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)\right) = \frac{1}{\sqrt{2}}(H|0\rangle + H|1\rangle) = \frac{1}{2}(|0\rangle + |1\rangle + |0\rangle - |1\rangle) = |0\rangle.$$

This phenomenon is known as *interference*, since the amplitudes for $|1\rangle$ cancel each other out. The Hadamard transform is commonly used in quantum algorithms due to this.

A common type of unitary operation is the query. Given some N-bit input $x = (x_0, \ldots, x_{N-1}) \in \{0,1\}^N$, where $N = 2^n$, the unitary operation

$$O_x : |i, 0\rangle \mapsto |i, x_i\rangle$$

reads the first n qubits of the quantum state, queries the input at the corresponding index, and places this in the final qubit of the state. These queries work on any pure quantum state, so we can query a superposition of indices at once, although measuring would still only result in one bit of the input.

3.2 Amplitude amplification

Suppose we have an m-qubit unitary A such that

$$A|0^m\rangle = \sqrt{p}\,|\phi_0\rangle|0\rangle + \sqrt{1-p}\,|\phi_1\rangle|1\rangle.$$

Suppose that $|\phi_0\rangle$ is the state we want to obtain, which we call the "good state", while $|\phi_1\rangle$ is the "bad state". We start with the state $|U\rangle := A|0^m\rangle$. If we measure the final qubit of $|U\rangle$ immediately, then the outcome will be 0 with probability p. If the outcome is indeed 0, we know that the remaining state is $|\phi_0\rangle$ as desired. We can simply repeat this process until we get the good state, which takes expected $O(1/p)$ attempts. With amplitude amplification [10], we can do better.

As the name implies, we can amplify the amplitude of the good state and thereby increase the probability of getting outcome 0 from the measurement. Note that due to the final qubit, the states $|G\rangle := |\phi_0\rangle|0\rangle$ and $|B\rangle := |\phi_1\rangle|1\rangle$ are orthogonal. If we view $|B\rangle$ as the x-axis and $|G\rangle$ as the y-axis, then $|U\rangle = \sqrt{p}\,|G\rangle + \sqrt{1-p}\,|B\rangle$ is in between, with an angle of $\theta = \arcsin\sqrt{p}$ between $|U\rangle$ and $|B\rangle$. If we rotate the state $|U\rangle$ towards $|G\rangle$, then we obtain a new state with a greater amplitude on $|G\rangle$.

Suppose we have a unitary R_G that puts a "$-$" in front of $|G\rangle$ but leaves $|B\rangle$ alone. This circuit can be seen as a reflection through $|B\rangle$. Next, suppose we have a unitary R_0 that puts a "$-$" in front of all basis states except for $|0^m\rangle$. Then we can implement a reflection through $|U\rangle$ as AR_0A^{-1}. The combined circuit $AR_0A^{-1}R_G$ rotates the state counterclockwise by 2θ. After k such rotations, we will have the state

$$\sin((2k+1)\theta)\,|G\rangle + \cos((2k+1)\theta)\,|B\rangle\,.$$

If $(2k+1)\theta \approx \pi/2$, then the state $|G\rangle$ will have an amplitude close to 1. To get to this point, we need $O(1/\sqrt{p})$ rotations. If we know p, we know how many rotations to use. As the result below shows, a lower bound on p is also sufficient to use amplitude amplification. This is done using decreasing guesses for the value of p.

Theorem 3.1 (Amplitude amplification, Theorem 6 in [16])**.** *Suppose we have an m-qubit unitary A such that*

$$A|0^m\rangle = \sqrt{p}\,|\phi_0\rangle|0\rangle + \sqrt{1-p}\,|\phi_1\rangle|1\rangle\,,$$

and we know a lower bound p' on p. Then there exists a quantum algorithm V using $O\left(1/\sqrt{p'}\right)$ applications of A and A^{-1}, and $\tilde{O}\left(1/\sqrt{p'}\right)$ other gates, such that

$$V|0^m\rangle = \sqrt{b}\,|\phi_0\rangle|0\rangle + \sqrt{1-b}\,|\phi_1\rangle|1\rangle\,,$$

where $b \in [1/2, 1]$.

Note that the notation $\tilde{O}(x) := O(x \cdot \mathrm{polylog}(x))$ ignores logarithmic factors. A good example of amplitude amplification is Grover's algorithm [15]. In fact, Grover's algorithm is what inspired the more general amplitude amplification.

Let $N = 2^n$ and suppose we are given an arbitrary $x \in \{0,1\}^N$. We want to find an index i such that $x_i = 1$ and if no such index exists, output "no solutions". Let t be the number of solutions in x. A classical algorithm requires $\Theta(N)$ queries, or $\Theta(N/t)$ with success probability $\geq 2/3$. In particular, if there are no solutions, then we need to query every single index to be able to guarantee this. A randomized classical algorithm

In this case, the good and bad states are

$$|G\rangle = \frac{1}{\sqrt{t}} \sum_{i:x_i=1} |i\rangle \text{ and } |B\rangle = \frac{1}{\sqrt{N-t}} \sum_{i:x_i=0} |i\rangle .$$

Measuring $|G\rangle$ would result in an outcome i such that $x_i = 1$. Note that the good and bad states do not need to end in $|0\rangle$ and $|1\rangle$ here, because the states are already orthogonal and we do not need to obtain the quantum state $|G\rangle$: a measurement is sufficient.

Grover's algorithm is a special case of amplitude amplification where $A = H^{\otimes n}$ and $R_G = O_{x,\pm}$, where $O_{x,\pm} |i\rangle = (-1)^{x_i} |i\rangle$ is a type of query for x. Grover's algorithm uses $O(\sqrt{N})$ queries in general, or $O(\sqrt{N/t})$ if t is known.

3.3 Other subroutines

Theorem 3.2 (Amplitude estimation, Theorem 2.5 in [7]). *There is a quantum algorithm $\mathcal{A}$ that satisfies the following: given access to a unitary U such that $U|0\rangle = |\psi\rangle$ where $|\psi\rangle = \sqrt{p}\,|\psi_0\rangle + \sqrt{1-p}\,|\psi_1\rangle$ and $|\psi_0\rangle, |\psi_1\rangle$ are orthogonal quantum states, $\mathcal{A}$ makes M queries to U and U^{-1} and with probability $\geq 2/3$ outputs $\tilde{p}$ such that*

$$|\tilde{p} - p| \leq 2\pi \frac{\sqrt{p(1-p)}}{M} + \frac{\pi^2}{M^2}.$$

Amplitude estimation starts with a similar setup to amplitude amplification, but instead of amplifying the amplitude, it computes an estimation of the amplitude.

Theorem 3.3 (Approximate counting, Theorem 7 in [16]). *Suppose we have query access to a string $z \in [0,1]^N$, with sum $s = \sum_{i=1}^N z_i \geq 1$. There exists a quantum algorithm that uses $O\left(\frac{1}{\varepsilon}\sqrt{N}\log(1/\delta)\right)$ queries and $\tilde{O}\left(\frac{1}{\varepsilon}\sqrt{N}\log(1/\delta)\right)$ other operations, and that outputs, with probability $\geq 1-\delta$, an $\tilde{s}$ such that $(1-\varepsilon)s \leq \tilde{s} \leq (1+\varepsilon)s$.*

With approximate counting [11], we can compute a sum approximately in sublinear time. This is a very useful for finding quantum speed-ups, since it can bypass certain linear-time steps that are unavoidable in the classical case.

Definition 3.1 (Gibbs sampler, Definition 4.11 in [2]). *A θ-precise Gibbs sampler for the input matrices $A_0, \ldots, A_m$ is a unitary that takes as input a data structure storing a vector $y \in \mathbb{R}_{\geq 0}^{m+1}$ and creates as output a θ-approximation in trace distance of the Gibbs state*

$$\exp\left(-\sum_{j=0}^m y_j A_j\right) / Tr\left(\exp\left(-\sum_{j=0}^m y_j A_j\right)\right).$$

Here, the trace distance between two density matrices ρ_1, ρ_2 is $T(\rho_1, \rho_2) = \frac{1}{2} \|\rho_1 - \rho_2\|_1$. Gibbs samplers are used to prepare quantum Gibbs states. Various different Gibbs samplers exist. The density matrix ρ in the matrix multiplicative weights algorithm is a Gibbs state, so that is one instance where it can be useful.

Theorem 3.4 (Trace estimation, Corollary 14 in [5])**.** *Let $A, H \in \mathbb{R}^{n\times n}$ be symmetric matrices such that $||A|| \leq 1$ and $||H|| \leq K$ for a known bound $K \in \mathbb{R}_+$. Assume A is s-sparse and H is d-sparse with $s \leq d$. An additive θ-approximation of*

$$Tr\,(A\rho) = \frac{A\exp(-H)}{Tr\,(\exp(-H))}$$

can be computed using $\tilde{O}\left(\frac{\sqrt{n}dK}{\theta}\right)$ queries to A and H, while using the same order of gates.

Computing this kind of trace value otherwise would require expensive matrix multiplication, so this is a fairly powerful result.

Chapter 4

Boosting

One of the application of the multiplicative weights method is boosting. In machine learning, a boosting algorithm is an algorithm that turns a "weak learner" into a "strong learner" through repeated calls of the weak learner. In order to explain this in more detail, we first need to go over some machine learning preliminaries.

In this chapter, we cover various boosting algorithms from different papers. These papers all differ in notation and some assumptions. We present these algorithms consistent with each other with our own notation and assumptions, so be aware of this when reading the referenced papers. This chapter aligns most closely with [16], which also explains most of the algorithms below.

4.1 Machine learning preliminaries

We can analyze machine learning using the PAC learning framework [21]. Suppose we have a domain $\mathcal{X}$ and there is some unknown target function $f : \mathcal{X} \to \{-1, 1\}$, that is part of a known concept class $\mathcal{C}$. For instance, the domain $\mathcal{X}$ could be the set of all images of a certain height and width, and a possible target function f could be a function that returns 1 if the image depicts a dog, and -1 otherwise. Since f has two possible outcomes, it is also called a binary classifier.

Given a sample $S = ((x_1, f(x_1)), \ldots, (x_n, f(x_n))) \in (\mathcal{X} \times \{-1, 1\})^n$ of n labeled examples generated i.i.d. from an unknown distribution $\mathcal{D}$ on $\mathcal{X}$, a learner is an algorithm that outputs a hypothesis $h : \mathcal{X} \to \{-1, 1\}$. This hypothesis is intended to be an approximation of the target function f. The set of all hypotheses that a learner can output is known as said learner's hypothesis class $\mathcal{H}$. We assume for the remainder of this chapter that any hypothesis h can be evaluated in constant time.

In order to measure how closely a hypothesis approximates a target function, we define two types of errors.

Definition 4.1. *The generalization error of a hypothesis $h : \mathcal{X} \to \{-1, 1\}$ w.r.t.*

the target function $f : \mathcal{X} \to \{-1, 1\}$ under distribution $\mathcal{D}$ is

$$err(h, f, \mathcal{D}) = \mathbb{P}_{x \sim \mathcal{D}}[h(x) \neq f(x)].$$

This is essentially the true error, and therefore the error we would like to be small. However, the learner cannot compute this error, because the distribution $\mathcal{D}$ is unknown.

Definition 4.2. *The empirical error of a hypothesis $h : \mathcal{X} \to \{-1, 1\}$ w.r.t. sample $S = ((x_1, f(x_1)), \ldots, (x_n, f(x_n)))$ is*

$$\widehat{err}(h, S) = \frac{1}{n} \sum_{i=1}^{n} [h(x_i) \neq f(x_i)].$$

The notation [.] in the summation is the Iverson bracket. It takes value 1 if the statement between the brackets is true, and value 0 otherwise. The empirical error is simply the fraction of errors that h makes on the sample set S. Unlike the generalization error, the empirical error can be computed from the sample set and h.

We want the hypothesis h to be probably approximately correct (PAC). This phrase is what the PAC-learning framework is named after. This framework is used to analyze the effectiveness of learning algorithms.

Definition 4.3. *An (ε, δ)-PAC learner for a concept class $\mathcal{C}$ with hypothesis class $\mathcal{H}$ and sample complexity n, is an algorithm that, for all target functions $f \in \mathcal{C}$ and all distributions $\mathcal{D}$ on $\mathcal{X}$, takes as input a sample $S = ((x_1, f(x_1)), \ldots, (x_n, f(x_n)))$ generated i.i.d. from $\mathcal{D}$, and outputs a hypothesis $h \in \mathcal{H}$ such that*

$$\mathbb{P}[err(h, f, \mathcal{D}) \leq \varepsilon] \geq 1 - \delta,$$

where the probability is taken over the sample and any randomness within the algorithm.

A γ-*weak learner* is a $(\frac{1}{2} - \gamma, 0)$-PAC learner, where γ is small. Setting $\delta = 0$ here is purely for convenience, but not strictly necessary. Since we are working with binary classifiers, setting $\varepsilon = \frac{1}{2} - \gamma$ means the weak learner is not necessarily much better than a learner that simply outputs a random classification. We want to turn such a weak learner into a *strong learner*, which is an (ε, δ)-PAC learner, where both ε and δ are small. A booster is an algorithm that can turn a weak learner into a strong learner.

A booster uses a weak learner $\mathcal{W}$, as well as the sample set S and should with probability $\geq 1 - \delta$ output a hypothesis with generalization error $\leq \varepsilon$. Essentially, this means that a boosting algorithm equipped with a weak learner is a strong learner. Boosting is an iterative process that repeatedly runs the weak learner. Since we only have one sample S, resampling is used to generate a new sample from the original sample S in each iteration. The distribution used for this resampling also changes in each iteration.

We can analyze boosting algorithms to show that the empirical error of their final hypothesis h on the sample S is likely to be small, but this does not guarantee that the generalization error is small as well. Intuitively, the larger the sample S is, the more representative it will be of the original distribution $\mathcal{D}$. If the sample S is very representative of $\mathcal{D}$, then the difference between the empirical error and generalization error should be small. What we want to know, is how large we need to choose n, the size of S, to ensure that the generalization error is small with high probability.

How large we choose n is partially dependent on the variables ε and δ of the desired strong learner. This does not yet capture how difficult it is to learn the concept class $\mathcal{C}$. Hence, it would make sense to analyze the complexity of $\mathcal{C}$. However, we can actually bypass this and only analyze the weak learner $\mathcal{W}$. This is because $\mathcal{W}$ already has certain assumed performance guarantees specifically on the concept class $\mathcal{C}$. If $\mathcal{C}$ is very complex, then the weak learner will require a certain level of complexity as well. We can analyze this using Vapnik-Chervonenkis theory (VC-theory) [19].

VC-theory introduces the VC-dimension, a measure of complexity of a hypothesis class. We say that a hypothesis class $\mathcal{H}$ *shatters* a set $A \subseteq \mathcal{X}$ of d points if for all 2^d labelings $\ell : A \to \{-1, 1\}$, there exists a hypothesis $h \in \mathcal{H}$ such that $h(x) = \ell(x)$ for all $x \in A$. The *VC-dimension* of a hypothesis class $\mathcal{H}$ is the size of a largest set A that is shattered by $\mathcal{H}$.

For example, consider the domain $\mathcal{X} = \mathbb{R}$. Let $\mathcal{H}_1 = \{h_\theta : \theta \in \mathbb{R}\}$, where $h_\theta(x) = 1$ if $x \geq \theta$ and $h_\theta(x) = -1$ otherwise. Then $\mathcal{H}_1$ shatters every set of size 1. For every set of size 2, however, there is no hypothesis that maps the left point to 1 and the right point to -1. Hence the VC-dimension of $\mathcal{H}_1$ is 1. Now let $\mathcal{H}_2 = \{h_{(a,b)} : a, b \in \mathbb{R}\}$, where $h_{(a,b)}(x) = 1$ if $x \in [a, b]$ and $h_{(a,b)}(x) = -1$ otherwise. Then $\mathcal{H}_2$ shatters every set of size 2, because we can find intervals that contain either, neither or both of the points. However, it does not shatter any set of size 3, because there is no interval that contains the two outer points but not the point in between.

In the following theorem, we find an upper bound on the probability that there exists a hypothesis for which the generalization error and empirical error are far apart. If we choose n sufficiently large, this means that the generalization error and empirical error are close with high probability, for every hypothesis.

Theorem 4.1 (Theorem 2.5 in [17]). *Let $\mathcal{H}$ be a hypothesis class of finite VC-dimension d. Let $S = ((x_1, f(x_1)), \ldots, (x_n, f(x_n)))$ be a sample generated i.i.d. from some distribution $\mathcal{D}$ with target function $f \in \mathcal{C}$. Then for every $\kappa > 0$ it holds that*

$$\mathbb{P}[\exists h \in \mathcal{H} : \mathit{err}(h, f, \mathcal{D}) > \widehat{\mathit{err}}(h, S) + \kappa] \leq 8 \left(\frac{en}{d}\right)^d \exp\left(\frac{-n\kappa^2}{32}\right).$$

It follows that if we set

$$n = O\left(\frac{d \ln(d/(\delta\varepsilon)) + \ln(1/\delta)}{\kappa^2}\right),$$

with a sufficiently large constant for n, then we will obtain with probability $\geq 1-\delta$, a sample S such that each hypothesis $h \in \mathcal{H}$ has generalization error at most κ bigger than its empirical error on S. We would like for the generalization error to be $\leq \varepsilon$ with probability $\geq 1-\delta$, so we should pick κ accordingly. If we can get the empirical error all the way down to 0, then we can set $\kappa = \varepsilon$ to get the desired result. Alternatively, we could ensure that the empirical error is $\leq \frac{\varepsilon}{2}$ and set $\kappa = \frac{\varepsilon}{2}$. Since we defined a weak learner to have $\delta = 0$ for convenience, we can achieve a sufficiently low empirical error with probability 1, as we will see in the upcoming sections. If we instead use a weak learner with $\delta > 0$, then we can only guarantee a sufficiently low empirical error with high probability. Since we run the weak learner many times, we would need the δ of the weak learner to be so small that there is still a high probability that it never fails in any of the runs. Luckily, lowering the δ of the weak learner is usually not that costly.

Since the above expression for n is a bit complicated, we would like to simplify it a bit. Unfortunately, simply using the $\tilde{O}$-notation to ignore logarithmic factors does not work, because δ only occurs within the logarithms. This also means lowering the δ of the strong learner has little impact, so we will assume from now on that δ is constant, say $\delta = 0.01$. Since we will always either have $\kappa = \varepsilon$ or $\kappa = \frac{\varepsilon}{2}$, we can rewrite the previous expression as

$$n = \tilde{O}\left(\frac{d}{\varepsilon^2}\right).$$

4.2 Basic booster

As previously mentioned, boosting uses the multiplicative weights method. Before we move on to established boosting algorithms, which deviate somewhat from the multiplicative weights framework, we will first construct a basic boosting algorithm from the multiplicative weights method as defined in Algorithm 2.1.

As the notation suggests, the n examples in the sample correspond to the n constraints in the multiplicative weights method. Unlike the stock market example from Section 2.1, this application has a proper oracle: the weak learner $\mathcal{W}$. It is called in each iteration t with distribution $p^{(t)}$ and returns an intermediate solution: a hypothesis $h^{(t)} \in \mathcal{H}_{\mathcal{W}}$. The cost $m_i^{(t)}$ of an example x_i is based on whether the hypothesis $h^{(t)}$ gets this example right or wrong. The most obvious option for the final hypothesis is simply a majority vote among the intermediate hypotheses $h^{(t)}$.

The cost vector $m^{(t)}$ is defined somewhat counter-intuitively, especially coming from the stock market example. If the hypothesis $h^{(t)}$ gets the example right, the cost is positive. Otherwise, the cost is negative. This means we decrease the weights of the examples we get right, while increasing the weights of examples we get wrong. The reason is that we want the weak learner to focus more on the examples it tends to get wrong. Where we previously guided the algorithm to the best experts, we are now guiding it to the hardest examples. This strategy works well when studying for exams too. If you are struggling with a particular

Algorithm 4.1: Basic multiplicative weights booster

```
  Input: $\gamma$-weak learner $\mathcal{W}$ with sample complexity $n_{\mathcal{W}}$,
         sample $S \in (\mathcal{X} \times \{-1,1\})^n$.
  Output: hypothesis $h : \mathcal{X} \to \{-1,1\}$.
1 Fix $\eta \leq \frac{1}{2}$. Set weight $w_i^{(1)} := 1$, for all $i \in [n]$.
2 for $t = 1, 2, \ldots, T$ do
3     Compute $\Phi^{(t)} = \sum_{i=1}^n w_i^{(t)}$.
4     Prepare $n_{\mathcal{W}}$ i.i.d. examples from $S$ with respect to $p^{(t)} = w^{(t)}/\Phi^{(t)}$.
5     Feed the $n_{\mathcal{W}}$ examples to $\mathcal{W}$ to obtain $h^{(t)}$.
6     For all $i \in [n]$, define cost $m_i^{(t)} = h^{(t)}(x_i) f(x_i)$.
7     Update weights: $w_i^{(t+1)} := w_i^{(t)} \exp(-\eta m_i^{(t)}), \forall i \in [n]$.
8 Output $h = \operatorname{sign}\left(\sum_{t=1}^T h^{(t)}\right)$.
```

type of question, it is more worthwhile to practice that type of question, rather than a type of question you are already good at.

Theorem 4.2. *Let $\mathcal{W}$ be a γ-weak learner with sample complexity $n_{\mathcal{W}}$ for concept class $\mathcal{C}$, with hypothesis class $\mathcal{H}_{\mathcal{W}}$. Given a sample $S \in (\mathcal{X} \times \{-1,1\})^n$, after $T > \frac{\ln n}{\gamma^2}$ rounds of Algorithm 4.1, with probability 1, we have $\widehat{err}(h, S) = 0$. This algorithm has time complexity*

$$O(T(W + n + n_{\mathcal{W}} \ln n)) = \tilde{O}\left(\frac{W + n + n_{\mathcal{W}}}{\gamma^2}\right),$$

where W is the time complexity of the weak learner $\mathcal{W}$.

Proof. Since Algorithm 4.1 is an instance of the multiplicative weights method, we can use Theorem 2.1. The weak learner uses the distribution $p^{(t)}$ in iteration t. By the definition of a γ-weak learner, we have that $\mathbb{P}[\operatorname{err}(h^{(t)}, f, p^{(t)}) \leq \frac{1}{2} - \gamma] = 1$, where the probability is taken over the sample and any randomness within the algorithm. Since a strong learner does not need to always have small error, we can safely condition on the event that $\operatorname{err}(h^{(t)}, f, p^{(t)}) \leq \frac{1}{2} - \gamma$, which occurs almost surely. Then we have $\mathbb{P}_{i \sim p^{(t)}}[h^{(t)}(x_i) \neq f(x_i)] \leq \frac{1}{2} - \gamma$. It follows that

$$m^{(t)} \cdot p^{(t)} = \mathbb{E}_{i \sim p^{(t)}} m_i^{(t)} \geq 1 \cdot \left(\frac{1}{2} + \gamma\right) - 1 \cdot \left(\frac{1}{2} - \gamma\right) = 2\gamma.$$

Note that we have binary costs. As mentioned in Section 2.3, this means that we minimize the loss on the right-hand side by picking $\eta = \sqrt{\ln(n)/T}$. It follows that after T rounds, we have for all $i \in [n]$

$$2\gamma T \leq \sum_{t=1}^T m^{(t)} \cdot p^{(t)} \leq \sum_{t=1}^T m_i^{(t)} + \eta \sum_{t=1}^T (m_i^{(t)})^2 \cdot p^{(t)} + \frac{\ln n}{\eta} = \sum_{t=1}^T m_i^{(t)} + 2\sqrt{T \ln n},$$

and thus,

$$\sum_{t=1}^{T} m_i^{(t)} \geq 2\gamma T - 2\sqrt{T \ln n}.$$

Note that the sign of the sum on the left-hand side in the above equation indicates whether the final hypothesis h correctly predicts example x_i. Hence, if we pick the number of iterations T such that the right-hand side is positive, then it is guaranteed that h will correctly predict all examples in the sample S. It follows that $T > \frac{\ln n}{\gamma^2}$ iterations are sufficient.

We will now analyze the time complexity of this algorithm. This depends on how line 4 is implemented. An example can be prepared by first generating a random number r uniformly from $[0,1]$. Then, find the lowest i such that $\sum_{j=1}^{i} p_j^{(t)} \geq r$ and pick the corresponding x_i as example. This is done in $O(n)$ time. However, we need to prepare $n_{\mathcal{W}}$ examples every iteration, so this can become quite costly. If we precompute the partial sums $\sum_{j=1}^{i} p_j^{(t)}$ for all i once per iteration in $O(n)$ time, then we can find the lowest i such that $\sum_{j=1}^{i} p_j^{(t)} \geq r$ using binary search in $O(\ln n)$ time. This means line 4 can be implemented to run in $O(n + n_{\mathcal{W}} \ln n) = \tilde{O}(n + n_{\mathcal{W}})$ time. Lines 3, 6 and 7 of the algorithm all cost $O(n)$ time. In total, the time complexity is

$$O(T(W + n + n_{\mathcal{W}} \ln n)) = \tilde{O}\left(\frac{W + n + n_{\mathcal{W}}}{\gamma^2}\right).$$

□

We have now shown that Algorithm 4.1 achieves an empirical error of 0, but we would like to also achieve a small generalization error.

Theorem 4.3. *Let $\mathcal{W}$ be a γ-weak learner with sample complexity $n_{\mathcal{W}}$ for concept class $\mathcal{C}$, with hypothesis class $\mathcal{H}_{\mathcal{W}}$ of VC-dimension d. Given a sample set $S \in (\mathcal{X} \times \{-1,1\})^n$ of size $n = \tilde{O}\left(\frac{d}{\varepsilon^2\gamma^2}\right)$ with a sufficiently large constant, with $T > \frac{\ln n}{\gamma^2}$ iterations, Algorithm 4.1 is an (ε,δ)-PAC learner for $\mathcal{C}$. This algorithm has time complexity*

$$\tilde{O}\left(\frac{W + n_{\mathcal{W}}}{\gamma^2} + \frac{d}{\varepsilon^2\gamma^4}\right).$$

Proof. Note that the hypothesis returned by our boosting algorithm (the strong learner) is the sign of a linear combination of hypotheses from the weak learner. It can be shown using VC-theory that the VC-dimension of the hypothesis class of the strong learner is then $O(dT \ln(dT))$ [19, Lemma 10.3]. It follows from Theorem 4.1 that $n = \tilde{O}\left(\frac{dT}{\varepsilon^2}\right) = \tilde{O}\left(\frac{d}{\varepsilon^2\gamma^2}\right)$ with a sufficiently large constant is sufficient to obtain with probability at least $1-\delta$ a hypothesis with generalization error at most ε bigger than the empirical error on S, which is 0 as shown in Theorem 4.2. It follows that Algorithm 4.1 is an (ε,δ)-PAC learner for $\mathcal{C}$, and the time complexity follows from substituting $n = \tilde{O}\left(\frac{d}{\varepsilon^2\gamma^2}\right)$ in the complexity given in Theorem 4.2. □

4.3 AdaBoost

One of the most commonly used boosting algorithms is Freund and Schapire's AdaBoost [13], short for Adaptive Boosting. The first lines of the algorithm match those of the Algorithm 4.1, but things change starting with line 6.

Algorithm 4.2: AdaBoost

```
  Input: $\gamma$-weak learner $\mathcal{W}$ with sample complexity $n_{\mathcal{W}}$,
         sample $S \in (\mathcal{X} \times \{-1,1\})^n$.
  Output: hypothesis $h : \mathcal{X} \to \{-1,1\}$.
1 Initialize weight vector $w_i^{(1)} = 1$ for all $i \in [n]$.
2 for $t = 1, \ldots, T$ do
3     Compute $\Phi^{(t)} = \sum_{i=1}^{n} w_i^{(t)}$.
4     Prepare $n_{\mathcal{W}}$ i.i.d. examples from $S$ with respect to $p^{(t)} = w^{(t)}/\Phi^{(t)}$.
5     Feed the $n_{\mathcal{W}}$ examples to $\mathcal{W}$ to obtain $h^{(t)}$.
6     Compute $\varepsilon^{(t)} = \mathbb{P}_{i \sim p^{(t)}}[h^{(t)}(x_i) \neq f(x_i)]$, set $\alpha^{(t)} = \frac{1}{2} \ln\left(\frac{1-\varepsilon^{(t)}}{\varepsilon^{(t)}}\right)$.
7     For all $i \in [n]$, set $w_i^{(t+1)} = \begin{cases} w_i^{(t)} \cdot e^{-\alpha^{(t)}}, & \text{if } h^{(t)}(x_i) = f(x_i), \\ w_i^{(t)} \cdot e^{\alpha^{(t)}}, & \text{if } h^{(t)}(x_i) \neq f(x_i). \end{cases}$
8 Output $h = \operatorname{sign}\left(\sum_{t=1}^{T} \alpha^{(t)} h^{(t)}\right)$.
```

In line 6, we compute the error $\varepsilon^{(t)}$ of the hypothesis $h^{(t)}$, which can be done exactly since it is weighted according to the known distribution $p^{(t)}$. Since $h^{(t)}$ was generated by a weak learner with $\delta = 0$, we know that $\varepsilon^{(t)} < \frac{1}{2}$. This implies that $\alpha^{(t)} > 0$. This $\alpha^{(t)}$ increases as the error decreases, and is used both when updating the weights, as well as in the final output. This means that iterations where $h^{(t)}$ has small error have a greater impact on the weights as well as on the final output.

Due to these changes, our earlier analysis of the multiplicative weights method does not apply here. We omit the proof, but it is shown in [13] that AdaBoost achieves empirical error $\leq \kappa$ for $T \approx \frac{1}{2\gamma^2} \ln \frac{1}{\kappa}$ iterations. Since κ is only present in a logarithmic factor, we can set $\kappa < \frac{1}{n}$ to achieve empirical error 0 with minimal additional costs. It also suffices to have sample size $n = \tilde{O}\left(\frac{dT}{\varepsilon^2}\right)$ as before. Since all differences with Algorithm 4.1 are in steps that take $O(n)$ time in both algorithms, the time complexity is again

$$O(T(W + n + n_{\mathcal{W}} \ln n)) = \tilde{O}\left(\frac{W + n + n_{\mathcal{W}}}{\gamma^2}\right) = \tilde{O}\left(\frac{W + n_{\mathcal{W}}}{\gamma^2} + \frac{d}{\varepsilon^2 \gamma^4}\right).$$

4.4 SmoothBoost

The SmoothBoost algorithm by Servedio [18] was made with the purpose of being more tolerant to *malicious noise*, which means that a fraction of the labels are flipped by an adversary. It achieves this tolerance by ensuring the distributions it generates are "smooth".

Algorithm 4.3: SmoothBoost

```
  Input: $\gamma$-weak learner $\mathcal{W}$ with sample complexity $n_{\mathcal{W}}$,
         sample $S \in (\mathcal{X} \times \{-1,1\})^n, \kappa \in (0,1), \theta \in [0,1/2)$.
  Output: hypothesis $h : \mathcal{X} \to \{-1,1\}$.
1 Initialize $t = 1$, $N_i^{(0)} = 0$, $w_i^{(1)} = 1$ for all $i \in [n]$.
2 while true do
3     Compute $\Phi^{(t)} = \sum_{i=1}^n w_i^{(t)}$.
4     If $\Phi^{(t)} < \kappa n$, then $T = t-1$, return $h = \text{sign}\left(\sum_{t=1}^T h^{(t)}\right)$, terminate.
5     Prepare $n_{\mathcal{W}}$ i.i.d. examples from $S$ with respect to $p^{(t)} = w^{(t)}/\Phi^{(t)}$.
6     Feed the $n_{\mathcal{W}}$ examples to $\mathcal{W}$ to obtain $h^{(t)}$.
7     For all $i \in [n]$, set $N_i^{(t)} = N_i^{(t-1)} + h^{(t)}(x_i)f(x_i) - \theta$,
      $w_i^{(t+1)} = \begin{cases} 1, & \text{if } N_i^{(t)} < 0, \\ (1-\gamma)^{N_i^{(t)}/2}, & \text{if } N_i^{(t)} \geq 0. \end{cases}$
8     $t = t + 1$.
9 Output $h = \text{sign}\left(\sum_{t=1}^T h^{(t)}\right)$.
```

Unlike the previous two algorithms, SmoothBoost does not run for a set number of iterations T, but instead has an exit condition: the algorithm terminates when the sum of the weights is sufficiently small. The updating of the weights is handled differently as well. Unlike AdaBoost, however, the output is still simply a majority vote.

In line 7, a new vector $N_i^{(t)}$ is introduced. This vector can be rewritten as

$$N_i^{(t)} = \left(\sum_{\tau=1}^{t} h^{(\tau)}(x_i) f(x_i) \right) - \theta t.$$

The term in parentheses is simply the number of correct classifications of example i minus the number of incorrect classifications of i, by the hypotheses $h^{(\tau)}$ up to iteration t. Hence, if $N_i^{(t)} \geq 0$, the example i is mostly being classified correctly. If we look at the weights $w_i^{(t)}$, then we can see that the weight decreases in this case, but it is otherwise exactly 1.

Lemma 4.1 (Lemma 1 in [18])**.** *For all iterations $1 \leq t \leq T$ of SmoothBoost, it holds that* $\max_{i \in [n]} p_i^{(t)} \leq \frac{1}{\kappa n}$.

Proof. As long as the exit condition has not been satisfied at iteration t, it holds that $\Phi^{(t)} \geq \kappa n$. Since it holds for all weights that $w_i^{(t)} \leq 1$, we have $p_i^{(t)} = w_i^{(t)}/\Phi^{(t)} \leq \frac{1}{\kappa n}$ for all $i \in [n]$. □

Lemma 4.1 shows the smoothness property the algorithm is named after. The main purpose of this property is making the algorithm more resistant to malicious noise. Algorithms such as AdaBoost have a tendency to assign too much weight to noisy examples in the sample S, which can result in poor performance. The smoothness property limits to what extent this can occur. This tolerance to malicious noise is not relevant to this thesis, but we will see some other useful consequences of the smoothness property in the upcoming sections.

Theorem 4.4 (Theorem 2 in [18])**.** *If SmoothBoost terminates, then the hypothesis h it returns has empirical error $\widehat{err}(h, S) \leq \kappa$.*

Proof. If $h(x_i) \neq f(x_i)$, then this implies that the majority vote is wrong on example x_i. Therefore, $N_i^{(T)} < 0$ and thus $w_i^{(T+1)} = 1$. It follows that the number of errors of the hypothesis h is at most $\sum_{i=1}^{n} w_i^{(T+1)} = \Phi^{(T+1)}$, which is less than κn due to the exit condition. It follows that $\widehat{\mathrm{err}}(h, S) < \kappa$. □

The proof of Theorem 4.4 is very simple due to the exit condition, which essentially checks whether the empirical error is sufficiently low. However, proving that SmoothBoost indeed terminates and finding a bound on T is significantly more difficult. We will not give the full proof here, but we will need some of the steps for later, so we will start from the following lemma.

Lemma 4.2. *If $\theta = \frac{\gamma}{2+\gamma}$ and it holds for all t that $err(h^{(t)}, f, p^{(t)}) \leq \frac{1}{2} - \gamma$, then*

$$\frac{2n}{\gamma\sqrt{1-\gamma}} > \gamma \sum_{t=1}^{T} \Phi^{(t)}.$$

Proof. This follows from combining the bounds in Lemma 4 and Lemma 5 in [18]. □

Theorem 4.5 (Theorem 3 in [18])**.** *If $\theta = \frac{\gamma}{2+\gamma}$ and it holds for all t that $err(h^{(t)}, f, p^{(t)}) \leq \frac{1}{2} - \gamma$, then SmoothBoost terminates in $T < \frac{2}{\kappa\gamma^2\sqrt{1-\gamma}}$ iterations.*

Proof. For $1 \leq t \leq T$, the exit condition has not yet been satisfied. Therefore we have $\Phi^{(t)} \geq \kappa n$ and thus $\gamma \sum_{t=1}^{T} \Phi^{(t)} \geq \gamma\kappa n T$. By combining this with Lemma 4.2, we obtain $T < \frac{2}{\kappa\gamma^2\sqrt{1-\gamma}}$. □

The parameter θ gives a threshold of what margin of correct classifications is required before the weights can be lowered. In other words, with a higher θ, more of a consensus between the hypotheses is required than simply a majority. If malicious noise is not a concern, then we can also set $\theta = 0$ to get a slightly better bound on T, without affecting any theoretical results relevant to this thesis. This new bound does not affect the time complexity, however. Recall that the second condition in Lemma 4.2 and Theorem 4.5 holds with probability 1.

Theorem 4.6. *Let $\mathcal{W}$ be a γ-weak learner with sample complexity $n_{\mathcal{W}}$ for concept class $\mathcal{C}$, with hypothesis class $\mathcal{H}_{\mathcal{W}}$ of VC-dimension d. Given a sample set $S \in (\mathcal{X} \times \{-1,1\})^n$ of size $n = \tilde{O}\left(\frac{d}{\varepsilon^2\gamma^2}\right)$ with a sufficiently large constant, SmoothBoost is an (ε,δ)-PAC learner for $\mathcal{C}$. This algorithm has time complexity*

$$\tilde{O}\left(\frac{W + n_{\mathcal{W}}}{\gamma^2} + \frac{d}{\varepsilon^2\gamma^4}\right).$$

Proof. Unfortunately, setting $\kappa < \frac{1}{n}$ as we did with AdaBoost is too costly here, due to the polynomial dependence on $\frac{1}{\kappa}$. Instead, we will set $\kappa = \frac{\varepsilon}{2}$ and choose $n = \tilde{O}\left(\frac{d}{\varepsilon^2\gamma^2}\right)$ such that the generalization error is at most $\frac{\varepsilon}{2}$ higher than the empirical error with probability at least $1-\delta$, as according to Theorem 4.1.

By Theorem 4.4, we have empirical error at most $\frac{\varepsilon}{2}$, and thus generalization error at most ε with probability at least $1-\delta$. The time complexity per iteration is the same as the previous boosting algorithms, but we now need $T = O\left(\frac{1}{\varepsilon\gamma^2}\right)$ iterations, as shown in Theorem 4.5. The total time complexity is therefore

$$O(T(W + n + n_{\mathcal{W}} \ln n)) = \tilde{O}\left(\frac{W + n + n_{\mathcal{W}}}{\varepsilon\gamma^2}\right) = \tilde{O}\left(\frac{W + n_{\mathcal{W}}}{\varepsilon\gamma^2} + \frac{d}{\varepsilon^4\gamma^4}\right).$$

□

4.5 Quantum AdaBoost

A quantum version of the AdaBoost algorithm was developed by Arunachalam and Maity [7]. A quantum booster can use either a classical weak learner, or a quantum weak learner, which can take quantum examples. In the previous classical boosting algorithms, each iteration had several steps that required $O(n)$ time. Using various quantum subroutines, these steps are made to run in $O(\sqrt{n})$ time instead. Quantum AdaBoost achieves a time complexity of

$$\tilde{O}\left(\frac{(W + n_{\mathcal{W}})^{1.5}\sqrt{n}}{\gamma^{10}}\right) = \tilde{O}\left(\frac{(W + n_{\mathcal{W}})^{1.5}\sqrt{d}}{\gamma^{11}}\right),$$

where ε, δ are both assumed to be constant. Compared to classical AdaBoost, which under the same assumptions has a total cost of $\tilde{O}\left(\frac{W+n_{\mathcal{W}}}{\gamma^2} + \frac{d}{\gamma^4}\right)$, the quantum version has a sublinear dependence on the sample size n and thus also d. While this is a big improvement, the dependence on γ is much worse than before. The main reason for this is that the error ε_t, which is approximated using a modified version of amplitude estimation, is used both in updating the weights and in the final output. The approximation error propagates through the algorithm, which has to take extra steps to ensure correctness, resulting in a more complicated algorithm with a bad dependence on γ. As we will see next, Quantum AdaBoost is outperformed by a much simpler algorithm, so we will not go into any further detail.

4.6 Quantum SmoothBoost

A quantum version of the SmoothBoost algorithm was developed by Izdebski and de Wolf [16]. As with Quantum AdaBoost, the goal is to achieve a sublinear dependence on n and thus d. Thanks to various properties of SmoothBoost, the quantization goes much more smoothly than it did with AdaBoost. The result is a much simpler algorithm with a much better dependence on γ than Quantum AdaBoost.

Algorithm 4.4: Quantum SmoothBoost

```
  Input: $\gamma$-weak (quantum) learner $\mathcal{W}$ with sample complexity $n_{\mathcal{W}}$,
         sample $S \in (\mathcal{X} \times \{-1,1\})^n, \kappa \in (0,1), \theta \in [0,1/2)$.
  Output: hypothesis $h : \mathcal{X} \to \{-1,1\}$.
1 Initialize $t = 1$, $N_i^{(0)} = 0$, $w_i^{(1)} = 1$ for all $i \in [n]$.
2 while true do
3     Compute an estimate $\tilde{\Phi}^{(t)}$ of $\Phi^{(t)} = \sum_{i=1}^n w_i^{(t)}$ using quantum
       approximate counting, where $N_i^{(t)} = \left(\sum_{\tau=1}^t h^{(\tau)}(x_i)f(x_i)\right) - \theta t$,
      $w_i^{(t+1)} = \begin{cases} 1, & \text{if } N_i^{(t)} < 0, \\ (1-\gamma)^{N_i^{(t)}/2}, & \text{if } N_i^{(t)} \geq 0. \end{cases}$
4     If $\tilde{\Phi}^{(t)} < \kappa n$, then $T = t-1$, return $h = \text{sign}\left(\sum_{t=1}^T h^{(t)}\right)$, terminate.
5     Prepare $n_{\mathcal{W}}$ copies of $|p^{(t)}\rangle = \sum_{i=1}^n \sqrt{p_i^{(t)}}\, |i, f(x_i)\rangle$, where
       $p^{(t)} = w^{(t)}/\Phi^{(t)}$.
6     Feed the $n_{\mathcal{W}}$ examples to $\mathcal{W}$ to obtain $h^{(t)}$.
7     $t = t+1$.
8 Output $h = \text{sign}\left(\sum_{t=1}^T h^{(t)}\right)$.
```

In order to achieve the sublinear dependence on n, we follow the approach of Quantum AdaBoost and compute the weights on demand whenever they are needed, as can be seen in line 3. Computing a weight $w_i^{(t)}$ on demand can be done in $O(t) = O(T)$ time, since the required $N_i^{(t)}$ is a sum over t elements. Normally, we need to know all n weights of an iteration to compute the sum $\Phi^{(t)}$ in line 3. However, using quantum approximate counting (Theorem 3.3), we can instead compute an estimate $\tilde{\Phi}^{(t)}$ of $\Phi^{(t)}$ in $\tilde{O}(T\sqrt{n})$ time. We can ensure that with high probability, quantum approximate counting outputs an estimate $\tilde{\Phi}^{(t)}$ such that $0.9\Phi^{(t)} \leq \tilde{\Phi}^{(t)} \leq 1.1\Phi^{(t)}$.

Instead of preparing $n_{\mathcal{W}}$ examples sampled from S, we now prepare W identical quantum states, that contain the entire sample in superposition. This allows the use of a quantum weak learner that takes such a quantum state as input, but we can also use a classical weak learner as before, by measuring in the computational basis. Generating these quantum states is done using amplitude amplification, and is much cheaper than it was in quantum AdaBoost, due to the smoothness property. Preparing one copy of $\left|p^{(t)}\right\rangle$ can be done in $\tilde{O}\left(\frac{T}{\sqrt{\kappa}}\right)$ time.

In total, Quantum SmoothBoost runs $O(Tn_{\mathcal{W}})$ quantum subroutines that each have a probability of failing. With an extra cost factor $\log(Tn_{\mathcal{W}})$, we can reduce the error probability to be $\ll 1/(Tn_{\mathcal{W}})$, so that, by the union bound, the probability that one of them fails is small. In particular, we want this probability to be at most $\frac{\delta}{2}$.

Notably, the core of Quantum SmoothBoost is identical to that of SmoothBoost. Given the same input, if $\mathcal{W}$ is a deterministic classical weak learner, the weights will be exactly the same in both algorithms. This is very different from Quantum AdaBoost, which has to approximate the errors $\varepsilon^{(t)}$ that are used to update the weights. This causes the approximation errors to propagate through the algorithm, which is the primary cause of all the complications in Quantum AdaBoost.

Because the core of the algorithm remains the same, we only need to slightly adjust the proofs to accommodate for the estimate $\tilde{\Phi}^{(t)}$. Lemma 4.2 is unaffected by this change. We will present the three theorems from before, adjusted for this new context.

Lemma 4.3. *For all iterations* $1 \leq t \leq T$ *of Quantum SmoothBoost, it holds that* $\max_{i\in[n]} p_i^{(t)} \leq \frac{1.1}{\kappa n}$*, assuming all quantum subroutines succeed.*

Proof. As long as the exit condition has not been satisfied at iteration t, it holds that $\tilde{\Phi}^{(t)} \geq \kappa n$. If quantum approximate counting succeeds, then we have $1.1\Phi^{(t)} \geq \tilde{\Phi}^{(t)}$. It follows that $\Phi^{(t)} \geq \frac{\kappa n}{1.1}$. Since it holds for all weights that $w_i^{(t)} \leq 1$, we have $p_i^{(t)} = w_i^{(t)}/\Phi^{(t)} \leq \frac{1.1}{\kappa n}$ for all $i \in [n]$. □

Theorem 4.7. *If Quantum SmoothBoost terminates, then the hypothesis h it returns has empirical error* $\widehat{err}(h,S) \leq \frac{\kappa}{0.9}$*, assuming all quantum subroutines succeed.*

Proof. If $h(x_i) \neq f(x_i)$, then this implies that the majority vote is wrong on example x_i. Therefore, $N_i^{(T)} < 0$ and thus $w_i^{(T+1)} = 1$. It follows that the number of errors of the hypothesis h is at most $\sum_{i=1}^n w_i^{(T+1)} = \Phi^{(T+1)}$. If quantum approximate counting succeeds, it holds that $0.9\Phi^{(T+1)} \leq \tilde{\Phi}^{(T+1)}$. Due to the exit condition, we have $\tilde{\Phi}^{(T+1)} < \kappa n$ and thus the number of errors of h is at most $\frac{\kappa n}{0.9}$. It follows that $\widehat{\text{err}}(h,S) < \frac{\kappa}{0.9}$. □

Theorem 4.8. *If* $\theta = \frac{\gamma}{2+\gamma}$ *and it holds for all t that* $err(h^{(t)}, f, p^{(t)}) \leq \frac{1}{2} - \gamma$*, then Quantum SmoothBoost terminates in* $T < \frac{2.2}{\kappa\gamma^2\sqrt{1-\gamma}}$*, assuming all quantum subroutines succeed.*

Proof. For $1 \leq t \leq T$, the exit condition has not yet been satisfied. Therefore we have $\tilde{\Phi}^{(t)} \geq \kappa n$. If quantum approximate counting succeeds, we have $1.1\Phi^{(t)} \geq \tilde{\Phi}^{(t)}$. Thus we have $1.1\gamma\sum_{t=1}^T \Phi^{(t)} \geq \gamma\kappa n T$. By combining this with Lemma 4.2, we obtain $T < \frac{2.2}{\kappa\gamma^2\sqrt{1-\gamma}}$. □

Theorem 4.9. *Let $\mathcal{W}$ be a γ-weak learner with sample complexity $n_{\mathcal{W}}$ for concept class $\mathcal{C}$, with hypothesis class $\mathcal{H}_{\mathcal{W}}$ of VC-dimension d. Given a sample set $S \in (\mathcal{X} \times \{-1,1\})^n$ of size $n = \tilde{O}\left(\frac{d}{\varepsilon^2\gamma^2}\right)$ with a sufficiently large constant, Quantum SmoothBoost is an (ε,δ)-PAC learner for $\mathcal{C}$. This algorithm has time complexity*

$$\tilde{O}\left(\frac{W}{\varepsilon\gamma^2} + \frac{n_{\mathcal{W}}}{\varepsilon^{2.5}\gamma^4} + \frac{\sqrt{d}}{\varepsilon^{3.5}\gamma^5}\right).$$

Proof. As with SmoothBoost, we want empirical error $\leq \frac{\varepsilon}{2}$, so we set $\kappa = 0.45\varepsilon$. We choose $n = \tilde{O}\left(\frac{d}{\varepsilon^2\gamma^2}\right)$ such that the generalization error is at most $\frac{\varepsilon}{2}$ higher than the empirical error with probability at least $1 - \frac{\delta}{2}$, as according to Theorem 4.1. Since we also assumed the quantum subroutines all succeed with probability at least $1 - \frac{\delta}{2}$, it follows from the union bound that the total error probability is at most δ.

By Theorem 4.7, we have empirical error at most $\frac{\varepsilon}{2}$, and thus generalization error at most ε with probability at least $1-\delta$. Approximating $\tilde{\Phi}^{(t)}$ costs $\tilde{O}(T\sqrt{n})$ time by Theorem 3.3. As explained in Section 4.1 of [16], preparing one copy of $\left|p^{(t)}\right\rangle$ costs $\tilde{O}(T/\sqrt{\varepsilon})$. As shown in Theorem 4.8, we still use $T = O\left(\frac{1}{\varepsilon\gamma^2}\right)$ iterations. It follows that the total time complexity of Quantum SmoothBoost is

$$\begin{aligned}\tilde{O}\left(T\left(W + \frac{Tn_{\mathcal{W}}}{\sqrt{\varepsilon}} + T\sqrt{n}\right)\right) &= \tilde{O}\left(\frac{W}{\varepsilon\gamma^2} + \frac{n_{\mathcal{W}}}{\varepsilon^{2.5}\gamma^4} + \frac{\sqrt{n}}{\varepsilon^2\gamma^4}\right)\\ &= \tilde{O}\left(\frac{W}{\varepsilon\gamma^2} + \frac{n_{\mathcal{W}}}{\varepsilon^{2.5}\gamma^4} + \frac{\sqrt{d}}{\varepsilon^{3.5}\gamma^5}\right).\end{aligned}$$

□

As with quantum AdaBoost, we have a sublinear dependence on the sample size n and thus d, which is a great quantum speed-up. While the dependence on γ is still slightly worse than in the classical algorithm, it is a huge improvement over Quantum AdaBoost.

4.7 Monte Carlo SmoothBoost

In this section, we present our own boosting algorithm, which we call Monte Carlo SmoothBoost, named after its usage of the Monte Carlo method. The algorithm is based closely on Quantum SmoothBoost, but it is classical. Despite this, it achieves not only a sublinear dependence on n, but even a logarithmic dependence on n, which is a significant improvement over the quantum boosting algorithms. This seems too good to be true, since this seems to imply a logarithmic dependence on the VC-dimension d, even though d indicates the complexity of the problem. We will see later why this implication is false.

There are two key differences between Quantum SmoothBoost and Monte Carlo SmoothBoost. Instead of quantum approximate counting, Monte Carlo SmoothBoost uses the Monte Carlo method to compute the estimate $\Phi^{(t)}$. Secondly, it prepares $n_{\mathcal{W}}$ i.i.d. examples instead of copies of a quantum state. However, it does not use the same binary search approach we saw in the classical algorithms. Instead, it uses rejection sampling, which we will explain later.

Algorithm 4.5: Monte Carlo SmoothBoost

```
  Input: $\gamma$-weak learner $\mathcal{W}$ with sample complexity $n_{\mathcal{W}}$,
         sample $S \in (\mathcal{X} \times \{-1,1\})^n, \kappa \in (0,1), \theta \in [0,1/2)$.
  Output: hypothesis $h : \mathcal{X} \to \{-1,1\}$.
1 Initialize $t = 1$, $N_i^{(0)} = 0$, $w_i^{(1)} = 1$ for all $i \in [n]$.
2 while true do
3     Compute an estimate $\tilde{\Phi}^{(t)}$ of $\Phi^{(t)} = \sum_{i=1}^n w_i^{(t)}$ using the Monte
      Carlo method, where $N_i^{(t)} = \left(\sum_{\tau=1}^t h^{(\tau)}(x_i) f(x_i)\right) - \theta t$,
      $w_i^{(t+1)} = \begin{cases} 1, & \text{if } N_i^{(t)} < 0, \\ (1-\gamma)^{N_i^{(t)}/2}, & \text{if } N_i^{(t)} \geq 0. \end{cases}$
4     If $\tilde{\Phi}^{(t)} < \kappa n$, then $T = t - 1$, return $h = \operatorname{sign}\left(\sum_{t=1}^T h^{(t)}\right)$, terminate.
5     Prepare $n_{\mathcal{W}}$ i.i.d. examples from $S$ with respect to $p^{(t)} = w^{(t)}/\Phi^{(t)}$.
6     Feed the $n_{\mathcal{W}}$ examples to $\mathcal{W}$ to obtain $h^{(t)}$.
7     $t = t + 1$.
8 Output $h = \operatorname{sign}\left(\sum_{t=1}^T h^{(t)}\right)$.
```

In step 3 of Quantum SmoothBoost, we use approximate counting to compute an estimate $\tilde{s}$ of $s = \sum_{i=1}^n M_i^t$, which is needed to check the exit condition of the algorithm. If $\frac{1}{1.1}s \leq \tilde{s} \leq 1.1s$ (with high probability), then we can use the logic of Quantum SmoothBoost to prove the algorithm works. The idea is to use the Monte Carlo method to estimate s, which does not require quantum computing.

In line 3, the Monte Carlo method is implemented as follows. We sample $j_1, \ldots, j_\ell$ i.i.d. uniformly from $\{1, \ldots, n\}$. Then we can estimate $\Phi^{(t)}$ by

$$\tilde{\Phi}^{(t)} = \frac{n}{\ell} \sum_{k=1}^{\ell} w_{j_k}^{(t)}.$$

To analyze this estimator, we will use Hoeffding's inequality.

Theorem 4.10 (Hoeffding's inequality). *Let $X_1, \ldots, X_n$ be independent random variables such that $a_i \leq X_i \leq b_i$. Let $S_n := \sum_{i=1}^n X_i$ be the sum of the random variables. Then, for all $t > 0$, it holds that*

$$\mathbb{P}(|S_n - \mathbb{E}[S_n]| \geq t) \leq 2 \exp\left(-\frac{2t^2}{\sum_{i=1}^n (b_i - a_i)^2}\right).$$

Theorem 4.11. *Let $\delta > 0$ and $\kappa > 0$, and let $T_u \geq T$ be an upper bound on the number of iterations of Monte Carlo SmoothBoost. The estimator $\tilde{\Phi}^{(t)} = \frac{n}{\ell}\sum_{k=1}^{\ell} w_{j_k}^{(t)}$ is an unbiased estimator of $\Phi^{(t)}$, and for $\ell \geq \frac{50}{\kappa^2}\ln\left(\frac{2T_u}{\delta}\right)$, with probability at least $1-\delta$, it holds that $|\tilde{\Phi}^{(t)} - \Phi^{(t)}| < 0.1\kappa n$ for all iterations $1 \leq t \leq T$. This estimation can be computed in $\tilde{O}\left(\frac{T\ln n}{\kappa^2}\right)$ time.*

Proof. We have

$$\mathbb{E}[\tilde{\Phi}^{(t)}] = \frac{n}{\ell}\cdot \ell \cdot \frac{\Phi^{(t)}}{n} = \Phi^{(t)},$$

so the estimate is unbiased. It follows from Hoeffding's theorem that for all $\eta > 0$, we have

$$\mathbb{P}(|\tilde{\Phi}^{(t)} - \Phi^{(t)}| \geq \eta) \leq 2\exp\left(-\frac{2\eta^2}{\sum_{k=1}^{\ell}\frac{n^2}{\ell^2}}\right) = 2\exp\left(-\frac{2\ell\eta^2}{n^2}\right).$$

We will set $\eta = 0.1\kappa n$, which yields

$$\mathbb{P}(|\tilde{\Phi}^{(t)} - \Phi^{(t)}| \geq 0.1\kappa n) \leq 2e^{-0.02\ell\kappa^2}.$$

We want the total error probability over all T iterations to be at most δ, which we can achieve by setting ℓ large enough such that the above error probability is upper bounded by δ/T. Then, by the union bound, the total error probability is at most δ. However, we do not know T ahead of time. In the upcoming Theorem 4.13, we obtain an upper bound on the number of iterations T. If we denote this upper bound by T_u, then it suffices to upper bound the error probability by $\delta/T_u \leq \delta/T$. Hence it suffices to pick ℓ such that

$$\begin{aligned}
2e^{-0.02\ell\kappa^2} &\leq \frac{\delta}{T_u}, \\
e^{-0.02\ell\kappa^2} &\leq \frac{\delta}{2T_u}, \\
-0.02\ell\kappa^2 &\leq \ln\left(\frac{\delta}{2T_u}\right), \\
\ell &\geq \frac{50}{\kappa^2}\ln\left(\frac{2T_u}{\delta}\right).
\end{aligned}$$

Note that the Monte Carlo method runs in $O(\ell T \ln n) = \tilde{O}\left(\frac{T\ln n}{\kappa^2}\right)$ time. Here, δ is hidden since it is assumed constant, but the dependence on δ is logarithmic, so we can cheaply lower δ. The logarithmic dependence on n here follows from the fact that we need $\log_2 n$ bits to work with indices in $[n]$. Note that now $|\tilde{\Phi}^{(t)} - \Phi^{(t)}| < 0.1\kappa n$ with probability δ/T_u. By the union bound, it holds that with probability at least $1-\delta$, the bound $|\tilde{\Phi}^{(t)} - \Phi^{(t)}| < 0.1\kappa n$ holds for all $1 \leq t \leq T \leq T_u$. □

While this is not the same guarantee as we got from quantum approximate counting with Quantum SmoothBoost, it is enough to prove the same properties as the two SmoothBoost algorithms. Again, the core of the algorithm is the same as the two previous SmoothBoost algorithms, so Lemma 4.2 still holds.

Lemma 4.4. *For all iterations $1 \leq t \leq T$ of Monte Carlo SmoothBoost, it holds that $\max_{i\in[n]} p_i^{(t)} < \frac{1}{0.9\kappa n}$, assuming the estimation never fails.*

Proof. As long as the exit condition has not been satisfied at iteration t, it holds that $\tilde{\Phi}^{(t)} \geq \kappa n$. If the estimation succeeds, we have $\Phi^{(t)} + 0.1\kappa n > \tilde{\Phi}^{(t)}$. It follows that $\Phi^{(t)} > 0.9\kappa n$. Since it holds for all weights that $w_i^{(t)} \leq 1$, we have $p_i^{(t)} = w_i^{(t)}/\Phi^{(t)} \leq \frac{1}{0.9\kappa n}$ for all $i \in [n]$. □

Theorem 4.12. *If Monte Carlo SmoothBoost terminates, then the hypothesis h it returns has empirical error $\widehat{err}(h, S) \leq 1.1\kappa$, assuming the estimation never fails.*

Proof. If $h(x_i) \neq f(x_i)$, then this implies that the majority vote is wrong on example x_i. Therefore, $N_i^{(T)} < 0$ and thus $w_i^{(T+1)} = 1$. It follows that the number of errors of the hypothesis h is at most $\sum_{i=1}^{n} w_i^{(T+1)} = \Phi^{(T+1)}$. If the estimation succeeds, $\Phi^{(T+1)} - 0.1\kappa n \leq \tilde{\Phi}^{(T+1)}$. Due to the exit condition, we have $\tilde{\Phi}^{(T+1)} < \kappa n$ and thus $\Phi^{(T+1)} < 1.1\kappa n$. Hence the number of errors of h is at most $1.1\kappa n$. It follows that $\widehat{\mathrm{err}}(h, S) < 1.1\kappa$. □

Theorem 4.13. *If $\theta = \frac{\gamma}{2+\gamma}$ and it holds for all t that $err(h^{(t)}, f, p^{(t)}) \leq \frac{1}{2} - \gamma$, then Monte Carlo SmoothBoost terminates in $T < \frac{1}{0.45\kappa\gamma^2\sqrt{1-\gamma}}$ iterations, assuming the estimation never fails.*

Proof. For $1 \leq t \leq T$, the exit condition has not yet been satisfied. Therefore we have $\tilde{\Phi}^{(t)} \geq \kappa n$. With high probability, we have $\Phi^{(t)} + 0.1\kappa n > \tilde{\Phi}^{(t)}$ and thus $\Phi^{(t)} > 0.9\kappa n$. Thus we have $\gamma \sum_{t=1}^{T} \Phi^{(t)} > 0.9\gamma\kappa n T$. By combining this with Lemma 4.2, we obtain $T < \frac{1}{0.45\kappa\gamma^2\sqrt{1-\gamma}}$. □

In previous classical algorithms, we used binary search to efficiently prepare examples from S. However, this required a precomputation that took $O(n)$ time, which we would like to avoid. Instead, we can prepare examples using rejection sampling as follows. We first sample $i \in [n]$ uniformly. With probability $w_i^{(t)}$, we output x_i. Otherwise, we reject it and restart the process. Since the probability of sampling an example x_i is proportional to $w_i^{(t)}$, it follows that the example is indeed sampled according to the distribution $p^{(t)}$ as desired. Note that we do not need to compute $\Phi^{(t)}$, even though it is used in the definition of $p^{(t)}$.

Theorem 4.14. *The expected number of repetitions before rejection sampling from $p^{(t)}$ in line 5 of Monte Carlo SmoothBoost terminates, is $O\left(\frac{1}{\kappa}\right)$, assuming the estimation never fails.*

Proof. We know that the exit condition of the algorithm has not been satisfied in line 5, so it holds that $\tilde{\Phi}^{(t)} \geq \kappa n$. With high probability, we have $\Phi^{(t)} + 0.1\kappa n > \tilde{\Phi}^{(t)}$. It follows that $\Phi^{(t)} > 0.9\kappa n$. The probability that we do accept the first sample is thus $\frac{\Phi^{(t)}}{n} > 0.9\kappa$. It follows that the expected number of repetitions before rejection sampling terminates is $O\left(\frac{1}{\kappa}\right) = O\left(\frac{1}{\varepsilon}\right)$. □

Theorem 4.15. *Let $\mathcal{W}$ be a γ-weak learner with sample complexity $n_{\mathcal{W}}$ for concept class $\mathcal{C}$, with hypothesis class $\mathcal{H}_{\mathcal{W}}$ of VC-dimension d. Given a sample set $S \in (\mathcal{X} \times \{-1, 1\})^n$ of size $n = \tilde{O}\left(\frac{d}{\varepsilon^2\gamma^2}\right)$ with a sufficiently large constant, Monte Carlo SmoothBoost is an (ε, δ)-PAC learner for $\mathcal{C}$. This algorithm has time complexity*

$$\tilde{O}\left(\frac{W}{\varepsilon\gamma^2} + \frac{n_{\mathcal{W}}}{\varepsilon^3\gamma^4} + \frac{\ln d}{\varepsilon^4\gamma^4}\right).$$

Proof. This time around, we set $\kappa = \frac{\varepsilon}{2.2}$ to achieve an empirical error $\leq \frac{\varepsilon}{2}$. We choose $n = \tilde{O}\left(\frac{d}{\varepsilon^2\gamma^2}\right)$ such that the generalization error is at most $\frac{\varepsilon}{2}$ higher than the empirical error with probability at least $1 - \frac{\delta}{2}$, as according to Theorem 4.1. By Theorem 4.11, we can run the estimator for long enough to ensure that with probability at least $1 - \frac{\delta}{2}$, it holds that $|\tilde{\Phi}^{(t)} - \Phi^{(t)}| < 0.1\kappa n$ for all iterations $1 \leq t \leq T$. It follows from the union bound that the total error probability is now at most δ.

By Theorem 4.12, we have empirical error at most $\frac{\varepsilon}{2}$, and thus generalization error at most ε with probability at least $1 - \delta$. Approximating $\tilde{\Phi}^{(t)}$ costs $\tilde{O}\left(\frac{T \ln n}{\varepsilon^2}\right)$ time by Theorem 4.11. As shown in Theorem 4.13, we still use $T = O\left(\frac{1}{\varepsilon\gamma^2}\right)$ iterations. It follows that the total time complexity of Quantum SmoothBoost is

$$\begin{aligned}\tilde{O}\left(T\left(W + \frac{Tn_{\mathcal{W}}}{\varepsilon} + \frac{T\ln n}{\varepsilon^2}\right)\right) &= \tilde{O}\left(\frac{W}{\varepsilon\gamma^2} + \frac{n_{\mathcal{W}}}{\varepsilon^3\gamma^4} + \frac{\ln n}{\varepsilon^4\gamma^4}\right)\\ &= \tilde{O}\left(\frac{W}{\varepsilon\gamma^2} + \frac{n_{\mathcal{W}}}{\varepsilon^3\gamma^4} + \frac{\ln d}{\varepsilon^4\gamma^4}\right).\end{aligned}$$

□

If we assume ε to be constant, Monte Carlo SmoothBoost is a strict improvement over both quantum algorithms. It is not a strict improvement over the classical algorithms, but the dependence on γ is similar if the time complexity is expressed in d instead of n, with only the second term having a significantly worse dependence on γ.

This result is quite surprising at first glance. It suggests that the VC-dimension d has very little impact on the time complexity of the boosting algorithm, even though the VC-dimension should indicate the difficulty of the learning problem. If we significantly increase the complexity of the concept class $\mathcal{C}$ and the hypothesis class $\mathcal{H}$, then we should intuitively see a significant increase in time complexity of the boosting algorithm. Yet this does not appear to be the case with Monte Carlo SmoothBoost.

The likely cause of this discrepancy, is that this seemingly missing complexity is hiding in the sample complexity $n_{\mathcal{W}}$ and the time complexity W of the weak learner $\mathcal{W}$. After all, the weak learner is defined as a PAC-learner for a specific concept class $\mathcal{C}$ and hypothesis class $\mathcal{H}_{\mathcal{W}}$. Assuming the existence of a weak learner is what allows us to express the complexity of boosting algorithms without any dependence on the complexity of $\mathcal{C}$. A more complex concept class would simply require a more complex hypothesis class in order for a weak learner to exist in the first place. We can see the dependence on the complexity of $\mathcal{H}$ through its VC-dimension d. However, the impact of the complexity of $\mathcal{H}$ on $n_{\mathcal{W}}$ and W is invisible in this analysis. Hence, the fact that the dependence on d is logarithmic in the time complexity of Monte Carlo SmoothBoost, is not necessarily that significant.

Suppose we know that a strong learner requires $\Omega(N)$ samples. If we use a boosting algorithm that uses T iterations with a weak learner $\mathcal{W}$ that requires $n_{\mathcal{W}}$ samples, then we obtain a strong learner that uses $Tn_{\mathcal{W}}$ samples. Hence, we know that $Tn_{\mathcal{W}} \geq N$ and thus $n_{\mathcal{W}} \geq N/T$. Note that T is chosen only based on ε and γ in the boosting algorithms we have seen. If we consider ε and γ to be constants, it follows that the weak learner requires $n_{\mathcal{W}} = \Omega(N)$ samples as well. This further suggests that much of the complexity is hidden in the weak learner.

The main accomplishment of the quantum boosting algorithms we have seen, is the sublinear dependence on n. Due to what we just discussed, this unfortunately does not imply a better dependence on the VC-dimension d, since both $n_{\mathcal{W}}$ and W depend on d as well. With Monte Carlo SmoothBoost having only a logarithmic dependence on n, the only advantage of using quantization in boosting algorithms is shown to be even more insignificant, since a classical algorithm can do even better. Existing classical boosting algorithms such as AdaBoost still have the best dependence on ε and γ, which likely outweighs a better dependence on n in most cases.

Chapter 5

Linear programs and semidefinite programs

Another application of the multiplicative weights method is solving linear programs (LPs) and the more general semidefinite programs (SDPs). In both of these types of problem, there is an objective function that needs to be minimized or maximized by tweaking a set of variables subject to a number of constraints.

5.1 Linear programs

In an LP, there is a linear objective function that we aim to maximize, subject to a number of linear constraints. We will write the standard form of an LP as follows.

$$\begin{aligned} \max\ & c^\top x \\ \text{s.t. } & Ax \leq b, \\ & x \geq 0. \end{aligned}$$

Here $x \in \mathbb{R}^m$ is the vector of variables over which we maximize. The vectors $b \in \mathbb{R}^n, c \in \mathbb{R}^m$ and the constraint matrix $A \in \mathbb{R}^{n \times m}$ are given. We can view the second line in the problem definition as a set of n linear constraints. For every $i \in [n]$, x is subject to the constraint $A_i x \leq b_i$, where A_i denotes the i-th row of A.

Since the multiplicative weights method is iterative and generally closes in on the solution to a problem, rather than finding an exact solution, we will relax the problem. The first step of the relaxation is to turn the LP from a maximization problem into a feasibility problem. That is, for some $\alpha \in \mathbb{R}$, we want to find $x \geq 0$, subject to $Ax \leq b$, such that $c^\top x \geq \alpha$. Note that the constraint $c^\top x \geq \alpha$ is simply another linear constraint and it can therefore be absorbed into A and b. If we know an upper bound for the maximum, then we can use a binary search over the value of α to find a solution x such that $c^\top x$ is arbitrarily close to the maximum. The number of times we need to solve the feasibility problem to approximate the maximization problem is logarithmic with respect to the desired precision.

The second step of the relaxation is to allow solutions x that only approximately satisfy the constraints. For a given $\varepsilon > 0$, we consider the constraints to be sufficiently satisfied if for all $i \in [n]$, we have $A_i x \leq b_i + \varepsilon$. Such an $x \in \mathbb{R}^m$ is an approximate solution to the LP. This in itself is not much of a relaxation, since we could simply absorb the ε into the vector b to obtain a problem of the same format as before. The difference is that we do not necessarily require finding an approximate solution even if one exists. If there exists a solution to the feasibility problem, then we must find at least an approximate solution. If the feasibility problem has no solution, however, then we are allowed to either find an approximate solution or simply return nothing. The final relaxed form of an LP can be written as follows.

$$\begin{aligned} \text{if } & \exists x \geq 0 \\ \text{s.t. } & Ax \leq b, \\ \text{then find } & x \geq 0 \\ \text{s.t. } & A_i x \leq b_i + \varepsilon, \forall i \in [n]. \end{aligned}$$

Note that the above problem definition does not specify what to return in case there exists no solution $x \geq 0$ such that $Ax \leq b$. As long as it does not return an infeasible approximate solution x, any output is acceptable. If an algorithm correctly solves the problem and does not return an approximate solution x, then this proves that the feasibility problem has no solution. The actual proof can then be derived from the proof that the algorithm functions as intended.

5.1.1 Basic LP solver

The first algorithm we will look at is a direct application of the multiplicative weights method as defined in Algorithm 2.2, with the alternative update rule. The choice for the n constraints in the multiplicative weights method is easy, since the LP we want to solve has n constraints as well.

Similar to the boosting algorithms, we have an oracle that outputs intermediate solutions. In iteration t, the oracle outputs $x^{(t)} \in \mathbb{R}^m$. Since the entire point of the algorithm is to find a feasible approximate solution, we cannot expect these intermediate solutions to be feasible. Instead, the oracle solves a different problem, based on the probability distribution $p^{(t)}$ over the n constraints. The problem is to find $x \geq 0$ such that $p^{(t)\top} Ax \leq p^{(t)\top} b$. If such a solution exists, then the oracle returns a solution $x^{(t)}$. Otherwise, it returns nothing. Since this problem is just a single linear constraint, it is much simpler to solve than the LP. Depending on the type of LP, different oracles might be convenient.

Note that if there exists an $x \in \mathbb{R}^m$ that satisfies $Ax \leq b$, then it also satisfies $p^{(t)\top} Ax \leq p^{(t)\top} b$. By contraposition, if the oracle fails to find a solution, then the original LP also has no solution. This means that we can end the algorithm as soon as the oracle fails to find a solution.

The cost vector is $c^{(t)} := b - Ax^{(t)}$. Note that this has no relation to the vector c from the standard form of the unrelaxed LP. The cost $c_i^{(t)} = b_i - A_i x^{(t)}$ can be seen as the margin by which the i-th constraint is satisfied by $x^{(t)}$. A positive cost implies that the constraint is satisfied, while a negative cost implies that it is not. As with the boosting algorithms, this seems somewhat backwards, but the reason is that we want the algorithm to focus on constraints that are not yet satisfied. Such constraints will have their weights increased.

Another assumption we need to make about the oracle is that this cost has an upper and lower bound. We will assume that for any intermediate solution $x^{(t)}$, it holds that $b - Ax^{(t)} \in [-1, 1]^n$, and thus $c^{(t)} \in [-1, 1]^n$ as desired. If the oracle has larger (finite) bounds, then we can simply scale A and b accordingly.

The oracle does not need to satisfy all constraints. However, for any unsatisfied constraints, there should be other constraints that make up for it by being satisfied by a large enough margin. Constraints with larger weights contribute more, both positively and negatively, and are therefore more important to satisfy, preferably by a large margin.

The output of the algorithm is simply the average $\bar{x} := \frac{1}{T} \sum_{t=1}^{T} x^{(t)}$ of the intermediate solutions returned by the oracle.

Algorithm 5.1: Basic multiplicative weights LP algorithm

```
1 Fix $\eta \leq \frac{1}{2}$. Set weight $w_i^{(1)} := 1, \forall i \in [n]$.
2 for $t = 1, 2, \ldots, T$ do
3     Feed into the oracle the distribution $p^{(t)} = w^{(t)}/\Phi^{(t)}$, where
        $\Phi^{(t)} = \sum_{i=1}^{n} w_i^{(t)}$, to obtain $x^{(t)}$. If the oracle returns nothing, then
        terminate.
4     For all $i \in [n]$, define cost $c_i^{(t)} = b_i - A_i x^{(t)}$.
5     Update weights: $w_i^{(t+1)} := w_i^{(t)}(1 - \eta c_i^{(t)}), \forall i \in [n]$.
6 Output $\bar{x} = \frac{1}{T} \sum_{t=1}^{T} x^{(t)}$.
```

Theorem 5.1. *If there exists a solution $x \geq 0$ such that $Ax \leq b$, then Algorithm 5.1 with $\eta = \sqrt{\frac{\ln n}{2T}}$ and $T = \lceil 8 \ln(n)/\varepsilon^2 \rceil$ returns an approximate solution $\bar{x}$ such that $A\bar{x} \leq b + \varepsilon$.*

Proof. Note that the oracle will never fail to find a solution if the original LP has a solution. Hence, we can assume the algorithm does not terminate early and outputs a solution $\bar{x}$. Since we are using the multiplicative weights method with the alternative update rule, Theorem 2.2 applies to this algorithm. Working out the left-hand side of the inequality in Theorem 2.1, we get

$$\sum_{t=1}^{T} c^{(t)} \cdot p^{(t)} = \sum_{t=1}^{T} \left(b - Ax^{(t)}\right) \cdot p^{(t)} = \sum_{t=1}^{T} \left({p^{(t)}}^\top b - {p^{(t)}}^\top Ax^{(t)}\right) \geq 0,$$

where the last inequality follows directly from the fact that the oracle gives a solution $x^{(t)}$ such that ${p^{(t)}}^\top b - {p^{(t)}}^\top Ax^{(t)} \geq 0$.

It now follows from Theorem 2.1 that after T rounds, for all $i \in [n]$, we have

$$\begin{aligned} 0 &\leq \sum_{t=1}^{T} c_i^{(t)} + \eta \sum_{t=1}^{T} |c_i^{(t)}| + \frac{\ln n}{\eta} \\ &= (1+\eta) \sum_{t=1}^{T} c_i^{(t)} + 2\eta \sum_{t: c_i^{(t)} < 0}^{T} |c_i^{(t)}| + \frac{\ln n}{\eta} \\ &\leq (1+\eta) \sum_{t=1}^{T} (b_i - A_i x^{(t)}) + 2\eta T + \frac{\ln n}{\eta}. \end{aligned}$$

We choose $\eta = \sqrt{\frac{\ln n}{2T}}$ such that the final two terms each become $\sqrt{2T \ln n}$. Dividing by T and substituting $\bar{x} = \frac{1}{T} \sum_{t=1}^{T} x^{(t)}$, we find

$$0 \leq (1+\eta) \frac{1}{T} \sum_{t=1}^{T} (b_i - A_i x^{(t)}) + \frac{2\sqrt{2T \ln n}}{T} = (1+\eta)(b_i - A_i \bar{x}) + \sqrt{\frac{8 \ln n}{T}}.$$

For $T \geq \lceil 8 \ln(n)/\varepsilon^2 \rceil$, we find

$$0 \leq (1+\eta)(b_i - A_i \bar{x}) + \varepsilon.$$

It follows that $(1+\eta) A_i \bar{x} \leq (1+\eta) b_i + \varepsilon$ and hence $A_i \bar{x} \leq b_i + \varepsilon$. This holds for all $i \in [n]$, so we find that $\bar{x}$ is a solution to the relaxed problem. $\square$

5.1.2 Zero-sum games

The notion that zero-sum games and LPs are equivalent was proven in [1]. Hence, we can also solve LPs by solving the corresponding zero-sum games.

In a zero-sum game, two players that we call Alice and Bob each have a finite number of pure strategies. Alice has m moves, while Bob has n. After both players select a move, the payoff is determined based on the payoff matrix $A \in [-1, 1]^{m \times n}$. In particular, if Alice plays $i \in [m]$ and Bob plays $j \in [n]$, then Alice receives payoff A_{ij}, while Bob receives $-A_{ij}$. The sum of these payoffs is always zero, hence the name zero-sum game.

Instead of a pure strategy, Alice and Bob can also use a randomized strategy, which is a probability distribution over the moves. If Alice uses the strategy $p \in \Delta^m$ and Bob uses $q \in \Delta^n$, then the expected payoff for Alice is $p^\top Aq$.

In order to solve a zero-sum game, we need to find a Nash equilibrium.

Definition 5.1. *A Nash equilibrium is a pair of strategies (p, q) such that, for all $i \in [m]$ and for all $j \in [n]$,*

$$e_i^\top Aq \le p^\top Aq \le p^\top Ae_j.$$

The first inequality implies that any pure strategy $i \in [n]$ that Alice chooses does not strictly improve her expected payoff if Bob sticks to his strategy q. It follows that no randomized strategy can improve her expected payoff either. The second inequality similarly implies that Bob cannot improve his expected payoff as long as Alice sticks to her strategy p.

Just as with the multiplicative weights algorithm for LPs, we will not find an exact solution to the zero-sum game. Instead, we will consider an approximate Nash equilibrium to be sufficient.

Definition 5.2. *For a given $\varepsilon > 0$, a ε-Nash equilibrium is a strategy pair (p, q) such that for all $i \in [m]$ and for all $j \in [m]$,*

$$e_i^\top Aq - \varepsilon \le p^\top Aq \le p^\top Ae_j + \varepsilon.$$

This means that both Alice and Bob can improve their expected payoff by no more than ε if their opponent does not change strategies.

The approach of solving zero-sum games to solve LPs is used by Grigoriadis and Khachiyan, in an algorithm that has similarities to multiplicative weights [14], but does not use the analysis of multiplicative weights. We will build up to this algorithm starting with a more direct application of multiplicative weights, which allows us to use the associated theorems in the analysis. The goal of this approach is to simplify the analysis of the algorithm in [14] by using Theorem 2.1.

5.1.3 Deterministic zero-sum LP solver

The largest difference between the zero-sum algorithms in this section and standard multiplicative weights algorithms, is that we keep track of two weight vectors, two probability distributions and two cost vectors. These all correspond to the two players.

Algorithm 5.2: Zero-sum multiplicative weights

```
1 Fix $\eta \le \frac{1}{2}, \theta \le \frac{1}{2}$. Initialize $P^{(1)} = 1 \in \mathbb{R}^m, Q^{(1)} = 1 \in \mathbb{R}^n$.
2 for $t = 1, 2, \ldots, T$ do
3     $p^{(t)} = P^{(t)} / \left\| P^{(t)} \right\|_1, q^{(t)} = Q^{(t)} / \left\| Q^{(t)} \right\|_1$.
4     $c^{(t)} = -Aq^{(t)}, d^{(t)} = A^\top p^{(t)}$.
5     $P^{(t+1)} = P^{(t)} \exp(-\eta c^{(t)}), Q^{(t+1)} = Q^{(t)} \exp(-\theta d^{(t)})$.
6 Output $(p, q)$, where $p := \frac{1}{T} \sum_{t=1}^{T} p^{(t)}$ and $q := \frac{1}{T} \sum_{t=1}^{T} q^{(t)}$.
```

We can view this algorithm as a repeated zero-sum game where Alice and Bob update their randomized strategies $p^{(t)}$ and $q^{(t)}$ in every iteration based on the strategy of their opponent. The cost vector $c^{(t)}$ belongs to Alice and $-c_i^{(t)}$ is the expected payoff for Alice if she plays i and Bob plays randomly according to $q^{(t)}$. The cost vector $d^{(t)}$ belongs to Bob and $-d_j^{(t)}$ is the expected payoff for Bob if he plays j and Alice plays randomly according to $p^{(t)}$. The reason for the minus is that we want the pure strategies with higher payoff to get more weight, which happens when the cost is negative.

Note that if we only look at Alice's side of the algorithm, then the algorithm is simply the multiplicative weights method. The only connection to Bob is found in the cost function, which is based on Bob's strategy $q^{(t)}$. Equivalently, Bob's side of the algorithm can also be seen as an instance of the multiplicative weights method. Essentially, Algorithm 5.2 simultaneously runs two instances of the multiplicative weights method that determine each other's cost vectors in every iteration. Since the multiplicative weights method allows for arbitrary cost vectors, these are indeed valid instances. As a consequence, we can utilize Theorem 2.1 to obtain two inequalities, one for Alice and one for Bob.

Theorem 5.2. *For a given $\varepsilon > 0$, after $T \geq \frac{16 \ln(\max\{m,n\})}{\varepsilon^2}$ rounds of Algorithm 5.2 with parameters $\eta = \sqrt{\frac{\ln m}{T}}$ and $\theta = \sqrt{\frac{\ln n}{T}}$, for all $i \in [m]$ and for all $j \in [n]$, we have*

$$e_i^\top Aq - \varepsilon \leq p^\top Aq \leq p^\top Ae_j + \varepsilon,$$

where $p := \frac{1}{T}\sum_{t=1}^{T} p^{(t)}$ and $q := \frac{1}{T}\sum_{t=1}^{T} q^{(t)}$.

Proof. As mentioned previously, we can view Algorithm 5.2 as two simultaneous instances of the multiplicative weights method. As such, we get two guarantees from Theorem 2.1. For all $i \in [m]$ and for all $j \in [n]$ respectively, we have

$$\sum_{t=1}^{T} c^{(t)} \cdot p^{(t)} \leq \sum_{t=1}^{T} c_i^{(t)} + \eta \sum_{t=1}^{T} (c^{(t)})^2 \cdot p^{(t)} + \frac{\ln m}{\eta},$$
$$\sum_{t=1}^{T} d^{(t)} \cdot q^{(t)} \leq \sum_{t=1}^{T} d_j^{(t)} + \theta \sum_{t=1}^{T} (d^{(t)})^2 \cdot q^{(t)} + \frac{\ln n}{\theta}.$$

We pick $\eta = \sqrt{\frac{\ln m}{T}}$ and $\theta = \sqrt{\frac{\ln n}{T}}$ such that each of the final two terms in both inequalities is bounded by $\sqrt{T \ln(\max\{m,n\})}$. Let $B \geq 2\sqrt{T \ln(\max\{m,n\})}$ be an upper bound on the final two terms. With some further rewriting, we obtain

$$\sum_{t=1}^{T} p^{(t)\top} Aq^{(t)} \geq \sum_{t=1}^{T} e_i^\top Aq^{(t)} - B,$$
$$\sum_{t=1}^{T} p^{(t)\top} Aq^{(t)} \leq \sum_{t=1}^{T} p^{(t)\top} Ae_j + B.$$

We can now combine the two inequalities and drop the middle expression. Dividing by T and substituting $p := \frac{1}{T}\sum_{t=1}^{T} p^{(t)}$ and $q := \frac{1}{T}\sum_{t=1}^{T} q^{(t)}$, we find

$$e_i^\top Aq - \frac{B}{T} \leq p^\top Ae_j + \frac{B}{T}.$$

Note that the above still holds for all $i \in [m]$ and for all $j \in [n]$. This means that for all $i \in [m]$, we have

$$e_i^\top Aq - \frac{B}{T} \leq \min_{j\in[n]} p^\top Ae_j + \frac{B}{T} \leq p^\top Aq + \frac{B}{T},$$

and for all $j \in [n]$, we have

$$p^\top Aq - \frac{B}{T} \leq \max_{i\in[m]} e_i^\top Aq - \frac{B}{T} \leq p^\top Ae_j + \frac{B}{T}.$$

Combining these two inequalities, we find that for all $i \in [m]$ and for all $j \in [n]$, it holds that

$$e_i^\top Aq - \frac{2B}{T} \leq p^\top Aq \leq p^\top Ae_j + \frac{2B}{T}.$$

We get the desired result if $\frac{2B}{T} \leq \varepsilon$, or equivalently, $B \leq \frac{\varepsilon T}{2}$. Since we assumed $B \geq 2\sqrt{T\ln(\max\{m,n\})}$, we need to choose T such that $2\sqrt{T\ln(\max\{m,n\})} \leq \frac{\varepsilon T}{2}$, which is true for $T \geq \frac{16\ln(\max\{m,n\})}{\varepsilon^2}$. $\square$

Algorithm 5.2 obtains the desired result, but computing the costs is computationally expensive due to the matrix-vector multiplication. This operation costs $O(mn)$ time every iteration. A significant speed-up can be achieved by having both players sample a move from their distribution in each iteration, and using these samples to determine the other players costs, as we will see next.

5.1.4 Zero-sum LP solvers using sampling

Algorithm 5.3: Zero-sum multiplicative weights with sampling

```
1 Fix $\eta \leq \frac{1}{2}, \theta \leq \frac{1}{2}$. Initialize $P^{(1)} = 1 \in \mathbb{R}^m, Q^{(1)} = 1 \in \mathbb{R}^n$.
2 for $t = 1, 2, \ldots, T$ do
3     $p^{(t)} = P^{(t)} / \left\|P^{(t)}\right\|_1, q^{(t)} = Q^{(t)} / \left\|Q^{(t)}\right\|_1$.
4     Sample $a^{(t)} \sim p^{(t)}, b^{(t)} \sim q^{(t)}$.
5     $c^{(t)} = -Ae_{b^{(t)}}, d^{(t)} = A^\top e_{a^{(t)}}$.
6     $P^{(t+1)} = P^{(t)} \exp(-\eta c^{(t)}), Q^{(t+1)} = Q^{(t)} \exp(-\theta d^{(t)})$.
7 Output $(p, q)$, where $p := \frac{1}{T}\sum_{t=1}^T p^{(t)}$ and $q := \frac{1}{T}\sum_{t=1}^T q^{(t)}$.
```

In this new algorithm, Alice samples $a^{(t)}$ and Bob samples $b^{(t)}$. Since Alice only has m moves and Bob has n, there are only n possible cost vectors $c^{(t)}$ and only m possible cost vectors $d^{(t)}$. In fact, these cost vectors are simply the columns (multiplied by -1) and rows of the matrix A. This means computing the costs and updating the weights of an iteration can now be done in $O(m+n)$ time, a quadratic improvement over the $O(mn)$ from before. In fact, since the matrix consists of $O(mn)$ elements, this means the time complexity is now sublinear in the input size.

Unfortunately, we have not been able to prove that Algorithm 5.3 achieves the desired result. An approach similar to the previous proof of Algorithm 5.2 appears to come very close, but not quite close enough. Due to the random sampling done in the algorithm, we cannot expect the algorithm to get the desired result with probability 1. Instead, we attempted to prove that it holds with arbitrarily high probability. We have the following conjecture.

Conjecture 5.1. *For given $\varepsilon > 0$ and $\delta > 0$, there exists a $T' = O\left(\frac{\log(mn)}{\varepsilon^2}\right)$, such that after $T \geq T'$ rounds of Algorithm 5.3 with parameters $\eta = \sqrt{\frac{\ln m}{T}}$ and $\theta = \sqrt{\frac{\ln n}{T}}$, with probability at least $1 - \delta$, for every $i \in [m]$ and for all $j \in [n]$, we have*

$$e_i^\top Aq - \varepsilon \leq p^\top Aq \leq p^\top Ae_j + \varepsilon,$$

where $p := \frac{1}{T}\sum_{t=1}^{T} p^{(t)}$ and $q := \frac{1}{T}\sum_{t=1}^{T} q^{(t)}$.

Since the sampling is just an extra step in computing the cost vectors, we can view Algorithm 5.3 as two simultaneous instances of the multiplicative weights method. As such, we get two guarantees from Theorem 2.1. For all $i \in [m]$ and for all $j \in [n]$ respectively, we have

$$\sum_{t=1}^{T} c^{(t)} \cdot p^{(t)} \leq \sum_{t=1}^{T} c_i^{(t)} + \eta \sum_{t=1}^{T} (c^{(t)})^2 \cdot p^{(t)} + \frac{\ln m}{\eta},$$
$$\sum_{t=1}^{T} d^{(t)} \cdot q^{(t)} \leq \sum_{t=1}^{T} d_j^{(t)} + \theta \sum_{t=1}^{T} (d^{(t)})^2 \cdot q^{(t)} + \frac{\ln n}{\theta}.$$

We pick $\eta = \sqrt{\frac{\ln m}{T}}$ and $\theta = \sqrt{\frac{\ln n}{T}}$ such that each of the final two terms in both inequalities is upper bounded by $\sqrt{T \ln(\max\{m, n\})}$. Let $B \geq 2\sqrt{T \ln(\max\{m, n\})}$ be an upper bound of the final two terms in both of the above inequalities. With some further rewriting, we obtain

$$\sum_{t=1}^{T} p^{(t)\top} A e_{b^{(t)}} \geq \sum_{t=1}^{T} e_i^\top A e_{b^{(t)}} - B,$$
$$\sum_{t=1}^{T} e_{a^{(t)}}^\top A q^{(t)} \leq \sum_{t=1}^{T} e_{a^{(t)}}^\top A e_j + B.$$

Unlike in the previous proof, the left-hand sides are not equal here. However, note that $\mathbb{E}[e_{a^{(t)}}] = p^{(t)}$ and $\mathbb{E}[e_{b^{(t)}}] = q^{(t)}$, due to how $a^{(t)}$ and $b^{(t)}$ are sampled. We now take the expectation over the $p^{(t)}, q^{(t)}, a^{(t)}, b^{(t)}$, in other words the expectation over a run of the algorithm, to combine the inequalities and obtain

$$\mathbb{E}\left[\sum_{t=1}^{T} e_i^\top A q^{(t)}\right] - B \leq \mathbb{E}\left[\sum_{t=1}^{T} p^{(t)\top} A q^{(t)}\right] \leq \mathbb{E}\left[\sum_{t=1}^{T} p^{(t)\top} A e_j\right] + B.$$

Dropping the middle expression, dividing by T and substituting $p := \frac{1}{T}\sum_{t=1}^{T} p^{(t)}$ and $q := \frac{1}{T}\sum_{t=1}^{T} q^{(t)}$, we find

$$\mathbb{E}\left[e_i^\top Aq\right] - \frac{B}{T} \le \mathbb{E}\left[p^\top Ae_j\right] + \frac{B}{T}.$$

Note that the above still holds for all $i \in [m]$ and for all $j \in [n]$. This means that for all $i \in [m]$, we have

$$\mathbb{E}\left[e_i^\top Aq\right] - \frac{B}{T} \le \min_{j\in[n]} \mathbb{E}\left[p^\top Ae_j\right] + \frac{B}{T} \le \mathbb{E}\left[p^\top Aq\right] + \frac{B}{T},$$

and for all $j \in [n]$, we have

$$\mathbb{E}\left[p^\top Aq\right] - \frac{B}{T} \le \max_{i\in[m]} \mathbb{E}\left[e_i^\top Aq\right] - \frac{B}{T} \le \mathbb{E}\left[p^\top Ae_j\right] + \frac{B}{T},$$

Combining these two inequalities, we find that for all $i \in [m]$ and for all $j \in [n]$, it holds that

$$\mathbb{E}\left[e_i^\top Aq\right] - \frac{2B}{T} \le \mathbb{E}\left[p^\top Aq\right] \le \mathbb{E}\left[p^\top Ae_j\right] + \frac{2B}{T}.$$

So far we have followed mostly the same steps as the previous proof, with the main difference being that we are only looking at the expectation. The above result is an approximate Nash equilibrium in expectation. In order to obtain a result that holds with high probability, we would like to use Markov's inequality.

Theorem 5.3 (Markov's inequality)**.** *If X is a non-negative random variable and $a > 0$, then*

$$\mathbb{P}(X \ge a) \le \frac{\mathbb{E}[X]}{a}.$$

Markov's inequality allows us to find an upper bound on a non-negative random variable with high probability if we have an upper bound on the expectation of the random variable. With some rewriting, we can obtain

$$\mathbb{E}\left[p^\top Aq\right] - \min_{j\in[n]} \mathbb{E}\left[p^\top Ae_j\right] \le \frac{2B}{T},$$

and

$$\max_{i\in[m]} \mathbb{E}\left[e_i^\top Aq\right] - \mathbb{E}\left[p^\top Aq\right] \le \frac{2B}{T}.$$

This is close to what we need to use Markov's inequality. In particular, it would be sufficient to find upper bounds for

$$\mathbb{E}\left[p^\top Aq - \min_{j\in[n]} p^\top Ae_j\right]$$

and

$$\mathbb{E}\left[\max_{i\in[m]} e_i^\top Aq - p^\top Aq\right].$$

The two random variables within the expectations above are non-negative and after using Markov's inequality here, the resulting upper bounds would also hold for all $j \in [n], i \in [m]$ respectively, which leads to the desired approximate Nash equilibrium. Unfortunately, we have not found any way to derive upper bounds for these expectations, since we cannot absorb the minimum and maximum into the expectation.

While we have no proof for Conjecture 5.1 and hence no proof that Algorithm 5.3 achieves the desired result, this algorithm is now quite close to the existing algorithm in [14]. In fact, a variant of this algorithm by van Apeldoorn and Gilyén in [4] is even closer and is proven to work, albeit not using the analysis of the multiplicative weights method.

Algorithm 5.4: Altered zero-sum algorithm [4]

```
1 Fix $\eta \leq \frac{1}{2}$. Initialize $P^{(1)} = 1 \in \mathbb{R}^m, Q^{(1)} = 1 \in \mathbb{R}^n$.
2 for $t = 1, 2, \dots, T$ do
3     $p^{(t)} = P^{(t)} / \left\|P^{(t)}\right\|_1, q^{(t)} = Q^{(t)} / \left\|Q^{(t)}\right\|_1$.
4     Sample $a^{(t)} \sim p^{(t)}, b^{(t)} \sim q^{(t)}$.
5     $c^{(t)} = -Ae_{b^{(t)}}, d^{(t)} = A^\top e_{a^{(t)}}$.
6     $P^{(t+1)} = P^{(t)} \exp(-\eta c^{(t)}), Q^{(t+1)} = Q^{(t)} \exp(-\eta d^{(t)})$.
7 Output $(x, y)$, where $x := \frac{1}{T} \sum_{t=1}^{T} e_{a^{(t)}}$ and $y := \frac{1}{T} \sum_{t=1}^{T} e_{b^{(t)}}$.
```

Theorem 5.4 ([4, 14])**.** *For a given $\varepsilon > 0$ and $\delta > 0$, after $T \geq \frac{16 \ln\left(\frac{nm}{\delta}\right)}{\varepsilon^2}$ rounds of Algorithm 5.4, for all $i \in [m]$ and for all $j \in [n]$, we have*

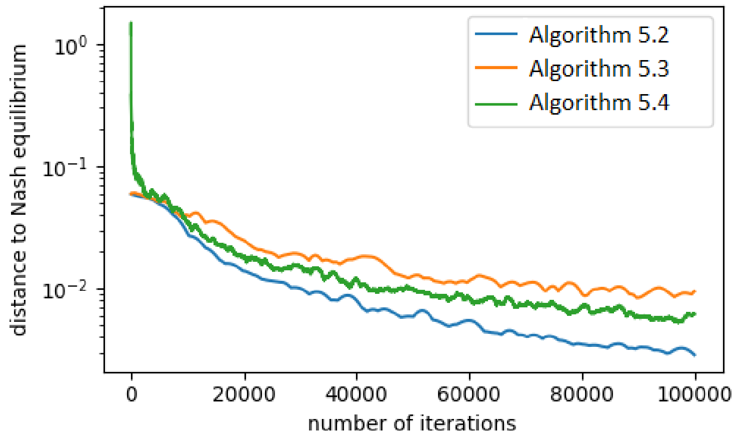
$$e_i^\top Ay - \varepsilon \leq x^\top Ay \leq x^\top Ae_j + \varepsilon,$$

where $x := \frac{1}{T} \sum_{t=1}^{T} e_{a^{(t)}}$ and $y := \frac{1}{T} \sum_{t=1}^{T} e_{b^{(t)}}$.

In particular, this is the variant of the algorithm where η is fixed and not decreasing. We have rewritten this algorithm to match the notation in the previous algorithms, but it is functionally the same. The only differences with Algorithm 5.3 are the parameters, since we now use the same parameter η for both Alice and Bob, and the way the output is computed. The output is no longer based on the probability distributions, but instead the samples taken from them.

Since the outputs are only decided by the samples, each iteration can only update one of the elements in the output probability vectors. Hence you would expect to need many iterations before it approaches the desired Nash equilibrium. This seems like a big disadvantage compared to Algorithm 5.3, which can update all elements of the vectors in one iteration. And yet, Algorithm 5.4 is proven to work, while Algorithm 5.3 is not. The proof that Algorithm 5.4 achieves the desired result, does not use the theorems that come with the multiplicative weights method.

Figure 5.1: Convergence of zero-sum algorithms



5.1.5 Comparison

To compare the zero-sum algorithms, we implemented them in Python (see Appendix A) and compared the results. We set $n = m = 1000$ and ran $T = 100000$ iterations of each of the algorithms for a randomly generated matrix A. Note that at this point, the only difference between Algorithm 5.3 and Algorithm 5.4 is the output, so we used the same run of the algorithm.

The results can be seen in Figure 5.1. The plot depicts how the distance to the Nash equilibrium decreases through the iterations. For a pair of strategies (p, q), this distance is defined as

$$d_{\text{Nash}}(p, q) := \max\{\max_{i\in[m]} e_i^\top Aq - p^\top Aq, p^\top Aq - \min_{j\in[m]} p^\top Ae_j\}.$$

Note that $d_{\text{Nash}}(p, q) \leq \varepsilon$ if and only if (p, q) is an ε-Nash equilibrium. The Nash distance is computed using the average up until iteration t. In other words, this would be the output if the algorithm stopped at iteration $T = t$.

In the figure, we can see that the deterministic algorithm (Algorithm 5.2) converges the fastest, albeit with far more costly iterations. And while Algorithm 5.4 has a much slower start, it does surpass Algorithm 5.3 in the long run. This slow start can be explained by the fact that the output strategies are extremely sparse in early iterations.

One explanation as to why Algorithm 5.3 might not work that well, is that the multiplicative weights method corrects its course based on the samples, even though the probability distributions decide the output. That is, the algorithm is repeatedly adjusting the probability distributions based on the mistakes made by the samples thus far. While this becomes less of an issue over time, the other two algorithms simply do not have this discrepancy. Hence it makes sense that Algorithm 5.3 lags behind. Comparatively, the fact that the output probability vectors are updated with only one element per iteration in Algorithm 5.4 does not seem to be as much of an issue.

5.1.6 Quantum zero-sum LP solvers

In [4], van Apeldoorn and Gilyén developed a quantum version of Algorithm 5.4. It has time complexity $\tilde{O}(\sqrt{n+m}/\varepsilon^3)$. Even in the sampling algorithms above, we required $O(m+n)$ time in each iteration to compute the cost vectors and to update the weight vectors. In order to achieve this sublinear dependency on m and n, these computations can only be done implicitly.

Note that all we need to implement is efficient sampling from $p^{(t)}$ and $q^{(t)}$, since the output (x, y) is only decided by these samples. We can rewrite $p^{(t)}$ and $q^{(t)}$ as

$$\begin{aligned} p^{(t)} &= \frac{\exp(\eta A y^{(t)})}{\left\|\exp(\eta A y^{(t)})\right\|_1} = G(Ay^{(t)}), \\ q^{(t)} &= \frac{\exp(\eta A^\top x^{(t)})}{\left\|\exp(\eta A^\top x^{(t)})\right\|_1} = G(A^\top x^{(t)}), \end{aligned}$$

where $x^{(t)} := \sum_{\tau=1}^t e_{a^{(t)}}, y^{(t)} := \sum_{\tau=1}^t e_{b^{(t)}}$ and $G(v) := \frac{\exp(v)}{\|\exp(v)\|_1}$ denotes the Gibbs distribution corresponding to v. Note that $x^{(t)}$ and $y^{(t)}$ are t-sparse and can therefore be stored and updated sublinearly in m and n.

The Gibbs distributions $p^{(t)}$ and $q^{(t)}$ can be efficiently sampled on a quantum computer, using amplitude estimation, amplitude amplification and other subroutines. We also need to assume access to an oracle that returns the queried matrix entry. Note that the Gibbs sampling here is a simpler special case of the general Gibbs sampler in Definition 3.1, since we have a vector in the exponent, instead of a matrix.

A recent result [8] improves the time complexity to $\tilde{O}(\sqrt{m+n}/\varepsilon^{2.5} + 1/\varepsilon^3)$ by using different quantum data structures for dynamic Gibbs sampling.

5.2 Semidefinite programs

Semidefinite programs (SDPs) are more general than LPs, since they contain not only linear constraints, but also the constraint that the variable matrix X is positive semidefinite, denoted as $X \succeq 0$. We write the standard form of an SDP as follows.

$$\begin{aligned} \max\ & \operatorname{Tr}(CX) \\ \text{s.t.}\ & \operatorname{Tr}(A_i X) \le b_i \text{ for all } i \in [m], \\ & X \succeq 0. \end{aligned}$$

Here, $X \in \mathbb{R}^{n\times n}$ is the variable matrix. The symmetric constraint matrices $A_1, \ldots, A_m \in [-1,1]^{n\times n}$, objective matrix $C \in [-1,1]^{n\times n}$ and the vector $b \in \mathbb{R}^n$

are given. Note that for symmetric matrices A, B, it holds that $\operatorname{Tr}(AB) = \sum_{i,j\in[n]} A_{ij}B_{ij}$, the Frobenius product of A and B. In the event that all matrices $C, A_1, \ldots, A_m$ are diagonal matrices, we simply obtain a standard-form LP.

5.2.1 Basic SDP solver

As shown in [6], SDPs can be approximately solved using matrix multiplicative weights. Since our SDP solver in the next section has a different structure, we will not go into too much detail here.

The SDP solver in [6] uses a primal-dual approach. The standard form of SDP we mentioned before is known as the primal form of the SDP. The dual form of this SDP is the following.

$$\begin{aligned} &\min\ b^\top y \\ &\text{s.t. } \sum_{j=1}^{m} y_j A_j - C \succeq 0, \\ &\qquad y \geq 0. \end{aligned}$$

If we assume that for some $R, r \geq 1$, it holds that $\operatorname{Tr}(X) \leq R$ is the first constraint in the primal and $\|y\|_1 \leq r$ holds for the dual optimizer y, then *strong duality* holds, which means the primal and dual problems have the same optimal values. The goal is to find an ε-approximation of this optimal value. As shown in [5], the parameters R, r and ε^{-1} can be seen as equivalent. That is, any two of the parameters can be made constant by increasing the third. Due to this, the complexity is expressed in terms of the combined parameter $\gamma := \frac{Rr}{\varepsilon}$.

In each iteration of the algorithm, the weight matrix $\rho^{(t)}$ is used to create the intermediate solution $X^{(t)} = R\rho^{(t)}$ to the primal, which is fed to the oracle. The oracle then outputs an intermediate solution $\hat{y}$ to the dual, that meets certain requirements. The cost matrix $M^{(t)}$ is based on how well $\hat{y}$ satisfies the constraints of the SDP, which is then used to update $\rho^{(t+1)}$.

The algorithm uses $T = \lceil \frac{9R^2 \ln n}{\varepsilon^2} \rceil$ iterations. While no general oracle is specified in [6], it is shown in [5] that a general time complexity of $\tilde{O}\left(nms\gamma^4 + ns\gamma^7\right)$ can be achieved, where s is the *sparsity* of the input matrices, which is the maximum number of non-zero entries in a row or column.

5.2.2 Deterministic zero-sum SDP solver

The algorithm we constructed uses a similar approach to the zero-sum algorithms for solving LPs. Instead of running two simultaneous instances of the multiplicative weights method, the following algorithm consists of one instance of multiplicative weights and one instance of matrix multiplicative weights, running simultaneously.

When viewed as a zero-sum game, Alice has m moves to choose from as before, but Bob can now choose any density matrix $\rho \in \mathbb{R}^{n\times n}$. Each of Alice's moves is associated with one of the matrices A_i. If Alice plays $i \in [m]$ and Bob

Algorithm 5.5: SDP multiplicative weights algorithm

```
1 Fix $\eta \leq \frac{1}{2}, \theta \leq \frac{1}{2}$. Initialize $P^{(1)} = 1 \in \mathbb{R}^m, Q^{(1)} = I \in \mathbb{R}^{n \times n}$.
2 for $t = 1, 2, \ldots, T$ do
3     $p^{(t)} = P^{(t)} / \left\| P^{(t)} \right\|_1, q^{(t)} = Q^{(t)} / \operatorname{Tr}\left(Q^{(t)}\right)$.
4     $c_i^{(t)} = -\operatorname{Tr}\left(A_i q^{(t)}\right)$ for all $i \in [m]$.
5     $d^{(t)} = \sum_{j=1}^m p_j^{(t)} A_j$.
6     $P^{(t+1)} = P^{(t)} \exp(-\eta c^{(t)}) = \exp\left(-\eta \sum_{\tau=1}^t c^{(\tau)}\right)$.
7     $Q^{(t+1)} = \exp\left(-\theta \sum_{\tau=1}^t d^{(\tau)}\right)$.
```

plays ρ, then Alice receives payoff $\operatorname{Tr}(A_i\rho)$, while Bob receives $-\operatorname{Tr}(A_i\rho)$. If Alice uses a randomized strategy $p \in \Delta^m$ instead, then her expected payoff is $\sum_{i=1}^m p_i \operatorname{Tr}(A_i\rho)$.

Alice's side of the algorithm is the same as Algorithm 5.2, apart from the definition of $c^{(t)}$, and thus an instance of the multiplicative weights method. Bob's side of the algorithm, however, follows the *matrix* multiplicative weights method instead. It has a weight matrix $Q^{(t)}$ instead of a weight vector and a density matrix $q^{(t)}$ instead of a probability vector. As a consequence, we can use Theorem 2.1 and Theorem 2.4 to analyze the algorithm.

Theorem 5.5. *For a given $\varepsilon > 0$, after $T \geq \frac{16\ln(\max\{m,n\})}{\varepsilon^2}$ rounds of Algorithm 5.5 with parameters $\eta = \sqrt{\frac{\ln m}{T}}$ and $\theta = \sqrt{\frac{\ln n}{T}}$, for all $i \in [m]$ and for all density matrices ρ, we have*

$$Tr(A_i q) - \varepsilon \leq \sum_{j=1}^m p_j \, Tr(A_j q) \leq \sum_{j=1}^m p_j \, Tr(A_j \rho) + \varepsilon.$$

where $p := \frac{1}{T}\sum_{t=1}^T p^{(t)}$ and $q := \frac{1}{T}\sum_{t=1}^T q^{(t)}$.

Proof. Since Algorithm 5.5 can be viewed as a simultaneous instance of the multiplicative weights method and the matrix multiplicative weights algorithm, we get the following guarantees from Theorem 2.1 and Theorem 2.4. For all $i \in [m]$ and for all unit vectors $v \in \mathbb{S}^{n-1}$ respectively, we have

$$\sum_{t=1}^T c^{(t)} \cdot p^{(t)} \leq \sum_{t=1}^T c_i^{(t)} + \eta \sum_{t=1}^T (c^{(t)})^2 \cdot p^{(t)} + \frac{\ln m}{\eta},$$
$$\sum_{t=1}^T \operatorname{Tr}\left(d^{(t)} q^{(t)}\right) \leq \sum_{t=1}^T v^\top d^{(t)} v + \theta \sum_{t=1}^T \operatorname{Tr}\left((d^{(t)})^2 q^{(t)}\right) + \frac{\ln n}{\theta}.$$

We pick $\eta = \sqrt{\frac{\ln m}{T}}$ and $\theta = \sqrt{\frac{\ln n}{T}}$ such that each of the final two terms in both inequalities is upper bounded by $\sqrt{T\ln(\max\{m,n\})}$. Let $B \geq 2\sqrt{T\ln(\max\{m,n\})}$

be an upper bound on the final two terms in both of the above inequalities. Note that

$$\sum_{t=1}^{T} v^\top d^{(t)} v = \sum_{t=1}^{T} \operatorname{Tr}\left(d^{(t)} vv^\top\right) = \sum_{t=1}^{T} \operatorname{Tr}\left(\sum_{j=1}^{m} p_j^{(t)} A_j vv^\top\right) = \sum_{t=1}^{T}\sum_{j=1}^{m} p_j^{(t)} \operatorname{Tr}\left(A_j vv^\top\right).$$

With some further rewriting, we obtain

$$\sum_{t=1}^{T}\sum_{j=1}^{T} p_j^{(t)} \operatorname{Tr}\left(A_j q^{(t)}\right) \geq \sum_{t=1}^{T} \operatorname{Tr}\left(A_i q^{(t)}\right) - B,$$
$$\sum_{t=1}^{T}\sum_{j=1}^{T} p_j^{(t)} \operatorname{Tr}\left(A_j q^{(t)}\right) \leq \sum_{t=1}^{T}\sum_{j=1}^{m} p_j^{(t)} \operatorname{Tr}\left(A_j vv^\top\right) + B.$$

We can now combine the two inequalities and drop the middle expression. Dividing by T and substituting $p := \frac{1}{T}\sum_{t=1}^{T} p^{(t)}$ and $q := \frac{1}{T}\sum_{t=1}^{T} q^{(t)}$, we find

$$\operatorname{Tr}\left(A_i q\right) - \frac{B}{T} \leq \sum_{j=1}^{m} p_j \operatorname{Tr}\left(A_j vv^\top\right) + \frac{B}{T}.$$

Note that the above still holds for all $i \in [m]$ and for all unit vectors $v \in \mathbb{S}^{n-1}$. Any density matrix $\rho \in \mathbb{R}^{n\times n}$ can be written as a convex combination of pure states $vv^\top$ where v is a unit vector. Hence, for all $i \in [m]$ and for all density matrices ρ, we have

$$\operatorname{Tr}\left(A_i q\right) - \frac{B}{T} \leq \sum_{j=1}^{m} p_j \operatorname{Tr}\left(A_j \rho\right) + \frac{B}{T}.$$

This means that for all $i \in [m]$, we have

$$\operatorname{Tr}\left(A_i q\right) - \frac{B}{T} \leq \min_{\rho} \sum_{j=1}^{m} p_j \operatorname{Tr}\left(A_j \rho\right) + \frac{B}{T} \leq \sum_{j=1}^{m} p_j \operatorname{Tr}\left(A_j q\right) + \frac{B}{T},$$

and for all density matrices ρ, we have

$$\sum_{j=1}^{m} p_j \operatorname{Tr}\left(A_j q\right) - \frac{B}{T} \leq \max_{i\in[m]} \operatorname{Tr}\left(A_i q\right) - \frac{B}{T} \leq \sum_{j=1}^{m} p_j \operatorname{Tr}\left(A_j \rho\right) + \frac{B}{T}.$$

Combining these two inequalities, we find that for all $i \in [m]$ and for all density matrices ρ, it holds that

$$\operatorname{Tr}\left(A_i q\right) - \frac{2B}{T} \leq \sum_{j=1}^{m} p_j \operatorname{Tr}\left(A_j q\right) \leq \sum_{j=1}^{m} p_j \operatorname{Tr}\left(A_j \rho\right) + \frac{2B}{T}.$$

We get the desired result if $\frac{2B}{T} \leq \varepsilon$, or equivalently, $B \leq \frac{\varepsilon T}{2}$. Since we assumed $B \geq 2\sqrt{T\ln(\max\{m,n\})}$, we need to choose T such that $2\sqrt{T\ln(\max\{m,n\})} \leq \frac{\varepsilon T}{2}$, which is true for $T \geq \frac{16\ln(\max\{m,n\})}{\varepsilon^2}$. $\square$

5.2.3 Zero-sum SDP solver using sampling

As with Algorithm 5.3, we can adapt the algorithm to use sampling. However, this time around, we will only sample on Alice's side.

Algorithm 5.6: SDP multiplicative weights algorithm with sampling

```
1 Fix $\eta \le \frac{1}{2}, \theta \le \frac{1}{2}$. Initialize $P^{(1)} = 1 \in \mathbb{R}^m, Q^{(1)} = I \in \mathbb{R}^{n\times n}$.
2 for $t = 1, 2, \ldots, T$ do
3     $p^{(t)} = P^{(t)} / \left\|P^{(t)}\right\|_1, q^{(t)} = Q^{(t)} / \mathrm{Tr}\left(Q^{(t)}\right)$.
4     $c_i^{(t)} = -\mathrm{Tr}\left(A_i q^{(t)}\right)$ for all $i \in [m]$.
5     Sample $a^{(t)} \sim p^{(t)}$, set $d^{(t)} = A_{a^{(t)}}$.
6     $P^{(t+1)} = P^{(t)} \exp(-\eta c^{(t)}) = \exp\left(-\eta \sum_{\tau=1}^{t} c^{(\tau)}\right)$.
7     $Q^{(t+1)} = \exp\left(-\theta \sum_{\tau=1}^{t} d^{(\tau)}\right)$.
```

Unfortunately, we have not been able to prove that Algorithm 5.6 achieves the desired result, as described in the conjecture below. We run into a similar issue as we did for Algorithm 5.3. Due to the random sampling done in the algorithm, we once again cannot expect the algorithm to get the desired result with probability 1. Instead, we attempted to prove that it holds with arbitrarily high probability.

Conjecture 5.2. *For given $\varepsilon > 0$ and $\delta > 0$, there exists a $T' = O\left(\frac{\log(mn)}{\varepsilon^2}\right)$, such that after $T \ge T'$ rounds of Algorithm 5.6 with parameters $\eta = \sqrt{\frac{\ln m}{T}}$ and $\theta = \sqrt{\frac{\ln n}{T}}$, with probability at least $1-\delta$, for every $i \in [m]$ and for all density matrices ρ, we have*

$$Tr(A_i q) - \varepsilon \le \sum_{j=1}^{m} p_j \, Tr(A_j q) \le \sum_{j=1}^{m} p_j \, Tr(A_j \rho) + \varepsilon.$$

where $p := \frac{1}{T}\sum_{t=1}^{T} p^{(t)}$ and $q := \frac{1}{T}\sum_{t=1}^{T} q^{(t)}$.

Since Algorithm 5.6 can be viewed as a simultaneous instance of the multiplicative weights method and the matrix multiplicative weights algorithm, we get the following guarantees from Theorem 2.1 and Theorem 2.4. For all $i \in [m]$ and for all unit vectors $v \in \mathbb{S}^{n-1}$ respectively, we have

$$\sum_{t=1}^{T} c^{(t)} \cdot p^{(t)} \le \sum_{t=1}^{T} c_i^{(t)} + \eta \sum_{t=1}^{T} (c^{(t)})^2 \cdot p^{(t)} + \frac{\ln m}{\eta},$$
$$\sum_{t=1}^{T} \mathrm{Tr}\left(d^{(t)} q^{(t)}\right) \le \sum_{t=1}^{T} v^\top d^{(t)} v + \theta \sum_{t=1}^{T} \mathrm{Tr}\left((d^{(t)})^2 q^{(t)}\right) + \frac{\ln n}{\theta}.$$

We pick $\eta = \sqrt{\frac{\ln m}{T}}$ and $\theta = \sqrt{\frac{\ln n}{T}}$ such that each of the final two terms in both inequalities is upper bounded by $\sqrt{T \ln(\max\{m, n\})}$. Let $B \geq 2\sqrt{T \ln(\max\{m, n\})}$ be an upper bound on the final two terms in both of the above inequalities. With some further rewriting, we obtain

$$\sum_{t=1}^{T} \sum_{j=1}^{m} p_j^{(t)} \operatorname{Tr}\left(A_j q^{(t)}\right) \geq \sum_{t=1}^{T} \operatorname{Tr}\left(A_i q^{(t)}\right) - B,$$
$$\sum_{t=1}^{T} \operatorname{Tr}\left(A_{a^{(t)}} q^{(t)}\right) \leq \sum_{t=1}^{T} \operatorname{Tr}\left(A_{a^{(t)}} v v^\top\right) + B.$$

Unlike in the previous proof, the left-hand sides are not equal here. We take the expectation over the $p^{(t)}$ and $a^{(t)}$, in other words the expectation over a run of the algorithm, to combine the inequalities and obtain

$$\begin{aligned}\mathbb{E}\left[\sum_{t=1}^{T} \operatorname{Tr}\left(A_i q^{(t)}\right)\right] - B &\leq \mathbb{E}\left[\sum_{t=1}^{T} \sum_{j=1}^{m} p_j^{(t)} \operatorname{Tr}\left(A_j q^{(t)}\right)\right] \\ &\leq \mathbb{E}\left[\sum_{t=1}^{T} \sum_{j=1}^{m} p_j^{(t)} \operatorname{Tr}\left(A_j v v^\top\right)\right] + B.\end{aligned}$$

Note that the above still holds for all $i \in [m]$ and for all unit vectors $v \in \mathbb{S}^{n-1}$. Recall that any density matrix $\rho \in \mathbb{R}^{n \times n}$ can be written as a convex combination of pure states $vv^\top$ where v is a unit vector. We drop the middle expression, substitute $p := \frac{1}{T}\sum_{t=1}^{T} p^{(t)}$ and $q := \frac{1}{T}\sum_{t=1}^{T} q^{(t)}$, and divide by T. For all $i \in [m]$ and for all density matrices ρ, we have

$$\mathbb{E}\left[\operatorname{Tr}\left(A_i q\right)\right] - \frac{B}{T} \leq \mathbb{E}\left[\sum_{j=1}^{m} p_j \operatorname{Tr}\left(A_j \rho\right)\right] + \frac{B}{T}.$$

Now, for all $i \in [m]$, we have

$$\mathbb{E}\left[\operatorname{Tr}\left(A_i q\right)\right] - \frac{B}{T} \leq \min_{\rho} \mathbb{E}\left[\sum_{j=1}^{m} p_j \operatorname{Tr}\left(A_j \rho\right)\right] + \frac{B}{T} \leq \mathbb{E}\left[\sum_{j=1}^{m} p_j \operatorname{Tr}\left(A_j q\right)\right] + \frac{B}{T},$$

and for all density matrices ρ, we have

$$\mathbb{E}\left[\sum_{j=1}^{m} p_j \operatorname{Tr}\left(A_j q\right)\right] - \frac{B}{T} \leq \max_{i \in [m]} \mathbb{E}\left[\operatorname{Tr}\left(A_i q\right)\right] - \frac{B}{T} \leq \mathbb{E}\left[\sum_{j=1}^{m} p_j \operatorname{Tr}\left(A_j \rho\right)\right] + \frac{B}{T}.$$

Combining these two inequalities, we find that for all $i \in [m]$ and for all density matrices ρ, it holds that

$$\mathbb{E}\left[\operatorname{Tr}\left(A_i q\right)\right] - \frac{2B}{T} \leq \mathbb{E}\left[\sum_{j=1}^{m} p_j \operatorname{Tr}\left(A_j q\right)\right] \leq \mathbb{E}\left[\sum_{j=1}^{m} p_j \operatorname{Tr}\left(A_j \rho\right)\right] + \frac{2B}{T}.$$

So far we have followed mostly the same steps as the previous proof, with the main difference being that we are only looking at the expectation. The above result is an approximate Nash equilibrium in expectation. In order to obtain a result that holds with high probability, we would like to use Markov's inequality (Theorem 5.3), which allows us to find an upper bound on a non-negative random variable with high probability if we have an upper bound on the expectation of the random variable. With some rewriting, we can obtain

$$\mathbb{E}\left[\sum_{j=1}^{m} p_j \operatorname{Tr}\left(A_j q\right)\right] - \min_{\rho} \mathbb{E}\left[\sum_{j=1}^{m} p_j \operatorname{Tr}\left(A_j \rho\right)\right] \leq \frac{2B}{T},$$

and

$$\max_{i \in [m]} \mathbb{E}\left[\operatorname{Tr}\left(A_i q\right)\right] - \mathbb{E}\left[\sum_{j=1}^{m} p_j \operatorname{Tr}\left(A_j q\right)\right] \leq \frac{2B}{T}.$$

This is close to what we need to use Markov's inequality. In particular, it would be sufficient to find upper bounds for

$$\mathbb{E}\left[\sum_{j=1}^{m} p_j \operatorname{Tr}\left(A_j q\right) - \min_{\rho} \sum_{j=1}^{m} p_j \operatorname{Tr}\left(A_j \rho\right)\right]$$

and

$$\mathbb{E}\left[\max_{i \in [m]} \operatorname{Tr}\left(A_i q\right) - \sum_{j=1}^{m} p_j \operatorname{Tr}\left(A_j q\right)\right].$$

The two random variables within the expectations above are non-negative and after using Markov's inequality here, the resulting upper bounds would also hold for all density matrices ρ and for all $i \in [m]$ respectively, which leads to desired approximate Nash equilibrium. Unfortunately, we have not found any way to derive upper bounds for these expectations, since we cannot absorb the minimum and maximum into the expectation.

5.2.4 Reduction

The last two algorithms do not solve a standard-form SDP directly, but instead give an approximate Nash equilibrium for a certain zero-sum game. From Bob's perspective, we can view the game as

$$\min_{q} \max_{p} \sum_{j=1}^{m} p_j \operatorname{Tr}\left(A_j q\right) = \min_{q} \max_{i} \operatorname{Tr}\left(A_i q\right).$$

We can now write this zero-sum game as an SDP.

$$\begin{aligned} \min\ & \lambda \\ \text{s.t.}\ & \operatorname{Tr}(A_i X) \leq \lambda \text{ for all } i \in [m], \\ & X \succeq 0, \operatorname{Tr}(X) = 1, \\ & \lambda \in \mathbb{R}. \end{aligned}$$

We will give a reduction from standard-form SDPs to this particular form of SDP. We start with an instance of a standard-form SDP and add the constraint $\operatorname{Tr}(X) = d$ for some d. We also assume without loss of generality that we start with a minimization problem, which can be done by multiplying C by -1.

$$\begin{aligned} \min\ & \operatorname{Tr}(CX) \\ \text{s.t.}\ & \operatorname{Tr}(A_i X) \leq b_i \text{ for all } i \in [m], \\ & X \succeq 0, \operatorname{Tr}(X) = d. \end{aligned}$$

Since we generally do not know the trace of the optimal solution, we simply try various possibilities for d. One way to do this is to set d to values of the form $\leq (1-\alpha)^{\ell}$, where $\alpha > 0$ is small, $\ell \in \mathbb{Z}$. To keep it finite, we also need an upper bound and a non-trivial lower bound for d. Unfortunately, we cannot do a binary search to speed up this process. We can also not guarantee that we will find a feasible solution even if one exists, since it is possible that only solutions of a specific trace are feasible.

For the next step, we set $A'_i := dA_i + dC - b_i I$ for all i. For the variable, we use X', which must have $\operatorname{Tr}(X') = 1$. To obtain the solution X for the original SDP, we can simply multiply by d. We substitute $\lambda = d\operatorname{Tr}(CX') = \operatorname{Tr}(CX)$, which is the value we want to minimize. Note that

$$\begin{aligned} \operatorname{Tr}(A'_i X') &= \operatorname{Tr}\left(dA_i \frac{X}{d}\right) + \operatorname{Tr}\left(dC \frac{X}{d}\right) - \operatorname{Tr}\left(b_i I \frac{X}{d}\right) \\ &= \operatorname{Tr}(A_i X) + \operatorname{Tr}(CX) - b_i. \end{aligned}$$

It follows that $\operatorname{Tr}(A_i X) \leq b_i$ if and only if $\operatorname{Tr}(A'_i X') \leq \operatorname{Tr}(CX) = \lambda$. Hence, with this step, the problem becomes equivalent to the following.

$$\begin{aligned} \min\ & \lambda \\ \text{s.t.}\ & \operatorname{Tr}(A'_i X') \leq \lambda \text{ for all } i \in [m], \\ & X' \succeq 0, \operatorname{Tr}(X') = 1, \\ & \lambda \in \mathbb{R}. \end{aligned}$$

This problem differs from the zero-sum game only in the names of the variables, so this completes the reduction.

5.2.5 Quantum SDP solvers

Brandão and Svore developed a quantum SDP solver [9] that builds upon the basic matrix multiplicative weights SDP solver from Section 5.2.1. It uses a

Gibbs sampler to prepare the density matrix $\rho^{(t)}$ as a Gibbs state, which provides a speedup in terms of n. It also uses a subroutine similar to Grover's algorithm, which provides a speedup in terms of m, leading to a time complexity of $\tilde{O}(\sqrt{mn} \cdot s^2\gamma^{32})$, where $\gamma = \frac{Rr}{\varepsilon}$ as before.

In [5], van Apeldoorn et al. improve upon this result by using Theorem 3.4 for the oracle, as well as various other modifications. The result is a simpler algorithm with time complexity $\tilde{O}(\sqrt{mn} \cdot s^2\gamma^8)$. This is a significant improvement in the dependence on γ.

In [3], van Apeldoorn and Gilyén further improved their result by switching to the quantum operator input model, in which the input matrices are given as quantum states known as *block-encodings*. Simply put, this means that each matrix A is contained within a unitary U. The operator input model uses a different parameter α, but it implies results for the sparse model if we take $\alpha = s$, so we can still compare the complexities. With improved search and minimum-finding techniques, the algorithm achieves time complexity $\tilde{O}((\sqrt{m}+\sqrt{n}\cdot\gamma)\alpha\gamma^4)$.

5.2.6 Quantum zero-sum SDP solver

We hoped to develop a quantization of Algorithm 5.5 of the unproven Algorithm 5.6, but unfortunately, we were not able to.

The main difficulty comes from Alice's cost vector $c^{(t)}$, which is defined as $c_i^{(t)} = -\mathrm{Tr}\left(A_i q^{(t)}\right)$ for all $i \in [m]$. We have seen in Theorem 3.4 that such values can be efficiently computed. However, we did not find any way to avoid computing this value for all $i \in [m]$, which means the time complexity is still worse than all aforementioned quantum SDP solvers, which are sublinear in m. Additionally, we need to account for the estimation error that occurs, which complicates the proof.

Alice's cost vector $c^{(t)}$ is used to compute the probability distribution $p^{(t)}$, which is a Gibbs distribution. While we have seen various Gibbs samplers, none of them fit this particular situation. Bob's density matrix $q^{(t)}$ is a Gibbs state and fits Definition 3.1 well, but it is still dependent on the more problematic $p^{(t)}$.

Chapter 6

Conclusion

We have seen the application of the multiplicative weights method on machine learning in the form of boosting algorithms, as well as for solving LPs and SDPs. We have also looked at how quantum computers can be used to improve the results. While we have not made any major discoveries, we have made several noteworthy observations.

Boosting algorithms are a well-known tool within machine learning, with which we can create a strong learner from a weak learner. All boosting algorithms use the multiplicative weights method. Although AdaBoost is the more commonly used boosting algorithm, SmoothBoost was a better fit for quantization. However, while the analysis suggests a quadratic improvement over classical algorithms, this is misleading since much of the complexity is hidden within the weak learner. We found a randomized classical algorithm for which the same type of analysis suggests an exponential improvement over existing methods, which is clearly untrue. This further shows how misleading the analysis for the quantum algorithms is.

For solving LPs, the multiplicative weights method can be used directly, but it is outclassed by a randomized algorithm that solves LPs by solving a zero-sum game. This algorithm is sublinear in the input size. While the similarity to multiplicative weights is obvious, we showed that this algorithm for zero-sum games can in fact be seen as two simultaneous instances of multiplicative weights. While we were able to prove that a deterministic version of the algorithm works using the analysis of multiplicative weights, we were not able to do so for the existing randomized version, for which a proof exists that does not use any theorems related to the multiplicative weights method. A quantum version of this algorithm also exists and yields a big improvement.

For solving SDPs, existing algorithms use matrix multiplicative weights directly. Quantization of these algorithms also leads to big improvements. We found an alternative method, inspired by the zero-sum algorithms for LPs, that consists of an instance of multiplicative weights and an instance of the matrix multiplicative weights. Unfortunately, we were once again unable to prove that the randomized version works, only the deterministic one. We also did not find any improvement from quantization, although this is something that could be explored further.

Multiplicative weights is used in many other fields, but we were not able to look into them due to lack of time. At least in the case of LPs and SDPs, quantization of multiplicative weights algorithms has led to large improvements in time complexity, so it seems likely that quantization will be similarly helpful in some of the other fields where multiplicative weights is already used classically.

Bibliography

[1] Ilan Adler. “The equivalence of Linear Programs and Zero-sum Games”. In: *International Journal of Games Theory* Volume 42:165-177 (Feb. 2013).

[2] Joran van Apeldoorn. “A quantum view on convex optimization”. PhD thesis. University of Amsterdam, 2020.

[3] Joran van Apeldoorn and András Gilyén. “Improvements in Quantum SDP-Solving with Applications”. In: *Proceedings of the 46th International Colloquium on Automata, Languages, and Programming.* Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik GmbH, Wadern/Saarbruecken, Germany, 2019.

[4] Joran van Apeldoorn and András Gilyén. *Quantum algorithms for zero-sum games.* 2019. arXiv: `1904.03180 [quant-ph]`.

[5] Joran van Apeldoorn, András Gilyén, Sander Gribling, and Ronald de Wolf. “Quantum SDP-Solvers: Better upper and lower bounds”. In: *Quantum* 4 (Feb. 2020), p. 230.

[6] Sanjeev Arora, Elad Hazan, and Satyen Kale. “The Multiplicative Weights Update Method: a Meta-Algorithm and Applications”. In: *Theory of Computing* 8.6 (2012), pp. 121–164.

[7] Srinivasan Arunachalam and Reevu Maity. “Quantum Boosting”. In: *Proceedings of the 37th International Conference on Machine Learning.* Ed. by Hal Daumé III and Aarti Singh. Vol. 119. Proceedings of Machine Learning Research. PMLR, 13–18 Jul 2020, pp. 377–387.

[8] Adam Bouland, Yosheb Getachew, Yujia Jin, Aaron Sidford, and Kevin Tian. *Quantum Speedups for Zero-Sum Games via Improved Dynamic Gibbs Sampling.* 2023. arXiv: `2301.03763 [quant-ph]`.

[9] Fernando Brandaø and Krysta Svore. “Quantum Speed-Ups for Solving Semidefinite Programs”. In: *IEEE 58th Annual Symposium on Foundations of Computer Science.* Oct. 2017, pp. 415–426.

[10] Gilles Brassard, Peter Høyer, Michele Mosca, and Alain Tapp. “Quantum amplitude amplification and estimation”. In: *Quantum Computation and Information* (2002), pp. 53–74.

[11] Gilles Brassard, Peter Høyer, and Alain Tapp. "Quantum counting". In: *Automata, Languages and Programming.* Ed. by Kim G. Larsen, Sven Skyum, and Glynn Winskel. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 820–831.

[12] Peter J. Forrester and Colin J. Thompson. "The Golden-Thompson inequality: Historical aspects and random matrix applications". In: *Journal of Mathematical Physics* 55.2 (Feb. 2014), p. 023503.

[13] Yoav Freund and Robert E Schapire. "A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting". In: *Journal of Computer and System Sciences* 55.1 (1997), pp. 119–139.

[14] Michael D. Grigoriadis and Leonid G. Khachiyan. "A sublinear-time randomized approximation algorithm for matrix games". In: *Operations Research Letters* 18.2 (1995), pp. 53–58.

[15] Lov K. Grover. "A Fast Quantum Mechanical Algorithm for Database Search". In: *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing.* STOC '96. Philadelphia, Pennsylvania, USA: Association for Computing Machinery, 1996, pp. 212–219.

[16] Adam Izdebski and Ronald de Wolf. *Improved Quantum Boosting.* 2020. arXiv: `2009.08360 [quant-ph]`.

[17] Robert E. Schapire and Yoav Freund. "Boosting: Foundations and Algorithms". In: *Kybernetes* 42.1 (2013), pp. 164–166.

[18] Rocco A. Servedio. "Smooth Boosting and Learning with Malicious Noise". In: *Journal of Machine Learning Research* 4 (2003), pp. 633–648.

[19] Shai Shalev-Shwartz and Shai Ben-David. *Understanding Machine Learning: From Theory to Algorithms.* Cambridge University Press, 2014.

[20] Peter W. Shor. "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer". In: *SIAM Journal on Computing* 26.5 (Oct. 1997), pp. 1484–1509.

[21] L. G. Valiant. "A theory of the learnable". In: *Commun. ACM* 27 (1984), pp. 1134–1142.

[22] Ronald de Wolf. *Quantum Computing: Lecture Notes.* 2022. arXiv: `1907.09415 [quant-ph]`.

Appendix A

Code

```
import numpy as np
import matplotlib.pyplot as plt

#takes a random sample from a distribution, returns the index
def sample(dist):
    r = np.random.rand()
    for i in range(dist.size):
        if r < dist[i]:
            return i
        else:
            r -= dist[i]
    return dist.size - 1

#the zero-sum algorithm without sampling
def zerosumdet(m, n, T, A):
    eta = np.sqrt(np.log(m) / T)
    theta = np.sqrt(np.log(n) / T)

    Pt = np.zeros((T + 1, m))
    Pt[0] = np.ones(m)
    pt = np.zeros((T, m))
    ct = np.zeros((T, m))

    Qt = np.zeros((T + 1, n))
    Qt[0] = np.ones(n)
    qt = np.zeros((T, n))
    dt = np.zeros((T, n))

    for t in range(T):
        if (t + 1) % 10000 == 0:
            print("iteration", t + 1)

        pt[t] = Pt[t] / np.linalg.norm(Pt[t], ord=1)
        qt[t] = Qt[t] / np.linalg.norm(Qt[t], ord=1)

        ct[t] = -np.matmul(A, qt[t])
        dt[t] = np.matmul(pt[t], A)

        Pt[t + 1] = Pt[t] * np.exp(-eta * ct[t])
        Qt[t + 1] = Qt[t] * np.exp(-theta * dt[t])

    return pt, qt

#the zero-sum algorithm with sampling
def zerosumsamp(m, n, T, A):
    eta = np.sqrt(np.log(m) / T)
    theta = np.sqrt(np.log(n) / T)
    AT = np.transpose(A)

    Pt = np.zeros((T + 1, m))
    Pt[0] = np.ones(m)
    pt = np.zeros((T, m))
    at = np.zeros(T)
    eat = np.zeros((T, m))
    ct = np.zeros((T, m))

    Qt = np.zeros((T + 1, n))
    Qt[0] = np.ones(n)
    qt = np.zeros((T, n))
    bt = np.zeros(T)
    ebt = np.zeros((T, n))
    dt = np.zeros((T, n))

    for t in range(T):
        if (t + 1) % 10000 == 0:
            print("iteration", t + 1)

        pt[t] = Pt[t] / np.linalg.norm(Pt[t], ord=1)
        qt[t] = Qt[t] / np.linalg.norm(Qt[t], ord=1)

        at[t] = sample(pt[t])
        eat[t, int(at[t])] = 1
        bt[t] = sample(qt[t])
        ebt[t, int(bt[t])] = 1

        ct[t] = -AT[int(bt[t])] #alternatively, -np.matmul(A, ebt[t])
        dt[t] = A[int(at[t])] #alternatively, np.matmul(eat[t], A)

        Pt[t + 1] = Pt[t] * np.exp(-eta * ct[t])
        Qt[t + 1] = Qt[t] * np.exp(-theta * dt[t])

    return pt, qt, eat, ebt

m = 1000
n = 1000
T = 100000
#generates random matrix with elements in [-1,1]
A = 2 * np.random.rand(m, n) - 1

print("running deterministic algorithm")
pt1, qt1 = zerosumdet(m, n, T, A)
print("running sampling algorithm")
pt2, qt2, eat, ebt = zerosumsamp(m, n, T, A)

spt = np.zeros(m)
sqt = np.zeros(n)

err1 = np.zeros(T)

print("evaluating deterministic algorithm")
for t in range(T):
    if (t + 1) % 10000 == 0:
        print("iteration", t + 1)

    spt += pt1[t]
    sqt += qt1[t]

    p = spt / (t + 1)
    q = sqt / (t + 1)

    eiAq = np.matmul(A, q)
    pAej = np.matmul(p, A)
    pAq = np.dot(p, eiAq)

    err1[t] = max(max(eiAq) - pAq, pAq - min(pAej))

spt = np.zeros(m)
sqt = np.zeros(n)

seat = np.zeros(m)
sebt = np.zeros(n)
```

```
err2 = np.zeros(T)
err3 = np.zeros(T)

print("evaluating sampling algorithm")
for t in range(T):
    if (t + 1) % 10000 == 0:
        print("iteration", t + 1)

    spt += pt2[t]
    sqt += qt2[t]

    seat += eat[t]
    sebt += ebt[t]

    p = spt / (t + 1)
    q = sqt / (t + 1)

    x = seat / (t + 1)
    y = sebt / (t + 1)

    eiAq = np.matmul(A, q)
    pAej = np.matmul(p, A)
    pAq = np.dot(p, eiAq)

    err2[t] = max(max(eiAq) - pAq, pAq - min(pAej))

    eiAy = np.matmul(A, y)
    xAej = np.matmul(x, A)
    xAy = np.dot(x, eiAy)

    err3[t] = max(max(eiAy) - xAy, xAy - min(xAej))

t = range(1, T + 1)
fig, ax = plt.subplots(figsize=(5,3), layout='constrained')
ax.set_yscale('log')
ax.plot(t, err1, label='Algorithm 5.2')
ax.plot(t, err2, label='Algorithm 5.3')
ax.plot(t, err3, label='Algorithm 5.4')
ax.set_xlabel('number of iterations')
ax.set_ylabel('distance to Nash equilibrium')
ax.set_title("Comparison of zero-sum algorithms")
ax.legend()
plt.savefig("zsplot")
```