Internship Report

Taralla David, Master in Computing Sciences

UoA mentor: Csaba Szepesvári

ULg mentor: Damien Ernst

University of Liège – Faculty of Applied Sciences

Academic year 2013 – 2014

Contents

Chapter 1

Introduction

This report is the result of my internship in the Computer Science department of the University of Alberta, under the direction of Csaba Szepesvári.

We will first introduce the context in which this research took place, then present the contributions this internship proposes to this context. Finally, the structure of this document will be described.

1.1 Algorithm Discovery

The field of reinforcement learning recently received the contribution by Ernst et al. (2013) who introduced a new way to conceive completely new algorithms. Moreover, it brought an automatic method to find the best algorithm to use in a particular situation.

Practically, Ernst et al. (2013) created a grammar over MCS algorithms, which allowed them to generate a rich space of candidate algorithms to solve particular problems[1]. They then used a multi-armed bandit approach to search this algorithm set for finding the best algorithm to solve a particular problem.

1.1.1 The Grammar

Their grammar contains five *search components*:

simulate Policy-driven simulation (usually, uniformly random policy).

repeat Repeats N times its sub-search component.

lookahead Calls its sub-search component on each possible action in the current state.

step For each remaining step, calls its sub-search component to find the best possible action to take in this step.

select Stores statistics like an MCTS and uses them to bias sub-searches using a particular policy[2].

[1] They considered the class of finite-horizon fully-observable deterministic sequential decision-making problems.

[2] To simplify this research, we used the UCB-tuned policy instead of UCB1. It allowed us not to bother with constants tuning.

Example `step(repeat(100, simulate))` is a *depth-3* algorithm (ie. it contains three search components) that will, for each step i,

1. run 100 random simulations;
2. pick the best simulation S among the 100;
3. record the i^{th} action of S.

When a final state is reached, it will return the recorded sequence of actions.

1.1.2 Finding the Best Algorithm From the Set

To each algorithm of the set, they associate an *arm* of the bandit. Pulling an arm A_k gives a reward r_k, which is the reward obtained by executing the algorithm associated to this arm once. You will notice that the problem of finding the best algorithm is solved by completing the exploration phase only; no exploitation is needed here.

While pulling the arms, they observe the sequence of rewards they get to select in a smart way the next algorithm to try. When the resources allocated to the exploration phase are exhausted, some high-quality algorithms can then be identified.

1.2 Internship Contribution

We want to address the problem of best arm identification in the particular case of Monte Carlo search algorithm discovery for one player games (Ernst et al., 2013).

The main problem is that the generated algorithm space (ie. the arm space) can be quite large as the depth of the generated algorithms increases, so we just can't sample each algorithm the right number of times to be confident enough on the final choice (ie., to be sure the regret is minimized). We need therefore an optimized, scalable method for selecting the best algorithm from bigger spaces.

The main idea is to see the reward of pulling an arm as a function of its *features* rather than directly exploring the algorithm space to find the best arm. This way, we demonstrate we are able to design a confident best arm identification algorithm, without suffering from the size of the space.

1.3 Structure of This Document

We first go through the problem statement in Chapter 2. We will define there our approach and the main ideas which drove this research. We also present there the data set we used for our calculations.

You will then notice than we made an intensive use of *kernels*, allowing us to have all benefits of a feature-driven approach without being penalized by inner products of feature vectors and without having to define these features; the kernel we used and the work we did with it will be presented in Chapter 3.

Thirdly, we present a highly efficient method to find how to extract information from the features of an algorithm called *Generalized Cross-Validation* (GCV). This way, we get rid of the necessity to run an algorithm on super calculators to know which mean reward it has on a given class of problems.

Sadly, we didn't have time to test if the proposed idea was truly better than the one from Ernst et al. (2013). We thus finally conclude this work with no practical results, but with summaries of what we were able to do and learn and what we would have further investigated if we had more time.

Chapter 2

Problem Statement

We describe here the problem statement.

We introduce first some notations and definitions. Second, we present our data set and what its properties are. We finally present our approach and an intuitive version of a first search algorithm we conceived.

2.1 Various Definitions

2.1.1 Algorithms as Strings of Symbols

We can see an algorithm as a sequence of elements picked from an *alphabet* of symbols Σ. Indeed, an algorithm is a chain of predefined symbols. The length of this chain is referenced as the *depth* of the algorithm. Given this, here are some definitions that will be useful in the next sections of this document:

String A *string* $s = s_1...s_{|s|}$ is any finite sequence of symbols from an alphabet Σ, including the empty sequence denoted ϵ, the only string of length 0. We denote by Σ^n the set of all finite strings of length n, and by $Sigma^*$ the set of all strings. In our case, Σ^n denotes the depth-n algorithm space, and $\Sigma^{\leq n}$ the space of algorithms of depth up to n.

Concatenation Given two strings s, t, st is the string obtained by concatenating s and t.

Substring Given two strings s, t, t is a *substring* of s if there exist (possibly empty) strings u and v such that $s = utv$.

If $u = \epsilon$, we say that t is a *prefix* of s.

If $v = \epsilon$, we say that t is a *suffix* of s.

Subsequence We say that a string u is a *subsequence* of a string s if there exist indexes $\mathbf{i} = (i_1, ..., i_{|u|})$, with $0 < i_1 < \cdots < i_{|u|} \leq |s|$, such that $u_j = s_{i_j}$ for $j = 1, ..., |u|$

Example: `cat` is a subsequence of `chat`, because we find each symbol of `cat` in the same order in `chat`.

Notice that if u is a substring of s, it is also a subsequence of s.

In addition to these definitions, we defined the following alphabet:

$$\begin{aligned}\Sigma = \{ \; & \texttt{simulate}, & \texttt{lookahead}, & \texttt{step}, & \texttt{select}, \\ & \texttt{repeat2}, & \texttt{repeat5}, & \texttt{repeat10}, & \texttt{repeat20}, \\ & \texttt{repeat50}, & \texttt{repeat100}, & \texttt{repeat200}, & \texttt{repeat500}, \\ & \texttt{repeat1000}, & \texttt{repeat2000}, & \texttt{repeat5000}, & \texttt{repeat10000} \; \}\end{aligned}$$

Notice that we distinguish two `repeat` if they have different arguments. Moreover, the pruning rules of Ernst et al. (2013) still apply, that is:

- no repeat to begin a string;
- no directly nested `repeats`;[1]
- no directly nested `selects`;
- all strings end with `simulate` and `simulate` cannot be used elsewhere in a string.

2.1.2 General Definitions

Inner product Let $\mathbf{x}, \mathbf{y} \in \mathbb{R}^{d \times 1}$ $(d \in \mathbb{N}^0)$. The *inner product* $\langle \mathbf{x}, \mathbf{y} \rangle$ is defined as the result of $\mathbf{x}^T \mathbf{y}$.

Feature vector Let $\phi : A \mapsto \mathbb{R}^{d \times 1}$ $(d \in \mathbb{N}^0)$ be a feature map, where A is a finite set of inputs. We denote by $\phi(a)$ (or ϕ_a for the sake of conciseness) the feature vector of input $a \in A$.

Kernel Let $A = \{a_1, ..., a_{|A|}\}$ be an arm set. A *kernel* is a function such that

$$K : (A \times A) \mapsto \mathbb{R} \mid K(a, a') = \langle \phi_a, \phi_{a'} \rangle$$

2.2 Description of our Data Set

Our data set is composed of 1088 samples. A sample is a pair (a, r) where

- a is an algorithm composed of symbols from Σ and respecting the aforementioned rules;
- r is the *mean* reward a obtained when run on 10 000 different problems of the same class.

The problem class was the game known as Sudoku, with grids of size 16 × 16. A particular problem from this class is a random 16 × 16 Sudoku grid with approximately 33% of the grid prefilled. The search algorithm has to fill the most cells it can with a budget of 1000 calls to the reward function. Once this budget is exhausted, the algorithm has to return the best reward it earned while looking for the best solution.

The reward is defined as the number of non-empty cells (the prefilled ones are included) when entering a final state (ie. when no more moves are possible). Therefore, a reward is always an integer from the interval $[0, 256]$.

[1] Notice that Ernst et al. (2013) call this rule as "canonization of repeats". Here we already generated the possible canonizations and assume we can't combine these `repeats` again.

To ensure fairness between all algorithms, each of them was run on the *same* set of Sudoku grids.

This data set was computed thanks to the free access we had to *Compute Canada*[2], and more particularly *Westgrid*[3].

2.3 Basic Idea

As we said earlier, the main idea is to see the reward of pulling an arm as a function of its *features* rather than sampling the algorithms many times to find the best arm. To this end, we introduce a new way of thinking to the problem.

2.3.1 The Reward of an Arm as a Function of its Features

Using a feature-driven approach, we basically want to approximate the parameter vector θ that is such that

$$\bar{r}_a = \langle \phi_a, \theta \rangle \tag{2.1}$$

where $\bar{r}_a$ is the mean reward of arm a.

Once we have an approximation $\hat{\theta}$ of the parameter vector, we can find the arm which maximizes the estimated mean reward by using

$$a^* = \underset{a \in A}{\operatorname{argmax}} \langle \phi_a, \hat{\theta} \rangle \tag{2.2}$$

where A is the set of available arms.

Notice that if $\hat{\theta}$ is a good approximation to θ, we expect the found arm to be close to optimal.

2.3.2 Approximating θ

As we mentioned it in Section 2.3.1, we have to approximate θ, the parameter vector. The core method we will use for that is the *Regularized Least-Squares* (RLS) method, that is

$$\hat{\theta} = \underset{\theta \in \mathbb{R}^d}{\operatorname{argmin}} \frac{1}{n} \sum_{t=1}^{n} \left(\langle \phi_{a_t}, \theta \rangle - r_{a_t} \right)^2 + \lambda \|\theta\|_2^2 \tag{2.3}$$

where n is the size of the data list, λ is the regularization parameter and r_{a_t} the reward for pulling arm a_t:[4]

$$r_{a_t} = \langle \phi_{a_t}, \theta \rangle + \eta_{a_t}. \tag{2.4}$$

Nevertheless, we want to maximize the confidence we have on the selected arm, so this method alone won't be sufficient enough. This brings us to the proposed algorithm.

2.3.3 Proposed Algorithm

We conceived a prototype of algorithm to get results faster than the multi-armed bandit approach. This should be considered as a first cut at the problem.

Given that

[2] http://computecanada.ca/

[3] http://www.westgrid.ca/

[4] η_{a_t} is the zero-mean noise on the reward for pulling arm a_t.

- $n_1, ..., n_k$ are the lengths of k *rounds*,
- K is a suitable kernel,
- λ is the regularization parameter,

we have the algorithm presented as Algorithm 2.1.

The idea is to refine our algorithm space k times (**incremental** algorithm). The refinement is done in each round, by

1. creating a *sampling plan* that will tell which sequence of arms should be played during this round, where these arms were chosen because of the way they represent "good" samples (regarding the features) of the action space,
2. collecting the rewards of these pulls,
3. using RLS to find an approximation of θ,
4. computing the upper and lower confidence bounds on each $\overline{r}_a$,[5]
5. discarding all arms for which the upper bound is smaller than the lower bound of the best action.

By repeating this refinement process multiple times, we are able to converge towards the best action, and this in being confident in that choice. Indeed, as the rounds go on, the algorithm space will be reduced (step 5) in such a way that each remaining algorithm is guaranteed to have a reward that can be at least greater or equal to the worse reward the best arm so far can earn[6]. That way at each round we become more and more confident in our identification of the best arm.

Given Algorithm 2.1 and what has been said so far, the research we conduct has to find a solution to the following problems:

1. Defining the sample plan function, ExpDesign.
2. Find a way to efficiently compute $\hat{\theta}$ using RLS and the kernel.
3. Defining the bound computation functions, RLS_LowConf & RLS_UpConf.

[5] As $\hat{\theta}$ is only an approximation, it puts "noise" on each $\overline{r}_a$ it allows to compute. Therefore there exist lower and upper confidence bounds on each $\overline{r}_a$, and we want to minimize their interval.

[6] By *can*, we mean that as the reward is included into $[r_a - \text{confidenceBound}, r_a + \text{confidenceBound}]$, its true value can be any value in that interval.

Algorithm 2.1: Best arm identification for huge action spaces

Input: $n_1, ..., n_k, K, \lambda, A$
Output: a^*, the best identified arm
/* Initialization */
$A_{\text{good}} \leftarrow A$;
$D \leftarrow [\,]$;
/* Procedure */
for $i = 1$ **to** k **do** /* round i */
  $m \leftarrow n_i$; /* length of round i */
  $(a_1, ..., a_m) \leftarrow \text{ExpDesign}(A, A_{\text{good}}, K, m, ...)$; /* sample plan */
  for $j = 1$ **to** m **do**
    Execute(a_j), get r_{a_j};
    D.append((a_j, r_{a_j}));
  end
  $\hat{\theta} \leftarrow \text{RLS}(D, \lambda, K)$;
  $a^* \leftarrow \text{argmax}_{a \in A_{\text{good}}} \langle \phi_a, \hat{\theta} \rangle$;
  lower_conf $\leftarrow$ RLS_LowConf(D, λ, K, a^*);
  $A_{\text{new}} \leftarrow [\,]$;
  for $a \in A_{\text{good}}$ **do**
    if RLS_UpConf(D, λ, K, a) $\geq$ *lower_conf* **then**
      A_{new}.append(a);
    end
  end
  $A_{\text{good}} \leftarrow A_{\text{new}}$;
end
return a^*

Chapter 3

Kernels Study

What features should we use to describe an algorithm? How can we characterize how distant two algorithms are from each other? This chapter answers these questions and also present which particular kernel was tested.

This chapter uses also the following property: as $\hat{\theta}$ is in our case an estimator obtained by minimizing penalized empirical error, where the penalty is a function of the two-norm of the weight, there exist some $\hat{\alpha_t} \in \mathbb{R}$, $t = 1, ..., n$ (with n being the size of the data list) such that

$$\langle \phi_a, \hat{\theta} \rangle = \left\langle \phi_a, \sum_{t=1}^{n} \hat{\alpha}_t \phi_{a_t} \right\rangle = \sum_{t=1}^{n} \hat{\alpha}_t \underbrace{\langle \phi_a, \phi_{a_t} \rangle}_{K(a, a_t)} . \tag{3.1}$$

This property allow us to *kernelize* the regularized least-squares method (cf. Section 3.2).

Equivalently, instead of (2.4), we may assume that for $t = 1, 2, \ldots, n$, the data is generated from

$$r_{a_t} = \sum_{a \in A} \alpha_a K(a_t, a) \, + \eta_{a_t} \ , \tag{3.2}$$

where $\alpha = (\alpha_a)_{a \in A}$ is an unknown $|A|$-dimensional vector and η_{a_t} is as in (2.4).

Note that (2.4) and (3.2) are equivalent assumptions: Given K, we can always find an appropriate feature map ϕ so that if (3.2) holds then (2.4) holds and vice versa. Furthermore, the unknown vectors θ and α are related through the simultaneous linear equations

$$\sum_{a' \in A} \alpha_{a'} K(a, a') = \langle \phi_a, \theta \rangle \, , \quad a \in A \, . \tag{3.3}$$

Without loss of generality, identifying A with the first $N = |A|$ integers, introducing the matrix $\mathbb{K} \in \mathbb{R}^{N \times N}$ whose (i, j)th element is $K(i, j)$ and introducing the $\Phi \in \mathbb{R}^{N \times d}$ matrix whose ith row is $\phi_i^\top$, we can write (3.3) in the short form

$$\mathbb{K}\alpha = \Phi\theta.$$

Without loss of generality we may assume that Φ has rank d (otherwise the dimensionality of the feature space could be reduced). Premultiplying with $\Phi^\top$ the above equation and noticing

that $\Phi^\top\Phi$ is non-singular by the assumption we just made, we get

$$\theta = (\Phi^\top\Phi)^{-1}\Phi^\top\mathbb{K}\alpha.$$

Noticing that

$$\mathbb{K} = \Phi\Phi^\top,$$

we get

$$\theta = (\Phi^\top\Phi)^{-1}\Phi^\top\Phi\Phi^\top\alpha = \Phi^\top\alpha\,.$$

Hence,

$$\|\theta\|_2^2 = \left\|\Phi^\top\alpha\right\|_2^2 = \alpha^\top\Phi\Phi^\top\alpha = \alpha^\top\mathbb{K}\alpha\,.$$

In what follows, for Q positive definite, we will use $\|x\|_Q^2$ to denote $x^\top Qx$. Hence, the above equation can be written as

$$\|\theta\|_2^2 = \|\alpha\|_{\mathbb{K}}^2\,. \tag{3.4}$$

3.1 Why Using String Kernels

In order to characterize how distant two algorithms are from each other, we decided to rely on *string kernels* (Shawe-Taylor and Cristianini, 2004).

We decided to base our features on *pattern matching analysis*. Indeed, it has already been shown by Ernst et al. (2013) that some patterns of algorithms (like for instance the UCT pattern: `step(repeat(·, select(sim)))`) are often repeated in the top ten of the selected algorithms, according to the particular problem they try to solve. It is therefore a natural first choice.

A particular kernel of this type will be discussed in Section 3.3.

3.2 Kernelized Regularized Least-Squares

3.2.1 Determining the Regressor Thanks to Kernels

By injecting results of (3.1) into RLS equation (2.3), the problem is now reduced to solve for the α_i coefficients. This can be done by equalizing to zero the derivative with respect to $\hat{\alpha}$:

$$\begin{aligned}
0 &= \frac{\partial}{\partial\hat{\alpha}}\left[\frac{1}{n}\sum_{t=1}^{n}\left(\left\langle\phi_{a_t},\sum_{i=1}^{n}\hat{\alpha}_i\phi_{a_i}\right\rangle - r_{a_t}\right)^2 + \lambda\left\langle\sum_{i=1}^{n}\hat{\alpha}_i\phi_{a_i},\sum_{i=1}^{n}\hat{\alpha}_i\phi_{a_i}\right\rangle\right] \\
&= \frac{\partial}{\partial\hat{\alpha}}\left[\frac{1}{n}\sum_{t=1}^{n}\left(\sum_{i=1}^{n}\hat{\alpha}_i\langle\phi_{a_t},\phi_{a_i}\rangle - r_{a_t}\right)^2 + \lambda\sum_{i=1}^{n}\sum_{j=1}^{n}\hat{\alpha}_i\hat{\alpha}_j\left\langle\phi_{a_i},\phi_{a_j}\right\rangle\right] \\
&= \frac{\partial}{\partial\hat{\alpha}}\left[\frac{1}{n}\sum_{t=1}^{n}\left(\sum_{i=1}^{n}\hat{\alpha}_iK(a_t,a_i) - r_{a_t}\right)^2 + \lambda\sum_{i=1}^{n}\sum_{j=1}^{n}\hat{\alpha}_i\hat{\alpha}_jK(a_i,a_j)\right].
\end{aligned} \tag{3.5}$$

Eq. (3.5) allows us to determine the $\hat{\alpha}_i$ minimizing the error (as we would find the θ minimizing the error if using "standard" RLS).

Nevertheless, this mathematical form doesn't suit well for numerical solving. Therefore, we used the following reasoning to deal with this problem.

Given

$$K \in \mathbb{R}^{n \times n} \; : \; K_{ij} = K(a_i, a_j) \tag{3.6}$$

we can rewrite (3.5):

$$0 = \frac{\partial}{\partial \hat{\alpha}} \left[\frac{1}{n} \left\| K^T \hat{\alpha} - R \right\|_2^2 + \lambda \hat{\alpha}^T K \hat{\alpha} \right] \tag{3.7}$$

where

$$\hat{\alpha} = \begin{pmatrix} \hat{\alpha}_1 \\ \vdots \\ \hat{\alpha}_n \end{pmatrix}, \quad R = \begin{pmatrix} r_{a_1} \\ \vdots \\ r_{a_n} \end{pmatrix}.$$

Therefore, applying the $\frac{\partial}{\partial \hat{\alpha}}$ operator,

$$\begin{array}{rcl} \mathbf{0} & = & \frac{2}{n} K^T \left(K^T \hat{\alpha} - R \right) + 2\lambda K \hat{\alpha} \\ & = & \frac{1}{n} K \left(K\hat{\alpha} - R \right) + \lambda K \hat{\alpha} \\ & \overset{(\lambda' := \lambda n)}{=} & K \left(K\hat{\alpha} - R \right) + \lambda' K \hat{\alpha} \end{array} \tag{3.8}$$

because by construction of the matrix K, as $K(a_i, a_j) = K(a_j, a_i)$, we have $K^T = K$. The matrix K is called the *Grammian* underlying the points $(a_1, \ldots, a_n)$ (note that $K \neq \mathbb{K}$). Let's also notice that K is positive semi-definite and $\lambda' I_n$ is positive definite as $\lambda' > 0$.

Therefore, the sum $K + \lambda' I_n$ is guaranteed to be non-singular and we can finally efficiently compute $\hat{\alpha}$ thanks to the following system of linear equations:

$$\begin{array}{rcl} \mathbf{0} & = & (K\hat{\alpha} - R) + \lambda' \hat{\alpha} \\ & = & K\hat{\alpha} - R + \lambda' \hat{\alpha} \\ & = & (K + \lambda' I_n)\, \hat{\alpha} - R \\ \\ \Leftrightarrow \quad \hat{\alpha} & = & (K + \lambda' I_n)^{-1} R \end{array} \tag{3.9}$$

Prediction Notation

Note that given the $\hat{\alpha}_i$ coefficients, a prediction at a new input a can be written as

$$f_{D,\lambda}(a) = \sum_{i=1}^{n} \hat{\alpha}_i K(a_i, a), \tag{3.10}$$

where $D = ((a_1, r_1), \ldots, (a_n, r_n))$ is used above to emphasize that the prediction depends on the data D (the dependence comes through both $\hat{\alpha}_i$ and a_i, $i = 1, \ldots, n$). We have also denoted the dependence on λ.

3.2.2 Normalizing the Data

Note that the larger the value of λ is, the closer is $K + \lambda I$ to λI and thus the closer the inverse of $K + \lambda I$ is to $\lambda^{-1} I$. Thus, for λ large, we expect $(K + \lambda I)^{-1} \approx \lambda^{-1} I$ and hence $\hat{\alpha} = (K + \lambda I)^{-1} R \approx \lambda^{-1} R$. Thus, $|\hat{\alpha}| \to 0$ as $\lambda \to \infty$ and as a consequence we also have

$f_{D,\lambda}(a) \to 0$ as $\lambda \to \infty$. However, this is not ideal: While it is expected that "regularization" compresses the range of predictions (after all, this is how you "reject" the noise), the predictions should converge to a meaningful value, say the mean of the rewards, $\overline{r} = \frac{1}{n}\sum_{i=1}^{n} r_{a_i}$, and not zero.

This can simply be achieved by computing $\hat{\alpha}' = (K + \lambda I)R'$, where $R'_i = R_i - \overline{r}$ and then change (3.10) to

$$f_{D,\lambda}(a) = \overline{r} + \sum_{i=1}^{n} \hat{\alpha}'_i K(a_i, a). \tag{3.11}$$

3.2.3 Regularizing the Data

It remains to specify how to select the value of λ. One idea is the following: Fix the data $D = ((a_1, r_1), \ldots, (a_n, r_n))$ and let D_{-i} be obtained from D by removing the i^{th} element from it:

$$D_{-i} = ((a_1, r_1), \ldots, (a_{i-1}, r_{i-1}), (a_{i+1}, r_{i+1}), \ldots, (a_n, r_n)),$$

$i \in \{1, \ldots, n\}$.

Let

$$e(\lambda) = \frac{1}{n}\sum_{i=1}^{n} (f_{D_{-i},\lambda}(a_i) - r_i)^2.$$

The value $e(\lambda)$ measures how well the predictors trained with a particular value of λ are able to generalize to the left out data-points. Putting computational issues aside, the idea is to identify $\lambda^* = \operatorname{argmin}_{\lambda>0} e(\lambda)$ and then use this value to make predictions (i.e., "future" predictions are made using f_{D,λ^*}).

Note that the calculation of λ^* is a one-dimensional optimization problem.

In practice, a simple bracketing search is used because there is no need to calculate λ^* up to a high accuracy.

It remains then to consider the computation of $e(\lambda)$ for a fixed value of λ.

The trivial procedure to calculate $e(\lambda)$ is to calculate the coefficients $\hat{\alpha}_{-i}$ based on the value of λ and D_{-i} using (3.9) and then use (3.10) to produce $f_{D_{-i},\lambda}(a_i)$. Assuming that matrix inversion takes $O(n^3)$ time, the total cost of calculating $e(\lambda)$ for a given value of λ is $O(n^4)$. This is prohibitively large for even moderate values of n.

Fortunately, it is possible to implement this calculation in $O(n^3)$ time with almost the same cost as just computing the coefficients $\hat{\alpha}$ for D. Even better, we will show that if M evaluation of $e(\lambda)$ is required by some procedure to find an approximation to λ^*, then $O(n^3 + Mn^2)$ computation will suffice (as opposed to $O(Mn^3)$).

This optimized approach is presented in Section 5.1 of Chapter 5.

3.2.4 Confidence Bands for Kernelized Least-Squares

It remains to determine the *confidence bands* around the predictions. Abbasi-Yadkori et al. (2011) proved the following useful theorem[1].

[1]Notice that we present its extension to the "kernel-case", proposed in Abbasi-Yadkori (2012).

Theorem 1. *Assume that the data $D = ((a_1, r_1), \dots, (a_n, r_n))$ is generated from the kernelized linear model (3.2) and the noise in this model is such that for some $L \in \mathbb{R}$ and $R > 0$, for all $1 \le t \le n$, $r_t \in [L, L+R]$ holds w.p.1. Assume further that the unknown $\alpha \in \mathbb{R}^{|A|}$ is such that $\|\alpha\|_{\mathbb{K}} \le S$, where $\mathbb{K}$ is the full kernel matrix that was introduced in Chapter 3. Let $f_{D,\lambda}$ be the predictor obtained using the regularized least-squares procedure with kernel K on data D when the regularization coefficient is fixed at $\lambda > 0$. For $a \in A$, define $k_a \in \mathbb{R}^n$ by $k_a = (K(a, a_1), \dots, K(a, a_n))^\top$ and let K be the $n \times n$ kernel matrix underlying D: $K_{ij} = K(a_i, a_j)$. Then, for any $0 \le \delta \le 1$, with probability at least $1 - \delta$,* simultaneously *for all $a \in A$ it holds that*

$$|f_{D,\lambda}(a) - \overline{r}_a| \le c_a \left(R \sqrt{2 \log \frac{\det\left(I + \frac{1}{\lambda} K\right)^{\frac{1}{2}}}{\delta}} + \lambda^{\frac{1}{2}} S \right), \tag{3.12}$$

where

$$c_a = \sqrt{\frac{K(a,a) - k_a^\top (K + \lambda I)^{-1} k_a}{\lambda}}\,.$$

Note that

$$\begin{aligned} B \doteq 2 \log \frac{\det\left(I + K/\lambda\right)^{\frac{1}{2}}}{\delta} &= 2 \log\left(\det\left(I + K/\lambda\right)^{\frac{1}{2}}\right) + 2 \log\left(\frac{1}{\delta}\right) \\ &= \log(\det I + K/\lambda) + 2 \log\left(\frac{1}{\delta}\right)\,. \end{aligned}$$

Based on the matrix determinant lemma, it may be possible to calculate the determinant recursively.[2] However, since the other parts of the algorithm need the eigendecomposition of K anyways, there is no need for this. Indeed, if $K = Q \Lambda Q^\top$, with $\Lambda = \mathrm{diag}(\lambda_1, \dots, \lambda_n)$ then $\det\left(I + K/\lambda\right) = \prod_{i=1}^n (1 + \lambda_i/\lambda)$ and thus $\log\det\left(I + K/\lambda\right) = \sum_{i=1}^n \log\left(1 + \frac{\lambda_i}{\lambda}\right)$. Plugging this into B, we get

$$B = 2 \log\left(\frac{1}{\delta}\right) + \sum_{i=1}^n \log\left(1 + \frac{\lambda_i}{\lambda}\right)\,.$$

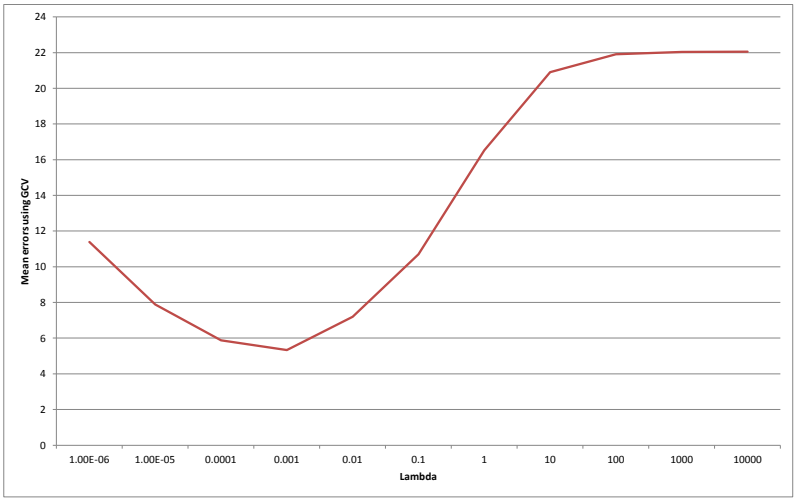
Note that calculating c_a requires n kernel evaluations to calculate the vector k_a. Again, if the eigendecomposition $K = Q \Lambda Q^\top$ is already available, $k_a^\top (\lambda I + K)^{-1} k_a = k_a^\top (Q(\lambda I + \Lambda) Q^\top)^{-1} k_a = k_a^\top Q (\lambda I + \Lambda)^{-1} Q^\top k_a$ can be calculated efficiently. First, precompute $Q^\top k_a$ (at a cost of n^2 floating point operations) and keep a copy Q_{k_a} of it, then scale the ith component of this vector by $(\lambda + \lambda_i)^{-1}$ and then take the inner product of the scaled vector and $Q_{k_a}^\top$ at the total cost of $2n + n^2 = O(n^2)$ floating point operations.

A simple way of speeding up the algorithm when the cardinality of A is huge is to randomly choose a subset $\tilde{A}$ of some smaller cardinality and then run the algorithm as if $\tilde{A}$ was the full space. In this case, the experimental design procedure will only consider the random subset. The cardinality of $\tilde{A}$ can be chosen so that the computations are feasible.

Note that the above theorem is valid for a fixed choice of the regularization parameter. In our procedure, λ is chosen based on the data. To account for this, the formula would need to be changed. However, this is left for future work.

[2] We tried this approach, and it seems that the memory requirement is $O(n^3)$ and the update is $O(n^2)$, so this may not worth the effort.

Figure 3.1: Mean error when predicting the mean reward of an algorithm



3.3 Tested String Kernel

This section presents the string kernel we studied and experimented with. We briefly summarize what it does before showing that it allows us to find an $\hat{\alpha}$ vector which makes the computed reward fit as much as possible the one we expect by simply running the algorithm.

3.3.1 All-Subsequences

This kernel consider a feature mapping defined by all contiguous or non-contiguous subsequences of a string. The main weakness of this kernel is that it doesn't make any difference between contiguous and *non*-contiguous subsequences when computing the inner product of two strings.

Experiment

We wanted to illustrate whether this kernel could help us to predict correctly the mean reward given by any algorithm or not. To this end, we measured for several values of λ the mean squared error committed on the left-out values using General Cross-Validation[3] on our data set.

We choose to use GCV to have results reflecting how well a choice of λ performed on new examples instead of training examples.

Figure 3.1 shows that there is a minimum for which the mean error is approximately 5.33. This is quite good and encouraging regarding the simplicity and naiveness of the kernel we used. This number means that the mean error committed when predicting the mean reward of an algorithm is approximately only 5.33!

[3]This is the optimized method we used to determine λ. See Section 3.2.3 for introduction, and Section 5.1 of Chapter 5 for more details on the method implementation.

Verification We ensured these results were correct by also implementing the naive method from Section 3.2.3 and comparing the results. Then both of these versions were also implemented in MATLAB to prove their correctness. The four versions agreed so we can assume these results are meaningful.

Chapter 4

G-optimal Design

Let $\phi : X \to \mathbb{R}^d$ be a feature map, where X is a finite set of inputs. Let N denote the cardinality of X. To simplify notation we identify the elements of X with the integers and thus write $X = \{1, \dots, N\}$. This can always be done without loss of generality as we will only need the identity of the points of x and their corresponding feature values. Further, for brevity we will denote by ϕ_i the value of $\phi(i)$.

In optimal experimental design the problem is to find out which points to sample so that if we estimate some model parameters the accuracy is maximized in some sense. The design methods differ in terms of what they assume about the model and the data and how the accuracy is measured. Here, we will assume that upon sampling a point i, the response Y collected satisfies

$$Y = \phi_i^\top \theta + \varepsilon,$$

where ε is a zero mean random variable and $\theta \in \mathbb{R}^d$ is an unknown parameter vector. Further, it is assumed that the variance of ε does not depend on i[1] and that if we sample at a number of sites, the corresponding noise random variables will be independent of each other.

In G-optimal designs the accuracy is measured by $\max_{1 \le i \le N} \mathbb{E}\left[(\phi_i^\top \hat{\theta} - Y)^2\right]$, where $\hat{\theta}$ is the least-squares estimate based on the data.[2] That is, the objective function is the maximum expected squared error. As it turns out, an equivalent criterion is to maximize the log determinant of the scaled "Fisher information matrix",

$$M(\gamma) = \sum_{i=1}^{N} \gamma_i \phi_i \phi_i^\top .$$

Defining the $N \times d$ matrix

$$\Phi = \begin{pmatrix} \text{———} \phi_1^\top \text{———} \\ \vdots \\ \text{———} \phi_N^\top \text{———} \end{pmatrix},$$

[1] If the variances can depend on the inputs i, we would need to measure them and adjust the method described below to use the estimated variances. This is left for future work for now. In statistics, the assumption that the variance is constant, is called the homoscedasticity assumption, while if the variance is allowed to depend on the input, we talk about a heteroscedastic situation.

[2] That we used penalized least-squares is thus not taken into account. Future work may consider adjusting the design so that this gap is removed, though I do not expect that the adjustment will result in significant differences.

and the diagonal matrix $\Gamma = \mathrm{diag}(\ldots, \gamma_i, \ldots)$, we can write

$$M(\gamma) = \Phi^\top \Gamma \Phi$$

and the optimization problem is defined to maximize $J : \Delta_N \to \mathbb{R}$, where

$$J(\gamma) = \log \det M(\gamma)\,. \tag{4.1}$$

4.1 Kernelizing Optimal Experimental Design

Consider the setting of the previous section and define the kernel function $K : X \times X \to \mathbb{R}$ using $K(i,j) = \langle \phi_i, \phi_j \rangle$. We can also view $K(i,j)$ as an $N \times N$ matrix K such that $K_{i,j} = K(i,j)$. Using the notation of the previous section, we can then write

$$K = \Phi \Phi^T\,.$$

Note that if one is given the $N \times N$ kernel matrix K only, Φ can be recovered up to a multiplication with a matrix of rank equal to the rank of Φ. For this take the eigendecomposition of K:

$$K = Q \Lambda Q^\top$$

and define $\Phi' = Q\Lambda^{1/2}$, where $\Lambda^{1/2}$ is obtained by taking the square root of the entries in Λ (which is a diagonal matrix). Since the rank of Φ' is the same as that of Φ (and K), clearly, the linear system $\Phi = \Phi' X$ has a solution $X \in \mathbb{R}^{d \times d}$ with rank matching that of Φ (and K). What is more, since if ϕ_i' is the transpose of the ith row of Φ', from $K = \Phi'(\Phi')^\top$ we see that $K_{i,j} = \langle \phi_i', \phi_j' \rangle$.

In summary, given the $N \times N$ kernel matrix K, if Φ' is defined as above, we can treat the ith row of Φ' as the feature vector underlying input i. In fact, without loss of generality we can assume that $\Phi = \Phi'$. Then we can solve the optimization problem of the previous section to get the optimal allocation vector γ^*.

In fact, the objective function $J(\gamma)$ is invariant to non-singular transformation of the features (as it should be). Indeed, if $\Phi' = \Phi A$, the (scaled) Fisher information matrix underlying Φ' is $M'(\gamma) = A^\top \Phi^\top \Gamma \Phi A$. Now, remembering the determinant multiplication theorem which states that for square matrices U, V, $\det(UV) = \det(U)\det(V)$, we see that $\det M'(\gamma) = \det(A^\top \Phi^\top \Gamma \Phi A) = \det(A^\top)\det(A)\det(M(\gamma))$. Hence, $\det M'(\gamma) = \mathrm{const} \times \det M(\gamma)$, and hence the optimal solutions defined by Φ' are the same as that of defined using Φ.

4.2 Algorithms for Finding Optimal Designs

We follow the paper of Yu (2011). Remember that the goal is to find an approximate minimizer of $J(\gamma)$ defined using (4.1) over the simplex Δ_N. Define

$$g(i,j,\gamma) = \phi_i^\top M^{-1}(\gamma) \phi_j, \quad g(i,\gamma) = g(i,i,\gamma). \tag{4.2}$$

As it turns out $g(i,\gamma) = \frac{\partial J(\gamma)}{\partial \gamma_i}$ and also $g(i,\gamma) - d = \frac{\partial J((1-\delta)\gamma + \delta e_i)}{\partial \delta}|_{\delta \to 0+}$ (i.e., $g(i,\gamma)$ is the partial derivative and also the directional derivative of J, the objective function.)

The method that we consider is called the "Cocktail" method by Yu (2011). This is an iterative method which keeps a current estimate γ of the optimizer $\mathrm{argmin}_{\gamma \in \Delta_N} J(\gamma)$. The name of the method comes from that it uses three different updates. The stopping criterion

proposed by Yu (2011) is to stop when $\frac{1}{d}\max_i g(i,\gamma) \leq 1+\varepsilon$ is satisfied for the current iterate for some small ε. The suggested initialization is to choose $2d$ points $\{X_1,\dots,X_{2d}\}$ at random from X and set $\gamma_i = (2d)^{-1}\sum_{k=1}^{2d}\mathbb{I}\{X_k = i\}$, $i=1,\dots,N$. When $2d \geq N$, we can start from the uniform vector: $\gamma_i = 1/N$ (this is the case if we use a kernel which gives a full-rank kernel matrix; and this will normally be the case for us).

Before introducing the Cocktail method, we introduce the three update rules that it uses. The algorithms input sometimes also includes the gradient-vector $g \doteq (g(1,\gamma),\dots,g(N,\gamma))^\top$ in addition to the last estimate γ. We will discuss later how to calculate g incrementally, exploiting that the updates touch only a few of the components of γ.

4.2.1 Multiplicative Weights Update (MWU)

Yu (2011) calls this the multiplicative algorithm, but this update is known as the multiplicative weights (MW) update in the machine learning literature. The update is shown in Algorithm Listing 4.1.

Algorithm 4.1: Multiplicate Weights Update ($\gamma' = MWU(\gamma, g)$)

Input: γ (allocation vector), g (gradient vector)
Output: updated allocation vector
`/* Procedure */`
for $i = 1$ **to** N **do**
  if $\gamma_i > 0$ **then**
    $\gamma_i \leftarrow \frac{1}{d}\gamma_i g_i$
  end
end
return γ

Note that after the update, $\sum_i \gamma_i = 1$ is still maintained (apart from numerical errors) as one can show that (before the update) $d = \sum_{i=1}^N \gamma_i g_i$.

4.2.2 Vertex Direction Update (VDU)

This corresponds to what is known as the "conditional gradient method" in the optimization literature (experimental optimal design people have reinvented it, hence the name difference). The update is shown in Algorithm Listing 4.2.

Algorithm 4.2: Vertex Direction Update ($\gamma' = VDU(\gamma, g)$)

Input: γ (allocation vector), g (gradient vector)
Output: updated allocation vector
`/* Procedure */`
$i^* \leftarrow \operatorname{argmax}_{1\leq i\leq N} g_i$;
$g^* \leftarrow g_{i^*}$;
$\delta \leftarrow \frac{g^*/d - 1}{g^* - 1}$;
$\gamma \leftarrow (1-\delta)\,\gamma$;
$\gamma_{i*} \leftarrow \gamma_{i*} + \delta$;
return γ

4.2.3 Nearest-Neighbor Exchange (NNE)

The Nearest-Neighbor Exchange (NNE) is based on the Vertex-Exchange Update (VEU), which we define first. VEU takes two indices, $1 \leq j, k \leq N$ in addition to the usual parameters and transfers weight from component j to component k to greedily optimize $J(\cdot)$. It also takes the gradient vector components at j and k, but it also takes $g_{jk} = g(j, k, \gamma)$. The update is shown in Algorithm Listing 4.3.

Algorithm 4.3: Vertex-Exchange Update ($\gamma' = VEU(j, k, \gamma, g_j, g_k, g_{jk})$)

Input: $1 \leq j, k \leq N$, γ (allocation vector), g_j, g_k, g_{jk}
Output: updated allocation vector
`/* Procedure */`

$\delta^* \leftarrow \frac{g_k - g_j}{2\{g_j g_k - g_{jk}\}}$;
$\delta \leftarrow \min(\gamma_j, \max(-\gamma_k, \delta^*))$;
$\gamma_j \leftarrow \gamma_j - \delta$;
$\gamma_k \leftarrow \gamma_k + \delta$;
return γ

For each index i, NEE computes the index whose feature vector is closest to the feature vector of index i. It then calls the Vertex-Exchange Update to transfer weights between these indices. The update is shown in Algorithm Listing 4.4.

Algorithm 4.4: Nearest-neighbor Exchange ($\gamma' = NNE(\gamma)$)

Input: γ (allocation vector)
Output: updated allocation vector
`/* Procedure */`

$x \leftarrow (0, \ldots, 0)^\top$;
for $i = 1$ **to** N **do**
  if $\gamma_i > 0$ **then**
    $x_i \leftarrow \underset{k:\gamma_k > 0, k \neq i}{\operatorname{argmin}} \|\phi_k - \phi_i\|$;
  end
end
for $i = 1$ **to** N **do**
  if $\gamma_i > 0$ **then**
    $\gamma \leftarrow VEU(i, x_i, \gamma, g(i, \gamma), g(x_i, \gamma), g(i, x_i, \gamma))$; `/* 2 values are changed */`
  end
end
return γ

4.2.4 Cocktail method

The update for the Cocktail method. This simply amounts to calling VDU, then NNE and then MWU. Algorithm listing 4.5 shows the update for completeness.

Algorithm 4.5: Cocktail Method Update ($\gamma' = CU(\gamma)$)

Input: γ (allocation vector), g
Output: updated allocation vector
`/* Procedure */`
$\gamma \leftarrow VDU(\gamma, g(\cdot, \gamma))$;
$\gamma \leftarrow NNE(\gamma)$;
$\gamma \leftarrow MWU(\gamma, g(\cdot, \gamma))$;
return γ

Implementation Notes

Note that to calculate $g(i, \gamma)$, or $g(i, j, \gamma)$ we need the inverse matrix $M^{-1}(\gamma)$. If only one, or a few components of γ are updated, then the inverse can be updated, too, at the cost of $O(N^2)$ using the Sherman-Morrison formula:

$$(A + uv^\top)^{-1} = A^{-1} - \frac{A^{-1}uv^\top A^{-1}}{1 + v^\top A^{-1}u} ,$$

which holds whenever $1 + v^\top A^{-1}u \neq 0$ (here, A is an arbitrary, nonsingular $d \times d$ matrix and u, v are d-dimensional vectors). Note that to calculate the matrix $A^{-1}uv^\top A^{-1}$, one should first calculate $A^{-1}u$, then $A^{-T}v$ and then take their outer product.

To see why this formula applies, it suffices to consider the case when only one component of γ is changed (if k components are changed, we can update the Sherman-Morrison formula k times, each time changing only one component of γ). Assume that γ' differs from γ in one component, let this be i. Thus, $\gamma'_j = \gamma_j$ except when $j = i$. Then, $M(\gamma') = M(\gamma) + (\gamma'_i - \gamma_i)\phi_i\phi_i^\top$. Hence, defining $A = M(\gamma)$, $u = (\gamma'_i - \gamma_i)\phi$ and $v = \phi$, we get an $O(N^2)$ update. What happens when $1 + (\gamma'_i - \gamma_i)\phi_i^\top M^{-1}(\gamma)\phi_i = 0$? In this case, we could use the Woodbury matrix identity updating multiple components simultaneously. Note that each algorithm is such that after the update, $M(\gamma)$ is full rank if it was full rank before the update. Hence, using the Woodbury matrix identity (which reduces the inversion of the $N \times N$ matrix to that of a $k \times k$ matrix if k components are changed) is guaranteed to succeed. In the worst case, one can resort to doing the full calculation of the inverse.

4.3 Low Rank Approximation to the Kernel Matrix

When using kernels with finite space, we can expect the kernel matrix K to be full rank (otherwise there would exist functions which cannot be represented using the space generated by the kernel). If the kernel matrix K is full rank, all the designs will include all the points, no matter how large N will be. This does not mean that some points will not receive more weights in the optimal allocation than others, but the whole computation may become too expensive. (The above updates have the nice property that they can be sped up considerably when the allocation vectors are sparse. Further, MWU keeps the current sparsity pattern, VDU introduces at most one new nonzero component and NNE changes only two components and is expected to zero out components time to time (the purpose of including NNE is exactly this).

Now, when K is full rank, a reasonable approach is to cut its eigenvalues when defining Φ. This can be done as follows: Let $K = Q\Lambda Q^\top$ be the eigendecomposition of K with $\Lambda = \mathrm{diag}(\ldots, \lambda_i, \ldots)$. Define $\tilde{\Lambda} = \mathrm{diag}(\ldots, \tilde{\lambda}_i, \ldots)$ as follows: Assume that λ_i is sorted in decreasing

order (we can always achieve this by permuting the columns of Q). If one is given a computation budget that dictates the use of Φ with a certain rank r (the rank of Φ is going to control the computational cost of optimizing $J(\cdot)$), then one can simply let $\tilde{\lambda}_i = \lambda_i \mathbb{I}\{i \leq r\}$. Once $\tilde{\Lambda}$ is given, set $\Phi = Q\tilde{\Lambda}^{1/2}$.

Another method is as follows: Choose $\alpha \in (0,1)$, a variable that controls how much of the "variation" is kept. Then let $s_i = \sum_{k=1}^{i} \lambda_k$ and set $\tilde{\lambda}_i = \lambda_i \mathbb{I}\{s_i \leq \alpha s_N\}$. In practice, one can start with a low value of α and then if given more computation time, gradually increase it.

4.4 Simple Rounding Procedure

Once we have the optimal allocation vector γ^*, we need a *rounding procedure* to get the sequence of inputs that should be chosen. One possible rounding procedure is presented in Algorithm Listing 4.6.

Algorithm 4.6: Rounding Procedure.

Input: γ^* (optimal allocation vector), n (desired length of output sequence),
$A = \{1, ..., |A|\}$ (alphabet)

Output: $(a_1, ..., a_n) \in A^n$ (sequence of inputs chosen for the experiment)

`/* Procedure */`

$\forall a \in A,\ e(a) \leftarrow (0, \cdots, 0, \overset{a}{1}, 0, \cdots, 0)^\top$;

$\gamma \leftarrow (0, \ldots, 0)^\top \in \mathbb{R}^{|A|}$;

for $i = 1$ **to** n **do**

  $a_i \leftarrow \text{argmax}_{a \in A} \left\{(\gamma^* - \gamma)^\top (e(a) - \gamma)\right\}$;

  $\gamma \leftarrow \gamma + \frac{e(a_i) - \gamma}{i}$

end

return $(a_1, ..., a_n)$

Chapter 5

Efficiency Concerns

This chapter present optimized ways to deal with problems encountered conducting this research.

5.1 Generalized Cross-Validation

This approach has a long history and goes back to Wahba (1979). Our treatment follows that of Rifkin and Lippert (2007). Remember that we have to determine the regularization parameter λ: it has to be small enough to prevent underfitting, but big enough to prevent overfitting. We introduced a trivial method to solve this problem in Section 3.2.3 that has a computational complexity of $O(n^4)$ per tested λ, which is highly unscalable to big spaces of algorithms. The idea of the more efficient method is as follows.

First observe that if we define $\hat{r}_i = f_{D_{-i},\lambda}(a_i)$ and

$$D'_i = ((a_1, r_1), \ldots, (a_{i-1}, r_{i-1}), (a_i, \hat{r}_i), (a_{i+1}, r_{i+1}), \ldots, (a_n, r_n))$$

then $f_{D'_i,\lambda}(a) = f_{D_{-i},\lambda}(a)$ holds for all inputs a. Roughly speaking introducing $(a_i, \hat{r}_i)$ does not "pisses off" $f_{D_{-i},\lambda}$. Once we realize the above identity, the next observation is that the Grammian matrix underlying D'_i is the same as the Grammian matrix underlying D. This observation gives us a $O(n^3)$ procedure to calculate $e(\lambda)$ for a fixed value of λ.

For this, define $K_\lambda = K + \lambda I$ ($K_{i,j} = K(a_i, a_j)$ as before). Next, observe that if we define $\hat{R}_{D,\lambda} = (\ldots, f_{D,\lambda}(a_i), \ldots)^\top$ to be the vector of predicted values when training on D with regularizer λ, then $\hat{R}_{D,\lambda} = KK_\lambda^{-1}R$. Hence, defining $R'_i = (r_1, \ldots, r_{i-1}, \hat{r}_i, r_{i+1}, \ldots, r_n)^\top$,

$$\hat{R}_{D_{-i},\lambda} - \hat{R}_{D,\lambda} = \hat{R}_{D'_i,\lambda} - \hat{R}_{D,\lambda} = KK_\lambda^{-1}(R'_i - R_i).$$

Since $R'_i - R_i = (\hat{r}_i - r_i)e_i$, i.e., it has zeroes everywhere except at the ith component where it evaluates to $(\hat{r}_i - r_i)$ (e_i is the ith unit vector of the standard Euclidean basis) and for the ith component of the difference vector on the right-hand side we have

$$e_i^\top(\hat{R}_{D_{-i},\lambda} - \hat{R}_{D,\lambda}) = \hat{r}_i - f_{D,\lambda}(a_i),$$

it follows that

$$\hat{r}_i - f_{D,\lambda}(a_i) = (\hat{r}_i - r_i)e_i^\top KK_\lambda^{-1}e_i.$$

Solving for $\hat{r}_i$ we get

$$\hat{r}_i = \frac{f_{D,\lambda}(a_i) - r_i e_i^\top K K_\lambda^{-1} e_i}{1 - e_i^\top K K_\lambda^{-1} e_i} = \frac{e_i^\top K K_\lambda^{-1} R - r_i e_i^\top K K_\lambda^{-1} e_i}{1 - e_i^\top K K_\lambda^{-1} e_i}\,.$$

To calculate $e(\lambda)$, we need to calculate $\frac{1}{n}\sum_i (r_i - \hat{r}_i)^2$. We have

$$\begin{aligned} r_i - \hat{r}_i &= r_i - \frac{e_i^\top K K_\lambda^{-1} R - r_i e_i^\top K K_\lambda^{-1} e_i}{1 - e_i^\top K K_\lambda^{-1} e_i} \\ &= \frac{r_i(1 - e_i^\top K K_\lambda^{-1} e_i) - \left\{e_i^\top K K_\lambda^{-1} R - r_i e_i^\top K K_\lambda^{-1} e_i\right\}}{1 - e_i^\top K K_\lambda^{-1} e_i} \\ &= \frac{r_i - e_i^\top K K_\lambda^{-1} R}{1 - e_i^\top K K_\lambda^{-1} e_i}. \end{aligned}$$

We calculate the numerators for all indices using $R - K K_\lambda^{-1} R = (I - K K_\lambda^{-1}) R$. We can also calculate the denominators for all indices as the diagonal elements of $I - K K_\lambda^{-1}$ (indeed, $e_i^\top (I - K K_\lambda^{-1}) e_i = 1 - e_i^\top K K_\lambda^{-1} e_i$). Now, it is easy to verify that $I - K K_\lambda^{-1} = \lambda K_\lambda^{-1}$ (just multiply from the left using K_λ), leading to

$$r_i - \hat{r}_i = \frac{e_i^\top K_\lambda^{-1} R}{e_i^\top K_\lambda^{-1} e_i}. \tag{5.1}$$

Thus, by inverting K_λ we can calculate $e(\lambda)$ using the above formula at an additional cost of $O(n^2)$ operations: calculating $K_\lambda^{-1} R$ takes $O(n^2)$ time. Once this vector and K_λ^{-1} are calculated, $r_i - \hat{r}_i$ takes two lookups and one division, i.e., $O(1)$ time.

The next improvement is to reduce the cost of evaluating K_λ^{-1} at M different values to $O(n^3 + Mn^2)$ from the trivial $O(Mn^3)$. This can be done by considering the eigendecomposition of K. Assume that $K = QDQ^\top$, where D is a diagonal matrix with non-negative diagonal elements and $QQ^\top = I$. Note that the eigendecomposition of K can be obtained in $O(n^3)$ time (and the actual cost is roughly within a factor of 4 of inverting an $n \times n$ matrix). Now,

$$K_\lambda = K + \lambda I = Q(D + \lambda I) Q^\top$$

and hence

$$K_\lambda^{-1} = Q(D + \lambda I)^{-1} Q^\top$$

thanks to $Q^{-1} = Q^\top$.

Consider now the calculation of $r_i - \hat{r}_i$ based on (5.1). Consider first the numerator: $K_\lambda^{-1} R = Q(D + \lambda I)^{-1} Q^\top R$. To evaluate this expression, we can precompute $\tilde{R} = Q^\top R$ in $O(n^2)$ time (this expression does not need to be reevaluated for a new value of λ). Once done, $(D + \lambda I)^{-1} \tilde{R}$ can be implemented in $O(n)$ time (just multiply the ith element of $\tilde{R}$ by $1/(D_{ii} + \lambda)$) and the resulting vector can be left-multiplied by Q at a cost of $O(n^2)$. Thus, evaluating $K_\lambda^{-1} R$ takes $O(n^2)$ time, which gives the numerator for all indices i. To evaluate the denominator, using Matlab notation ($Q_{i:}$ denotes the ith row vector of Q) write

$$(Q(D + \lambda I)^{-1} Q^\top)_{i,j} = Q_{i:}^\top (D + \lambda I)^{-1} Q_{j:} = \sum_{k=1}^{n} \frac{Q_{i,k} Q_{j,k}}{D_{kk} + \lambda},$$

which takes $O(n)$ time. Hence, evaluating $(Q(D + \lambda I)^{-1} Q^\top)_{i,i}$ for all $i = 1, \ldots, n$ takes $O(n^2)$

time. Thus, we see that evaluating (5.1) takes $O(n^2)$ time, once we have the eigendecomposition of K.

5.1.1 Normalizing the Data With GCV

To normalize the data when we use GCV optimization, we apply a slightly different trick than the one presented in Section 3.2.2 for the naive version. The trick was reduced to just modify the vector of rewards R such that

$$R' = R - \frac{1}{n}\sum_{i=1}^{n} R_i$$

Indeed, we do not need to add the average back anymore, as

$$r_i - \hat{r}_i = (r_i - avg) - (\hat{r}_i - avg) = r'_i - \hat{r}'_i$$

Moreover, the average value towards which GCV version of $e(\lambda)$ will converge when $\lambda \to \infty$ is the real mean of all the rewards from D, instead of the mean of the $n-1$ rewards from $D_{-i,\lambda}$ for the naive version. Therefore, even for small training sets, normalized GCV is much more reliable than normalized naive version, as for small training sets it will at least converge to the real average of the rewards if *lambda* is too big, leading to a smaller error on the predictions.

5.1.2 Automatic Determination of Lambda Interval

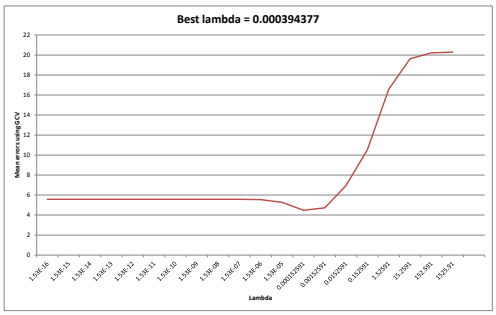
Again to get rid of constants, we implemented a way to determine dynamically the interval where to search for the best lambda. Once we have the eigendecomposition of K, and in particular when we have the diagonal matrix D, we define the interval as

$$\left[0.1 \min_{D_i > 0} D_i \; ; \; 10 \max D_i\right].$$

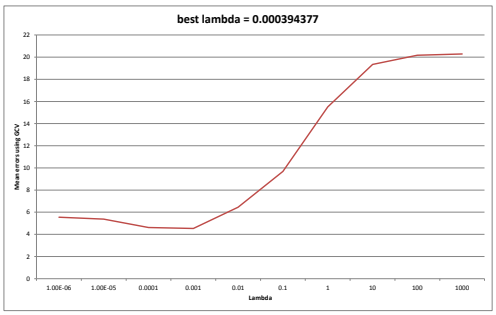
We tested this automatic interval discovery against a predefined interval to compare whether or not the automatic way would leave the best lambda out of the interval.

Figure 5.1 shows the results for a training set composed of 720 samples, while Figure 5.2 shows the results for a training set of 1088 samples. We clearly see that for both situation, the automatic discovery always enclosed the minimum, leading to the same best lambda.

Figure 5.1: Mean error on the reward for different intervals (|Dataset| = 720)

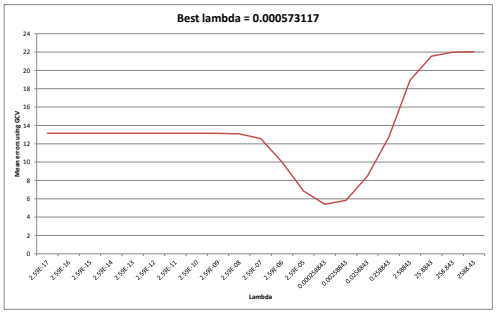


(a) Automatic interval

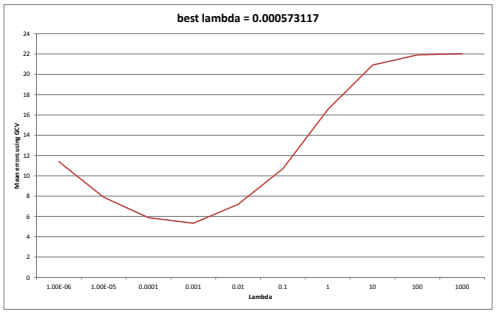


(b) Predefined interval

Figure 5.2: Mean error on the reward for different intervals (|Dataset| = 1088)



(a) Automatic interval



(b) Predefined interval

Chapter 6

Conclusion

The goal of this internship was to find a faster and more reliable way to discover the best algorithm to use from a huge space of candidate algorithms.

We succeeded in describing this new method and developing the tools allowing to implement it. A study has been conducted for each part of the proposed algorithm, to find the best ways to deal with the challenges we faced during its elaboration.

Sadly, the internship was not long enough to conduct tests in order to gather meaningful results. The reason for that is mainly the undetermined constants: you can see in (3.12) for example the S constant that we cannot determine for now; magnitude of S could be in fact determined empirically, but the impact of the choice of such an S is left as future work. Therefore Algorithm 2.1 cannot be run in an optimal way, and thus we cannot compare it to some baseline algorithms. We will present in Section 6.1 what kind of baseline should be used and on which criterion(s) one should compare the performance of both algorithms in order to have meaningful results.

Nevertheless, a quasi-complete implementation of the Proposed Algorithm (2.1) has been developed. At the end of this internship, the algorithm has been implemented and can be run with only a slight difference from what has been told in this document: as we ran into some problems implementing Cocktail algorithm 4.5, and in particular when calling NNE algorithm 4.4, we ignored the call to NNE. This way, the algorithm that computes the confidence bands is still able to run, even if it is a bit slower to converge than the real Cocktail algorithm.

6.1 Future Work

The first thing to do, obviously, is to conduct proper tests on the implemented prototype algorithm, at least just to see if in optimal conditions (ie., with optimal constants) it beats a baseline. One should first plot the regret on the best arm chosen by our algorithm as a function of the number of samples the algorithm took so far, then plot the regret as a function of the amount of samples for the baseline algorithm. An example of baseline algorithm in the particular case of MCS algorithm discovery is described in Algorithm Listing 6.1.

Next, there is still a big work to do with kernels. We tried only one, but maybe with others we could achieve better results: indeed, this method hugely depends on the kernel it uses, so a kernel change can drastically increase (or decrease!) the results of this method. Unluckily we did not have the time to implement more than one, as we have not conducted tests for the first one yet.

Algorithm 6.1: Baseline algorithm.

Input: $D = [(a_1, \overline{r}_{a_1}), ..., (a_{1088}, \overline{r}_{a_{1088}})]$, our data set containing 1088 algorithms and their true mean rewards

Output: a^*, the list of best arms found, and R, the list of best arms regrets.

`/* Procedure */`

$a^* \leftarrow []$;
$R \leftarrow []$;
$data \leftarrow []$;
$bestTrueMean \leftarrow \max D$.trueMean($a$);
for $i = 8$; $i \leq 1024$; $i \leftarrow 2i$ **do**
    for $j =$ data.*size()* **to** $i - 1$ **do**
        $a \leftarrow$ randomly pick an algorithm from D;
        $r \leftarrow a$.execute();
        data.append((a, r));
    end
    Train a regressor *Reg* with *data*;
    a^*.append(findBestArm(*Reg*, *data*));
    R.append(*bestTrueMean* $-$ D.trueMean(a^*));
end
return a^*, R

Nevertheless, we presented encouraging results in Section 3.3.1. A future work on this subject should really explore more the possible kernels and deeply study their characteristics and properties regarding this problem.

Bibliography

Abbasi-Yadkori, Y. (2012). Online learning for linearly parametrized control problems.

Abbasi-Yadkori, Y., Szepesvári, C., and Pál, D. (2011). Improved algorithms for linear stochastic bandits. In *Advances in Neural Information Processing Systems*, pages 2312–2320.

Ernst, D., Lupien St-Pierre, D., and Maes, F. (2013). Monte carlo search algorithm discovery for one player games. *IEEE Transactions on Computational Intelligence and AI in Games*.

Rifkin, R. M. and Lippert, R. A. (2007). Computer science and artificial intelligence laboratory technical report.

Shawe-Taylor, J. and Cristianini, N. (2004). *Kernel Methods for Pattern Analysis*. Cambridge University Press.

Wahba, G. (1979). How to smooth curves and surfaces with splines and cross-validation.

Yu, Y. (2011). D-optimal designs via a cocktail algorithm. *Statistics and Computing*, 21(4):475–481.