Machine Learning and AI

- Methods and Algorithms -

Personnal Notes

François Bouvier d'Yvoire

CentraleSupélec & Imperial College

Current Branch : `master`
Commit : `8de300ebd968cb77fe60cbab317d86a51c43a991`

Contents

Todo list

Intro

This document will use the following classification for the machine learning algorithms. However their might be some changes. For exemple, some of them will be part of the commons algorithms and not from their real class.

Figure 1 – Simple graph for algorithms classification in ML

Chapter 1

Common Machine Learning algorithms

This chapter is dedicated to the most common ML algorithms, a major part of the notes come from the mml-books.com

Add bibtex reference

Find better paragraph layout

1.1 Linear Regression

1.1.1 Maximum Likelihood Estimation (MLE)

Closed-Form Solution

In some cases, a closed-form solution exist, which make computation easy (but not necesseraly cheap)

Maximum A Posteriori Estimation (MAP)

1.2 Gradient Descent

1.2.1 Simple Gradient Descent

1.2.2 Gradient Descent with Momentum

1.2.3 Stochastic Gradient Descent

1.3 Model Selection and Validation

1.3.1 Cross-Validation

1.3.2 Marginal Likelihood

1.4 Bayesian Linear Regression

1.4.1 Mean and Variance

1.4.2 Sample function

Chapter 2

Reinforcement Learning

2.1 Markov Reward and Decision Process

2.1.1 State Value Function Closed-form

For a Markov Reward Process $(\mathcal{S}, \mathcal{P}, \mathcal{R}, \gamma)$, with

- $\mathcal{S}$ the States
- $\mathcal{P}$ The transition matrix
- $\mathcal{R}$ the reward matrix
- $\gamma$the discount

We define then the Return R_t and the State Value Function $\mathsf{v}(s) = \mathbb{E}[\mathsf{R}_t S_t = s]$
Then we have, in a vector form :

$$\mathbf{v} = (\mathbb{1} - \gamma\mathcal{P})^{-1}\mathcal{R}$$

Unfortunately, Matrix inversion in costly, so this is only feasible in small Markov Reward Process

2.1.2 Iterative Policy Evaluation Algorithm

In RL, we need not only the value of different state, but also a policy to define which action to take in any state. Then, we had an action space $\mathcal{A}$ and a policy π to a MRP to obtaine what is called a Markov Decision Process (MDP).

The first algorithm is a method to evaluate a policy. We don't need to store all the previous value of V while updating, and furthermore, it converges faster this

way.

Data: a MDP $(\mathcal{S}, \mathcal{P}, \mathcal{A}, \mathcal{R}, \gamma)$ and π the policy to be evaluate
Result: The value function V^{π}
Initialise V(s) =0 $\forall s$;
while **Convergence condition (number of epoch, $\Delta \leq$ threshold, ...)** **do**
  forall $s \in S$ **do**
    $v \leftarrow V(s)$;
    $V(s) \leftarrow \sum_{\alpha} \pi(s, \alpha) \sum_{s'} \mathcal{P}^{a}_{ss'} [\mathcal{R}^{a}_{ss'} + \gamma V(s')]$;
    $\Delta \leftarrow \max(\Delta, \mid v - V(s) \mid)$;
  end
end
Return V

Algorithm 1: Iterative Policy Evaluation Algorithm

2.2 Dynamic Programming in RL

In order to compute an optimal policy, we can do better than trying every policy and evaluating their V^{π}, the first class of algorithm enter in the Dynamic programming family. Convergence are assured by theorem relative to the Belmann equation (Policy Improvement theorem, Belmann principle of optimality, ...).

2.2.1 Policy Iteration Algorithm

First algorithm simply apply a greedy **deterministic** policy relative to the Value function, and recalculate the new Value function, until the policy is stable (and according to theorem, optimal).

Data: a MDP $(\mathcal{S}, \mathcal{P}, \mathcal{A}, \mathcal{R}, \gamma)$
Result: The optimal policy $\pi*$ and his corresponding value function $V^{\pi*}$
Initialise V(s)=0 and $\pi(s) \in \mathcal{A}, \forall s$;
while **Convergence condition (policy is stable, or number of epoch)** **do**
  Do a Policy Evaluation for π (with the Iterative Policy Evaluation for exemple) **forall** $s \in S$ **do**
    $b \leftarrow \pi(s)$;
    $\pi(s) \leftarrow \arg\max_{a} \sum_{s'} \mathcal{P}^{a}_{ss'} [\mathcal{R}^{a}_{ss'} + \gamma V(s')]$;
    policy-not-stable $\leftarrow (b \neq \pi(s)) \vee$ policy-not-stable
  end
end
Return V

Algorithm 2: Iterative Policy Evaluation Algorithm

This can be seen as the following scheme :

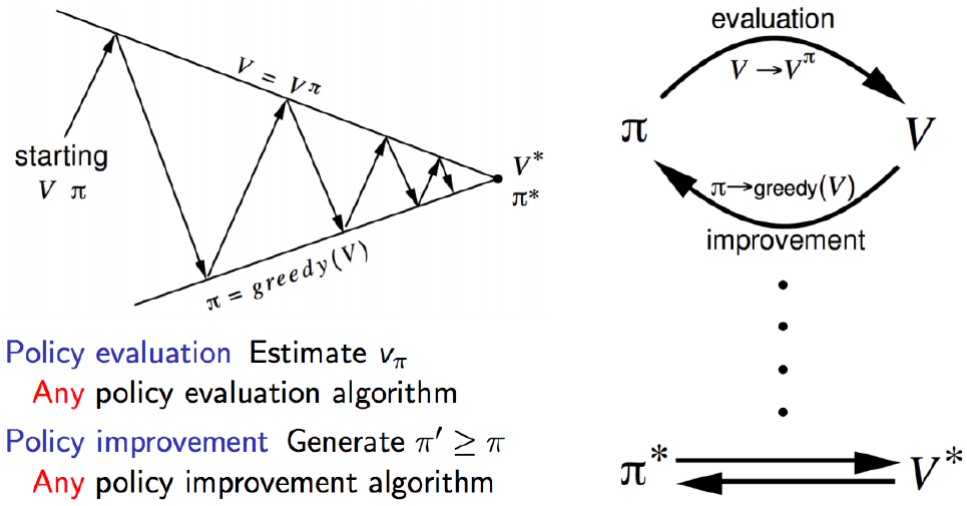

Figure 2.1 – Generialised Policy Iteration Algorithm

2.2.2 Value Iteration Algorithm

It can be shown that we don't need to wait for the convergence of the Value Function before iterating the policy in the previous algorithm. We can shorten some steps, and even get rid of explicity updating a policy (we are always using the greedy policy wrt. $V(s)$). This lead to the following algorithm :

Data: a MDP $(\mathcal{S}, \mathcal{P}, \mathcal{A}, \mathcal{R}, \gamma)$
Result: The optimal policy $\pi*$ and his corresponding value function $V^{\pi*}$
Initialise V(s)=0 $\forall s$;
while **Convergence condition (Value function is stable, or number of epoch)** **do**
  $\Delta = 0$;
  forall $s \in S$ **do**
    $v \leftarrow V(s)$;
    $V(s) \leftarrow \max_a \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V(s')]$;
    $\Delta \leftarrow \max(\Delta, | v - V(s) |)$;
  end
end
Return $\forall s, \pi*(s) = \arg\max_a \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V(s')]$;

Algorithm 3: Iterative Policy Evaluation Algorithm

2.2.3 Assynchronous Backup in RL

Definition 1

- *The **Backup** in RL is the fact to update the value of a state using values of futures state states. This can be understood as you check what's going on in the future, and backup this information to present time to update your knowledge.*
- *A **synchronous backup** is a backup made simultaneously for all states (can be done in parallel).*
- *An **assynchronous backup** apply a backup to only one selected state. Those methods can be much more efficient in reducing computation (only deal we required backup) and are still guaranteed to converge.*

For exemple the policy iteration algorithm can be compute both way, but the assynchronous way converge faster.

Prioritised Sweeping

Definition 2

*A **sweep** consists of applying a backup operation to each state. A synchronous backup compute a sweep at every iteration whereas a assynchronous backup may never compute a complete sweep (depending on the strategy).*

*A **prioritised swweping** use a criteria and a priority queue in order to decide which value will be backup next.*

An implementation of a prioritised sweeping can be made using the **Bellman Error**

$$\| v(s) - \max_a \left(\mathcal{R}_s^a + \gamma \sum_{s'} P[s'|s,a] v(s') \right) \|$$

as priority criteria. It requires the knowledge of the **reverse dynamics** $P[s'|s,a]$.

Data: a MDP $(\mathcal{S}, \mathcal{P}, \mathcal{A}, \mathcal{R}, \gamma)$
Result: The optimal policy $\pi*$ and his corresponding value function $V^{\pi*}$
Initialise V(s)=0 and the Bellman Error $B(s), \forall s$;
Initialise the priority queue in ordering the s with $B(s)$;
while **Convergence condition (Value function is stable, or number of epoch)** **do**
  select s the head of the queue $v \leftarrow V(s)$;
  $V(s) \leftarrow \max_a \sum_{s'} \mathcal{P}_{ss'}^a [\mathcal{R}_{ss'}^a + \gamma V(s')]$;
  Update $B(s)$ and the priority queue ;
end
Return $\forall s, \pi*(s) = \arg\max_a \sum_{s'} \mathcal{P}_{ss'}^a [\mathcal{R}_{ss'}^a + \gamma V(s')]$;

Algorithm 4: Prioritised sweeping with Bellman error

Real-time Dynamic Programming The idea here is to use the **agent** experience to guide the selection of states. Each iteration, you backup the state was just visited.

Data: a MDP $(\mathcal{S}, \mathcal{P}, \mathcal{A}, \mathcal{R}, \gamma)$
Result: The optimal policy $\pi*$ and his corresponding value function $V^{\pi*}$
Initialise V(s)=0 $\forall s$;
choose a starting state $s = s_0$;
while **Convergence condition (Value function is stable, or number of epoch)** **do**
  $v \leftarrow V(s)$;
  $V(s) \leftarrow \max_a(\mathcal{R}^a_s + \gamma \sum_{s'} P[s'|s,a]V(s'))$;
  choose the next step from s (greedy policy);
end
Return $\forall s, \pi*(s) = \arg\max_a \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V(s')]$;

Algorithm 5: Prioritised sweeping with Bellman error

2.2.4 Properties and drawbacks of Dynamic Programming

- Dynamic Programming uses **full-width** backup $\Longrightarrow$ faster convergence but requires lot of ressource
- Require **complete** knowledge of the MDP $\Longrightarrow$ this is not really Machine Learning, but more optisation (all parameters are actually known).
- DP is effective for **medium size problems** (millions of state, backup not to expensive)

2.3 Model-Free Learning

The dynamic programing requires lot of information to be compute, especially what is called the **model**, the states, the actions, the rewards and the transitions. We might want to learn form model which are difficult to observe, and with partial information, etc. That's what we need **model-free learning** algorithms, which can deal with model without knowing in advance the transition and the rewards.

2.3.1 Monte-Carlo Algorithms

Monte-Algorithms are the class of algorithms which learn from episodes of experiences. Which mean basicaly, you play the story fully with what you know, then analyse what append, learn some knowledge from this, and try again. This is different from DP because here you are learning from "real life" experience.

(First Visit) Monte-Carlo Policy Evaluation The first visit MC only take into account the returns from the first visit (i.e. if a state is visited more than once in a trace, it does not count)

Data: a uncomplete MDP ($\mathcal{S}$, $\mathcal{R}$, traces T wich contain series of state and reward, π a policy, and still γ)
Result: The State Value of the MDP for π
Initialise $\hat{V}(s), \forall s$;
Initialise $Returns(s), \forall s$ with empty list **while** **Convergence condition (Value function is stable, or number of epoch)** **do**
  Get a trace τ from T ;
  forall **s in τ** **do**
    R $\leftarrow$ return from the **first appearance** of s in τ ;
    Append R to Returns(s) ;
    $\hat{V}(s) \leftarrow average(Returns(s))$;
  end
end
Return $\hat{V}$;

Algorithm 6: First Visit Monte-Carlo Policy Evaluation

Every Visit Monte-Carlo Policy Evaluation You can have the same algorithm, with taking into account all the visit from a trace.

Data: a uncomplete MDP ($\mathcal{S}$, $\mathcal{R}$, traces T wich contain series of state and reward, π a policy, and still γ)
Result: The State Value of the MDP for π
Initialise $\hat{V}(s), \forall s$;
Initialise $Returns(s), \forall s$ with empty list **while** **Convergence condition (Value function is stable, or number of epoch)** **do**
  Get a trace τ from T ;
  forall **s in τ** **do**
    R $\leftarrow$ returns from **all appearances** of s in τ ;
    Append R to Returns(s) ;
    $\hat{V}(s) \leftarrow average(Returns(s))$;
  end
end
Return $\hat{V}$;

Algorithm 7: Every Visit Monte-Carlo Policy Evaluation

You cannot backup death : if the trace lead to a death state (with no reward during the trace, or infinite negative reward), then the MC will not be able to backup

any sensitive data.

Batch vs Online Monte-Carlo

Incremental Monte-Carlo Update

Runing Mean for Non-Stationnary World

2.3.2 Monte-Carlo Control Algorithms

Monte-Carlo Policy Improvement

Greedy Policy Improvement over State Value Function

Greed Policy Improvement over State-Action Value Function

Exploring Starts Problem

Don't forget Starting to explore

On Policy Soft Control

On-Policy ϵ-greedy first-visit Monte-Carlo control Algorithm

Monte-Carlo Batch Learning to Control

Monte-Carlo Iterative Learning to Control

2.3.3 Temporal Difference Learning

Temporal Difference Value Function Estimation Algorithm

Add Comparison between MC and TD learning

2.3.4 Temporal Difference Learning Control Algorithm

SARSA - On Policy learning Temporal Difference Control: Here is the Sarsa Algorithm :

Data: State $\mathcal{S}$, Action $\mathcal{A}$, Reward $\mathcal{R}$ and Discount γ
Result: The optimal Q(S, A) State-Action Value Function and a greedy policy w.r.t Q
Initialise Q(s, a) $\forall a, s$ with Q(terminal state, a) = 0 ;
while **Convergence condition (number of epoch, $\Delta \leq$ threshold, ...)** **do**
  Initialise a state S;
  Choose action A from S with ϵ-greedy policy derived from Q;
  while **S is not a terminal State** **do**
    Take action A, observe reward R and next state S';
    Choose action A' from S' with ϵ-greedy policy derived from Q;
    Update $Q(S,A) \leftarrow Q(S,A) + \alpha(R + \gamma Q(S',A') - Q(S,A))$;
    $S \leftarrow S', A \leftarrow A'$
  end
end
Return Q and π the derived policy

Algorithm 8: SARSA algorithm with ϵ-greedy policy

Theorem 1
Convergence of Sarsa
$Q(s,a) \rightarrow Q^{\infty}(s,a)$ *under :*

- *GLIE (Greedy in the Limite with infinite exploration), which mean every state is visited infinitely many times and that the policy converge toward a greedy-policy (ex: ϵ-greedy with $\epsilon \rightarrow 0$).*
- *Robbins-Monroe sequence of step-sizes α_t : which imply $\sum \alpha_t$ diverge and $\sum \alpha_t^2$ converge.*

Remark 1
the ϵ-greedy policy can be replaced by any policy derived from Q. (Because Q is the one updated by the algorithm)

SARSA-Lambda

Hindsight Experience Replay

Q-Learning: Off-Policy Temporal Difference Learning

2.4 Reinforcement Learning with Function Approximation

2.4.1 Exemple of features

Coarse Coding

Tile Coding

Radial-Basis Function

Deep Learning

2.4.2 Monte-Carlo with Value Function Approximation

2.4.3 Temporal Difference Learning with Value Function Approximation

2.4.4 Q-Learning with FA

2.4.5 subsection name

2.5 Deep Learning Reinforcement Learning

2.5.1 Experience Replay

2.5.2 Target Network

2.5.3 Clipping of Rewards

2.5.4 Skipping of Frames

Chapter 3

Dimensionality Reduction and Feature Extraction

3.1 Principal Component Analysis

The objective of PCA is to find a set of features, via linear projections, that maximise the variance of the sample data. We need to decide how many dimension d we want to keep.

3.1.1 Simple algorithm

Data: *Vectors x_i of **centered** data, number F features and n sample. Dimension d of reduction.*
Result: *Y of size $d \times n$*
Compute the product matrix of centered data : $\boldsymbol{X}\boldsymbol{X}^\top$;
Compute the Eigen Analysis $XX^\top = V\Lambda V^\top$;
Order the Eigen Value by descending value, and permute Column of V correspondly;
Compute the eigenvectors : $U = XV\Lambda^{-1/2}$;
Keep specific number of first components : U_d the d first d column of U ;
Compute the new features vectors : $Y = U_d^\top X$

Algorithm 9: Simple PCA Algorithm

3.1.2 Whitening PCA

The feature given by the PCA algorithm are un-correlated, but the variance in each dimension are not the same (in fact this are the eigen value of $XX^\top$). We can whitening the

features (or "sphering" them) by making the covariance matrix equal to Identity.

Data: *Vectors x_i of **centered** data, number F features and n sample. Dimension d of reduction.*
Result: *Y of size $d \times n$ with Identity Covariance Matrix*
Compute the PCA of X and keep U and Λ;
Compute the Eigen Analysis $XX^\top = V\Lambda V^\top$;
Compute the whitened features vectors : $Y = U_d \Lambda^{-1/2} X = (XV\Lambda^{-1})_d X$

Algorithm 10: Whitened PCA Algorithm

3.1.3 Kernel PCA

Sometimes we want to compute non-linear features extractions. We use the kernel method to compute this. For a dataset $X = (x_i)_{1\dots n}$ we know only the kernel matrix $K = [\phi(x_i)\phi(x_j)^\top]_{(1\dots n)^2} = X^\phi X^{\phi^\top}$ with ϕ the non-linear mapping.

Data: *Vectors x_i of **centered** data, number F features and n sample. Dimension d of reduction. K the kernel matrix*
Result: *Y of size $d \times n$*
Compute the Eigen Analysis $K = \boldsymbol{X}^\phi \boldsymbol{X}^{\phi^\top} = V\Lambda V^\top$;
Order the Eigen Value by descending value, and permute Column of V correspondly;
Keep specific number of first components : V_d the d first d column of V ;
Compute the vector $g(x_t) = [k(x_i, x_t)]_{1..n}$;
Compute the $E = \frac{1}{n}\mathbf{1}\mathbf{1}^\top$ matrix ;
Compute the new features vectors : $y_t = \Lambda^{-1/2} V^\top (I - E)\left(g(x_t) - \frac{1}{n} K\mathbf{1}\right)$

Algorithm 11: Kernel PCA Algorithm

Chapter 4

Argumentation Framework

This chapter are notes from the Imperial Course Machine Arguing from Francesca Toni. add ref

introduction *Argument Framework are a field in AI which provide way of evaluate any debate problem. It is useful to resolve conflict, to explain decision or to deal with incomplete information.*

4.1 Abstract Argumentation

4.1.1 Simple AA

Definition 3
*an **AA framework** is a set* Args *of arguments and a binary relation* attacks. $(\alpha, \beta) \in$ attacks *means α attacks β.*

Semantics in AA *In order to define a "winning" set of argument, we need to provide semantics over the the framework. This is like recipes which determine good set of arguments.*

Definition 4

- ***conflict-free***
- ***admissible:*** *c-f and attacks each attacking argument.*
- ***preferred:*** *maximally admissible.*
- ***complete:*** *admissible + contains each argument it defends.*
- ***stable:*** *c-f + attacks each argument not in it.*
- ***grounded:*** *minimally complete.*

- ***sceptically preferred:*** *Intersection of all prefered.*
- ***ideal:*** *maximal admissible and contain in all prefered (i.e. in the sceptically preferred).*

Definition 5
Semi-stable extension: *complete such as $A \cup A^+$ is maximal. A^+ is the set of attacked argument by A.*

add ref to ASPARTIX and CONARG

4.1.2 Algorithms for AA

Computing Grounded extensions *Use the same algorithms as grounded labelling, but only output the IN arguments as the grounded extensions.*
the grounded extensions in unique.

Computing the grounded labelling: *Here is an algorithm to compute a grounded labelling*

Data: *An AA Framework*
Result: *The grounded Labelling*
Label all unatacked argument with IN ;
while **The IN and the OUT are not stable** **do**
    Label OUT the arguments attacks by IN ;
    Label IN the arguments only attacked by OUT;
end
Label the still unlabelled UNDEC;

Algorithm 12: Computing the grounded labelling

Computing membership in preferred/grounded/ideal extensions; *In order to compute membership, we use Dispute Tree.*

We compute a dispute tree for an argument, and apply the different semantics which are easier to compute on a tree than on a graph.

add algos of computing dispute tree + def of semantics

Data: *An AA Framework, and a arg* α
Result: *Finite or infinite dispute tree corresponding to* α
add a root node Label α *as proponent ;*
while The tree is not stable **do**
  for every $\beta \in$ *proponent, and for every* (γ, β) *add a child-node* γ *as opponent;*
  for every $\beta \in$ *opponent, and for every* (γ, β) *add a node* γ *as proponent in the limit of one child-node per opponent ;*
end

Algorithm 13: Computing dispute Tree

Computing stable extension: *We use answer set programming with logical program.*

4.1.3 AA with Suppport

Bipolar Abstract Argumentation

We add a ***Support*** *relation to a classic AA Framework (*$BAA = \langle Args, Attacks, Supports \rangle$*). There are different semantics :*

Semantics in BAA with deductive support *We can deduce an AA Framework from a BAA with* $Attacks' = Attacks \cup Attacks_{sup} \cup Attacks_{s-med}$ *with*

Definition 6

- $Attacks_{sup}$ *is* ***the supported attacks*** $\implies$ α *attacks every argument that its supports attacks (supports of supports are supports)*
- $Attacks_{s-med}$ *is the super mediated attacks* $\implies$ α *attacks every argument whose supports an argument attacked or* $attacked_{sup}$ *by* α

Then, we apply the AA semantics to $\langle Args, Attacks' \rangle$*. Those semantics are the d-X semantics, where X replace every semantic from AA (grounded, complete, etc.)*

QuAD Framework *We focus here one the QuAD (****Quantitative Argument Debate****) which add a numerical strenght to any argument, and give rule for updating strenght regarding the supporters or attackers.*

Definition 7

- *We define the QuAD Framework as* $\langle \mathcal{A}, \mathcal{C}, \mathcal{P}, \mathcal{R}, \mathcal{BS} \rangle$ *Where* $\mathcal{A}$ *is the set of answer arguments,* $\mathcal{C}$ *is the set of con args,* $\mathcal{P}$ *are the pro args, which are disjoint,* $\mathcal{R}$ *the relation (with* $\mathcal{R}^+$ *the supporters and* $\mathcal{R}^-$ *the attackers) and* $\mathcal{BS}$ *the Base Score, which give score to each args.*
- *The* ***Base funcion*** *is* $f(v_1, v_2) = v_1 + (1 - v_1)v_2$

- *the **aggregation function** as a recursive definition :*

$$\begin{aligned} n = 0 &\implies \mathcal{F}(S) = 0 \\ n = 1 &\implies \mathcal{F}(S) = v_1 \\ n = 2 &\implies \mathcal{F}(S) = f(v_1, v_2) \\ n > 2 &\implies \mathcal{F}(S) = f(\mathcal{F}(S \backslash \{v_n\}, v_n)) \end{aligned}$$

- *the **Combination function** is defined by:*
  $c(v_0, v_a, v_s) = v_0 - v_0 \cdot |v_s - v_a|$ *if* $v_a \geq v_s$
  $c(v_0, v_a, v_s) = v_0 + (1 - v_0) \cdot |v_s - v_a|$ *if* $v_a < v_s$
- *the **Strength Function** is defined as :* $\mathcal{S}(\alpha) = c(\mathcal{BS}(\alpha), \mathcal{F}(SEQ_S(\mathcal{R}^-(\alpha))), \mathcal{F}(SEQ_S(\mathcal{R}^+(\alpha))))$

*The DF-QuAD (for **Discontinuity free**-QuAD) algorithm is simply to apply these semantics to a BAA framework, in setting initial weights to any arguments.*

4.1.4 Argument Mining

4.1.5 AA with Preference

When taking into account preference in a AA, (in the form $X < Y = X$ is less prefered to Y) we can still use the classic AA semantics but we can improved it by using the preferences informations :

- *By **deleting attacks** (an (x, y) attacks succeed iff $x \not\prec y$)*
- *By **reversing attacks** (if (x, y) and $x < y$ then we reverse by replacing this attack by (y, x)).*
- *By **Selecting Amongst extension**, using the preference to select extension with the same extension (ex : differentiate 2 stable extensions). Look at **Rich PAF** for some formal definition.*

4.1.6 AA with Probabilities

4.2 Assumption-Based Argumentation

***Assumption Based Argumentation** introduce arguments as Assumptions $a \in \mathcal{A}$, or deduction σ with premises and conclusions achevied with Rules $R \subseteq \mathcal{R}$(deductive rules, as $q \wedge a \implies p$) defined in a language $\mathcal{L}$*

4.2.1 Simple ABA

*We can define **attacks** in ABA : an argument $A_1 \vdash \sigma_1$ attacks $A_2 \vdash \sigma_2$ iff σ_1 is the contrary of one of the assumptions in A_2.*

4.2.2 ABA more DDs

4.2.3 p-acyclic ABA

Definition 8
*a **positive-acyclic ABA** is a AB framework where the dependancy graph of AB is acyclic.*
*The **dependancy graph** is a graph of all the rules, where assumption ($\mathcal{A}$) are deleted.*

4.3 ArgGame

Chapter 5

Useful Computation

5.1 Data Centering using Matrix Multiplication

$$X - M = X\left(I_n - \frac{1}{n}\mathbf{1}_n\mathbf{1}_n^\top\right)$$