PROFILE TILDECRF: A NEW TOOL FOR PROTEIN HOMOLOGY DETECTION

Rômulo Barroso Victor

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientador: Gerson Zaverucha

Rio de Janeiro
Outubro de 2014

PROFILE TILDECRF: A NEW TOOL FOR PROTEIN HOMOLOGY DETECTION

Rômulo Barroso Victor

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Examinada por:

Prof. Gerson Zaverucha, Ph.D.

Prof. Valmir Carneiro Barbosa, Ph.D.

Prof. André Carlos Ponce de Leon Ferreira de Carvalho, Ph.D.

Prof. Juliana Silva Bernardes, D.Sc.

RIO DE JANEIRO, RJ – BRASIL
OUTUBRO DE 2014

Victor, Rômulo Barroso

Profile TildeCRF: a new tool for protein homology detection/Rômulo Barroso Victor. – Rio de Janeiro: UFRJ/COPPE, 2014.

XV, 97 p.: il.; 29, 7cm.

Orientador: Gerson Zaverucha

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2014.

Bibliography: p. 92 – 97.

1. Profile Hidden Markov Models. 2. Top-Down Induction of Logical Decision Trees. 3. Conditional Random Fields. I. Zaverucha, Gerson. II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

Eu dedico esse trabalho à minha esposa, Mayra.

Agradecimentos

Primeiramente, eu gostaria de agradecer a essa força que sustenta nossas vidas e que chamamos de Deus, mas que pode também se chamar Javé, Alá, Buda, Vishnu...

Agradeço à minha esposa, Mayra, por ter permanecido junto ao meu lado durante os momentos mais difíceis. Agradeço à minha mãe, Odenilza, e ao meu pai, Sergio, pelo seu amor e pela compreensão da minha ausência nos dias em que me dediquei a esse trabalho. Agradeço à minha avó, Nilza, pelas suas orações carinhosas e aos meus tios, Robson e Barbara, por suas palavras de coragem. Agradeço às minhas irmãs, Lívia e Iasmim, e à minha prima, Lara, pelo orgulho que sempre tiveram de mim. Agradeço à memória do meu avô Barroso, cuja lembrança é sentida a cada nova conquista e a cada nova etapa.

Agradeço de modo especial ao meu orientador, Prof. Gerson Zaverucha, pelo projeto desafiador, e à Profa. Juliana Silva Bernardes, a quem sempre considerei minha coorientadora, pelos seus ensinamentos sobre o tema e pela sua paciência com o desenvolvimento do meu raciocínio.

Agradeço aos meus antigos e atuais superiores por terem possibilitado a conciliação do presente trabalho com as minhas atividades empregatícias: Edna Rodrigues Campello, do IBGE, Valéria Ferreira da Silva e Verônica Maria Masferrer Schara, da PETROBRAS. Agradeço aos meus colegas petroleiros pelo apoio dado a mim no dia-a-dia da empresa.

Agradeço a dois pesquisadores que, mesmo sem me conhecerem pessoalmente, foram pessoas muito especiais ao demonstrarem grande prontidão em suas respostas aos meus e-mails. Eles são: Bernd Gutmann, que forneceu preciosas sugestões à minha implementação e à minha dissertação, sendo uma das razões pelas quais a redigi em língua inglesa; e Sean R. Eddy, cujos valiosos insights me abriram os olhos para os aspectos biológicos e probabilísticos do problema.

A todos vocês, muito obrigado!

Mathematical reasoning may be regarded rather schematically as the exercise of a combination of two facilities, which we may call intuition and ingenuity.

— Alan Turing

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

PROFILE TILDECRF: UMA NOVA FERRAMENTA PARA DETECÇÃO DE PROTEÍNAS HOMÓLOGAS

Rômulo Barroso Victor

Outubro/2014

Orientador: Gerson Zaverucha

Programa: Engenharia de Sistemas e Computação

A detecção de proteínas homólogas tem sido uma das questões mais importantes no campo da bioinformática. O *profile hidden Markov model* tem sido bem-sucedido na resolução desse problema, porém ele permanece um classificador do tipo generativo, o que é hipoteticamente menos preciso que modelos discriminativos, tais como *Conditional Random Fields*. Esforços para combinar profile HMM com CRF têm sido sugeridos na literatura, porém nenhuma implementação prática ou nenhum resultado experimental foram disponibilizados até então. A fim de testar a superioridade do CRF sobre o HMM na detecção de homologias de proteínas, nós implementamos o profile TildeCRF, uma nova ferramenta que combina o desenho do profile HMM com a abordagem discriminativa do CRF e com o poder de expressão das Árvores de Regressão Lógicas. Nós executamos experimentos do tipo *leave-one-family-out* em um conjunto de sequências de proteínas para mensurar a acurácia do método proposto e compará-lo ao profile HMM. Ao contrário das expectativas, os resultados mostraram que o profile TildeCRF é menos preciso que o profile HMM. De fato, o CRF é mais adequado para problemas de rotulagem de sequências do que para a deteção de homologias entre proteínas, porque ele mede a probabilidade de uma sequência de rótulos dada uma sequência de observações e não, a probabilidade de uma observação ter sido gerada pelo modelo treinado, como faz o HMM.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

PROFILE TILDECRF: A NEW TOOL FOR PROTEIN HOMOLOGY DETECTION

Rômulo Barroso Victor

October/2014

Advisor: Gerson Zaverucha

Department: Systems Engineering and Computer Science

Protein homology detection has been one of the major issues in the field of bioinformatics. Profile hidden Markov models have been successful in tackling this problem, but it remains a generative classifier, which is hypothetically less accurate than discriminative models like Conditional Random Fields. Efforts in combining profile HMM with CRF have been suggested in the literature, but no practical implementation or experimental results have been made available. In order to test the superiority of CRF over HMM in protein homology detection, we implemented profile TildeCRF, a new tool that combines the design of profile HMM with the discriminative approach of CRF and the expressiveness of Logical Regression Trees. We ran *leave-one-family-out* experiments on a dataset of protein sequences to measure the accuracy of the proposed method and compare it to profile HMM. Contrary to expectation, results showed that profile TildeCRF is less accurate than profile HMM. In fact, CRF is more suitable for sequence labeling tasks than for protein homology detection, because it measures the probability of a label sequence given the observation sequence, rather than the probability of the observation being generated by the trained model, as HMM does.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

It has been over 150 years since the outstanding scientific discoveries that laid the foundations for modern biology and genetics. Charles Darwin described the mechanism of natural selection, whereby individuals that are best adapted to their environment are more likely to survive and reproduce, passing their characteristics on to their offspring, while maladapted individuals tend to be eliminated [DARWIN, 1859]. Although Darwin already believed in the existence of a unit of inheritance [DARWIN, 1868], Gregory Mendel provided in parallel a logical explanation for the mechanism of heredity after his experiments on pea plant hybridization, which led him to formulate his laws of inheritance [MENDEL, 1866]. Nonetheless, it was not until 1953 that James D. Watson and Francis Crick first described the helical structure of *desoxyribose nucleic acid* (DNA), suggesting a possible copying mechanism of genetic material [WATSON and CRICK, 1953]. Since then, the interest in mapping the genomes of living organisms has grown significantly, as they hold the key to understanding how life is 'coded'. One example is the Human Genome Project (HGP), which was an international, collaborative research program whose goal was the complete mapping and understanding of all the genes of human beings [NHGRI, b]. Completed in 2003, the HGP revealed that there are probably 20,500 genes in the human genome, which is significantly fewer than previous estimates that ranged from 50,000 to 140,000 genes. In addition, the HGP demanded the implementation of tools that continue to characterize not only human genes, but also genes of several other organisms like mice, fruit-flies and flatworms. Since most organisms have similar, or *homologous*, genes with similar functions, the identification of a sequence or a function in a model organism, such as the roundworm *C. elegans*, may be used to explain a homologous gene in human beings. Hence comes the importance of DNA and protein *homology detection*.

There is a close connection between DNA and proteins: the DNA holds the 'recipe' for the ribosomes of a cell to produce proteins, which consist of sequences of elementary molecules called *amino acids*. Since proteins are translations of the DNA,

they reflect billions of years of life evolution and they may exist in different shapes with different functions inside a living organism [KIMBALL]. Therefore, proteins may also be organized in a genealogical tree with common ancestors and classified into families or superfamilies based on properties they share. There exist protein data repositories, such as PDB [BERMAN *et al.*, 2000], UniProt [MAGRANE and CONSORTIUM, 2011] and SCOPe [FOX *et al.*, 2013], which organize and annotate the protein structures, providing the biological community access to the experimental data in a useful way.

Typically, evaluating the similarity of different sequences involves finding a plausible alignment between them. However, the alignment of protein sequences that are coded by the DNA presents some advantages over the alignment of DNA sequences themselves [WERNERSSON and PEDERSEN, 2003]. First, protein alignment presents a better signal-to-noise ratio: while there are only 4 types of nucleotides within a DNA sequence, there can be up to 20 types of amino acids within a protein sequence, which makes it easier to distinguish real homology from random similarity. Second, owing to the redundancies of the genetic code, a silent mutation, which is a DNA mutation that does not correspond to a protein mutation, can be overrated by DNA alignment. Third, unlike for nucleotide data, the substitution matrices available for amino acid data (such as BLOSUM62 [HENIKOFF and HENIKOFF, 1992]) are empirically derived and they benefit from the fact that most amino acid replacements are conservative in terms of physico-chemical properties, which makes these matrices more biologically realistic. Finally, protein alignment may have a faster performance than DNA alignment, because DNA sequences are three times as long as the translated amino acid sequences [BININDA-EMONDS, 2005]. As a matter of fact, there is a three-to-one correspondence between nucleotides and amino acid residues during the translation from DNA to proteins.

In view of the significance of protein homology detection, there has been much effort to address this problem in technological fields. Simplistically, deciding that two biological sequences are similar is not very different from deciding that two text strings are similar, which is a usual problem addressed by computer applications [DURBIN *et al.*, 1998]. Thus, methods based on statistics and computer science have had a strong presence in biological sequence analysis. Most of these methods are based on dynamic programming and may perform DNA sequence alignment as well. They may be divided in two categories: the pairwise methods and the 'profile' methods.

Pairwise methods, such as BLAST [ALTSCHUL *et al.*, 1990] and FASTA [PEARSON and LIPMAN, 1988], perform alignment among a group of sequences, two by two. They implement the Needleman-Wunsh algorithm for global pairwise alignment [NEEDLEMAN and WUNSCH, 1970] or the Smith-Waterman algorithm for

local pairwise alignment [SMITH and WATERMAN, 1981].

Profile methods model the profile of a multiple sequence alignment, mostly based on the theory of *hidden Markov models* (or HMM), which is a class of probabilistic models that are generally applicable to time series or linear sequences. HMMs are widely applied to recognizing words in digitized sequences of the acoustics of human speech [RABINER, 1989]. A specific HMM architecture was designed for representing profiles of multiple sequence alignments: the *profile hidden Markov models* (profile HMM) [KROGH *et al.*, 1994]. The best-known algorithms based on profile HMMs are SAM [HUGHEY and KROGH, 1996] and HMMER [EDDY, 1998]. However, profile HMM remains a generative model, which is commonly believed to be less accurate than discriminative models [NG and JORDAN, 2001], because it makes strong independence assumptions among its inputs. In contrast, *Conditional Random Fields* (or CRFs) [LAFFERTY *et al.*, 2001] lack the need to model how their inputs are generated probabilistically, forming a discriminative-generative pair with HMMs [SUTTON and MCCALLUM, 2006]. The *profile* CRF, which adapts the architecture of profile HMM to the concepts of CRF, was proposed by KINJO [2009], but this proposal uses pre-defined biologically-based feature functions and it has neither any implementation nor any experimental results made available.

In order to test the hypothesis that CRF outperforms HMM, we present *profile* TildeCRF: a new tool for protein homology detection. Our method relies on a different version of CRF, called TildeCRF [GUTMANN and KERSTING, 2006], which replaces CRF's analytic feature functions with logical regression trees, that are created using the algorithm for Top-Down Induction of First-Order Logical Decision Trees (or Tilde) [BLOCKEEL and RAEDT, 1998]. TildeCRF was inspired by the algorithm TreeCRF [DIETTERICH *et al.*, 2004], which uses propositional regression trees as feature functions and train the CRF model via gradiant tree boosting. Regression trees in TreeCRF are induced using the classic algorithm C4.5 [QUINLAN, 1993].

This study is divided in 5 chapters: Introduction (Chapter 1), Background Knowledge (Chapter 2), Profile TildeCRF (Chapter 3), Experimental Results (Chapter 4), and Discussion and Conclusions (Chapter 5).

In Chapter 2, we present the basic knowledge to understand profile TildeCRF, which merges three main subjects: profile HMM, which provides an architecture that is specifically designed to represent the multiple alignment of a group of proteins; Tilde, which grows decision trees with the expressiveness of first-order logic; and CRF, which is a stochastic process with the advantages of the discriminative approach.

In Chapter 3, we propose our new method, profile TildeCRF, which also adapts profile HMM to CRF as done by KINJO [2009]. We also intend to give more

expressive power to the problem representation by using logical regression trees in CRF as in TildeCRF. Notwithstanding, we show that converting profile HMM into profile CRF is not as straightforward as we thought it would be initially. In fact, as profile HMM addresses sequence alignment and linear-chain CRF addresses sequence tagging, each model uses a different graph structure: profile HMM uses a cyclic finite state machine, whereas CRF uses a type of linear acyclic graph that is called *trellis*. This difficulty has made us resort to our own implementation of TildeCRF in Python so as to have more flexibility in terms of problem design, despite having more limitations in terms of unifying logic predicates.

In Chapter 4, we assess our method's performance, after training models based on the sequences of all families from a given superfamily, except for one. The family excluded from the training stage is later used as the positive set of sequences during the test stage. In our tests, we perform inference on positive and negative sequences in order to measure the accuracy of our algorithm. The results are compared to the ones returned by HMMER. We assume that a fair comparison between these algorithms implies that no prior knowledge should be used.

In Chapter 5, we discuss the results obtained from the experiments, the conclusions drawn by the results and our recommendations for future research.

In this study we define *target* as the sequence on which we perform an inference and *query* as the multiple alignment that is used to build the model. We also assume that *residues* and *amino acids* are synonyms and, unless explicitly stated otherwise, we assume CRF to be *linear-chain* CRF, which is its most common form.

Chapter 2

Background Knowledge

In this chapter, we provide a literature review for the basic understanding of profile TildeCRF. Section 2.1 makes a brief introduction to proteomics. Section 2.2 explains the basics of Markov models. Section 2.3 compares pairwise alignment to pair hidden Markov models. Section 2.4 presents the profile Hidden Markov Models, which is a specific HMM architecture suited for biological sequence alignment. Section 2.5 presents its discriminative dual: the Conditional Random Fields. Section 2.6 shows a different way of training CRFs using gradient tree boosting. Section 2.7 details the mathematical expressions used for evaluating statistical inference. Section 2.8 explains the top-down induction of logical decision trees. Lastly, Section 2.9 presents the area under the curve of receiver operating characteristic or precision-recall as a way of measuring the accuracy of a method.

2.1 Introduction to Proteomics

In this section, we will make a brief introduction to what proteins essentially are, how they are produced, how they have evolved and how different proteins can relate to each other.

2.1.1 What are proteins?

Proteins are macromolecules and they are responsible for several vital functions that have made life possible on Earth. For example, a well-known protein is the hemoglobin, which is present in our blood cells. Hemoglobin carries the oxygen from our lungs to our cells and then brings the carbonic gas produced by our cells back to our lungs. Other protein functions range from catalysis of biochemical reactions to locomotion of cells and organisms.

Proteins are chemically described as polymers and they are formed by one or more *polypeptide chains*. Each chain is a sequence of elementary molecules called

amino acids. There are 20 types of amino acids (Table 2.1) that may compose proteins and they have different features. Nevertheless, a protein is not defined merely by the linear sequence of amino acids that compose it. In fact, this simple perspective is called the *primary structure* of a protein. There are other three types of perspectives whereby proteins may be viewed and analyzed. Each perspective happens at a different scale and has a different level of complexity as it is illustrated in Figure 2.1.

The *secondary structure* is the the way amino acids are locally organized in the three-dimensional space along the polypeptide chain. The most common forms of organization are the *alpha helix* and the *beta conformation* [KIMBALL].

The *tertiary structure* is the three-dimensional shape of the entire polypeptide chain and the *quaternary structure* is the way two or more chains are connected spatially.

In this present work we will focus mainly on the primary structures of proteins, due to its relative simplicity and its usefulness in our statistical methods.

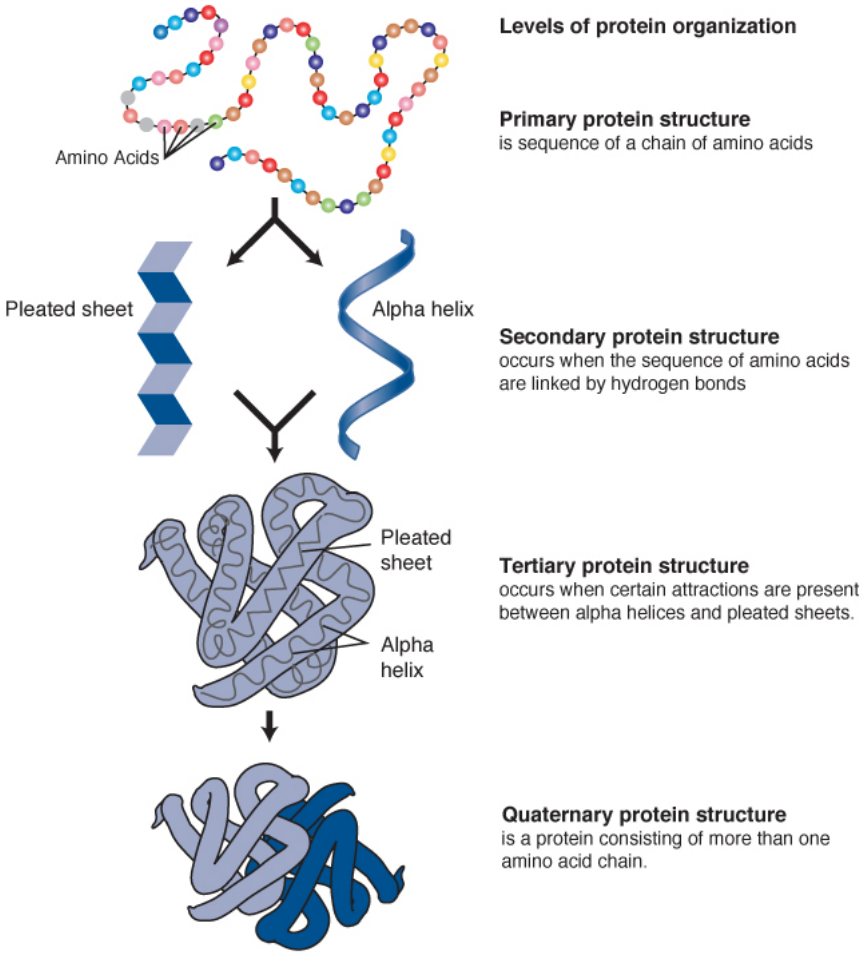


Figure 2.1: Protein levels of organization [NHGRI, a]

Alanine	Ala	A
Arginine	Arg	R
Asparagine	Asn	N
Aspartic acid	Asp	D
Cysteine	Cys	C
Glutamic acid	Glu	E
Glutamine	Gln	Q
Glycine	Gly	G
Histidine	His	H
Isoleucine	Ile	I

Leucine	Leu	L
Lysine	Lys	K
Methionine	Met	M
Phenylalanine	Phe	F
Proline	Pro	P
Serine	Ser	S
Threonine	Thr	T
Tryptophan	Trp	W
Tyrosine	Tyr	Y
Valine	Val	V

Table 2.1: List of the 20 amino acids (name, 3-letter code, 1-letter code)

2.1.2 Translation from DNA to proteins

One of the most important life components is the *deoxyribonucleic acid*, also known as *DNA*. Proteins and DNA are very closely related. One good analogy is that proteins are like cakes and DNA is like the recipe for those cakes. In a DNA structure *genes* contain instructions on what proteins to synthesize and on how the amino acids are ordered to form a sequence.

A gene is a physical unit of inheritance that is passed from parents to offspring and contains the necessary information to specify characteristics *collins*. Genes are made of elementary units called *nucleotides*, which may have four different bases, which are *adenine* (A), *guanine* (G), *thymine* (T) and *cytosine* (C).

The process whereby the information encoded in a gene is used so as to assemble a protein molecule is called *gene expression* [CANDOTTI] and it goes through two general steps: *DNA transcription* (Figure 2.2) and *RNA translation* (Figure 2.3).

DNA transcription takes place when a gene is transcripted into a special type of genetic material, the *messenger ribonucleic acid* or *mRNA*, that has similar bases for its nucleotides, except for *thymine* (T), which is replaced with *uracine* (U).

The mRNA acts a like a messenger and passes information from the cell nucleus to the *ribosomes*, which act like "protein factories" inside the cell structure. It is in the ribosomes where **RNA translation** takes place, with the mRNA code being translated into a sequence of amino acids. During translation, each amino acid is represented by a sequence of three nucleotides called *codon*.

Most amino acids are encoded by synonymous codons that differ in the third position [KIMBALL] (see Table 2.2). In special, the codon AUG may signal the start of translation or may insert Methionine (M) within the polypeptide chain, whereas the codons UAA, UAG, and UGA are STOP codons, indicating the end of translation [KIMBALL].

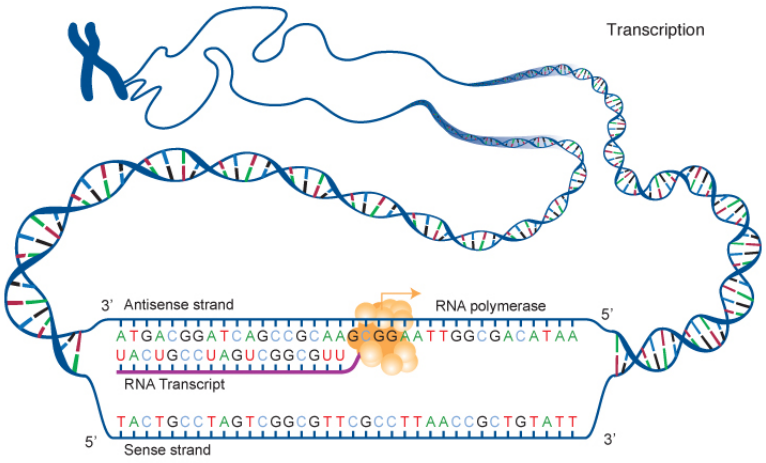


Figure 2.2: DNA transcription [NHGRI, a]

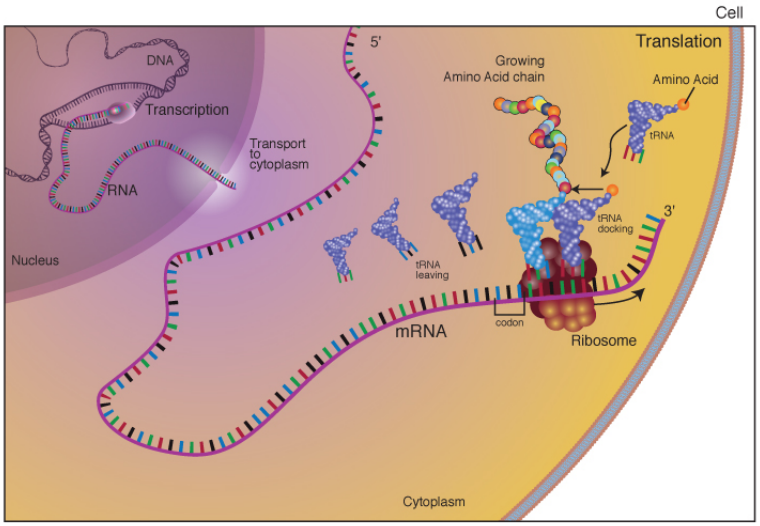


Figure 2.3: RNA translation [NHGRI, a]

2.1.3 Amino acid classification

It is possible to classify amino acids according to their chemical properties. Figure 2.4 shows the general structure of an amino acid, which is composed of a central α carbon atom that is attached to:

- a hydrogen atom;
- an amino group (hence "amino" acid);
- a carboxyl group (-COOH). This gives up a proton and is thus an acid (hence amino "acid");
- and one of 20 different "R" groups or side chains. It is the structure of the R

group that determines which of the 20 it is and its special properties [KIMBALL].

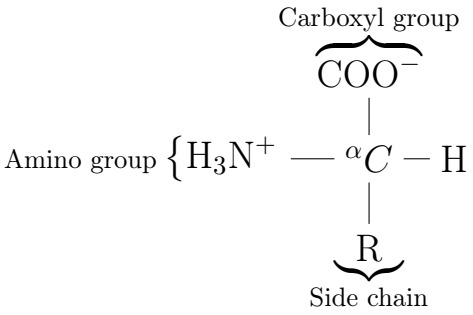


Figure 2.4: Amino acid general structure

Based on the R group's chemical composition, an amino acid can be classified according to its hydropathy (how soluble it is in water) and its acidity, as we can observe in Figure 2.5. Polar R groups tend to be more hydrophilic, whereas nonpolar R groups tend to be more hydrophobic. The type of polarity also influences the acidity: negatively-charged amino acids are acidic, while positively-charged ones are basic. Another classification criterion is the aromaticity. Amino acids are called aromatic, if they contain an aromatic ring[1], or aliphatic, otherwise.

	2nd nucleotide								
1st	U		C		A		G		3rd
U	UUU	Phe	UCU	Ser	UAU	Tyr	UGU	Cys	U
U	UUC	Phe	UCC	Ser	UAC	Tyr	UGC	Cys	C
U	UUA	Leu	UCA	Ser	UAA	STOP	UGA	STOP	A
U	UUG	Leu	UCG	Ser	UAG	STOP	UGG	Try	G
C	CUU	Leu	CCU	Pro	CAU	His	CGU	Arg	U
C	CUC	Leu	CCC	Pro	CAC	His	CGC	Arg	C
C	CUA	Leu	CCA	Pro	CAA	Glu	CGA	Arg	A
C	CUG	Leu	CCG	Pro	CAG	Gln	CGG	Arg	G
A	AUU	Iso	ACU	Thr	AAU	Asp	AGU	Ser	U
A	AUC	Ile	ACC	Thr	AAC	Asn	AGC	Ser	C
A	AUA	Ile	ACA	Thr	AAA	Lys	AGA	Arg	A
A	AUG	Met/START	ACG	Thr	AAG	Lys	AGG	Arg	G
G	GUU	Val	GCU	Ala	GAU	Asp	GGU	Gly	U
G	GUC	Val	GCC	Ala	GAC	Asp	GGC	Gly	C
G	GUA	Val	GCA	Ala	GAA	Glu	GGA	Gly	A
G	GUG	Val	GCG	Ala	GAG	Glu	GGG	Gly	G

Table 2.2: The RNA codons and its respective amino acids

[1] An aromatic ring is a six-membered carbon-hydrogen ring with three conjugated double bonds.

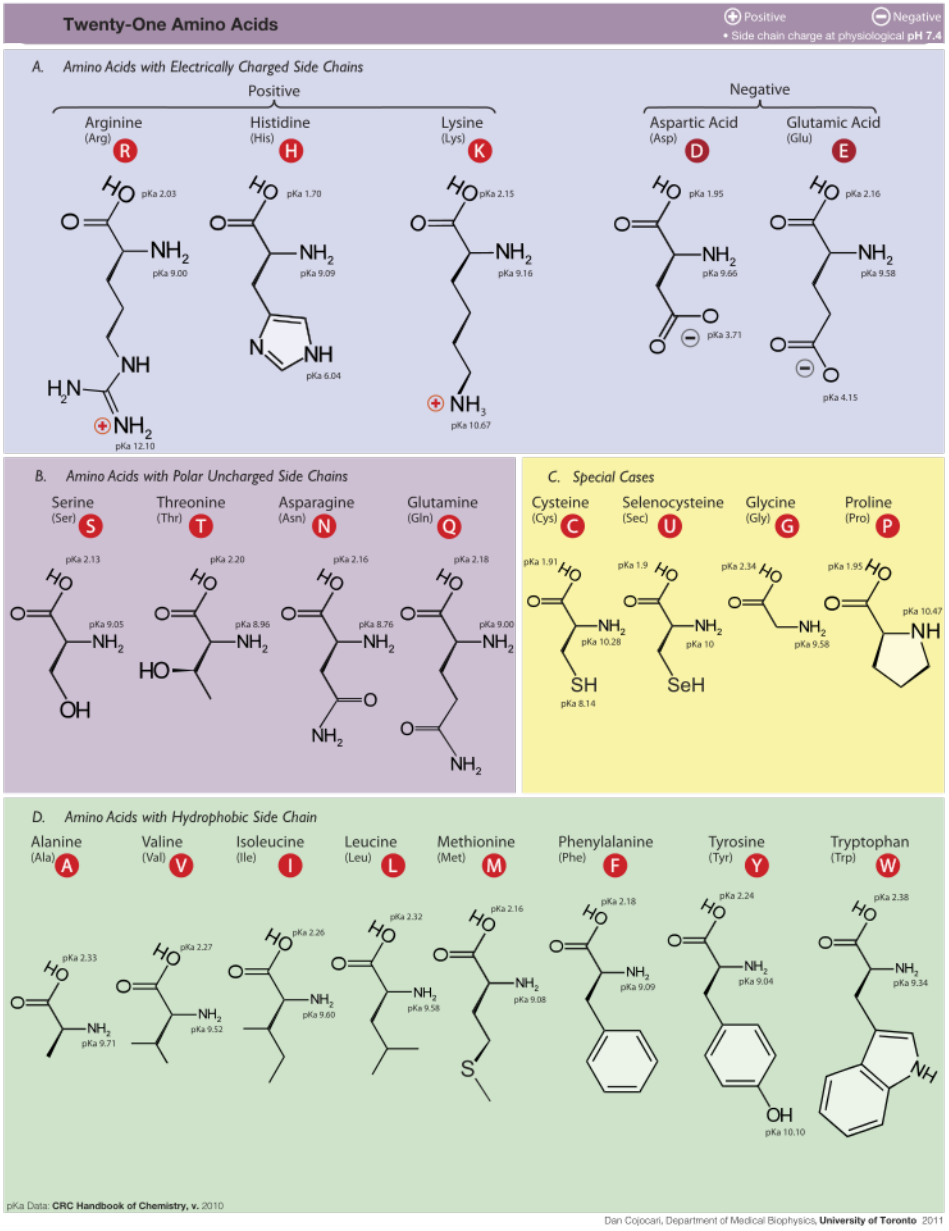


Figure 2.5: Classification of amino acids [COJOCARI]

2.1.4 Mutations

A mutation is a change in a DNA sequence and it can result from DNA copying errors made during cell division, exposure to radiation or mutagens, or infection by viruses [LEJA, 2010]. The majority of mutations can be harmful, affecting the possibility of life of the offspring, or silent, providing none or neutral characteristics for the individuals [KIMBALL]. The main types of mutations are *point mutations* (Figure 2.6) and *frameshift mutations* (Figure 2.7).

In **point mutations**, a single base within a gene is changed, causing: a *silent mutation*, where the mutated codon continues to encode the same amino acid; a

missense mutation, where the mutated codon encodes a different amino acid from the original one; or a *nonsense mutation* where the mutated codon is turned into a STOP codon and the gene expression is truncated.

In **frameshift mutations**, a single nucleotide is deleted or a new one is inserted in between two existing nucleotides. Because every amino acid is coded by sets of 3 nucleotides, a frameshift presents more serious consequences. As long as the number of insertions or deletions are not multiple by 3, the gene's nucleotides are no longer separated into the same groups of 3 as they were originally. Thus, a frame-shifted gene expression will assemble a completely different sequence of amino acids from the mutation point onward.

Briefly, changes in DNA are directly reflected in changes in protein synthesis. However, despite the harmfulness or silentness of most of these changes, mutations are the basic mechanism that made life evolution possible [KIMBALL].

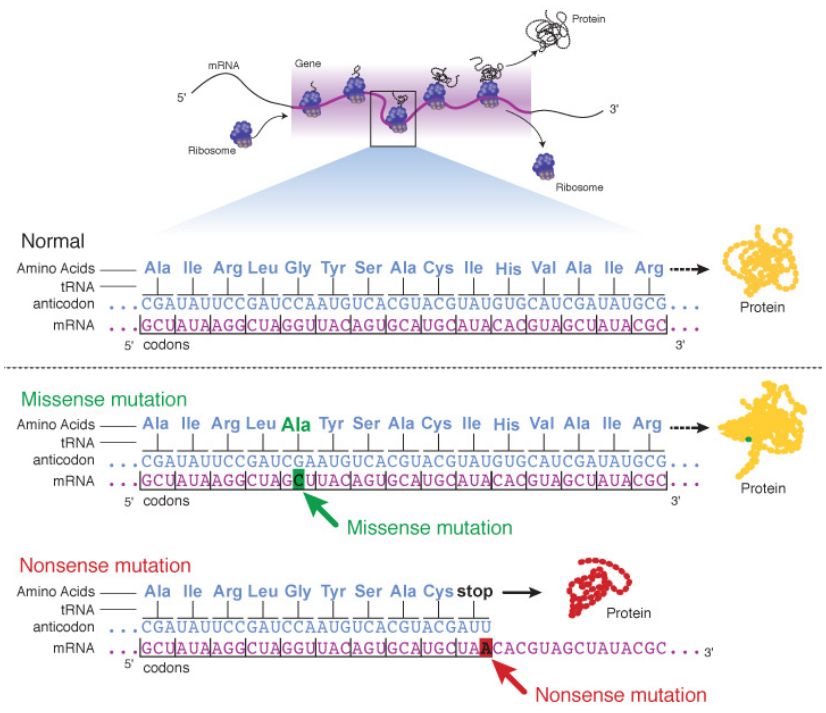


Figure 2.6: Point mutation [NHGRI, a]

2.1.5 Protein homology

Proteins that are related to one another are said to be *homologous* and they usually share similar three-dimensional structure and often execute identical or similar molecular functions [MARGELEVICIUS and VENCLOVAS, 2010]. In recognizing similarities between a well-known sequence and a new one, we can get information

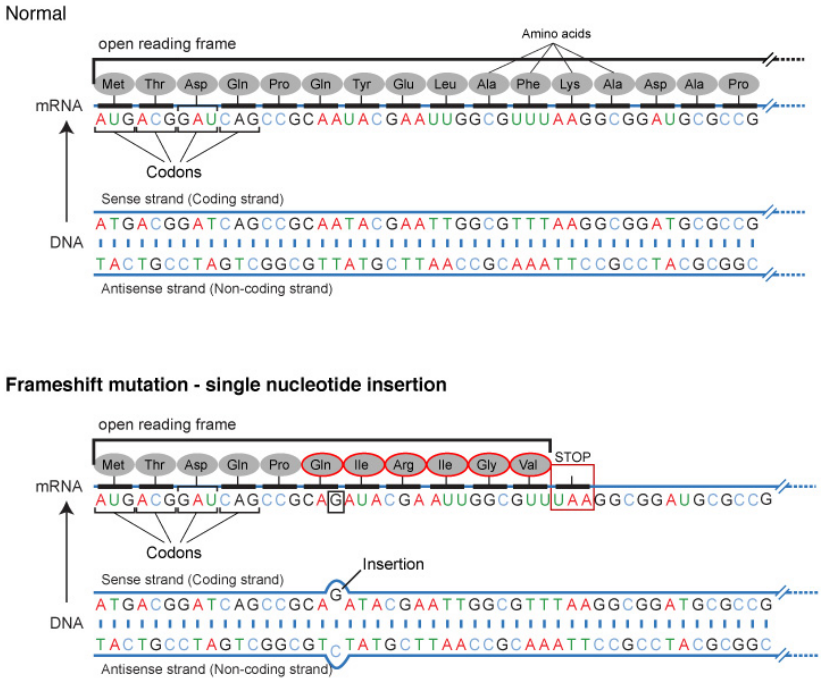


Figure 2.7: Frameshift mutation [NHGRI, a]

on structure, function and evolution of the new sequence. It is called inference *by homology* [DURBIN *et al.*, 1998].

The primary way of comparing two different proteins and establishing homology between them is by analyzing and aligning their primary structures. Genes that underwent insert or delete mutations may cause its translated polypeptide chain to have more or less residues (another name for the amino acids in this case) than its homologous sequences. When we unroll two different homologous sequences and put them side by side, we notice they do not have the same length. Nevertheless, aligning two sequences helps us analyze sequence similarities and identify points of deletion, insertion and match. An example of a two-sequence alignment is shown in Figure 2.8.

A point of deletion in a sequence is the position where an amino acid was supposed to be but has been deleted after mutations. A point of insertion is just the opposite: an amino acid was inserted where it was not supposed to be. A point of match is where the original amino acid has been maintained or has been replaced with another one.

One might ask how to align two different sequences with different residues and different lengths. In fact, there are statistical techniques that help us perform inferences about protein homology.

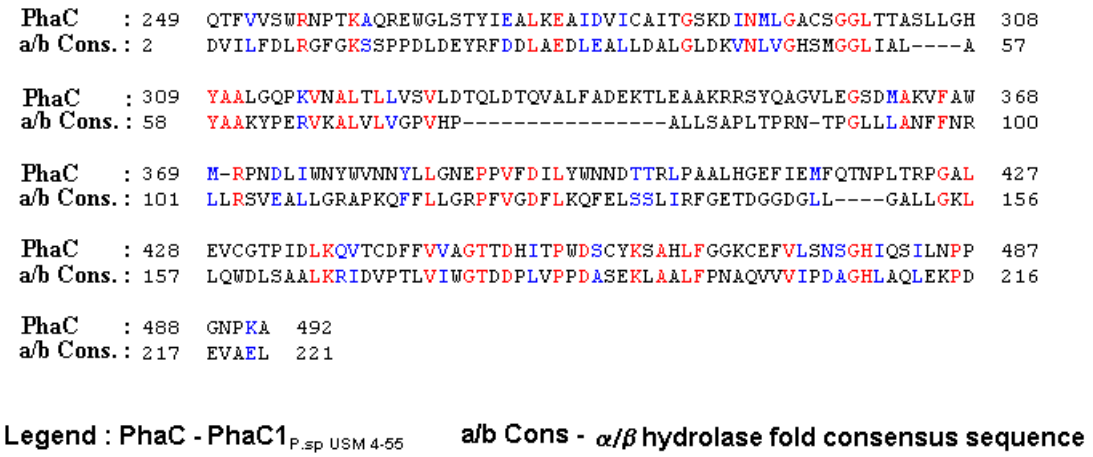


Figure 2.8: Two-sequence alignment example [WAHAB *et al.*, 2006]

Other forms of protein homology detection also include comparing the proteins' originary DNA codons [GIRDEA *et al.*, 2010] and comparing secondary structures GRONT *et al.* [2012]. Nevertheless, in this study we focus on primary structures mainly, as they are much simpler and they provide enough information to have significant results as previously shown by EDDY [1998].

2.1.6 Protein hierarchical classification

Proteins may be classified and organized in a hierarchical structure, according to their genealogy. The Structural Classification of Proteins (or SCOP [CONTE *et al.*, 2000]) database is a well-known protein database that has been constructed by visual inspection and comparison of structures, based on their similarities and properties. Proteins in SCOP is classified on hierarchical levels: families, superfamilies, folds and classes (see Figure 2.9).

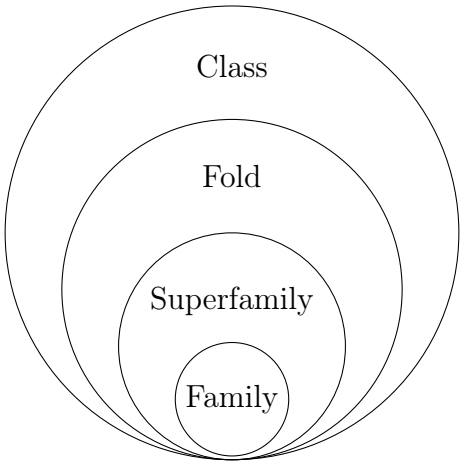


Figure 2.9: Hierarchical classification of proteins

Family

Proteins that share the same family have a high probability of having near evolutionary relationships. The classification of families in SCOP has to respect one of two conditions: first, all proteins must have residue identities of 30% or greater; or second, proteins must have very similar functions and structures; for example, globins with sequence identities of 15%.

Superfamily

A superfamily involves families whose proteins have low sequence identities but whose structures and functional features suggest that a distant evolutionary relationship; for example, the variable and constant domains of immunoglobulins.

Fold

A common fold involves superfamilies and families if their proteins have the same major secondary structures in the same arrangement and with the same topological connections.

Class

Most folds are assigned to one of the 5 following classes:

1. all-α, those whose structure is essentially formed by α-helices;
2. all-β, those whose structure is essentially formed by β-sheets;
3. α/β, those with α-helices and β-strands;
4. $\alpha+\beta$, those in which α-helices and β-strands are largely segregated;
5. multi-domain, those with domains of different fold and for which no homologues are known at present.

Other classes have been assigned for peptides, small proteins, theoretical models, nucleic acids and carbohydrates.

2.2 Fundamental Markov models

In this section, we introduce *Markov chains* and its enhanced version *hidden Markov models* (HMM) [RABINER, 1989]. They are both fundamental concepts for the object of our study.

2.2.1 Markov Chains

Markov chains are the starting point of our mathematical literature review. In fact, it is the foundation on which more complex methods stand, hence the importance of briefly introducing them here.

A *Markov chain* consists in a sequence of random variables that obeys the *Markov property*: the probability of future states depends only on the present and does not depend on the past. It is thus called in honor of Russian Mathematician Andrey Markov (1856-1922).

Consider a sequence of random variables $\mathbf{Y} = \{Y_0, Y_1, Y_2, \ldots, Y_{t+1}\}$, a sequence of states $\mathbf{y} = \{y_0, y_1, y_2, \ldots, y_{t+1}\}$ and a given instant t. The Markov property is thus expressed in the equation below:

$$P(Y_{t+1} = y_{t+1}|Y_t = y_t, y_{t-1} = y_{t-1}, ..., Y_0 = y_0) = P(Y_{t+1} = y_{t+1}|Y_t = y_t) \quad (2.1)$$

The Markov property is satisfied if the conditional probability of the future state given all previous states is equal to the conditional probability of the future state given the present state only. This must be true for every instant $t \geq 0$ and for every sequence $\mathbf{y} = \{y_0, y_1, y_2, \ldots, y_{t+1}\}$. In this case, the sequence of random variables $\mathbf{Y}$ is a *Markov chain* [OCONE, 2009].

For calculating the probability of having a sequence of states $\mathbf{y} = \{y_0, y_1, \ldots, y_t\}$, we must proceed as follows:

$$\begin{aligned}
&P(\mathbf{Y} = \mathbf{y}) = P(Y_t = y_t, \ldots, Y_0 = y_0) \\
&= P(Y_t = y_t|Y_{t-1} = y_{t-1}, \ldots, Y_0 = y_0) \times P(Y_{t-1} = y_{t-1}, \ldots, Y_0 = y_0) \\
&= P(Y_t = y_t|Y_{t-1} = y_{t-1}) \\
&\quad \times P(Y_{t-1} = y_{t-1}|Y_{t-2} = y_{t-2}, \ldots, Y_0 = y_0) \times P(Y_{t-2} = y_{t-2}, \ldots, Y_0 = y_0) \\
&= P(Y_t = y_t|Y_{t-1} = y_{t-1}) \\
&\quad \times P(Y_{t-1} = y_{t-1}|Y_{t-2} = y_{t-2}) \times \cdots \times P(Y_1 = y_1|Y_0 = y_0) \times P(Y_0 = y_0)
\end{aligned} \quad (2.2)$$

In Equation 2.2, we use Bayes' Theorem to decompose the joint probability: $P(A, B) = P(A|B)P(B)$. We simplify the conditional probability by applying the Markov property to $P(A|B)$. Then, we repeat the decomposing and simplifying procedure for $P(B)$.

Markov chains can be graphically represented by a *finite state machine* (or FSM), which is a graph whose nodes correspond to the states and whose edges correspond to the transitions between one state and another. The real number next to an edge consists in the probability of the transition to the destination state given the

origin state. In Figure 2.10 there is an example of a Markov chain that models a hypothetical weather forecast.

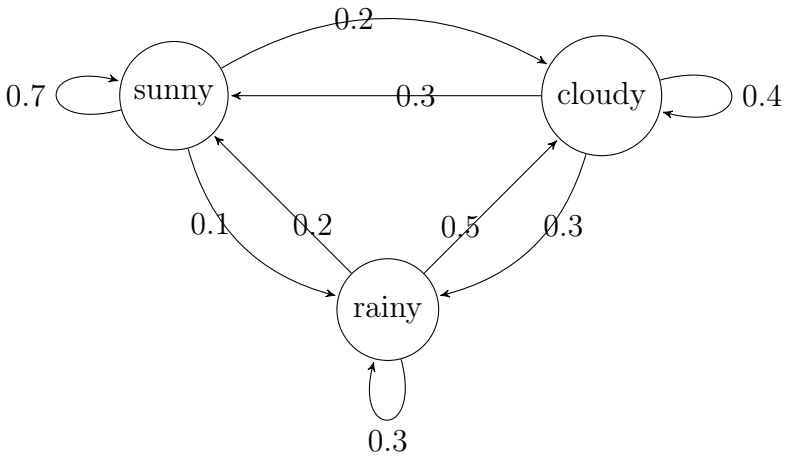


Figure 2.10: Finite state machine of a Markov chain for a hypothetical weather forecast

Markov chains have a particularly simple form for modeling one-step-ahead state probabilities along a period of time. When we add another set of random variables to represent the *observations*, we obtain an improved version of Markov chain, as we will see next.

2.2.2 Hidden Markov models

In Markov chains, there are transition probabilities from one state to another, but there is not the concept of *observation*. An observation is the expression of a given state that is captured by the observer. In Markov chains states and observations are related one-to-one, while in HMM each state has its own probability distribution as a function of the observation.

To illustrate this, let us take a look at the example shown in Figure 2.11. It represents the Markov chain for a baby's actions or expressions. There are three possible states: smiling, sleeping and crying. In each of these, the information received by our eyes leaves no doubt what state we are actually in. For instance, if we see and hear the baby crying, there is one hundred percent certainty that he is actually crying.

Hidden Markov models [RABINER, 1989] are different from Markov chains in one aspect: states and observations are decoupled. It means that in the HMM the observation we capture or the information we receive does not give absolute certainty of what state we are actually in.

Let us take the baby example from above. Consider that the states we used before are now our observations ('smiling', 'crying' and 'sleeping') and the new states are

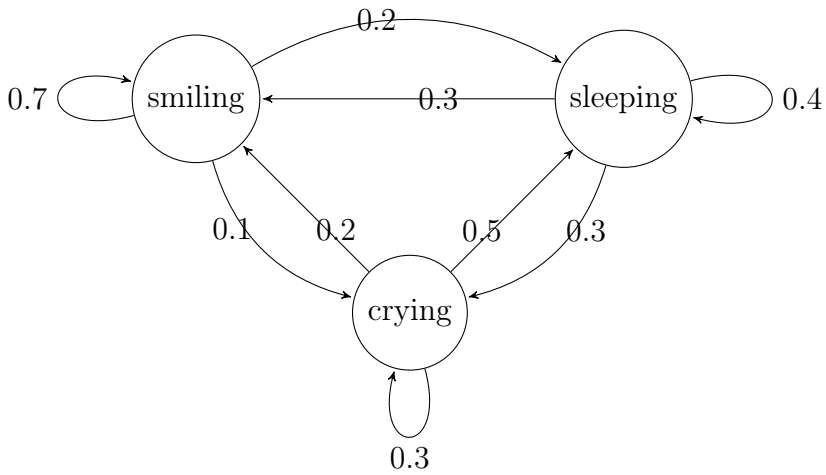


Figure 2.11: Finite state machine of a Markov chain for baby's behavior

related to the baby's feelings ('well', 'ill', 'hungry') as we see in Figure 2.12.

If the baby is seen sleeping, it is not possible to say for sure if he is well or hungry or ill. The same applies to when the baby is crying and many parents have difficulties to know (or guess) what is causing their child to weep. Of course, there is a reasonable way of measuring how probable it is to have a certain observation given the actual state. If the baby is smiling, it is more probable that he is well than ill or hungry.

Mathematically speaking, state-observation decoupling results in a new set of random variables, which is that of observations perceived along time. Consider two sets of random variables: the random variables of states $\mathbf{Y} = \{Y_0, Y_1, \ldots, Y_t\}$ and the random variables of observations $\mathbf{X} = \{X_0, X_1, \ldots, X_t\}$. In this model, there is another independence assumption, according to which every observation depends on the present state only and does not depend on previous states or previous observations. For the HMM, not only is it important to calculate the conditional probability of the current state given the previous one, but also the probability of the current observation given the current state, which is now uncertain.

In order to calculate the joint probability of having a sequence of observations $\mathbf{x} = \{x_0, x_1, \ldots, x_t\}$ and a sequence of states $\mathbf{y} = \{y_0, y_1, \ldots, y_t\}$, where x_t is observed in y_t, we have:

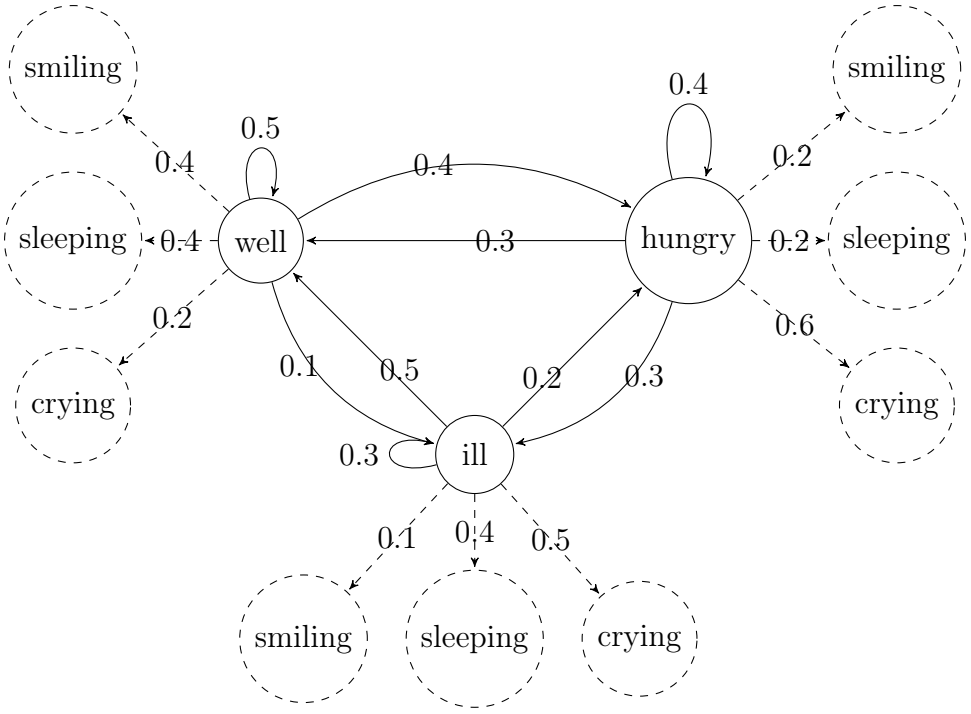


Figure 2.12: Finite state machine of HMM for baby's behavior

$$
\begin{aligned}
&P(\mathbf{Y}=\mathbf{y},\mathbf{X}=\mathbf{x}) = P(Y_t=y_t,\ldots,Y_0=y_0,X_t=x_t,\ldots,X_0=x_0)\\
&=P(X_t=x_t|Y_t=y_t,\ldots,Y_0=y_0,X_{t-1}=x_{t-1},\ldots,X_0=x_0)\\
&\quad\times P(Y_t=y_t,\ldots,Y_0=y_0,X_{t-1}=x_{t-1},\ldots,X_0=x_0)\\
&=P(X_t=x_t|Y_t=y_t)\\
&\quad\times P(Y_t=y_t|Y_{t-1}=y_{t-1},\ldots,Y_0=y_0,X_{t-1}=x_{t-1},\ldots,X_0=x_0)\\
&\quad\times P(Y_{t-1}=y_{t-1},\ldots,Y_0=y_0,X_{t-1}=x_{t-1},\ldots,X_0=x_0)\\
&=P(X_t=x_t|Y_t=y_t)\times P(Y_t=y_t|Y_{t-1}=y_{t-1})\\
&\quad\times P(X_{t-1}=x_{t-1}|Y_{t-1}=y_{t-1})\times P(Y_{t-1}=y_{t-1}|Y_{t-2}=y_{t-2})\\
&\quad\times\cdots\times P(X_0=x_0|Y_0=y_0)\times P(Y_0=y_0)
\end{aligned}
\tag{2.3}
$$

In Equation 2.3, we use Bayes' Theorem to decompose the joint probability: $P(A,B) = P(A|B)P(B)$. Firstly, we separate the observation variable X_t (term A) from the other variables (term B). We simplify the conditional probability by applying the Markov property to $P(A|B)$ such that the observation variable only depends on the state it is emitted by, which results in the *emission probability* $P(X_t = x_t|Y_t = y_t)$. Secondly, we reuse Bayes' Theorem to separate the state variable Y_t (new term A) from the other variables (new term B). Once again, the conditional probability $P(A|B)$ is simplified by the Markov property such that the current state only depends on the previous state, resulting in the *transition prob-*

ability $P(Y_t = y_t|Y_{t-1} = y_{t-1})$. This procedure is then repeated and new emission and transition probabilities are defined until we reach the variable Y_0.

This equation is of great importance for developing the concepts in *pair hidden Markov models* and *profile hidden Markov models*.

2.3 Pairwise alignment and pair hidden Markov models

Pairwise alignment is a method for aligning two non-identical sequences by matching or shifting the amino acids of one sequence in relation to the amino acids of the other sequence.

In this section, we will discuss *dynamic programming*, which is a imperative tool in order to achieve Pairwise Alignment. Secondly, we will see how Pairwise Alignment can be applied in the problem of aligning two protein sequences. Lastly, we will present the *Pair HMM*, which is an evolution of pairwise alignment that captures probabilistic concepts from hidden Markov Models.

2.3.1 Dynamic Programming

Dynamic programming is a fundamental tool for pairwise alignment. A problem in dynamic programming can only be solved if the answers to its subproblems are previously known [DASGUPTA *et al.*, 2006]. Dynamic programming problems have an underlying *directed acyclic graph* (DAG) structure. Let us take the spell checker example to show how this method works. When the user mistypes a word, the spell checker compares the misspelled word with the lexicon located in its internal database. Then, the words from the database that best match the mistyped word are chosen and recommended to the user. In this case, the suggestions are the words that present the best pairwise alignment with the target.

For instance, let us say the user mistook the word *umbrella* for *umvrrwla*. One possible alignment between these two words might be:

1	2	3	4	5	6	7	8	9
U	M	B	R	E	L	-	L	A
U	M	V	R	-	R	W	L	A

The score of the alignment above is 5, because both aligned sequences match in columns #1, #2, #4, #8 and #9. In order to verify if a pairwise alignment is optimal, a two-dimensional shaped DAG structure must be designed as shown in Figure 2.13.

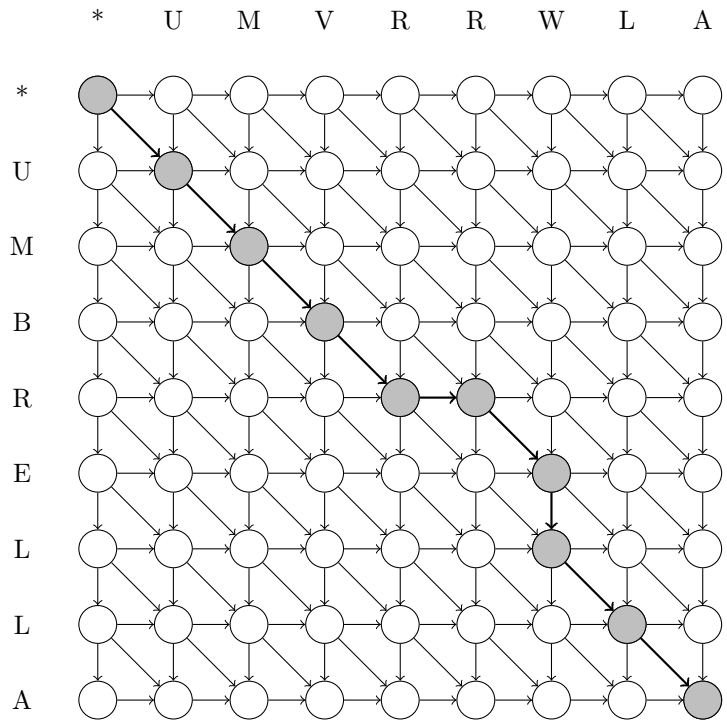
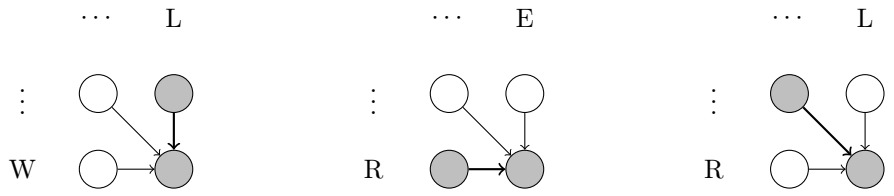


Figure 2.13: DAG structure for spell checker

An important feature of DAG representation is how we visualize the possible alignments between two sequences: every path that can be traced from the first node to the last represents a distinct, feasible way of aligning. Consequently, the optimal alignment is represented by a specific path in this graph.

Regarding each step isolatedly, we observe three different moves: vertical (Figure 2.14a), horizontal (Figure 2.14b) and diagonal (Figure 2.14c). A **vertical** move means that the mistyped sequence does not have a match with the correct sequence, and for this reason it receives a gap symbol "-". A **horizontal** move means the opposite: the correct sequence does not have a match with the mistyped sequence. A **diagonal** move means that both symbols are inserted and they can be a match, if they are equal, or a substitution, otherwise.



(a) Vertical move (-, W) (b) Horizontal move (E, -) (c) Diagonal move (L, R)

Figure 2.14: Types of moves with corresponding aligned pair (x, y), where x is the symbol emitted by the correct sequence and y is the one emitted by the wrong sequence.

This graph can also be viewed as a matrix, where each node is indexed by the

pair (i, j) with i being the column index and j being the row index. Let $\mathcal{M}_{i,j}$ be the submatrix that ranges from node (0,0) to node (i, j). Therefore, the subproblems for submatrix $\mathcal{M}_{i,j}$ are submatrices $\mathcal{M}_{i-1,j}$, $\mathcal{M}_{i-1,j-1}$ and $\mathcal{M}_{i,j-1}$, corresponding to the three types of moves.

In addition to designing a directed acyclic graph, finding the optimal pairwise alignment requires two procedures. The first one (Algorithm 1) will calculate the alignment scores for every submatrix $\mathcal{M}_{i,j}$. Once these scores are calculated, the second procedure (Algorithm 2) will backtrack the optimal path from the last node up to the first node of this graph.

Let x be the array of characters of the mistyped word and y to be the array of characters of the correct word. Let i and j be the index cursors for x and y, respectively. Let M be the table where the scores of subproblems are stored. That means that $M[i, j]$ will store the result of the subproblem related to submatrix $\mathcal{M}_{i,j}$. The algorithms 1 and 2 show how to calculate scores for all subproblems.

Algorithm 1 Calculate scores for all subproblems

```
1: Initialize M and θ
2: M[0,0] ← 0
3: x ← CONCATENATE('#', wrongWord)
4: y ← CONCATENATE('#', rightWord)
5: for i ← 0 to n do
6:     for j ← 0 to m do
7:         CALCULATESCORE(M, x, y, i, j)
8:
9: procedure CALCULATESCORE(M, x, y, i, j)
10:    if i < 0 or j < 0 or (i = 0 and j = 0) then
11:        return 0
12:    else
13:        if M[i, j] is not calculated then
14:            if x[i] = y[j] then                    ▷ letters are equal
15:                score ← 1
16:            else                                   ▷ letters are different
17:                score ← 0
18:            matchScore ← score + CALCULATESCORE(M, x, y, i − 1, j − 1)
19:            xInsertScore ← 0 + CALCULATESCORE(M, x, y, i − 1, j)
20:            yInsertScore ← 0 + CALCULATESCORE(M, x, y, i, j − 1)
21:            M[i, j] ← MAX(xInsertScore, yInsertScore, matchScore)
22:            θ[i, j] ← ARGMAX(matchScore, xInsertScore, yInsertScore)
23:                ▷ Argmax returns 1 for match, 2 for xInsert and 3 for yInsert
```

Algorithm 2 Backtrack best alignment procedure

```
1: procedure GetBestAlignment(M, θ, x, y)
2:     i ← n                                              ▷ Initialization
3:     j ← m
4:     alignedX ← ""
5:     alignedY ← ""
6:     while i > 0 and j > 0 do
7:         maxArg ← θ[i, j]
8:         if maxArg = 1 then                  ▷ matchScore is max, diagonal move
9:             alignedX ← Concatenate(x[i], alignedX)
10:            alignedY ← Concatenate(y[j], alignedY)
11:            i ← i − 1
12:            j ← j − 1
13:        else if maxArg = 2 then          ▷ xInsertScore is max, horizontal move
14:            alignedX ← Concatenate(x[i], alignedX)
15:            alignedY ← Concatenate('−', alignedY)
16:            i ← i − 1
17:            j ← j
18:        else if maxArg = 3 then          ▷ yInsertScore is max, vertical move
19:            alignedX ← Concatenate('−', alignedX)
20:            alignedY ← Concatenate(y[j], alignedY)
21:            i ← i
22:            j ← j − 1
23:    return alignedX, alignedY
```

2.3.2 Pairwise alignment for amino acid sequences

The spell checker example is a good analogy for the alignment between two sequences of amino acids. The fitness score used in the former is the identity between two letters, whereas the fitness score that is used in the latter is the similarity between two amino acids. The dynamic programming algorithm for the pairwise alignment of protein sequences is known as the *Needleman-Wunsch* algorithm [NEEDLEMAN and WUNSCH, 1970].

For means of protein comparison, two amino acids do not need to be exactly equal for the score to be positive. There is a pre-defined score between every two residues, which is proportional to the likeliness of one being replaced by the other. If we look back to the spell checker, it is as though the distance between the keys of your keyboard mattered when calculating the score: it is more probable to mistype a letter by accidentally pressing one of the neighbor keys.

There is a number of pre-calculated tables that contain the substitution scores between two given amino acids. They are called *Substitution Matrices*. One of the best known matrices is the BLOSUM (which stands for *BLOcks SUbstitution Matrix*) [HENIKOFF and HENIKOFF, 1992]. BLOSUM62 is a version of this

matrix where the similarities are calculated using clusters of sequences that are at least 62% identical.

In order to calculate the scores in BLOSUM matrix, let us consider two residues a and b. Let q_a and q_b be the background probabilities of a and b, that is, the probabilities of these residues occurring at random. We may consider that p_{ab} is the probability of residues a and b being derived from a common ancestor c (which might be the same as a or b or neither) [DURBIN *et al.*, 1998]. The formula used to calculate each one of the scores is as follows:

$$s(a,b) = \left(\frac{1}{\lambda}\right) \log \frac{p_{ab}}{q_a q_b} \tag{2.4}$$

The value $\log \frac{p_{ab}}{q_a q_b}$ is the log probability that the pair (a,b) occurs together as an aligned pair. The denominator λ is a constant that converts the log probability into an easily workable integer. The score $s(a,b)$ is obtained for each possible pair of amino acids so that the BLOSUM matrix is created (see BLOSUM62 matrix in Table 2.3).

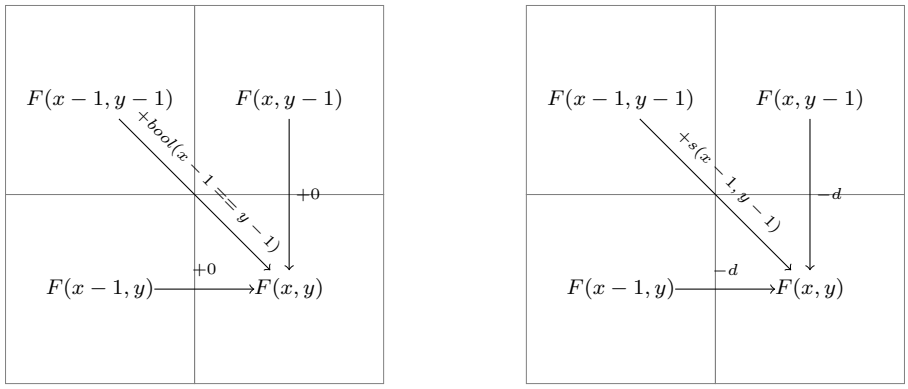
All things considered, if we replace the scoring criteria in the Algorithm 1 with the BLOSUM62 substitution matrix, we have an algorithm adapted for finding optimal alignments between two protein sequences (Algorithm 3). The Algorithm 2, which backtracks the optimal path in the graph and gets the best alignment, remains unchanged. Adjusting the scoring criteria requires two modifications (see Figure 2.15): the 0-or-1 score is replaced by the values in the BLOSUM matrix given the residues in $x[i]$ and $y[j]$; and there is a gap penalty $-d$ instead of a null score in case of insertions from either sequence.

	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V	B	Z	X	-
A	4	-1	2	-2	0	-1	-1	0	-2	-1	-1	-1	-1	-2	-1	1	0	-3	-2	0	-2	-1	0	-4
R	-1	5	0	-2	-3	1	0	-2	0	-3	-2	2	-1	-3	-2	-1	-1	-3	-2	-3	-1	0	-1	-4
N	-2	0	6	1	-3	0	0	0	1	-3	-3	0	-2	-3	-2	1	0	-4	-2	-3	3	0	-1	-4
D	-2	-2	1	6	-3	0	2	-1	-1	-3	-4	-1	-3	-3	-1	0	-1	-4	-3	-3	4	1	-1	-4
C	0	-3	3	-3	9	-3	-4	-3	-3	-1	-1	-3	-1	-2	-3	-1	-1	-2	-2	-1	-3	-3	-2	-4
Q	-1	1	0	0	-3	5	2	-2	0	-3	-2	1	0	-3	-1	0	-1	-2	-1	-2	0	3	-1	-4
E	-1	0	0	2	-4	2	5	-2	0	-3	-3	1	-2	-3	-1	0	-1	-3	-2	-2	1	4	-1	-4
G	0	-2	0	-1	-3	-2	-2	6	-2	-4	-4	-2	-3	-3	-2	0	-2	-2	-3	-3	-1	-2	-1	-4
H	-2	0	1	-1	-3	0	0	-2	8	-3	-3	-1	-2	-1	-2	-1	-2	-2	2	-3	0	0	-1	-4
I	-1	-3	3	-3	-1	-3	-3	-4	-3	4	2	-3	1	0	-3	-2	-1	-3	-1	3	-3	-3	-1	-4
L	-1	-2	3	-4	-1	-2	-3	-4	-3	2	4	-2	2	0	-3	-2	-1	-2	-1	1	-4	-3	-1	-4
K	-1	2	0	-1	-3	1	1	-2	-1	-3	-2	5	-1	-3	-1	0	-1	-3	-2	-2	0	1	-1	-4
M	-1	-1	2	-3	-1	0	-2	-3	-2	1	2	-1	5	0	-2	-1	-1	-1	-1	1	-3	-1	-1	-4
F	-2	-3	3	-3	-2	-3	-3	-3	-1	0	0	-3	0	6	-4	-2	-2	1	3	-1	-3	-3	-1	-4
P	-1	-2	2	-1	-3	-1	-1	-2	-2	-3	-3	-1	-2	-4	7	-1	-1	-4	-3	-2	-2	-1	-2	-4
S	1	-1	1	0	-1	0	0	0	-1	-2	-2	0	-1	-2	-1	4	1	-3	-2	-2	0	0	0	-4
T	0	-1	0	-1	-1	-1	-1	-2	-2	-1	-1	-1	-1	-2	-1	1	5	-2	-2	0	-1	-1	0	-4
W	-3	-3	4	-4	-2	-2	-3	-2	-2	-3	-2	-3	-1	1	-4	-3	-2	11	2	-3	-4	-3	-2	-4
Y	-2	-2	2	-3	-2	-1	-2	-3	2	-1	-1	-2	-1	3	-3	-2	-2	2	7	-1	-3	-2	-1	-4
V	0	-3	3	-3	-1	-2	-2	-3	-3	3	1	-2	1	-1	-2	-2	0	-3	-1	4	-3	-2	-1	-4
B	-2	-1	3	4	-3	0	1	-1	0	-3	-4	0	-3	-3	-2	0	-1	-4	-3	-3	4	1	-1	-4
Z	-1	0	0	1	-3	3	4	-2	0	-3	-3	1	-1	-3	-1	0	-1	-3	-2	-2	1	4	-1	-4
X	0	-1	1	-1	-2	-1	-1	-1	-1	-1	-1	-1	-1	-1	-2	0	0	-2	-1	-1	-1	-1	-1	-4
-	-4	-4	4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	1

Table 2.3: BLOSUM62 table

Algorithm 3 Procedure that calculate scores for protein pairwise alignment

```
1: procedure CalculateScore(M, x, y, i, j)
2:     if i < 0 or j < 0 or (i = 0 and j = 0) then
3:         return 0
4:     else
5:         if M[i, j] is not calculated then
6:             score ← s(x[i], x[j])
7:             matchScore ← score + CalculateScore(M, x, y, i − 1, j − 1)
8:             xInsertScore ← −d + CalculateScore(M, x, y, i − 1, j)
9:             yInsertScore ← −d + CalculateScore(M, x, y, i, j − 1)
10:            M[i, j] ← Max(xInsertScore, yInsertScore, matchScore)
11:            θ[i, j] ← Argmax(matchScore, xInsertScore, yInsertScore)
12:                ▷ Argmax returns 1 for match, 2 for xInsert and 3 for yInsert
```



(a) Spell checker recursion step (b) Pairwise alignment recursion step

Figure 2.15: Recursion step of pairwise alignment for the spell checker 2.15a and for protein sequence comparison 2.15b

2.3.3 From Pairwise Alignment to Pair HMM

It is possible to represent the dynamics of pairwise alignment in a finite state machine that contains four states: *begin*, *match*, *X insert*, *Y insert* and *end* (see Figure 2.16). Each move made in the DAG structure corresponds to a transition between states in this FSM. Furthermore, each transition corresponds to a different addition to the total score and each state corresponds to an increment of the position index in sequence X or sequence Y or both.

This kind of representation makes it easier to convert pairwise alignment into a hidden Markov Model, that is built on a very similar finite state machine (see Figure 2.17). The main difference in the HMM is the probability values assigned to each state and transition, instead of the score values we have used so far (see Figure 2.17). The HMM designed for pairwise alignment is called *Pair HMM* [DURBIN *et al.*, 1998].

Recalling section 2.2.2, let us consider a set of states and a set of observations. Each state has a probability distribution for emitting a given observation. In Pair

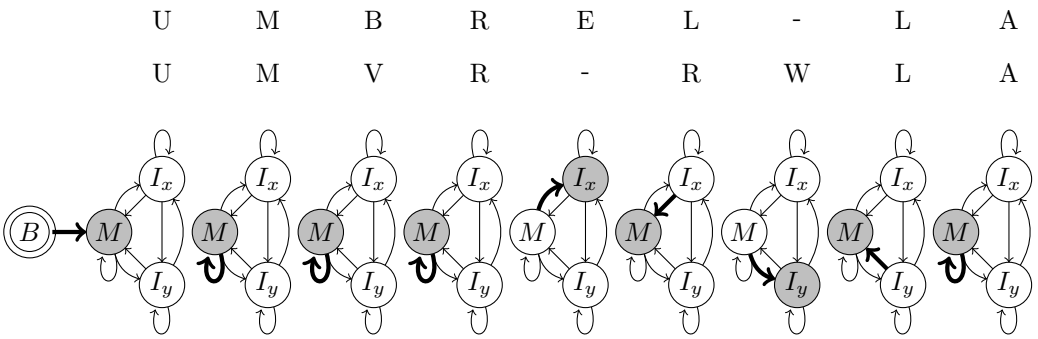
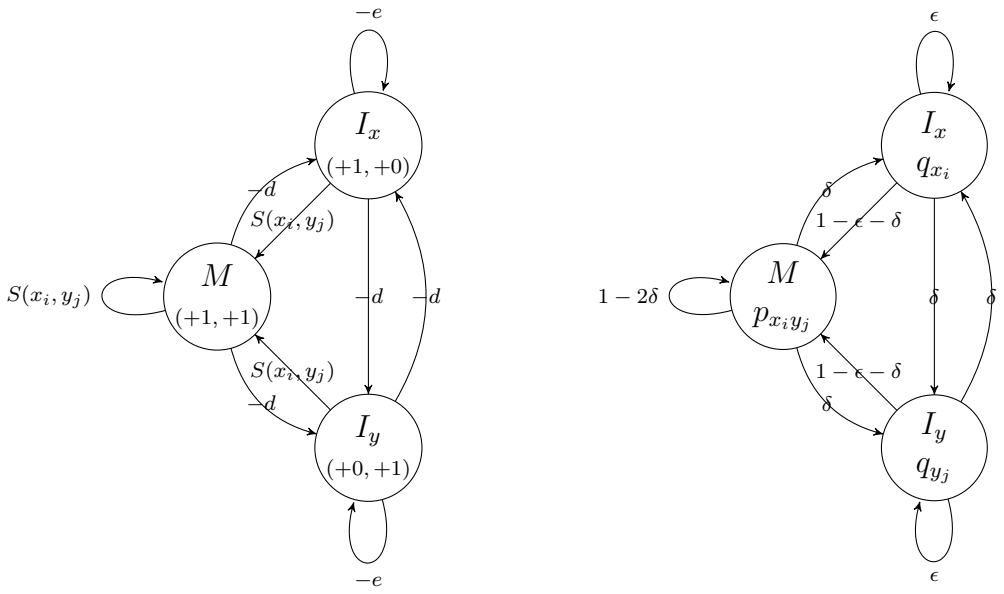


Figure 2.16: Pairwise alignment represented by finite state machine

HMM, the states are those we have seen in the finite state machine (match, x insert, y insert), whereas the set of observations consists of the possible pairs of two amino acids, one from each sequence (x_i, y_j), in case of match state, or the possible pairs of one amino acid and one gap $((x_i, -)$ or $(-, y_j))$, in case of x insert state or y insert state.



(a) FSM representing pairwise alignment

(b) FSM representing Pair HMM

Figure 2.17: Converting pairwise alignment into Pair HMM: probability values in transitions and states are used instead of pre-defined scores.

Note that the pair (x_i, y_j) is composed of the i-th amino acid of sequence $\mathbf{x}$ and the j-th amino acid of sequence $\mathbf{x}$. In a match state, the pair x_i, y_j is assumed to be a match and it is assigned a probability value $p_{x_i y_j}$. In an X-insert state, the assumption is that there is no match between x_i and y_j and the pair $(x_i, -)$ is assumed with probability q_{x_i}. In a Y-insert state, the pair (x_i, y_j) are not considered to be a match, either, but here the pair $(-, y_j)$ has a probability q_{y_j}.

It is important to highlight that every assumption we make for a given pair of

residues is going to have an effect on the following possible pairs in the FSM. It is because every transition between states represents a different increment $\Delta(i, j)$ to the indexes of sequence $\mathbf{x}$ and $\mathbf{y}$ respectively. Each sequence index represents a cursor pointing to a specific position and this cursor moves accordingly to the states we visit along the path. Transitions to a match state have a delta $\Delta(1, 1)$, those to an X-insert state have a delta $\Delta(1, 0)$, while the ones to a Y-insert state have a delta $\Delta(0, 1)$.

Starting from pair (x_0, y_0), there are several possible paths in the FSM and each one of them depends on the current step we take. These paths correspond to the ones that can be traced in the DAG structure in the pairwise alignment. The algorithms 2 and 3 remain valid if we replace the score values with emission and transition probabilities. This probabilistic method lets us give a step forward in our analysis: instead of comparing only two sequences, we can now compare a sequence to a *profile model*.

2.4 Profile hidden Markov models

Using Pairwise Alignment can be very effective when comparing two sequences, but it may not be appropriate when a greater number of sequences is involved. In biological sequence analysis it is usual to compare an unknown target sequence to a family of related sequences that are known to share common properties and common ancestry [DURBIN *et al.*, 1998]. One may suggest using pairwise alignment for comparing the target sequence to every single instance of the family. Notwithstaning, a more efficient alternative is to build a HMM architecture that is specially suited for representing profiles of multiple sequence alignments: the *profile hidden Markov model* [KROGH *et al.*, 1994]. In this section, we will present the definition of a "profile" and we will introduce the concept behind the profile HMM. We will also approach how to optimize the model parameters.

2.4.1 What is a profile?

The term "profile" refers to the pattern found in a multiple sequence alignment [GRIBSKOV *et al.*, 1987]. Before building a profile model from a family of proteins, we will assume that the *multiple alignment* of the family sequences is available. This multiple alignment can be achieved by methods that use structural information or by Expectation-Maximization algorithms such as the Baum-Welch algorithm [BAUM, 1972]. There are also more recent and sophisticated programs, such as T-Coffee [NOTREDAME *et al.*, 2000] and Clustal Omega [SIEVERS *et al.*, 2011], that can align multiple protein sequences efficiently.

In a multiple alignment of a given family we can observe that there are more similarities in some aligned columns than there are in others. The *consensus* line shows the most conserved columns, which reflect the positions that were least modified during evolution. They are the *consensus* columns on which our probabilistic profile model will be based. An extract of multiple alignment of seven globins [BASHFORD *et al.*, 1987] is illustrated below in Figure 2.18.

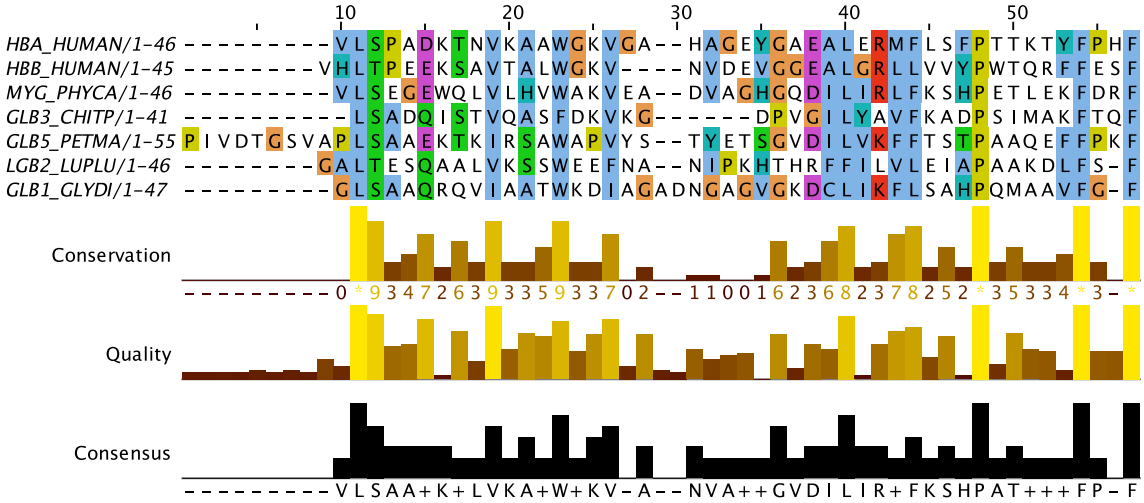


Figure 2.18: An extract of multiple alignment of seven globins using Jalview software [WATERHOUSE *et al.*, 2009]: the *Conservation* graph measures the conservation of physicochemical properties in a column; the *Quality* graph indicates the BLOSUM62 score based on observed substitutions; the *Consensus* graph gives the most common residues and their percentage for each column.

2.4.2 Developing a profile HMM

The aim of a profile hidden Markov model is to represent the aligned sequences of a whole family, not a particular sequence [DURBIN *et al.*, 1998]. The *consensus* sequence or *query* reflects the common ancestor of all sequences from a given group and it represents the multiple alignment the model will learn from, even though this ancestor has never been known. Comparing a *target sequence*, a sequence that does not belong to the training set, to a profile HMM model is the same as comparing the target sequence to the consensus sequence. That is why we can use in profile HMM methods of dynamic programming that are similar to those applied to pairwise alignment. The main aim is to predict if a target sequence $\mathbf{x}$ is homologous to the query or not [EDDY, 2008].

The graphical representation of profile HMM (Figure 2.19) consists of an unrolled version of the FSM of Pair HMM (Figure 2.17b). In the case of Pair HMM, the observations are pairs of residues from both sequences or pairs consisting of a residue

and a gap. Each observation has an *emission probability* and there are also *transition probabilities* between two of the available states: *begin*, *match*, *X-insert*, *Y-insert* and *end*. According to Markov property, the emission probability only depends on the current state of the FSM and the transition probability of reaching the current state only depends on the previous one. In the case of profile HMM, we should now consider that the sequence Y is no longer a specific protein, but a consensus sequence that represents a multiple alignment instead. As to the set of states, there are some changes concerning their names. The Y-insert state is now called *delete* state, once it represents the alignment between a consensus position and a gap, indicating that the amino acid occupying that position has been deleted in relation to the consensus sequence. The X-insert state is now called simply *insert* state, as it represents the alignment between a gap and an amino acid of sequence X, indicating that this amino acid has been inserted in relation to the consensus sequence. The *match* state keeps its name, since it corresponds to the match between a consensus position and an amino acid of sequence X, showing that the amino acid has been kept or replaced during evolution. In profile HMMs, the states no longer emit pairs of residues as observations: match and insert states emit single residues and delete states are 'silent', since they emit gaps only.

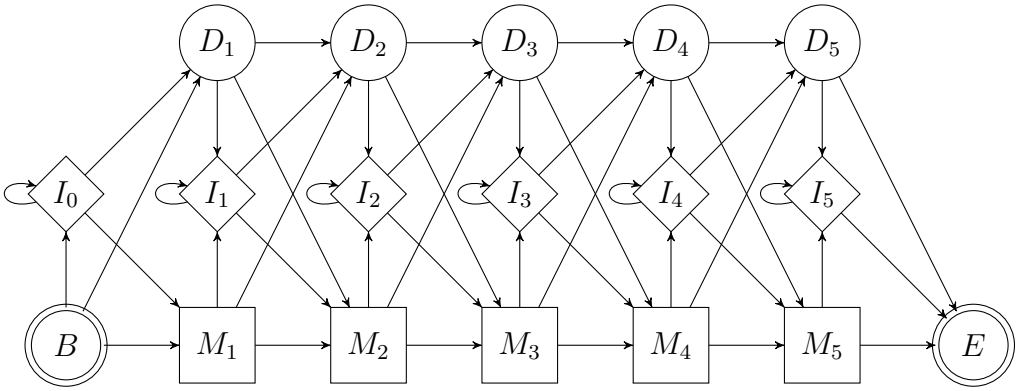


Figure 2.19: FSM representing a profile HMM with length 5

Another crucial difference of profile HMM in relation to Pair HMM is that states in profile HMM are indexed by the position in the multiple alignment. There is no longer only one single match, delete or insert state, but there are three times as many states as there are consensus positions, plus two reffering to begin and end states (B and E). In fact, each consensus position i is represented by a *node* consisting of tree states: the states M_i and D_i, which indicate a match and a deletion in position i, and the state I_i, which refers to an insertion after position i. Also, unlike Pair HMMs, emission and transition probability distributions vary accordingly to the alignment position.

In profile HMMs match states refer to the well conserved positions of the consen-

sus sequence, insert states refer to the scarcely distributed regions of the multiple alignment and delete states refer to the gaps located inside the conserved regions. This is illustrated in Figure 2.20.

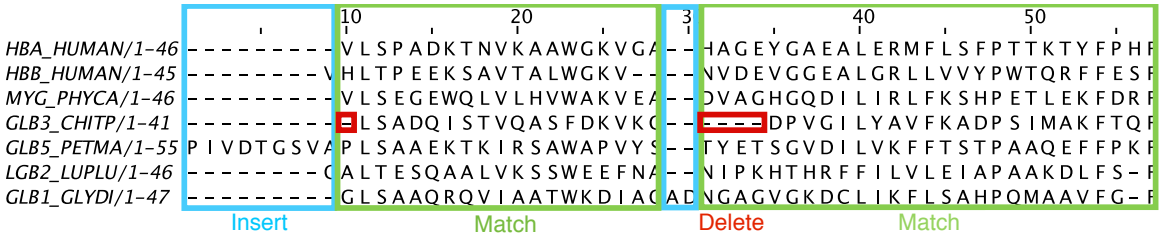


Figure 2.20: An extract of multiple alignment of seven globins: insert state regions are in blue, match state regions are in green and delete state elements are in red.

2.4.3 Profile HMM parameterization

Defining a profile HMM architecture involves choosing the model length and calculating emission and transition probabilities.

Estimating model length

The length of a model consists of the number of states the model contains. Given a multiple alignment, the setup of the model length is basically the decision of which of the multiple alignment columns will be marked as match and which ones will not [DURBIN *et al.*, 1998]. One basic method for choosing the consensus columns is a heuristic rule that selects columns having a number of gaps lower than 50% of its residues.

More sophisticated methods such as *maximum a posteriori* (MAP) demand dynamic programming in order to optimize the model length. Basically, the MAP algorithm calculates the probability of marking each column given the previous columns in the multiple alignment. In the end, it backtracks the optimal model from the last position down to the beginning.

Estimating probabilities

As we discussed earlier, there are two types of probabilities: emission probabilities and transition probabilities. Once the model length is defined and the consensus columns are marked, each state is assigned an emission probability distribution. This distribution determines the probability of emitting a symbol given that state or, in other words, the probability of observing an amino acid in that state.

The delete states always signal a "gap". As to match and insert states, one basic approach of estimating their emission probabilities is to count how many times each

amino acid occurs in a given state and divide this count by the number of emissions from that state. Analogously, transition probabilities can also be estimated by counting the transitions coming out of a previous state and then dividing it by the total number of transitions coming into the current state. In Equations 2.5 we can see the basic formulas for estimating probabilities:

$$a_{kl} = \frac{A_{kl}}{\sum_{l'} A_{kl'}}, \qquad e_k(a) = \frac{E_k(a)}{\sum_{a'} E_k(a')} \tag{2.5}$$

where a_{kl} is the transition probability of current state l given previous state k, A_{kl} is the count of transitions from state k to state l, $e_k(a)$ is the emission probability of emitting residue a given state k and $E_k(a)$ is the count of occurrences of a at state k [DURBIN *et al.*, 1998].

One problem of this approach is that amino acids that do not occur in a certain state during the training stage will be assigned a null probability, which is not necessarily true. A solution for this problem is the addition of pseudocounts, which are constants added to all counts. If this constant is 1, this method is called *Laplace's rule* [DURBIN *et al.*, 1998]. More sophisticated pseudocounts use constants in proportion to the background distribution:

$$e_k(a) = \frac{E_k(a) + Aq_a}{\sum_{a'} E_k(a') + A} \tag{2.6}$$

where A is a constant and q_a is the background probability of amino acid a.

There are also more sophisticated methods, such as *Dirichlet Mixtures* [SJÖLANDER *et al.*, 1996] and *Substitution Matrix Mixtures*, which combine count estimation with richer prior information than background distribution only.

2.4.4 Alignment modes

In general, sequence similarity search methods present three alignment modes: *global*, *glocal* (or *semi-global*) and *local* [EDDY, 2008]. Figure 2.21 illustrates the difference between these modes.

Global alignment mode is the simplest of the configurations above and it is represented by the solid (—) edges and nodes in Figure 2.21. This is the mode we have considered so far and it aligns the entire target sequence to the entire consensus sequence.

Nevertheless, biological sequences may share only domains, which are conserved subparts of a protein sequence that exist independently of the rest of the sequence. Therefore, if we also consider the dashed (- - -) ones in the FSM of Figure 2.21,

then we have the **glocal** or **semi-global** alignment mode, in which the model is local with respect to the target sequence, but global with respect to the consensus sequence. The *flanking* states N and C allow assigning background probabilities to the non-homologous parts in the beginning and in the end of the target sequence, thus making it possible to align a subsequence $k \dots l$ of the target to the entire consensus sequence. Besides, glocal and local alignments require new begin and end states: S and T.

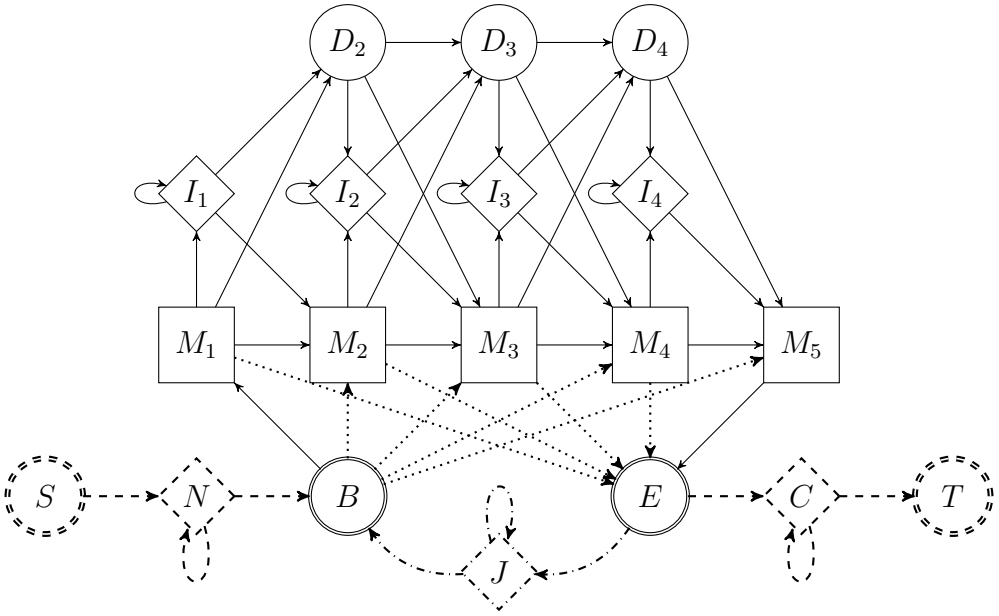


Figure 2.21: FSM representing a profile HMM in different alignment modes. Dashed (- - -) edges and nodes represent the elements required for local or glocal alignment. Dotted ($\cdots$) edges represent the transitions required for local alignment. Dash-dotted (-·-) edges and nodes represent the elements required for local or glocal multihit alignment.

Furthermore, if we also include the dotted ($\cdots$) edges in the analysis, then we have the **local** alignment mode, in which the model is local with respect to both target and consensus sequences. Because there are transitions that skip some match states in local alignment, a subsequence $k \dots l$ of the target sequence can be aligned to a subsequence $i \dots j$ of the consensus sequence. This alignment mode is inspired by the Smith-Waterman algorithm for local pairwise alignment [SMITH and WATERMAN, 1981].

At last, glocal and local alignment modes can be either *multihit* or *unihit*. If we take the dash-dotted (-·-) edges and nodes into consideration, then we have the **multihit** alignment, since the dash-dotted transitions make it possible to perform alignments with repeated matches. The state J allows non-homologous parts in the middle of the target sequence. Without the dash-dotted edges and nodes, glocal

or local algorithms may allow only one aligned region per target sequence, which results in a **unihit** alignment.

2.5 Conditional Random Fields

In this section, we will discuss about *Conditional Random Fields* (CRF), which is another stastistical model besides HMM. Firstly, we will justify the use of discriminative models over generative models. Secondly, we will briefly introduce the theory of CRF and present a simple example for means of illustration. Lastly, we will present the standard parameter estimation process in order to compare it to the gradient tree boosting method, which our work focuses on.

2.5.1 Generative models vs. discriminative models

Before we introduce CRF, let us introduce two different categories of classifiers: *generative* and *discriminative*. The set of generative models includes Naive Bayes Classifier and HMM, while the set of discriminative models includes Logistic Regression and CRF. In a classification problem, where the goal is to evaluate the probability of a target $\mathbf{y}$ given the $\mathbf{x}$, both types of classifiers aim to find the best $p(\mathbf{y}|\mathbf{x})$ [BISHOP and LASSERRE, 2007]. The main difference between them is that discriminative methods model $p(\mathbf{y}|\mathbf{x})$ directly, whereas generative methods model the joint distribution $p(\mathbf{x}, \mathbf{y})$ and use it as a means to calculate $p(\mathbf{y}|\mathbf{x})$.

As we stated earlier, **generative models** are the ones that estimate the joint distribution $p(\mathbf{x}, \mathbf{y})$, which can be viewed as a class-conditioned density $p(\mathbf{x}|\mathbf{y})$ multiplied by a class prior $p(\mathbf{y})$, once $p(\mathbf{x}, \mathbf{y}) = p(\mathbf{x}|\mathbf{y})p(\mathbf{y}))$ according to Bayes' theorem. This joint distribution is then used as an intermediate step to estimate $p(\mathbf{y}|\mathbf{x})$ for new inputs of $\mathbf{x}$. These models are called so, because they describe how the outputs $\mathbf{y}$ *generate* the inputs $\mathbf{x}$ probabilistically [SUTTON and MCCALLUM, 2006]. In fact, generative models are capable of *generating* synthetic examples of $\mathbf{x}$ by sampling from the joint distribution [BISHOP and LASSERRE, 2007]. Once the joint distribution $p(\mathbf{x}, \mathbf{y})$ is modeled, $p(\mathbf{y}|\mathbf{x})$ is calculated as follows:

$$p(\mathbf{y}|\mathbf{x}) = \frac{p(\mathbf{x}, \mathbf{y})}{\sum_{y'} p(\mathbf{x}, \mathbf{y}')} = \frac{p(\mathbf{y})p(\mathbf{x}|\mathbf{y})}{\sum_{y'} p(\mathbf{y}')p(\mathbf{x}|\mathbf{y}')} \tag{2.7}$$

Profile HMM is generative, because it models both $p(\mathbf{y})$ and $p(\mathbf{x}|\mathbf{y})$, as shown in Equation 2.8:

$$
\begin{aligned}
p(\mathbf{y}) &= p(y_1) \times p(y_2|y_1) \times \cdots \times p(y_t|y_{t-1}) \times \cdots \times p(y_T|y_{T-1}) \\
&= a_{0y_1} \times a_{y_1y_2} \times \cdots \times a_{y_{t-1}y_t} \times \cdots \times a_{y_{T-1}y_T} \\
p(\mathbf{x}|\mathbf{y}) &= p(x_1|y_1) \times \cdots \times p(x_t|y_t) \times \cdots \times p(x_T|y_T) \\
&= e_{y_1}(x_1) \times \cdots \times e_{y_t}(x_t) \times \cdots \times e_{y_T}(x_T)
\end{aligned}
\tag{2.8}
$$

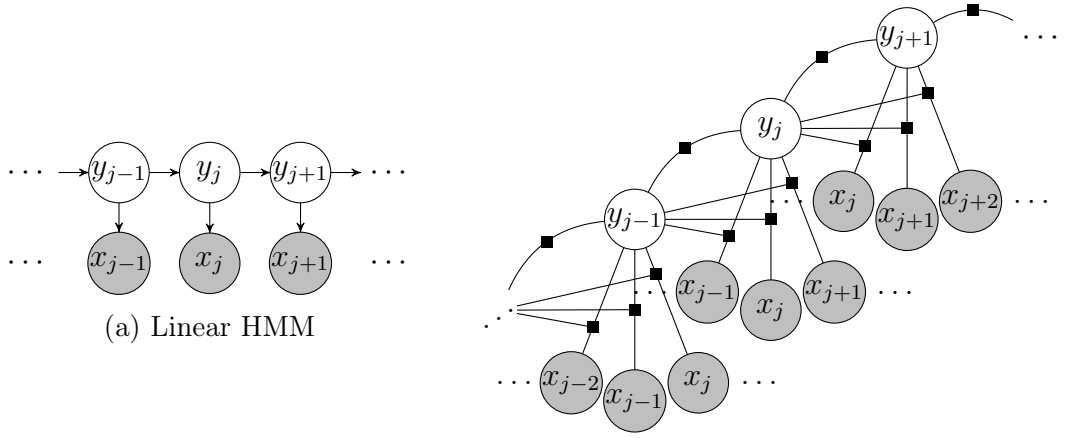
where $a_{y_{t-1}y_t}$ is the transition probability passing to state y_t given the previous state y_{t-1}, and $e_{y_t}(x_t)$ is the emission probability of residue x_t being emitted by state y_t. The parameters $a_{y_{t-1}y_t}$ and $e_{y_t}(x_t)$, $t \in \{1, \ldots, T\}$, belong to the structure of profile HMM itself.

Profile HMM works very well for making inferences on target sequences given a trained model. Nonetheless, in order to model a distribution that is able to *generate* examples of $\mathbf{x}$, HMM has to make very strong assumptions among the components of $\mathbf{x}$, such as the Markov property, according to which each component x_i of $\mathbf{x}$ depends *only* on the state y_i that is associated to it. Even so, it might happen that x_i depends not only on y_i, but also on its neighbors x_{i-1} and x_{i+1} or on other states than y_i, for instance. The complex dependencies between the components of $\mathbf{x}$ could make it intractable to model the joint distribution $p(\mathbf{x}, \mathbf{y})$, but simply ignoring them might compromise the performance of the method [SUTTON and MCCALLUM, 2006]. This is where **discriminative models** present an advantage: they save us the trouble of modeling $p(\mathbf{x}|\mathbf{y})$ in the first place, which allows us to introduce new features of sequence $\mathbf{x}$ (such as the neighbors of x_i) without the need to model new dependencies or make strong assumptions. Indeed, discriminative models directly estimate $p(\mathbf{y}|\mathbf{x})$ from the training dataset and their conditional distribution *discriminates* different values of $\mathbf{y}$ given an input $\mathbf{x}$ in a straightforward way [BISHOP and LASSERRE, 2007]. Furthermore, discriminative models may present a lower asymptotic error, especially when the number of training examples become large, although generative models may converge faster — possibly with a number of training examples that is only logarithmic, rather than linear, in the number of parameters [NG and JORDAN, 2001]. To better understand the discriminative approach, we will introduce *Conditional Random Fields* in Section 2.5.2.

2.5.2 Introduction to Conditional Random Fields

Conditional Random Fields (or CRF) are the discriminative dual of hidden Markov models. In fact, both of them are sequence labeling models that have similar applications, such as named-entity recognition. Nonetheless, because CRF is discriminative, it only makes strong assumptions among the states in $\mathbf{y}$, but not among the observations in $\mathbf{x}$, unlike HMM. Figure 2.22 illustrates the difference between HMM

and CRF.



(a) Linear HMM

(b) Linear-chain CRF

Figure 2.22: CRF and HMM graphs

Similar HMM problems are presented for CRF applications: let $\mathbf{X}$ be a random variable over observation sequences to be labeled and let $\mathbf{Y}$ be a random variable over the corresponding label (or state) sequences. Our main aim is to match each state/label to each observation. This class of problems is called *sequential supervised learning* (SSL).

Definition 1 *Let $G = \{V, E\}$ be graph with vertices V and edges E such that every element $Y_v \in \mathbf{Y}$ is indexed by the vertex $v \in V$. Thus, the pair of random variables $(\mathbf{X}, \mathbf{Y})$ is a **conditional random field** if the random variables Y_v, when conditioned on X, present the Markov property regarding the graph G: $p(Y_v|X, Y_w, \forall w \neq v) = p(Y_v|X, Y_w, w \in N(v))$, where $N(v)$ is the set of vertices that are neighbors of v* [LAFFERTY *et al.*, 2001].

In other words, the definition above means that, given a sequence $\mathbf{X}$, every random variable Y_v in vertex v depends only on the ones indexed by the neighbors of v in CRF. The simplest form that graph G may assume is a linear chain, where $G = (V = \{1, \ldots, n\}, E = \{(i, i+1), 1 \leq i < n\}$ [LAFFERTY *et al.*, 2001]. In the *linear-chain CRF*, the domain of each random variable Y_i is a finite alphabet $\mathcal{Y}$ of labels that can possibly relate to the component X_i.

Let graph $G = (V, E)$ of $\mathbf{Y}$ be a tree whose cliques are edges and vertices. According to the fundamental theory of random fields [HAMMERSLEY and CLIFFORD, 1971], the joint distribution over the label sequence $\mathbf{Y}$ given the observation sequence $\mathbf{X}$ has the following form:

$$P(\mathbf{Y}=\mathbf{y}|\mathbf{X}=\mathbf{x}) = \frac{\exp\left(\sum_{e\in E,k}\lambda_k f_k(e,\mathbf{y}|_e,\mathbf{x}) + \sum_{v\in V,k}\mu_k g_k(v,\mathbf{y}|_v,\mathbf{x})\right)}{Z(\mathbf{X}=\mathbf{x})} \tag{2.9}$$

where $\mathbf{x}$ is an observation sequence, $\mathbf{y}$ is a label sequence, $\mathbf{y}|_e$ is the pair of labels (y_i, y_j) associated with the edge e and $\mathbf{y}|_v$ is the label y_i associated with vertex v. The functions f_k and g_k are binary *feature functions*, and λ_k and μ_k are their respective weights. These functions are assumed to be fixed and given. The weighted sums of feature functions f_k and g_k for all k are called *potential functions*:

$$\begin{aligned}\Psi_e(e,\mathbf{y}|_e,\mathbf{x}) &= \sum_{e\in E,k}\lambda_k f_k(e,\mathbf{y}|_e,\mathbf{x}),\\ \Psi_v(v,\mathbf{y}|_v,\mathbf{x}) &= \sum_{v\in V,k}\mu_k g_k(v,\mathbf{y}|_v,\mathbf{x})\end{aligned} \tag{2.10}$$

The denominator $Z(\mathbf{X}=\mathbf{x})$ is a normalization factor as a function of the input $\mathbf{x}$ [SUTTON and MCCALLUM, 2006] so that the sum of the scores of all possible label sequences for an observation sequence can be equal to 1. This normalization guarantees that we have a probability distribution conditioned on $\mathbf{X}$.

$$Z(\mathbf{X}=\mathbf{x}) = \sum_{\mathbf{y}}\exp\left(\sum_{e\in E,k}\lambda_k f_k(e,\mathbf{y}|_e,\mathbf{x}) + \sum_{v\in V,k}\mu_k g_k(v,\mathbf{y}|_v,\mathbf{x})\right) \tag{2.11}$$

For the sake of simplicity, the random variables $\mathbf{X}$ and $\mathbf{Y}$ will be omitted in the expressions from now on.

Simple grammatical analyzer example

To illustrate the concept of CRF, let us look at an example related to a simple grammatical analyzer. Let us consider that the sequence of observations X are words of a sentence and that we intend to label each word with one of the following word classes: *noun*, *determiner*, *adjective*, *verb*. Let $\mathbf{y}$ be a sequence of labels and $\mathbf{x}$ be the input word sequence.

One useful way of representing this problem is using a *trellis* diagram, as shown in Figure 2.23. A trellis is a graph in which the vertices are organized in columns and all vertices are connected to at least one node to its left and one node node to its right, except for the first and last nodes. In the i-th column of the trellis, there is a node corresponding to each possible value of the domain of random variable Y_i. Let the

input sequence be the sentence *A noisy noise annoys an oyster*. Every path that can be traced from the first column to the last one corresponds to a different possibility of labeling each word of the given sentence. As to the feature functions, they can be viewed as binary functions that are activated when a particular characteristic or situation is present. For example, we could define g_1 for vertex Y_i as a function that returns 1 if and only if the label *determiner* is assigned to Y_i and the word '*a*' is assigned to X_i. We could also define f_1 for the edge between Y_{i-1} and Y_i as a function that returns 1 if and only if the label *noun* is assigned to Y_i, the label *determiner* is assigned to Y_{i-1} and the word assigned to X_i begins with a capital letter. The more complex the problem, the higher the number of features we can come up with. Once a set of feature functions is defined, we optimize the weights associated to them so that each feature exerts its due influence in Equation 2.9, thus maximizing the total likelihood, as we will see in Section 2.5.3.

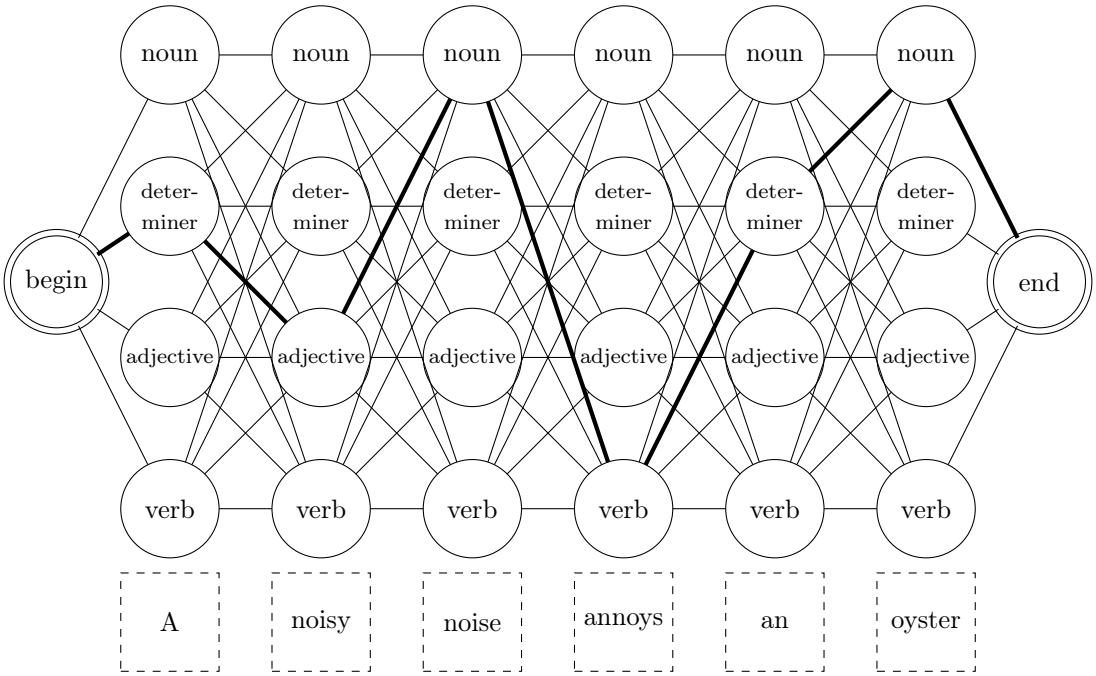


Figure 2.23: A trellis diagram representing a simple grammatical analyzer

2.5.3 Standard parameter estimation of CRF

Suppose we have the training dataset $\mathcal{D} = \{\mathbf{x}^{(i)}, \mathbf{y}^{(i)}\}_{i=1}^{N}$, where each sequence $\mathbf{x}^{(i)} = \{x_1, \ldots, x_T\}$ is an observation sequence and each sequence $\mathbf{y}^{(i)} = \{y_1, \ldots, y_T\}$ is the desired label sequence for that observation.

As the feature functions f_k and g_k are assumed to be given and fixed, standard parameter estimation of CRFs focuses on adjusting the weights $\theta = \{\lambda_k, \mu_k\}$ so that the *conditional log likelihood* is maximized:

$$\ell(\theta) = \sum_{i=1}^{N} \log p(\mathbf{y}^{(i)}|\mathbf{x}^{(i)}) \tag{2.12}$$

To prevent overfitting, a *regularization penalty* $-d$ may be added to the log likelihood so as to penalize weight vectors whose norm is too large. By inserting the Equation 2.9 into Equation 2.12 and taking into account the regularization term, we obtain the following expression:

$$\ell(\theta) = \sum_{i=1}^{N} \sum_{k=1}^{K} \left[\sum_{t=2}^{T} \lambda_k f_k(y_{t-1}^{(i)}, y_t^{(i)}, \mathbf{x}^{(i)}) + \sum_{t=1}^{T} \mu_k g_k(y_t^{(i)}, \mathbf{x}^{(i)}) \right] - \sum_{i=1}^{N} \log Z(\mathbf{x}^{(i)}) - d \tag{2.13}$$

Parameters $\theta = \{\lambda_k, \mu_k\}$ are obtained by the maximization of the above expression. Since this expression cannot be maximized in a closed form, numerical optimization is used based on the partial derivatives of $\ell(\theta)$ in respect to λ_k and μ_k [SUTTON and MCCALLUM, 2006].

There are two major difficulties in learning CRFs. The first one is that the learning procedure involves making inferences for each iteration and for each example $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$ in order to calculate the log likelihood $\log(p(\mathbf{y}^{(i)}|\mathbf{x}^{(i)}))$. Therefore, CRF learning speed are limited by the performance of inference operations, such that CRF learning procedures usually look to minimize the total number of iterations and maximize the learning progress within each iteration [DIETTERICH *et al.*, 2004].

The second difficulty lies in the number of feature combinations that can describe potential functions Ψ even in simple problems. Notice that features f_k and g_k are fixed and given throughout the standard parameter estimation process, where they are treated as independent features. On the one hand, the assumption that these basic features are independent contributions to the potential functions Ψ may compromise the method performance. On the other hand, creating all possible combinations of basic features can lead to a combinatory explosion. This would result in an infeasible solution, depending on the number of initial basic features of the problem [DIETTERICH *et al.*, 2004]. One alternative is to use *feature induction* along the learning process, in which basic features are gradually added to the ones existing in the model and new combinations are evaluated based on the impact they have on the potential functions [MCCALLUM, 2003]. One other option is the one this present study focuses on: using gradient tree boosting method [BREIMAN *et al.*, 1984] to iteratively construct regression trees that will play the role of more complex feature functions [DIETTERICH *et al.*, 2004].

2.5.4 Profile Conditional Random Fields

With conditional random fields being viewed as an evolution of hidden Markov models, it is natural to look to integrate the profile HMM idea into CRF. The concept of a *profile Conditional Random Fields* was presented by KINJO [2009]. It basically follows the finite state machine of profile HMM (see Figure 2.19) and it considers that the states $M_1, M_2, \ldots, M_N, I_1, I_2, \ldots, I_N, D_1, D_2, \ldots, D_N$ are now the labels in CRF.

As to the feature functions, this version of profile CRF uses 3 types of feature functions: singlet (or vertex-related) feature functions g, doublet (or edge-related) feature functions f and pairwise feature functions u.

$$P(\mathbf{y}|\mathbf{x}) = \frac{\exp\left(\sum_{t,k} \lambda_k f_k(y_{t-1}, y_t, \mathbf{x}) + \sum_{t,k} \mu_k g_k(y_t, \mathbf{x}) + \sum_{t'<t,k} \nu_k u_k(y_{t'}, y_t, \mathbf{x})\right)}{Z(\mathbf{x})} \quad (2.14)$$

To illustrate these functions, let us suppose that we are in position t of the alignment between observation sequence $\mathbf{x}$ and $\mathbf{y}$.

Singlet feature functions are related to a vertex of the CRF graph. For instance, it can be a function that returns 1 if the label in y_t is I_j and the elements in a 5-length window $w_t(\mathbf{x}) = \{x_{t-2}, x_{t-1}, x_t, x_{t+1}, x_{t+2}\}$ are all hydrophilic. This is represented by the following equation:

$$g_{00000}(y_t, w_t(\mathbf{x})) = \delta_{I_j}(y_t)\delta_{00000}(bin(w_t(\mathbf{x}))) \quad (2.15)$$

where $bin(w_t(\mathbf{x}))$ is a function that converts the window elements into a binary sequence in which 0 stands for hydrophilic and 1 for hydrophobic.

Doublet feature functions are related to an edge of the CRF graph, which means that they take as input two consecutive elements of label sequence $\mathbf{x}$. The example shown in singlet feature functions can be used, but two consecutive labels, such as M_j, I_j, are inputted into the function instead of just one, as the following expression shows:

$$f_{00000}(y_{t-1}, y_t, w_t(\mathbf{x})) = \delta_{M_j}(y_{t-1})\delta_{I_j}(y_t)\delta_{00000}(bin(w_t(\mathbf{x}))) \quad (2.16)$$

Pairwise feature functions are like doublet feature functions, with the exception that the two input states are not necessarily consecutive in the sequence. This kind of feature functions demands deeper knowledge on secondary and tertiary structures of proteins, where interactions between two non-consecutive amino acids are visible. So, let us ignore this category for the present work.

In [KINJO, 2009], the parameters are estimated by maximizing the log likelihood, similarly to the standard procedure shown in Section 2.5.3. The log likelihood is derived with respect to the weights $\theta = (\alpha_k, \beta_k, \nu_k)$ and then the local maximum is found after equationing the first derivative to zero.

To calculate the probability of a given transition, we use the following expression:

$$P(y_{t-1}, y_t|\mathbf{x}) = \frac{\alpha_{y_{t-1}}(t-1) \times \exp(F_{y_t}(y_{t-1}, \mathbf{x})) \times \beta_{y_t}(t)}{Z(\mathbf{x})} \tag{2.17}$$

where $\alpha_{y_{t-1}}(t-1)$ is the forward score from the first position up to position $t-1$, $\beta_{y_t}(t)$ is the backward score from the last position down to position t and $Z(\mathbf{x})$ is the normalization factor. More details on the forward-backward algorithm will be given later in Section 2.7.2. For the sake of clarity, function F is defined as the exponent from Equation 2.14:

$$F_{y_t}(y_{t-1}, \mathbf{x}) = \sum_{t,k} \lambda_k f_k(y_{t-1}, y_t, \mathbf{x}) + \sum_{t,k} \mu_k g_k(y_t, \mathbf{x}) + \sum_{t'<t,k} \nu_k u_k(y_{t'}, y_t, \mathbf{x}) \tag{2.18}$$

Let us analyze a different method for performing parameter estimation in CRF, to which the following section is wholly dedicated.

2.6 Learning CRFs with Gradient Tree Boosting

Despite being related to a different approach to CRF parameter estimation, the concept of learning CRFs with gradient tree boosting [DIETTERICH *et al.*, 2004] deserves a section of its own.

In the beginning of this section, we will briefly present the concept of *functional gradient ascent*. Then, we will analyze how to perform the gradient tree boosting method using propositional logic. Lastly, we will see an adaptation of the method that includes first-order logic [GUTMANN and KERSTING, 2006].

2.6.1 Functional Gradient Ascent

As we mentioned earlier in Section 2.5.3, the gradient tree boosting method implies that regression trees are treated as feature functions in CRF. Therefore, a potential function, which is the sum of feature functions, are a forest of regression trees that is grown each iteration. A tree can be viewed as a complex combination of different features, as it is built up based on the average characteristics of training examples. This reduces the effects of any assumption regarding independence between features.

Regression trees are a class of decision trees that has a specific property: the

values they return are real and continuous. So, it is not a farfetched idea to use them as mathematical functions that return an output value for a set of input arguments.

Let us review the expression of CRF probability distribution in Equation 2.19, after replacing the sums of feature functions in Equation 2.9 with the potential functions from Equation 2.10). As we will focus on linear-chain CRFs, we may consider that each vertex is indexed by a position t and each edge is indexed by a pair $(t-1, t)$.

$$P(\mathbf{y}|\mathbf{x}) = \frac{\exp\left(\sum_{t=2}^{T} \Psi(y_{t-1}, y_t, \mathbf{x}) + \sum_{t=1}^{T} \Psi(y_t, \mathbf{x})\right)}{Z(\mathbf{x})} \tag{2.19}$$

Gradient tree boosting is based on the idea of *functional gradient ascent*, which assumes that a function Ψ behaves like a series of component functions. Each component takes a step in the gradient direction to approach a local or global maximum.

$$\Psi_m = \Psi_0 + \Delta_1 + \cdots + \Delta_m \tag{2.20}$$

In our case, the function we intend to maximize is the log likelihood $\sum_{i=1}^{N} \log p(\mathbf{y}^{(i)}|\mathbf{x}^{(i)})$. Thus, each *function gradient* Δ_m corresponds to the log likelihood gradient in respect to the potential function of the current iteration. As there is no analytical function for the log likelihood, we collect the gradient values from each example $(\mathbf{y}^{(i)}, \mathbf{x}^{(i)})$ from the dataset $\mathcal{D}$.

$$\Delta_m(\mathbf{y}^{(i)}, \mathbf{x}^{(i)}) = \nabla_\Psi \left[\log p(\mathbf{y}^{(i)}|\mathbf{x}^{(i)}; \Psi))\right]\big|_{\Psi_{m-1}} \tag{2.21}$$

Therefore, for every training example there is a functional gradient value associated to it. Based on the points $(\mathbf{y}^{(i)}, \mathbf{x}^{(i)}, \Delta_m(\mathbf{y}^{(i)}, \mathbf{x}^{(i)}))$, a regression tree $h_m(\mathbf{y}, \mathbf{x})$ is constructed in such a way that its distance to the gradient is minimized, as we will see in Section 2.6.2.

$$\sum_{i=1}^{N} \left[h_m(\mathbf{y}^{(i)}, \mathbf{x}^{(i)}) - \Delta_m(\mathbf{y}^{(i)}, \mathbf{x}^{(i)})\right]^2 \tag{2.22}$$

Then, the new tree h_m joins the collection Ψ_{m-1} of regression trees that were built in earlier iterations. Taking a step in the direction of the function h_m, we have:

$$\Psi_m = \Psi_{m-1} + \eta h_m$$

where η is the step size parameter, to which we usually assign 1. It is important to notice that h_m is an approximation of the desired Δ_m and both of them will point in the same general direction approximately. In the end, Ψ_m behaves like a forest

of regression trees and its value is the sum of the results returned by all regression trees of this forest.

2.6.2 Gradient Tree Boosting Algorithm

In practice, gradient tree boosting iteratively creates new regression trees that helps potential functions reach their maximum. Function F^{y_t} can be viewed as a combination of vertex and edge potential functions. This function represents the "desirability" or an "unnormalized log probability" of state y_t given the previous state y_{t-1} and the observation $\mathbf{x}$. There is a function F^k for possible each class label k [DIETTERICH *et al.*, 2004].

$$F^{y_t}(y_{t-1}, \mathbf{x}) = \Psi(y_{t-1}, y_t, \mathbf{x}) + \Psi(y_t, \mathbf{x}) \tag{2.23}$$

So, we can rewrite the expression of CRF probability distribution as follows:

$$P(\mathbf{y}|\mathbf{x}) = \frac{\exp \sum_t F^{y_t}(y_{t-1}, \mathbf{x})}{Z(\mathbf{x})} \tag{2.24}$$

For the sake of computability, let us replace the sequence of observations $\mathbf{x}$ with a window w_d centered in position d such that $w_d(\mathbf{x}) = \langle x_{d-\lceil L/2 \rceil}, \ldots, x_d, \ldots, x_{d+\lceil L/2 \rceil} \rangle$, where L is the window length.

If we want the new regression tree to contribute in the gradient direction, it is necessary to calculate the derivative of the log probability distribution with respect to the function $F^v(u, w_d(\mathbf{x}))$, where u and v can be any pair of consecutive class labels.

$$\begin{aligned}
&\frac{\partial \log P(\mathbf{y}|\mathbf{x})}{\partial F^v(u, w_d(\mathbf{x}))} = \\
&= \frac{\partial}{\partial F^v(u, w_d(\mathbf{x}))} \left[\sum_t F(y_{t-1}, y_t, w_d(\mathbf{x})) - \log Z(\mathbf{x}) \right] \\
&= I(y_{d-1} = u, y_d = v) - \frac{1}{Z(\mathbf{x})} \frac{\partial \left\{ \sum_k \left[\sum_{k'} \alpha(k', d-1) \exp F^k(k', w_d(\mathbf{x})) \right] \beta(k, d) \right\}}{\partial F^v(u, w_d(\mathbf{x}))} \\
&= I(y_{d-1} = u, y_d = v) - \frac{\alpha(u, d-1) \exp F^v(u, w_d(\mathbf{x})) \beta(v, d)}{Z(\mathbf{x})} \\
&= I(y_{d-1} = u, y_d = v) - P(y_{d-1} = u, y_d = v|\mathbf{x})
\end{aligned} \tag{2.25}$$

Notice that the α and β functions are analogous to the forward-backward algorithms we saw in profile HMM. The first term is the identity function $I(y_{d-1} =$

$u, y_d = v$), which returns 1, in case labels u and v really occur in positions $d-1$ and d of the training example, and 0, otherwise. The second term $P(y_{d-1} = u, y_d = v|\mathbf{x})$ consists of the probability of labels u and v being assigned to positions $d-1$ and d, given the observation sequence $\mathbf{x}$, according to the current model.

The gradient value represents a feedback value that results from the comparison between the desired value (what really is according to the training set) and the value obtained by the model as is. For example, if we consider the case in which labels u and v are really present in positions $d-1$ and d of sequence $\mathbf{x}$ but this pair receives a null probability from the model, then the resulting gradient would be $1 - 0 = 1$, which is a positive feedback. On the opposite, if u and v do not happen to occur in same positions but they receive a probability value of 0.5 from the model, then the resulting gradient would be $0 - 0.5 = -0.5$, which is a negative feedback.

Algorithm 4 shows the procedure for gradient tree boosting. As we mentioned earlier, in order to train CRFs by gradient tree boosting, we must use inference at each iteration. This inference is evident in the functional gradient expression shown in Equation 2.25, where it is necessary to execute forward and backward algorithms, in order to estimate the probability $P(y_{d-1} = u, y_d = v|\mathbf{x})$. A thorough training process checks every possible pair of labels (u, v) (or (k', k) in Algorithm 5) for every position d within the observation sequence $\mathbf{x}$. Then, negative or positive feedback values are provided accordingly to the functional gradient calculated for each pair (u, v). These feedback values, in their turn, are stored to be later used in the induction of a new regression tree that will join the group of trees created so far. Then, a new iteration of gradient tree boosting is started over again.

Algorithm 4 Algorithm for gradient tree boosting [DIETTERICH *et al.*, 2004]

procedure TREEBOOST($\mathcal{D}, L$) ▷ $\mathcal{D} = \{(\mathbf{x}^{(i)}, \mathbf{y}^{(i)}) : i, \ldots, N\}$
  for class k from 1 to K **do**
    Initialize $F_0^k(\cdot, \cdot) = 0$
  for iteration m from 1 to M **do**
    for class k from 1 to K **do**
      $Set(k) \leftarrow$ GENERATEEXAMPLES($k, \mathcal{D}, Pot_{m-1}$) ▷ See Algorithm 5
      ▷ where $Pot_{m-1} = \{F_{m-1}^{k'} : k' = 1, \ldots, K\}$
      $h_m(k) \leftarrow$ FITREGRESSIONTREE($Set(k), L$)
      $F_m^k \leftarrow F_{m-1}^k + h_m(k)$
  return F_M^k for every class k

Algorithm 5 Procedure for generating examples [DIETTERICH *et al.*, 2004] that is called by Algorithm 4

procedure GENERATEEXAMPLES($k, \mathcal{D}, Pot_{m-1}$)
  $Set(k) \leftarrow \{\}$
  for each example i in dataset $\mathcal{D}$) **do**
    Execute the forward-backward algorithm on $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$ to get $\alpha(k,t)$ and $\beta(k,t)$ for all k and t
    for position t from 1 to $T^{(i)}$ **do** ▷ $T^{(i)}$ is the length of $\mathbf{y}^{(i)}$
      for class k' from 1 to K **do**
        $P(y_{t-1}^{(i)} = k', y_t^{(i)} = k|\mathbf{x}^{(i)}) \leftarrow \frac{\alpha(k',t-1)\exp F_{m-1}^{k}(k', w_t(\mathbf{x}^{(i)}))\beta(k,t)}{Z(\mathbf{x}^{(i)})}$
        $\Delta(k', k, i, t) \leftarrow I(y_{t-1}^{(i)} = k', y_t^{(i)} = k) - P(y_{t-1}^{(i)} = k', y_t^{(i)} = k|\mathbf{x}^{(i)})$
        Insert input-output pair $((k', k, w_t(\mathbf{x}^{(i)}), \Delta(k', k, i, t))$ into $Set(k)$
  return $Set(k)$

2.7 Inference scores

In this section we will present algorithms for computing the likelihood of a target sequence based on a profile model. With due adjustments, these algorithms work for both HMMs and CRFs. There are two ways of estimating the likelihood of a target sequence based on a given model: Viterbi algorithm and forward-backward algorithm. Although these algorithms are generic, the versions we will present are adjusted specifically for profile models.

2.7.1 Viterbi algorithm

The Viterbi algorithm returns the probability of the best state sequence $\mathbf{y}^*$ resulted from the alignment between the target sequence $\mathbf{x}$ and the model H. Because HMMs are generative and CRFs are discriminative, in HMMs this likelihood is represented by the conjoint probability $P(\mathbf{y}^*, \mathbf{x}|H)$ (which is conditioned on the model H), while in CRFs this is represented by the conditional probability $P(\mathbf{y}^*|\mathbf{x}, H)$.

In practice, the Viterbi algorithm consists of a set of equations that are calculated in a recursive way. The base case is $V_0^{BEGIN}(0,0)$, which is initialized to 0. The Viterbi recursion steps are as follows:

$$V_j^M(i) = \max \begin{cases} V_{j-1}^M(i-1) + f(M_{j-1}, M_j, \mathbf{x}, i), \\ V_{j-1}^I(i-1) + f(I_{j-1}, M_j, \mathbf{x}, i), \\ V_{j-1}^D(i-1) + f(D_{j-1}, M_j, \mathbf{x}, i) \end{cases}$$

$$V_j^I(i) = \max \begin{cases} V_j^M(i-1) + f(M_j, I_j, \mathbf{x}, i), \\ V_j^I(i-1) + f(I_j, I_j, \mathbf{x}, i), \\ V_j^D(i-1) + f(D_j, I_j, \mathbf{x}, i) \end{cases} \tag{2.26}$$

$$V_j^D(i) = \max \begin{cases} V_{j-1}^M(i) + f(M_{j-1}, D_j, \mathbf{x}, i), \\ V_{j-1}^I(i) + f(I_{j-1}, D_j, \mathbf{x}, i), \\ V_{j-1}^D(i) + f(D_{j-1}, D_j, \mathbf{x}, i) \end{cases}$$

where i is the index for sequence $\mathbf{x}$, t is the index for sequence $\mathbf{y}$ and j is the index for the profile states. The score function f refers to either f_{HMM} in case of HMM or f_{CRF} in case of CRF, as shown in Equation 2.27.

$$\begin{aligned} f_{HMM}(y_{t-1}, y_t, \mathbf{x}, i) &= \underbrace{\log\left(a_{y_{t-1}y_t}\right)}_{\text{Transition log probability}} + \underbrace{\log\left(e_{y_t}(x_i)\right)}_{\text{Emission log probability}} \\ f_{CRF}(y_{t-1}, y_t, \mathbf{x}, i) &= \underbrace{F^{y_t}(y_{t-1}, \mathbf{x})}_{\text{Potential function}} \end{aligned} \tag{2.27}$$

where $a_{y_{t-1}y_t}$ is the probability of the transition from state y_{t-1} to state y_t, $e_{y_t}(x_i)$ is the probability of x_i being emitted by y_t and F is the CRF potential function.

To calculate the likelihood of the optimal alignment, we must execute the Viterbi algorithm starting at the last element (x_M, y_T), where M and T are the lengths of sequences $\mathbf{x}$ and $\mathbf{y}$. Let us consider that E corresponds to the last state of a size-N profile model. In case of HMMs, we have:

$$P(\mathbf{y}^*, \mathbf{x}|H) = \exp(V_N^{END}(T, M)) \tag{2.28}$$

In case of CRFs, the result must be normalized by $Z(\mathbf{x})$:

$$P(\mathbf{y}^*|\mathbf{x}, H) = \frac{\exp(V_N^{END}(T, M))}{Z(\mathbf{x})} \tag{2.29}$$

2.7.2 Forward-backward algorithm

The forward-backward algorithm returns the probability $P(\mathbf{x}|M)$ of all possible alignments between the target sequence $\mathbf{x}$ and the model M. As the name suggests, the forward algorithm sums the scores of all paths coming from the beginning up to the current state, while the backward algorithm sums the scores of all paths coming from the end down to the current state.

With $\alpha_0^{BEGIN}(0)$ initialized to 0, the forward equations are as follows:

$$\begin{aligned}
\alpha_j^M(i) =& \alpha_{j-1}^M(i-1) \times \exp(f(M_{j-1}, M_j, \mathbf{x}, i)) \\
&+ \alpha_{j-1}^I(i-1) \times \exp(f(I_{j-1}, M_j, \mathbf{x}, i)) \\
&+ \alpha_{j-1}^D(i-1) \times \exp(f(D_{j-1}, M_j, \mathbf{x}, i)) \\
\\
\alpha_j^I(i) =& \alpha_j^M(i-1) \times \exp(f(M_j, I_j, \mathbf{x}, i)) \\
&+ \alpha_j^I(i-1) \times \exp(f(I_j, I_j, \mathbf{x}, i)) \\
&+ \alpha_j^D(i-1) \times \exp(f(D_j, I_j, \mathbf{x}, i)) \\
\\
\alpha_j^D(i) =& \alpha_j^M(i) \times \exp(f(M_{j-1}, D_j, \mathbf{x}, i)) \\
&+ \alpha_j^I(i) \times \exp(f(I_{j-1}, D_j, \mathbf{x}, i)) \\
&+ \alpha_j^D(i) \times \exp(f(D_{j-1}, D_j, \mathbf{x}, i))
\end{aligned} \tag{2.30}$$

With $\beta_N^{END}(0)$ initialized to 0, the backward equations are as follows:

$$\begin{aligned}
\beta_j^M(i) =& \exp(f(M_j, M_{j+1}, \mathbf{x}, i+1) \times \beta_{j+1}^M(i+1) \\
&+ \exp(f(M_j, I_j, \mathbf{x}, i+1) \times \beta_j^I(i+1) \\
&+ \exp(f(M_j, D_{j+1}, \mathbf{x}, i) \times \beta_{j+1}^D(i) \\
\\
\beta_j^I(i) =& \exp(f(I_j, M_{j+1}, \mathbf{x}, i+1)) \times \beta_{j+1}^M(i+1) \\
&+ \exp(f(I_j, I_j, \mathbf{x}, i+1)) \times \beta_j^I(i+1) \\
&+ \exp(f(I_j, D_{j+1}, \mathbf{x}, i)) \times \beta_{j+1}^D(i) \\
\\
\beta_j^D(i) =& \exp(f(D_j, M_{j+1}, \mathbf{x}, i+1)) \times \beta_{j+1}^M(i+1)) \\
&+ \exp(f(D_j, I_j, \mathbf{x}, i+1)) \times \beta_j^I(i+1)) \\
&+ \exp(f(D_j, D_{j+1}, \mathbf{x}, i)) \times \beta_{j+1}^D(i))
\end{aligned} \tag{2.31}$$

The basic difference between Viterbi equations and forward equations is that

the first ones choses the maximum previous score, whereas the second ones sum the previous scores. The backward algorithm is analogous to the forward algorithm, with the exception that it begins at the last position (x_M, y_T), where x_M is the last observation and y_T is the last state.

The normalization factor $Z(\mathbf{x})$ is obtained by calculating either the forward algorithm for the last alignment element or the backward algorithm for the first alignment element. Both methods should give the same results.

$$Z(\mathbf{x}) = \alpha_N^{END}(M) = \beta_0^{BEGIN}(0) \tag{2.32}$$

2.7.3 Scaled forward-backward algorithm

In last subsection we saw the forward-backward algorithm, which is used to calculate the α and β values. Nonetheless, some floating point engines may not have the necessary precision to compute α and β based on the Equations 2.30 and 2.31. In fact, these operations will normally raise an underflow or overflow exception during the algorithm execution. The *scaled* forward-backward is a workaround for this problem, as it normalizes the alpha values at each column of trellis [RAHIMI and RABINER].

Let $\hat{\alpha}$ and $\hat{\beta}$ be the *scaled forward* and *scaled backward variables*. With scaled variables, it is no longer necessary to calculate the normalization term $Z(\mathbf{x})$ in order to estimate the transition probability between states u and v [RAHIMI and RABINER]:

$$\begin{aligned} P(y_t = u, y_{t+1} = v|\mathbf{x}) &= \left.\frac{\alpha_t(u) \times f(u, v, \mathbf{x}, i(t+1, v)) \times \beta_{t+1}(v)}{Z(\mathbf{x})}\right\} \text{standard} \\ P(y_t = u, y_{t+1} = v|\mathbf{x}) &= \left.\hat{\alpha}_t(u) \times f(u, v, \mathbf{x}, i(t+1, v)) \times \hat{\beta}_{t+1}(v)\right\} \text{scaled} \end{aligned} \tag{2.33}$$

where t is an index for a trellis column, u is a state in column t, v is the state in column $t+1$. Notice that $i(t+1, v)$ is a function that returns the amino acid position i that corresponds to the state v in trellis column $t+1$.

The variable $\hat{\alpha}$ is expressed as follows:

$$\hat{\alpha} = \frac{\alpha_t(u)}{\sum\limits_{u' \in col(t)} \alpha_t(u') = C_t \alpha_t(u')} \tag{2.34}$$

where C_t is a normalization coefficient and the values $\alpha_t(j)$ for $j \in \{1, \ldots, N\}$ correspond to the forward scores of all elements in a given trellis column.

In order to calculate the scaled forward scores for the nodes of a trellis, we must follow the recursion steps below:

$$\begin{aligned}
\bar{\alpha}_1(u) &= \alpha_1(u) \\
\bar{\alpha}_{t+1}(v) &= \sum_{u \in col(t)} \hat{\alpha}_t(u) \times f(u, v, \mathbf{x}, i(t+1, v)) \\
c_{t+1} &= \frac{1}{\sum_{v \in col(t+1)} \bar{\alpha}_{t+1}(v)} \\
\hat{\alpha}_{t+1}(v) &= c_{t+1} \bar{\alpha}_{t+1}(v)
\end{aligned} \tag{2.35}$$

where c_t is an auxiliary coefffient and $\bar{\alpha}_t$ is an auxiliary scaled forward variable. The variable $\hat{\beta}_t$ is expressed as follows:

$$\begin{aligned}
\hat{\beta}_t(k) &= D_t \beta_t(k) \\
D_t &= \frac{C_T}{C_{t-1}}
\end{aligned} \tag{2.36}$$

where T is the last trellis column and D_t is the backward normalization coefficient. To use $\hat{\beta}_t$, the recursion steps below must be followed:

$$\begin{aligned}
\bar{\beta}_T(u) &= \beta_T(u) \\
\bar{\beta}_t(u) &= \sum_{v \in col(t+1)} f(u, v, \mathbf{x}, i(t+1, v)) \times \hat{\beta}_t(v) \\
c_t &= \frac{1}{\sum_{u \in col(t)} \bar{\beta}_{t+1}(u)} \\
\hat{\beta}_t(u) &= c_t \bar{\beta}_t(u)
\end{aligned} \tag{2.37}$$

2.8 Top-Down Induction of Logical Decision Trees

Once the general gradient tree algorithm has been discussed, there is one point left to mention: fitting regression trees using first-order logic.

2.8.1 Propositional Logic vs. First-Order Logic

The TreeCRF algorithm proposed by DIETTERICH *et al.* [2004] uses propositional logic mainly. It means that the input sequence of observation $\mathbf{x}$ is a set of logic propositions. Notwithstanding, many observation sequences in the real world, may present a relational structure, which causes the elements of such sequences to behave like atoms in first-order logic [GUTMANN and KERSTING, 2006]. It would

not be possible to express such relational structure if we used only propositional logic.

For example, let P, Q and R be propositions from our input observation sequence. Firstly, let us consider that each proposition refers to the occupation of a given position in the input string by a certain amino acid. We can illustrate this with the following propositions.

$$P \triangleq \text{The amino acid in position 1 is Glycine}$$
$$Q \triangleq \text{The amino acid in position 2 is Methionine}$$
$$R \triangleq \text{The amino acid in position 3 is Tryptophan}$$

The idea behind these three propositions above are properly expressed in propositional logic. However, it would not be the case if we took propositions that contain quantified variables such as the following propositions.

$$P \triangleq \text{Every amino acid in position 1 is tiny}$$
$$Q \triangleq \text{Glycine is located in position 1}$$
$$R \triangleq \text{Glycine is tiny}$$

Although we know that $P \wedge Q \rightarrow R$, it is not possible to reach that conclusion based on the propositional terms P, Q and R only. So, first-order logic becomes useful when not only do we express facts, but also objects and their relations. In the example below, by using the formula in line 1 and the atom in line 2, we can infer the atom in line 3.

$$\begin{array}{l} 1 : \forall x \, (position(x, 1) \rightarrow tiny(x)) \\ 2 : position(Glycine, 1) \\ \hline 3 : tiny(Glycine) \end{array}$$

Regarding the greater expressiveness of first-order logic, a new version of the gradient tree boosting algorithm was created by GUTMANN and KERSTING [2006] in order to support first-order logic atoms in the input observation sequence. This more sophisticated algorithm is named TildeCRF, as it is based on *Top-Down Induction of Logical Decision Trees*, or Tilde [BLOCKEEL and RAEDT, 1998].

2.8.2 Introduction to TILDE

The algorithm TILDE aims to build up decision trees from examples expressed in first-order logic. In our specific case, we refer to TILDE-RT algorithm, which is adapted for induction of logical regression trees.

Using TILDECRF algorithm, the observation sequence becomes a set of first-order logical facts. The dataset of all known observation sequences and their respective real numbers consists of the set of logical examples from which regression trees are induced.

The specification of this problem fits in the setting of *learning with interpretations* [DE RAEDT, 1996]:

Given: the set of real numbers $\mathbb{R}$, a set of classified examples E (each element of E is in the form (e, r), where e is a set of facts and r is a real number) and a background theory B.

Find: a hypothesis H, such that for all $(e, r) \in E$, $H \wedge e \wedge B \models r$ and $H \wedge e \wedge B \not\models r'$, where $r' \in \mathbb{R}$ and $|r' - r| > \epsilon$.

It is important to remember that our aim here is to replace an otherwise analytical function F with a regression tree. This tree behaves like the function F as it returns an "unnormalized log probability" as a function of the observation $\mathbf{x}$, of the previous label $y_{t-1} = k'$ and of the current label $y_t = k$.

Algorithm 6 Predicting an example value using a logical regression tree [BLOCKEEL, 1998]

```
1:  procedure PREDICT(example e, background knowledge B)
2:      query := true
3:      node := root
4:      while node is not a leaf do
5:          if ← query ∧ node.conjunction succeeds in e ∧ B then
6:              query := query ∧ node.conjunction
7:              node := node.yes_child
8:          else
9:              node := node.no_child
                                              ▷ node is a leaf
10:     return node.value
```

A logical decision or regression tree is a binary tree where, except for the leaves, every node contains a logical conjunction and two children: a "yes" child and a "no" child. In Algorithm 6 we observe the procedure for inferring what value is associated with an input example. The procedure begins at the root node and the combination of the node conjunction with the query so far created is confronted with the example. In case of success, we move to the "yes" child and the node conjunction is added to the query. In case of failure, we move to the "no" child. The process repeats until a leaf is found and, then, the value associated to it is returned.

The mechanism of inducing logical trees is similar to the classic algorithm C4.5 [QUINLAN, 1993] for binary attributes. Induction starts from the whole dataset of examples and the first step is to find the logical conjunction that best splits the dataset in two. The two resulting groups have to be as divergent as possible. In other words, elements of one group have to be different from the ones of the other group and elements of the same group have to be similar to each other. This divergence is measured by a splitting criterion.

Let us take a look at Algorithm 7 to observe how a logical tree induction is performed.

Algorithm 7 Growing a logical decision tree using TILDE [BLOCKEEL, 1998]

```
1: procedure GROWTREE(E: example set, Q: query)
2:     ← Q_b := BESTSPLIT(ρ(← Q), E)
3:     if STOPCRITERIA(← new_query, example_set) then
4:         tree := LEAF(INFO(E))
5:     else
6:         conjunction := Q_b − Q
7:         E_yes := {e ∈ E| ← Q_b succeeds in e ∧ B}
8:         E_no := {e ∈ E| ← Q_b fails in e ∧ B}
9:         yes_child := GROWTREE(E_yes, Q_b)
10:        no_child := GROWTREE(E_no, Q)
11:        tree := NODE(conjunction, yes_child, no_child)
12:    return tree
```

The procedure in Algorithm 7 takes as input a set of examples and a query that is initialized to **true**. Afterwards, the algorithm uses a classical refinement operator ρ, which is defined as follows:

Definition 2 *A refinement operator ρ under θ-subsumption maps a clause c onto a set of clauses such that $\forall c' \in \rho(c)$, $c \leq_\theta c'$ (it reads "c θ-subsumes c'") [BLOCKEEL, 1998].*

In order to understand the concept of θ-subsumption, let us take a look at the following definition:

Definition 3 *A clause c θ-subsumes another clause c' ($c \leq_\theta c'$) if and only if there is a variable substitution θ such that $Lits(c\ \theta) \subseteq Lits(c')$, where $Lits(c)$ is the set of literals ocurring in clause c when it is written as a disjunction. [BLOCKEEL, 1998]*

For example, the clause $c : r(X,Y) \leftarrow s(X,Y)$ θ-subsumes the clauses $c' : r(X,Y) \leftarrow s(X,Y), t(X)$ and $c'' : r(X,a) \leftarrow s(X,a)$.

As $\rho(\leftarrow Q)$ returns a set of clauses that are θ-subsumed by $\leftarrow Q$, a splitting heuristic chooses among them the one ($\leftarrow Q_b$) that performs the best split. The

node is then assigned the conjunction $Q_b - Q$, which consists of the literals that were added to Q to obtain Q_b.

If the stop condition is not satisfied, the set of examples is divided into two subsets: the ones in which the query Q_b succeeds (subset E_{yes}) and the ones in which the query Q fails (subset E_{no}). The subsets E_{yes} and E_{no} are used respectively for growing the "yes" and "no" subtrees of the node.

This procedure continues recursively to lower levels until the stop condition is satisfied. That is when the function LEAF($INFO(E)$) returns a leaf node whose value equals the average of the values of the examples in E that have reached that leaf.

Specifying the basic refinement operator

A basic refinement operator has to be given "hints" on what literals that might be added to a query to create a new one. These hints are what is called *rmodes* inside the TILDE algorithm. These rmodes define the predicates that should be used in new atoms as well as their argument types. Here are some examples:

```
rmode(10:  aa(-Pos, -Amino))
rmode(10:  tiny(+Amino))
```

The predicate *aa* states that amino acid *Amino* is located in position *Pos*. The predicate *tiny* states that amino acid *Amino* is tiny. The sign '$-$' before a variable indicates that it is an output and **must not** have occured in the clause, whereas the sign '+' indicates that it is an input and **must** have occured in the clause. If a variable can be either an input or an output, then the sign '$+-$' should be used.

Based on the rmodes of the sample above, we can imagine the result of a refinement operator applied to $\leftarrow aa(Pos1, Amino1)$:

$$\begin{aligned}\rho(\leftarrow aa(Pos1, Amino1)) = \{ &\leftarrow aa(Pos1, Amino1), aa(Pos2, Amino2), \\ &\leftarrow aa(Pos1, Amino1), tiny(Amino1)\}\end{aligned} \tag{2.38}$$

Regression splitting heuristics

One important question about finding the best split for an example set is what criterion we will use to determine if one split is better than the other. Regression algorithms in general (e.g. CART [BREIMAN *et al.*, 1984] and SRT [KRAMER, 1996]) use the inverse of within-subset variance as the quality indicator of a split.

According to the techniques of analysis of variance (ANOVA), in case of a split of larger set into two or more subsets, the variation of the superset SS_T can be decomposed in two terms: the within-subset (or residual) variation SS_W and the between-subset (or explained) variation SS_B. This decomposition is called the *partition of sums of squares*.

Let y_i be the value of example i, $\hat{y}_i$ be the average of the group y_i belongs to and $\bar{y}$ be the total average. Thus, we have the following expression:

$$SS_T = SS_B + SS_W$$

$$\underbrace{\sum_{i=1}^{n}(y_i - \bar{y})^2}_{\text{Total variation } (SS_T)} = \underbrace{\sum_{i=1}^{n}(\hat{y}_i - \bar{y})^2}_{\text{Between-subset variation } (SS_B)} + \underbrace{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}_{\text{Within-subset variation } (SS_W)} \tag{2.39}$$

In other words, the within-subset variation evaluates how elements are dispersed from their respective subset average, whereas the between-subset variation evaluates how the subset averages are dispesed from the total average (in other words, how far apart the subsets are from each other). So, the best split must minimize the within-subset variation, which automatically implies to maximize of the between-subset variation, according to Equation 2.39. The feature the best split uses to separate one subset from the other must explain the differences between them and decrease the residual error within each subset, hence the alternative names *explained variation* and *residual variation.*

Tilde example

So as to clarify how Tilde works, let us see an example that illustrates the induction of logical decision tree.

First of all, let us consider a dataset of 6 examples. Each example has a set of facts that give information about the real number associated to it in format `value(real_number)` and about the amino acids in positions 0 and 1 in format `aa(position, amino_acid)`.

Example 1: `value(0.9), aa(0, s), aa(1, a)`
Example 2: `value(0.6), aa(0, s), aa(1, c)`
Example 3: `value(0.5), aa(0, s), aa(1, g)`
Example 4: `value(-0.6), aa(0, s), aa(1, r)`
Example 5: `value(-0.7), aa(0, s), aa(1, n)`
Example 6: `value(-0.8), aa(0, s), aa(1, d)`

Moreover, thare is a background knowledge represented by the substitution matrix BLOSUM62, which contains the scores of replacing one amino acid with another one. Here is a sample of these scores represented by a set of atoms in format *blosum62(amino_acid, amino_acid, score)*: `blosum62(a, a, 4), blosum62(a, r, -1), blosum62(a, n, -2), blosum62(a, d, -2), blosum62(a, c, 0), blosum62(a, g, 0)`

The regression tree grown by TILDE in this example is illustrated in Figure 2.24.

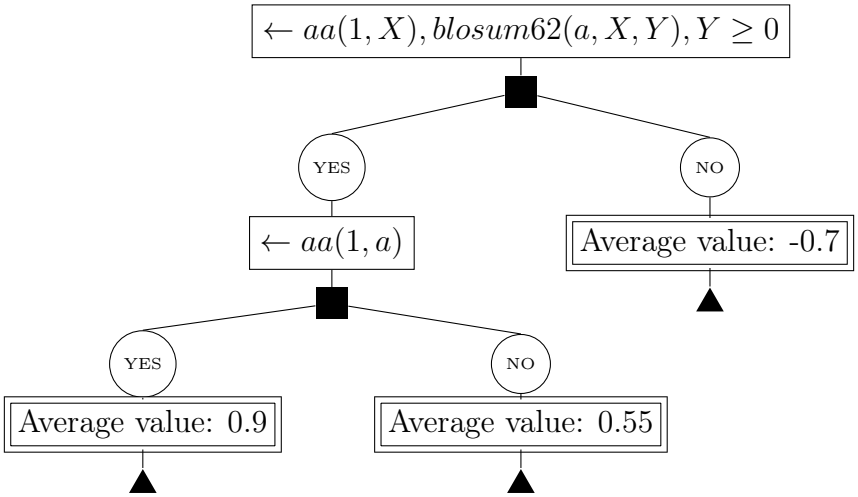


Figure 2.24: Regression tree grown by TILDE

To explain the growing procedure of the regression tree in Figure 2.24, let us consider that the stop condition is met when the standard deviation is lower than or equal to 0.1. Firstly, all examples are considered in order to determine the best split at the tree root. The query that performs the best split at the first level is $\leftarrow aa(1, X), blosum62(a, X, Y), Y \geq 0$. From now on, examples 1, 2 and 3 are associated to the YES node, while examples 4, 5 and 6 are associated to the NO node. In the NO node, the standard deviation is 0.1, which fulfills the stop condition. The value associated to this node is the average value of examples 4, 5 and 6. In the YES node, however, the standard deviation is approximately 0.2, so another split must be done to grow one more level of this branch. The best split at this branch is $\leftarrow aa(1, a)$ and separates example 1 from examples 2 and 3. In the YES-YES node, example 1 is the only one, which makes the standard deviation equal zero. In the YES-NO node, examples 2 and 3 also present a standard deviation below the minimum limit such that the stop condition is satisfied. The value associated to this node is the average value of examples 2 and 3. The induction is done once all branches have finished growing.

2.9 Measuring inference accuracy

In this section we will present two methods for measuring the inference accuracy on a set of test examples: the *Area Under a Receiver Operating Characteristic Curve* (AUC-ROC) and the *Area Under a Precision-Recall Curve* (AUC-PR). Both methods consist in summing up the area under a curve that is drawn using the accuracy scores of the test examples. However, each method plots its curve based on different cartesian coordinates.

2.9.1 Definitions

There are three definitions that must be known in order to understand how ROC and PR curves are plotted: *true positive rate* (or *recall*), *false positive rate* and *precision*. These definitions are shown below.

$$\text{true positive rate (or recall)} = \frac{\text{Correctly classified positives}}{\text{Total number of positives}} \tag{2.40}$$

$$\text{false positive rate} = \frac{\text{Incorrectly classified negatives}}{\text{Total number of negatives}} \tag{2.41}$$

$$\text{precision} = \frac{\text{Correctly classified positives}}{\text{Classified positives}} = \frac{\text{True positives}}{\text{True positives} + \text{False positives}} \tag{2.42}$$

2.9.2 AUC-ROC

The receiver operating characteristics (ROC) graph is a two-dimensional graph in which the *true positive rate* is plotted on the Y axis and the *false positive rate* is plotted on the X axis [FAWCETT, 2006].

A ROC curve has its points plotted according to the coordinates $(fp\ rate, tp\ rate)$ drawn from binary classifiers. In our case, we do not a have a group of binary classifiers; instead, we are dealing with a probabilistic classifier[2] that returns a different continuous score for every example submitted to its classification. Notwithstanding, a continuous classifier can generate a set of binary classifiers if we establish thresholds below which the examples are classified as negative and above or equal to which the examples are classified as positive. Thus, each threshold corresponds to a different "binary classifier" with its specific true and false positive rates.

The first step for tracing the thresholds in a continuous score set is to sort them in a descending score order, which means that better scores come first in the list [3]. Once all scores with its respective real classes (positive or negative) are sorted, the first threshold begins in $+\infty$, where $(fp\ rate, tp\ rate)$ is always $(0, 0)$. As there is no example with a higher score than $+\infty$, no example is tested positive according to the first threshold. From this point on, the next thresholds always follow the example scores in descending order. In the last threshold, which is the lowest example score,

[2]Note that it is normally said that a non-binary classifier is a probabilistic classifier, even though it does not necessarily return normalized probability values. Normalization is not mandatory for ROC curves [FAWCETT, 2006].

[3]In this present work better scores mean higher scores, because we only use Viterbi score to evaluate an inference. However, it would be the opposite if we used E-value scores, which are the more significant the lower they are.

the pair $(fp\ rate, tp\ rate)$ is always $(1, 1)$, because all examples are tested positive as they are all above or equal to the minimum score.

After the ROC curve is plotted, the area under this curve is used as a classification performance measure. Better ROC curves are closer to the northwest corner of the graph, having a larger area under them. If there is some point (or threshold) where the true positive rate is high and the false positive rate is low, this point represents a high accuracy.

Table 2.4 and figure 2.25 illustrate an example of ROC curve plotting.

Order #	**Class**	**Score**	**True positives**	**False positives**	**TP rate**	**FP rate**
1	**p**	0.9	1	0	0.2	0.0
2	**n**	0.8	1	1	0.2	0.2
3	**p**	0.7	2	1	0.4	0.2
4	**p**	0.65	3	1	0.6	0.2
5	**n**	0.6	3	2	0.6	0.4
6	**p**	0.5	4	2	0.8	0.4
7	**p**	0.4	5	2	1.0	0.4
8	**n**	0.3	5	3	1.0	0.6
9	**n**	0.2	5	4	1.0	0.8
10	**n**	0.1	5	5	1.0	1.0

Table 2.4: List of scores of 5 positive examples and 5 negative examples in descending order

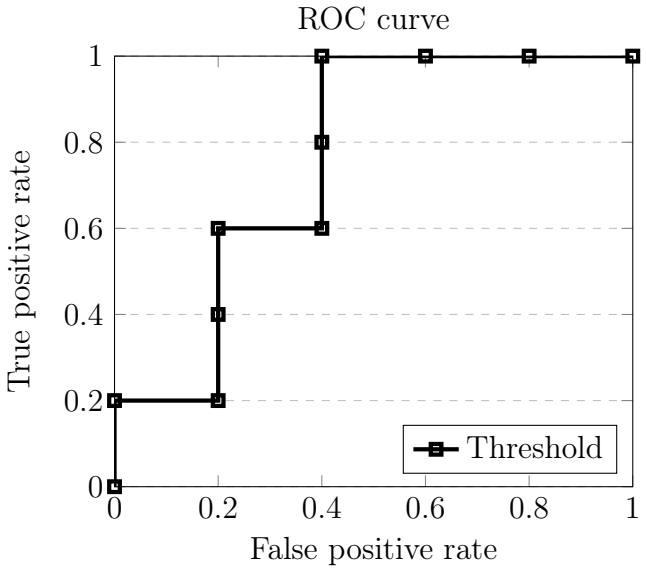


Figure 2.25: Example of ROC curve based on the scores listed in Table 2.4

2.9.3 AUC-PR

Precision-Recall curves are said to give a more informative picture of an algorithm's performance in case of highly unbalanced datasets. They are two-dimensional curves, where the precision rate is plotted on the Y axis and the recall rate is plotted on the X axis. The same procedure we used for plotting ROC curves can be similarly applied to PR curves. However, PR curves use a nonlinear interpolation between their points, which makes it more difficult to trace them and to calculate the area underneath in comparison with ROC curves. On example of PR curve is shown in Figure 2.26.

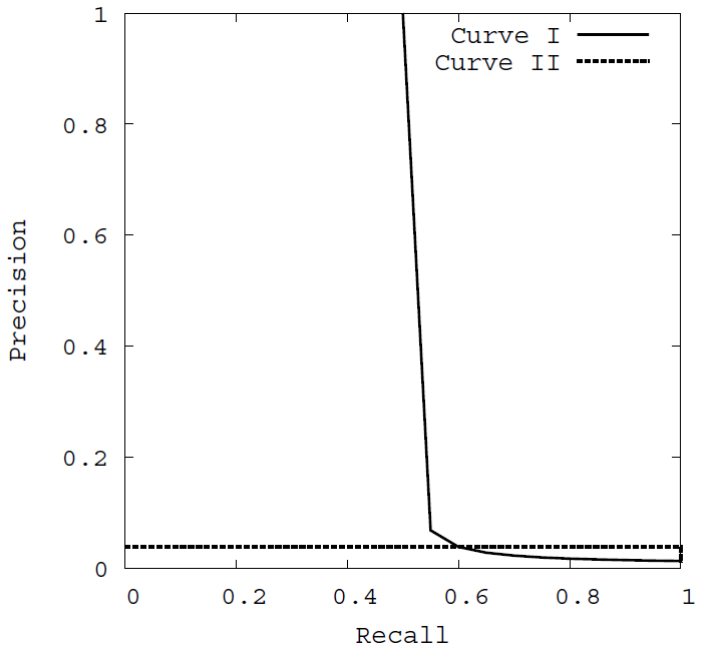


Figure 2.26: Example of PR curve [DAVIS and GOADRICH, 2006]

Nonetheless, it is demonstrated that a curve dominates in the PR space if and only if it also dominates in ROC space [DAVIS and GOADRICH, 2006]. This means that, despite yielding different results, both ROC and PR curves provide the same comparison between performances. Given the easiness of its use and its friendlier numbers, we chose to use the AUC-ROC method.

Chapter 3

Profile TildeCRF

In last chapter we covered background subjects that are fundamental to the understanding of this study. The main objective of this present chapter is to introduce *profile* TildeCRF, which is a new methodology in the field of protein homology detection and supervised sequential learning. Section 3.1 presents the issues in the conversion of profile HMM into profile TildeCRF. Then, Section 3.2 details the learning algorithm we implemented in our method. Lastly, Section 3.3 shows how we tested and evaluated the performance of our proposed method.

3.1 From profile HMM to profile TildeCRF

KINJO [2009] has shown the possibility of designing a CRF model that is analogue to profile HMM, inspiring us to implement profile TildeCRF and to test its effectiveness. To do so, we relied on the FSM design of profile HMM and on its Viterbi and Forward equations for statistical inference, in the same way as KINJO [2009] did. Regardless of the similarity between our graph structures and inference equations, our training procedures were considerably different: while KINJO [2009] used pre-defined biologically-based boolean feature functions and trained his model by optimizing the function weights (Section 2.5.4), we used logical regression trees as feature functions and trained them via gradient tree boosting (Section 2.6). Besides, our method implementation shed light to practical issues concerning the data structure to be used for dynamic programming, when combining profile HMM with TildeCRF. Also, we limited the scope of our method to the global alignment mode, due to the much higher complexity of local and glocal alignments. In this section, we intend to explain our roadmap for converting profile HMM to profile TildeCRF. We will also present a new trellis specifically designed for profile TildeCRF with the necessary adjustments.

The algorithms TreeCRF and TildeCRF use *linear-chain* CRF, which is a type of CRF whose graph consists of vertices and edges connected in a linear chain

as shown in Figure 2.22b. Linear-chain CRF applications use an acyclic graph called *trellis*, whereas profile HMM uses a cyclic finite state machine. Trying to fit profile HMM into linear-chain CRF may sound like 'forcing a square peg into a round hole'. On the one hand, linear-chain CRF is designed for supervised sequence labeling or *tagging*, strictly speaking: given the data sequence $\mathbf{x} = \langle x_0, \ldots, x_t, \ldots, x_T \rangle$ as the input, the output is a label sequence $\mathbf{y} = \langle y_0, \ldots, y_t, \ldots, y_T \rangle$ such that y_t is the label corresponding to x_t. On the other hand, profile HMM is specifically suited for inferring the alignment between an observed protein sequence $\mathbf{x}$ and a hidden state sequence $\mathbf{y}$. Although performing a protein alignment can be viewed as 'labeling' the protein sequence with the hidden states of the profile model, we cannot use here the same approach that is usually adopted for tagging problems due to the presence of *delete* states. To illustrate this problem, let us analyze the trellises in Figures 3.1 and 3.2.

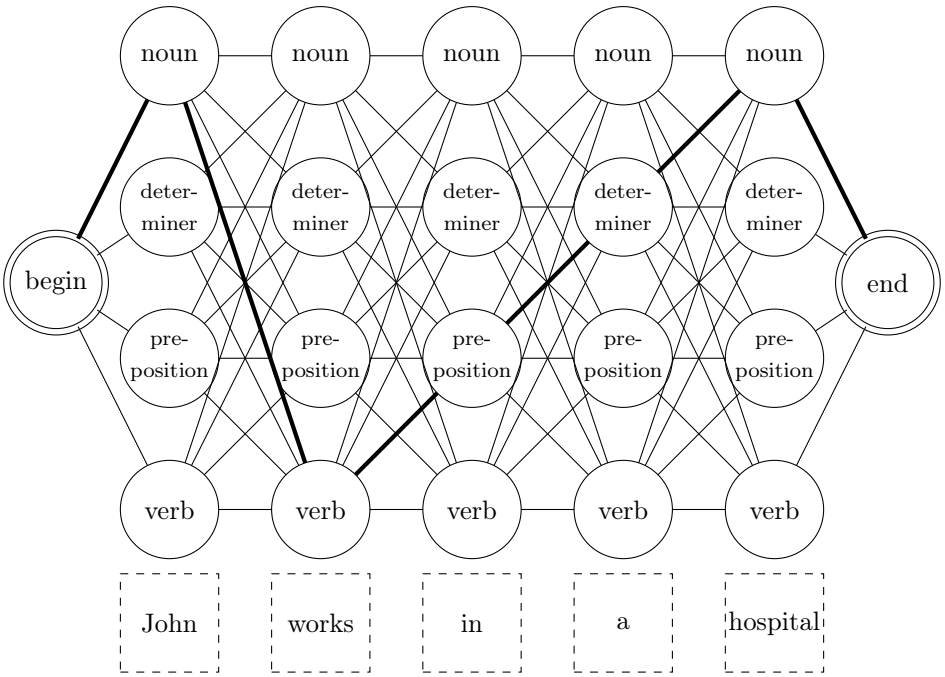


Figure 3.1: A trellis diagram representing a simple grammatical analyzer

In Figure 3.1, we present a simple tagging problem. Let $\mathcal{Y} = \{noun, determiner, preposition, verb\}$ be the set of possible labels that can be assigned to each word of the input data sequence $\mathbf{x} = \langle John, works, in, a, hospital \rangle$. The i-th column of the trellis corresponds to the possible choices for labeling the word x_i with a tag y_i from $\mathcal{Y}$. The length of the input sequence equals the number of columns in the trellis, as there is a fixed correspondence between the word x_i and the i-th column. The optimal output label sequence $\mathbf{y} = \langle noun, verb, preposition, determiner, noun \rangle$ has the same length as the input and so does any other non-optimal output.

However, a different situation is presented when we follow the *profile* model for labeling a protein sequence. To demonstrate this, let us take a look at Figure 3.2, where we use the states of a size 2 profile model as labels, adopting the same approach of usual sequence tagging applications. Let the protein sequence $\mathbf{x} = \langle A, C, D \rangle$ be the input observation sequence and let $\mathcal{Y} = \{I_0, M_1, D_1, I_1, M_2, D_2, I_2\}$ be the set of possible states (or labels). Similarly to the previous example, the i-th column corresponds to the possible options for associating the residue x_i with label y_i from $\mathcal{Y}$. The input sequence $\mathbf{x}$ is unaligned (it has no gaps) and unknown to the model.

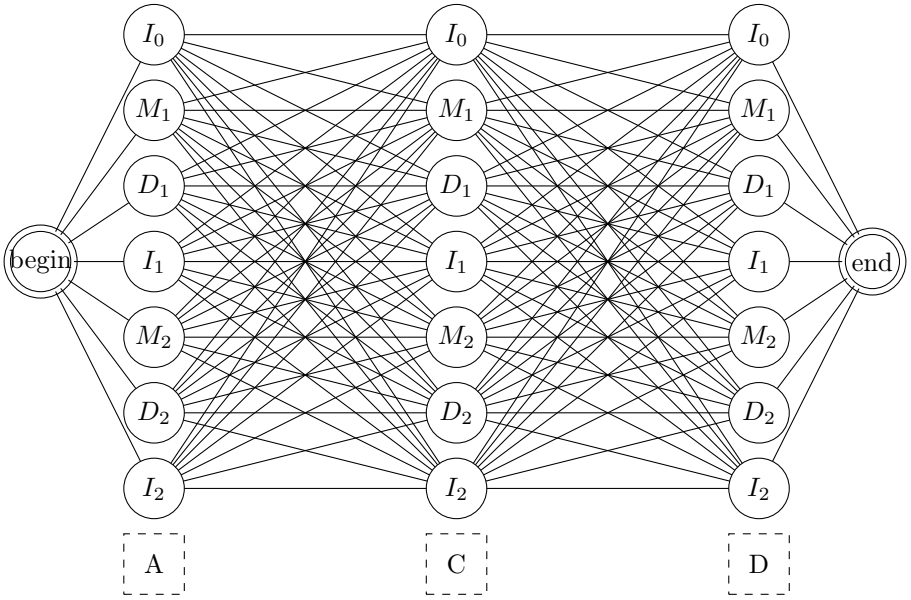
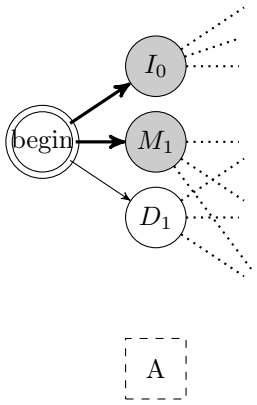


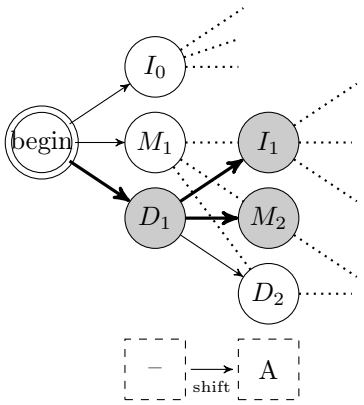
Figure 3.2: An attempt to use the states of a size 2 profile model in a tagging-based trellis.

Initially, the trellis of Figure 3.2 disregards the restrictions of a profile model, because not every possible connection is allowed and not every state is possible in every column. Let us take for example the first residue A of the input sequence $\mathbf{x}$. The trellis allows A to be labeled with any state, whereas I_0 and M_1 would actually be the only states allowed for A. It may be argued that transitions between certain states are forbidden and should therefore be assigned a null probability, which is indeed necessary, but not sufficient for representing a profile model. Specifically, the transition between B and D_1 is possible, so it may have a non-zero probability. Yet, if we choose D_1 for the first column, a gap will occupy the first position of the resulting aligned sequence and the residue A will be shifted to the right, as shown in Figure 3.3. Thus, the i-th element of sequence $\mathbf{x}$ is not really fixed in the i-th column of the trellis unlike in standard sequence tagging. Actually, in alignment inferences that use sequence labeling, the output label sequence $\mathbf{y}$ has the same length as the aligned version of the input observation sequence $\mathbf{x}$, which may contain gaps and be bigger than the original unaligned version of $\mathbf{x}$. To sum it up,

input and output sequences do not necessarily have equal lengths in a profile-based alignment inference, contrarily to the sequence labeling problems that linear-chain CRFs usually address.



(a) I_0 or M_1 is chosen for column 1.



(b) D_1 is chosen for column 1, and I_1 or M_2 is chosen for column 2. Residue A is shifted right.

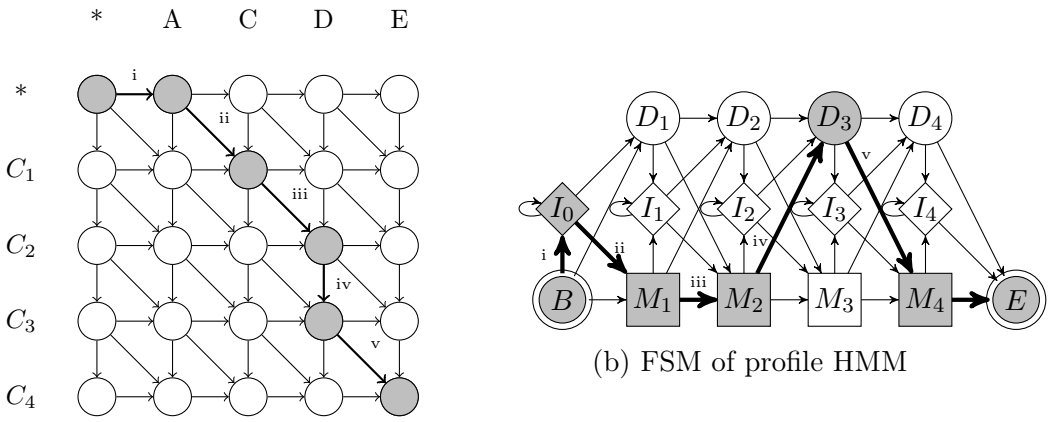
Figure 3.3: Consequences for chosing each of the possible states for the first column of the trellis of Figure 3.2.

In sight of this apparent incompatibility between profile HMMs and linear-chain CRFs, it is necessary to implement a trellis that is capable of mapping all and only the possible alignments between a protein sequence and a profile state sequence. For this purpose, we resorted to the *directed acyclic graph* (or DAG) that we use for pairwise alignment (Section 2.3.1). This specific DAG has two important features:

- It helps us visualize the conversion of profile HMMs to profile CRFs as it serves as a "bridge" between the two models;
- It makes it easier to implement the data structure for dynamic programming.

The two-dimensional DAG is very important in the way that it is analogue to both profile HMM finite state machine (or FSM) and CRF trellis. Firstly, we will use the analogy between the DAG and profile HMMs to unify these models and, then, we will turn the unified model into a trellis.

Aligning an unknown sequence to a model is the same as aligning it to the consensus positions of a profile HMM. The sequence that represents these consensus positions is called a *query* or *consensus* sequence. Let us reuse the DAG structure from the spell checker example (see Section 2.3.1), but let us replace the mistyped and the correct words with the amino acid sequence ACDE and the consensus positions C_i, $i \in \{1, 2, 3, 4\}$, respectively, as shown in Figure 3.4a.



(a) DAG structure

(b) FSM of profile HMM

Figure 3.4: A path traced in the DAG structure corresponds to a path traced in the profile HMM

The analogy between the DAG structure and the FSM of profile HMM is justifiable, as there is a one-to-one correspondence between the sets of possible paths in these two models, as shows the example in Figure 3.4. A diagonal move in the DAG corresponds to a transition to a match state in profile HMM, a vertical move corresponds to a transition to a delete state and a horizontal move corresponds to a transition to an insert state. Since each move in the DAG represents a move in the FSM, an entire path can be 'translated' from one representation to another.

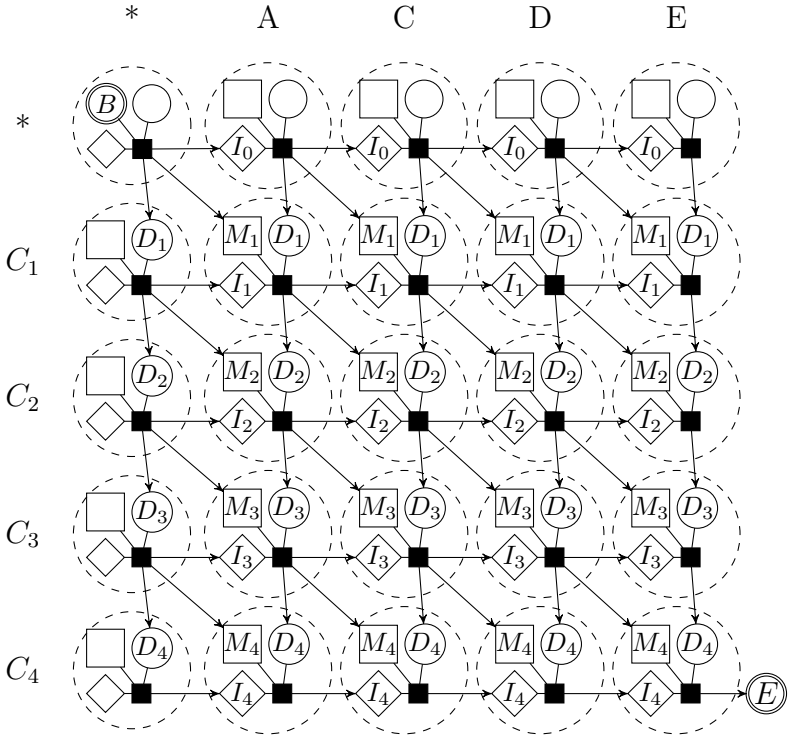
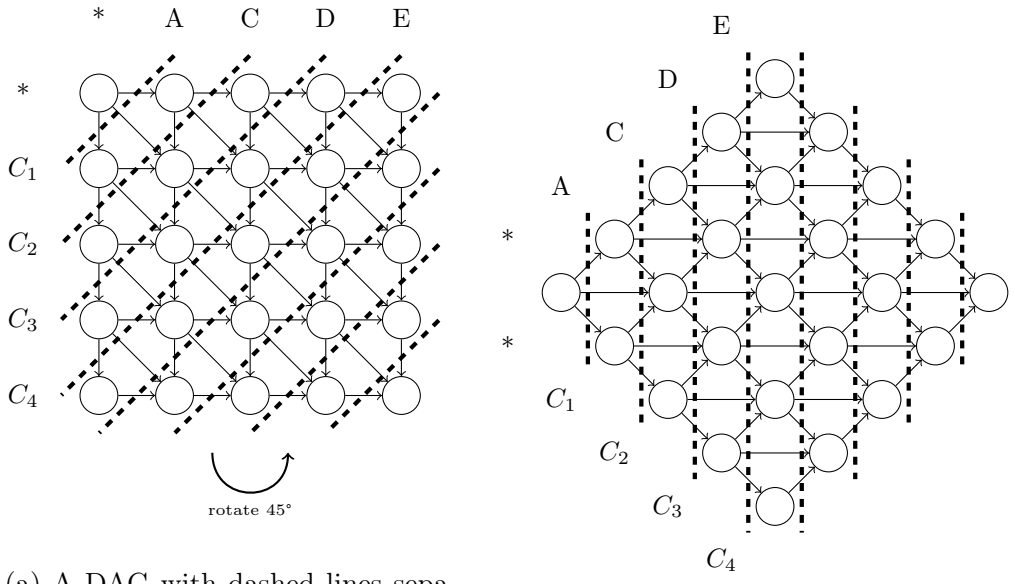


Figure 3.5: DAG unmerged: profile states are visible inside DAG. Each node of the DAG of Figure 3.4a is divided into three profile states with the same index.

Although paths in profile HMM and in the DAG are related one-to-one, each node in the DAG of Figure 3.4a has to be 'unmerged' in such a way that we can distinguish the profile states inside the DAG and explicit the correspondence between a move in this graph and its respective transition in the profile model (see Figure 3.5). In fact, apart from the input data sequence $\mathbf{x}$, CRF potential functions shall also take as input the specific states y_{t-1} and y_t of a given transition.



(a) A DAG with dashed lines separating the nodes in diagonals. Detailed version in Figure 3.5

(b) Rotating the DAG 45° counterclockwise. Detailed version in Figure 3.7

Figure 3.6: The trellis of profile TildeCRF results from the counterclockwise rotation of the DAG by 45°. DAG diagonals turn into trellis columns. The merged trellis is used for clarity.

Once the profile states are made visible inside the DAG strutucture (Figure 3.5), we create the trellis of profile TildeCRF by rotating the DAG 45° counterclockwise, as shown in Figure 3.6. By doing so, we turn the diagonals of the DAG into the columns of the new trellis. The final result can be seen in Figure 3.7. One adjustment had to be made, though: transitions to match states ought to have an intermediate node which acts like a mirror of the following match state. These auxiliary states M_i' are linked to their respective match states M_i by edges with probability 1 and they are represented by dotted rectangles in Figure 3.7. This adjustment is of utmost importance for the scaled forward-backward algorithm (see Section 2.7.3). Actually, the scaled algorithm should have the same final results as the standard forward-backward algorithm when calculating $P(y_{t-1} = u, y_t = v|\mathbf{x})$ (Equations 2.33), which is the probability of the transition from state u in position $t-1$ to state v in position t, given the observation sequence $\mathbf{x}$. However, the scaled algorithm demands that each trellis column should be 'visited' by all possible paths in order to normalize the α (forward) and β (backward) values column-wise. Without intermediate match

states, transitions to match states necessarily skip a column and the results returned by the scaled version differ from those returned by the standard version, as we have empirically observed. In contrast, with intermediate match states, every possible path has to 'visit' every column in the trellis and the results from both algorithms become equivalent, as desired.

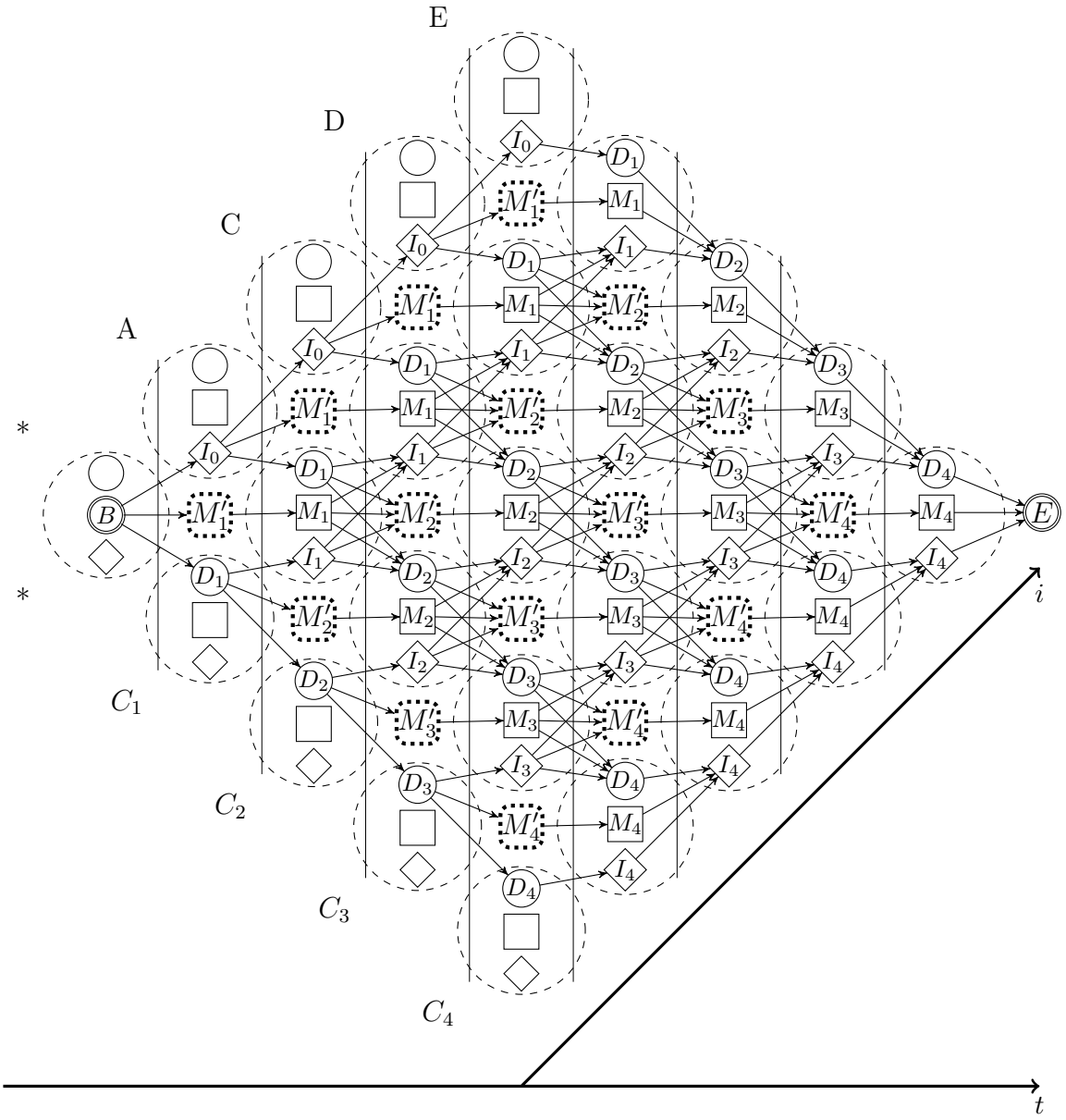
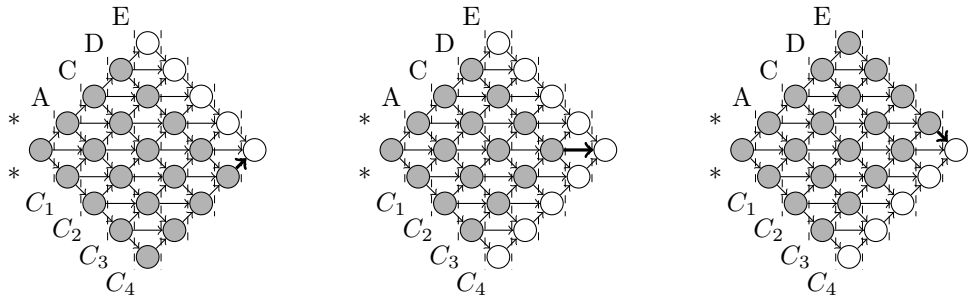


Figure 3.7: Trellis implementation in profile TildeCRF: it combines DAG with profile HMM and it includes intermediate match states. Indexes t and i for state and observation sequences $\mathbf{y}$ and $\mathbf{x}$ vary accordingly to their respective axes.

We observe that the trellis in Figure 3.7 is very different from the one we initially modeled in Figure 3.2. Interestingly, the new trellis is similar to what we would have had, had we finished mapping all the possible paths of Figure 3.3.

Besides aiding the conversion of profile HMM to profile TildeCRF, another ad-

vantage of the DAG is that it makes it easier to implement dynamic programming, once it satisfies the condition that solution of the problem depends on the solutions of its subproblems. If we consider a two-dimensional grid, the bigger problem would involve the whole matrix and its subproblems would be the three possible submatrices that are adjacent to the last node (Figure 3.8). Sequently, the same perspective is applied to each submatrix in a recursive way.



(a) Subproblem at the last *insert* state.

(b) Subproblem at the last *match* state.

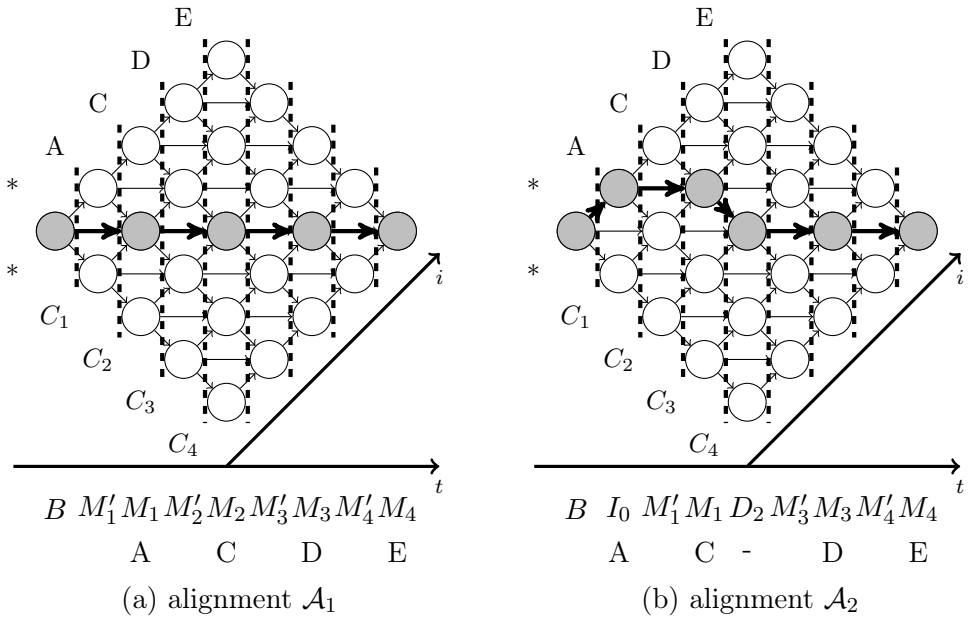
(c) Subproblem at the last *delete* state.

Figure 3.8: The three subproblems to be solved before the final solution. The same approach is then applied to each subproblem recursively.

Another issue in converting profile HMM to profile TildeCRF concerns the indexes of both observation sequence $\mathbf{x}$ and label sequence $\mathbf{y}$. A sequence index works like a cursor pointing to a specific position in the sequence. In sequence tagging applications as the one in Figure 3.1, the sequence index t serves for both sequences $\mathbf{x}$ and $\mathbf{y}$ in the resulting set of pairs $\mathcal{A} = \{(x_0, y_0), \ldots, (x_t, y_t), \ldots, (x_T, y_T)\}$ between data sequence $\mathbf{x} = \langle x_0, \ldots, x_n \rangle$ and label sequence $\mathbf{y} = \langle y_0, \ldots, y_n \rangle$, where y_t is the label corresponding to x_t. In this case, the trellis is organized in such a way that each element x_t is related to a specific trellis column t, which contains the possible values of y_t. Nonetheless, alignment problems present a different behavior as to the relationship between $\mathbf{x}$ and $\mathbf{y}$. There must be two separate indexes for inferring an alignment: one index i that refers to a specific element in $\mathbf{x}$ and another index t that refers to a specific element in $\mathbf{y}$ (see Section 2.3.3). The decoupling of these two indexes is necessary, because an observation x_i may appear at different positions of the resulting alignment $\mathcal{A}$. As we stated earlier in this section, one should keep in mind that, in structures suited for alignment inference, positions in the input sequence $\mathbf{x}$ are not fixed, as gaps may be repeatedly inserted by *delete* states during the inference procedure.

For example, let us observe Figure 3.9, which contains two trellises with a different path in each. Both of these paths have the same transition between columns 7 and 9, in which the edge goes from state M_3 emitting residue D to state M_4 emitting residue E. Before this transition, the two paths are different from each

other. In Figure 3.9a, amino acids A and C are assigned to columns 3 and 5, resulting the alignment $\mathcal{A}_1 = \langle(\mathrm{A}, M_1), (\mathrm{C}, M_2), (\mathrm{D}, M_3), (\mathrm{E}, M_4)\rangle$. In Figure 3.9b, the same amino acids A and C are assigned to columns 2 and 4, resulting the alignment $\mathcal{A}_2 = \langle(\mathrm{A}, I_0), (\mathrm{C}, M_1), (\text{-}, D_2), (\mathrm{D}, M_3), (\mathrm{E}, M_4)\rangle$. Although the trellis and the initial input sequence remain unchanged from one case to another in Figure 3.9, the way the amino acids are assigned to the trellis columns and their positions along the alignment depend on the path taken along the trellis. If we take for instance the element $x_2 = \mathrm{D}$ of the input sequence $\mathbf{x}$, we notice that it is related to the label $y_2 = M_3$ in $\mathcal{A}_1$, but it is also related to the label $y_3 = M_3$ in $\mathcal{A}_2$. This might seem a trivial implementation issue, but it affects the way we create our window function as we will see next.



(a) alignment $\mathcal{A}_1$ (b) alignment $\mathcal{A}_2$

Figure 3.9: Differences in the observation $\mathbf{x}$ depending on the path taken on the graph. This figure uses the merged version of the trellis for clarity.

In order to calculate the score of a specific transition, we must recall from Section 2.6.2 the CRF potential function $F^{y_t}(y_{t-1}, w_t(\mathbf{x}))$, where y_t is the current state, y_{t-1} is the previous state and $w_t(\mathbf{x})$ is a fixed-length window of input sequence $\mathbf{x}$ centered in t. In linear-chain CRF, we define w_t as a function of the input sequence $\mathbf{x}$, the position t and the window length L, as follows:

$$w_t(\mathbf{x}) = \langle x_{t-\lfloor L/2 \rfloor}, \ldots, x_t, \ldots, x_{t+\lfloor L/2 \rfloor} \rangle \tag{3.1}$$

Nevertheless, it is not the case for profile CRF, because the index i of observation sequence $\mathbf{x}$ is decoupled from index t of label sequence $\mathbf{y}$ so the window w_i should be centered in i. There are two options considered for obtaining the window w_i. The first option consists in applying the window function seen in Equation 3.1 to

the unaligned (gapless) sequence $\mathbf{x}$, replacing t with i. Using the first option with a window length of 3 in the example of Figure 3.9, we have $w_2(\mathbf{x}) = \langle C, D, E \rangle$ for both trellises. The second option consists in using the GetBestPath function presented in Algorithm 8 in order to get the window of the best path that visits both states y_{t-1} and y_t, where the residue x_i is emitted by state y_t. This function depends not only on i, $\mathbf{x}$ and L, but also on the states y_{t-1} and y_t that take part in the transition as well, and it has to be executed after the Viterbi scores of all trellis nodes are calculated, otherwise the information on the best paths will not be known beforehand. Using this second option with a window length of 3 in the example of Figure 3.9, we have $w_2(y_1 = M_2, y_2 = M_3, \mathbf{x}) = \langle C, D, E \rangle$ for alignment $\mathcal{A}_1$ and $w_2(y_2 = D_2, y_3 = M_3, \mathbf{x}) = \langle -, D, E \rangle$ for alignment $\mathcal{A}_2$. For our experiments, we selected this second option, because it yielded slightly better results.

Algorithm 8 Get the best path in which both states y_{t-1} and y_t are visited and in which the residue x_i is emitted by state y_t. This function is called by Algorithm 11.

```
1: procedure GetBestPath(y_{t-1}: previous state, y_t: current state, i: residue position, L: window length)
2:     node ← GetNodeByStateAndResiduePosition(y_t, i)
3:                 ▷ get the trellis node corresponding to the state y_t for residue x_i
4:     prevNode ← node.GetPreviousNodeByState(y_{t-1})
5:                                   ▷ get the previous node for state y_{t-1}
6:                                                 ▷ AA stands for 'amino acid'
7:     bestForwardPath ← empty list
8:     for i ← 1 to ⌊L/2⌋ do                                  ▷ build best forward path
9:         bestForwardPath ← Concatenate(prevNode.AA, bestForwardPath)
10:        prevNode ← prevNode.bestPreviousNode
11:    bestBackwardPath ← empty list
12:    for i ← 0 to ⌊L/2⌋ do                                  ▷ build best backward path
13:        bestBackwardPath ← Concatenate(bestBackwardPath, node.AA)
14:        node ← node.bestNextNode
15:    bestPath ← Concatenate(bestForwardPath, bestBackwardPath)
16:    return bestPath
```

As to the statistical inference in our method, we had recourse to the Viterbi and forward-backward expressions of Sections 2.7.1 and 2.7.2, keeping index i for residue $x_i \in \mathbf{x}$, and index j for S_j, where $S \in \{M, D, I\}$, similarly to what was done by KINJO [2009]. However, in order to explicitly establish the relationship between x_i, y_t and S_j, we adjusted these equations to include index t for state $y_t \in \mathbf{y}$ as shown in the Viterbi expressions[1] of Equation 3.2.

[1]The adjustments to the forward-backward expressions are analogous.

$$V_j^M(t,i) = \max \begin{cases} V_{j-1}^M(t-1,i-1) + f(M_{j-1}, M_j, \mathbf{x}, i), \\ V_{j-1}^I(t-1,i-1) + f(I_{j-1}, M_j, \mathbf{x}, i), \\ V_{j-1}^D(t-1,i-1) + f(D_{j-1}, M_j, \mathbf{x}, i) \end{cases}$$

$$V_j^I(t,i) = \max \begin{cases} V_j^M(t-1,i-1) + f(M_j, I_j, \mathbf{x}, i), \\ V_j^I(t-1,i-1) + f(I_j, I_j, \mathbf{x}, i), \\ V_j^D(t-1,i-1) + f(D_j, I_j, \mathbf{x}, i) \end{cases} \tag{3.2}$$

$$V_j^D(t,i) = \max \begin{cases} V_{j-1}^M(t-1,i) + f(M_{j-1}, D_j, \mathbf{x}, i), \\ V_{j-1}^I(t-1,i) + f(I_{j-1}, D_j, \mathbf{x}, i), \\ V_{j-1}^D(t-1,i) + f(D_{j-1}, D_j, \mathbf{x}, i) \end{cases}$$

The f function is a bit different from those in standard CRF, as it uses the new window function $w_i(y_{t-1}, y_t, \mathbf{x})$ in this case, as follows:

$$f(y_{t-1}, y_t, \mathbf{x}, i) = F^{y_t}(y_{t-1}, w_i(y_{t-1}, y_t, \mathbf{x})) \tag{3.3}$$

3.2 Learning algorithm

After detailing the issues of applying the concepts of profile HMM in CRF, we may now present the learning algorithm of profile TildeCRF.

3.2.1 Overview of the algorithm

The intent of this section is to present an overview of profile TildeCRF (see Algorithm 9) in a high level to make it easier to understand the most important steps of the training stage.

The original TildeCRF algorithm was implemented by GUTMANN and KERSTING [2006] in Java and in Prolog, which is a logic programming language. Nevertheless, due to the specificities of the profile trellis diagram and the high execution cost of this algorithm, we chose to use Cython, which is a pre-compiled version of the language Python. This choice is based on the simplicity provided by Python and on the higher execution performance provided by Cython. As a disadvantage, our implementation used a simplified version of logic predicate unification,

Algorithm 9 Profile TILDECRF overview

```
1: Load training information                                   ▷ see Section 3.2.2
2: training_set ← superfamily_set − target_family_set
3: test_set ← target_family_set + sample_of_negative_examples
4:
5: for iteration i ← 0 to N do
6:     Select batch from training_set                          ▷ see Section 3.2.3
7:     for each sequence in batch do
8:         Build trellis data structure                        ▷ see Section 3.2.4
9:         Trace the golden path                               ▷ see Section 3.2.5
10:        Calculating potential function values for each edge ▷ see Section 3.2.6
11:        Performing gradient boosting                        ▷ see Section 3.2.7
12:    Grow first-order logical regression trees               ▷ see Section 3.2.8
13:
14: for each test_sequence in test_set do
15:    Perform inference from test_sequence                    ▷ see Section 3.3
16:
17: Evaluate prediction performance                            ▷ see Section 3.3
```

which only contemplates the substitution of variables for constants and does not deal with functions inside logical atoms. Since there was no need to use functions for logical representation in our problem, this particular drawback did not seem to compromise our methodology. For storing generated logical examples, SQLite 3.0 was used because it is a serverless database management system and it works well in a local environment.

In the next subsections we will see the details for each command line presented in Algorithm 9.

3.2.2 Loading training information

In the beginning of of the training procedure, it is necessary to load the data set containing the training and test examples. For remote protein homology detection, we implemented the *leave-one-family-out* strategy, where proteins from a superfamily are used for learning, except for the ones belonging to one target family, which will later be used as positive test examples. On the one hand, this strategy ensures that a distant homology relationship is kept between the training sequences and the test sequences. On the other hand, this constraint makes it more difficult to attain good accuracy in predition, as similarities between training and test sequences become lower.

The training set consists of previously aligned sequences from the same superfamily as the target family, which is excluded during the training stage. The previous multiple alignment of the training set is performed by another program called

Clustal Omega [SIEVERS *et al.*, 2011]. The test set includes the sequences from the target family and also negative sequences, which belong to other superfamilies. It is worth to mention that we do not use negative training examples during the learning stage of profile TildeCRF. Actually, since profile TildeCRF executes a sequence labeling procedure, it considers the negative training examples to be those alignments that do not match the previously aligned sequence. In other words, negative training examples correspond to the paths along the trellis that do not follow what we will later call the *golden path* in Section 3.2.5.

3.2.3 Selecting the training information

Once the training information loading is done, it is time to begin the learning procedure with the sequences in the training set. The learning procedure can be repeated for N iterations, in each of which a new regression tree is grown for every potential function in the CRF structure. We will see this growing procedure in more details later in Section 3.2.8. Each iteration may use either the whole training set or just a subset, which we call a *batch*. The advantage of executing the learning procedure in batches is the lower execution cost per iteration, leading to similar results at the end. In contrast, using the whole training set in the same iteration may lead to RAM memory overflow, because each additional training sequence demands a new trellis data structure that generates a new set of logical examples. A batch can be a random sample from the training set, but it is interesting to have all training set covered by the end of the learning stage.

3.2.4 Building trellis data structure

For each sequence in the selected batch of training sequences, a new trellis data structure is built to make it possible to calculate the transition scores and the gradient values. The data structure in profile TildeCRF follows the representation illustrated in Figure 3.7. In the program the trellis is composed of two types of object: *nodes* and *edges*. These objects are detailed in Table 3.1.

Object	**What it represents**	**What information it contains**
Nodes	States/labels	• amino acid assigned to the node • previous and following edges • forward-backward scores up to the node • the best forward-backward paths that visit the node
Edges	Transitions between nodes	• nodes in the endpoints of the edge • potential function value

Table 3.1: Objects used in the trellis data structure

The dimension of a trellis structure depend mainly on the length of the gapless amino acid sequence L_{aa} and on the number of consensus positions L_{cons}. The number of trellis columns M can be derived directly from these parameters, as $M = L_{aa} + L_{cons} + 1$ (the first node is not related to any amino acid or any consensus position). Therefore, the process of constructing a trellis runs in loop for each of the M columns, creating node and edge objects and linking them together.

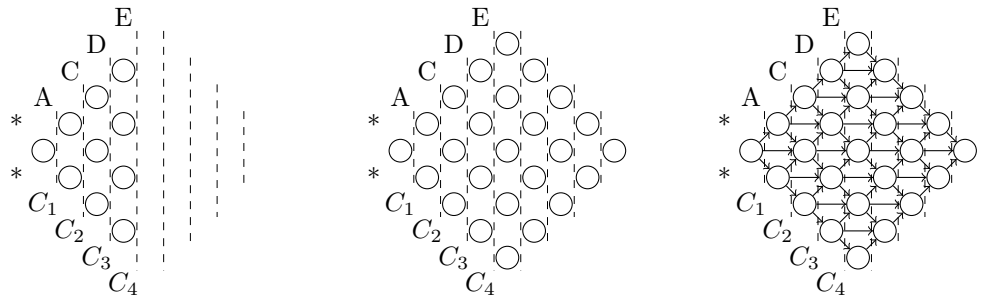


Figure 3.10: Building the trellis diagram, by creating the nodes and then linking them together with edges

3.2.5 Tracing the golden path

Profile TildeCRF is a supervised learning method, which means that the training protein sequences must be labeled and aligned beforehand so that the model can adapt to the desired classification. To have the pre-defined labels before training, both the aligned training sequences and the consensus positions must be captured by other programs that are specialized in multiple alignment of protein sequences, such as Clustal Omega [SIEVERS *et al.*, 2011]. In fact, we use this information to get the labels and, consequently, the desired alignment between sequences $\mathbf{x}$ and $\mathbf{y}$. We call *golden path* the path in the trellis traced by the desired alignment and we call *golden edges* the edges that compose the golden path. The golden path serves to signalize the desired transitions that we wish the model to adapt to. By calculating the gradient ascent in each iteration, the training procedure shortens the distance between the model results and the desired results represented by the golden path. Details on how to calculate the gradient ascent will be seen in Section 3.2.7.

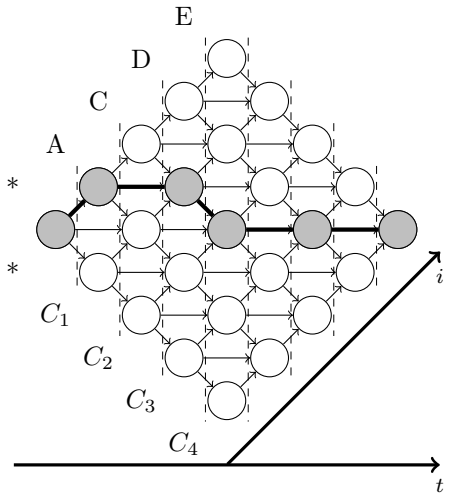
In order to translate the aligned sequence and the consensus positions into a golden path along the trellis, we execute the procedure illustrated in Algorithm 10. For example, let us say that the input aligned sequence is "AC-DE" and the marked consensus positions are represented by "01111", where a '1' in position i indicates that the i-th amino acid belongs to the consensus and a '0' indicates the contrary. So, each position i in both sequences is analyzed as follows:

- If the position i refers to an amino acid in the aligned sequence and to a '1' in the consensus position, then it is a **match**.

- If the position i refers to an amino acid in the aligned sequence and to a '0' in the consensus position, then it is an **insertion**.
- If the position i refers to a gap in the aligned sequence and to a '1' in the consensus position, then it is a **deletion**.
- If the position i refers to a gap in the aligned sequence and to a '0' in the consensus position, then no state is assigned.

Algorithm 10 Set the golden path in trellis based on an aligned sequence

```
1: procedure SetGoldenPath(trellis, alignedSequence, consensusPositions)
2:     node ← trellis.firstNode
3:     for position i ← 1 to N do
4:         if consensusPositions[i] = 1 then
5:             if alignedSequence[i] ≠ '−' then
6:                 nextEdge ← node.nextMatchEdge
7:             else
8:                 nextEdge ← node.nextDeleteEdge
9:         else if alignedSequence[i] ≠ '−' then
10:            nextEdge ← node.nextInsertEdge
11:        else
12:            nextEdge ← null
13:        if nextEdge is not null then
14:            nextEdge.isGolden ← true
15:            node ← nextEdge.nextNode
```



Aligned x	Consensus	State
		B
A	0	I_0
C	1	M_1
-	1	D_2
D	1	M_3
E	1	M_4

Figure 3.11: Creating the golden path: $Begin \rightarrow I_0 \rightarrow M_1 \rightarrow D_2 \rightarrow M_3 \rightarrow M_4$

Therefore, the information given by "AC-DE" and "01111" indicates that the golden path is: $Begin \rightarrow I_0 \rightarrow M_1 \rightarrow D_2 \rightarrow M_3 \rightarrow M_4$ (see Figure 3.11).

3.2.6 Calculating potential function values for each edge

After the trellis data structure is built and the golden path is traced, the next step is to calculate the value returned by the potential functions for each edge in the trellis, as we shown in Algorithm 11. These potential functions are calculated as a sum of the results that the logical regression trees return. In case it is the first iteration of the training stage loop in line 5 of Algorithm 9, no regression trees have been grown yet and initial values are used instead, as we will see later on. The potential functions are expressed as $F^{y_t}(y_{t-1}, w_t(\mathbf{x}))$ (see Equation 3.4), where y_t is the state of the current node, y_{t-1} is the state of the previous node and $w_t(\mathbf{x})$ is a fixed-length window of the observation sequence $\mathbf{x}$. There is one different potential function for each current state and that explains why the function F is indexed with y_t.

Algorithm 11 Calculate potential function value for trellis transitions

```
1: procedure CalculateTrellis(trellis, F: array of potential functions, M: # of iterations,
   N_w: window length)
2:     for iteration ← 1 to M do
3:         for columnIndex ← 1 to T do                                   ▷ Forward step
4:             for each node n in trellis[columnIndex] do
5:                 for each previous edge e of n do
6:                     path ← GetBestPath(n.state, e.prev_node.state, n.position, N_w)
7:                                          ▷ for details on GetBestPath see Algorithm 8
8:                     e.value ← F[n.state](e.prev_node.state, path)
9:                     score ← e.prev_node.forward_path_score * exp(e.value)
10:                    n.forward_score ← n.forward_score + score
11:                    if e.prev_node.best_forward_path + e is the best forward path then
12:                        n.best_forward_path ← e.prev_node.best_forward_path + e
13:                        n.best_forward_score ← e.prev_node.best_forward_score + e.value
14:
15:        for columnIndex ← M − 1 down to 0 do                          ▷ Backward step
16:            for each node n in trellis[columnIndex] do
17:                for each following edge e in n do
18:                    score ← exp(e.value) * e.next_node.backward_score
19:                    node.backward_score ← node.backward_score + score
20:                    if e + e.next_node.best_backward_path is the best backward path then
21:                        n.best_backward_path ← e + e.next_node.best_backward_path
22:                        n.best_backward_score ← e.value+e.next_node.best_backward_score
```

The **forward step** in Algorithm 11 calculates the potential function values for every edge from the first column to the last one in the trellis. During this step, only the first half of the best path window is returned by the window function $w_t(y_{t-1}, y_t, \mathbf{x})$, which is represented here by the GetBestPath function (see Algorithm 8). The forward scores and the best forward path, which is the first half of the best path window, are also calculated and stored inside the node objects. The **background step** is analogous to the forward step, but it starts at the last node and ends at the first one. The values that have been calculated for every edge and node during the forward step are then reused. Information such as backward score and best backward path, which is the second half of the best path window, is

also stored inside the node objects. If the number of iterations M is greater than 1, then the forward and backward steps are repeated, with the advantage that the GetBestPath function can now return the two halves of the best path window. In fact, it is only after the first iteration that the best backward paths become available for every node.

Initial potential function values

Using the gradient boosting method, we will create potential functions out of the sum of smaller functions, which will be represented by logical regression trees in this case:

$$F^{y_t}(y_{t-1}, w_t(\mathbf{x})) = F_0^{y_t}(y_{t-1}, w_t(\mathbf{x})) + \Delta_1^{y_t}(y_{t-1}, w_t(\mathbf{x})) + \cdots + \Delta_N^{y_t}(y_{t-1}, w_t(\mathbf{x})) \quad (3.4)$$

The function $\Delta_i^{y_t}$ is the regression tree grown in iteration i based on gradient values for current state y_t. However, the function $F_0^{y_t}$ returns an initial value, even before any gradient boosting iteration is done. Two assumptions may be made for this initial function: a function that returns 0 for every input; or a function that returns the transition and emission log probabilities of the equivalent profile HMM, which are extracted from a pre-defined profile HMM model using the program HMMER [EDDY, 1998].

The profile HMM initial function would be:

$$F_0^{y_t}(y_{t-1}, w_t(\mathbf{x})) = \log(a_{y_{t-1}y_t} \times e_{y_t}(x_t)) \quad (3.5)$$

where $a_{y_{t-1}y_t}$ is the profile HMM transition probability from state y_{t-1} to state y_t and $e_{y_t}(x_t)$ is the profile HMM emission probability of amino acid x_t at state y_t.

3.2.7 Performing gradient boosting

In last subsection we saw that the potential function results are calculated for every transition as well as the forward-backward scores. These values are extremely important for calculating the function gradient value of every transition.

In profile TildeCRF, the training procedure uses the gradient tree boosting algorithm in order to decrease the difference between the model results, which are obtained by inference, and the desired results, which are obtained by the previously aligned protein sequence.

Let us recall the gradient ascent expression, which was demonstrated in Equation 2.25:

$$\frac{\partial \log P(\mathbf{y}|\mathbf{x})}{\partial F^v(u, w_d(\mathbf{x}))} = I(y_{d-1} = u, y_d = v, \mathbf{x}) - P(y_{d-1} = u, y_d = v|\mathbf{x})$$

In order to calculate the expression above, two versions of the training sequence $\mathbf{x}$ are necessary: the previously aligned version, which is used as the golden model for supervised traning; and the unaligned version, which is used for statistical inference during the training stage. On the one hand, gradient tree boosting uses the previously aligned sequence $\mathbf{x}$ to determine the first term of its expression $I(y_{d-1} = u, y_d = v, \mathbf{x})$, which consists in a binary function that returns 1 if the transition between labels u and v really exists in positions $d-1$ and d of the previously aligned sequence and it returns 0, otherwise. Once we used the aligned sequence information in order to define the golden path in the trellis, we may say that the function I returns 1 if and only if the transition between $(y_{d-1} = u)$ and $(y_d = v)$ is represented by a golden edge. On the other hand, gradient tree boosting also demands that inference should be performed on training sequences in order to calculate the second term $P(y_{d-1} = u, y_d = v|\mathbf{x})$, which is the current probability[2] of the transition from state u in position $d-1$ in y to state v in position d in y, given the input sequence $\mathbf{x}$. Statistical inference is always made on unaligned input sequences, because it reflects the model's response to an unknown input.

Recalling the probability expression, we have:

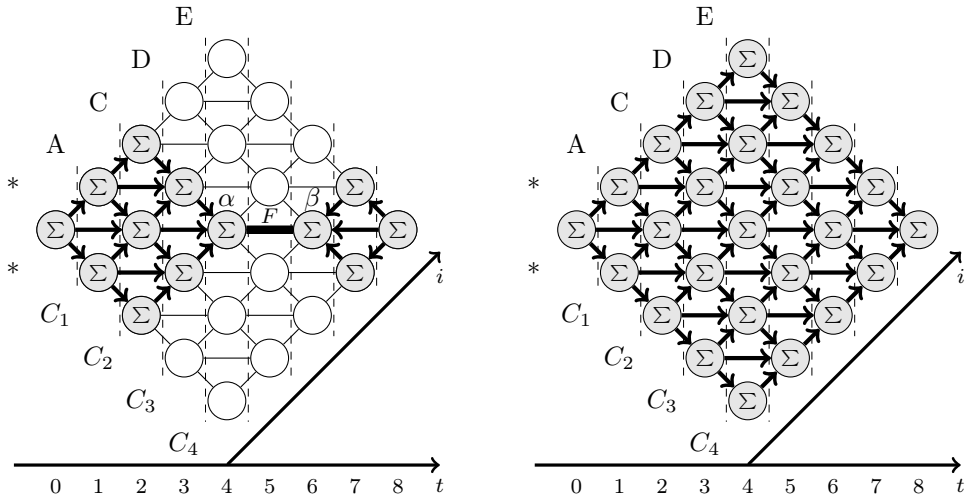
$$P(y_{d-1} = u, y_d = v|\mathbf{x}) = \frac{\alpha(u, d-1) \times \exp F^v(u, w_d(\mathbf{x})) \times \beta(v, d)}{Z(\mathbf{x})} \tag{3.6}$$

It is important to notice that in the current stage of the algorithm the necessary terms have already been calculated for every transition: the forward score $\alpha(u, d-1)$ of the previous node, the potential function $F^v(u, w_d(\mathbf{x}))$ of the edge, the backward score $\beta(v, d)$ of the current node and the normalization term $Z(\mathbf{x})$, which is the forward score obtained in the last node.

The probability expression in Equation 3.6 can be viewed as the ratio between the sum of the scores of all paths that visit $(y_{d-1} = u)$ and $(y_d = v)$ and the sum of the scores of all possible paths, as illustrated in Figure 3.12. This ratio expresses the transition probability from state u in $d-1$ to state v in d respectively, conditioned to the observation sequence $\mathbf{x}$.

Based on the expression in Equation 3.2.7, the algorithm will calculate the gradient for every edge of the trellis. As probability values are between 0 and 1, the gradient tends to be positive in golden edges, where function I always returns 1, and negative in ordinary edges, where function I always returns 0.

[2]It is according to the probability distribution at the current stage of the model.



(a) $\alpha(u, d-1) \times \exp F^v(u, w_d(\mathbf{x})) \times \beta(v, d)$ (b) $Z(\mathbf{x})$

Figure 3.12: Figure 3.12a represents the sum of the scores of all paths that go through state M_2 in column $t = 4$ and state M_3 in column $t = 6$ (this transition skips column $t = 5$ because there is no intermediate match states in this merged trellis). Figure 3.12b represents the the sum of all path scores.

The algorithm we run for gradient tree boosting is almost identical to the one we mentioned in Algorithm 4 in the end of Section 2.6.2. The only difference is that the loops for each class (which also means label or state) k and k' will now consider the constraints of the trellis in profile TildeCRF. Another point worth mentioning is the way the logical examples are generated in Algorithm 5 also in Section 2.6.2. For each edge, a new logical example is generated and inserted into a specific database according to its current state. The potential function F^k whose current state is k has its trees grown based on the database $Set(k)$. As the current state for the examples in $Set(k)$ is already known (the current state of an example in $Set(k)$ is k), each example must also contain its previous state, the observation sequence $\mathbf{x}$ (or the window $w_t(\mathbf{x})$) and its gradient value. In the case of profile TildeCRF, the example information is expressed in first-order logic and it consists of a set of logical atoms. The regression tree induction will be based on the gradient values associated to the training examples.

Building logical examples

Incorporating Tilde into the algorithm demands converting sequence x into a first-order logic example. To do this, some logical predicates are previously established before the training procedure. The predicates we used and their descriptions are included in Table 3.2.

Let us say that we wish to calculate the function $F^{M_3}(M_2, \mathbf{x})$ for sequence ACDE, where amino acid D is emitted by state M_3. Firstly, there shall be an atom indicating

Predicate	Description
`delta_value(RealNumber)`	Defines the gradient delta value
`previous_state(State)`	Defines the previous state of the example
`aa(Pos, Amino)`	Defines which amino acid is located in position `Pos`
`blosum62(Pos, Amino, Distance)`	Defines the BLOSUM62 distance from the amino acid located in position `Pos` to amino acid `Amino`
`type(Pos, Type)`	Defines the type of the amino acid located in position `Pos`$see Section 2.1.3)

Table 3.2: List of predicates and their descriptions

the previous state of the example: `previous_state(m2)`.

To define the location of the elements in x, we use positions that are relative to the residue emitted by the current state. So, if sequence ACDE is expressed in terms of predicates `aa`, we will have: `aa(-2, a)`, `aa(-1, c)`, `aa(0, d)` and `aa(1, e)`.

In terms of predicates `type`, we have the following atoms: `type(-2, hydrophobic)`, `type(-1, neutral)`, `type(0, negatively_charged)` and `type(1, negatively_charged)`.

As to predicates `blosum62`, each atom represents the distance from the amino acid in position i to one of the 20 possible residues. Therefore, there are 20 `blosum62`-type atoms for each position in the sequence. Based on table 2.3, the amino acid D in position 0 would be expressed with the following atoms: `blosum62(0, A, -2)`, `blosum62(0, R, -2)`, ...[3], `blosum62(0, X, -1)`.

Let us also assume that the transition $(y_{d-1} = M_2) \rightarrow (y_d = M_3)$ really exists in the training aligned sequence so that $I(y_d = M_3, y_{d-1} = M_2, \mathbf{x})$ is equal to 1, while we suppose that the probability $P(y_d = M_3, y_{d-1} = M_2, \mathbf{x})$ is equal to 0.5. According to the gradient expression, the delta value should be :

$$I(y_d = M_3, y_{d-1} = M_2, \mathbf{x}) - P(y_d = M_3, y_{d-1} = M_2, \mathbf{x}) = 1 - 0.5 = 0.5$$

We can observe all the example information gathered in Table 3.3.

3.2.8 Growing first-order logical regression trees

When all examples are properly generated and stored in their respective datasets, the algorithm is ready for growing the logical regression trees that will be incorporated

[3] 17 atoms were hidden for better visualization.

```
delta_value(0.5),
previous_state(m2),
aa(-2, a), aa(-1, c), aa(0, d), aa(1, e),
type(-2, hydrophobic), type(-1, neutral),
type(0, negatively_charged), type(1, negatively_charged),
blosum62(-2, a, 4), blosum62(-2, r, -1), ..., blosum62(-2, x, 0),
blosum62(-1, a, 0), blosum62(-1, r, -3), ..., blosum62(-1, x, -2),
blosum62(0, a, -2), blosum62(0, r, -2), ..., blosum62(0, x, -1),
blosum62(1, a, -1), blosum62(1, r, 0), ...., blosum62(1, x, -1)
```

Table 3.3: First-order logic representation of sequence ACDE, where D is emitted by current state M_3 that comes from previous state M_2.

into potential functions for the next iterations. This process is executed accordingly to TILDE's growing-tree procedure that we presented in Algorithm 7 in Section 2.8.2.

It is important to avoid the confusion of the term *node* here. There are nodes in the trellis graph, which make part of the CRF problem structure. There are also nodes in the logical regression trees, which are "inside" the potential functions F^k. Regression tree nodes contain information about the best split, which is essentially the logical conjunction that drives an example inference towards the leaf containing the answer.

The *rmodes* we used (which are the predicates to be considered at every split during tree induction and which are better explained in Section 2.8.2) are listed in Table 3.4.

rmode definition	**Splits according to...**
`rmode(previous_state(+State))`	... the previous state
`rmode(aa(+Pos,+Amino))`	... the amino acid located in position `Pos`
`rmode(blosum62(+Pos,+Amino,+Distance))` with lookahead to `rmode(greater_than(-Distance, 0.0))`	... the non-negativity of BLOSUM62 distance from the amino acid located in position `Pos` to `Amino`
`rmode(type(+Pos,+Type))`	... the type of the amino acid located in position `Pos`

Table 3.4: List of rmodes and their splitting criteria

An example of a tree grown by this induction process can be seen in Figure 2.24 in Section 2.8.2.

The growing tree procedure is done for every potential function F^k in our CRF structure, where k is the current state of the transition the function is calculated upon. So, the number of potential functions is proportional to the number of possible current states and each potential function has a set of regression trees.

For example, suppose we have a size-3 profile with the following states: B, I_0, M_1, I_1, D_1, M_2, I_2, D_2, M_3, I_3, D_3 and E. There is no potential function for the begin state B, because it can never be a current state in a transition. If we execute the steps from Section 3.2.3 to Section 3.2.8 for 10 iterations, we would have 10 different regression trees for each one of 11 potential functions (see Table 3.5) or 110 regression trees.

Current state	**Potential function**	**Decomposition at iteration 10**
I_0	F^{I_0}	$F_0^{I_0} + \Delta_1^{I_0} + \Delta_2^{I_0} + \cdots + \Delta_{10}^{I_0}$
M_1	F^{M_1}	$F_0^{M_1} + \Delta_1^{M_1} + \Delta_2^{M_1} + \cdots + \Delta_{10}^{M_1}$
I_1	F^{I_1}	$F_0^{I_1} + \Delta_1^{I_1} + \Delta_2^{I_1} + \cdots + \Delta_{10}^{I_1}$
D_1	F^{D_1}	$F_0^{D_1} + \Delta_1^{D_1} + \Delta_2^{D_1} + \cdots + \Delta_{10}^{D_1}$
M_2	F^{M_2}	$F_0^{M_2} + \Delta_1^{M_2} + \Delta_2^{M_2} + \cdots + \Delta_{10}^{M_2}$
I_2	F^{I_2}	$F_0^{I_2} + \Delta_1^{I_2} + \Delta_2^{I_2} + \cdots + \Delta_{10}^{I_2}$
D_2	F^{D_2}	$F_0^{D_2} + \Delta_1^{D_2} + \Delta_2^{D_2} + \cdots + \Delta_{10}^{D_2}$
M_3	F^{M_3}	$F_0^{M_3} + \Delta_1^{M_3} + \Delta_2^{M_3} + \cdots + \Delta_{10}^{M_3}$
I_3	F^{I_3}	$F_0^{I_3} + \Delta_1^{I_3} + \Delta_2^{I_3} + \cdots + \Delta_{10}^{I_3}$
D_3	F^{D_3}	$F_0^{D_3} + \Delta_1^{D_3} + \Delta_2^{D_3} + \cdots + \Delta_{10}^{D_3}$
E	F^{E}	$F_0^{E} + \Delta_1^{E} + \Delta_2^{E} + \cdots + \Delta_{10}^{E}$

Table 3.5: List of potential functions and their decomposition

3.3 Test and performance evaluation

We intend to test the performance of profile TildeCRF in the context of remote protein homology detection. This problem involves learning from a set of training sequences and trying to predict if distantly homologous proteins are indeed related to the training examples.

When all iterations are complete, the potential functions shall be used for evaluating the test examples. Then, using the trained model, inference is performed on every test sequence in order to calculate its alignment score. This is done by executing the same procedures presented in Sections 3.2.4 and 3.2.6, which respectively refer to building the trellis diagram and calculating the potential function values for each trellis edge. Because positive test examples belong to the same superfamily as the training examples (see Section 2.1.6), they are supposed to score higher than negative test examples. So, the higher the positive test examples score, the more accurate the model is. However, we could not find a standard method for calculating a test score in discriminative models in such a way that it represents the distance from the target sequence to the consensus sequence of the trained model. In generative models like profile HMM, it is possible to measure this distance by calculating the odds ratio between the probability of the target sequence given the trained model

and the probability of the target sequence given the null hypothesis (Equation 3.10) [EDDY, 2008]. Therefore, we created and tested three different scoring methods in order to evaluate which option better corresponds to the similarity between the target and consensus sequences. The three different options will be presented later in this section.

After inference is performed on all sequences of the test set, we will have all the necessary information to evaluate the overall performance of the model for a target family. This performance is measured by calculating the AUC-ROC value (Section 2.9) based on the test scores.

Average Viterbi conditional log probability

We present now the first of the three options we created to calculate the inference score. The basic way of measuring an inference score is to use the Viterbi algorithm and to assume that the best path log conditional probabilty represents the test score for that sequence.

$$S_0 = \log P(\mathbf{y}^* | \mathbf{x}_{target}, H) \tag{3.7}$$

where S_a is the test score suggested above, $\mathbf{y}^*$ is the best state sequence for target sequence x_{target} and H is trained model.

This method, however, presents a flaw: longer sequences always tend to have lower probabilities than short ones. In order to have a fairer scoring method regardless the sequence length, the average Viterbi conditional log probability is used instead. The average is obtained by dividing the log probability by the number of columns in the trellis.

$$S_1 = \frac{\log P(\mathbf{y}^* | \mathbf{x}_{target}, H)}{N_{columns}} \tag{3.8}$$

where S_b is the test score suggested above, $\mathbf{y}^*$ is the best state sequence for target sequence x_{target}, H is trained model and $N_{columns}$ is the number of columns in the trellis.

Average unnormalized log probability

Another score worth considering is the unnormalized log probability. Let us recall Equation 2.24, which gives the CRF conditional probability as a function of input sequence $\mathbf{x}$. The numerator may be viewed as the unnormalized joint log probability of sequences, since it must be divided by the $Z(\mathbf{x})$, which is the sum of the unnormalized probabilities of all possible alignments of $\mathbf{x}$), in order to have the conditional normalized probability.

$$P(\mathbf{y}^*|\mathbf{x}_{target}, H) = \frac{\exp \sum_t F^{y_t^*}(y_{t-1}^*, \mathbf{x})}{Z(\mathbf{x})} = \frac{\tilde{P}(\mathbf{x}, \mathbf{y}^*|H)}{\tilde{P}(\mathbf{x}|H)}$$

where $\mathbf{y}^*$ is the best state sequence for target sequence x_{target}, y_t^* is the t-th element of $\mathbf{y}^*$, $\tilde{P}(\mathbf{x}, \mathbf{y}^*|H)$ is the unnormalized joint log probability of $\mathbf{x}$ and $\mathbf{y}$ given the trained model H and $\tilde{P}(\mathbf{x}|H)$ is the unnormalized log probability of $\mathbf{x}$.

So, the second score option S_2 is the unnormalized log probability averaged by the number of columns in the trellis, in order to give a fair score regardless the target sequence length.

$$S_2 = \frac{\sum_t F^{y_t^*}(y_{t-1}^*, \mathbf{x})}{N_{columns}} = \frac{\log \tilde{P}(\mathbf{x}, \mathbf{y}^*|H)}{N_{columns}} \tag{3.9}$$

Unnormalized Log-Odds Ratio

In addition, one other option is to use a measure inspired by the Viterbi score used in profile HMM [EDDY, 2008] (Equation 3.10). This score corresponds to the log-odds ratio between the value returned by the trained model and the value returned by the random model, which is a model where alignments occur at random. Alternatively, this log-odds ratio can be viewed as a hypothesis test where the hypothesis H is compared to the null hypothesis R.

$$S_{\text{log-odds ratio}} = \underbrace{\log \frac{P(\mathbf{x}_{target}|H)}{P(\mathbf{x}_{target}|R)}}_{\text{Forward score}} \approx \underbrace{\log \frac{P(\mathbf{x}_{target}, \pi^*|H)}{P(\mathbf{x}_{target}|R)}}_{\text{Viterbi score}} \tag{3.10}$$

where π^* represents the best alignment in profile HMM. The Viterbi approximation to $P(x|H)$ is a way of assuming that all the probability mass is concentrated in the best state sequence π^* so that $P(x, \pi^*|H) \approx P(x|H)$. To calculate the random model probability, it suffices to multiply the background probability[4] q_{x_i} of each residue $x_i \in \mathbf{x}$ by each other:

$$P(\mathbf{x}|R) = \prod_{x_i \in \mathbf{x}} q_{x_i} \tag{3.11}$$

Even so, we cannot use $S_{\text{log-odds ratio}}$ in profile TildeCRF as it is, because in discriminative models like CRF the Viterbi algorithm return the conditional probability $P(\mathbf{y}^*|\mathbf{x}_{target}, H)$, but not the joint probability $P(\mathbf{x}_{target}, \mathbf{y}^*|H)$. In truth, the closest we can get to a joint probability in CRF is the unnormalized probability $\tilde{P}(\mathbf{x}_{target}, \mathbf{y}^*|H)$. To obtain the normalized joint probability, it would be necessary to calculate a universal normalization factor Z, which consists in the sum of the un-

[4]The background probability of a residue is the probability that it occurs in nature randomly.

normalized probabilities of all possible state sequences $\mathbf{y}$ of all possible observation sequences $\mathbf{x}$.

$$\begin{aligned} P(\mathbf{x}, \mathbf{y}^*|H) &= \frac{\tilde{P}(\mathbf{x}, \mathbf{y}^*|H)}{Z} \\ Z &= \sum_{\mathbf{x}'} Z(\mathbf{x}') = \sum_{\mathbf{x}'} \sum_{\mathbf{y}'} \tilde{P}(\mathbf{x}', \mathbf{y}'|H) \end{aligned} \tag{3.12}$$

Calculating the universal normalization factor Z is generally intractable [MCCALLUM, 2003], but Z is constant for all inference scores given a trained model. Therefore, it is reasonable to neglect the influence of Z, if our main aim is to compare the scores between each other.

$$\log \frac{P(\mathbf{x}_{target}, \mathbf{y}^*|H)}{P(\mathbf{x}_{target}|R)} = \log \frac{\tilde{P}(\mathbf{x}_{target}, \mathbf{y}^*|H)}{P(\mathbf{x}_{target}|R)} - \underbrace{\log Z}_{\text{constant}} \tag{3.13}$$

So, the third option S_3 of measuring the inference score is a version of the log-odds ratio, where Z is discarded and the unnormalized log probability is used instead of the normalized one.

$$S_3 = \log \frac{\tilde{P}(\mathbf{x}_{target}, \mathbf{y}^*|H)}{P(\mathbf{x}_{target}|R)} \tag{3.14}$$

Chapter 4

Experimental Results

In this chapter we describe the results obtained by profile TildeCRF. First, Section 4.1 presents the dataset we used for our experiments. Then, in Section 4.2 we present the results obtained by our method and we compare them to the results that HMMER returned.

4.1 Dataset description

In our experiments we used the dataset provided by LIAO and NOBLE [2002] that contains 4352 distinct sequences grouped into families and superfamilies[1]. These sequences were selected from the Astral database [CHANDONIA *et al.*] by removing similar sequences with an E-value[2] threshold of 10^{-25}. Although the data set suggested by LIAO and NOBLE [2002] has positive and negative examples for both training and tests, no negative training examples are used, because profile *TildeCRF* is not compatible with entirely negative sequences.

For remote protein homology detection, our experiments followed the *leave-one-family-out* strategy, in which the training set consists of a superfamily's sequences, excluding those belonging to the target family. The proteins of the target family are later used as positive test examples in order to maintain remote similarities between the training set and the test set. The dataset proposed by LIAO and NOBLE [2002] has 54 target families altogether. For each target family the accuracy was measured by calculating the AUC-ROC (see Section 2.9) with the inference scores of both positive and negative sequences from the test set.

[1] For hierarchical classification description, see Section 2.1.6.

[2] *E-value* stands for *Expectation value* or *Expect value* and it represents the number of different alignments that are expected to occur in a database search at random with scores greater than or equal to the current alignment score. The lower the E-value is, the less probable it is for random alignments to have a bigger score and the more significant the current alignment become [FASSLER and COOPER, 2011].

4.2 Experimental comparison between HMMER and profile TildeCRF

The target families of the dataset were used as the test sets of our experiments. For each target family, we used the algorithm Clustal Omega [SIEVERS *et al.*, 2011] with the standard parameters so as to obtain the multiple alignment of its respective positive training sequences. These are sequences that belong to the same *superfamily* as the target family, but not to the same *family*. Once the positive training sequences are aligned to each other, we executed the training procedure on them, using zero as the initial value of the gradient tree boosting algorithm (for more details on initial values, see Section 3.2.6). The output of the training procedure is the trained CRF model with its 'forest' of regression trees. Then, based on the trained model, we performed inference on the target sequences using three different ways of calculating the inference scores (see Table 4.1). Lastly, we used versions 2 and 3 of the algorithm HMMER for the same target families in order to compare its AUC ROC values to the ones obtained by profile TildeCRF. Two different versions of HMMER were used, because HMMER 2 has *global* alignment mode enabled and, although HMMER 3 is the latest stable version, it only performs *local* alignment (for a description of the alignment modes, see Section 2.4.4). In both HMMER versions we turned off the prior information parameter so that a fair comparison could be made between HMMER and profile TildeCRF. Apart from the 'priors off' option, the other HMMER parameters were kept at default.

Score	**Description**	**Expression**
S_1	Average Viterbi conditional log probability	$\frac{\log P(\mathbf{y}^*\|\mathbf{x}_{target}, H)}{N_{columns}}$ (Eq. 3.8)
S_2	Average unnormalized log probability	$\frac{\log \tilde{P}(\mathbf{x}, \mathbf{y}^*\|H)}{N_{columns}}$ (Eq. 3.9)
S_3	Unnormalized Log-Odds Ratio	$\log \frac{\tilde{P}(\mathbf{x}_{target}, \mathbf{y}^*\|H)}{P(\mathbf{x}_{target}\|R)}$ (Eq. 3.14)

Table 4.1: Score descriptions and expressions (see details in Section 3.3)

In Table 4.2 we show the AUC ROC values obtained with HMMER versions 2 and 3. In addition, we also show the AUC ROC values that were returned by our proposed method, using the scoring formulas S_1, S_2 and S_3 presented in Equations 3.8, 3.9, 3.14 in Section 3.3. Due to limitations related to our method's execution time, only 37 of the 54 original target families were tested successfully. The other 17 target families could not be fully examined due to stack overflow or timeout

exceptions. In addition, Figure 4.1 sums up the number of cases in which the scores S_1, S_2 and S_3 are greater than the values resulted by HMMER.

Target family	HMMER2	HMMER3	S_1	S_2	S_3
1.27.1.2	**0.820**	0.711	0.531	0.663	0.649
1.36.1.2	**0.478**	0.366	0.178	0.443	0.361
1.36.1.5	0.300	**0.632**	0.393	0.336	0.016
1.4.1.1	**0.698**	0.621	0.279	0.287	0.323
1.4.1.2	0.518	**0.908**	0.395	0.277	0.268
1.4.1.3	**0.876**	0.852	0.427	0.301	0.304
1.41.1.2	0.794	**0.896**	0.256	0.499	0.477
1.41.1.5	0.825	**0.968**	0.452	0.525	0.516
1.45.1.2	0.295	0.242	0.518	**0.891**	0.267
2.1.1.1	0.911	**0.946**	0.446	0.612	0.580
2.1.1.2	0.895	**0.949**	0.371	0.527	0.524
2.1.1.3	0.981	**0.989**	0.323	0.450	0.437
2.1.1.4	**0.981**	0.971	0.546	0.517	0.512
2.1.1.5	**0.862**	0.798	0.544	0.618	0.610
2.28.1.1	0.285	**0.804**	0.543	0.358	0.339
2.28.1.3	0.611	0.435	0.474	0.612	**0.632**
2.38.4.1	0.384	**0.708**	0.408	0.555	0.585
2.38.4.3	0.581	0.580	0.581	**0.665**	0.664
2.38.4.5	0.501	**0.560**	0.338	0.480	0.431
2.5.1.1	**0.854**	0.745	0.352	0.604	0.559
2.5.1.3	**0.948**	0.646	0.433	0.808	0.759
2.52.1.2	**0.857**	0.580	0.436	0.674	0.671
2.56.1.2	**0.719**	0.376	0.452	0.718	0.675
2.9.1.4	**0.948**	0.764	0.609	0.412	0.778
3.1.8.1	**0.998**	0.558	0.800	0.936	0.967
3.2.1.2	**0.933**	0.795	0.821	0.862	0.911
3.2.1.3	**0.896**	0.688	0.599	0.783	0.766
3.2.1.5	0.916	**0.924**	0.647	0.758	0.705
3.2.1.6	0.846	**0.860**	0.515	0.807	0.808
3.2.1.7	**0.977**	0.914	0.815	0.846	0.848
3.3.1.2	0.898	**0.964**	0.796	0.894	0.884
3.32.1.11	**0.992**	0.813	0.679	0.798	0.863
3.32.1.13	0.914	**0.965**	0.761	0.862	0.863

3.32.1.8	**0.971**	0.966	0.478	0.782	0.813
7.39.1.3	0.539	**0.674**	0.491	0.187	0.176
7.41.5.1	**0.714**	0.651	0.256	0.281	0.296
7.41.5.2	**0.571**	0.279	0.259	0.466	0.147

Table 4.2: AUC ROC values according to the target family and the tested methods

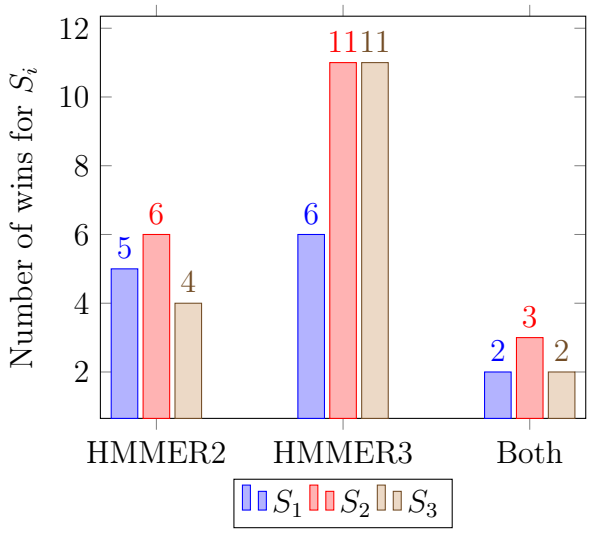


Figure 4.1: Number of cases in which scores S_1, S_2 and S_3 are greater than the scores returned by HMMER2, HMMER3 and both

In Figure 4.2 we show the ROC curve of the experiment scenario involving target family 1.45.1.2, whose description is *Arc1p N-terminal domain-like*. This is one of the 3 cases where profile TildeCRF scored higher than both versions of HMMER. The reasons why this scenario was considerably better than the others are not clear. One intuition is that the probability for the null hypothesis (random model) played a major role on this specific case. While the score S_2 did not consider the influence of $P(\mathbf{x}/R)$, the score S_3 did and it happened to be as low as the scores obtained with HMMER. Another possible reason is that averaging out the unnormalized probabilities in S_2 by the sequence length somehow favored smaller sequences in this scenario, where the average length of positive target sequences are 37.3% of the average length of negative target sequences.

In Figure 4.3 we observe the ROC curve for target family 3.1.8.1, whose description is *Amylase, catalytic domain*, according to the SCOP database. In this case, however, HMMER2 presented higher scores than HMMER3 and profile TildeCRF.

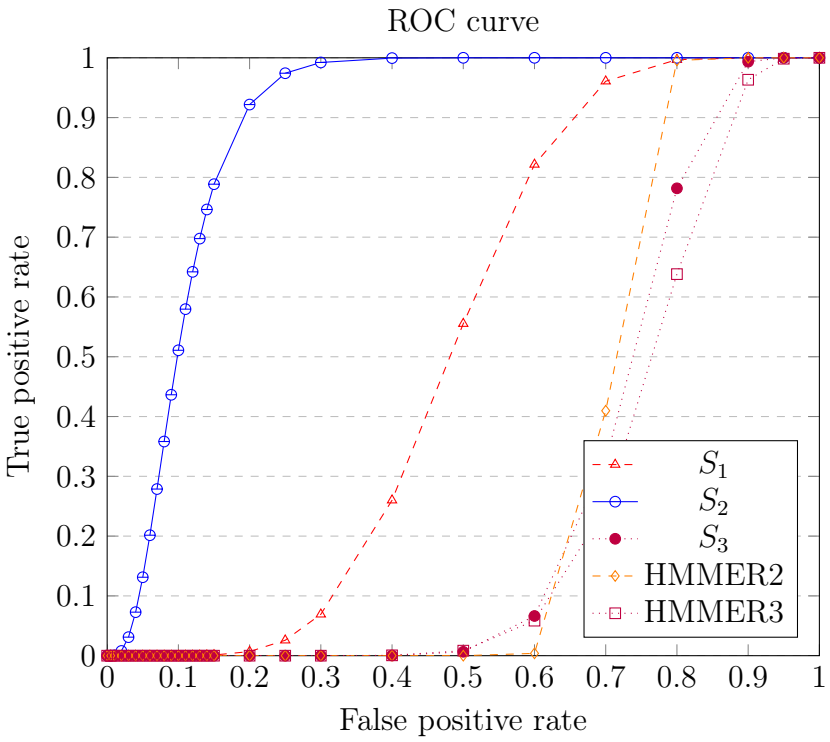


Figure 4.2: ROC curve for target family 1.45.1.2

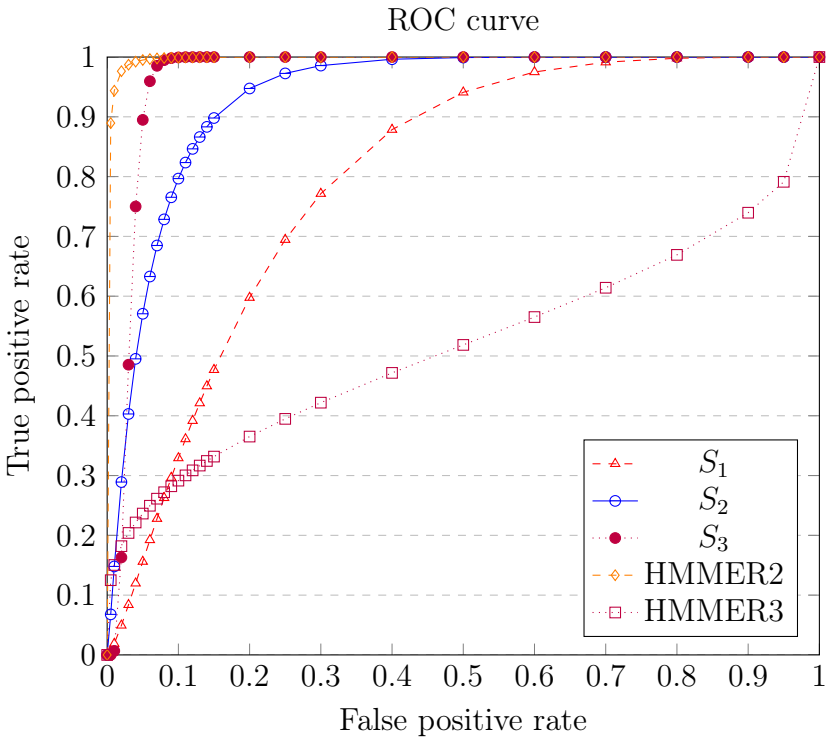


Figure 4.3: ROC curve for target family 3.1.8.1

Chapter 5

Discussion and Conclusions

In this chapter we present our discussion and conclusions about this study. Section 5.1 summarizes the findings emerged from this work. Then, Section 5.2 presents the conclusions we draw from the results. Lastly, Section 5.3 shows the recommendations we propose for future research.

5.1 Summary of Findings

This study aimed to investigate the performance of profile TildeCRF in addressing the problem of protein distant homology detection. Profile TildeCRF is an algorithm that merges three characteristics from three different sources: the graph structure from profile HMM, the discriminative model from CRF and the logic representation from Tilde. Our initial hypothesis was that profile TildeCRF could outperform profile HMM, since discriminative models do not need the same strong assumptions as generative models, and since first-order logic offers a greater expressive power to the observation input. In order to prove our point, we developed our proposed algorithm in Python and we compared its results with the ones returned by HMMER, which is the most well-known implementation of profile HMM. The algorithm execution showed itself to be very time and resource consuming. Despite being apparently simpler, HMMER outperformed profile TildeCRF in the vast majority of experimental scenarios. Contrary to the widely held belief, generative models seems to be more efficient than discriminative models, at least as far as it concerns protein homology detection.

5.2 Conclusions Drawn by Results

This study showed that the higher complexity of the algorithm profile TildeCRF did not correspond to a higher accuracy of the model in comparison to simpler

generative models. In this section we intend to explain the adequacy of generative models in protein homology detection over discriminative models. Also, we intend to explain the complexity of profile TildeCRF, as it turned out to be time and resource consuming.

5.2.1 Higher effectiveness of generative models in this study

As we have seen in Section 2.5.1, generative and discriminative models are both generally used in classification problems. Comparisons between these two types of classifiers part from the premise that the main goal is to optimize the posterior probability $P(\mathbf{y}|\mathbf{x})$. However, the objective is not the same when it comes to protein homology detection. In this kind of problem, the main aim is to build a hypothetical model H that correctly evaluates how similar a target sequence $\mathbf{x}$ is to the consensus sequence, which is the multiple alignment that originated the model H. This model is built by optimizing the likelihood $P(\mathbf{x}|H)$ of a positive target $\mathbf{x}$ given the hypothetical model H (or the likelihood $P(\mathbf{x}, \mathbf{y}^*|H)$ if we consider that the probability mass is concentrated in the one case where $\mathbf{x}$ is aligned with the optimal state sequence $\mathbf{y}^*$). The log-odds ratio between this likelihood value and the likelihood $P(\mathbf{x}|R)$ of $\mathbf{x}$ under the null hypothesis gives an accurate measure of how likely it is for $\mathbf{x}$ to be generated from model H in comparison to being generated at random.

Therefore, in the case of protein homology detection, the optimization of the joint probability $P(\mathbf{x}, \mathbf{y}^*|H)$ is no longer an intermediate step to reach the final result, but it becomes the ultimate goal itself. On the other hand, $P(\mathbf{y}^*|\mathbf{x}, H)$ loses its significance, because it just measures the adequacy of a state sequence $\mathbf{y}^*$ given an observation sequence $\mathbf{x}$ according to the model H, but it does not represent the distance between $\mathbf{x}$ and H. — Imagine, for instance, that $\mathbf{x}$ is a one-element target sequence. The probability of the optimal state sequence $\mathbf{y}^*$ for $\mathbf{x}$ may be relatively high, as the range of possible state sequences is very short for a one-element target sequence. However, a high conditional probability does not necessarily mean that this target is close to the consensus sequence. In this example, it can be quite the opposite. — That is probably the reason why HMMER does not use $P(\mathbf{y}^*|\mathbf{x}, H)$ as a similarity measure, despite having itself all the information available to do such a calculation. Remember:

$$P(\mathbf{y}^*|\mathbf{x}, H) = \frac{P(\mathbf{x}, \mathbf{y}^*|H)}{\sum_{\mathbf{y}} P(\mathbf{x}, \mathbf{y}|H)} \tag{5.1}$$

In fact, calculating the probability of a label sequence $\mathbf{y}^*$ given an input sequence $\mathbf{x}$ seems to be more suitable for sequence labeling tasks such as named-entity recognition and part-of-speech tagging. In subsection 3.3, we used the concepts of log-odds

ratio from HMMER [EDDY, 2008] in order to formulate the scoring method S_3. As there is no joint probability in CRF, we used the ration between the 'unnormalized' probability and the joint probability under the null hypothesis. Nevertheless, although we tried to measure the inference with CRF in a generative way, the training procedure was based on the gradient ascent that had been designed to optimize $\log P(\mathbf{y}|\mathbf{x}, H)$. Thus, the learning process of profile TildeCRF was not consistent with the desired goal of maximizing the joint likelihood $\log P(\mathbf{x}, \mathbf{y}|H)$ of the training sequences.

5.2.2 Tilde implementation

In this study we chose a new implementation of Tilde algorithm in Python/Cython over the original implementation in Prolog or Java for the reasons expressed in Section 3.2.1. One might question if this choice could have compromised the quality of the results. Moreover, our implementation used a simplified version of logic predicate unification, where only the substitution of variables for constants is contemplated, leaving out the case where there are functions inside the logical atoms. Notwithstanding, this particular limitation did not affect our problem handling, because the logical atoms that were used to represent the observations or the decision tree queries contained only predicates with constants and/or variables. No functions were used inside the logical atoms. In very trivial cases, which we simulated for test purposes only, the model presented a good accuracy. The issues mentioned in last section were far more determinant in the low effectiveness of our method regarding distant homology detection.

5.2.3 High cost of profile TildeCRF

Another drawback of profile TildeCRF is its high complexity in terms of algorithm. The execution time for the experiments shown in Chapter 4 was about 72 hours in a cluster of 12 computers, which shows that profile TildeCRF is very time-consuming and resource-intensive, compared to the 2-minute execution performed by both versions of HMMER.

One reason for this high cost is the fact that the algorithm's learning procedure requires performing inferences on training sequences. In order to calculate the gradient ascent, the algorithm uses the inference results to approach to the desired value iteratively.

Another reason for the expensiveness of our proposed method is due to the behaviour of the top-down induction of regression trees. While HMMER simply calculates statistical counts based on the input multiple alignment, which are computationally cheap arithmetic operations, profile TildeCRF has to generate a log-

ical dataset for each possible state in the trellis and build a tree from it at every iteration. If the model presents K different states and the learning procedure is set for N iterations, NK different regression trees have to be built in the end. Given the great length of certain protein sequences and the high number of possible states, the process of generating examples and inducing trees repeatedly turned out to be very costly, depite our efforts to keep all data inside the RAM memory and to use as much parallelism as possible.

5.3 Recommendations for Further Research

First of all, a recommendation for further research would be a deeper analysis of why some specific experimental scenarios were successful in comparison with others. It is important to identify in which circumstances and for which types of family our method performed better. For example, we may deduce that the more accurate the model is, the higher the number of training examples is or the shorter the training sequences are. Perhaps, if we understand the reason for those specific sucesses, we could apply it to the other cases and optimize our method globally. In fact, statistical comparison and inference between the model results and protein family features would be much appreciated, but, unfortunately, this could not be done within the time limits set for this study.

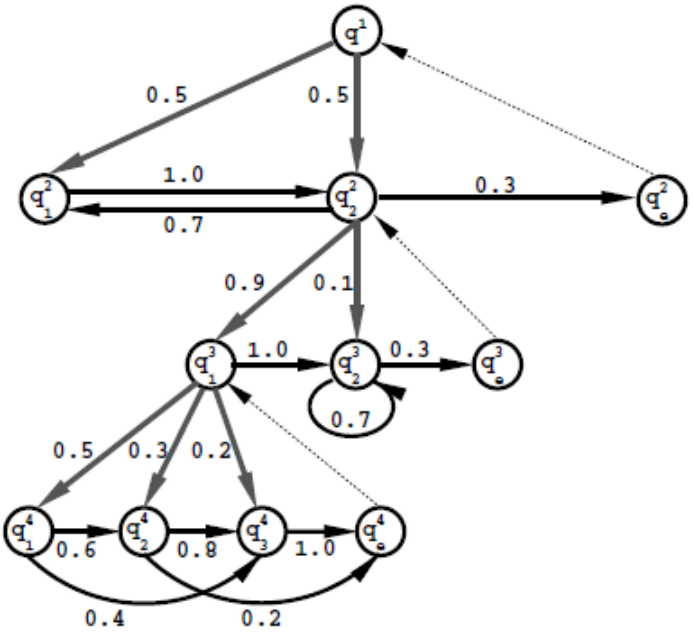


Figure 5.1: An illustration of an HHMM of four levels. Gray and black edges respectively denote vertical and horizontal transitions. Dashed thin edges denote (forced) returns from the end state of each level to the level's parent state. [FINE *et al.*, 1998]

A suggestion for correctly dealing with the distant homology detection problem is to consider generative alternatives to CRF. Generative models are capable of measuring the distance between the target sequence and the consensus sequence. Profile HMM remains the basic solution and any attempt to insert first-order logic into the scope should take the generative property into consideration. We could combine decision-trees with HMMs using *Hierarchical hidden Markov models* [FINE *et al.*, 1998] (Figure 5.1), which are structured multi-level stochastic processes that generalize the standard HMMs by making each hidden state an "autonomous" probabilistic model on its own. Therefore, we could initially use propositional logic in order to build decision trees that will act like HHMMs. Afterwards, an upgrade to first-order logic might be contemplated.

The problem of protein homology detection has been very much studied and explored throughout the literature of bioinformatics, so any attempt to make progress in this field should be carefully analyzed in terms of time investment and gains. However, there are still some gaps to be filled in, such as homology detection using protein secondary and tertiary structures. We believe that the use of relational logic can be of great help in this direction.

Bibliography

ALTSCHUL, S. F., GISH, W., MILLER, W., et al., 1990, "Basic local alignment search tool", *Journal of Molecular Biology*, v. 215, n. 3, pp. 403–410. ISSN: 0022-2836.

BASHFORD, D., CHOTHIA, C., LESK, A. M., 1987, "Determinants of a protein fold. Unique features of the globin amino acid sequences." *Journal of Molecular Biology*, v. 196, n. 1 (July), pp. 199–216. ISSN: 0022-2836.

BAUM, L. E., 1972, "An inequality and associated maximization technique in statistical estimation for probabilistic functions of Markov processes", *Inequalities*, v. 3, pp. 1–8.

BERMAN, H. M., WESTBROOK, J., FENG, Z., et al., 2000, "The Protein Data Bank", *Nucleic Acids Research*, v. 28, n. 1, pp. 235–242.

BININDA-EMONDS, O., 2005, "transAlign: using amino acids to facilitate the multiple alignment of protein-coding DNA sequences", *BMC Bioinformatics*, v. 6, n. 1, pp. 156. ISSN: 1471-2105.

BISHOP, C. M., LASSERRE, J., 2007, "Generative or Discrimative? Getting the Best of Both Worlds". In: Bernardo, J. M., Bayarri, M. J., Berger, J. O., et al. (Eds.), *Bayesian Statistics 8*, pp. 3–24. International Society for Bayesian Analysis, Oxford University Pres.

BLOCKEEL, H., 1998, *Top-down induction of first order logical decision trees*. Phd, Department of Computer Science, K.U.Leuven, Leuven, Belgium, December. 202+xv pagesURL: `http://www.cs.kuleuven.ac.be/cgi-bin-dtai/publ_info.pl?id=20977`.

BLOCKEEL, H., RAEDT, L. D., 1998, "Top-down induction of first-order logical decision trees", *Artificial Intelligence*, v. 101, n. 1–2, pp. 285–297. ISSN: 0004-3702.

BREIMAN, L., FRIEDMAN, J. H., OLSHEN, R. A., et al., 1984, *Classification and Regression Trees*. Belmont, CA, Wadsworth International Group.

CANDOTTI, F. "National Human Genome Research Insitutue". http://www.genome.gov/.

CHANDONIA, J.-M., ZHI, D., HON, G., et al. "The ASTRAL Compendium for Sequence and Structure Analysis". Available at: http://astral.berkeley.edu/.

COJOCARI, D. "pKa Data: CRC Handbook of Chemistry and Physics v.2010". Available at: http://commons.wikimedia.org/wiki/File:Amino_Acids.svg under Wikimedia Commons License.

CONTE, L. L., AILEY, B., HUBBARD, T. J. P., et al., 2000, "SCOP: a Structural Classification of Proteins database", *Nucleic Acids Research*, v. 28, n. 1, pp. 257–259.

DARWIN, C., 1859, *On the Origin of Species by Means of Natural Selection*. John Murray. or the Preservation of Favored Races in the Struggle for Life.

DARWIN, C., 1868, *The Variation of Animals and Plants Under Domestication*. John Murray.

DASGUPTA, S., PAPADIMITRIOU, C. H., VAZIRANI, U. V., 2006, *Algorithms*. McGraw-Hill.

DAVIS, J., GOADRICH, M., 2006, "The Relationship Between Precision-Recall and ROC Curves". In: *Proceedings of the 23rd International Conference on Machine Learning*, ICML '06, pp. 233–240, New York, NY, USA. ACM. ISBN: 1-59593-383-2.

DE RAEDT, L., 1996, "Induction in logic". In: Michalski, R., Wnek, J. (Eds.), *Proceedings of the 3rd International Workshop on Multistrategy Learning,*, pp. 29–38. AAAI Press.

DIETTERICH, T. G., ASHENFELTER, A., BULATOV, Y., 2004, "Training Conditional Random Fields via Gradient Tree Boosting". In: *Proceedings of the Twenty-first International Conference on Machine Learning*, ICML '04, pp. 217–224. ACM.

DURBIN, R., EDDY, S. R., KROGH, A., et al., 1998, *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press.

EDDY, S. R., 1998, "Profile hidden Markov models", *Bioinformatics*, v. 14, n. 9, pp. 755–763.

EDDY, S. R., 2008, "A Probabilistic Model of Local Sequence Alignment That Simplifies Statistical Significance Estimation", *PLoS Computational Biology*, v. 4, n. 5 (May), pp. e1000069.

FASSLER, J., COOPER, P., 2011. "BLAST Glossary". Available at: `http://www.ncbi.nlm.nih.gov/books/NBK62051/`, July.

FAWCETT, T., 2006, "An Introduction to ROC Analysis", *Pattern Recogn. Lett.*, v. 27, n. 8 (June), pp. 861–874. ISSN: 0167-8655.

FINE, S., SINGER, Y., TISHBY, N., 1998, "The Hierarchical Hidden Markov Model: Analysis and Applications", *Machine Learning*, v. 32, n. 1, pp. 41–62. ISSN: 0885-6125.

FOX, N. K., BRENNER, S. E., CHANDONIA, J.-M., 2013, "SCOPe: Structural Classification of ProteinsâĂŤextended, integrating SCOP and ASTRAL data and classification of new structures", *Nucleic Acids Research*.

GIRDEA, M., NOE, L., KUCHEROV, G., 2010, "Back-translation for discovering distant protein homologies in the presence of frameshift mutations", *Algorithms for Molecular Biology*, v. 5, n. 1, pp. 6. ISSN: 1748-7188.

GRIBSKOV, M., MCLACHLAN, A. D., EISENBERG, D., 1987, "Profile analysis: detection of distantly related proteins", *Proceedings of the National Academy of Sciences*, v. 84, n. 13 (July), pp. 4355–4358. ISSN: 1091-6490.

GRONT, D., BLASZCZYK, M., WOJCIECHOWSKI, P., et al., 2012, "BioShell Threader: protein homology detection based on sequence profiles and secondary structure profiles." *Nucleic Acids Research*, v. 40, n. Web-Server-Issue, pp. 257–262.

GUTMANN, B., KERSTING, K., 2006, "TildeCRF: Conditional random fields for logical sequences". In: *In Proceedings of the 15th European Conference on Machine Learning (ECML-06)*, pp. 174–185. Springer.

HAMMERSLEY, J. M., CLIFFORD, P. E., 1971, "Markov fields on finite graphs and lattices", Unpublished manuscript.

HENIKOFF, S., HENIKOFF, J. G., 1992, "Amino acid substitution matrices from protein blocks." *Proceedings of the National Academy of Sciences of the United States of America*, v. 89, n. 22 (November), pp. 10915–10919.

HUGHEY, R., KROGH, A., 1996, "Hidden Markov models for sequence analysis: extension and analysis of the basic method", *Computer applications in the biosciences : CABIOS*, v. 12, n. 2, pp. 95–107.

KIMBALL, J. W. "Kimball's Biology Pages". http://biology-pages.info/.

KINJO, A. R., 2009, "Profile conditional random fields for modeling protein families with structural information", *BIOPHYSICS*, v. 5, pp. 37–44.

KRAMER, S., 1996, "Structural Regression Trees". In: *Proceedings of the Thirteenth National Conference on Artificial Intelligence and Eighth Innovative Applications of Artificial Intelligence Conference, AAAI 96, IAAI 96, Portland, Oregon, August 4-8, 1996, Volume 1.*, pp. 812–819.

KROGH, A., BROWN, M., MIAN, I. S., et al., 1994, "Hidden Markov models in computational biology: applications to protein modeling", *Journal of Molecular Biology*, v. 235, pp. 1501–1531.

LAFFERTY, J. D., MCCALLUM, A., PEREIRA, F. C. N., 2001, "Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data". In: *Proceedings of the Eighteenth International Conference on Machine Learning*, ICML '01, pp. 282–289, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc. ISBN: 1-55860-778-1.

LEJA, D., 2010. "National Human Genome Research Insitutue". http://www.genome.gov/.

LIAO, L., NOBLE, W. S., 2002, "Combining pairwise sequence similarity and support vector machines for remote protein homology detection". In: *Proceedings of the Sixth Annual Conference on Research in Computational Molecular Biology*, pp. 225–232.

MAGRANE, M., CONSORTIUM, U., 2011, "UniProt Knowledgebase: a hub of integrated protein data", *Database*, v. 2011.

MARGELEVICIUS, M., VENCLOVAS, C., 2010, "Detection of distant evolutionary relationships between protein families using theory of sequence profile-profile comparison", *BMC Bioinformatics*, v. 11, n. 1, pp. 89.

MCCALLUM, A., 2003, "Efficiently Inducing Features of Conditional Random Fields". In: *Proceedings of the Nineteenth Conference on Uncertainty in Artificial Intelligence*, UAI'03, pp. 403–410, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

MENDEL, G., 1866, "Versuche über Pflanzen-Hybriden [Experiments in Plant Hybridisation]", *Verhandlungen des naturforschenden Vereines in Brünn*, v. 42, pp. 3–47.

NEEDLEMAN, S. B., WUNSCH, C. D., 1970, "A general method applicable to the search for similarities in the amino acid sequence of two proteins", *Journal of Molecular Biology*, v. 48, n. 3, pp. 443–453. ISSN: 0022-2836.

NG, A. Y., JORDAN, M. I., 2001, "On Discriminative vs. Generative Classifiers: A comparison of logistic regression and naive Bayes". In: Dietterich, T. G., Becker, S., Ghahramani, Z. (Eds.), *Advances in Neural Information Processing Systems 14 (NIPS 2001)*, pp. 841–848. MIT Press.

NHGRI, a. "National Human Genome Research Institute. Talking Glossary of Genetic Terms." `http://www.genome.gov/glossary/`, a.

NHGRI, b. "National Human Genome Research Institute. An Overview of the Human Genome Project". `http://www.genome.gov/12011238`, visited on September 13th, 2014, b.

NOTREDAME, C., HIGGINS, D. G., HERINGA, J., 2000, "T-Coffee: A novel method for fast and accurate multiple sequence alignment." *Journal of Molecular Biology*, v. 302, n. 1, pp. 205–17.

OCONE, D., 2009, *Discrete and Probabilistic Models in Biology*. Rutgers, The State University of New Jersey.

PEARSON, W. R., LIPMAN, D. J., 1988, "Improved tools for biological sequence comparison", *Proceedings of the National Academy of Sciences*, v. 85, n. 8, pp. 2444–2448.

QUINLAN, J. R., 1993, *C4.5: Programs for Machine Learning*. San Francisco, CA, USA, Morgan Kaufmann Publishers Inc. ISBN: 1-55860-238-0.

RABINER, L. R., 1989, "A tutorial on hidden Markov models and selected applications in speech recognition", *Proceedings of the IEEE*, v. 77, n. 2 (February), pp. 257–286. ISSN: 0018-9219.

RAHIMI, A., RABINER, L. R. "An Erratum for "A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition"". Available at: `http://alumni.media.mit.edu/~rahimi/rabiner/rabiner-errata/rabiner-errata.html`.

SIEVERS, F., WILM, A., DINEEN, D., et al., 2011, "Fast, scalable generation of high-quality protein multiple sequence alignments using Clustal Omega", *Molecular Systems Biology*, v. 7, n. 1. ISSN: 1744-4292.

SJÖLANDER, K., KARPLUS, K., BROWN, M., et al., 1996, "Dirichlet mixtures: a method for improved detection of weak but significant protein sequence homology", *Computer applications in the biosciences : CABIOS*, v. 12, n. 4, pp. 327–345.

SMITH, T., WATERMAN, M., 1981, "Identification of common molecular subsequences", *Journal of Molecular Biology*, v. 147, n. 1, pp. 195–197. ISSN: 0022-2836.

SUTTON, C., MCCALLUM, A., 2006, "Introduction to Conditional Random Fields for Relational Learning". In: Getoor, L., Taskar, B. (Eds.), *Introduction to Statistical Relational Learning*, MIT Press.

WAHAB, H., AHMAD KHAIRUDIN, N., SAMIAN, M., et al., 2006, "Sequence analysis and structure prediction of type II Pseudomonas sp. USM 4-55 PHA synthase and an insight into its catalytic mechanism", *BMC Structural Biology*, v. 6, n. 1, pp. 23. ISSN: 1472-6807.

WATERHOUSE, A. M., PROCTER, J. B., MARTIN, D. M. A., et al., 2009, "Jalview Version 2—a multiple sequence alignment editor and analysis workbench", *Bioinformatics*, v. 25, n. 9, pp. 1189–1191.

WATSON, J. D., CRICK, F. H. C., 1953, "Molecular Structure of Nucleic Acids: A Structure for Deoxyribose Nucleic Acid", *Nature*, v. 171, n. 4356 (April), pp. 737–738.

WERNERSSON, R., PEDERSEN, A. G., 2003, "RevTrans: multiple alignment of coding DNA from aligned amino acid sequences", *Nucleic Acids Research*, v. 31, n. 13, pp. 3537–3539.