FRAMEWORK FOR EXTRACTION OF PROTEIN FUNCTIONS AND INTERACTIONS FROM BIOMEDICAL LITERATURE

A DISSERTATION

Submitted in partial fulfillment of the requirements for the award of the degree
of
MASTER OF TECHNOLOGY
in
COMPUTER SCIENCE AND ENGINEERING

By

MUMMIDI LAKSHMI NARAYANA

DEPARTMENT OF ELECTRONICS & COMPUTER ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY ROORKEE
ROORKEE -247 667 (INDIA)
MAY, 2007

Candidate's Declaration

I hereby declare that the work being presented in the dissertation report titled "Framework for Extraction of Protein Functions and Interactions from Biomedical Literature" in partial fulfillment of the requirement for the award of the degree of Master of Technology in Computer Science and Engineering, submitted in the Department of Electronics and Computer Engineering, Indian Institute of Technology Roorkee, is an authenticate record of my own work carried out under the guidance of Dr. R. C. Joshi, Professor, Department of Electronics and Computer Engineering, Indian Institute of Technology Roorkee.

I have not submitted the matter embodied in this dissertation report for the award of any other degree.

M. L. Narayana.

Dated: 15-5-07 **(Mummidi Lakshmi Narayana)**

Place: IIT Roorkee.

Certificate

This is to certify that above statements made by the candidate are correct to the best of my knowledge and belief.

[illegible]

Dated: 15.5.07 **Dr. R. C. Joshi,**

Place: IIT Roorkee.

Professor,

Department of Electronics and

Computer Engineering, IIT Roorkee,

Roorkee -247667 (India).

ACKNOWLEDGEMENTS

I am thankful to Indian Institute of Technology Roorkee for giving me this opportunity. It is my privilege to express thanks and my profound gratitude to my supervisor Prof. R. C. Joshi for his invaluable guidance and constant encouragement throughout the dissertation. I was able to complete this dissertation in this time due to constant motivation and support obtained from Prof. R. C. Joshi.

I am also grateful to the staff of software laboratory and Research Scholar's laboratory for their kind cooperation extended by them in the execution of this dissertation. I am also thankful to all my friends who helped me directly and indirectly in completing this dissertation.

Most importantly, I would like to extend my deepest appreciation to my family for their love, encouragement and moral support. Finally I thank God for being kind to me and driving me through this journey.

M. L. Narayana.

(MUMMIDI LAKSHMI NARAYANA)

Abstract

Significant advances are being made in the field of biomedicine. The biomedical researchers present their research finally in the form of text. It has become a challenge for biomedical researchers to search for the latest discoveries of protein functions and interactions and required information in the ever increasing published literature. Text mining thus becomes helpful in conversion of this information into structured database format. In this report, we propose the method of function phrase chunking to extract protein functions and interaction phrase chunking for protein-protein interactions from biomedical literature. The proposed methods can be used to assist database curators in interpreting protein function and interaction terms and to aid bio-medical researchers in searching protein functions and protein-protein interactions from biomedical text literature. The experimental results on more than 0.12 million abstracts of protein functions and 0.17 million abstracts of protein-protein interactions shows the effectiveness of our system. The high precision and recall ranges of 87.6%-70% and 70.6%-59% respectively of our protein function extraction system and similarly the high precision and recall ranges of 94%-85% and 90%-73% respectively of protein-protein interaction extraction system indicate the outperforming efficiency of our system in comparison with existing systems.

Table of Contents

1.1 Introduction

In this era of genomics, understanding protein functions (PFs) and protein-protein interactions (PPIs) is one of the most challenging and important goal. Research in the area of proteins and their behavior is increasing. However, since most of the research is published in journal or conference papers instead of being directly accessible from a database, it consumes a lot of time for biomedical researchers to read volumes of papers to understand functions of the proteins and interactions of the proteins of interest. Text mining thus becomes an indispensable technique to aid biomedical researchers in surveying them rapidly. Hence, our protein functions and interactions extraction system aims at providing knowledge of protein functions and protein interactions in structured database form.

There has been much study towards the application of text mining techniques to automatically extract knowledge from biomedical literature [1-3]. The need of biomedical researchers can be summarized as follows: They have a list of proteins they are interested in and they want to know which papers describe these proteins and what properties of these are reported in these papers. In recent years, the study has greatly advanced from term recognition, such as gene and protein names [4-6] to extraction of complex relationships between different types of terms, such as protein sub-cellular location [7], biomolecular relations [8], pathways [9], protein functions [10-11] and protein-protein interactions [12-13]. Literature information is also integrated into biological analysis.

For most of these issues, recognition of gene or protein names in text is the first step and has been well studied to be near a mature technique. Extracting protein function and protein-protein interaction is a challenging task which needs more sophisticated approaches of natural language processing.

1.2 Motivation

Electronic storage of literature and access to it does not help in extracting and discovering the knowledge about the biomedical entities. Database curators and researchers use PubMed (PubMed Central (PMC) is the U.S. National Institutes of Health (NIH) free digital archive of biomedical and life sciences journal literature) to perform complex searches to retrieve documents and spend substantial time reading and extracting important knowledge from them. This is a time consuming, manual method that requires reading many texts, often heterogeneous in structure, contents and semantics. Nevertheless, electronic storage has made it possible to apply text-based knowledge discovery tools and methods (Text Mining) to discover precise and fine-grained facts and relationships. Some of the examples include identification of protein-protein interactions and prediction of protein functions. Moreover, by combining the electronically available biomedical literature to other data resources, such as ontology, information extraction can be customized. The terms from ontology can be extracted from the literature and can be associated to the biological entities.

Most of the cases, the main cause of diseases in living beings is malfunctioning of proteins or improper interactions between the proteins, so inorder to find the root cause of a disease one has to identify the protein that is malfunctioning or the type of interaction that is causing the disease. The extraction of protein functions and interactions from biomedical literature helps in identifying the function that a particular protein is supposed to do, the types of interactions that a protein should participate and helps in identifying the malfunctioning and improper interacting proteins. Thereby the diseases can be cured by making the protein function in proper way by administering medicines. There are many techniques which dealt with the extraction of these function and interaction terms from biomedical literature.

Recently, the state-of-the-art methods in text mining were presented in a competition for assessment of text mining systems in biology, the BioCreAtIvE (Critical Assessment of Information Extraction systems in Biology) (BioCreAtIvE, 2003). One of the two "biologically meaningful" tasks defined by BioCreAtIvE was the automatic extraction of functional annotations to proteins from full-text documents related to them, by using the Gene Ontology (GO) classification system. Among the 20 participants, the best annotations were achieved with a perfect prediction percentage equal to 11.80%, which is still too low. This shows that automated methods for functional annotation of genes are still far from being perfect. And also the automated systems for the extraction of protein-protein interactions are performing with low accuracy rates.

These observations have motivated the need of an automatic text mining tool which extracts the protein functions and protein-protein interactions efficiently with higher accuracies from biomedical literature.

1.3 Problem Statement

The problem is to extract the protein functions and protein-protein interactions from biomedical literature using a text mining technique called phrase chunking, a variant of text chunking. In this dissertation we have made an attempt to design and implement the framework to solve the mentioned problem. The main problem can be further divided into the following subtasks:

- To classify the input biomedical text articles into the either articles related to protein function and protein-protein interaction or not.
- To tagg the names of proteins in the texts using a tagger which uses Conditional Random Fields.
- To extract the actual term from the text describing the protein function or protein-protein interaction.

1.4 Organization of the Report

This dissertation proposes a new and efficient technique for the extraction of protein function and protein-protein interactions from the biomedical literature. The organization of the dissertation is as follows:

Chapter 2 gives the background of protein functions and protein-protein interactions, description of some well known information extraction techniques in this field.

Chapter 3 gives the description of data mining techniques that are used in the proposed framework for the extraction of the protein function and protein-protein interactions.

Chapter 4 describes the proposed framework of the protein information extraction with detailed description of each module.

Chapter 5 discusses the performance metrics used, the data set used for the training and testing purpose, the performance of the system, the snapshots of various screens and graphs depicting the performance.

Chapter 6 concludes the dissertation work and gives suggestions for future work.

2.1 Gene Ontology

Gene Ontology (GO) - The Gene Ontology Consortium's ontology (http://www.geneontology.org), GO, provides a dynamic controlled vocabulary for all organisms, with sufficient flexibility to accommodate the constant changes in biological knowledge. GO is aimed at providing a controlled terminology for labeling gene functions in a more precise, reliable, computer-readable manner. It maintains three separate taxonomies of terms, namely, “Molecular Function”, “Biological Process”, and “Cellular Component”. Unlike other schemes, GO is not a tree-like hierarchy, but a directed acyclic graph (DAG), where any term may have more than one parent as well as zero, one, or more children. This permits a more complete and realistic description of a term. Protein functions of any organism are described using the gene ontology.

The shape of a protein determines its biological activity. A single protein may have varying structure and more than one function. Proteins have many different biological functions. Proteins are classified according to their biological roles.

Enzymatic Proteins: The most varied and most highly specialized proteins are those with catalytic activity--the enzymes. Virtually all the chemical reactions of organic biomolecules in cells are catalyzed by enzymes. Many thousands of different enzymes, each capable of catalyzing a different kind of chemical reaction, have been discovered in different organisms. Digestive enzymes hydrolyze the polymers in food.

Transport Proteins: These proteins are involved in transporting other substances. For example, hemoglobin, the iron-containing protein of blood, transports oxygen from the lungs to other parts of the body. Other proteins transport molecules across cell membranes.

Structural Proteins: Structural proteins are very important for support. Collagen and elastin provide a fibrous framework in animal connective tissues, such as tendons and ligaments. Keratin is the protein of hair, horns, feathers, quills, and other skin appendages of animals.

Storage Proteins: These proteins store amino acids. Ovalbumin is the protein of egg white, used as an amino acid source for the developing embryo. Casein, the protein of milk, is the major source of amino acids for baby mammals. Plants store proteins in seeds.

Hormonal Proteins: Hormonal proteins coordinate the bodily activities. Insulin, a hormone secreted by the pancreas, helps regulate the concentration of sugar in the blood.

Receptor Proteins: Receptor proteins are built into the membrane of a nerve cell and they detect chemical signals released by other nerve cells. They are involved in the cell's response to chemical stimuli.

Contractile Proteins: These proteins are very important in movement. Actin and myosin are responsible for the movement of muscles. Contractile proteins are responsible for the undulations of cilia and flagella, which propel many cells.

Defensive Proteins: These proteins protect against diseases. Antibodies combat bacteria and viruses.

Gene ontology describes the protein functions using three terms namely Molecular functions, Biological processes and Cellular components. Each of these is described in detail as follows:

2.1.1 Molecular functions

Molecular function describes activities, such as catalytic or binding activities, that occur at the molecular level. GO molecular function terms represent activities rather than the entities (molecules or complexes) that perform the actions, and do not specify where or when, or in what context, the action takes place. Molecular functions generally correspond to activities that can be performed by individual gene products, but some activities are performed by assembled complexes of gene products. Examples of broad functional terms are catalytic activity, transporter activity, or binding; examples of narrower functional terms are adenylate cyclase activity or Toll receptor binding.

The following molecular function terms have standard definitions:

***x* binding**
Interacting selectively with *x*.
[Enzyme] activity
Catalysis of the reaction: [reaction catalyzed by enzyme].
***x* receptor activity**
Combining with *x* to initiate a change in cell activity.
***x* transporter activity**
Enables the directed movement of *x* into, out of, within or between cells.

2.1.2 Biological Processes

A biological process is a process of a living organism. Biological processes are made up of any number of chemical reactions or other events that result in a transformation. A biological process is series of events accomplished by one or more ordered assemblies of molecular functions. Examples of broad biological process terms are cellular physiological process or signal transduction. Examples of more specific terms are pyrimidine metabolism or alpha-glucoside transport. It can be difficult to distinguish between a biological process and a molecular function, but the general rule is that a process must have more than one distinct steps.

Regulation of biological processes occurs where any process is modulated in its frequency, rate or extent. Biological processes are regulated by many means; examples include the control of gene expression, protein modification or interaction with a protein or substrate molecule.

Biological processes are often regulated by genetics. Mutant phenotypes may lead to interruptions to a biological process.
Biological processes include:

- Cell adhesion, the attachment of a cell, either to another cell or to an underlying substrate such as the extracellular matrix, via cell adhesion molecules.
- Intercellular communication, any process that mediates interactions between a cell and its surroundings. Encompasses interactions such as signaling or

attachment between one cell and another cell, between a cell and an extracellular matrix, or between a cell and any other aspect of its environment.

- Morphogenesis, cell growth and cellular differentiation
- Cell physiological process, the processes pertinent to the integrated function of a cell.
- Cell recognition, the process by which a cell in a multicellular organism interprets its surroundings.
- Physiological process, those processes specifically pertinent to the functioning of integrated living units: cells, tissues, organs, and organisms.
- Pigmentation
- Biological reproduction
- Response to stimulus, a change in state or activity of a cell or an organism (in terms of movement, secretion, enzyme production, gene expression, etc.) as a result of a stimulus.
- Interaction between organisms, the processes by which an organism has an observable effect on another organism of the same or different species.
- Also fermentation, fertilization, germination, geotropism, heliotropism, hybridization, metamorphosis, photosynthesis, transpiration.

2.1.3 Cellular component

A cellular component is just that, a component of a cell, but with the provision that it is part of some larger object; this may be an anatomical structure (e.g. rough endoplasmic reticulum or nucleus) or a gene product group (e.g. ribosome, proteasome or a protein dimer). Biological matter or biological material refers to the unique, highly organized substances of which cellular life is composed of, for instance membranes, proteins, and nucleic acids. They may also be called cellular components.

Most biological matter has the characteristics of soft matter, being governed by relatively small energies. All known life is made of biological matter. To be differentiated from other theoretical or fictional life forms, such life may be called *carbon-based, cellular, organic, biological,* or even simply *living*—as some definitions of life exclude alternative biochemistry.

The following cellular component terms have standard definitions:

***Organelle* envelope**

The double lipid bilayer enclosing the *organelle* and separating its contents from the rest of the cytoplasm; includes the intermembrane space.

***Organelle* membrane, organelle with a single membrane**

The lipid bilayer surrounding a(n) *organelle*.

***Organelle* membrane, organelle with a double membrane**

Either of the lipid bilayers that enclose the *organelle* and form the *organelle* envelope.

***Organelle* inner membrane**

The inner, i.e. lumen-facing, lipid bilayer of the *organelle* envelope.

***Organelle* outer membrane**

The outer, i.e. cytoplasm-facing, lipid bilayer of the *organelle* envelope.

***Organelle* membrane lumen**

The region between the inner and outer lipid bilayers of the *organelle* envelope.

2.2 Protein-Protein Interactions

Protein-protein interactions (PPIs) (http://en.wikipedia.org/wiki/Protein-protein_interaction) refer to the association of protein molecules. The interactions between proteins are important for many biological functions. For example, signals from the exterior of a cell are mediated to the inside of that cell by protein-protein interactions of the signaling molecules. This process, called signal transduction, plays a fundamental role in many biological processes and in many diseases (e.g. cancer). Proteins might interact for a long time to form part of a protein complex, a protein may be carrying another protein (for example, from cytoplasm to nucleus or vice versa in the case of the nuclear pore importins), or a protein may interact briefly with another protein just to modify it (for example, a protein kinase will add a phosphate to a target protein). This modification of proteins can itself change protein-protein interactions. For example, some proteins with SH2 domains only bind to other proteins when they are phosphorylated on the amino acid tyrosine. In conclusion, protein-protein interactions are of central importance for virtually every process in a living cell. Information about these interactions improves our understanding of diseases and can provide the basis for new therapeutic approaches.

2.3 Text Mining In Biomedicine

In biomedical science, the increasing need for obtaining semantic information from biomedical literature and its easy availability has encouraged many research groups to develop text mining methods for automatic analysis and extraction of facts from it. The earlier works focused on tasks needing limited linguistic context and processing at the level of words, such as identifying protein names in literature [4-6], or on tasks relying on word co-occurrences [14] and pattern matching. Later linguistic techniques were used to interpret the data from expression array experiments [15] and to handle biological relations represented in complex sentences, such as protein localization, protein function identification [10-11] and protein-protein interactions [12-13]. Finally, there was the emergence of natural language technologies to handle more complex relations across sentences. The common aim of all these approaches is to acquire knowledge about the functions or interactions of protein or genes.

In the biomedical field, text mining can be used for the following purposes:

- Extract gene and protein names from biological literature.
- Uncover underlying themes or concepts contained in large document collections, such as in EDGAR, an application developed by Rindesch et al. [16] for extracting drug, genes and relations among them.
- Develop knowledge repositories by automatic knowledge acquisition from biomedical literature, like creating ontology.
- Classify documents into predefined categories and sub-categories based on their contents.
- Discover knowledge about biological processes and their regulation, for example, prediction of metabolic pathways [9] and acquisition of knowledge about functions of genes from microarray data.
- Cluster documents automatically, into categories that are intelligently selected from the words and phrases contained within the documents themselves.
- Identify the relationships between sets of concepts, extracted from the text in a graphical format. This "lexical network" enables the end user to identify previously unrecognized or unknown relationships in the contents. For example, construction of a protein-protein interactions network based on the associations identified from the freely available biomedical literature [17].

2.4 Prôtein Function and Interaction Extraction Related Work

2.4.1 Protein Function Extraction Systems

This section provides the various systems developed to extract the protein functions from biomedical literature using various text mining techniques.

a. Mining protein function from text using term-based support vector machines

Simon B. Rice et al. [18] employed a supervised machine learning approach to assignment of GO terms to proteins, together with an extensive terminological processing of documents (which aimed at generation of relevant features for classification and protein annotation). They based their method on SVMs, which has been demonstrated to perform well at the document classification task, as they construed the protein function assignment task as a modified form of this problem. The approach is mainly based on the idea that biological entities (represented by domain terms) that co-occur in text with a protein of interest are indicative of its function, and that proteins with similar co-occurrences of terms have related roles. Consequently, learning relevant and informative co-occurring terms for a given GO term should give clues for assignment of that GO term to proteins that have similar distributional patterns.

Assignments of GO terms (both for learning and predicting) were based on collecting "weak" co-occurrence evidence within documents, rather than on explicit statement(s) of protein function. Therefore, an important facet of this approach was that GO assignments were not derived from a single, "relevant" passage or sentence, but from document(s) relevant to a given protein. Further, selection of supporting passages (as minimal retrieval units they used paragraphs as tagged in an SGML-tagged version of distributed documents) was based on a similar idea. Each paragraph pertaining to a given protein was assessed with respect to a given GO term, and the highest scoring passage was selected. More specifically, the employed method involved three steps: a) pre-processing of documents and feature selection, b) training the SVMs on the released training data, and c) predicting GO terms and selection of paragraphs for target (testing) genes.

b. Automatic extraction of gene/protein biological functions from biomedical text
A.Koike et al. [11] proposed automatic extraction of gene/protein biological functions from biomedical text using the following procedure. The steps are as follows.

Step 1. Recognition of gene/protein/family names and GO functional terms
Gene name recognition was carried out using the GENA gene name dictionary (http://gena.ontology.ims.u-tokyo.ac.jp/search/servlet/gena) and family name dictionary (http://marine.ims.u-tokyo.ac.jp:8080/Dict/family), which was constructed based on major database entries. In this system, a protein name that does not specify the gene locus was treated as a family name. For example, since '14-3-3' does not specify the gene locus ('14-3-3 alpha', '14-3-3 beta', etc.), it was registered as a protein family name. The variations in gene name were generated based on these dictionaries and were quickly searched against abstracts using a devised trie with many heuristics, such as replacing special characters with spaces, searching inside and outside the parenthesis separately [e.g. mitogen-activated protein kinase (MAPK) 1→mitogen-activated kinase 1+MAPK1], and using continuous expressions (e.g. GATA-4/5/6→GATA4, GATA5, GATA6).

Step 2. Shallow parsing, noun phrase bracketing and sentence structure analysis
Shallow parsing was done for sentences with gene name IDs using FDG-Lite (http://www.connexor.com/). After noun phrase bracketing using dependency/syntactic tags and morphological tags, parentheses, coordinate clauses, subordinate clauses, etc. were analyzed using various standard rules.

FDG-Lite, developed by Voutilainen et al. at the University of Helsinki, gives the base form, dependency/syntactic tags and morphological tags. When a determiner, adverbial and adjective modifiers, coordinating conjunction, participle, noun and pronoun are contiguous, they are regarded as a noun phrase. Boundary recognition of noun phrases including a coordinating conjunction and comma requires the use of certain devices. The number of coordinating conjunctions before the target coordinating conjunction, whether or not a 'past_participle_modifier' is located after the target coordinating conjunction, whether or not the verb is before or after the target coordinating conjunction, and whether or not the target coordinating conjunction is in a subordinate phrase or adverbial phrase beginning with an interrogative are checked for the boundary of the noun phrase including coordinating conjunctions and comma.

In principle, a predecessor noun phrase of the predicate verb was regarded as a subject, and just behind the noun phrase or preposition phrase of the predicate verb was regarded as an object. Certain rules were used for complicated sentence structures, such as coordinate-conjunction and insertion-phrase structures.

Step 3. ACTOR–OBJECT relationships extraction The gene–function relationships were extracted when they were expressed in ACTOR–OBJECT relationships with predefined verbs or in modification relationships. Here, ACTOR (agent) means the doer of action and OBJECT means the receiver of action (higher concept of 'object' of subject–object). Basically, only when 'ACTOR' is a gene name and 'OBJECT' is a gene function, the relationship was extracted. For some verbs, such as 'require', the reverse relation was extracted. They have used these terms, since relationships between ACTOR/OBJECT and gene name/function are not affected by the passive voice or active voice although subject–object relationships are affected (in most cases, the subject is protein and the object is its function in active voice, while the opposite holds true in passive voice).

c. Literature Extraction of Protein Functions Using Sentence Pattern Mining

Jung-Hsien Chiang and Hsu-Chun Yu proposed a technique [19] for literature extraction of protein functions. This technique used sentence pattern mining for the purpose. Sentence pattern mining involves usage of predefined patterns (rules) to identify the protein function from the texts. The input documents can be full text articles or abstracts and the output is protein-GO-document relations. Input documents were processed through the steps of preprocessing, protein name indexing, GO term indexing, co-occurrence extraction, and phrase parsing to transform sentences into phrase structures. Then, the work flow was divided into two phases: mining and matching. In the mining phase, sentence patterns were mined from sample sentences that describe protein functions. In the matching phase, these sentence patterns were then matched with new sentences to extract protein-GO-document relations.

Recognition of GO Term Variants

Since authors describe protein functions in various forms instead of following the controlled vocabulary of GO, many terms in the articles are equivalent to GO terms in meaning, but appear in different forms. The recognition of these variants is, hence, a critical issue for robust GO term indexing. They have classified variations of GO terms into three major categories: *morphological*, *syntactic*, and *semantic*.

Morphological variants: One or more words of the original term are replaced with their morphologically related words in the variant and the other words remain unchanged. For example, cellular membrane is a morphological variant of the term cell membrane (GO: 0005886).
Syntactic variants: The content words of the original term are found in the variant, but the syntactic structure of the term is modified. For example, transport from the ER to the Golgi is a syntactic variant of ER to Golgi transport (GO: 0006888), and binding to the origin of DNA replication is a variant of DNA replication origin binding (GO: 0003688).
Semantic variants: One or more words of the original term are replaced with their synonyms in the variant and the other words remain unchanged. For example, delivery of copper ion is a semantic variant of copper ion transport (GO: 0006825).

Morphological variants were identified by adopting the Java Lexical Tools, which uses the UMLS SPECIALIST Lexicon to handle lexical variants. For each word in a GO term, they have used the Lexical Variant Generation (LVG) program of the Java Lexical Tools to generate its inflectional and derivational variants. Inflectional variants include the singular and plural forms of nouns, the various tenses of verbs, and the positive, comparative, and superlative of adjectives and adverbs. Derivational variants are words that change syntactic category from the original words.

To cope with syntactic variants, they have mined GO variation rules from biomedical literature. These variation rules represent common formats of changes between a GO term and its variant. Variation rules have the following format:

$$(X_i \mid Y_j)+ \rightarrow (X_i \mid Y_k)+$$

where the antecedent is a GO term and the consequent is its variant. X_i are token sequences that both appear in the term and the variant and X_i in the variant can be morphological variants of X_i in the term. Y_j are token sequences that appear only in the term, i.e., deleted parts of the term in the variant. Y_k are token sequences that appear only in the variant, i.e., inserted parts of the term in the variant. The antecedent and the consequent are represented by regular expressions and, hence, the symbol | represents "or" and + means that the previous item occurs one or more times.

To deal with semantic variants, they have compiled synonyms of GO terms with the following three methods:

1. They utilized the mappings of other classification systems to GO, provided by the Gene Ontology Consortium. GO is not the only attempt to build structured controlled vocabularies from genome annotation; hence, it makes translation tables between concepts of other classification system and GO. We adopt the mappings of UniProt Knowledgebase, EGAD, TIGR Role, and MIPS Funcat at present.
2. The Lexical Variant Generation program was used to generate synonyms of words in GO terms.
3. They have compiled synonyms of GO term words from GO itself and from biomedical literature. Each pair of term and synonym in GO is compared to collect word pairs, one of which appears only in the term and the other appears only in the synonym. The frequencies of these word pairs in GO are counted and high-frequency word pairs are screened to acquire correct synonyms.

Sentence Pattern Mining

Observing the sentences that report protein functions, they could find plenty of sentence patterns, i.e., wording or writing styles, commonly used by authors to describe protein functions, e.g., "<protein> participates in <GO>" and "<protein> is localized to <GO>." These sentence patterns were very useful characteristics for identifying sentences describing protein functions. They defined sentence patterns as follows:

A sentence pattern

$$SP = \{C_{prefix} E_1 C_{\inf ix} E_2 C_{suffix}\}$$

is a sequence of parsed phrases. E_1 *and* E_2 *are parsed phrases, which represent named entities to be extracted. One of the parsed tokens in* E_1 *has a "P" slot and one of the parsed tokens in* E_2 *has a "G" slot and vice versa.* $C_{prefix}, C_{\inf ix}$ *and* C_{suffix} *are sequences of parsed phrases which represent contextual phrases of the named entities.*

To acquire sentence patterns from text, the problem of sentence pattern mining is defined as follows:

Given a set of co-occurrence sentences which have been transformed into parsed phrases, sentence pattern mining is to find the complete set of sentence patterns in this set of sentences.

For the purpose of sentence pattern mining, they have divided the co-occurrence sentences, which have been parsed for phrases, into positive and negative examples. Positive examples are co-occurrence sentences where the occurring protein has the co-occurring GO function in database annotation, and negative examples are the other cooccurrence sentences. Sentence pattern mining consists of three steps:

1. Candidate sentence patterns are mined from the positive examples by aligning each pair of sentences.
2. The support and confidence of each candidate sentence pattern is calculated by matching the pattern with each positive or negative example.
3. Candidate sentence patterns are screened according to their support and confidence levels, in order to acquire appropriate sentence patterns.

2.4.2 Protein-Protein Interaction Extraction Systems

This section provides the various systems developed earlier for the extraction of protein-protein interactions from biomedical literature using various text mining techniques.

a. Extraction using Longest Common Subsequences (ELCS)

Blaschke et al. [15] manually developed rules for extracting interacting proteins. Each of their rules (or frames) is a sequence of words (or POS tags) and two protein-name tokens. Between every two adjacent words is a number indicating the maximum number of intervening words allowed when matching the rule to a sentence. In

Bunescu et al. [20], they described a new method ELCS (Extraction using Longest Common Subsequences) that automatically learns such rules. ELCS' rule representation is similar to that in Blaschke et al. [15], except that it currently does not use POS tags, but allows disjunctions of words.

- (7) interaction (0) [between | of] (5) PROT (9) PROT (17) .

shows an example of a rule learned by ELCS. Words in square brackets separated by '|' indicate disjunctive lexical constraints, i.e. one of the given words must match the sentence at that position. The numbers in parentheses between adjacent constraints indicate the maximum number of unconstrained words allowed between the two (called a *word gap*). The protein names are denoted here with PROT. A sentence matches the rule if and only if it satisfies the word constraints in the given order and respects the respective word gaps.

b. Extraction using a Relation Kernel (ERK)

Both Blaschke and ELCS do interaction extraction based on a limited set of matching rules, where a rule is simply a sparse (gappy) subsequence of words (or POS tags) anchored on the two protein-name tokens. Therefore, the two methods share a common limitation: either through manual selection (Blaschke), or as a result of the greedy learning procedure (ELCS), they end up using only a subset of all possible anchored sparse subsequences. Ideally, they would want to use all such anchored sparse subsequences as features, with weights reflecting their relative accuracy. However explicitly creating for each sentence a vector with a position for each such feature is infeasible, due to the high dimensionality of the feature space. Here they could exploit an idea used before in string kernels [21]: computing the dot-product between two such vectors amounts to calculating the number of common anchored subsequences between the two sentences. This could be done very efficiently by modifying the dynamic programming algorithm from [21] to account only for anchored subsequences i.e. sparse subsequences which contain the two protein-name tokens. Besides restricting the word subsequences to be anchored on the two protein tokens, they could further prune down the feature space by utilizing the following property of natural language statements: whenever a sentence asserts a relationship between two entity mentions, it generally does this using one of the following three patterns:

- **[FI]** Fore–Inter: words before and between the two entities mentions are simultaneously used to express the relationship. Examples: ‘interaction of $< P_1 >$ with $< P_2 >$’, ‘activation of $< P_1 >$ by $< P_2 >$’.
- **[I]** Inter: only words between the two entity mentions are essential for asserting the relationship. Examples: ‘$< P_1 >$ interacts with $< P_2 >$’, ‘$< P_1 >$ is activated by $< P_2 >$’.
- **[IA]** Inter–After: words between and after the two entity mentions are simultaneously used to express the relationship. Examples: ‘$< P_1 >-< P_2 >$ complex’, ‘$< P_1 >$ and interact $< P_2 >$’.

3.1 Support Vector Machines

The Support Vector Machines (SVMs) [22] are very effective methods for general purpose supervised pattern recognition. The SVM approach is not only well founded theoretically because it is based on extremely well developed machine learning theory and Statistical Learning Theory [23], but is also superior in practical applications. The following subsections present the working and implementation of the SVM for our problem.

a. Classification using support vector machines

When used for classification, SVMs separate a given set of binary labeled data with a hyper-plane that is maximally distant from them. Since, most practical classification problems are non-linear; the SVMs employ a technique of kernels that automatically realizes a non-linear mapping to a feature space. The hyperplane found by the SVM in the feature space corresponds to a nonlinear boundary in the input space.

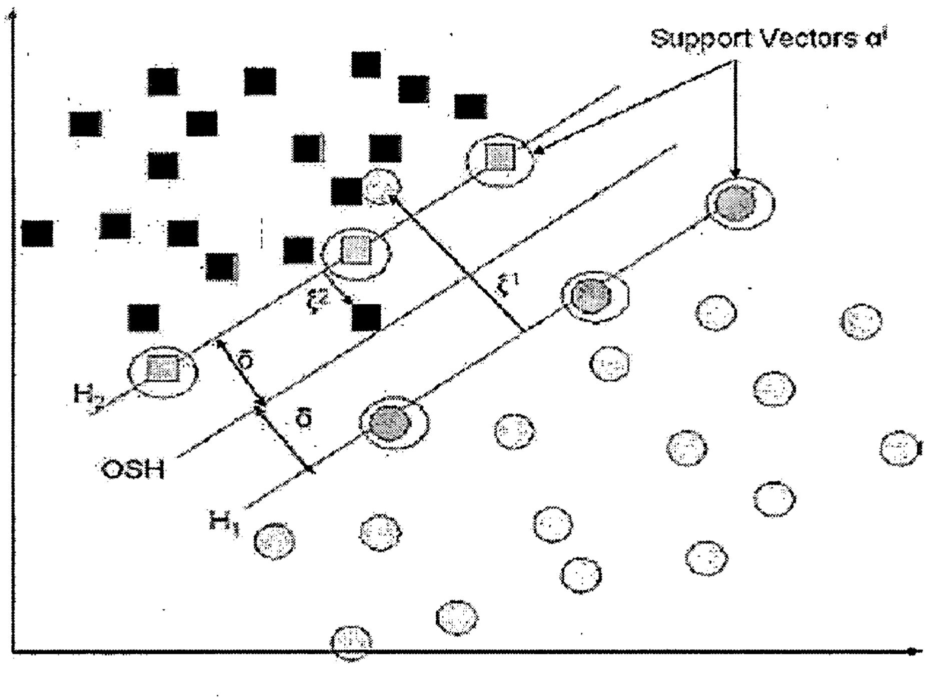


Fig. 3.1 The optimal separating hyperplane (OSH), support vectors α_i and the slack variables ξ_i

In their basic form, SVMs learn linear decision rules $h(\vec{x}) = sign(\vec{w}.\vec{x} + b)$ described by a weight vector $\vec{w}$ and a threshold b. Let the input be a sample of n training examples with the j^{th} input point being $x^j = (x_1^j, x_2^j, \ldots, x_n^j)$.

Let this input point be labeled by the random variable $Y^j \in \{-1,+1\}$. For a linearly separable input, the SVM finds the hyperplane with maximum Euclidean distance to the closest training examples. This distance is called the margin δ as depicted in Fig. 3.1. For non separable training sets, the amount of training error is measured using slack variable ξ^j as shown in Fig. 3.1 for a two class problem. Computing hyperplanes is equivalent to solving the following primal optimization problem.

minimize

$$V(\vec{w},\vec{b},\vec{\xi}) = \frac{1}{2}\vec{w}.\vec{w} + C\sum_{i=1}^{n}\xi^j \qquad (3.1.1)$$

subject to

$$\forall_{j=1}^{n} : y^j[\vec{w}.\vec{x}^j + b] \geq 1 - \xi^j \qquad (3.1.2)$$

$$\forall_{j=1}^{n} : \xi^j > 0 \qquad (3.1.3)$$

The second constraint requires that all the training examples are classified properly up to a slack ξ. Therefore, $\sum_{j=1}^{n}\xi^j$ is an upper bound on the number of training errors. The factor C in Eq. (3.1.1) is a parameter that allows trading off training error verses model complexity. Note that the margin of the resulting hyperplane is $\delta = 1/\|\vec{w}\|$. The hyperplane that separates the positive from the negative examples and has maximal margin is called the maximal margin hyperplane or the optimal separating hyperplane (OSH) as shown in Fig. 3.1. The hyperplanes that contain the training points with the minimal distance to the OSH are called the margin hyperplanes and they form the boundary of the margin. They are represented as $H1$ and $H2$ in Fig. 3.1.

b. Parameter and kernel selection

The performance of SVM classification is strongly related to the choice of the kernel function and the penalty parameter C. There are a large number of kernel functions available. In general, radial basis function (RBF) is a reasonable first choice. The RBF kernel non-linearly maps samples into a higher dimensional space, and can handle the case when the relation between class labels and attributes is nonlinear. The RBF kernel can be described as

$$k(x,z) = \exp(-\gamma \times \|x-z\|^2) \tag{3.1.4}$$

For finding the optimum values of parameters *(C, γ)* automatically, a grid search technique is used using cross validation. Basically pairs of *(C, γ)* are tried and the one with the best cross-validation accuracy is picked.

In our work, we have used SVM based classification for classifying the input text articles. The articles are classified as being related to protein functions and interactions or not. The complete discussion is given in next chapter.

3.2 Conditional Random Fields (CRFs)

Conditional random fields [24] are probabilistic models that were designed for segmenting and labeling sequence data. The following subsections present the working of the CRF.

a. Conditional Random Fields

Let $o = \{o_1,...,o_T\}$ be some input data observation sequence. Let S be a finite set of states, each is associated with a label $l(\in \mathbf{L} = \{l_1,...,l_Q\})$. Let $s = \{s_1,...,s_T\}$ be some state sequence. CRFs are defined as the conditional probability of a state sequence given an input observation sequence as follows,

$$p_\theta(s \mid o) = \frac{1}{Z(o)} \exp\left(\sum_{t=1}^{T} F(s,o,t)\right) \tag{3.2.1}$$

Where $Z(o) = \sum_{s'} \exp\left(\sum_{t=1}^{T} F(s',o,t)\right)$ is a normalized factor summing over all label sequences. F(s, o, t) is the sum of CRF features at time position t:

$$F(s,o,t) = \sum_i \lambda_i f_i(s_{t-1}, s_t) + \sum_j \lambda_j g_j(o, s_t) \tag{3.2.2}$$

in which f_i and g_j are *edge* and *state* feature functions, respectively. λ_i and $\lambda_j (\in \theta = \{\lambda_1, \lambda_2, ...\})$ are the feature weights associated with f_i and g_j.

$$f_i(s_{t-1}, s_t) \equiv [s_{t-1} = l'][s_t = l]$$

$$g_j(o,s_t) \equiv [x_j(o,t)][s_t = l]$$

where $s_t = l$ means that label l is associated with state s_t. And $x_j(o,t)$ is a logical context predicate that indicates whether or not the observation sequence o (at time t) holds a particular property or fact of empirical data.

b. Inference in CRFs

Inference in CRFs is to find the most likely state sequence s^* given the input observation sequence o,

$$s^* = \arg\max_s p_\theta(s \mid o) = \arg\max_s \exp\left(\sum_{t=1}^{T} F(s,o,t)\right) \tag{3.2.3}$$

To find s^*, one can apply the dynamic programming using the Viterbi algorithm [25]. To avoid an exponential-time search over all possible settings of s, Viterbi stores the probability of the most likely path up to time t which accounts for the first t observations and ends in state s_i. We denote this probability to be $\varphi_t(s_i)(0 \le t < T)$ and $\varphi_0(s_i)$ to be the probability of starting in each state s_i. The recursion is given by:

$$\varphi_{t+1}(s_i) = \max_{s_j}\{\varphi_t(s_j)\exp F(s,o,t+1)\} \tag{3.2.4}$$

The recursion terminates when $t = T\text{-}1$ and the biggest value is $p^* = \arg\max_i \varphi_T(s_i)$. At this time, we can backtrack through the stored information to find the most likely sequence s^*.

c. Training CRFs

CRFs are trained by searching the set of weights $\theta = \{\lambda_1, \lambda_2, \ldots\}$ to maximize the log-likelihood, L, of a given training data set $D = \{o^{(j)}, s^{(j)}\}_{j=1\ldots N}$:

$$L = \sum_{j=1}^{N} \log(p_\theta(s^{(j)} \mid o^{(j)})) - \sum_k \frac{\lambda_k^2}{2\sigma^2} \tag{3.2.5}$$

where the second sum is a Gaussian prior over feature weights with variance σ^2, which provides smoothing to deal with sparsity in the training data.

When the labels make the state sequence unambiguous, the likelihood function in exponential models such as CRFs is convex, thus searching the global optimum is guaranteed. However, the optimum cannot be found analytically. Parameter estimation for CRFs requires an iterative procedure. It has been shown that quasi-

This method can avoid the explicit estimation of the Hessian matrix of the log-likelihood by building up an approximation of it using successive evaluations of the gradient.

L-BFGS is a limited-memory quasi-Newton procedure for convex optimization that requires the value and the gradient vector of the function to be optimized. Let $s^{(j)}$ denote the state path of training sequence j in the training set D, then the log-likelihood gradient component of λ_k is

$$\frac{\delta L}{\delta \lambda_k} = \left[\sum_{j=1}^{N} C_k(s^{(j)}, o^{(j)})\right] - \left[\sum_{j=1}^{N}\sum_{s} p_\theta(s \mid o^{(j)}) C_k(s, o^{(j)})\right] - \frac{\lambda_k}{\sigma^2} \qquad (3.2.6)$$

where $C_k(s, o)$ is the count of feature f_k given s and o. The first two terms correspond to the difference between the empirical and the model expected values of feature f_k. The last term is the first-derivative of the Gaussian prior.

d. Second-order Conditional Random Fields

Although the first-order Markov CRFs described above perform well for many segmenting and labeling tasks, they fail to encode the long-range interactions among states due to the limitation of the first-order Markov dependency (i.e., the current state depends only on one previous state). The second-order (Markov) CRFs are stronger in capturing such interactions, and thus perform better on labeling/segmenting tasks where the sequential dependencies are essential facts for inference.

In the second-order CRFs, we divide features into four categories: edge feature type 1 (e^1), state feature type 1 (s^1), edge feature type 2 (e^2), and state feature type 2 (s^2). Only e^1 and s^1 are used for first-order CRFs and all of those four are used for second-order models. The sum of feature, F(s, o, t), is now rewritten as follows,

$$F(s,o,t) = \sum_i \lambda_i f_i(s_{t-1}, s_t) + \sum_j \lambda_j g_j(o, s_t) + \sum_k \lambda_k f_k(s_{t-2}, s_{t-1}, s_t) + \sum_h \lambda_h g_h(o, s_{t-1}, s_t)$$

(3.2.7)

where f_i (type e^1), g_j (type s^1), f_k (type e^2), and g_h (type s^2) are defined as follows,

$$f_i(s_{t-1}, s_t) \equiv [s_{t-1}s_t = l'l]$$

$$g_j(o, s_t) \equiv [x_j(o,t)][s_t = l]$$

$$f_k(s_{t-2}, s_{t-1}, s_t) \equiv [s_{t-2}s_{t-1} = l''l'][s_{t-1}s_t = l'l]$$

$$g_h(o, s_t) \equiv [x_h(o,t)][s_{t-1}s_t = l'l]$$

A feature of type e^1 is a special case of type s^2 if the logical predicate $x_h(o, t)$ is always true. Because t starts from 1, we need to add a pseudo-state s_0 at the beginning of each sequence. In principle, s_0 can be associated with any label l ($\in \mathbf{L} = \{l_1, ..., l_Q\}$). However, this would distort or influence the actual sequential dependencies among labels in training data. Therefore, it is better to use a pseudo-label l_0 for s_0. The label set is now $\mathbf{L} = \{l_0, l_1, ..., l_Q\}$.

Training for and inference in CRFs need an efficient forward-backward computation which manipulates on *transition matrix* M_t at every time position t of each sequence [24]. Unlike in first-order CRFs, the dimension of transition matrixes in second-order CRFs is $|\mathbf{L}|^2 \times |\mathbf{L}|^2$,

$$M_t[l''l'][l'l] = \exp F(s, o, t) \qquad (3.2.8)$$

Supposing that labels l'', l', and l are represented in integer numbers, the real index of $l'l$ is $l'|\mathbf{L}| + l$, and similarly for $l''l'$. The four types of features can be summed to build the transition matrix M_t as follows: feature type e^2 is corresponding to matrix cell $[l''l'][l'l]$; feature type e^1 and s^2 are corresponding to matrix column $[l'l]$; and feature s^1 is corresponding to matrix columns $[*l]$ (where * is an arbitrary label l').

e. Inference in Second-order CRFs

Inference in second-order CRFs using Viterbi algorithm also involves the transition matrixes. The recursive variable for second-order CRFs is as,

$$\varphi_{t+1}(s_j, s_i) = \max_{s_k s_j}\{\varphi_t(s_k, s_j) \exp F(s, o, t+1)\} \qquad (3.2.9)$$

where s_k, s_j, and s_i are states of time positions t -1, t, and t + 1, respectively.

If we have some constraints for Viterbi inference, we can apply them at this level. For example, every matrix cell $M_t[l''l_1'][l_2'l]$ must be zero if $l_1' \neq l_2'$ because a state cannot be associated with two different labels on the same label path. Also, if we want to prevent the occurrence of a particular pair of consecutive labels $l^u l^v$, we only need to set the column $[l^u l^v]$ of the transition matrix to zero. This will disable all label paths going through this pair of labels.

3.3 Phrase Chunking

Phrase chunking is a variant of relation extraction. It is the task of discovering semantic connections between entities. In text, this usually amounts to examining pairs of entities in a document and determining whether a relation exists between them. Common approaches to this problem include pattern matching, kernel [26], and logistic regression. The pair wise classification approach of kernel methods and logistic regression is commonly a two phase method: first the entities in a document are identified, and then a relation type is predicted for each pair of entities. This approach presents two difficulties: (1) enumerating all pairs of entities, even when restricted to pairs within a sentence, results in a low density of positive relation examples; and (2) errors in the entity recognition phase can propagate to errors in the relation classification stage. We avoid these difficulties by formulating above task of relation extraction as a sequence labeling task such as named entity recognition or part-of-speech tagging, so we can now apply models that have been successful on those tasks. The sequential labeling approach can handle many correlated features, as demonstrated in work on maximum-entropy [24].

Text chunking consists of dividing a text in syntactically correlated parts of words. For example, the sentence *He reckons the current account deficit will narrow to only # 1.8 billion in September.* can be divided as follows:
[NP He] [VP reckons] [NP the current account deficit] [VP will narrow] [PP to] [NP only # 1.8 billion] [PP in] [NP September].

The goal of this machine learning method is to recognize the chunk segmentation of the test data after the training as well as possible. The training data can be used for training the text chunker. The chunkers will be evaluated with the F rate, which is a combination of the precision and recall rates: F = 2*precision*recall / (recall+precision). The precision and recall numbers will be computed over all types of chunks.

Extraction of protein functions and protein-protein interactions involves lot of text processing techniques and computational methods. Here we have used a number of techniques to employ the system. These techniques range from wide areas of data mining such as preprocessing the text articles, parts-of-speech tagging of sentences, tagging the biomedical terms in the texts, classification of articles and relation extraction etc.

The framework of our proposed automated protein function (PF) and protein-protein interaction (PPI) extraction system is as shown in Fig. 4.1. The skeleton of the framework is same for both the protein function and protein-protein interaction system, except that the input articles, features used for the classification purpose and training data for the taggers vary.

Separate components are provided in the framework for the following:

- Preprocessing the input text articles.
- Feature extraction, the theme from the abstracts containing PFs and PPIs is extracted.
- Classification of abstracts for relevance with PFs and PPIs.
- The protein name tagging to extract the protein names.
- Classification to identify the sentence containing the function or interaction sentence. Phrase chunking to finally extract the protein function or protein-protein interaction term from the sentence.

Each of these components is discussed in detail in next sub-sections.

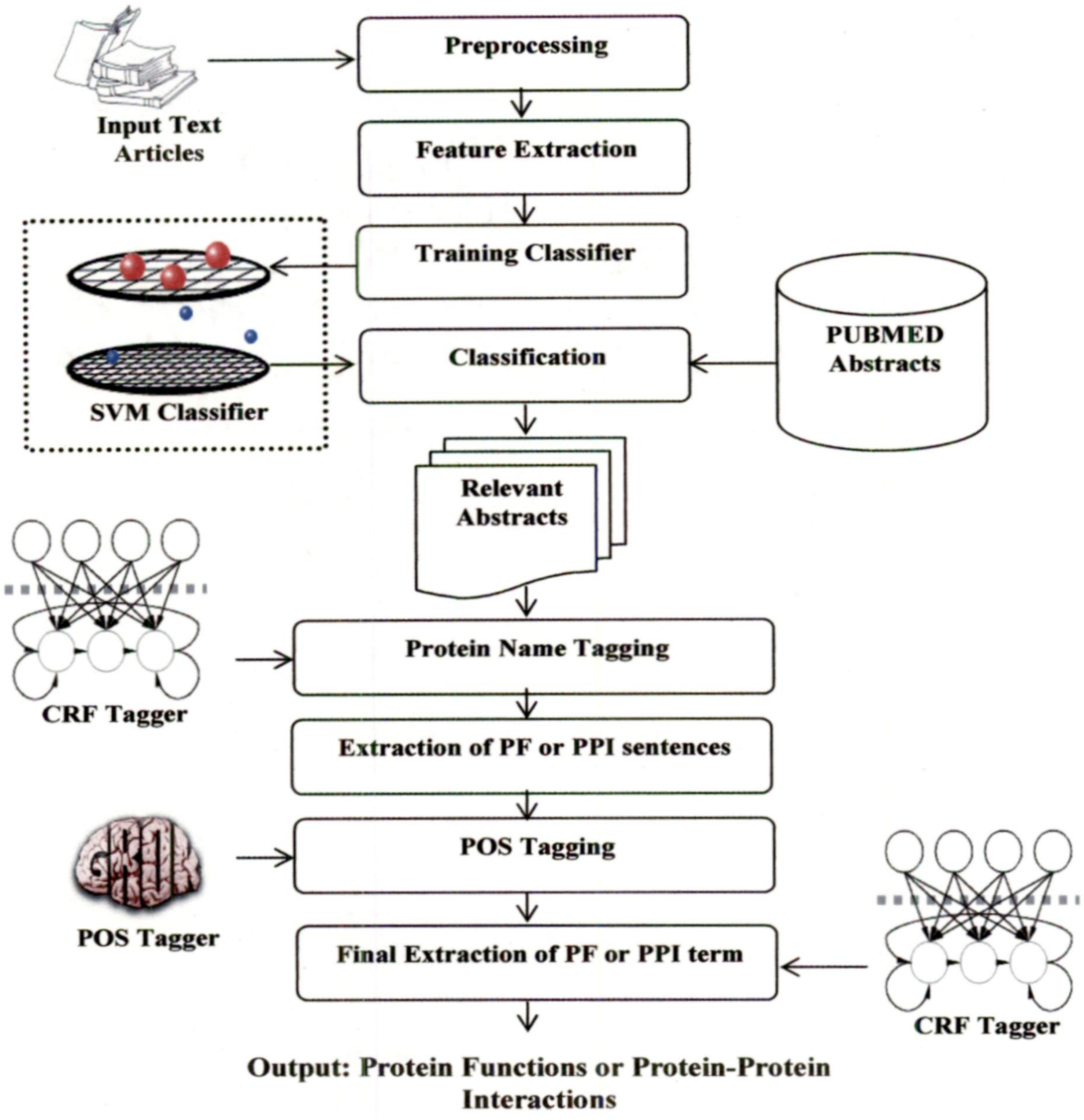


Fig. 4.1 The proposed framework for protein function and protein-protein interaction extraction

4.1 Protein Functions and Interactions Data Set

We adopted the corpus taken from PubMed for protein functions. PubMed Central (PMC) is the U.S. National Institutes of Health (NIH) free digital archive of biomedical and life sciences journal literature. The corpus consists of 13,014 abstracts related to protein functions. These abstracts are obtained from PubMed using the pmids given in the task 2 of BioCreAtIvE (Critical Assessment of Information

Extraction Systems in Biology; http://www.mitre.org/public/biocreative) Competition. A total of 79,231 sentences are detected in these abstracts.

We used the information from a domain-specific database to gather labeled data for the task of classifying the interactions between proteins in text. The manually-curated HIV-1 Human Protein Interaction Database (www.ncbi.nlm.nih.gov/RefSeq/HIVInteractions/index.html) provides a summary of documented interactions between HIV-1 proteins and host cell proteins, other HIV-1 proteins, or proteins from disease organisms associated with HIV or AIDS. We use this database also because it contains information about the *type* of interactions, as opposed to other protein interaction databases (BIND, MINT, and DIP) that list the protein pairs interacting, without specifying the type of interactions. In this database, the definitions of the interactions depend on the proteins involved and the articles describing the interactions; thus there are several definitions for each interaction type. For the interaction *bind* and the proteins *ANT* and *Vpr*, we find (among others) the definition *"Interaction of HIV- 1 Vpr with human adenine nucleotide translocator (ANT) is presumed based on a specific binding interaction between Vpr and rat ANT."*

4.2 Preprocessing of Data Sets

The PubMed abstracts are processed initially to make them useful for the next phases. As different authors use different writing styles in representations and naming conventions of biomedical terms, the preprocessing is a must phase. Text is made into tokens using white-space as delimiter and removing all punctuation marks and stop words. The parts-of-speech tags are assigned to each token using Grok, an open-source natural language processing library (Grok, http://grok.sourceforge.net/, 2006). The POS tagger, Grok uses a maximum entropy model trained on English data from the Wall Street Journal and the Brown corpus and achieves greater than 96% accuracy on unseen data.

4.3 Feature Extraction from input articles

The feature extraction phase concentrates on picking up the theme from the abstracts containing the protein functions and human protein-protein interactions. The features are extracted using probability distribution model. The standard probabilistic model for the distribution of a certain type of event over units of a fixed size is the Poisson distribution as discussed in earlier section.

For the purpose of feature extraction in protein function abstracts, we have used 9,716 general PubMed abstracts and 12,945 PubMed abstracts that are manually-curated as dealing with Protein functions. First we constructed a dictionary containing the frequencies of most common words in 9,716 general PubMed abstracts. Next the frequencies of words from training set of 12,945 function abstracts were compared against the calculated dictionary frequencies of general abstracts for unexpectedly higher or lower frequencies, indicating words that are useful for classifying the function abstracts from the ordinary abstracts.

Much similarly for the purpose of feature extraction in protein-protein interaction abstracts, we have used 9,716 general PubMed abstracts that contain the term 'HIV' and 923 PubMed abstracts that are manually-curated as dealing with HIV-1 Human Protein Interactions. Next the features are extracted as described above.

The probabilities $p(k,\lambda_i)$ of the word 'i' is calculated as in equation (4.3.1).

$$p(k,\lambda_i)=\frac{e^{-\lambda_i}\lambda_i^{\,k}}{k!} \tag{4.3.1}$$

where 'k' is the frequency of the word 'i' with dictionary frequency as 'λ_i'.

We might say that $p_i(k)=p(k,\lambda_i)$ is the probability of a document having exactly k occurrences of word 'i', where 'λ_i' is appropriately estimated for each word. Generally to avoid floating point errors, the log of the probability was calculated as shown in equation (4.3.2).

$$\ln p(k,\lambda_i)\approx -\lambda_i + k\ln(\lambda_i)-\ln(k!) \tag{4.3.2}$$

The tokens with part-of-speech as noun were removed in order to remove the bias from the training data. The words with most negative log probability value imply the rare occurrence of these words in general abstracts, thereby implying their relevance to function and interaction words. Hence such words are considered as features for our system.

4.4 Training the SVM classifier

The abstracts from PubMed may not all deal with protein functions and protein-protein interactions. The abstracts which deal with the protein functions and protein interactions are of main concern for the work. The abstracts are classified using a SVM trained on the features extracted in the above phase to obtain the required abstracts.

For this phase, 1,225 abstracts without PFs and 1964 abstracts with PFs and a well pruned feature set of 2000 words are used to train the SVM classifier for protein functions. And 978 abstracts without PPIs and 923 abstracts with PPIs are used to train the SVM classifier for protein-protein interactions. The SVMs are trained using these features

4.5 Protein Name Tagging Using CRF Tagger

Conditional random fields are probabilistic models that were designed for segmenting and labeling sequence data as discussed in section 3. As protein names are normalized in the preprocessing phase, the names are now a set of biomedical words appearing as consequent words. The protein names from the biomedical texts are tagged using the CRF tagger. Words in the sentence are assigned labels by states in the CRF framework. For this purpose we used well known Abner [27] protein name tagger, an open source tool which makes use of CRFs.

4.6 Term Extraction using a variant of Phrase Chunking

Abstracts describe the entire work carried out by the researchers. The sentences describing the relation between proteins are of main concern for us. These sentences are extracted by identifying functions and interaction terms between proteins from all the sentences.

Text chunking an intermediate step towards full parsing of natural language - recognizes phrase types in input text sentences. Phrase chunking, a variant of text chunking deals with a similar kind of task: it involves recognizing the chunks that consist of protein function and protein-protein interaction phrases. This task was performed using second order sequential CRF and it labels each word with a label indicating whether the word is inside the function and interaction chunk (I), or outside a chunk (O). Our chunking CRF has a second-order Markov dependency between chunk tags. This is easily encoded by making the CRF labels pairs of consecutive chunk tags. That is, the label at position i is $y_i = c_{i-1}c_i$, where c_i is the chunk tag of word i, one of I or O. In addition, successive labels are constrained: $y_{i-1} = c_{i-2}c_{i-1}, y_i = c_{i-1}c_i$, and $c_0 =$ O. These constraints on the model topology were enforced by giving appropriate features a weight of $-\infty$, forcing all the forbidden labeling to have zero probability.

The input chunks were initially tagged with POS tags and function phrase tags to generate the training data. The training data was generated using 7,454 PF sentences and 3,852 non-PF sentences for tagging protein function terms. Then the actual function terms from the data were extracted using the trained CRF. The sentence with a protein name and the function chunk extracted using the above methodology was considered as the PF sentence. For the training data of interaction terms, 940 PPI sentences and 852 non-PPI sentences are used. Then the actual interaction terms from the data were extracted using the trained CRF. The sentence with more than two protein names and the interaction chunk extracted using the above methodology was considered as the PPI sentence.

This chapter presents the implementation details of the framework discussed in the earlier chapter. The individual modules in the previously discussed framework can perform independently from each other but in the same order as shown in the framework. The modules are implemented according to implementation convenience using different language tools like C, C++ and Java. The implementation details are discussed in the following sections.

5.1 System Requirements

The programs are written in C, C++ and Java. So the system requires a standard C++ compiler and Java Development Kit with Java Virtual Machine on the system. Memory requirements depend upon the number of input text abstracts or full articles. The operating system requirements for the programs written in C and C++ are as shown below

• Linux/Unix:

Compiler: GNU C Compiler (gcc) and GNU C++ Compiler (g++)

Library: STL

• MS Windows 2000, XP:

Compiler: MS Visual C++ 7.0

Library: STL

The system should have a UNIX or Windows Operating System and a Java Compiler for the programs written in Java. The system with a Pentium IV processor, having minimum 256MByte RAM is needed for the entire framework.

5.2 Implementation of Preprocessing Module

The input abstracts are initially processed to remove the unnecessary terms. The stop words which occur with high frequencies in general obstruct the efficient extraction of feature. The stop words are removed from the input texts by using a predefined set of stop words. The preprocessing module is implemented as a Java program which matches the words in the input text with the stop words and removes them from the input text. The special characters are also removed using the same program to make the text useful for the remaining phases.

5.3 Implementation of Classification Module

The classification module is one of the major modules of the framework. The main purpose of this module is to extract the relevant text articles from the input articles which may not all contain the protein functions or protein-protein interactions. The classification module is implemented using the modified support vector machines in Java. The classifier requires the input in a standard format as described in the LIBSVM [29] software package.

A classification task usually involves with training and testing data which consist of some data instances. Each instance in the training set contains one "target value" (class labels) and several "attributes" (features). The goal of SVM is to produce a model which predicts target value of data instances in the testing set which are given only the attributes.

Given a training set of instance-label pairs $(x_i, y_i), i = 1,\dots,l$ where $x_i \in R^n$ and $y \in \{1,-1\}^l$, the support vector machines (SVM) require the solution of the following optimization problem:

$$\min_{w,b,\xi} \quad \frac{1}{2} w^T w + C\sum_{i=1}^{l} \xi_i \tag{5.3.1}$$

$$\text{subject to } \; y_i(w^T \phi(x_i) + b) \geq 1 - \xi_i,$$

$$\xi_i \geq 0. \tag{5.3.2}$$

Here training vectors x_i are mapped into a higher (maybe infinite) dimensional space by the function ϕ. Then SVM finds a linear separating hyperplane with the maximal margin in this higher dimensional space. C > 0 is the penalty parameter of the error term. Furthermore, $K(x_i, x_j) \equiv \phi(x_i)^T \phi(x_j)$ is called the kernel function.

The kernel used in our program is Radial Basis Function (RBF) kernel:

$$K(x_i, x_j) = \exp(-\gamma \|x_i - x_j\|^2), \gamma > 0 .$$

Here, γ is kernel parameter.

The following procedure has been used to implement the svm classification:

- Transform data to the format of SVM software

- Conduct simple scaling on the data
- Consider the RBF kernel $K(x,y) = e^{-\gamma\|x-y\|^2}$
- Use cross-validation to find the best parameter C and γ.
- Use the best parameter C and γ to train the whole training set.
- Test

We discuss this procedure in detail in the following sub sections.

5.3.1 Representation of Feature Vectors

SVM requires that each data instance is represented as a vector of real numbers. Hence, if there are categorical attributes, we first have to convert them into numeric data. We recommend using *m* numbers to represent an *m*-category attribute. Only one of the *m* numbers is one, and others are zero. For our problem, a sample four-category attribute such as {protein, binding, activation, response} can be represented as (0,0,0,1), (1,0,0,0), (0,0,1,0) and (0,1,0,0).

5.3.2 Scaling

Scaling the vectors before applying SVM is very important. The main advantage is to avoid attributes in greater numeric ranges dominate those in smaller numeric ranges. Another advantage is to avoid numerical difficulties during the calculation. Because kernel values usually depend on the inner products of feature vectors, e.g. the linear kernel and the polynomial kernel, large attribute values might cause numerical problems. Here we have scaled the attributes of vectors to range [0,1].

5.3.3 RBF Kernel

The RBF kernel nonlinearly maps samples into a higher dimensional space, so it, unlike the linear kernel, can handle the case when the relation between class labels and attributes is nonlinear. Furthermore, the linear kernel is a special case of RBF as shows that the linear kernel with a penalty parameter $\tilde{C}$ has the same performance as the RBF kernel with some parameters (C, γ).

The second reason is the number of hyperparameters which influences the complexity of model selection. The polynomial kernel has more hyperparameters than the RBF kernel. Finally, the RBF kernel has less numerical difficulties. One key point is $0 < K_{ij} \leq 1$ in contrast to polynomial kernels of which kernel values may go to infinity $(\gamma x_i^T x_j + r > 1)$ or zero $(\gamma x_i^T x_j + r < 1)$ while the degree is large.

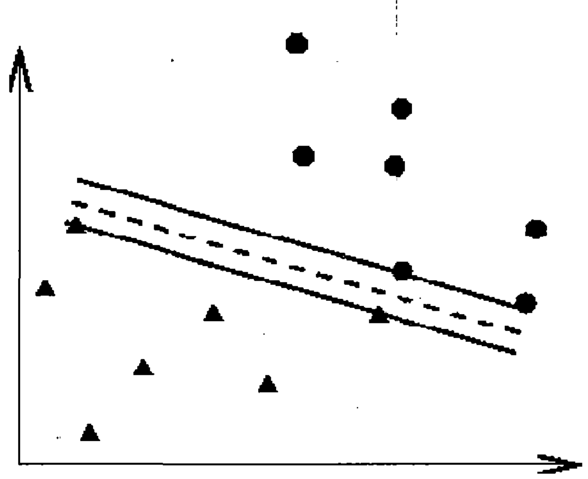
5.3.4 Cross-validation

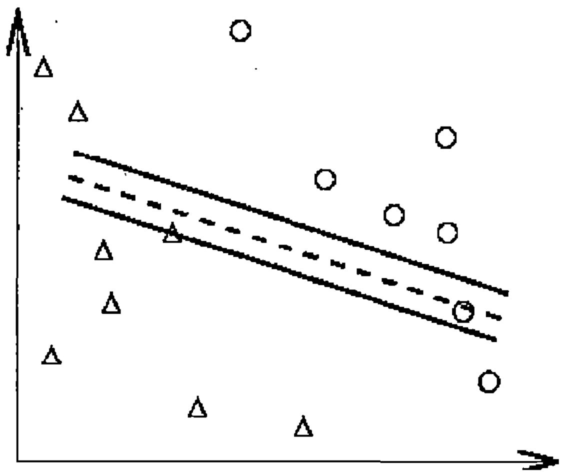
There are two parameters while using RBF kernels: C and γ. It is not known beforehand which C and γ are the best for one problem; consequently some kind of model selection (parameter search) must be done. The goal is to identify good (C, γ) so that the classifier can accurately predict unknown data (i.e., testing data). Note that it may not be useful to achieve high training accuracy (i.e., classifiers accurately predict training data whose class labels are indeed known). Therefore, a common way is to separate training data to two parts of which one is considered unknown in training the classifier. Then the prediction accuracy on this set can more precisely reflect the performance on classifying unknown data. An improved version of this procedure is cross-validation.

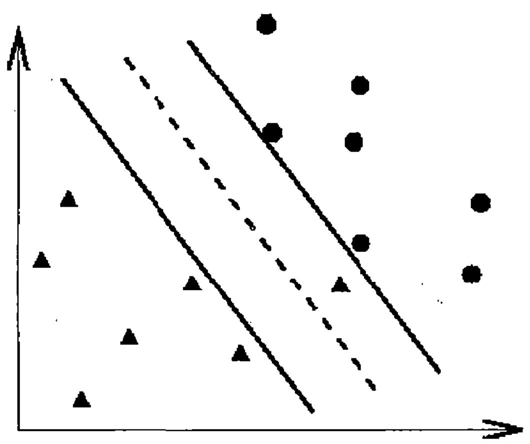
In ν-fold cross-validation, we first divide the training set into ν subsets of equal size. Sequentially one subset is tested using the classifier trained on the remaining ν-1 subsets. Thus, each instance of the whole training set is predicted once so the cross-validation accuracy is the percentage of data which are correctly classified.

The cross-validation procedure can prevent the overfitting problem. We use Figure 5.1 which is a binary classification problem (triangles and circles) to illustrate this issue. Filled circles and triangles are the training data while hollow circles and triangles are the testing data. The testing accuracy the classifier in Figures 5.1(a) and 5.1(b) is not good since it overfits the training data. If we think training and testing data in Figure 5.1(a) and 5.1(b) as the training and validation sets in cross-validation, the accuracy is not good. On the other hand, classifier in 5.1(c) and 5.1(d) without overfitting training data gives better cross-validation as well as testing accuracy.

The program for the svm based classification is done in Java. The entire module is implemented using various sub modules for solving the mathematical part, for preparing the matrix in the required form, for making the kernel related calculations and for finally cross validating sub module for the input data.


(a) Training data and an overfitting classifier


(b) Applying an overfitting classifier on testing data


(c) Training data and a better classifier

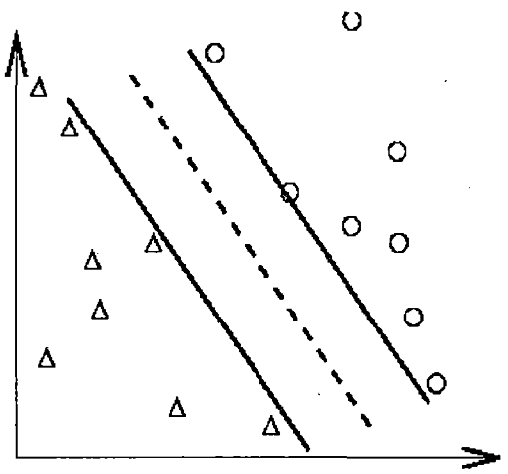

(d) Applying a better classifier on testing data

Figure 5.1: An overfitting classifier and a better classifier (Dark circle and triangle: training data; Hollow circle and triangle: testing data)

5.4 Implementation of CRF Tagging

The CRF tagging is done at two phases of the framework, the protein name tagging and tagging the final terms of protein functions or protein-protein interactions. The protein name tagging is done using a well known tagger Abner, as protein name tagging is of not main concern for us. The important phase in framework is tagging the exact terms of functions and interactions.

The program for tagging is done using a variant of phrase chunking using FlexCRFs in C++ language. FlexCRFs makes use of Viterbi algorithm for this purpose. The relevant POS tagged text articles are given as input to the tagging phase to extract the terms. The tagging is done manually first to get the training data for the CRF tagger. The input sentences are tagged with parts-of-speech initially using Grok, and the protein function or protein-protein interaction terms are tagged manually.

The important options for the FlexCRF are f_rare_threshold and cp_rare_threshold. The rare thresholds for features (f_rare_threshold) and context predicates (cp_rare_threshold) are 1 and 1. These thresholds mean that context predicates and features whose occurrence frequencies are smaller than or equal to 1 will be removed.

To use FlexCRFs for segmenting and labeling sequence data, Users must first prepare training (and testing) data. Training and testing data should have the format specified by the following rules:

<**Data**>:= a list of <**Data Sequences**>

<**A Data Sequence**>:= a list of <**Data Observations**>

<**A Data Observation**>:= a list of <**Context Predicates**> + <**A Label**>

<**A Context Predicate**>:= A string token

<**A label**>:= A string token

In other words, training or testing data sets consist of a list of data sequences; and two consecutive data sequences are separated by a blank line. Each data sequence consists of a series of data observations; and each data observation is placed on a line. Each data observation contains a list of context predicates and a label that are separated by blank characters. Context predicates and labels are represented as string tokens, i.e., strings without blank characters.

5.4.1 Implementation of Viterbi Algorithm

The Viterbi algorithm provides an efficient way of finding the most likely state sequence in the maximum *a posteriori* probability sense of a process assumed to be a finite-state discrete-time Markov process. Such processes can be subsumed under the general statistical framework of **compound decision theory** as discussed below.

Suppose we have a text of n characteres. Each character yields a feature vector z_i, $i=1,2,\ldots,n$. Let $p(Z \mid C)$ denote the probability density function of the vector sequence $Z = z_1, z_2, \ldots, z_n$ conditioned on the sequence of identities $C = c_1, c_2, \ldots, c_n$, where z_k is the feature vector for the *k-th* character, and where c_k takes on M values (number of letters in the alphabet) for $k=1,2,\ldots,n$. Also, let P(C) be the *a priori* probability of the sequence of values C. In other words P(C) is the *a priori* probability distribution of all sequences of n characters. The probability of correctly classifying the text is maximized by choosing that sequence of characters that has a maximum posterior probability or the so called: maximum *a posteri* (MAP) probability, given by P(C|Z). From Bayes' rule we obtain

$$P(C \mid Z) = \frac{p(Z \mid C)P(C)}{p(Z)}$$

Since $p(Z)$ is independent of the sequence C (it is just a scale factor) we need only maximize the discriminant function

$$g_c(Z) = p(Z \mid C)P(C)$$

The amount of storage required for these probabilities is huge in practice, for that reason, assumptions are made in order to reduce the problem down to manageable size. These assumptions are:

- The size of the sequence of observations is not very large. Let n be the size of a word. Then P(C) is the frequency of occurrence of words.
- Conditional independence among the features vectors. The shape of a character, which generates a given feature vector, is independent of the shapes of neighboring characters and is, therefore, dependent only on the character in question.

Under these assumptions, and taking logarithms, discriminant function reduces to

$$g_c(Z) = \sum_{i=1}^{n} \log p(z_i \mid c_i) + \log p(c_1.c_2, \ldots, c_n)$$

For the case of the Viterbi algorithm, if we assume that the process is first-order Markov, then above equation reduces to:

$$g_c(Z) = \sum_{i=1}^{n} \log p(z_i \mid c_i) + \log[P(c_1 \mid c_0) + P(c_2 \mid c_1) + \ldots + P(c_n \mid c_{n+1})]$$

To illustrate how the Viterbi algorithm obtains this shortest path, we need to represent the Markov process in an easy way. A **state diagram**, like one shown in Fig. 5.2, is often used. In this state diagram, the *nodes* (circles) represent *states*, *arrows* represent *transitions*, and over the course of time the process traces some path from state to state through the state diagram.

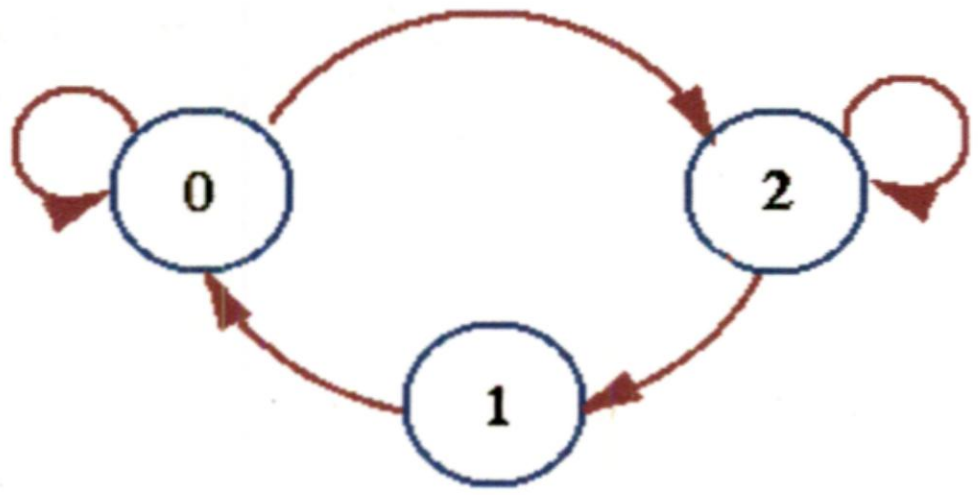


Fig. 5.2 State diagram of a three-state process

A more redundant description of the same process is shown in Fig. 5.3, this description is called *trellis*. In a trellis, each node corresponds to a distinct state at a given time, and each arrow represents a transition to some new state at the next instant of time. The trellis begins and ends at the known states c_0 and c_n. Its most important property is that to every possible state sequence **C** there corresponds a unique path through the trellis, and vice versa.

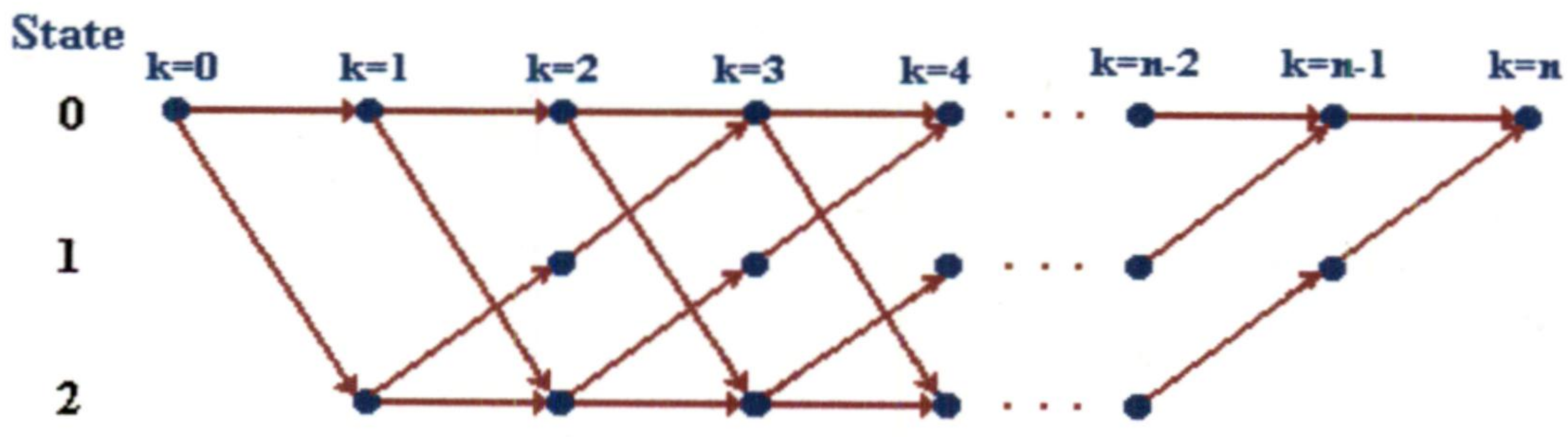


Fig. 5.3 Trellis for the three-state process of Fig. 5.2.

Now, suppose we assign to every path a length proportional to $-\log[p(Z\mid C)+P(C)]$. Since log() is a monotonic function and there is a one-to-one correspondence between paths and sequences, we only need to find the path whose $-\log[p(Z\mid C)+P(C)]$ is **minimum**, this will give us the state sequence for which $p(Z\mid C)P(C)$ is **maximum**, in other words, the state sequence with the maximum *a posteriori* (MAP) probability, which take us back to the original problem we want to solve. The **total length** of the path corresponding to some state sequence C is

$$-\log[p(Z \mid C)P(C)] = \sum_{k=1}^{n} l(t_k)$$

where $l(t_k)$ is the associated length to each transition t_k from c_k to c_{k+1}. The shortest such path segment is called the ***survivor*** corresponding to the node c_k, and is denoted $S(c_k)$. For any time k>0, there are M survivors in all, one for each c_k. The observation is this: the shortest complete path S must begin with one of these survivors. Thus for any time k we need to remember only the M survivors $S(c_k)$ and their correspondent lengths. To get to time *k+1*, we need only extend all time-*k* survivors by one time unit, compute the lengths of the extended path segments, and for each node c_{k+1} as the corresponding time-*(k+1)* survivor. Recursion proceeds indefinitely without the number of survivors ever exceeding *M*. This algorithm is a simple version of forward dynamic programming.

The Viterbi algorithm seen as finding the shortest route through a graph is:

```
Input:
    Z= $z_1, z_2, \ldots, z_n$                the input observed sequence
Initialization:
    $k=1$                                time index
    $S(c_1)=c_1$
    $L(c_1)=0$                           this is a variable that accumulates the lengths,
                                         the initial length is 0
Recursion:
    For all transitions $t_k = (c_k, c_{k+1})$
        compute: $L(c_k, c_{k+1}) = L(c_k) + l[t_k = (c_k, c_{k+1})]$ among all $c_k$
    Find $L(c_{k+1}) = \min L(c_k, c_{k+1})$
    For each $c_{k+1}$
        store $L(c_{k+1})$ and the corresponding survivor $S(c_{k+1})$
    $k=k+1$
    Repeat until $k=n$
```

With finite state sequences C the algorithm terminates at time *n* with the shortest complete path stored as the survivor $S(c_k)$.

The above discussed Viterbi algorithm is implemented in C++. There are separate sub modules for the string tokenizer, building the model using the training data and the crf module to finally extract the terms of protein functions and protein-protein interactions. The training data for protein function tagging data is prepared as shown below for the sentence "*The DT C-terminal domain of HIV-1 Vpu (amino acids) interacts with the cytoplasmic domain of CD4 (amino acids) and causes the rapid degradation of CD4 in the endoplasmic reticulum*":

```
The      DT     NREL
C-terminal      JJ      NREL
domain          NN      NREL
of       IN     NREL
HIV-1 NNP       NREL
Vpu      NNP    NREL
(amino          VBD     NREL
acids)          NNS     NREL
interacts       NNS     NREL
with     IN     NREL
the      DT     NREL
cytoplasmic JJ          NREL
domain          NN      NREL
of       IN     NREL
CD4      CD     NREL
(amino          NN      NREL
acids)          NNS     NREL
and      CC     NREL
causes          VBZ     NREL
the      DT     NREL
rapid JJ        NREL
degradation NN          REL
of       IN     NREL
CD4      NNP    NREL
in       IN     NREL
the      DT     NREL
endoplasmic JJ          NREL
reticulum       NN      NREL
. . O
```

The first column gives the tokens of the string, the second column are the parts-of-speech tags and the third column is the class to which the token belongs to (NREL-represents non-function term and REL-represents the protein function term). The test data has to be presented without the third column to the trained model. And similar algorithm is used for the extraction of the protein interaction extraction.

6.1 Accuracy of Classification

The protein function extraction system consists of a module for classifying whether input abstracts are discussing the protein functions or not as shown in Fig. 4.1. The accuracy of the classifier was compared with two types of classifiers namely Naïve Bayesian based classification and SVM based classification, and the results of SVM based classifier outperformed the earlier. For protein function abstracts, the SVM classifier obtained a precision value of 0.97 at recall value of 0.94. Till the recall value of 0.9 the precision was found to be around 95%. The precision – recall curve for this classification is as shown in Fig. 6.1. The graph points are obtained by varying a threshold on the minimum acceptable extraction confidence, based on the probability estimates from LibSVM [29]. Due to efficient extraction of features, the classification performed much better and classifier is not biased towards the training data.

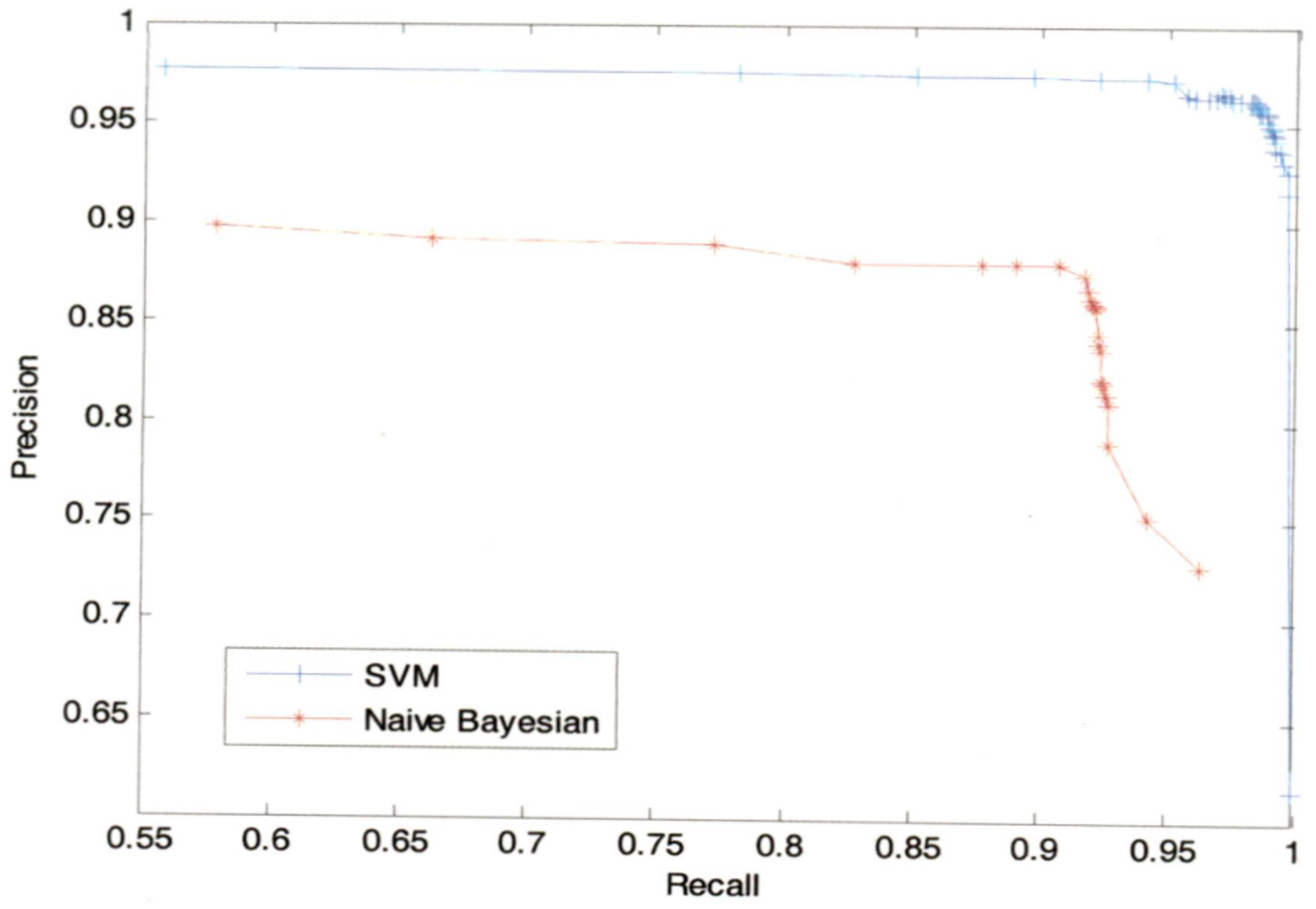


Fig. 6.1 The performance of Naïve Bayesian and SVM classifiers on protein function abstracts

Accuracy is estimated as 95%. Accuracy is a percentile expression of the number of times that the SVM is correct in its classification (either function abstract or not). Estimated precision is 95%. Precision is the percentage of times that the SVM is correct in its classification of an abstract as describing a function. Recall is estimated as 94% and is the percentage of known function articles that the SVM would classify as being about a function.

For classification of protein interactions abstracts, the accuracy of the classification was compared with two types of classifiers namely Naïve Bayesian based classification and SVM based classification, and the results of SVM based classifier outperformed the earlier in this case also. The SVM classifier obtained a precision value of 0.95 at recall value of 0.85. Till the recall value of 0.6 the precision was found to be 100%. The precision-recall curve for this classification is as shown in Fig. 6.2.

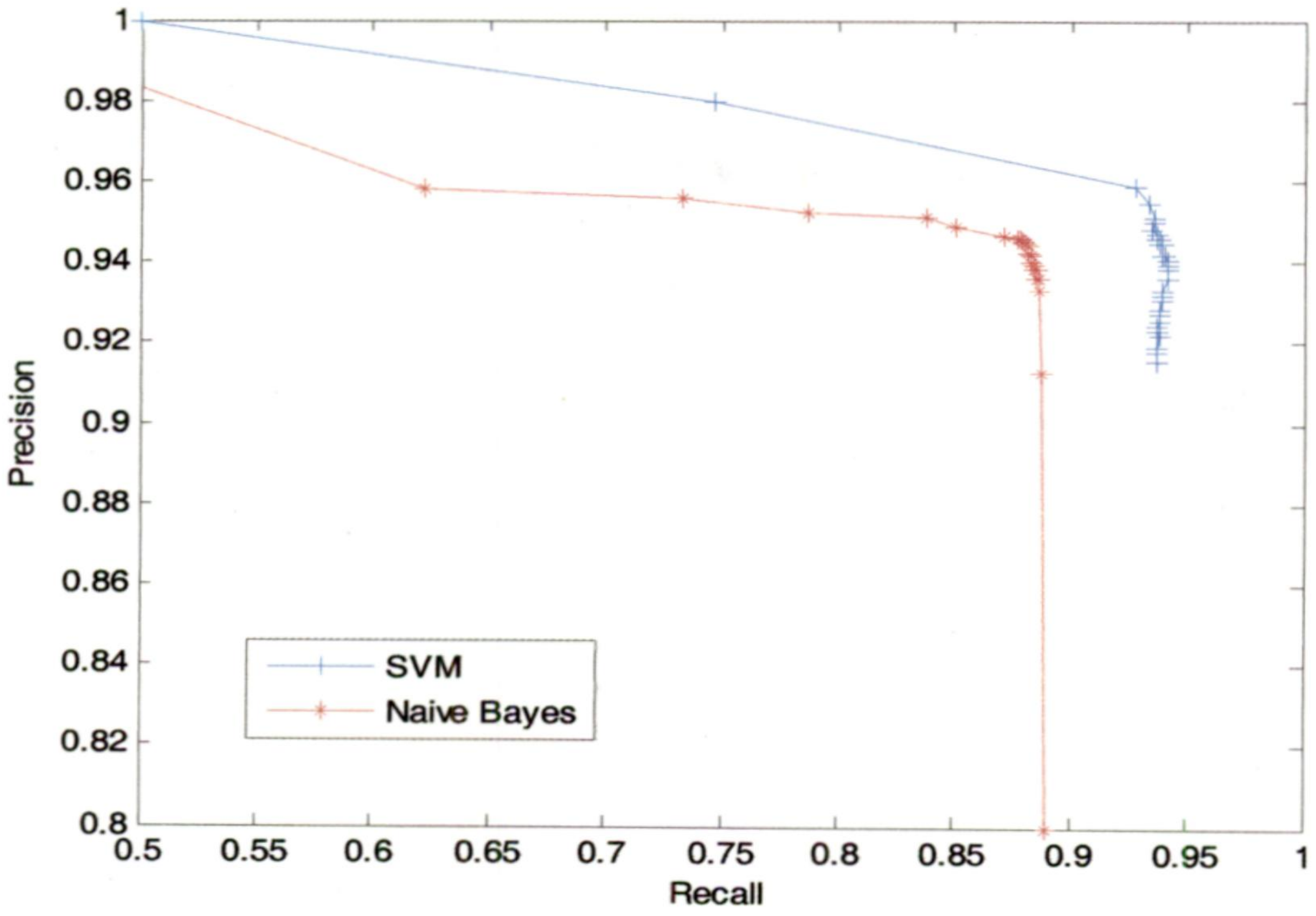


Fig. 6.2 The performance of Naïve Bayesian and SVM classifiers on protein-protein interaction abstracts

Accuracy is estimated at 94%. (either interaction abstract or not). Estimated precision is 94%. Precision is the percentage of times that the SVM is correct in its classification of an abstract as describing an interaction. Recall is estimated as 88% and is the percentage of known interaction articles that the SVM would classify as being about an interaction.

6.2 Performance of Function and Interaction Phrase Chunking

The main task of the system was the extraction of the PF sentence with function phrase from the abstract. This task was done using function phrase chunking technique. The performance of which is as shown in the Fig. 6.3.

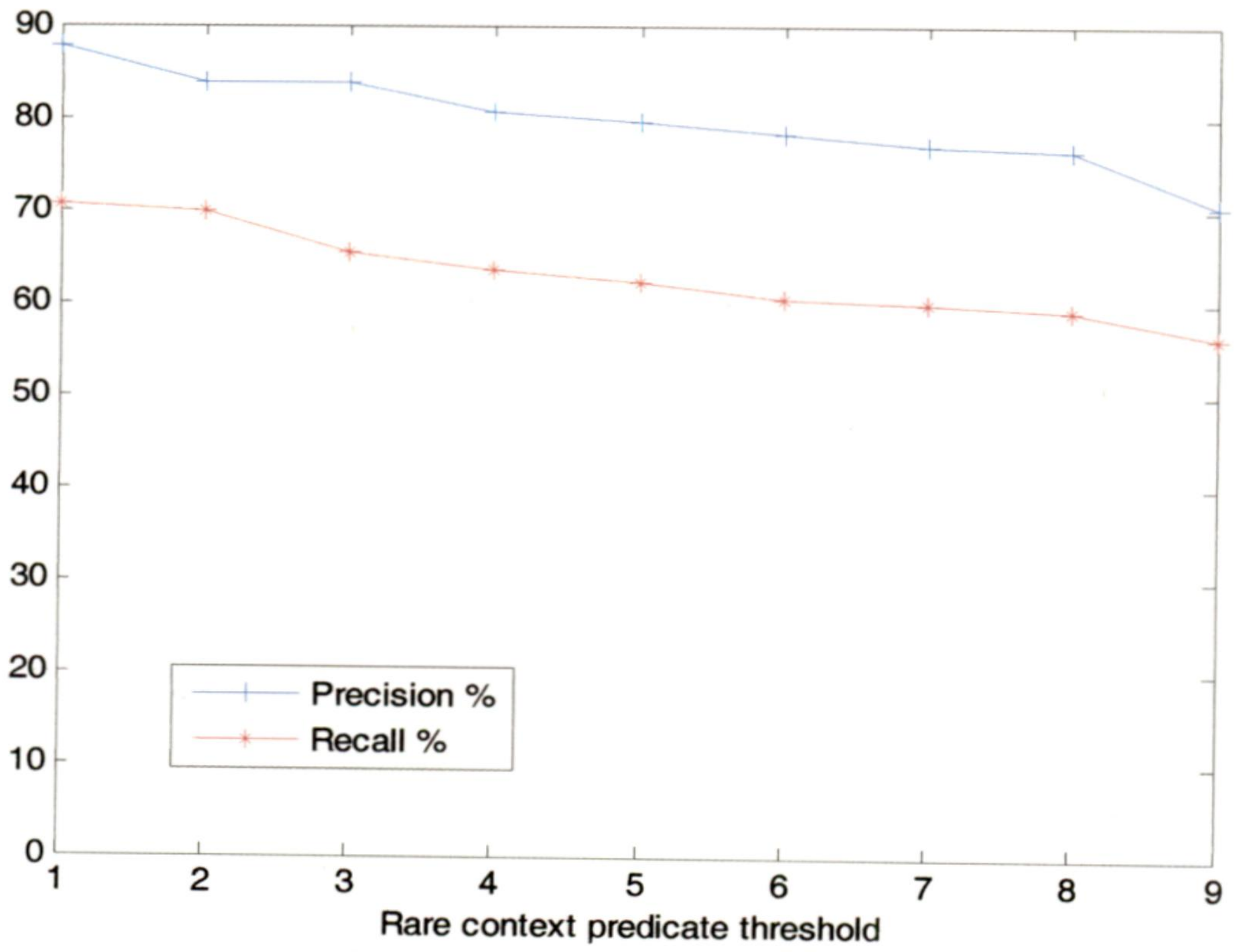


Fig. 6.3 The performance of the function phrase chunking technique

The precision value of this phase of the system was found to be 79.49% and recall was found to be 62.28% at moderate rare context predicate threshold value of 5. Hence, the system has got the precision value much superior to many other existing function extraction systems.

Extraction of the PPI sentence with interaction phrase from the abstract is done using interaction phrase chunking technique. The performance of which is as shown in the Fig. 6.4.

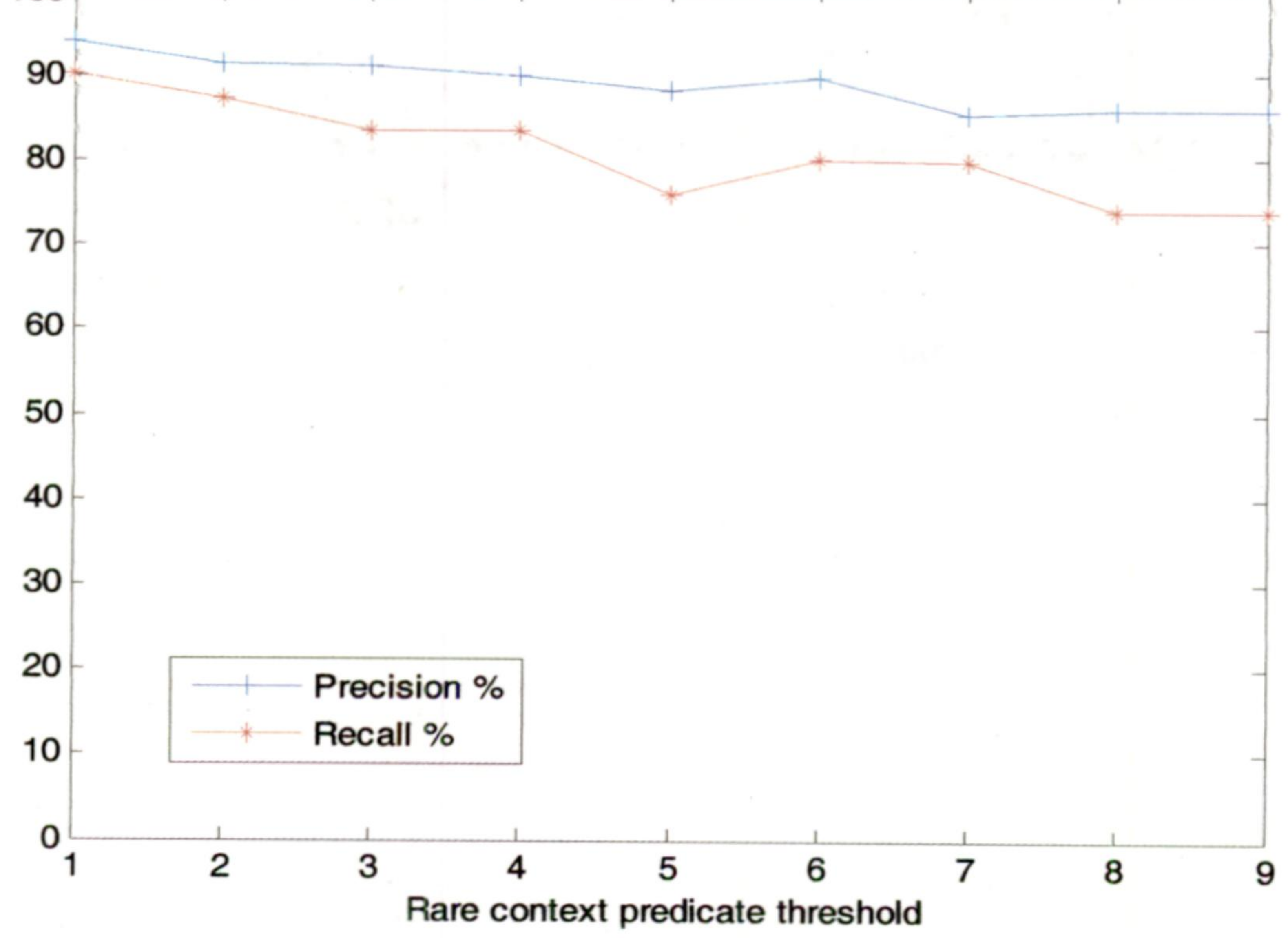


Fig. 6.4 The performance of the interaction phrase chunking technique

The precision value of this phase of the system was found to be 85.85% and recall was found to be 85.74% at minimum rare context predicate threshold value of 9. Hence, the system has got the precision value much superior to many other existing kernel based classification systems.

6.3 Comparative Study

We compare the following three systems on the task of retrieving protein functions with our system implemented using phrase chunking (CRF) and are shown in table 1.

- **[Sentence Pattern Mining (SPM)]:** we report the performance of the sentence pattern mining based system of [19].
- **[Term-based SVM (TbSVM)]:** We report the performance of mining protein functions using term-based SVMs of [18].

- **[Full Sentence Parsing (FSP)]:** Extraction using a Full-Sentence Parser, we report the performance of this from [28].

Method	**Precision %**	**Recall %**
CRF(Proposed System)	79.49	62.28
SPM	67.2	40.0
TbSVM	50.0	-
FSP	50.0	30.4

Table 1: The comparison of existing function extraction systems with our implementation

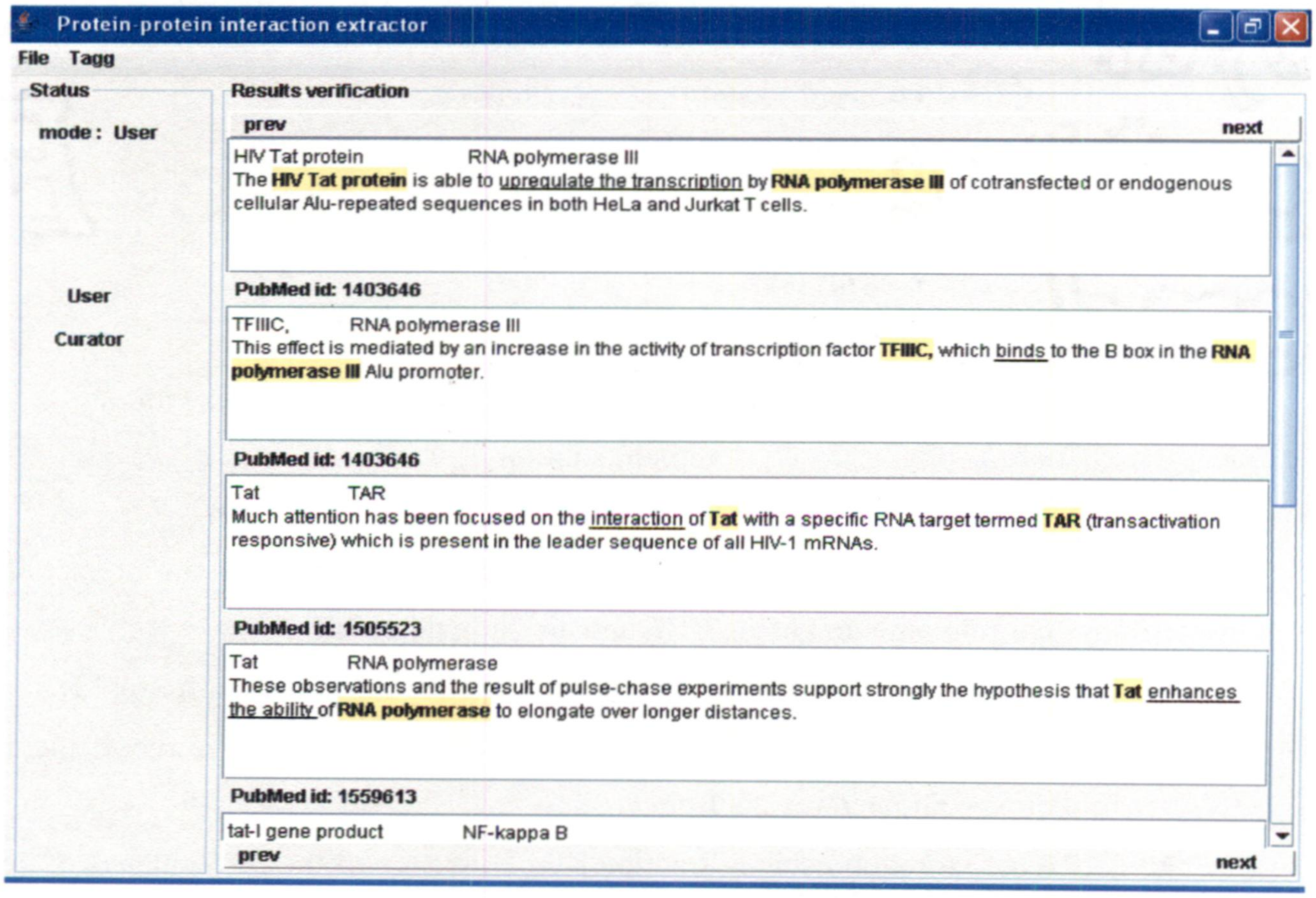
We compare the following three systems on the task of retrieving protein interactions with our system implemented using CRF and are shown in table 2.

- **[Manual]:** we report the performance of the rule-based system from [12].
- **[ELCS]:** Extraction using Longest Common Subsequence, we report the 10 fold cross-validated results from [12].
- **[ERK]:** Extraction using a Relation Kernel, we report the performance of this from [12].

Method	**Precision %**	**Recall %**
CRF (Proposed System)	85.85	85.74
Manual	68.5	32.4
ELCS	55.3	35.2
ERK	73.9	35.2

Table 2: The comparison of existing interaction extraction systems with our implementation

The screen shot of the implemented system is as shown in Fig. 6.5, in which the user is allowed to search for the interaction between the protein names provided by him as input. In the output, the protein names are highlighted and the interaction between them is shown as being underlined.



HIV Tat protein RNA polymerase III
The **HIV Tat protein** is able to upregulate the transcription by **RNA polymerase III** of cotransfected or endogenous cellular Alu-repeated sequences in both HeLa and Jurkat T cells.

PubMed id: 1403646

TFIIIC, RNA polymerase III
This effect is mediated by an increase in the activity of transcription factor **TFIIIC,** which binds to the B box in the **RNA polymerase III** Alu promoter.

PubMed id: 1403646

Tat TAR
Much attention has been focused on the interaction of **Tat** with a specific RNA target termed **TAR** (transactivation responsive) which is present in the leader sequence of all HIV-1 mRNAs.

PubMed id: 1505523

Tat RNA polymerase
These observations and the result of pulse-chase experiments support strongly the hypothesis that **Tat** enhances the ability of **RNA polymerase** to elongate over longer distances.

PubMed id: 1559613

tat-I gene product NF-kappa B

Fig. 6.5 The screenshot showing the interaction term extraction technique

Conclusion and Scope for Future Work CHAPTER 7

7.1 Conclusion

In the proposed technique to extracting protein functions and interactions from text, we have used SVMs and variants of Phrase Chunking to extract the function and interaction terms.

This system is tested on 127137 PubMed abstracts containing keywords 'protein' and 'function' and on 171106 PubMed abstracts containing keywords 'protein' and 'interaction'. The test resulted in the extraction of 18477 protein functions and 26923 protein-protein interactions.

The proposed variants of phrase chunking mechanism can efficiently reduce the effort of manual recognition of the protein functions and interactions in biomedical text articles. The evaluation of the results shows the capabilities and limitations of supervised machine-learning approaches in text mining. The following conclusions can be made from the results obtained using the proposed system and above mentioned data:

- The system can yield good performance only if sufficient training data is obtained, and significant amount of supporting data is used for prediction.
- The classification module resulted in very good precision and recall values of around 95% and 96% for protein functions and 94% and 93% for interactions.
- The results show that performance improves as the number of relevant documents increases. This implies that our function and interaction term assignments are accurate and with good recall.
- The performance of protein function and protein-protein interaction term extraction is far better than the existing techniques as shown in comparative study of chapter 6.

Hence, this work can aid the understanding of protein functions and interactions for biomedical researchers and assist database curators in annotating protein functions and interactions efficiently, thus promoting the progress of genomics research.

7.2 Scope for Future Work

There is obviously significant room for improving the methods that we used for the extraction of protein function and protein-protein interaction terms. The possible improvements in the future are listed as below:

- Protein name tagging is itself an area of research. Other efficient techniques can also be explored in this area for tagging the protein names.
- The extraction of function and interaction terms is done using Conditional Random Fields in our system, but many more techniques can also be explored for this purpose.
- An extra module can be implemented for extracting the specific functions or interactions related to a particular protein from the entire database of functions and interactions.
- The work can be extended to parallel processing environment where extraction of function and interaction term can be made using parallel CRFs.

In the future, there will be enormous need for the rapid annotations of protein functions and interactions for biomedical researchers to access the biomedical problems of human beings and to prescribe the drugs for their cure.

References

[1] L. Hirschman, J.C. Park, J. Tsujii, D. Wong, and C.H. Wu, "Accomplishments and Challenges in Literature Data Mining for Biology", *Bioinformatics*, vol. 18, no. 12, pp. 1553-1561, 2002.

[2] J. H. Chiang, H. C. Yu and H. J. Hsu, "GIS: A Biomedical Text-Mining System for Gene Information Discovery", *Bioinformatics*, vol. 20, no. 1, pp. 120-121, 2004.

[3] Emilia Stoica and Marti Hearst, "Predicting Gene Functions from Text Using a Cross-Species Approach", *Proceedings of Pacific Symposium on Biocomputing*, pp. 88-99, September, 2006.

[4] L. Tanabe and W.J. Wilbur, "Tagging Gene and Protein Names in Biomedical Text", *Bioinformatics*, vol. 18, no. 8, pp. 1124-1132, 2002.

[5] H. Yu and E. Agichtein, "Extracting Synonymous Gene and Protein Terms from Biological Literature", *Bioinformatics*, vol. 19, Suppl. 1, pp. i340-i349, 2003.

[6] G.D. Zhou, J. Zhang, J. Su, D. Shen, and C. L. Tan, "Recognizing Names in Biomedical Texts: A Machine Learning Approach", *Bioinformatics*, vol. 20, no. 7, pp. 1178-1190, 2004.

[7] B.J. Stapley, L.A. Kelley, and M.J.E. Sternberg, "Predicting the Sub-Cellular Location of Proteins from Text Using Support Vector Machines", *Proceedings of Pacific Symposium on Biocomputing (PSB)* 2002, pp. 374-385, 2002.

[8] J. Pustejovsky, J. Castano, J. Zhang, M. Kotecki, and B. Cochran, "Robust Relational Parsing over Biomedical Literature: Extracting Inhibit Relations," *Proceedings of Pacific Symposium on Biocomputing (PSB)* 2002, pp. 362-373, 2002.

[9] D. M. Yao, J. B. Wang, Y. M. Lu, N. Noble, H. D. Sun, X. Y. Zhu, N. Lin, D.G. Payan, M. Li, and K. B. Qu, "Pathway Finder: Paving the Way towards Automatic Pathway Extraction," *Proceedings of Second Asia-Pacific Bioinformatics Conf. (APBC2004)*, pp. 53-62, 2004.

[10] Jung-Hsien Chiang and Hsu-Chun Yu, "Extracting Functional Annotations of Proteins Based on Hybrid Text Mining Approaches", In *Proceedings of BioCreative Workshop*, 2004.

[11] Asako Koike, Yoshiki Niwa and Toshihisa Takagi, "Automatic extraction of gene/protein biological functions from biomedical text", *Bioinformatics*, vol. 21, no. 7, pp. 1227–1236, 2005.

[12] A.K. Ramani, R.C. Bunescu, R.J. Mooney and E.M. Marcotte, "Consolidating the set of known human protein-protein interactions in preparation for large-scale mapping of the human interactome", *Genome Biology 2005*, Vol. 6, Issue 5, Article r40, pp. R40.1-R40.12, 2005.

[13] X. Chen and M. Liu "Domain-Based Predictive Models for Protein-Protein Interaction Prediction", *Hindawi Publishing Corporation EURASIP Journal on Applied Signal Processing*, Article ID 32767, pp 1–8, 2006.

[14] Stapley, B. and Benoit, G., "Biobibliometrics: information retrieval and visualization from co-occurrences of gene names in medline abstracts", In *Proceedings of the Pacific Symposium on Biocomputing*, pp. 529-540, 2000.

[15] Blaschke, C., Oliveros, J.C. & Valencia, A., "Mining functional information associated with expression arrays", *Functional and Integrative Genomics*, vol. 1, pp. 256-268, 2001.

[16] Rindesch, T., Tanabe, L., Weinstein, J. & Hunter, L., "EDGAR: extraction of drugs, genes and relations from the biomedical literature", In *Proceedings of the Pacific Symposium on Biocomputing (PSB'00)*, pp. 517-528, 2000.

[17] Blaschke, C., Andrade, M.A., Ouzounis, C. and Valencia, A., "Automatic extraction of biological information from scientific text: protein-protein interactions", In *International Conference on Intelligent Systems for Molecular Biology (ISMB'99)*, pp. 60-67, 1999.

[18] Simon B Rice, Goran Nenadic and Benjamin J Stapley, "Mining protein function from text using term-based support vector machines", *BMC Bioinformatics*, Vol. 6, no. (Suppl 1): S22, pp. 1-11, May 2005.

[19] Jung-Hsien, Chiang and Hsu-Chun Yu, "Literature Extraction of Protein Functions Using Sentence Pattern Mining", *IEEE Transactions on Knowledge and Data Engineering*, Vol. 17, no. 8, pp. 1088-1098, August 2005.

[20] R. Bunescu, R. Ge, R. Kate, E.M. Marcotte, R.J. Mooney, A.K. Ramani, Y.W. Wong, "Comparative experiments on learning information extractors for proteins and their interactions", *Artificial Intelligence in Medicine, Special Issue on Summarization and Information Extraction from Medical Documents*, 2005.

[21] Huma Lodhi, Craig Saunders, John Shawe-Taylor, Nello Cristianini, and Chris Watkins, "Text classification using string kernels", *Journal of Machine Learning Research*, vol. 2, pp. 419–444, 2002.

[22] A. Ratnaparkhi, "A maximum entropy model for part-of-speech tagging", In *Proceedings of the Conference on Empirical Methods in Natural Language Processing,* pp.133-142, Philadelphia, USA, 1996.

[23] K. Nigam, J. Lafferty, and A. McCallum, "Using maximum entropy for text classification", *In IJCAI99 Workshop on Machine Learning for Information Filtering,* pp. 61- 67, 1999.

[24] Lafferty, John, A. McCallum, and F. Pereira, "Conditional random fields: Probabilistic models for segmenting and labeling sequence data", *Proceedings of Eighteenth International Conference on Machine Learning (ICML-2001),* pp: 282-289, June 28-July 01, 2001.

[25] L. R. Rabiner, "A tutorial on hidden Markov models and selected applications in speech recognition", *Proceedings of the IEEE,* 77(2): pp. 257–286, February 1989.

[26] R. Bunescu and R. Mooney, "Subsequence kernels for relation extraction", *Advances in Neural Information Processing Systems,* 18, MIT Press, Cambridge, MA. 2006.

[27] Burr Settles, "ABNER: an open source tool for automatically tagging genes, proteins and other entity names in text", *Bioinformatics,* Vol. 21, no 14, pp. 3191-3192, April 2005.

[28] Daraselia, N., Yuryev, A., Egorov, S., Novichkova, S., Nikitin, A. and Mazo, I "Extracting Protein Function Information from MEDLINE using a full-sentence parser", In *Proceedings of the Second European Workshop on Data Mining and Text Mining in Bioinformatics,* Pisa, Italy, pp. 15-21, September 2004.

[29] Chih-Chung Chang and Chih-Jen Lin, "LIBSVM: a library for support vector machines," 2001. (http://www.csie.ntu.edu.tw/~cjlin/libsvm).

APPENDIX

Source Code Listing

/* svm_problem.java */

```
public class svm_problem implements java.io.Serializable
{
        public int l;  // number of instanses  or lines
        public double[] y;  //  the given output class
        public svm_node[][] x;  //   the feature vectores
}
```

/ * svm_node.java */

```
/* creates a new instance of svm_node, this is sparse representation*/
public class svm_node implements java.io.Serializable
{
        public int index;
        public double value;
}
```

/* decision_function.java */

```
public class decision_function
{
   double[] alpha;
   double rho;
}
```

/* QMatrix.java */

```
abstract class QMatrix {
        abstract float[] get_Q(int column, int len);
        abstract float[] get_QD();
        abstract void swap_index(int i, int j);
}
```

/* svm_model.java */

```
public class svm_model implements java.io.Serializable
{
        svm_parameter param;         // parameter
        int nr_class;               // number of classes, = 2 in regression/one class svm
        int l;                      // total #SVs
        svm_node[][] SV;            // SVs (SV[l])
        double[][] sv_coef;         // coefficients for SVs in decision functions (sv_coef[n-
1][l])
        double[] rho;               // constants in decision functions (rho[n*(n-1)/2])
        double[] probA;             // pariwise probability information
        double[] probB;

        // for classification only
        int[] label;            // label of each class (label[n])
        int[] nSV;              // number of SVs for each class (nSV[n])
                                // nSV[0] + nSV[1] + ... + nSV[n-1] = l
}
```

/* svm.java */

```
import java.io.*;
import java.util.*;
public class svm {

        private static void solve_c_svc(svm_problem prob, svm_parameter param,
                                        double[] alpha, Solver.SolutionInfo si,
                                        double Cp, double Cn)  // here class labels are
only +1 ,-1
        {
                int l = prob.l;
                double[] minus_ones = new double[l];
                byte[] y = new byte[l];

                int i;

                for(i=0;i<l;i++)
                {
                        alpha[i] = 0;
                        minus_ones[i] = -1;
                        if(prob.y[i] > 0) y[i] = +1; else y[i]=-1;
                }

                Solver s = new Solver();
                s.Solve(l, new SVC_Q(prob,param,y), minus_ones, y,
                        alpha, Cp, Cn, param.eps, si, param.shrinking);

                double sum_alpha=0;
                for(i=0;i<l;i++)
                        sum_alpha += alpha[i];

                if (Cp==Cn)
                        System.out.print("nu = "+sum_alpha/(Cp*prob.l)+"\n");
                for(i=0;i<l;i++)
                        System.out.println(alpha[i]+"   "+ y[i]);
                for(i=0;i<l;i++)
                        alpha[i] *= y[i];
        }


    static decision_function svm_train_one(
                svm_problem prob, svm_parameter param,
                double Cp, double Cn)
        {
```

```
		double[] alpha = new double[prob.l];
		Solver.SolutionInfo si = new Solver.SolutionInfo();

		solve_c_svc(prob,param,alpha,si,Cp,Cn);

		System.out.print("obj = "+si.obj+", rho = "+si.rho+"\n");

		// output SVs

		int nSV = 0;
		int nBSV = 0;
		for(int i=0;i<prob.l;i++)
		{
			if(Math.abs(alpha[i]) > 0)
			{
				++nSV;
				if(prob.y[i] > 0)
				{
					if(Math.abs(alpha[i]) >= si.upper_bound_p)
					++nBSV;
				}
				else
				{
					if(Math.abs(alpha[i]) >= si.upper_bound_n)
						++nBSV;
				}
			}
		}

		System.out.print("nSV = "+nSV+", nBSV = "+nBSV+"\n");

		decision_function f = new decision_function();
		f.alpha = alpha;
		f.rho = si.rho;
		return f;
	}

	public static svm_model svm_train(svm_problem prob, svm_parameter param)
	{
		svm_model model = new svm_model();
		model.param = param;


	    model.nr_class = 2;
	    model.label = null;
	    model.nSV = null;
	    model.probA = null; model.probB = null;
	    model.sv_coef = new double[1][];
		decision_function f = svm_train_one(prob,param,1,1);
		model.rho = new double[1];
		model.rho[0] = f.rho;

		int nSV = 0;
		int i;
		for(i=0;i<prob.l;i++)
			if(Math.abs(f.alpha[i]) > 0) ++nSV;
		model.l = nSV;
		model.SV = new svm_node[nSV][];
		model.sv_coef[0] = new double[nSV];
		int j = 0;
		for(i=0;i<prob.l;i++)
			if(Math.abs(f.alpha[i]) > 0)
			{
				model.SV[j] = prob.x[i];
				model.sv_coef[0][j] = f.alpha[i];
				++j;
			}

		return model;
	}



	public static int svm_predict(svm_model model, svm_node[] x)
	{

			double[] res = new double[1];
			svm_predict_values(model, x, res);

			return (res[0]>0)?1:-1;
	}

	public static void svm_predict_values(svm_model model, svm_node[] x,
double[] dec_values)
	{

			double[] sv_coef = model.sv_coef[0];
			double sum = 0;
			for(int i=0;i<model.l;i++)
			sum += sv_coef[i] * Kernel.k_function(x,model.SV[i],model.param);
			sum -= model.rho[0];
			dec_values[0] = sum;
	}

	public static void svm_save_model(String model_file_name, svm_model model)
throws IOException
	{
			FileOutputStream fos = new FileOutputStream(model_file_name);
```

```
                ObjectOutputStream oos = new ObjectOutputStream(fos);
                oos.writeObject(model);
                oos.close();
        }

      public static svm_model svm_load_model(String model_file_name) throws
IOException
        {

                FileInputStream fis = new FileInputStream(model_file_name);
                ObjectInputStream ois = new ObjectInputStream(fis);
                svm_model model = null;

                try {
                  model = (svm_model) ois.readObject();
                } catch (IOException ex) {
                        ex.printStackTrace();
                } catch (ClassNotFoundException ex) {
                        ex.printStackTrace();
                }
                ois.close();

                return model;
         }

   static String svm_check_parameter(svm_problem prob, svm_parameter param) {
      return null;
        }

      public static int svm_get_svm_type(svm_model model)
        {
                return model.param.svm_type;
        }

        public static int svm_get_nr_class(svm_model model)
        {
                return model.nr_class;
        }
}
```

/* svm_parameter.java */

```
public class svm_parameter implements Cloneable,java.io.Serializable
{
        /* svm_type */
        public static final int C_SVC = 0;
        public static final int NU_SVC = 1;
        public static final int ONE_CLASS = 2;
        public static final int EPSILON_SVR = 3;
        public static final int NU_SVR = 4;
```

```
	/* kernel_type */
	public static final int LINEAR = 0;
	public static final int POLY = 1;
	public static final int RBF = 2;
	public static final int SIGMOID = 3;
	public static final int PRECOMPUTED = 4;

	public int svm_type;
	public int kernel_type;
	public int degree;	// for poly
	public double gamma;// for poly/rbf/sigmoid
	public double coef0;	// for poly/sigmoid

	// these are for training only
	public double cache_size; // in MB
	public double eps;	// stopping criteria
	public double C;	// for C_SVC, EPSILON_SVR and NU_SVR
	public int nr_weight;		// for C_SVC
	public int[] weight_label;	// for C_SVC
	public double[] weight;			// for C_SVC
	public double nu;	// for NU_SVC, ONE_CLASS, and NU_SVR
	public double p;	// for EPSILON_SVR
	public int shrinking;	// use the shrinking heuristics
	public int probability; // do probability estimates

	public Object clone()
	{
		try
		{
			return super.clone();
		} catch (CloneNotSupportedException e)
		{
			return null;
		}
	}

}
```

/* SVC_Q.java */

```
public class SVC_Q extends Kernel
{
	private final byte[] y;
	private final Cache cache;
	private final float[] QD;

	SVC_Q(svm_problem prob, svm_parameter param, byte[] y_)
	{
		super(prob.l, prob.x, param);
```

```
			y = (byte[])y_.clone();
			cache = new Cache(prob.l,(int)(param.cache_size*(1<<20)));
			QD = new float[prob.l];
			for(int i=0;i<prob.l;i++)
				QD[i]= (float)kernel_function(i,i);
		}

		float[] get_Q(int i, int len)
		{
			float[][] data = new float[1][];
			int start;
			if((start = cache.get_data(i,data,len)) < len)
			{
				for(int j=start;j<len;j++)
					data[0][j] = (float)(y[i]*y[j]*kernel_function(i,j));
			}
			return data[0];
		}

		float[] get_QD()
		{
			return QD;
		}

		void swap_index(int i, int j)
		{
			cache.swap_index(i,j);
			super.swap_index(i,j);
			do {byte _=y[i]; y[i]=y[j]; y[j]=_;} while(false);
			do {float _=QD[i]; QD[i]=QD[j]; QD[j]=_;} while(false);
		}


	}
```

/* Solver.java */

```
// Solves:
//
//	min 0.5(\alpha^T Q \alpha) + b^T \alpha
//
//		y^T \alpha = \delta
//		y_i = +1 or -1
//		0 <= alpha_i <= Cp for y_i = 1
//		0 <= alpha_i <= Cn for y_i = -1
//
// Given:
//	Q, b, y, Cp, Cn, and an initial feasible point \alpha
//	l is the size of vectors and matrices
//	eps is the stopping criterion
```

```
//
// solution will be put in \alpha, objective value will be put in obj
//
public class Solver {
	int active_size;
	byte[] y;
	double[] G;		// gradient of objective function
	static final byte LOWER_BOUND = 0;
	static final byte UPPER_BOUND = 1;
	static final byte FREE = 2;
	byte[] alpha_status;	// LOWER_BOUND, UPPER_BOUND, FREE
	double[] alpha;
	QMatrix Q;
	float[] QD;
	double eps;
	double Cp,Cn;
	double[] b;
	int[] active_set;
	double[] G_bar;			// gradient, if we treat free variables as 0
	int l;
	boolean unshrinked;	// XXX

	static final double INF = java.lang.Double.POSITIVE_INFINITY;

	double get_C(int i)
	{
		return (y[i] > 0)? Cp : Cn;
	}
	void update_alpha_status(int i)
	{
		if(alpha[i] >= get_C(i))
			alpha_status[i] = UPPER_BOUND;
		else if(alpha[i] <= 0)
			alpha_status[i] = LOWER_BOUND;
		else alpha_status[i] = FREE;
	}
	boolean is_upper_bound(int i) { return alpha_status[i] == UPPER_BOUND; }
	boolean is_lower_bound(int i){ return alpha_status[i] == LOWER_BOUND; }
	boolean is_free(int i) {  return alpha_status[i] == FREE; }

	// java: information about solution except alpha,
	// because we cannot return multiple values otherwise...
	static class SolutionInfo {
		double obj;
		double rho;
		double upper_bound_p;
		double upper_bound_n;
		double r;	// for Solver_NU
	}
	void swap_index(int i, int j)
	{
		Q.swap_index(i,j);
		do {byte _=y[i]; y[i]=y[j]; y[j]=_;} while(false);
		do {double _=G[i]; G[i]=G[j]; G[j]=_;} while(false);
		do {byte _=alpha_status[i]; alpha_status[i]=alpha_status[j];
		    alpha_status[j]=_;} while(false);
		do {double _=alpha[i]; alpha[i]=alpha[j]; alpha[j]=_;} while(false);
		do {double _=b[i]; b[i]=b[j]; b[j]=_;} while(false);
		do {int _=active_set[i]; active_set[i]=active_set[j]; active_set[j]=_;}
		while(false);
		do {double _=G_bar[i]; G_bar[i]=G_bar[j]; G_bar[j]=_;} while(false);
	}

	void reconstruct_gradient()
	{
		// reconstruct inactive elements of G from G_bar and free variables

		if(active_size == l) return;

		int i;
		for(i=active_size;i<l;i++)
			G[i] = G_bar[i] + b[i];

		for(i=0;i<active_size;i++)
			if(is_free(i))
			{
				float[] Q_i = Q.get_Q(i,l);
				double alpha_i = alpha[i];
				for(int j=active_size;j<l;j++)
					G[j] += alpha_i * Q_i[j];
			}
	}

	void Solve(int l, QMatrix Q, double[] b_, byte[] y_,
		   double[] alpha_, double Cp, double Cn, double eps, SolutionInfo si,
int shrinking)
	{
		this.l = l;
		this.Q = Q;
		QD = Q.get_QD();
		b = (double[])b_.clone();
		y = (byte[])y_.clone();
		alpha = (double[])alpha_.clone();
		this.Cp = Cp;
		this.Cn = Cn;
		this.eps = eps;
		this.unshrinked = false;

		// initialize alpha_status
```

```
{
        alpha_status = new byte[l];
        for(int i=0;i<l;i++)
                update_alpha_status(i);
}

// initialize active set (for shrinking)
{
        active_set = new int[l];
        for(int i=0;i<l;i++)
                active_set[i] = i;
        active_size = l;
}

// initialize gradient
{
        G = new double[l];
        G_bar = new double[l];
        int i;
        for(i=0;i<l;i++)
        {
                G[i] = b[i];
                G_bar[i] = 0;
        }
        for(i=0;i<l;i++)
                if(!is_lower_bound(i))
                {
                        float[] Q_i = Q.get_Q(i,l);
                        double alpha_i = alpha[i];
                        int j;
                        for(j=0;j<l;j++)
                                G[j] += alpha_i*Q_i[j];
                        if(is_upper_bound(i))
                                for(j=0;j<l;j++)
                                        G_bar[j] += get_C(i) * Q_i[j];
                }
}

// optimization step

int iter = 0;
int counter = Math.min(l,1000)+1;
int[] working_set = new int[2];

while(true)
{
        // show progress and do shrinking

        if(--counter == 0)
        {
```

```
		counter = Math.min(l,1000);
		if(shrinking!=0) do_shrinking();
		System.err.print(".");
	}

	if(select_working_set(working_set)!=0)
	{
		// reconstruct the whole gradient
		reconstruct_gradient();
		// reset active set size and check
		active_size = l;
		System.err.print("*");
		if(select_working_set(working_set)!=0)
			break;
		else
			counter = 1;
		// do shrinking next iteration
	}

	int i = working_set[0];
	int j = working_set[1];

	++iter;

	// update alpha[i] and alpha[j], handle bounds carefully

	float[] Q_i = Q.get_Q(i,active_size);
	float[] Q_j = Q.get_Q(j,active_size);

	double C_i = get_C(i);
	double C_j = get_C(j);

	double old_alpha_i = alpha[i];
	double old_alpha_j = alpha[j];

	if(y[i]!=y[j])
	{
		double quad_coef = Q_i[i]+Q_j[j]+2*Q_i[j];
		if (quad_coef <= 0)
			quad_coef = 1e-12;
		double delta = (-G[i]-G[j])/quad_coef;
		double diff = alpha[i] - alpha[j];
		alpha[i] += delta;
		alpha[j] += delta;

		if(diff > 0)
		{
			if(alpha[j] < 0)
			{
				alpha[j] = 0;
```

```
                        alpha[i] = diff;
                    }
                }
                else
                {
                    if(alpha[i] < 0)
                    {
                        alpha[i] = 0;
                        alpha[j] = -diff;
                    }
                }
                if(diff > C_i - C_j)
                {
                    if(alpha[i] > C_i)
                    {
                        alpha[i] = C_i;
                        alpha[j] = C_i - diff;
                    }
                }
                else
                {
                    if(alpha[j] > C_j)
                    {
                        alpha[j] = C_j;
                        alpha[i] = C_j + diff;
                    }
                }
            }
            else
            {
                double quad_coef = Q_i[i]+Q_j[j]-2*Q_i[j];
                if (quad_coef <= 0)
                    quad_coef = 1e-12;
                double delta = (G[i]-G[j])/quad_coef;
                double sum = alpha[i] + alpha[j];
                alpha[i] -= delta;
                alpha[j] += delta;

                if(sum > C_i)
                {
                    if(alpha[i] > C_i)
                    {
                        alpha[i] = C_i;
                        alpha[j] = sum - C_i;
                    }
                }
                else
                {
                    if(alpha[j] < 0)
                    {
                        alpha[j] = 0;
                        alpha[i] = sum;
                }
        }
        if(sum > C_j)
        {
                if(alpha[j] > C_j)
                {
                        alpha[j] = C_j;
                        alpha[i] = sum - C_j;
                }
        }
        else
        {
                if(alpha[i] < 0)
                {
                        alpha[i] = 0;
                        alpha[j] = sum;
                }
        }
}

// update G

double delta_alpha_i = alpha[i] - old_alpha_i;
double delta_alpha_j = alpha[j] - old_alpha_j;

for(int k=0;k<active_size;k++)
{
        G[k] += Q_i[k]*delta_alpha_i + Q_j[k]*delta_alpha_j;
}

// update alpha_status and G_bar

{
        boolean ui = is_upper_bound(i);
        boolean uj = is_upper_bound(j);
        update_alpha_status(i);
        update_alpha_status(j);
        int k;
        if(ui != is_upper_bound(i))
        {
                Q_i = Q.get_Q(i,l);
                if(ui)
                        for(k=0;k<l;k++)
                                G_bar[k] -= C_i * Q_i[k];
                else
                        for(k=0;k<l;k++)
                                G_bar[k] += C_i * Q_i[k];
        }
```

```
                    if(uj != is_upper_bound(j))
                    {
                            Q_j = Q.get_Q(j,l);
                            if(uj)
                                    for(k=0;k<l;k++)
                                            G_bar[k] -= C_j * Q_j[k];
                            else
                                    for(k=0;k<l;k++)
                                            G_bar[k] += C_j * Q_j[k];
                    }
            }

        }

        // calculate rho

        si.rho = calculate_rho();

        // calculate objective value
        {
                double v = 0;
                int i;
                for(i=0;i<l;i++)
                        v += alpha[i] * (G[i] + b[i]);

                si.obj = v/2;
        }

        // put back the solution
        {
                for(int i=0;i<l;i++)
                        alpha_[active_set[i]] = alpha[i];
        }

        si.upper_bound_p = Cp;
        si.upper_bound_n = Cn;

        System.out.print("\noptimization finished, #iter = "+iter+"\n");
}

// return 1 if already optimal, return 0 otherwise
int select_working_set(int[] working_set)
{
        // return i,j such that
        // i: maximizes -y_i * grad(f)_i, i in I_up(\alpha)
        // j: mimimizes the decrease of obj value
        //    (if quadratic coefficeint <= 0, replace it with tau)
        //    -y_j*grad(f)_j < -y_i*grad(f)_i, j in I_low(\alpha)
```

```
            double Gmax = -INF;
            double Gmax2 = -INF;
            int Gmax_idx = -1;
            int Gmin_idx = -1;
            double obj_diff_min = INF;

            for(int t=0;t<active_size;t++)
                    if(y[t]==+1)
                    {
                            if(!is_upper_bound(t))
                                    if(-G[t] >= Gmax)
                                    {
                                            Gmax = -G[t];
                                            Gmax_idx = t;
                                    }
                    }
                    else
                    {
                            if(!is_lower_bound(t))
                                    if(G[t] >= Gmax)
                                    {
                                            Gmax = G[t];
                                            Gmax_idx = t;
                                    }
                    }

            int i = Gmax_idx;
            float[] Q_i = null;
            if(i != -1) // null Q_i not accessed: Gmax=-INF if i=-1
                    Q_i = Q.get_Q(i,active_size);

            for(int j=0;j<active_size;j++)
            {
                    if(y[j]==+1)
                    {
                            if (!is_lower_bound(j))
                            {
                                    double grad_diff=Gmax+G[j];
                                    if (G[j] >= Gmax2)
                                            Gmax2 = G[j];
                                    if (grad_diff > 0)
                                    {
                                            double obj_diff;
                                            double quad_coef=Q_i[i]+QD[j]-
2*y[i]*Q_i[j];

                                            if (quad_coef > 0)
                                                    obj_diff = -
(grad_diff*grad_diff)/quad_coef;
                                            else
```

```
                                                obj_diff = -
(grad_diff*grad_diff)/1e-12;

                                        if (obj_diff <= obj_diff_min)
                                        {
                                                Gmin_idx=j;
                                                obj_diff_min = obj_diff;
                                        }
                                }
                        }
                }
                else
                {
                        if (!is_upper_bound(j))
                        {
                                double grad_diff= Gmax-G[j];
                                if (-G[j] >= Gmax2)
                                        Gmax2 = -G[j];
                                if (grad_diff > 0)
                                {
                                        double obj_diff;
                                        double
quad_coef=Q_i[i]+QD[j]+2*y[i]*Q_i[j];
                                        if (quad_coef > 0)
                                                obj_diff = -
(grad_diff*grad_diff)/quad_coef;
                                        else
                                                obj_diff = -
(grad_diff*grad_diff)/1e-12;

                                        if (obj_diff <= obj_diff_min)
                                        {
                                                Gmin_idx=j;
                                                obj_diff_min = obj_diff;
                                        }
                                }
                        }
                }
        }

        if(Gmax+Gmax2 < eps)
                return 1;

        working_set[0] = Gmax_idx;
        working_set[1] = Gmin_idx;
        return 0;
}

// return 1 if already optimal, return 0 otherwise
int max_violating_pair(int[] working_set)
```

```
	{
		// return i,j which maximize -grad(f)^T d , under constraint
		// if alpha_i == C, d != +1
		// if alpha_i == 0, d != -1

		double Gmax1 = -INF;			// max { -y_i * grad(f)_i | i in
I_up(\alpha) }
		int Gmax1_idx = -1;

		int Gmax2_idx = -1;
		double Gmax2 = -INF;			// max { y_i * grad(f)_i | i in
I_low(\alpha) }

		for(int i=0;i<active_size;i++)
		{
			if(y[i]==+1)	// y = +1
			{
				if(!is_upper_bound(i))// d = +1
				{
					if(-G[i] >= Gmax1)
					{
						Gmax1 = -G[i];
						Gmax1_idx = i;
					}
				}
				if(!is_lower_bound(i))// d = -1
				{
					if(G[i] >= Gmax2)
					{
						Gmax2 = G[i];
						Gmax2_idx = i;
					}
				}
			}
			else		// y = -1
			{
				if(!is_upper_bound(i))// d = +1
				{
					if(-G[i] >= Gmax2)
					{
						Gmax2 = -G[i];
						Gmax2_idx = i;
					}
				}
				if(!is_lower_bound(i))// d = -1
				{
					if(G[i] >= Gmax1)
					{
						Gmax1 = G[i];
						Gmax1_idx = i;
```

```
                        }
                    }
                }
            }

            if(Gmax1+Gmax2 < eps)
                return 1;

            working_set[0] = Gmax1_idx;
            working_set[1] = Gmax2_idx;
            return 0;
        }

        void do_shrinking()
        {
            int i,j,k;
            int[] working_set = new int[2];
            if(max_violating_pair(working_set)!=0) return;
            i = working_set[0];
            j = working_set[1];
            double Gm1 = -y[j]*G[j];
            double Gm2 = y[i]*G[i];

            // shrink

            for(k=0;k<active_size;k++)
            {
                if(is_lower_bound(k))
                {
                    if(y[k]==+1)
                    {
                        if(-G[k] >= Gm1) continue;
                    }
                    else    if(-G[k] >= Gm2) continue;
                }
                else if(is_upper_bound(k))
                {
                    if(y[k]==+1)
                    {
                        if(G[k] >= Gm2) continue;
                    }
                    else    if(G[k] >= Gm1) continue;
                }
                else continue;

                --active_size;
                swap_index(k,active_size);
                --k;    // look at the newcomer
            }
        // unshrink, check all variables again before final iterations

        if(unshrinked || -(Gm1 + Gm2) > eps*10) return;

        unshrinked = true;
        reconstruct_gradient();

        for(k=l-1;k>=active_size;k--)
        {
                if(is_lower_bound(k))
                {
                        if(y[k]==+1)
                        {
                                if(-G[k] < Gm1) continue;
                        }
                        else    if(-G[k] < Gm2) continue;
                }
                else if(is_upper_bound(k))
                {
                        if(y[k]==+1)
                        {
                                if(G[k] < Gm2) continue;
                        }
                        else    if(G[k] < Gm1) continue;
                }
                else continue;

                swap_index(k,active_size);
                active_size++;
                ++k;    // look at the newcomer
        }
}

double calculate_rho()
{
        double r;
        int nr_free = 0;
        double ub = INF, lb = -INF, sum_free = 0;
        for(int i=0;i<active_size;i++)
        {
                double yG = y[i]*G[i];

                if(is_lower_bound(i))
                {
                        if(y[i] > 0)
                                ub = Math.min(ub,yG);
                        else
                                lb = Math.max(lb,yG);
                }
                else if(is_upper_bound(i))
```

```
                        {
                                if(y[i] < 0)
                                        ub = Math.min(ub,yG);
                                else
                                        lb = Math.max(lb,yG);
                        }
                        else
                        {
                                ++nr_free;
                                sum_free += yG;
                        }
                }

                if(nr_free>0)
                        r = sum_free/nr_free;
                else
                        r = (ub+lb)/2;

                return r;
        }

}
```

/* Cache.java */

```
// Kernel Cache
//
// l is the number of total data items
// size is the cache size limit in bytes
//

class Cache {
        private final int l;
        private int size;
        private final class head_t
        {
                head_t prev, next;      // a cicular list
                float[] data;
                int len;        // data[0,len) is cached in this entry
        }
        private final head_t[] head;
        private head_t lru_head;

        Cache(int l_, int size_)
        {
                l = l_;
                size = size_;
                head = new head_t[l];
                for(int i=0;i<l;i++) head[i] = new head_t();
                size /= 4;
                size -= l * (16/4);        // sizeof(head_t) == 16
                size = Math.max(size, 2*l);  // cache must be large enough for two
columns
                lru_head = new head_t();
                lru_head.next = lru_head.prev = lru_head;
        }

        private void lru_delete(head_t h)
        {
                // delete from current location
                h.prev.next = h.next;
                h.next.prev = h.prev;
        }

        private void lru_insert(head_t h)
        {
                // insert to last position
                h.next = lru_head;
                h.prev = lru_head.prev;
                h.prev.next = h;
                h.next.prev = h;
        }

        // request data (0,len)
        // return some position p where (p,len) need to be filled
        // (p >= len if nothing needs to be filled)
        // java: simulate pointer using single-element array
        int get_data(int index, float[][] data, int len)
        {
                head_t h = head[index];
                if(h.len > 0) lru_delete(h);
                int more = len - h.len;

                if(more > 0)
                {
                        // free old space
                        while(size < more)
                        {
                                head_t old = lru_head.next;
                                lru_delete(old);
                                size += old.len;
                                old.data = null;
                                old.len = 0;
                        }

                        // allocate new space
                        float[] new_data = new float[len];
                        if(h.data != null) System.arraycopy(h.data,0,new_data,0,h.len);
                        h.data = new_data;
                        size -= more;
                do {int _=h.len; h.len=len; len=_;} while(false);
            }

            lru_insert(h);
            data[0] = h.data;
            return len;
        }

        void swap_index(int i, int j)
        {
            if(i==j) return;

            if(head[i].len > 0) lru_delete(head[i]);
            if(head[j].len > 0) lru_delete(head[j]);
            do {float[] _=head[i].data; head[i].data=head[j].data; head[j].data=_;}
             while(false);
            do {int _=head[i].len; head[i].len=head[j].len; head[j].len=_;}
             while(false);
            if(head[i].len > 0) lru_insert(head[i]);
            if(head[j].len > 0) lru_insert(head[j]);

            if(i>j) do {int _=i; i=j; j=_;} while(false);
            for(head_t h = lru_head.next; h!=lru_head; h=h.next)
            {
                if(h.len > i)
                {
                    if(h.len > j)
                        do {float _=h.data[i]; h.data[i]=h.data[j];
                        h.data[j]=_;} while(false);
                    else
                    {
                        // give up
                        lru_delete(h);
                        size += h.len;
                        h.data = null;
                        h.len = 0;
                    }
                }
            }
        }
    }
```

/* Kernel.java */

```
abstract class Kernel extends QMatrix {
        private svm_node[][] x;
        private final double[] x_square;

        // svm_parameter
        private final int kernel_type;
```

```
	private final int degree;
	private final double gamma;
	private final double coef0;

	abstract float[] get_Q(int column, int len);
	abstract float[] get_QD();

	void swap_index(int i, int j)
	{
		do {svm_node[] _=x[i]; x[i]=x[j]; x[j]=_;} while(false);
		if(x_square != null) do {double _=x_square[i];
		x_square[i]=x_square[j]; x_square[j]=_;} while(false);
	}

	private static double powi(double base, int times)
	{
		double tmp = base, ret = 1.0;

		for(int t=times; t>0; t/=2)
		{
			if(t%2==1) ret*=tmp;
				tmp = tmp * tmp;
		}
		return ret;
	}

	private static double tanh(double x)
	{
		double e = Math.exp(x);
		return 1.0-2.0/(e*e+1);
	}

	double kernel_function(int i, int j)
	{
		switch(kernel_type)
		{
			case svm_parameter.LINEAR:
				return dot(x[i],x[j]);
			case svm_parameter.POLY:
				return powi(gamma*dot(x[i],x[j])+coef0,degree);
			case svm_parameter.RBF:
				return Math.exp(-gamma*(x_square[i]+x_square[j]-
2*dot(x[i],x[j])));
			case svm_parameter.SIGMOID:
				return tanh(gamma*dot(x[i],x[j])+coef0);
			case svm_parameter.PRECOMPUTED:
				return x[i][(int)(x[j][0].value)].value;
			default:
				return 0;	// java
		}
```

```
}

Kernel(int l, svm_node[][] x_, svm_parameter param)
{
        this.kernel_type = param.kernel_type;
        this.degree = param.degree;
        this.gamma = param.gamma;
        this.coef0 = param.coef0;

        x = (svm_node[][])x_.clone();

        if(kernel_type == svm_parameter.RBF)
        {
                x_square = new double[l];
                for(int i=0;i<l;i++)
                        x_square[i] = dot(x[i],x[i]);
        }
        else x_square = null;
}

static double dot(svm_node[] x, svm_node[] y)
{
        double sum = 0;
        int xlen = x.length;
        int ylen = y.length;
        int i = 0;
        int j = 0;
        while(i < xlen && j < ylen)
        {
                if(x[i].index == y[j].index)
                        sum += x[i++].value * y[j++].value;
                else
                {
                        if(x[i].index > y[j].index)
                                ++j;
                        else
                                ++i;
                }
        }
        return sum;
}

static double k_function(svm_node[] x, svm_node[] y,
                                svm_parameter param)
{
        switch(param.kernel_type)
        {
                case svm_parameter.LINEAR:
                        return dot(x,y);
                case svm_parameter.POLY:
```

```
                                    return
powi(param.gamma*dot(x,y)+param.coef0,param.degree);
                        case svm_parameter.RBF:
                        {
                                double sum = 0;
                                int xlen = x.length;
                                int ylen = y.length;
                                int i = 0;
                                int j = 0;
                                while(i < xlen && j < ylen)
                                {
                                        if(x[i].index == y[j].index)
                                        {
                                                double d = x[i++].value - y[j++].value;
                                                sum += d*d;
                                        }
                                        else if(x[i].index > y[j].index)
                                        {
                                                sum += y[j].value * y[j].value;
                                                ++j;
                                        }
                                        else
                                        {
                                                sum += x[i].value * x[i].value;
                                                ++i;
                                        }
                                }

                                while(i < xlen)
                                {
                                        sum += x[i].value * x[i].value;
                                        ++i;
                                }

                                while(j < ylen)
                                {
                                        sum += y[j].value * y[j].value;
                                        ++j;
                                }
                                return Math.exp(-param.gamma*sum);
                        }
                        case svm_parameter.SIGMOID:
                                return tanh(param.gamma*dot(x,y)+param.coef0);
                        case svm_parameter.PRECOMPUTED:
                                return  x[(int)(y[0].value)].value;
                        default:
                                return 0;
                }
        }
}
```

/* viterbi.cpp */

```
#include <math.h>
#include "viterbi.h"
#include "model.h"

using namespace std;

viterbi::viterbi() {
   pmodel = NULL;
   popt = NULL;
   pdata = NULL;
   pdict = NULL;
   pfgen = NULL;

   Mi = NULL;
   Vi = NULL;
}

viterbi::~viterbi() {
   if (Mi) {
         delete Mi;
   }

   if (Vi) {
         delete Vi;
   }
}

void viterbi::init(model * pmodel) {
   this->pmodel = pmodel;
   popt = pmodel->popt;
   pdata = pmodel->pdata;
   pdict = pmodel->pdict;
   pfgen = pmodel->pfgen;

   int dim = popt->num_labels;
   if (popt->order == SECOND_ORDER) {
         dim = popt->num_2orderlabels;
   }

   Mi = new doublematrix(dim, dim);
   Vi = new doublevector(dim);

   for (int i = 0; i < dim; i++) {
         temp.push_back(pair<double, int>(0.0, -1));
   }

   // mapping label (string) => label (index) for constraints
   maplbstr2int::iterator it;
```

```
popt->prevfixedintlabels.clear();
popt->nextfixedintlabels.clear();

vector<int> labels;

int len = popt->prevfixedstrlabels.size();
for (int i = 0; i < len; i++) {
      labels.clear();

      it = pdata->plbs2i->find(popt->prevfixedstrlabels[i][0]);
      if (it != pdata->plbs2i->end()) {
         labels.push_back(it->second);
      } else {
         continue;
      }

      for (int j = 1; j < popt->prevfixedstrlabels[i].size(); j++) {
         it = pdata->plbs2i->find(popt->prevfixedstrlabels[i][j]);
         if (it != pdata->plbs2i->end()) {
               labels.push_back(it->second);
         }
      }

      if (labels.size() <= 1) {
         continue;
      }

      popt->prevfixedintlabels.push_back(labels);
}

len = popt->nextfixedstrlabels.size();
for (int i = 0; i < len; i++) {
      labels.clear();

      it = pdata->plbs2i->find(popt->nextfixedstrlabels[i][0]);
      if (it != pdata->plbs2i->end()) {
         labels.push_back(it->second);
      } else {
         continue;
      }

      for (int j = 1; j < popt->nextfixedstrlabels[i].size(); j++) {
         it = pdata->plbs2i->find(popt->nextfixedstrlabels[i][j]);
         if (it != pdata->plbs2i->end()) {
               labels.push_back(it->second);
         }
      }

      if (labels.size() <= 1) {
            continue;
        }

        popt->nextfixedintlabels.push_back(labels);
    }
}

void viterbi::computeMi() {
    *Mi = 0;

    pfgen->start_scan_efeatures();
    while (pfgen->has_next_efeature()) {
        feature f;
        pfgen->next_efeature(f);

        if (f.ftype == EDGE_FEATURE1) {
            // edge feature type 1

            if (popt->order == FIRST_ORDER) {
                Mi->get(f.yp, f.y) += pmodel->lambda[f.idx] * f.val;

            } else if (popt->order == SECOND_ORDER) {
                int col = f.yp * popt->num_labels + f.y;
                for (int row = 0; row < Mi->rows; row++) {
                    Mi->get(row, col) += pmodel->lambda[f.idx] * f.val;
                }
            }

        } else if (f.ftype == EDGE_FEATURE2) {
            // edge feature type 2

            if (popt->order == FIRST_ORDER) {
                // do nothing

            } else if (popt->order == SECOND_ORDER) {
                Mi->get(f.yp, f.y) += pmodel->lambda[f.idx] * f.val;
            }
        }
    }

    if (popt->order == FIRST_ORDER) {
        for (int i = 0; i < Mi->rows; i++) {
            for (int j = 0; j < Mi->cols; j++) {
                Mi->get(i, j) = exp(Mi->get(i, j));
            }
        }

    } else if (popt->order == SECOND_ORDER) {
        for (int i = 0; i < Mi->rows; i++) {
            for (int j = 0; j < Mi->cols; j++) {
                if (i % popt->num_labels == j / popt->num_labels) {
                    Mi->get(i, j) = exp(Mi->get(i, j));
                } else {
                    Mi->get(i, j) = 0.0;
                }
            }
        }
    }
}

void viterbi::apply(dataset * pdataset) {
    computeMi();

    dataset::iterator datait;
    int count = 0;
    for (datait = pdataset->begin(); datait != pdataset->end(); datait++) {
        if (popt->order == FIRST_ORDER) {
            apply_1order(*datait);
        } else if (popt->order == SECOND_ORDER) {
            apply_2order(*datait);
        }
        count++;
        // cout << "sequence: " << count << endl;
    }
}

void viterbi::apply(dataset * pdataset, int n) {
    computeMi();

    dataset::iterator datait;
    int count = 0;
    for (datait = pdataset->begin(); datait != pdataset->end(); datait++) {
        if (popt->order == FIRST_ORDER) {
            apply_1order(*datait, n);
        } else if (popt->order == SECOND_ORDER) {
            apply_2order(*datait, n);
        }
        count++;
        // cout << "sequence: " << count << endl;
    }
}

void viterbi::apply_1order(sequence & seq) {
    int i, j, k;

    int seq_len = seq.size();
    if (seq_len == 0) {
        return;
    }
    int memorysize = memory.size();
    // if the current sequence is the longest one (up to the current point),
    // then allocate more memory
    if (memorysize < seq_len) {
        for (i = 0; i < seq_len - memorysize; i++) {
          memory.push_back(temp);
        }
    }

    // we need to scale forward variable to [0, 1] to avoid numerical problems
    int scalesize = scale.size();
    // if the current sequence is the longest one (up to the current point),
    // then allocate more room for scale variable
    if (scalesize < seq_len) {
        for (i = 0; i < seq_len - scalesize; i++) {
          scale.push_back(1.0);
        }
    }

    // compute Mi and Vi for the first position in the sequence
    compute_log_Mi_1order(seq, 0, Mi, Vi, 1);
    for (j = 0; j < popt->num_labels; j++) {
        memory[0][j].first = (*Vi)[j];
        memory[0][j].second = j;
    }
    // calculate scale factor for the first position
    scale[0] = (popt->is_scaling) ? viterbi::sum(memory[0]) : 1;
    // scaling for the first position
    viterbi::divide(memory[0], scale[0]);

    // the main loop
    for (i = 1; i < seq_len; i++) {
        // compute Mi matrix and Vi vector at position "i"
        compute_log_Mi_1order(seq, i, Mi, Vi, 1);

        // applying constraints
        int num_cnts = popt->prevfixedintlabels.size();
        for (int cc = 0; cc < num_cnts; cc++) {
          int col = popt->prevfixedintlabels[cc][0];
          for (int row = 0; row < popt->num_labels; row++) {
              int in = 0;
              for (int count = 1; count < popt->prevfixedintlabels[cc].size();
count++) {
                if (row == popt->prevfixedintlabels[cc][count]) {
                    in = 1;
                }
              }
              if (!in) {
                Mi->mtrx[row][col] = 0;
              }
```

```
            }
        }

        num_cnts = popt->nextfixedintlabels.size();
        for (int cc = 0; cc < num_cnts; cc++) {
            int row = popt->nextfixedintlabels[cc][0];
            for (int col = 0; col < popt->num_labels; col++) {
                int in = 0;
                for (int count = 1; count < popt->nextfixedintlabels[cc].size();
count++) {
                    if (col == popt->nextfixedintlabels[cc][count]) {
                        in = 1;
                    }
                }
                if (!in) {
                    Mi->mtrx[row][col] = 0;
                }
            }
        }

        // for all possible labels at the position "i"
        for (j = 0; j < popt->num_labels; j++) {
            memory[i][j].first = 0.0;
            memory[i][j].second = 0;

            // find the maximal value and its index and store them in memory
            // for later tracing back to find the best path
            for (k = 0; k < popt->num_labels; k++) {
                double tempval = memory[i-1][k].first * Mi->mtrx[k][j] * (*Vi)[j];

                if (tempval > memory[i][j].first) {
                    memory[i][j].first = tempval;
                    memory[i][j].second = k;
                }
            }
        }

        // scaling for memory at position "i"
        scale[i] = (popt->is_scaling) ? viterbi::sum(memory[i]) : 1;
        viterbi::divide(memory[i], scale[i]);
    }

    // viterbi backtrack to find the best path
    int max_idx = viterbi::find_max(memory[seq_len - 1]);
    seq[seq_len - 1].model_label = max_idx;
    for (i = seq_len - 2; i >= 0; i--) {
        seq[i].model_label = memory[i + 1][max_idx].second;
        max_idx = seq[i].model_label;
    }
}
```

```
void viterbi::apply_1order(sequence & seq, int n) {
  int i, j, k, h;

  int seq_len = seq.size();
  if (seq_len == 0) {
       return;
  }

  mem infor;
  infor.pathval = 0.0;
  infor.previouslabel = -1;
  infor.previousindex = -1;

  int premaxlen = statelens.size();
  if (premaxlen < seq_len) {
       for (i = 0; i < seq_len - premaxlen; i++) {
          statelens.push_back(0);
       }
  }

  if (statelbls.size() != n * popt->num_labels) {
       for (i = 0; i < n * popt->num_labels; i++) {
          statelbls.push_back(infor);
          sortidxes.push_back(pair<int, double>(0, 0.0));
       }
  }

  premaxlen = seqlbls.size();
  if (premaxlen < seq_len) {
       for (i = 0; i < seq_len - premaxlen; i++) {
          seqlbls.push_back(statelbls);
       }
  }

  // scaling
  premaxlen = scale.size();
  if (premaxlen < seq_len) {
       for (i = 0; i < seq_len - premaxlen; i++) {
          scale.push_back(1.0);
       }
  }

  // compute Mi and Vi for the first position in the sequence
  compute_log_Mi_1order(seq, 0, Mi, Vi, 1);

  statelens[0] = 1;
  // for the first position
  for (j = 0; j < popt->num_labels; j++) {
       seqlbls[0][j * n].pathval = (*Vi)[j];
```

```
        seqlbls[0][j * n].previouslabel = j;
        seqlbls[0][j * n].previousindex = 0;
    }

    // scaling
    scale[0] = (popt->is_scaling) ?
                viterbi::sum(seqlbls[0], popt->num_labels, n, statelens[0]) : 1;
    viterbi::divide(seqlbls[0], popt->num_labels, n, statelens[0], scale[0]);

    // the main loop
    for (i = 1; i < seq_len; i++) {
        // compute Mi matrix and Vi vector at position "i"
        compute_log_Mi_1order(seq, i, Mi, Vi, 1);

        statelens[i] = n;
        if (statelens[i] > statelens[i - 1] * popt->num_labels) {
          statelens[i] = statelens[i - 1] * popt->num_labels;
        }

        // for all possible labels at the position "i"
        for (j = 0; j < popt->num_labels; j++) {
          int count = 0;
          // for all possible labels at the position "i-1"
          for (k = 0; k < popt->num_labels; k++) {
                for (h = 0; h < statelens[i - 1]; h++) {
                  sortidxes[count].first = k * n + h;
                  sortidxes[count].second =
                        seqlbls[i - 1][k * n + h].pathval *
                        Mi->mtrx[k][j] * (*Vi)[j];

                  count++;
                }
          }

          quicksort(sortidxes, 0, count - 1);

          for (k = 0; k < statelens[i]; k++) {
                seqlbls[i][j * n + k].pathval = sortidxes[k].second;
                seqlbls[i][j * n + k].previouslabel = sortidxes[k].first / n;
                seqlbls[i][j * n + k].previousindex = sortidxes[k].first % n;
          }
        } // end of (for all possible labels at the position "i")

        // scaling for the current position
        scale[i] = (popt->is_scaling) ?
                viterbi::sum(seqlbls[i], popt->num_labels, n, statelens[i]) : 1;
        viterbi::divide(seqlbls[i], popt->num_labels, n, statelens[i], scale[i]);

    } // end of the main loop
```

```
int count = 0;
for (j = 0; j < popt->num_labels; j++) {
      for (k = 0; k < statelens[seq_len - 1]; k++) {
         sortidxes[count].first = j * n + k;
         sortidxes[count].second = seqlbls[seq_len - 1][j * n + k].pathval;
         count++;
      }
}

quicksort(sortidxes, 0, count - 1);

int realsize = n;
if (realsize > count) {
      realsize = count;
}

// allocate memory for n-best path information
for (i = 0; i < seq_len; i++) {
      seq[i].pnbestinfo = new nbestinfo;
      while (seq[i].pnbestinfo->model_labels.size() < realsize) {
         seq[i].pnbestinfo->model_labels.push_back(-1);
      }

      if (i == 0) {
         while (seq[0].pnbestinfo->pathvals.size() < realsize) {
               seq[0].pnbestinfo->pathvals.push_back(0.0);
         }
      }
}

double sumpathvals = 0.0;
// n-best backtracking
for (i = 0; i < realsize; i++) {
      seq[0].pnbestinfo->pathvals[i] = sortidxes[i].second;
      sumpathvals += seq[0].pnbestinfo->pathvals[i];

      int major = sortidxes[i].first / n;
      seq[seq_len - 1].pnbestinfo->model_labels[i] = major;
      int minor = sortidxes[i].first % n;

      for (j = seq_len - 2; j >= 0; j--) {
         seq[j].pnbestinfo->model_labels[i] =
               seqlbls[j + 1][major * n + minor].previouslabel;
         int mj = seqlbls[j + 1][major * n + minor].previouslabel;
         int mn = seqlbls[j + 1][major * n + minor].previousindex;
         major = mj;
         minor = mn;
      }
}
```

```
    // scaling path values
    if (sumpathvals > 0) {
        for (i = 0; i < realsize; i++) {
            seq[0].pnbestinfo->pathvals[i] = seq[0].pnbestinfo->pathvals[i] /
sumpathvals;
        }
    }

    // calculating entropy
    vector<double> ps;
    for (i = 0; i < seq_len; i++) {
        // the best path
        seq[i].model_label = seq[i].pnbestinfo->model_labels[0];

        ps.clear();
        for (j = 0; j < popt->num_labels; j++) {
            ps.push_back(0.0);
        }

        for (j = 0; j < realsize; j++) {
            ps[seq[i].pnbestinfo->model_labels[j]] += seq[0].pnbestinfo->pathvals[j];
        }

        int count = 0;
        for (j = 0; j < popt->num_labels; j++) {
            if (ps[j] > 0.0) {
                count++;
            }
        }

        seq[i].pnbestinfo->entropyval = 0.0;
        if (count > 1) {
            for (j = 0; j < popt->num_labels; j++) {
                if (ps[j] > 0.0) {
                    seq[i].pnbestinfo->entropyval -= ps[j] * log(ps[j]);
                }
            }
        }
        seq[i].pnbestinfo->entropyval /= ps.size();
    }
}

void viterbi::apply_2order(sequence & seq) {
    int i, j, k;

    map<int, pair<int, int> >::iterator lbmapit;

    int seq_len = seq.size();
    if (seq_len == 0) {
        return;
```

```
}

int lfo = popt->num_labels - 1;
if (popt->lfo >= 0) {
    lfo = popt->lfo;
}

int memorysize = memory.size();
// if the current sequence is the longest one (up to the current point),
// then allocate more memory
if (memorysize < seq_len) {
    for (i = 0; i < seq_len - memorysize; i++) {
      memory.push_back(temp);
    }
}

// we need to scale forward variable to [0, 1] to avoid numerical problems
int scalesize = scale.size();
// if the current sequence is the longest one (up to the current point),
// then allocate more room for scale variable
if (scalesize < seq_len) {
    for (i = 0; i < seq_len - scalesize; i++) {
      scale.push_back(1.0);
    }
}

// compute Mi and Vi for the first position in the sequence
compute_log_Mi_2order(seq, 0, Mi, Vi, 1);

for (j = 0; j < popt->num_2orderlabels; j++) {
    memory[0][j].first = (*Vi)[j];
    memory[0][j].second = j;
}
// calculate scale factor for the first position
scale[0] = (popt->is_scaling) ? viterbi::sum(memory[0]) : 1;
// scaling for the first position
viterbi::divide(memory[0], scale[0]);

// the main loop
for (i = 1; i < seq_len; i++) {
    // compute Mi matrix and Vi vector at position "i"
    compute_log_Mi_2order(seq, i, Mi, Vi, 1);

    // applying constraints
    int num_cnts = popt->prevfixedintlabels.size();
    for (int cc = 0; cc < num_cnts; cc++) {
      int col = popt->prevfixedintlabels[cc][0];
      for (int row = 0; row < popt->num_labels; row++) {
          int in = 0;
```

```
                for (int count = 1; count < popt->prevfixedintlabels[cc].size();
count++) {
                    if (row == popt->prevfixedintlabels[cc][count]) {
                        in = 1;
                    }
                }
                if (!in) {
                    int index = row * popt->num_labels + col;
                    (*Vi)[index] = 0;
                }
            }
        }

        num_cnts = popt->nextfixedintlabels.size();
        for (int cc = 0; cc < num_cnts; cc++) {
            int row = popt->nextfixedintlabels[cc][0];
            for (int col = 0; col < popt->num_labels; col++) {
                int in = 0;
                for (int count = 1; count < popt->nextfixedintlabels[cc].size();
count++) {
                    if (col == popt->nextfixedintlabels[cc][count]) {
                        in = 1;
                    }
                }
                if (!in) {
                    int index = row * popt->num_labels + col;
                    (*Vi)[index] = 0;
                }
            }
        }

        // for all possible labels at the position "i"
        for (j = 0; j < popt->num_2orderlabels; j++) {
            memory[i][j].first = 0.0;
            memory[i][j].second = 0;

            // find the maximal value and its index and store them in memory
            // for later tracing back to find the best path
            for (k = 0; k < popt->num_2orderlabels; k++) {
                double tempval = memory[i-1][k].first * Mi->mtrx[k][j] * (*Vi)[j];

                if (tempval > memory[i][j].first) {
                    memory[i][j].first = tempval;
                    memory[i][j].second = k;
                }
            }
        }

        // scaling for memory at position "i"
        scale[i] = (popt->is_scaling) ? viterbi::sum(memory[i]) : 1;
        viterbi::divide(memory[i], scale[i]);
    }

    // viterbi backtrack to find the best path
    int max_idx = viterbi::find_max(memory[seq_len - 1]);
    seq[seq_len - 1].model_label = max_idx;
    for (i = seq_len - 2; i >= 0; i--) {
        seq[i].model_label = memory[i + 1][max_idx].second;
        max_idx = seq[i].model_label;
    }

    // converting from second-order labels to first-order ones
    for (i = 0; i < seq_len; i++) {
        lbmapit = pdata->plb2to1->find(seq[i].model_label);
        if (lbmapit != pdata->plb2to1->end()) {
            seq[i].model_label = lbmapit->second.second;
        }
    }
}

void viterbi::apply_2order(sequence & seq, int n) {
    int i, j, k, h;

    map<int, pair<int, int> >::iterator lbmapit;

    int seq_len = seq.size();
    if (seq_len == 0) {
        return;
    }

    mem infor;
    infor.pathval = 0.0;
    infor.previouslabel = -1;
    infor.previousindex = -1;

    int premaxlen = statelens.size();
    if (premaxlen < seq_len) {
        for (i = 0; i < seq_len - premaxlen; i++) {
            statelens.push_back(0);
        }
    }

    if (statelbls.size() != n * popt->num_2orderlabels) {
        for (i = 0; i < n * popt->num_2orderlabels; i++) {
            statelbls.push_back(infor);
            sortidxes.push_back(pair<int, double>(0, 0.0));
        }
    }

    premaxlen = seqlbls.size();
    if (premaxlen < seq_len) {
        for (i = 0; i < seq_len - premaxlen; i++) {
            seqlbls.push_back(statelbls);
        }
    }

    // scaling
    premaxlen = scale.size();
    if (premaxlen < seq_len) {
        for (i = 0; i < seq_len - premaxlen; i++) {
            scale.push_back(1.0);
        }
    }

    // compute Mi and Vi for the first position in the sequence
    compute_log_Mi_2order(seq, 0, Mi, Vi, 1);

    statelens[0] = 1;
    // for the first position
    for (j = 0; j < popt->num_2orderlabels; j++) {
        seqlbls[0][j * n].pathval = (*Vi)[j];
        seqlbls[0][j * n].previouslabel = j;
        seqlbls[0][j * n].previousindex = 0;
    }

    // scaling
    scale[0] = (popt->is_scaling) ?
                    viterbi::sum(seqlbls[0], popt->num_2orderlabels, n, statelens[0]) : 1;
    viterbi::divide(seqlbls[0], popt->num_2orderlabels, n, statelens[0], scale[0]);

    // the main loop
    for (i = 1; i < seq_len; i++) {
        // compute Mi matrix and Vi vector at position "i"
        compute_log_Mi_2order(seq, i, Mi, Vi, 1);

        statelens[i] = n;
        if (statelens[i] > statelens[i - 1] * popt->num_2orderlabels) {
            statelens[i] = statelens[i - 1] * popt->num_2orderlabels;
        }

        // for all possible labels at the position "i"
        for (j = 0; j < popt->num_2orderlabels; j++) {
            int count = 0;
            // for all possible labels at the position "i-1"
            for (k = 0; k < popt->num_2orderlabels; k++) {
                for (h = 0; h < statelens[i - 1]; h++) {
                    sortidxes[count].first = k * n + h;
                    sortidxes[count].second =
                                        seqlbls[i - 1][k * n + h].pathval * Mi->mtrx[k][j]
* (*Vi)[j];
```

```
                count++;
            }
        }

        quicksort(sortidxes, 0, count - 1);

        for (k = 0; k < statelens[i]; k++) {
            seqlbls[i][j * n + k].pathval = sortidxes[k].second;
            seqlbls[i][j * n + k].previouslabel = sortidxes[k].first / n;
            seqlbls[i][j * n + k].previousindex = sortidxes[k].first % n;
        }
    } // end of (for all possible labels at the position "i")

    // scaling for the current position
    scale[i] = (popt->is_scaling) ?
            viterbi::sum(seqlbls[i], popt->num_2orderlabels, n, statelens[i]) : 1;
    viterbi::divide(seqlbls[i], popt->num_2orderlabels, n, statelens[i], scale[i]);

} // end of the main loop

int count = 0;
for (j = 0; j < popt->num_2orderlabels; j++) {
    for (k = 0; k < statelens[seq_len - 1]; k++) {
        sortidxes[count].first = j * n + k;
        sortidxes[count].second = seqlbls[seq_len - 1][j * n + k].pathval;
        count++;
    }
}

quicksort(sortidxes, 0, count - 1);

int realsize = n;
if (realsize > count) {
    realsize = count;
}

// allocate memory for n-best path information
for (i = 0; i < seq_len; i++) {
    seq[i].pnbestinfo = new nbestinfo;
    while (seq[i].pnbestinfo->model_labels.size() < realsize) {
        seq[i].pnbestinfo->model_labels.push_back(-1);
    }

    if (i == 0) {
        while (seq[0].pnbestinfo->pathvals.size() < realsize) {
            seq[0].pnbestinfo->pathvals.push_back(0.0);
        }
    }
}
```

```
    double sumpathvals = 0.0;
    // n-best backtracking
    for (i = 0; i < realsize; i++) {
        seq[0].pnbestinfo->pathvals[i] = sortidxes[i].second;
        sumpathvals += seq[0].pnbestinfo->pathvals[i];

        int major = sortidxes[i].first / n;
        seq[seq_len - 1].pnbestinfo->model_labels[i] = major;
        int minor = sortidxes[i].first % n;

        for (j = seq_len - 2; j >= 0; j--) {
            seq[j].pnbestinfo->model_labels[i] =
                    seqlbls[j + 1][major * n + minor].previouslabel;
            int mj = seqlbls[j + 1][major * n + minor].previouslabel;
            int mn = seqlbls[j + 1][major * n + minor].previousindex;
            major = mj;
            minor = mn;
        }
    }

    // scaling path values
    if (sumpathvals > 0) {
        for (i = 0; i < realsize; i++) {
            seq[0].pnbestinfo->pathvals[i] = seq[0].pnbestinfo->pathvals[i] /
sumpathvals;
        }
    }

    // calculating entropy
    vector<double> ps;
    for (i = 0; i < seq_len; i++) {
        // the best path
        seq[i].model_label = seq[i].pnbestinfo->model_labels[0];

        ps.clear();
        for (j = 0; j < popt->num_2orderlabels; j++) {
            ps.push_back(0.0);
        }

        for (j = 0; j < realsize; j++) {
            ps[seq[i].pnbestinfo->model_labels[j]] += seq[0].pnbestinfo->pathvals[j];
        }

        int count = 0;
        for (j = 0; j < popt->num_2orderlabels; j++) {
            if (ps[j] > 0.0) {
                count++;
            }
        }
```

```
            seq[i].pnbestinfo->entropyval = 0.0;
            if (count > 1) {
                for (j = 0; j < popt->num_2orderlabels; j++) {
                        if (ps[j] > 0.0) {
                            seq[i].pnbestinfo->entropyval -= ps[j] * log(ps[j]);
                        }
                }
            }
            seq[i].pnbestinfo->entropyval /= ps.size();
    }

    // converting from second-order labels to first-order ones
    for (i = 0; i < seq_len; i++) {
            lbmapit = pdata->plb2to1->find(seq[i].model_label);
            if (lbmapit != pdata->plb2to1->end()) {
                seq[i].model_label = lbmapit->second.second;
            }

            for (j = 0; j < realsize; j++) {
                lbmapit = pdata->plb2to1->find(seq[i].pnbestinfo->model_labels[j]);
                if (lbmapit != pdata->plb2to1->end()) {
                        seq[i].pnbestinfo->model_labels[j] = lbmapit->second.second;
                }
            }
    }
}

// compute log Mi (for first-order Markov)
void viterbi::compute_log_Mi_1order(sequence & seq, int pos, doublematrix * Mi,
                    doublevector * Vi, int is_exp) {
    *Vi = 0.0;

    // start scan features for sequence "seq" at position "i"
    pfgen->start_scan_sfeatures_at(seq, pos);
    // examine all features at position "pos"
    while (pfgen->has_next_sfeature()) {
            feature f;
            pfgen->next_sfeature(f);

            if (f.ftype == STAT_FEATURE1) {
                // state feature (type 1)
                (*Vi)[f.y] += pmodel->lambda[f.idx] * f.val;

            }
    }

    // take exponential operator
    if (is_exp) {
            for (int i = 0; i < Mi->rows; i++) {
```

```
            // update for Vi
            (*Vi)[i] = exp((*Vi)[i]);
        }
    }
}

// compute log Mi (second-order Markov)
void viterbi::compute_log_Mi_2order(sequence & seq, int pos, doublematrix * Mi,
                    doublevector * Vi, int is_exp) {
    *Vi = 0.0;

    // start scan features for sequence "seq" at position "i"
    pfgen->start_scan_sfeatures_at(seq, pos);
    // examine all features at position "pos"
    while (pfgen->has_next_sfeature()) {
        feature f;
        pfgen->next_sfeature(f);

        if (f.ftype == STAT_FEATURE1) {
            // state feature (type 1)
            for (int i = 0; i < popt->num_labels; i++) {
                (*Vi)[i * popt->num_labels + f.y] += pmodel->lambda[f.idx] * f.val;
            }

        } else if (f.ftype == STAT_FEATURE2) {
            // state feature (type 2)
            (*Vi)[f.y] += pmodel->lambda[f.idx] * f.val;
        }
    }

    int lfo = popt->num_labels - 1;
    if (popt->lfo >= 0) {
        lfo = popt->lfo;
    }

    // take exponential operator
    if (is_exp) {
        if (pos == 0) {
            for (int j = 0; j < Mi->rows; j++) {
                if (j / popt->num_labels == lfo) {
                    (*Vi)[j] = exp((*Vi)[j]);
                } else {
                    (*Vi)[j] = 0;
                }
            }

        } else {
            for (int j = 0; j < Mi->rows; j++) {
                (*Vi)[j] = exp((*Vi)[j]);
            }
```

```
      }
    }
}

// this is used by viterbi search
double viterbi::sum(vector<pair<double, int> > & vect) {
   double res = 0.0;

   for (int i = 0; i < vect.size(); i++) {
         res += vect[i].first;
   }

   // if the sum in (-1, 1), then set it to 1
   if (res < 1 && res > -1) {
         res = 1;
   }

   return res;
}

// this is necessary for scaling
double viterbi::divide(vector<pair<double, int> > & vect, double val) {
   for (int i = 0; i < vect.size(); i++) {
         vect[i].first /= val;
   }
}

// this is called once in the viterbi search to trace back the best path
int viterbi::find_max(vector<pair<double, int> > & vect) {
   int max_idx = 0;
   double max_val = -1.0;

   for (int i = 0; i < vect.size(); i++) {
         if (vect[i].first > max_val) {
            max_val = vect[i].first;
            max_idx = i;
         }
   }

   return max_idx;
}

double viterbi::sum(vector<pair<vector<mem>, int> > & vect) {
   double res = 0.0;

   for (int i = 0; i < vect.size(); i++) {
         for (int j = 0; j < vect[i].second; j++) {
            res += vect[i].first[j].pathval;
         }
   }
```

```
    if (res < 1 && res > -1) {
          res = 1;
    }

    return res;
}

double viterbi::sum(vector<mem> & vect, int num_labels, int n, int len) {
    double res = 0.0;

    for (int i = 0; i < num_labels; i++) {
          for (int j = 0; j < len; j++) {
             res += vect[i * n + j].pathval;
          }
    }

    if (res < 1 && res > -1) {
          res = 1;
    }

    return res;
}

double viterbi::divide(vector<pair<vector<mem>, int> > & vect, double val) {
    for (int i = 0; i < vect.size(); i++) {
          for (int j = 0; j < vect[i].second; j++) {
             vect[i].first[j].pathval /= val;
          }
    }
}

double viterbi::divide(vector<mem> & vect, int num_labels, int n,
          int len, double val) {

    for (int i = 0; i < num_labels; i++) {
          for (int j = 0; j < len; j++) {
             vect[i * n + j].pathval /= val;
          }
    }
}

void viterbi::quicksort(vector<pair<int, double> > & vect, int left, int right) {
    int l_hold, r_hold;
    pair<int, double> pivot;

    l_hold = left;
    r_hold = right;
    int pivotidx = left;
    pivot = vect[pivotidx];
    while (left < right) {
          while (vect[right].second <= pivot.second && left < right) {
             right--;
          }
          if (left != right) {
             vect[left] = vect[right];
             left++;
          }
          while (vect[left].second >= pivot.second && left < right) {
             left++;
          }
          if (left != right) {
             vect[right] = vect[left];
             right--;
          }
    }

    vect[left] = pivot;
    pivotidx = left;
    left = l_hold;
    right = r_hold;

    if (left < pivotidx) {
          quicksort(vect, left, pivotidx - 1);
    }
    if (right > pivotidx) {
          quicksort(vect, pivotidx + 1, right);
    }
}
```

/* crf.cpp*/

```
#include <stdio.h>
#include <stdlib.h>
#include "strtokenizer.h"
#include "option.h"
#include "data.h"
#include "dictionary.h"
#include "feature.h"
#include "featuregen.h"
#include "trainer.h"
#include "viterbi.h"
#include "valuation.h"
#include "model.h"

using namespace std;

int main(int argc, char ** argv) {
   // the command line must be:
   // segment -all/-trn/-tst/-prd -d <model directory> -o <option filename>
```

```
int is_all = !strcmp(argv[1], "-all");
int is_trn = !strcmp(argv[1], "-trn");
int is_tst = !strcmp(argv[1], "-tst");
int is_prd = !strcmp(argv[1], "-prd");

// the first parameter must be "-all", "-train", or "-test"
if (!is_all && !is_trn && !is_tst && !is_prd) {
            return 0;
}

// the second parameter must be "-d"
if (strcmp(argv[2], "-d")) {
      return 0;
}

// the third parameter must be the model directory
string model_dir = argv[3];
if (model_dir[model_dir.size() - 1] != '/') {
      model_dir += "/";
}

// the fourth parameter must be "-o"
if (strcmp(argv[4], "-o")) {
            return 0;
}

// the fifth parameter must be the option filename
string optfile = model_dir + argv[5];

// open the option file to read parameters
FILE * optf;
optf = fopen(optfile.c_str(), "r");
if (!optf) {
      // if can not open the option file
      printf("cannot open the option file for reading\n");
      return 0;
}

// create an option object
option opt(optf, model_dir);

FILE * itrndataf;
FILE * itstdataf;
FILE * iulbdataf;

FILE * imodelf;

if (is_all || is_trn) {
      // try to open training data file
```

```
        itrndataf = fopen((opt.model_dir + opt.trndata_file).c_str(), "r");
        if (!itrndataf) {
           printf("cannot open the training data file for reading\n");
           return 0;
        }
  }

  if (is_all || is_tst) {
        // try to open testing data file
        itstdataf = fopen((opt.model_dir + opt.tstdata_file).c_str(), "r");
        if (!itstdataf) {
           printf("cannot open the testing data file for reading\n");
           return 0;
        }
  }

  if (is_prd) {
        // try to open unlabeled data file
        iulbdataf = fopen((opt.model_dir + opt.ulbdata_file).c_str(), "r");
        if (!iulbdataf) {
           printf("cannot open the unlabeled data file for reading\n");
           return 0;
        }
  }

  if (is_tst || is_prd) {
        // try to open model file (contain cpmap, labelmap, dictionary, and features)
        imodelf = fopen((opt.model_dir + opt.model_file).c_str(), "r");
        if (!imodelf) {
           printf("cannot open the model file for reading\n");
           return 0;
        }
  }

  // data object
  data dt(&opt);
  // dictionary object
  dictionary dict(&dt, &opt);
  // featuregen object
  featuregen fgen(&opt, &dict, &dt);

  // both training and testing
  if (is_all) {
        printf("reading training data ...\n");
        dt.read_trndata(itrndataf);
        printf("reading %d training sequence completed.\n\n", opt.num_trnseqs);

        printf("reading testing data ...\n");
        dt.read_tstdata(itstdataf);
        printf("reading %d testing sequences completed.\n\n", opt.num_tstseqs);
```

```
printf("generating dictionary ...\n");
dict.dict_gen();
printf("generating %d context predicates completed.\n\n", opt.num_cps);

printf("generating CRF features from training data ...\n");
fgen.gen_features();
printf("generating %d CRF features completed.\n\n", opt.num_features);

printf("pruning unused context predicates ...\n");
dt.cp_prune(&dict);
printf("the number of context predicates after pruning: %d\n\n", opt.num_cps);

// create model file
FILE * omodelf;
omodelf = fopen((opt.model_dir + opt.model_file).c_str(), "w");

printf("saving context predicate map ...\n");
dt.write_cp_map(omodelf, &dict);
printf("saving context predicate map completed.\n\n");

printf("saving label map file ...\n");
dt.write_lb_map(omodelf);
printf("saving label map completed.\n\n");

if (opt.order == SECOND_ORDER) {
   printf("saving second-order label map ...\n");
   dt.write_2order_lb_map(omodelf);
   printf("saving second-order label map completed.\n\n");

   printf("saving second-order to first-order label map ...\n");
   dt.write_2to1_lb_map(omodelf);
   printf("saving second-order to first-order label map completed.\n\n");
}

// create the trainer object
trainer trn;
// create the viterbi object
viterbi vtb;
// create the evaluation object
evaluation eval;

// create the CRF model
model mdl(&opt, &dt, &dict, &fgen, &trn, &vtb, &eval);

// create training log file
FILE * otrnlogf;
if (opt.is_logging) {
   otrnlogf = fopen((opt.model_dir + opt.trainlog_file).c_str(), "w");
}
```

```
		// saving dictionary
		printf("saving the context predicate dictionary to file ...\n");
		dict.write_dict(omodelf);
		printf("saving the dictionary completed.\n\n");

		// start to train the CRF model
		printf("start to train the CRF model ...\n\n");
		mdl.train(otrnlogf);

		// saving the set of features
		printf("saving the CRF features to file ...\n");
		fgen.write_features(omodelf);
		printf("saving the CRF features completed.\n\n");

		// start to label for testing data
		printf("labeling for testing data ...\n");
		mdl.apply_tstdata();
		printf("labeling for testing data completed.\n\n");

		// saving testing output to file
		FILE * otstdataf;
		otstdataf = fopen((opt.model_dir + opt.tstdata_file + ".model").c_str(), "w");
		printf("writing testing output to file ...\n");
		dt.write_tstdata(otstdataf);
		printf("writing testing output completed.\n\n");

		// performance evaluation on testing dataset
		mdl.peval->evaluate(otrnlogf);
		printf("\n");

		fclose(optf);
		fclose(itrndataf);
		fclose(itstdataf);
		fclose(otstdataf);
		fclose(otrnlogf);
		fclose(omodelf);
	}

	// training only
	if (is_trn) {
		printf("reading training data ...\n");
		dt.read_trndata(itrndataf);
		printf("reading %d training sequence completed.\n\n", opt.num_trnseqs);

		printf("generating dictionary ...\n");
		dict.dict_gen();
		printf("generating %d context predicates completed.\n\n", opt.num_cps);

		printf("generating CRF features from training data ...\n");
```

```
fgen.gen_features();
printf("generating %d CRF features completed.\n\n", opt.num_features);

printf("pruning unused context predicates ...\n");
dt.cp_prune(&dict);
printf("the number of context predicates after pruning: %d\n\n", opt.num_cps);

// create model file
FILE * omodelf;
omodelf = fopen((opt.model_dir + opt.model_file).c_str(), "w");

printf("saving context predicate map ...\n");
dt.write_cp_map(omodelf, &dict);
printf("saving context predicate map completed.\n\n");

printf("saving label map file ...\n");
dt.write_lb_map(omodelf);
printf("saving label map completed.\n\n");

if (opt.order == SECOND_ORDER) {
   printf("saving second-order label map ...\n");
   dt.write_2order_lb_map(omodelf);
   printf("saving second-order label map completed.\n\n");

   printf("saving second-order to first-order label map ...\n");
   dt.write_2to1_lb_map(omodelf);
   printf("saving second-order to first-order label map completed.\n\n");
}

// create the trainer object
trainer trn;

// create the CRF model
model mdl(&opt, &dt, &dict, &fgen, &trn, NULL, NULL);

// create training log file
FILE * otrnlogf;
if (opt.is_logging) {
   otrnlogf = fopen((opt.model_dir + opt.trainlog_file).c_str(), "w");
}

// save the dictionary
printf("saving the context predicate dictionary to file ...\n");
dict.write_dict(omodelf);
printf("saving the dictionary completed.\n\n");

// start to train the CRF model
printf("start to train the CRF model ...\n\n");
mdl.train(otrnlogf);
```

```
        // saving the set of features
        printf("saving the CRF features to file ...\n");
        fgen.write_features(omodelf);
        printf("saving the CRF features completed.\n\n");

        fclose(optf);
        fclose(itrndataf);
        fclose(otrnlogf);
        fclose(omodelf);
    }

    // testing only
    if (is_tst) {
        printf("reading context predicate map from file ...\n");
        dt.read_cp_map(imodelf);
        printf("reading %d context predicate mappings completed.\n\n",
opt.num_cps);

        printf("reading label map from file ...\n");
        dt.read_lb_map(imodelf);
        printf("reading %d label mappings completed.\n\n", opt.num_labels);

        if (opt.order == SECOND_ORDER) {
          printf("reading second-order label map ...\n");
          dt.read_2order_lb_map(imodelf);
          printf("reading second-order label map completed.\n\n");

          printf("reading second-order to first-order label map ...\n");
          dt.read_2to1_lb_map(imodelf);
          printf("reading second-order to first-order label map completed.\n\n");
        }

        printf("reading dictionary from file...\n");
        dict.read_dict(imodelf);
        printf("reading %d context predicates completed.\n\n", opt.num_cps);

        printf("reading CRF features from file ...\n");
        fgen.read_features(imodelf);
        printf("reading %d CRF features completed.\n\n", opt.num_features);

        printf("reading testing data ...\n");
        dt.read_tstdata(itstdataf);
        printf("reading %d testing sequences completed.\n\n", opt.num_tstseqs);

        // create the viterbi object
        viterbi vtb;
        // create the evaluation object
        evaluation eval;

        // create the CRF model
        model mdl(&opt, &dt, &dict, &fgen, NULL, &vtb, &eval);

        // start to label for testing data
        printf("labeling for testing data ...\n");
        if (opt.nbest <= 1) {
          mdl.apply_tstdata();
        } else {
          mdl.apply_tstdata(opt.nbest);
        }
        printf("labeling for testing data completed.\n\n");

        // saving testing output to file
        FILE * otstdataf;
        otstdataf = fopen((opt.model_dir + opt.tstdata_file + ".model").c_str(), "w");
        printf("writing testing output to file ...\n");
        dt.write_tstdata(otstdataf);
        printf("writing testing output completed.\n\n");

        // performance evaluation on testing dataset
        opt.is_logging = 0;
        mdl.peval->evaluate(NULL);
        printf("\n");

        fclose(optf);
        fclose(itstdataf);
        fclose(otstdataf);
        fclose(imodelf);
    }

    // prediction
    if (is_prd) {
        printf("reading context predicate map from file ...\n");
        dt.read_cp_map(imodelf);
        printf("reading %d context predicate mappings completed.\n\n",
opt.num_cps);

        printf("reading label map from file ...\n");
        dt.read_lb_map(imodelf);
        printf("reading %d label mappings completed.\n\n", opt.num_labels);

        if (opt.order == SECOND_ORDER) {
          printf("reading second-order label map ...\n");
          dt.read_2order_lb_map(imodelf);
          printf("reading second-order label map completed.\n\n");

          printf("reading second-order to first-order label map ...\n");
          dt.read_2to1_lb_map(imodelf);
          printf("reading second-order to first-order label map completed.\n\n");
        }
        printf("reading dictionary from file...\n");
        dict.read_dict(imodelf);
        printf("reading %d context predicates completed.\n\n", opt.num_cps);

        printf("reading CRF features from file ...\n");
        fgen.read_features(imodelf);
        printf("reading %d CRF features completed.\n\n", opt.num_features);

        printf("reading unlabeled data ...\n");
        dt.read_ulbdata(iulbdataf);
        printf("reading %d unlabeled sequences completed.\n\n", opt.num_ulbseqs);

        // create the viterbi object
        viterbi vtb;

        // create the CRF model
        model mdl(&opt, &dt, &dict, &fgen, NULL, &vtb, NULL);

        // start to label for testing data
        printf("predicting for unlabeled data ...\n");
        mdl.apply_ulbdata();
        printf("predicting for unlabeled data completed.\n\n");

        // saving data output to file
        FILE * oulbdataf;
        oulbdataf = fopen((opt.model_dir + opt.ulbdata_file + ".model").c_str(), "w");
        printf("writing data with predicted labels to file ...\n");
        dt.write_ulbdata(oulbdataf);
        printf("writing data with predicted labels completed.\n\n");

        fclose(optf);
        fclose(iulbdataf);
        fclose(oulbdataf);
        fclose(imodelf);
    }

    return 0;
} // end of main function
```