Mismatch kernel con componenti pesate per la classificazione di proteine

(Mismatch kernels with weighted components for protein classification)

Autore: Daniele Barilaro

Relatore: Dott.ssa Cinzia Pizzi

Università degli Studi di Padova- Dipartimento di Ingegneria dell'Informazione

Corso di Laurea Magistrale in Ingegneria Informatica

Abstract

Nella tesi si è analizzato il problema della classificazione delle sequenze di proteine mediante machine a support vettoriale, utilizzando gli string kernel. Essi sono delle funzioni su due stringhe che valutano la similarità tra esse. Sono stati sviluppati l'Approximate Mismatch Kernel e il kernel combinatorio con ordine. Si è implementato un algoritmo che permette il calcolo di questi kernel in tempo lineare ammortizzato sul numero di tuple distinte nella stringa di lunghezza maggiore. Inoltre la complessità temporale è indipendente dai parametri dei kernel, cioè dalla lunghezza delle tuple e dal numero di mismatch ammessi. Il kernel combinatorio con ordine è stato, tra I kernel studiati, quello che ha presentato prestazioni migliori. L'obiettivo è stato quello di studiare le prestazioni della classificazione mediante tale kernel al variare dei parametri di lunghezza delle tuple e di numero di mismatch ammessi. Le prestazioni dei kernel sono state valutate sul problema del rilevamento dell'omologia remota. Tale kernel è stato analizzato per i valori di mismatch k=1,2,3 variando per ciascuno di essi il parametro m al fine di capirne l'andamento della bontà di classificazione.

Sommario

Introduzione

In questa tesi è stato studiato il problema della classificazione delle sequenze di proteine con metodi alignment free. I metodi di confronto classici tra sequenze di proteine, tuttora ampiamente utilizzati, sono basati sugli allineamenti. Tuttavia, nell'ultimo Decennio sono state sviluppate tecniche di confronto alternative basate su metodi non vincolati dall'allineamento delle sequenze chiamati pertanto aligment-free. In essi le sequenze sono rappresentate come vettori le cui caratteristiche sono funzione del numero di tuple che in essi occorrono.

Gli approcci storici per classificare le proteine sono di tipo generativo, cioè per il tipo di classificazione che si vuole effettuare viene creato un modello. Ad esempio nella classificazione di una sequenza proteica come appartenente o meno a una data famiglia, si costruisce prima un modello che descrive tutte le sequenze appartenenti a tale famiglia, e poi si verifica se la sequenza si adatta a tale modello definendo un certo punteggio di soglia oltre il quale essa si accetta come appartenente.

Negli approci discriminativi invece le descrizioni delle sequenze di proteine, ad esempio tramite i vettori dei metodi aligment free, vanno a costituire gli esempi per i dataset di un classificatore. In tale approccio sia gli esempi positivi che negativi sono utilizzati.

Nella tesi sono stati studiati i metodi di confronto aligment free basati su string kernel per il loro utilizzo col classificatore Support Vector Machine (SVM).

Il primo string kernel introdotto fu lo Spectrum Kernel in [1], e poi come evoluzione di questo fu introdotto il Mismatch Kernel in [2].

All'epoca della presentazione di tali kernel, due grandi vincoli ne hanno limitato lo studio. Il primo è dovuto all'hardware disponibile all'epoca, se, infatti, oggi un singolo processore può vantare potenze di calcolo da 140-200 GFLOPS, all'epoca un singolo processore non superava i 7 GFLOPS [3]. Il secondo vincolo è dovuto al fatto che il mismatch kernel, implementato con una struttura dati di tipo mismatch tree, è di veloce calcolo solo per bassi valori dei parametri m=lunghezza delle tuple e k=numero di mismatch.

Nel contesto di questa tesi è stato sviluppato un nuovo kernel, l'Approximate Mismatch Kernel. L'algoritmo usato per tale kernel implementa in maniera efficiente l'algoritmo descritto in [4]. Grazie a tale algoritmo il valore del kernel è calcolato in tempo lineare ammortizzato sul numero di tuple distinte di una stringa, indipendentemente dai parametri m e k. Ciò costituisce un grandissimo vantaggio rispetto al mismatch kernel, di cui è un'approssimazione, in quanto permette di effettuare un vasto numero di test per qualunque combinazione di parametri m, k.

Successivamente tale kernel è stato generalizzato, introducendo teoricamente il «Mismatch Kernel approssimato pesato con funzione sul numero di mismatch». Esso dà dei pesi ai contributi dati al kernel da diversi valori di mismatch e inoltre sostituisce il prodotto tra il numero di mismatch di una tupla nelle due stringhe in ingresso con una loro funzione.

Inoltre si è proposto un nuovo Approximate Mismatch Kernel in cui s'introduce una funzione sul numero di mismatch. Esso è chiamato «Kernel Combinatorio con ordine» e introduce in un metodo aligment-free il concetto di allineamento, limitato alle tuple, abbinato al classico calcolo della frequenza.

Questo kernel, molto più scalabile dell'originale Mismatch kernel, ha permesso di compiere uno studio più approfondito delle dinamiche della classificazione di proteine, variando lunghezza e numero di mismatch in un intervallo molto più ampio che in precedenza.

A questo scopo si è sviluppato il software per eseguire test con la versione di SCOP 1.53 e 1.75B, l'ultima rilasciata al momento della stesura di tale tesi.

Per la valutazione dei test si è utilizzata la curva operativa caratteristica ricevente, adatta alla valutazione delle prestazioni di un classificatore con dataset sbilanciati, come nel caso del rilevamento delle omologie remote.

Sono stati effettuati test per valori di mismatch k=1,2,3.

Le prestazioni migliori per tale kernel si sono ottenute per i parametri (10,2), (11,3) e (12,3). In particolare si è visto che per (11,3) il classificatore rileva correttamente un buon numero di esempi positivi senza compromettere eccessivamente il rilevamento degli esempi negativi.

Nel Capitolo 1 si introducono concetti fondamentali riguardanti proteine e classificazione. Nel Capitolo 2 si introduce il problema specifico della classificazione delle proteine. Nel Capitolo 3 si descrivono il database SCOP e la raccolta ASTRAL. Nel Capitolo 4 vengono introdotti gli string kernels e i nuovi kernel definiti. Nel Capitolo 5 si forniscono i dettagli dell'implementazione dell'Approximate Mismatch Kernel. Nel Capitolo 6 si descrive il software realizzato per effettuare i test. Nel Capitolo 7 sono riportati i risultati dei test effettuati. Infine nel Capitolo 8 vi sono le conclusioni.

1 Concetti fondamentali

In questo capitolo sono definiti i concetti fondamentali necessari a comprendere la classificazione delle proteine mediante macchine a supporto vettoriale. Nel seguito vengono prima brevemente introdotte le proteine e in seguito la classificazione mediante macchine a supporto vettoriale focalizzando l'attenzione sulla classificazione nel caso di data set sbilanciati.

1.1 Le proteine

Le proteine sono macromolecole polimeriche, e sono i costituenti fondamentali di ogni essere vivente. Costituiscono ad esempio i tessuti degli organi, le fibre muscolari, i neurotrasmettitori ecc. Esse sono costituite da lunghe catene di amminoacidi, molecole composte da un gruppo amminico(basico), un gruppo carbossilico(acido) e un residuo. Tale residuo è variabile in base al tipo di amminoacido e da a questo specifiche proprietà. Le proteine vengono chiamate anche catene polipeptidiche in quanto sono costituite dalla ripetizione lineare di legami polipeptidici che si formano dall'unione tra loro dei gruppi acidi e di quelli basici degli amminoacidi. In Figura 1-1 si può vedere in alto la struttura di un amminoacido, con a sinistra il gruppo amminico, a destra il gruppo carbossilico legati dal carbonio alfa associato ad un atomo di idrogeno e al residuo variabile, indicato con R. La seconda parte dell'immagine illustra il legame tra due amminoacidi.

Tra i residui possono esserci delle iterazioni chimiche. Residui appartenenti ad aminoacidi distanti tra loro possono interagire tramite legami deboli, portando a un ripiegamento (spesso definito col termine inglese di *folding*) della catena lineare. Queste iterazioni sono alla base della struttura 3D delle proteine. Esistono 20 diversi tipi di amminoacidi. Questi hanno struttura e proprietà chimiche diverse. Possono essere idrofobici (aromatici e alifatici), polari (negativi e positivi), acidi, basici, aromatici, piccoli [5]. Le iterazioni tra gli amminoacidi dipendono dalle loro caratteristiche chimiche. Ad esempio un amminoacido polare positivo e uno negativo tendono ad interagire tra loro, mentre uno polare e uno idrofobico no [5].

Gli amminoacidi sono identificati mediante 20 lettere maiuscole dell'alfabeto. In tabella Tabella 1-1 sono riportati i nomi degli amminoacidi, le loro lettere che li indentificano, e il loro tipo di gruppo laterale, mentre in Figura 1-2 è riportata la loro classificazione chimica sotto forma di insiemi.

Le proteine sono codificate grazie all'informazione contenuta nell'RNA messaggero(mRNA), che a sua volta contiene una trascrizione di sezioni di DNA. Nel passaggio da RNA a proteina il tipo di informazione cambia notevolmente. Si passa da un alfabeto chimico costituito da 4 nucleotidi ad un alfabeto basato su 20 amminoacidi. Data l'impossibilità di mantenere una corrispondenza univoca tra nucleotidi e amminoacidi è necessaria una codifica, chiamata codice genetico. Ogni gruppo di tre nucleotidi sequenziali codifica un amminoacido. Tali triplette sul mRNA sono chiamate *codoni*. Dato che le possibili combinazioni di tre nucleotidi sono sessantaquattro, e gli amminoacidi sono venti, si ha una ridondanza informativa. Il fenomeno per il quale esistono diverse triplette che codificano lo stesso amminoacido è chiamato "degenerazione del codice". Ciò implica che da una sequenza di amminoacidi non è possibile ottenere in modo univoco la sequenza di nucleotidi che l'ha generata.

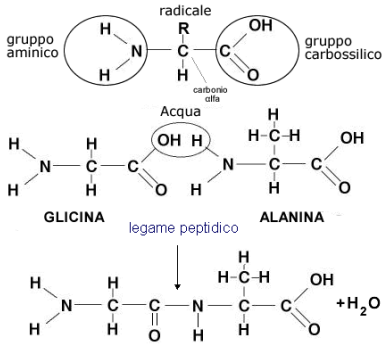


Figura 1-1: Struttura degli amminoacidi e legame peptidico

simbolo		nome	tipo di *R*
A	Ala	Alanina	idrofobo
C	Cys	Cisteina	idrofilo
D	Asp	Acido aspartico	acido
E	Glu	Acido glutammico	acido
F	Phe	Fenilalanina	idrofobo aromatico
G	Gly	Glicina	idrofobo
H	His	Istidina	basico
I	Ile	Isoleucina	idrofobo
K	Lys	Lisina	basico
L	Leu	Leucina	idrofobo
M	Met	Metionina	idrofobo
N	Asn	Asparagina	idrofilo
P	Pro	Prolina	idrofobo
Q	Gln	Glutammina	idrofilo
R	Arg	Arginina	basico
S	Ser	Serina	idrofilo
T	Thr	Treonina	idrofilo
V	Val	Valina	idrofobo
W	Trp	Triptofano	idrofobo aromatico
Y	Tyr	Tirosina	idrofilo aromatico

Tabella 1-1: amminoacidi e loro lettere identificative

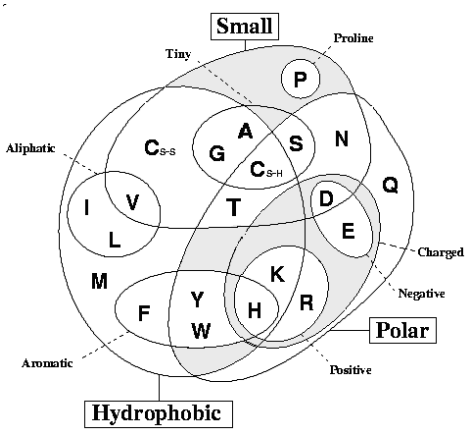


Figura 1-2: Classificazione degli amminoacidi

Le proteine sono costituite da lunghe catene lineari di amminoacidi. Tali catene delle proteine vengono definite *struttura primaria*. Gli amminoacidi di una proteina possono interagire tra loro a seconda delle loro proprietà chimiche dando luogo a ripiegamenti della struttura primaria nello spazio. La forma di una proteina ne influenza poi la sua funzione. Una proteina è costituita da quattro livelli strutturali [5].

- Struttura primaria: è la catena lineare di amminoacidi
- Struttura secondaria: è costituita dalla formazione di ripiegamenti locali regolari. I due schemi più comuni sono α-elica e foglietto β pieghettato.
- Struttura super secondaria: le strutture secondarie formano motivi strutturali legandosi tra di loro. Le più comuni sono α elica-loop- α elica costitutita da un ripiegamento di due α-elica , β turn costituita da un ripiegamento di due foglietti β e la chiave greca b-a-b costituito da due filamenti b paralleli, intercalati da un a-elica.
- Struttura terziaria: è data dalla formazione di una struttura tridimensionale definitiva dovuta all'iterazione di più strutture super-secondarie.
- Struttura quaternaria: è data dall'iterazione tra diversi polipetidi (ripiegati nella loro struttura 3D) per la formazione di un complesso proteico completo.

1.1.1 Predizione della struttura delle proteine

Alcune sequenze hanno motivi di amminoacidi che formano sempre una struttura caratteristica. Predizioni di tali strutture dalle sequenze sono in parte possibili usando metodi disponibili di allineamento sequenza-per-sequenza. Infatti, a volte, una similarità tra sequenze implica una similarità strutturale [6].

Uno dei metodi usati che fornisce un link diretto tra le sequenze e la struttura sono le scoring matrix (profili e PSSM) e i profili HMM(Hidden Markov Models). Se una delle sequenze rappresentata dal profilo ha una struttura conosciuta, allora ogni altra sequenza che ha un buon match col profilo avrà la stessa struttura. I profili possono essere usati anche per rilevare l'appartenenza di una sequenza ad una famiglia, generando un profilo per ogni famiglia di proteine. Più è forte la similarità e l'identità di due sequenze e più è probabile che sezioni di strutture tridimensionali delle sequenze siano simili. L'accuratezza di questi metodi in genere non supera il 75% [6].

Esistono anche metodi per effettuare l'allineamento struttura-per-struttura [6]. In essi le posizioni sequenziali degli atomi di carbonio per ogni amminoacido nelle due sequenze sono confrontate per determinare se le catene degli atomi tracciano la stessa posizione nello spazio. Database di elementi strutturali sono stati creati per tali metodi.

Protein Data Bank (PDB) è un vasto database contente le strutture 3D delle proteine ed acidi nucleici ottenute con vari metodi, i principali sono la diffrazione X-Ray, la spettroscopia a risonanza magnetica nucleare di proteine e la microsopia Cryo-electron. Ad oggi vi sono più di 90000 strutture. Tale database è utilizzato come base di partenza per molti database strutturali usati per la predizione della struttura delle proteine. Essi classificano in vario modo la struttura delle proteine basandosi sul loro confronto e allineamento. Il tipo, l'ordine e la posizione relativa delle strutture secondarie sono confrontate usando le coordinate atomiche conosciute in ciascuna struttura.

1.1.2 Classificazione delle proteine

Le proteine possono essere classificate sia in base alla loro struttura che alla loro similarità sequenziale [6]. Quest'ultimo approccio prevede l'allineamento delle sequenze usando uno dei metodi classici conosciuti. Nella classificazione delle proteine ci sono alcune cose cruciali da tenere in considerazione. Per prima cosa, sequenze di proteine provenienti da origini evolutive differenti possono assumere strutture simili. Inoltre, le sequenze con la stessa origine strutturale possono essersi differenziate nel tempo

considerevolmente, in differenti specie, ma mantenendo la stessa struttura funzionale [6]. Riconoscere perciò una qualche similarità in questo caso può essere un compito molto difficile. Si hanno altresì casi in cui due proteine che condividono un alto grado di similarità sequenziale, avendo la stessa origine evolutiva, hanno funzionalità diverse. Questo è dovuto a duplicazioni e ri-arrangiamenti genetici [6].

1.2 Classificazione e macchine a supporto vettoriale

L'apprendimento automatico è un settore dell'Intelligenza Artificiale che si occupa di derivare nuova conoscenza da osservazioni e dati noti. Formalmente è definita come la capacità di un agente, un'entità capace di percepire l'ambiente tramite sensori e di eseguire azioni tramite attuatori, di sintetizzare nuova conoscenza dall'osservazione del suo ambiente circostante e dall'esperienza maturata. Si dice che tale agente è dotato di stato dinamico. L'apprendimento automatico si può suddividere in tre tipologie: l'apprendimento non supervisionato, l'apprendimento supervisionato e l'apprendimento per rinforzo [7].

Nell'apprendimento supervisionato un agente impara una funzione dell'input a partire da esempi di coppie di input-output. In Figura 1-3 si mostra il concetto base. Da un insieme di coppie di ingresso (X,Y), dove X,Y possono essere vettori di valori, si cerca di approssimare la funzione che lega i valori di X ad Y [7].

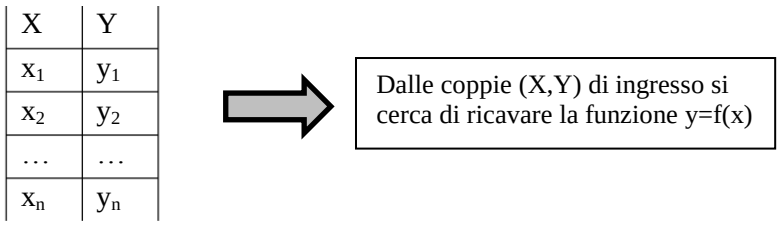


Figura 1-3: Apprendimento supervisionato

Esso si può suddividere in due tipologie: i) la classificazione che prevede un insieme di valori di output discreti e limitati e, ii) la regressione, nella quale l'output è continuo. Un esempio di classificazione è dato dal riconoscimento delle cifre numeriche scritte a mano, dove l'insieme di valori di ingresso sono tutte le possibili immagini di cifre scritte a mano e l'insieme di valori di output sono tutte le possibili cifre, che costituiscono un codominio discreto e limitato (tutte le cifre da 0 a 9).

Nell'apprendimento non supervisionato un agente impara a riconoscere pattern o schemi nell'input senza alcuna indicazione dei valori di output. Esempi di tali tecniche sono il clustering, l'analisi delle componenti principali, l'analisi del cestino di mercato, l'analisi delle componenti indipendenti e la decomposizione in valori singolari. Ad esempio, il *clustering* è un insieme di tecniche volte alla selezione e al raggruppamento di elementi omogenei in un insieme di dati. In Figura 1-4 è un esempio d'immagine su cui effettuare il clustering (k-means clustering) per individuare le zone di colore principali.

Figura 1-4: Immagine su cui effettuare il clustering

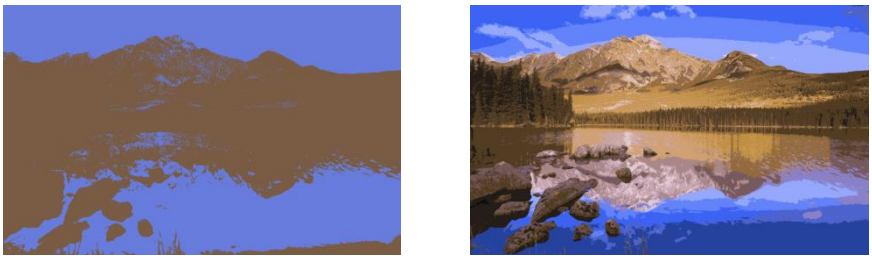


Figura 1-5: A sinistra l'immagine con 2 cluster, a destra l'immagine con 16 cluster

Nella Figura 1-5 vi è la stessa immagine su cui è stato effettuato il clustering dei colori per due numeri differenti di cluster.

L'apprendimento per rinforzo prevede invece che l'agente apprenda esplorando l'ambiente e ricevendo feedback positivi in caso di azioni positive.

1.2.1 Il problema della classificazione

Si consideri una generica dipendenza funzionale g: X → Y ed un insieme di coppie di valori (chiamato training set) $T = \{ (x^i,y^i) \mid x^i \in X, y^i \in Y, i = 1,\ldots,n\}$, con X insieme di input e Y insieme di output. L'obiettivo è ottenere una buona stima Ω di g. Lo spazio di input viene suddiviso in k sottoinsiemi $X_1,X_2,..,X_k \in X$ tali che $X_i \cap X_j = \emptyset \; \forall \; i,j = 1,\ldots,k \mid i \neq j$ con k pari alla dimensione dello spazio di output $Y = \{1,\ldots,k\}$. L'obiettivo della classificazione è assegnare ciascun valore di input x al sottoinsieme a cui appartiene. Il problema più semplice di classificazione è la classificazione binaria dove lo spazio di input viene diviso in due insiemi $X_1,X_2 \in X$. Le reti neurali a percettroni e le Support Vector Machine sono esempi di classificatori.

La procedura di apprendimento si divide in due fasi. Nella prima fase, detta di *training* viene fornito al sistema un un sottoinsieme di coppie x-y di T. Il sistema cerca di creare un modello che descriva la dipendenza che intercorre tra l'input e l'output. Nela seconda fase, detta di *testing*, dove al sistema viene fornito un altro sottoinsieme noto di valori di input x per i quali si conosce il rispettivo output e successivamente si valuta l'accuratezza del sistema in termini di percentuale di risposte corrette. Per evitare di introdurre bias con una particolare suddivisione del dataset T si adotta una procedura chiamata k-fold cross validation dove il dataset viene suddiviso in K sottoinsiemi, e si allena di volta in volta il sistema su K-1 sottoinsiemi e lo si testa sul sottoinsieme rimanente, interando la procedura per K volte. Successivamente si prende la media dei risultati sulla bontà di classificazione.

La stima Ω di g deve avere buone capacità di generalizzazione. Una macchina per l'apprendimento generalizza bene quando ha la capacità di calcolare correttamente l'output per dati di input non inclusi nel training set. La generalizzazione è strettamente connessa con la complessità di un classificatore. Se la funzione Ω è eccessivamente complessa si rischia non avere una buona approssimazione di g sul test set, fenomeno conosciuto come overfitting. In questo caso il modello Ω è adattato troppo ai dati di training. Se Ω invece è troppo semplice, si ha una scarsa approssimazione di g sul training set, fenomeno noto come underfitting. L'ideale è trovare un classificatore con complessità ottimale di Ω, cioè il modello più semplice che fornisce buone performance sui dati di training.

Per evitare di adattare troppo il modello ai dati in alcuni metodi si utilizza una procedura chiamata early-stopping, nella quale si suddivide il training set tenendo da parte un insieme di validazione chiamato validation set. Durante l'allenamento del sistema di

verifica ogni volta l'accuratezza del validation set. Se l'errore aumenta (e quindi aumenta l'overfitting), si arresta l'allenamento.

1.2.2 Support Vector Machines (SVM)

Le SVM sono un efficace strumento per la classificazione e la regressione. Nel corso degli anni hanno dato risultati considerevoli in molte applicazioni, ad esempio nel riconoscimento del testo scritto a mano, nell'elaborazione del linguaggio naturale e nell'identificazione di immagini. Sono state introdotte da Vapnik nel 1990 . Nel seguito si introdurranno i concetti base dietro alle SVM, rimandando a letteratura specifica per una trattazione più rigorosa ed approfondimenti [8](pag. 256-276). L'SVM trattata nel seguito risolve il problema della classificazione binaria.

Si consideri un insieme di dati contenente esempi che appartengono a due differenti classi, rappresentabile su un piano bidimensionale. Tale insieme è anche linearmente separabile, cioè si può trovare un iperpiano tale che tutti gli elementi di una classe siano in un lato di esso, e gli elementi dell'altra classe siano sull'altro lato. In generale si possono trovare infiniti iperpiani che dividono il data set in tale modo.

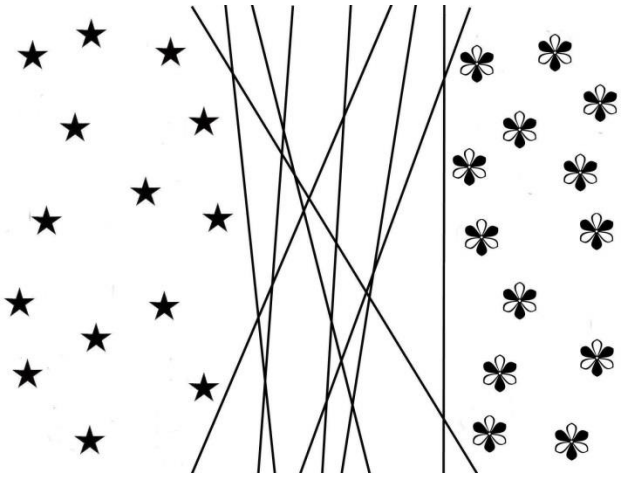


Figura 1-6: esempi di possibili iperpiani che dividono un dataset

In Figura 1-6 vi è un esempio di tali iperpiani. Il classificatore deve scegliere uno di tali iperpiani per rappresentare il suo limite di decisione B, basandosi su quanto bene si aspetta che funzioni sul test set. Differenti scelte dell'iperpiano influenzano l'errore di generalizzazione. In Figura 1-7 si possono vedere due limiti di decisione B1 e B2. Entrambi separano le classi senza errori di classificazione.

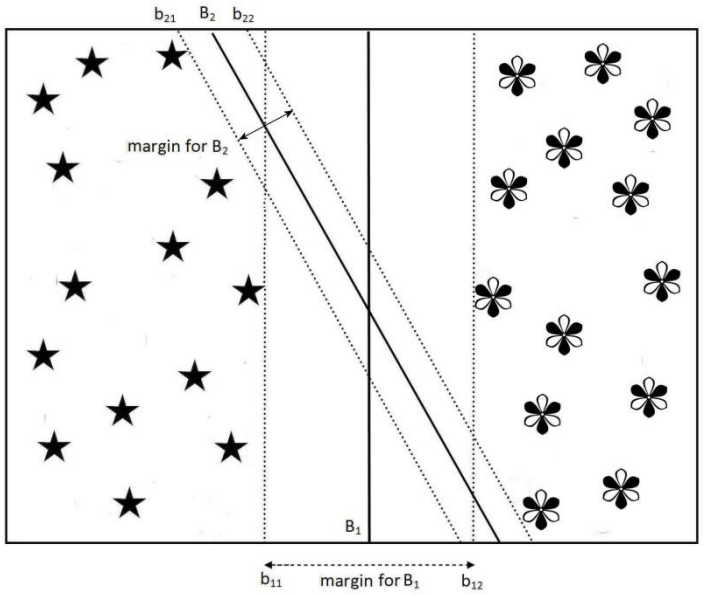


Figura 1-7: Esempi di due limiti di decisione B1 e B2

L'SVM associa ad ogni limite di decisione due iperpiani, chiamati b1 e b2, che separano in modo più preciso gli elementi delle due classi del data set. La distanza tra b1 e b2 è chiamata **margine del classificatore**. L'obbiettivo del classificatore è quello di trovare i limiti di decisione con il **massimo** margine. Essi infatti hanno un miglior errore di generalizzazione rispetto a quelli con piccolo margine. Se il margine è piccolo ogni leggera perturbazione al limite di decisione può avere influenza significativa sulla classificazione e rende il classificatore più sensibile all'overfitting, generalizzando male.

Un classificatore SVM lineare è un classificatore che cerca l'iperpiano col margine più elevato, ed è quindi conosciuto anche come classificatore a massimo margine.

Nella maggior parte dei casi il dataset è costituito da dati non perfettamente linearmente separabili. E' quindi necessario modificare l'SVM per l'apprendimento di limiti di decisione che sono tollerabili a piccoli errori di training. Ciò è effettuato con un metodo chiamato approccio soft margin, che permette di costruire limiti di decisione lineari anche in situazioni in cui i dati non sono linearmente separabili.

Le SVM sono applicabili anche a insiemi di dati che non hanno limiti di decisione lineari. Ciò è in primo luogo fattibile trasformando i dati dal loro spazio di coordinate originale X in un nuovo spazio dimensionalmente superiore, chiamato feature space, in modo tale che il limite di decisione lineare possa separare i dati nel nuovo spazio trasformato. Una trasformazione Φ: X $\rightarrow$ R^N è necessaria per mappare i dati dallo spazio delle features originale in un nuovo spazio dove il limite di decisione è lineare. In Figura 1-8 vi è un esempio di dati non linearmente separabili, mentre in Figura 1-9 vi è lo stesso insieme di dati in uno spazio trasformato per l'applicazione di un limite di decisione lineare.

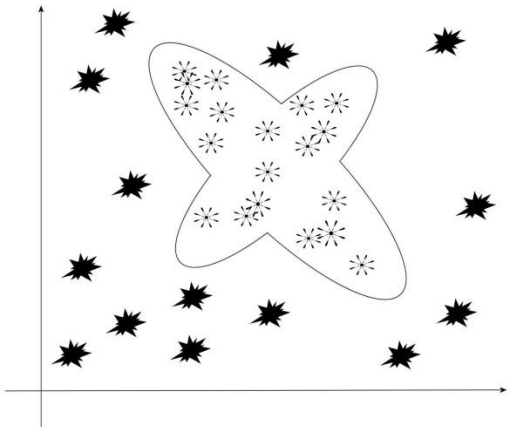


Figura 1-8:Dati non linearmente separabili e confine di decisione (linea continua)

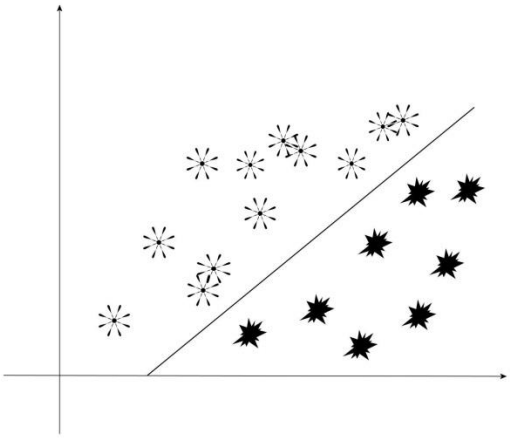


Figura 1-9: Stessi dati della figura precedente in uno spazio trasformato e limite di decisione lineare

Un problema è che quest' approccio soffre del problema della dimensionalità dovuto ad un incremento, che può potenzialmente rilevarsi anche molto elevato, delle dimensioni dello spazio considerato. Inoltre, nella maggior parte delle volte, non è chiaro quale funzione debba essere usata per assicurarsi che un limite di decisione lineare possa esser costruito nello spazio trasformato, e a volte tale funzione non è nemmeno possibile da ricavare.

Nelle SVM il problema è stato risolto col "trucco" del kernel, un metodo per calcolare la ***similarità*** di due dati nello spazio trasformato usando l'insieme di attributi originali. Si costruisce una funzione, chiamata funzione di kernel, che restituisce un valore di similarità tra due attributi nello spazio originale.

1.2.3 Minimizzazione del rischio strutturale

L'SVM implementa il principio della minimizzazione del rischio strutturale.

Vapnik nella sua trattazione ritiene che la flessibilità di un classificatore non deve essere caratterizzata dal numero di parametri, ma solo dalle capacità del classificatore. Ha formalizzato il concetto con la dimensione "VC" di un classificatore. Si consideri un classificatore lineare in uno spazio bidimensionale a due classi. Se si hanno tre dati di training, essi possono sempre essere classificati perfettamente indipendentemente da come vengono etichettati (l'etichetta è la classe). Infatti è sempre possibile trovare un iperpiano che li divide. Ma se si hanno quattro punti, si può trovare un modo di etichettare i punti in modo tale che il classificatore fallisca. In questo caso la dimensione-VC nello spazio bidimensionale è 3. Più alta è tale dimensione e più flessibile risulta il classificatore. Tale dimensione ad ogni modo è un concetto teorico e nella maggior parte dei classificatori è difficile da calcolare. In generale si cerca di trovare un classificatore che minimizza l'errore di training e un termine che è funzione della flessibilità del classificatore, considerata come complessità del modello. Si introduce a tale scopo l'intervallo di confidenza per l'errore di generalizzazione. Più flessibile è un classificatore, e più ampio è l'intervallo di confidenza. L'ampiezza può essere limitata superiormente da una funzione della dimensione-VC del classificatore, preferendo i classificatori con basso limite superiore all'errore di generalizzazione e con ampio intervallo di confidenza.

1.2.4 Il problema dello squilibrio di classe

Gli insiemi di dati con classi sbilanciate sono molto frequenti in applicazioni reali.

Nello sbilanciamento di classe c'è un numero sproporzionato di elementi che appartengono alle differenti classi. La misura di accuratezza, che è usata per comparare le performance di un classificatore, tratta ogni classe come ugualmente importante e non è adeguata per valutare modelli derivati da data set sbilanciati.

Dato un insieme di dati appartenenti a due possibili classi, tale insieme è a classi sbilanciate se gli elementi appartenenti ad una di esse sono molto maggiori di quelli appartenenti alla classe complementare. A esempio nella produzione di valvole di sicurezza per il gas, le valvole difettose prodotte in genere sono molto poche rispetto a quelle a norma. Nonostante le occorrenze infrequenti, una classificazione corretta della classe rara ha molta più importanza della classificazione della classe in maggioranza. Nel nostro esempio se lo 0,01% delle valvole è difettose, un modello per un analizzatore di conformità che rileva ogni valvola come a norma ha un'accuratezza del 99,99%, ma fallisce a rilevare un qualunque difetto.

Per la classificazione binaria nei data set sbilanciati, la classe rara è di solito denotata come classe positiva. La matrice di confusione riassume il numero di esempio predetti correttamente o incorrettamente da un modello di classificazione. Essa è illustrata in Tabella 1-1.

		Classe predetta	
		+	-
Classe reale	+	TP	FN
	-	FP	TN

Tabella 1-2: Matrice di confusione

TP, acronimo di True Positive, è il numero di esempi positivi predetti correttamente dal classificatore. TN (True Negative), è il numero di esempi negativi predetti correttamente. FN (False Negative) è il numero di esempi positivi predetti erroneamente come negativi e FP (False Positive) è il numero di esempi negativi predetti erroneamente come positivi.

Si definisce **frequenza dei veri positivi** (TPR: True Positive Rate) la frazione di esempi positivi predetti correttamente:

$$TPR = TP/(TP + FN)$$

Si definisce **frequenza dei falsi positivi** (FPR: False Positive Rate) la frazione di esempi negativi predetti erroneamente:

$$FPR = FP/(TN + FP)$$

Si definisce **precisione** la frazione di esempi predetti come positivi che risultano esserlo veramente:

$$p = TP / (TP + FP)$$

Si definisce **recall** la frazione di esempi positivi predetti correttamente, ed è equivalente alla misura TPR.

Un buon classificatore ha valori di precisione e recall elevati. Infatti se la precisione è pari ad 1, significa che tutti gli esempi negativi sono stati classificati correttamente, e se il recall è a 1 che tutti gli esempi positivi sono stati classificati correttamente. In genere è possibile costruire modelli che massimizzano una misura ma non l'altra.

1.2.5 La curva operativa caratteristica ricevente

La curva operativa caratteristica ricevente (in inglese Receiver Operating Characteristic - ROC) è un approccio grafico per la visualizzazione del compromesso tra la *frequenza dei veri positivi* e quella dei *falsi positivi* di un classificatore. TPR è visualizzata lungo l'asse delle ordinate e FPR lungo le ascisse. Ogni punto lungo la curva corrisponde a uno dei modelli prodotti dal classificatore. Ci sono tre punti significativi nel grafico:

- (TPR=0, FPR=0): corrisponde a un modello che predice ogni esempio come negativo
- (TPR=1, FPR=1): corrisponde a un modello che predice ogni esempio come positivo
- (TPR=1, FPR=0): modello perfetto che predice ogni esempio correttamente

Un buon modello deve esser il più vicino possibile al terzo punto, cioè deve esser localizzato nell'angolo in alto a sinistra del grafico, mentre un modello che fa previsioni casuali risiede lungo la diagonale.

La curva ROC serve per comparare visualmente le performance relative tra classificatori. Invece l'area sotto la curva fornisce un comodo metodo di valutazione numerico. Se il modello è perfetto, allora l'area sotto la curva è pari ad 1, mentre se il modello fa previsioni casuali, l'area sotto la curva è pari a 0.5.

Come generare la curva ROC

Per generare la curva il classificatore deve produrre un output a valori continui che può essere usato per valutare le sue predizioni, dal record più probabile ad esser classificato positivamente a quello meno probabile. Ciò ad esempio è facilmente ottenibile nel caso delle SVM in quanto esse producono in output per ogni esempio di ingresso il valore della funzione obbiettivo [8](pag. 256-276). La seguente procedura, tratta da [8] (pag. 294-301) , può essere utilizzata per la generazione:

1. Si ordinano gli esempi di test in ordine crescente dei loro valori di valutazione in output, creando una lista ordinata.
2. Si seleziona l'esempio di test con la valutazione minore. Si assegna tale record e quelli sotto di esso alla classe positiva, cioè si classificano tutti i record di test come positivi. Ciò significa che in questa situazione si ha TPR = FPR = 1
3. Si seleziona il successivo record di test dalla lista ordinata. Si classifica il record selezionato e quelli sotto ad esso come positivi, mentre quelli con valutazione inferiore come negativi. Si aggiornano i valori TP, FP, TN, FN esaminando la reale classe degli esempi precedenti. Se l'esempio precedentemente selezionato è positivo, TP è decrementato di un'unità, FP rimane invariato e FN incrementato di un'unità. Se l'esempio precedente è negativo, FP è decrementato di un'unità e TN incrementato di un'unità.
4. Si ripete il passo precedente fino a quando l'esempio con voto più elevato è selezionato.
5. Si traccia TRP rispetto al FPR e si calcola l'area della curva generata.

2 Stato dell'arte sulla classificazione delle proteine

Inizialmente gli algoritmi di analisi delle sequenze sono stati sviluppati dalle tecniche di string matching e in seguito sono stati raffinati con concetti della statistica computazionale (quali Hidden Markov Model e teoria Bayesana). La nozione di similarità si è fino a un decennio fa basata quasi solamente su metodi per l'allineamento (locale o globale) delle sequenze. Tutti questi metodi hanno il pregiudizio di vedere le molecole come sequenze lineari simili, nonostante la loro natura fisica 3D e la natura dinamica dell'evoluzione molecolare. La dinamica evolutiva è dovuta a fenomeni su piccola scala quali mutazioni, inserzioni e delezioni su singoli nucleotidi, fenomeni su media scala quali la comparsa e scomparsa di introni e fenomeni su larga scala quali il ri-arrangiamento della sequenza genomica e duplicazioni di intere sequenze. Nonostante ciò, assumere la conservazione della contiguità negli elementi delle sequenze permette l'impiego di procedure computazionali ben sviluppate ed efficienti.

Di seguito si illustrano i principali metodi alignment-based e alignment-free [10].

2.1 Metodi alignment based

Lo sviluppo di metodi di sequenziamento delle proteine, a partire dal 1951, ha portato i biologi alla necessità di sviluppare atlanti di sequenze di proteine già dal 1960. Al National Biomedical Research Foundation (NBRF) è stata sviluppato il primo database informatico di tali sequenze nel 1984, noto come Protein Information Resource. Tale database era già allora organizzato in famiglie e superfamiglie di proteine in base al grado di similarità tra le sequenze.

Grazie all'avvento di Internet sono state in seguito sviluppati siti web per permettere l'accesso online ai database delle sequenze di proteine e DNA, contenenti sia le sequenze che informazioni aggiuntive su di esse.

Nel corso degli anni si è inoltre sentita la necessità di sviluppare software per analizzare le sequenze in vari modi, in particolare metodi per confrontarle tra di loro al fine di rilevare similitudini.

Uno dei primi metodi di confronto è stata la dot matrix, dove due sequenze erano confrontate creando una matrice e scrivendo una sequenza lungo le righe e l'altra lungo le colonn. Quando una stessa lettera appare in entrambe le sequenze un punto(dot) è piazzato all'intersezione delle posizioni corrispondenti nella matrice. La matrice poi è analizzata per cercare le serie di punti che formano diagonali, le quali rilevano similarità dato che corrispondono ad un allineamento locale tra due sotto-sequenze. Tale metodo permette inoltre di rilevare inserzioni, cancellazioni, presenza di pattern ripetuti e pattern invertiti. Essendo un metodo visuale esso non trova in modo veloce la similarità tra due sequenze se vi sono regioni che non corrispondono molto bene o che sono presenti in una sola delle due sequenze.

Sono stati in seguito sviluppati metodi per l'allineamento di due sequenze, per far fronte alle debolezze del metodo dot-matrix. L'allineamento è importante per trovare informazioni strutturali, funzionali ed evoluzionarie di sequenze biologiche. Infatti, sequenze che sono molto simili probabilmente hanno la stessa funzione e/o potrebbero avere una stessa sequenza antenata. La similarità quindi può indicare un qualche tipo di relazione antenata.

Needleman e Wunsch (1970) hanno proposto un algoritmo di programmazione dinamica che riesce a trovare un percorso attraverso la dot-matrix che fornisce il miglior possibile allineamento, chiamato allineamento ottimale, tra due sequenze. Tale metodo trova l'allineamento ottimale considerando l'interezza delle due sequenze, e viene perciò chiamato allineamento globale [11]. Smith e Waterman (1981) hanno riconosciuto che le

regioni biologiche più significative nelle sequenze di proteine e DNA sono sotto-regioni che allineano tra di loro bene e che le rimanenti regioni costituite da sequenze meno relazionate sono meno importanti. Hanno perciò modificato l'algoritmo Needleman-Wunsch per individuare tali regioni e costruire un allineamento chiamato locale, dove tratti di sequenze col più alto numero di accoppiamenti sono allineati, generando una o più regioni di sotto-allineamenti nelle sequenze da allineare. Metodi per valutare l'allineamento sono stati sviluppati, in particolare basandosi su punteggi di distanza e di similarità tra due sequenze allineate.

Un problema incontrato con l'allineamento di sequenze è quello di dover decidere se un allineamento sia significativo. Sono stati perciò analizzati i punteggi di sequenze casuali e scorrelate e ne è stata derivata la distribuzione nota come Extreme Value Distribution. Essa permette di valutare la probabilità che un punteggio tra due sequenze possa essere trovato anche in un allineamento tra sequenze scorrelate e casuali della stessa lunghezza.

Analizzando sempre più sequenze, i biologi molecolari hanno scoperto che molti organismi condividono geni simili che possono essere identificati grazie alla loro similarità sequenziale. La necessità di trovare un gene in un nuovo organismo confrontando la sua similarità con geni di organismi presi come modelli è diventata sempre più presente, e sono quindi stati sviluppati algoritmi e database per facilitare tale compito. Al rapido aumentare delle dimensioni dei database delle sequenze biologiche gli algoritmi di programmazione dinamica si sono rilevati inefficienti allo scopo di confrontare le sequenze di insiemi molto ampi a causa del carico computazionale notevole.

Sono stati quindi sviluppati degli algoritmi euristici per permettere una ricerca per similarità rapida, anche se meno accurata, nei database.

Il primo algoritmo, sviluppato nel 1988, è FASTA [12], un programma che effettua una scansione di un database di sequenze alla ricerca di similarità in un tempo ragionevole. Esso fornisce un metodo rapido per trovare sotto-sequenze corte di similarità tra una sequenza e qualunque sequenza nel database. Un programma ancora più veloce è stato successivamente sviluppato nel 1990, chiamato BLAST [13]. Esso è raggiungibile dal sito web del National Center for Biotechnolohy Information. Come FASTA esso prepara una tabella di sequenze di parole corte in ogni sequenza, ma determina anche quale di queste parole sono più significanti per la similarità. BLAST ha varie versioni in base al tipo di sequenza da confrontare. I principali sono *blastp*, che permette di cercare similarità in banche dati proteiche a partire da una query di amminoacidi, *blastn* che cerca similarità in banche di dati nucleotidi a partire da una query di nucleoitidi e *blastx* che cerca similarità in banche di dati proteiche a partire da una query di nucelotidi che viene tradotta in tutti i frame.

BLAST e FASTA si basano su metodi di allineamento tra due sequenze considerando piccole porzioni di sequenza e riflettono l'assunzione della conservazione della contiguità tra segmenti omologi. Tali approcci euristici inoltre rendono più difficile valutare la rilevanza statistica dei punteggi risultanti dagli allineamenti.

Per rimediare alle limitazioni dei criteri di confronto mediante allineamenti sono stati sviluppati vari metodi di confronto **liberi dall'allineamento** basati su diversi presupposti teorici.

2.2 Metodi alignment-free

Nei metodi liberi dall'allineamento si rappresentano le sequenze come vettori di caratteristiche che spesso sono una funzione del numero di parole che in esse occorrono. Le dimensioni dei vettori sono in generale date da tutte le possibili parole di una qualche lunghezza prefissata m. Una volta definiti i vettori che rappresentano le sequenze si definisce una misura di similarità tra due sequenze come una funzione sui vettori che le rappresentano. La scelta di tale funzione è cruciale in quanto definisce il metodo di confronto libero dall'allineamento.

Nel seguito si introducono brevemente i concetti dietro le tecniche alignment-free delle sequenze mediante l'analisi delle parole che le compongono, rimandando a [14] per approfondimenti. Si introduce ora la rappresentazione di una stringa come un vettore sulle frequenze delle parole che la costituiscono.

Una sequenza X di lunghezza n, è una successione di n simboli presi da un alfabeto A, di cardinalità r. Un segmento di L simboli, con L <= n, viene detto L-tupla, L-word o L-mer. Con W_L si denota l'insieme di tutte le K possibili L-tuple, con $K = r^L$.

$$W_L = \{w_{L,1}, w_{L,2,\cdots,}w_{L,K}\}$$

La ricerca di L-tuple in una sequenza X consiste nel effettuare un conteggio delle occorrenze degli elementi di W_L. Ciò si effettua facendo scorre lungo X una finestra di lunghezza L, dalla posizione 1 alla posizione n – L + 1. In tale modo si ottiene il vettore word-count:

$$C^X_L = \{ c^X_{L,1}, c^X_{L,2,\cdots,} c^X_{L,K}\}$$

Dove $c^X_{L,i}$ è il numero di occorrenze di $w_{L,i}$ in X. In modo simile è possibile calcolare il vettore delle frequenze delle L-tuple in X:

$$F^X_L = \{ f^X_{L,1}, f^X_{L,2},\ldots, f^X_{L,K}\}$$

con:

$$f^X_{L,i} = c^X_{L,i} / (n - L + 1)$$

Da $f^X_{L,i}$ si possono stimare le probabilità di trovare ogni possibile L-tupla in X, ricavando:

$$P^X_L = \{ p^X_{L,1}, p^X_{L,2},\ldots, p^X_{L,K}\}$$

Con $p^X_{L,i}$ la probabilità di $w_{L,i}$ in X.

Sono state trovate formule per contare il valore atteso, la varianza e la covarianza tra le frequenze di due parole, cioè la distribuzione di P^X_L e la determinazione dei suoi momenti. Per una corretta stima delle covarianze di P^X_L è necessario che tali misure prevedano la caratteristica di sovrapposizione delle L-tuple, cioè la capacità di conteggiare correttamente le L-tuple che si susseguono codividendo prefissi e suffissi.

Tale problema si ha ad esempio nelle metriche basate sulla distanza di Mahalanobis.

2.2.1 Metodi fondamentali

In letteratura sono stati presentati metodi basati sul conteggio delle L-tuple ad una specifica risoluzione, cioè metriche definite nello spazio dei vettori word-count, e metodi non basati sul conteggio delle L-tuple a lunghezza fissata (ad esempio le mappe iterative).

La tecnica più semplice di confronto alignment-free di due sequenze è la distanza euclidea. Per una data risoluzione L, la ***distanza quadratica euclidea*** tra due sequenze X e Y è definita come:

$$d_L^E(X,Y) = (C_L^X - C_L^Y)^T(C_L^X - C_L^Y) = \sum_{i=1}^{K}\left(c_{L,i}^X - c_{L,i}^Y\right)^2$$

Tale metrica è stata validata applicandola al confronto di lunghe sequenze relative ad organismi in relazione filogenetica ampiamente documentata.

Una successiva evoluzione di tale tecnica è stata la ***distanza euclidea pesata***. Gli studi delle sequenze biologiche hanno dimostrato che alcuni segmenti sono più frequenti di altri. Per tenere conto di tale fattore, è stata proposta tale metrica pesa che dà maggiore peso al conteggio di determinate parole piuttosto che altre.

$$d^2(X,Y) = \sum_{L=l}^{u}\sum_{i=1}^{K}\rho_i\left(c_{L,i}^X - c_{L,i}^Y\right)^2$$

La metrica pesata d^2 utilizza pesi ρ_i per ogni possibile parola, combinando inoltre diverse risoluzioni di L da l ad u. Tale metrica si è dimostrata abbastanza efficace ed implementabile efficientemente. Le metriche euclidee ad oggi sono spesso usate come filtro di pre-elaborazione per isolare da larghi database le migliori sequenze candidate per l'applicazione successiva di algoritmi quali FASTA e BLAST.

2.2.2 STRING KERNELS

Semplicemente parlando uno string kernel è una funzione su due stringhe che ne misura la similarità. Più è alto il valore del kernel e più simili sono tra loro le sequenze.

Il primo string kernel presentato in letteratura è stato lo spectrum kernel, introdotto da Leslie et al. in [15]. Esso è basato sul conteggio esatto delle parole che sono presenti nelle stringhe da confrontare. Si è poi passati ad introdurre la possibilità che vi siano dei mismatch nelle parole che costituiscono le stringhe definendo il mismatch kernel, introdotto in [15]. Tale kernel verrà descritto nel dettaglio in 4.2. Lo spectrum kernel è un caso particolare di tale kernel nel caso in cui si considerino k=0 mismatch.

Aumentando la qualita' della descrizione (perdendo però in velocità) si è passati ai metodi basati sui profili delle sequenze di ciascuna famiglia [16] e che rappresentano a oggi lo stato dell'arte nella classificazione di proteine.

3 Database SCOP e raccolta ASTRAL

Il database Structural Classification of Proteins (SCOP) [18] fornisce una descrizione dettagliata e comprensiva delle relazioni tra tutte le strutture di proteine. La classificazione è a livelli gerarchici. Nella classificazione vi è distinzione tra relazioni evoluzionarie e quelle che derivano dalla fisica e chimica delle proteine. Il database sorgente contente le strutture delle proteine da cui SCOP parte per effettuare la classificazione è il database PDB.

Nel seguito si descriveranno brevemente i concetti dietro tale database e i livelli di classificazione delle proteine in SCOP.Successivamente si introduce ASTRAL, una raccolta di strumenti atti a migliorare l'analisi delle proteine tramite lo studio delle loro sequenze.

3.1 Protein Data Bank (PDB)

Protein Data Bank (PDB) è un archivio centralizzato che fornisce accesso alla struttura delle proteine. Esso contiene file che descrivono la struttura tridimensionale delle proteine e degli acidi nuclieici. Tali strutture sono ottenute principalmente tramite la cristallografia a raggi X e la spettrografia NMR.

Nel metodo tramite raggi X si ottiene per prima cosa il cristallo della proteina, usualmente tramite il metodo della "goccia sospesa". Successivamente si utilizzano i raggi X contro il cristallo della proteina che si vuole studiare. Un fascio di raggi viene ottenuto da una sorgente. In seguito quando il raggio primario colpisce il cristallo della proteina, una parte di esso viene diffranta. Tali raggi diffranti vengono poi registrati da un rilevatore elettronico bidimensionale. In Figura 3-1, tratta da [19], un'illustrazione semplificata di tale processo.

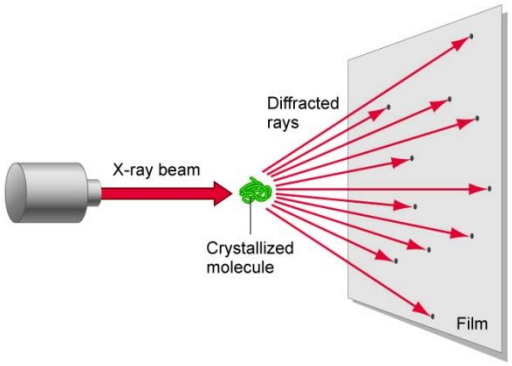


Figura 3-1: Cristallografia a raggi X

I dati ottenuti dalla cristallografia vengono inseriti in un file PDB. Esso è composto da diverse sezioni, tra le più importanti si trovano:

- HEADER: riporta il nome della famiglia della proteina, la data di creazione del file e il codice identificativo.
- TITLE: riporta il nome completo della proteina.
- COMPND: descrive la catene da cui la proteina è formata.
- SOURCE: contiene informazioni sulla famiglia della proteina e sui sistemi biologici di provenienza.
- KEYWDS: parole chiave che descrivono il file.
- EXPDTA: metodo di analisi della struttura della proteina utilizzato.
- AUTHOR: nomi degli autori del file.
- REVDATA: i file PDB possono essere revisionati per migliorarli e correggerli, tale sezione contiene tutte le revisioni effettuate sul file.

- JNRL: pubblicazioni scientifiche sulla proteina analizzata.
- REMARK: informazioni dettagliate su come è stata ottenuta la struttura.
- SEQRES: contiene la sequenza di amminoacidi che costituisce la sequenze primaria della proteina.
- ATOM: contiene la struttura atomica della proteina. Per ogni atomo definisce il tipo e la sua posizione nello spazio.

In Figura 3-2 un esempio di file PDB per la proteina di ID 1HV4. Si noti in particolare la sezione SEQRES, che verrà utilizzata nel seguito per ottenere le sequenze di amminoacidi.

```
HEADER    OXYGEN STORAGE/TRANSPORT                 07-JAN-01   1HV4
TITLE     CRYSTAL STRUCTURE ANALYSIS OF BAR-HEAD GOOSE HEMOGLOBIN
TITLE    2 (DEOXY FORM)
COMPND    MOL_ID: 1;
COMPND   2 MOLECULE: HEMOGLOBIN ALPHA-A CHAIN;
COMPND   3 CHAIN: A, C, E, G;
COMPND   4 MOL_ID: 2;
COMPND   5 MOLECULE: HEMOGLOBIN BETA CHAIN;
COMPND   6 CHAIN: B, D, F, H
SOURCE    MOL_ID: 1;
SOURCE   2 ORGANISM_SCIENTIFIC: ANSER INDICUS;
SOURCE   3 ORGANISM_COMMON: BAR-HEADED GOOSE;
SOURCE   4 ORGANISM_TAXID: 8846;
SOURCE   5 TISSUE: BLOOD;
SOURCE   6 CELL: ERYTHROCYTES;
SOURCE   7 CELLULAR_LOCATION: CYTOPLASM;
SOURCE   8 MOL_ID: 2;
SOURCE   9 ORGANISM_SCIENTIFIC: ANSER INDICUS;
SOURCE  10 ORGANISM_COMMON: BAR-HEADED GOOSE;
SOURCE  11 ORGANISM_TAXID: 8846;
SOURCE  12 TISSUE: BLOOD;
SOURCE  13 CELL: ERYTHROCYTES;
SOURCE  14 CELLULAR_LOCATION: CYTOPLASM
KEYWDS    ALLOSTERIC MECHANISM, OXYGEN TRANSPORT, HEME, RESPIRATORY
KEYWDS   2 PROTEIN, OXYGEN STORAGE/TRANSPORT COMPLEX
EXPDTA    X-RAY DIFFRACTION
AUTHOR    Y.LIANG,Z.HUA,X.LIANG,Q.XU,G.LU
[…]
SEQRES   1 A  141  VAL LEU SER ALA ALA ASP LYS THR ASN VAL LYS GLY VAL
SEQRES   2 A  141  PHE SER LYS ILE SER GLY HIS ALA GLU GLU TYR GLY ALA
SEQRES   3 A  141  GLU THR LEU GLU ARG MET PHE THR ALA TYR PRO GLN THR
SEQRES   4 A  141  LYS THR TYR PHE PRO HIS PHE ASP LEU GLN HIS GLY SER
SEQRES   5 A  141  ALA GLN ILE LYS ALA HIS GLY LYS LYS VAL VAL ALA ALA
SEQRES   6 A  141  LEU VAL GLU ALA VAL ASN HIS ILE ASP ASP ILE ALA GLY
SEQRES   7 A  141  ALA LEU SER LYS LEU SER ASP LEU HIS ALA GLN LYS LEU
[…]
ATOM      1  N   VAL A   1      50.477  17.846  26.910  1.00 57.45   N
ATOM      2  CA  VAL A   1      50.912  18.466  28.201  1.00 52.61   C
ATOM      3  C   VAL A   1      50.038  19.679  28.502  1.00 50.42   C
ATOM      4  O   VAL A   1      49.618  19.883  29.642  1.00 53.55   O
ATOM      5  CB  VAL A   1      52.399  18.949  28.150  1.00 53.04   C
ATOM      6  CG1 VAL A   1      53.298  17.855  27.571  1.00 50.41   C
[…]
END
```

Figura 3-2: File PDB per la proteina 1HV4

Esistono vari tool per la visualizzazione 3D delle proteine a partire dal file PDB che la descrive. I più utilizzati sono JMOL Viewer e RASMOL. In Figura 3-3 la struttura tridimensionale visualizzata con JMOL della prima catena della proteina 1HV4.

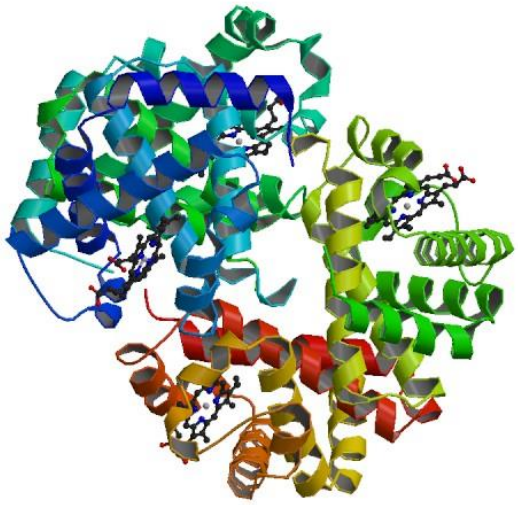


Figura 3-3: Biological Assembly 1 di 1HV4

3.2 Livelli gerarchici di classificazione delle proteine

La classificazione delle proteine in SCOP è effettuata mediante l'ispezione visuale e il confronto delle strutture. Sono stati definiti 6 livelli, illustrati nella figura Figura 3-4 tratta da [18].

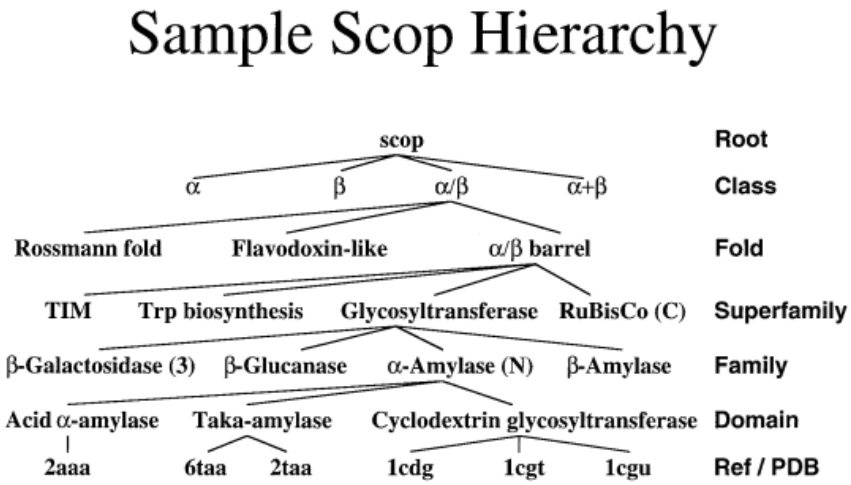


Figura 3-4: gerarchia SCOP

- Class: definisce il tipo di fold della proteina dato dalla struttura secondaria. Le classi più comuni sono:
  - All-α: contiene le strutture che sono principalmente formate da α-eliche.
  - All-β: contiene le strutture che sono principalmente formate da foglietti-β.
  - α/β: contiene le strutture che hanno foglietti-β e α-eliche.
  - α+β: proteine nelle quali le α-eliche e i foglietti-β sono molto isolati.
- Fold: le famiglie e le superfamiglie hanno una fold comune se le loro proteine hanno qualche struttura secondaria maggiore comune con la stessa disposizione e tipologie di connessione [18].

- Superfamiglia: contiene proteine che hanno bassa identità tra di loro ma le quali strutture e, in molti casi, le caratteristiche funzionali suggeriscono un'origine evolutiva comune molto probabile.
- Famiglia: Le proteine sono raggruppate insieme in famiglie sulla base di due criteri che implicano che esse hanno un'origine evolutiva comune:
    - Tutte le proteine devono avere similarità sequenziale significante o
    - Le proteine con bassa similarità devono avere funzione e struttura molto simile.
- Dominio di proteina: i domini nelle famiglie sono raggruppati nei domini di proteina. Le proteine sono inserite nello stesso dominio di proteina se sono tra loro isoformi o se sono essenzialmente la stessa proteina, ma da specie differenti. I domini sono ulteriormente divisi in specie.
- Specie: i domini sono raggruppati infine in specie di appartenenza. (non illustrato nella figura)
- Ref/PDB: è il riferimento al file PDB della proteina e alla singola sequenza di amminoacidi del dominio in una determinate specie.

Ogni nodo della gerarchia SCOP è dotato di un identificatore univoco chiamato "sunid". Ad esempio il sunid della classe "*a: All alpha proteins*" è 46456.

3.3 ASTRAL

ASTRAL è una raccolta di strumenti che ha l'obbiettivo di aiutare nell'analisi della struttura delle proteine, in particolare tramite l'utilizzo delle loro sequenze. Essa definisce dei punteggi, chiamati *SPACI scores*, con lo scopo di riassumere le caratteristiche della struttura di una proteina. Un file PDB contiene coordinate della struttura atomica che possono contenere irregolarità. Per le strutture che sono state determinate con la cristallografia a raggi X, il punteggio AEROSPACI fornisce una misura approssimata della qualità strutturale ed è molto utile per selezionare un sottoinsieme rappresentativo dei file PDB per proteine altamente relazionate tra loro [20], [21].

Vengono inoltre definiti dei file che mappano per un dato PDB la sequenza SEQRES con la struttura atomica ATOM. Le porzioni di SEQRES che contengono la sequenza di una molecola possono non avere una relazione diretta ed immediata con la struttura molecolare ATOM. Il Crystallographic Information Format [20] cerca di rimediare a tali problemi mappando gli atomi nella sequenza lineare. I file CIFMAP in ASTRAL vengono prima generati automaticamente col programma pdb2cif e poi corretti manualmente per eliminare gli errori causati dalla mappatura automatica. La mappatura è distribuita nel formato RAF (Rapid Access Format).

ASTRAL fornisce due tipi di sequenze per i domini di proteine nel database SCOP. Il primo tipo fornisce le sequenze dei residui che corrispondono ai record ATOM nell'intervallo specificato nel dominio SCOP. Il secondo, utilizzato usualmente, produce le sequenze in base ai record SEQRES.

Un dominio SCOP può includere frammenti da differenti catene. Nella maggior parte dei casi esso è il prodotto di un singolo gene. Non è facile riordinare i frammenti nell'ordine trovato nella sequenza di gene originale, dato che l'ordine nel quale le catene sono presentate nel file PDB spesso non ha correlazione con l'ordine genetico. L'ordine corretto delle catene è assicurato grazie ad una cura manuale effettuata dagli autori di SCOP e ASTRAL [1].

Le sequenze nelle versioni ASTRAL precedenti al 2001 sono rappresentate con lo stile "***originale***", nel quale c'è una voce separata per ogni catena inclusa in un dominio SCOP, con l'iniziale 'd' (indicante un dominio), sostituita con 'e'. La figura Figura 3-5, tratta da [1], mostra un esempio della proteina d1cph1, un dominio multi-catena. Tale dominio è diviso in una sequenza per ogni catena (catene a e b): *e1cph.1a* e *e1cph.1b*.

Figura 3-5: dominio 1cph rappresentato in stile originale

Le sequenze nelle versioni successive di ASTRAL sono rappresentate con un nuovo stile chiamato "***genetic domain***". In esso le sequenze sono concatenate in una singola voce con la 'd' iniziale nel nome del dominio rimpiazzata da 'g'. In Figura 3-6 vi è rappresentato il dominio 1cph nel nuovo stile.

Figura 3-6: dominio 1cph nello stile genetic domain

Le catene di sequenze sono messe nell'ordine corretto, separate dalla lettera 'X'.

Sotto-insiemi rappresentativi

La maggior parte dei domini in PDB sono gli uni molto simili agli altri. ASTRAL aiuta a ridurre la ridondanza selezionando domini rappresentativi di alta qualità a differenti livelli di similarità. Per ogni dominio SCOP esso crea quattro insiemi, due utilizzando le sequenze con lo stile originale e due utilizzando le sequenze con lo stile "genetic domain". Per ognuna di queste due metodologie un insieme di sequenze è derivato dalle sequenze nei record ATOM dei file PDB e l'altro usando i record SEQRES.

Gli insiemi SEQRES sono infine utilizzati per derivare sotto-insiemi rappresentativi. Ogni insieme è confrontato con se stesso usando BLAST [21] e i sotto-insiemi sono creati usando tre criteri di similitudine: BLAST E-value, sequence identity e SCOP classification, descritti in [1] e [20]. I sottoinsiemi rappresentativi più utilizzati sono quelli ai livelli 40% e 95% di sequence identity ID.

Un database ID di percentuale xx viene usualmente indicato con PDBxx, ad esempio PDB90 è il sotto-insieme rappresentativo al livello 90% di identità sequenziale.

Esso è creato ordinando tutti i domini in SCOP secondo i loro punteggi AEROSPACI in una lista. Il dominio con più alto punteggio è rimosso dalla lista e incluso del sotto-insieme rappresentativo, successivamente sono rimossi da essa anche tutti i domini sotto il livello soglia di identità con la sequenza appena rimossa. Il meccanismo viene ripetuto fino a quando la lista di svuota.

3.4 Integrazione di SCOP con ASTRAL

Il database SCOP, a partire dalla versione 1.75A è stato integrato con ASTRAL. Il database è consultabile online all'indirizzo [22]. La versione attuale al momento della stesura di tale documento è la 1.75B, pubblicata nel gennaio 2013. Contiene 52316 voci PDB e 140293 domini. Dalla versione 1.75A ASTRAL e SCOP vengono forniti, oltre che in versione file di testo analizzabili, anche sotto forma di database MySQL. Nel seguito viene descritto il database MySQL relativo alla versione 1.75B.

Il database può essere suddiviso in quattro sezioni relazionate tra loro:

- Archivio e descrizione dei file PDB e dei loro punteggi AEROSPACI
- SCOP
- ASTRAL
- PFAM e immagini

Tali sezioni vengono analizzate separatamente nel seguito. Negli schemi relazionali sottostanti viene utilizzata per le relazioni la notazione di cardinalità "a zampa di gallina", dove una linea indica l'associazione tra due tabelle, e alle estremità di queste ci sono dei segni indicanti il tipo di relazione. In Tabella 3-1 si illustra il significato di tali segni.

Simbolo	Significato
	Zero o più
	Uno o più
	Uno e solo uno
	Zero o uno

Tabella 3-1: Simboli utilizzati nella notazione "crow's foot"

3.4.1 Archivio file PDB

In Figura 3-7 si mostra la sezione di schema relazionale contenente le tabelle che descrivono i file PDB.

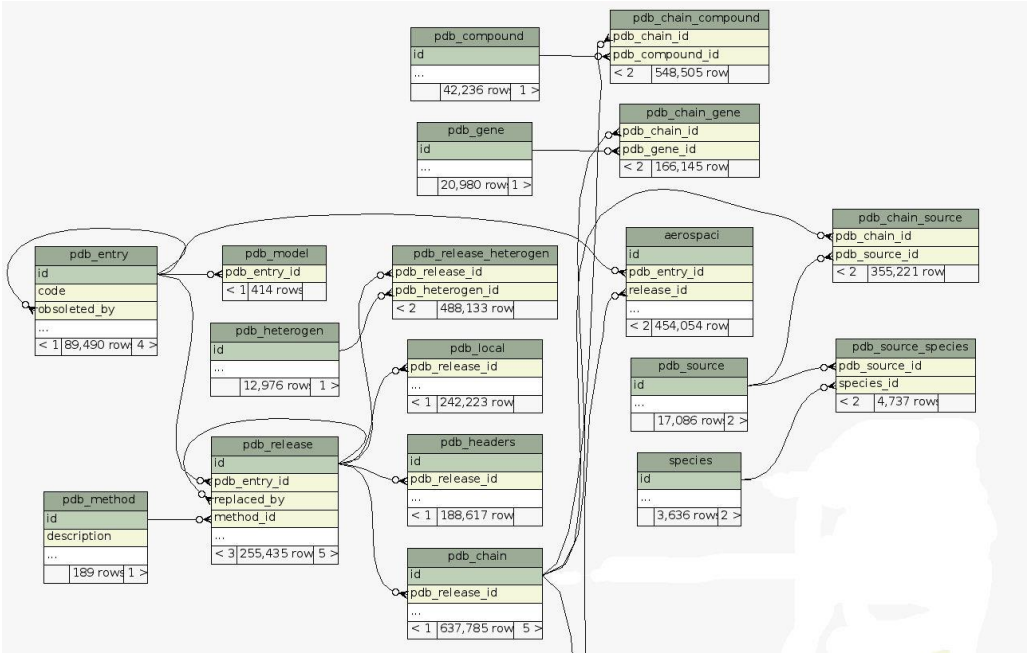


Figura 3-7: Sezione per file PDB

Si descrive nel seguito il contenuto delle tabelle.

■ **Tabella pdb_entry**

Una entry in tale tabella rappresenta un file PDB e contiene una descrizione del file, la data di deposito, la data di release, la data in cui eventualmente il file è diventato obsoleto e un riferimento al file PDB che lo ha rimpiazzato.

■ **Tabella pdb_model**

In essa sono memorizzati i riferimenti alle entità pdb_entry che sono scelte come modello di riferimento

■ **Tabella pdb_release**

Serve a descrivere i file PDB. Un'entità contiene tutte le informazioni principali associate ad una pdb_entry. Nel dettaglio contiene la data di pubblicazione e quella di revisione del file PDB associato, le eventuali release che hanno rimpiazzato tale release, il metodo di creazione del file PDB tra quelli specificati nella tabella pdb_method, il punteggio SPACI del file PDB ed informazioni per il calcolo di tale punteggio (resolution ed r_factor, procheck e what_check values) .

■ **Tabella pdb_method**

Elenca i metodi sperimentali PDB

■ **Tabella pdb_headers**

Contiene gli header di un file PDB. Ogni entità della tabella è associata ad una voce pdb_release.

■ **Tabella pdb_local**

Contiene le informazioni sulle copie locali nel server dei file PDB, quali il percorso del file locale. Ogni entità della tabella è associata ad una voce pdb_release.

■ **Tabella pdb_release_heterogen**

Associa ogni pdb_release ad uno o più pdb_heterogen.

■ **Tabella pdb_heterogen**

La sezione heterogen di un file PDB contiene le descrizioni complete dei residui non standard nelle entry. La tabella elenca solo gli heterogen presenti.

■ **Tabella pdb_chain**

Contiene le catene associate ad una proteina. Un'entità pdb_release può avere una o più catene.

■ **Tabella pdb_gene**

Contiene i geni che producono le proteine nel DB PDB. I dati vengono ricavati dai campi SOURCE dei file PDB.

■ **Tabella pdb_chain_gene**

Associa una catena pdb_chain col rispettivo gene che la genera. Non tutte le catene hanno associate informazioni sui geni.

■ **Tabella pdb_compound**

Descrive i compound delle catene. Un compound è un complesso di proteina che combina gli amminoacidi con altre sostanze.

■ **Tabella pdb_chain_compound**

Relazione N-N tra catene e compound. Nel database attuale la tabella non è normalizzata e ci sono erroneamente rari casi di catene elencate più volte nello stesso compound. Ad esempio la catena di chain_id=206941 ha in tale tabella le entry (chain_id=206941,compund_id=5692) e (20694,5692).

■ **Tabella aerospaci**

Contiene i punteggi AEROSPACI dati ai pdb_entry per ogni release SCOP.

■ **Tabella pdb_source_species**

In essa le sorgenti PDB sono mappate alle specie SCOP contenute nella tabella species.

■ **Tabella pdb_source**

Contiene le sorgenti PDB, cioè le specie presenti nei file PDB. Le informazioni includono il nome scientifico, il nome comune, e la tassonomia NCBI.

■ **Tabella pdb_chain_source**

Relazione N-N tra catene e specie PDB.

3.4.2 Sezione SCOP

In Figura 3-8 si mostra la sezione di schema relazionale contenente l'archivio SCOP. Di seguito si descrivono i contenuti delle tabelle.

■ **Tabella scop_level**

Elenca i livelli gerarchici SCOP, cioè root, class, fold, superfamily, family, protein domain, species e PDB entry.

■ **Tabella scop_node**

E' una della tabelle più importanti del database. Contiene le entry rappresentanti i nodi della gerarchia SCOP. I campi di tale tabella sono:

- Id: identificativo numerico
- sunid: identificativo universale SCOP
- sccs: id breve SCOP
- sid: ID del dominio (il campo non è vuoto solo per le entry rappresentanti domini)
- description: descrizione estesa SCOP
- level_id: campo chiave esterna che associa l'entità al suo livello in scop_level
- parent_node_id: chiave esterna che specifica il nodo scop_node padre
- release_id: versione SCOP in cui si trova il nodo

■ **Tabella scop_comment**

Contiene i commenti associati a scop_node.

■ **Tabella species**

Contiene le specie SCOP.

■ **Tabella link_species**

Relazione N-N tra species e scop_node.

■ **Tabella link_pdb**

Relazione N-N tra un nodo scop di livello domain e catena pdb padre.

■ **Tabella link_pfam**

Contiene link al database esterno PFAM. Esso è un DB di allineamenti e HMM(Hidden Markov Models) per famiglie di proteine.

■ **Tabella link_uniprot**

Contiene link al database esterno UNIPROT. Esso è un database di sequenze di proteine e informazioni funzionali.

■ **Tabella scop_subset_level**

Indica i nodi rappresentativi selezionati da ogni livello SCOP. Per ogni nodo di un livello si seleziona un nodo dell'ultimo livello (PDB) che più lo rappresenta. In tale release un nodi di un livello è rappresentato da un solo nodo PDB, mentre un nodo PDB può rappresentare più nodi di livello.

■ **Tabella scop_history**

Contiene la cronologia tra i nodi SCOP. Indica un dato nodo SCOP da quale nodo è stato sostituito nella release successiva, se ciò avviene. Il campo old_node_id contiene l'id del vecchio nodo scop, il campo new_node_id contiene l'id del nuovo nodo scop.

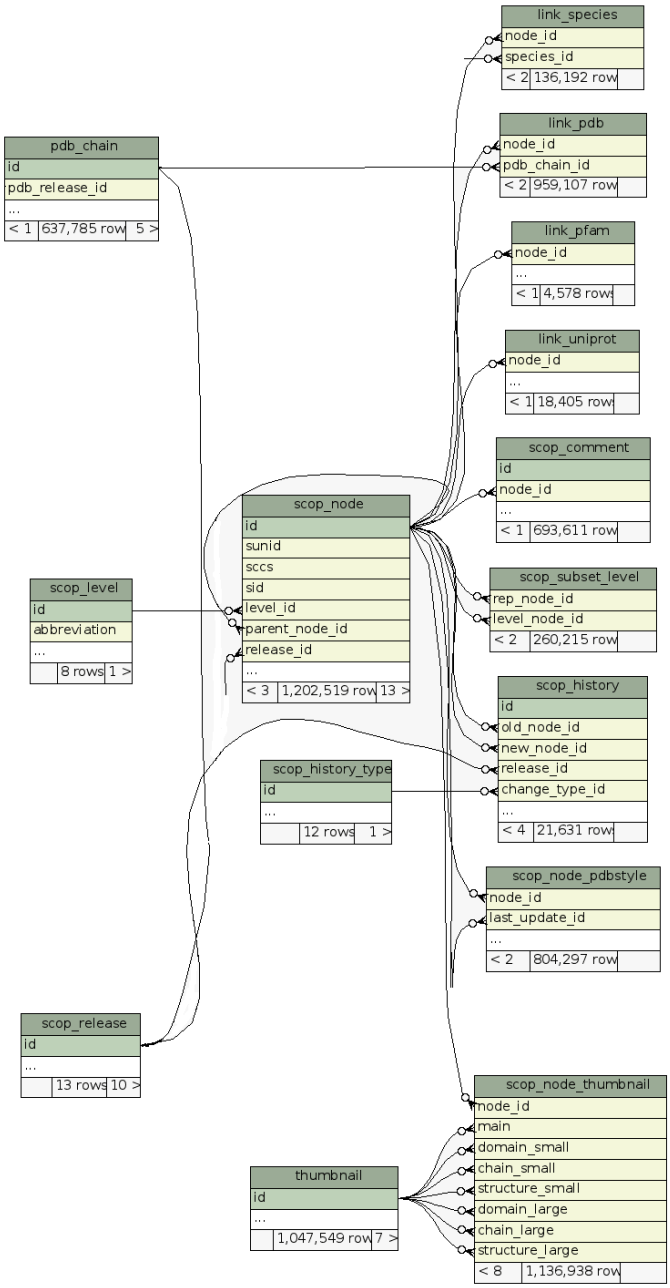


Figura 3-8: Sezione SCOP

■ **Tabella scop_history_type**

Descrive il tipo di cambiamento effettuato in scop_history. I tipi sono: cambio per riferimento obsoleto, per unione, divisione e spostamento.

■ **Tabella scop_node_pdbstyle**

E' una relazione N-N tra scop_node e scop_release contenente il percorso ai file Astral in stile PDB.

■ **Tabella scop_release**

Elenca le release SCOP dalla versione 1.55 a 1.75B.

■ **Tabella thumbail**

Contiene le immagini delle strutture tridimensionali dei domini delle proteine.

■ **Tabella scop_node_thumbnail**

Associa ogni nodo SCOP (scop_node) di ultimo livello alle sue immagini rappresentative in thumbnail.

3.4.3 Sezione ASTRAL

In Figura 3-9 si mostra la sezione di schema relazionale inerente all'archivio ASTRAL e di seguito verranno descritte le varie tabelle.

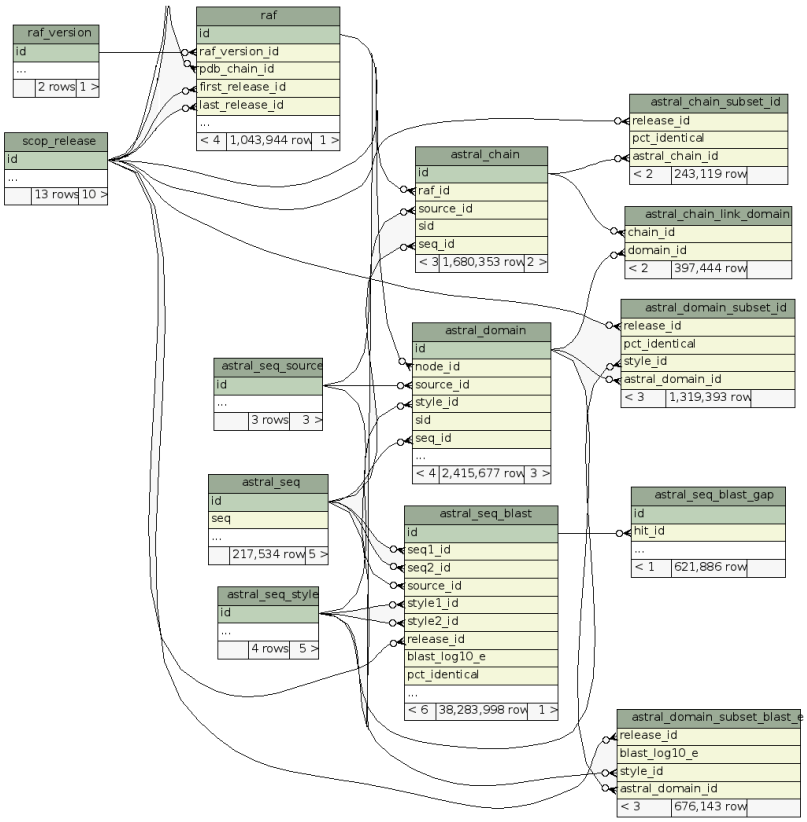


Figura 3-9: Sezione ASTRAL

■ **Tabella raf**

Descrive le mappe RAF (Rapid Access Format) che mappano le strutture atomiche nelle sequenze di amminoacidi. I campi di un entità contengono la versione di mappatura effetuata, la catena pdb (pdb_chain) mappata, la prima e l'ultima release SCOP in cui viene usata la mappatura RAF, e la descrizione della mappatura.

■ **Tabella raf_version**

Elenca le versione delle mappature RAF

■ **Tabella astral_seq_source**

Tipo di sorgente da cui deriva una sequenza ASTRAL. I tipi sono "ATOM", per le sequenze derivate dai record PDB ATOM, "SEQRES" per le sequenze derivate dai record PDB SEQRES e "SEQRES with first/last ATOM boundaries", usato solo per le release precedenti 1.65.

■ **Tabella astral_seq_style**

Elenca gli stili delle sequenze astral. Essi sono “single chain” per le sequenze di dominio in un'unica catena, “multi-chain original style” per le sequenze derivanti da domini in più catene scritte secondo lo stile originale, “multi-chain genetic domain” per le sequenze scritte nello stile “genetic domain” e “ASTEROIDS” per le sequenze inserite automaticamente nel database.

■ **Tabella seq_astral**

Contiene le sequenze ASTRAL.

■ **Tabella astral_chain**

Associa ogni mappatura RAF(entità raf) per ogni tipo di sorgente (astral_seq_source) alla relativa sequenza in seq_astral che rappresenta una catena proteica.

■ **Tabella astral_chain_subset_id**

Contiene i riferimenti alle catene PDB rappresentative (in astral_chain) per i sottoinsiemi significativi di tipo ID %.

■ **Tabella astral_domain**

Associa ogni nodo SCOP (scop_node) alla relative sequenze astral (astral_seq) per sorgente e stile.

■ **Tabella astral_chain_link_domain**

Associa i domini astral alle catene di appartenenza per le mappatura automatiche (sequenze di tipo ASTEROIDS).

■ **Tabella astral_domain_subset_id**

Contiene i riferimenti ai domini rappresentativi SCOP selezionati per % ID.

■ **Tabella astral_seq_blast**

Contiene i risultati BLAST tra coppie di sequenze. Sono confrontate le sequenze con sorgente (astral_seq_source) uguale, mentre gli stili possono essere diversi.

■ **Tabella astral_seq_blast_gap**

Contiene informazioni sui gap degli allineamenti BLAST della tabella astral_seq_blast.

■ **Tabella astral_domain_subset_blast_e**

Contiene i riferimenti ai domini rappresentativi selezionati per valore E-value BLAST.

3.4.4 Sezione PFAM

Pfam è un database di famiglie di proteine che include le loro annotazioni e gli allineamenti di sequenze multiple generati usando i modelli di Markov nascosti [22]. La sezione contiene informazioni su tale database e la mappatura tra le sequenze ASTRAL e i modelli HMM di PFAM. In Figura 3-10 le tabelle di tale sezione.

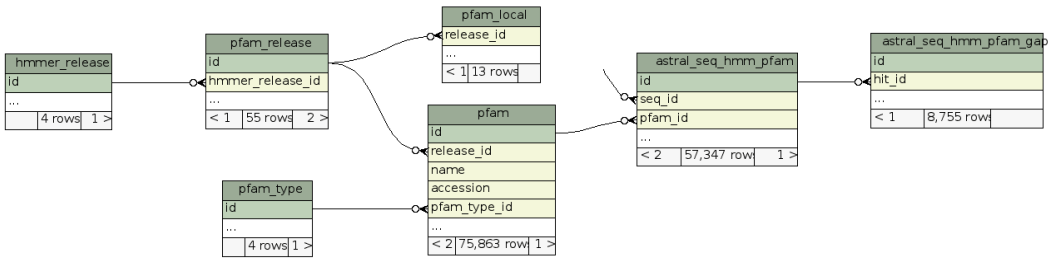


Figura 3-10: Sezione PFAM

3.5 Alcune interrogazioni significative su SCOP-ASTRAL

In questa sezione si descrivono alcune interrogazioni utili per creare insiemi di dati che possono essere utilizzati per la valutazione delle prestazioni dei classificatori nei problemi di rilevamento delle omologie remote.

Selezione di tutte le sequenze in una famiglia

In Figura 3-11 si descrive l'interrogazione necessaria per selezionare tutte le sequenze di una famiglia, usando un dato sotto-insieme significativo % ID. Questa interrogazione è utile in quanto può essere utilizzata per ottenere l'insieme degli esempi di test positivi per il problema del rilevamento delle omologie remote.

```
01 set @id_family       := 1076961;
02 set @pct_id          := 90;
03 set @rel_id          := 13;
04
05 SELECT DISTINCT admn.id ,admn.sid, admn.style_id, ss.style_id , admn.source_id,
06          dm.sccs, seq.seq
07               FROM ((astral.scop_node AS pr                          #protein domain
08                         INNER JOIN astral.scop_node AS sp            #specie
09                         ON sp.parent_node_id = pr.id)
10                                  INNER JOIN astral.scop_node AS dm   #pdb domain
11                                  ON (dm.parent_node_id = sp.id))
12
13                         #Join con subsets significativi
14                         INNER JOIN astral.astral_domain AS admn      #domino
15                         ON (admn.node_id = dm.id)
16                                  #subset  domini%
17                                  INNER JOIN astral.astral_domain_subset_id AS ss
18                                  ON (ss.astral_domain_id = admn.id)
19
20                            #selezione sequenza
21                             INNER JOIN astral.astral_seq AS seq    #sequenze
22                                   ON seq.id=admn.seq_id
23
24 WHERE pr.parent_node_id = @id_family     AND   #selezione della famiglia
25       pr.release_id = @rel_id            AND   #controllo della release famiglia
26       #Selezione livello %ID
27       ss.pct_identical = @pct_id         AND
28       (admn.style_id = 1 OR admn.style_id=3) AND
29       ss.release_id = @rel_id            AND   #Selezione release
30       admn.source_id = 2;                      #Selezione sequenze SEQRES
```

Figura 3-11: Query per la selezione dei domini di una data famiglia

Si effettuano tre self join nella tabella scop_node per la selezione dei pdb domain di una famiglia. In riga 07 la tabella scop_node serve per la selezione dei record corrispondenti ai nodi figli alla famiglia @id_family oggetto d'interrogazione. In riga 24 vi è la condizione per la selezione del nodo padre corrispondente alla famiglia. In tale modo si selezionano tutti i record di nodo padre @id_family, corrispondenti ai nodi di livello protein domain. In riga 08 si procede a selezionare tutti i nodi delle specie appartenenti alla famiglia. Alla fine in riga 10 si selezionano tutti i nodi di ultimo livello (pdb domain) corrispondenti alle sequenze.

Successivamente i nodi di dominio PDB vengono uniti alle informazioni ASTRAL contenute in astral_domain. Si filtrano in riga 30 solo le sequenze aventi come sorgente

SEQRES. Infine si effettua l'unione con la tabella astral_domain_subset_id per selezionare solo le sequenze del sotto-insieme di tipo % ID a livello @pct_id (riga 27). In riga 25 si effettua il filtro dei nodi scop appartenenti alla release voluta @rel_id ed infine in riga 28 si selezionano solo i domini di stile *single-chain* o *multi-chain genetic domain*.

Nel database a volte vi è un incongruenza tra lo stile di una sequenza specificato in astral_domain e quello specificato in astral_domain_subset_id. In particolare vi sono casi in cui seq_style per astral_domain è uguale a 1, mentre per astral_domain_subset_id e uguale sia a 2 che a 3.

Ad esempio, ponendo la clausola SELECT pari a:

```
SELECT admn.id, admn.sid, admn.style_id, ss.style_id , admn.source_id, dm.sccs
```

Si ottiene:

id	sid	style_id	style_id	source_id	scss
'7763817'	'd1b33a_'	'1'	'2'	'2'	'a.1.1.3'
'7763817'	'd1b33a_'	'1'	'2'	'2'	'a.1.1.3'
…	…	…	…	…	…

È perciò necessario specificare DISTINCT (riga 05) nell'interrogazione per evitare duplicati.

```
01 set @id_family       := 1076961;
02 set @pct_id          := 90;
03 set @rel_id          := 13;
04
05 SELECT DISTINCT admn.id ,admn.sid, admn.style_id, ssb.style_id , admn.source_id,
06          dm.sccs, seq.seq
07               FROM ((astral.scop_node AS pr                          #protein domain
08                         INNER JOIN astral.scop_node AS sp            #specie
09                         ON sp.parent_node_id = pr.id)
10                                  INNER JOIN astral.scop_node AS dm   #pdb domain
11                                  ON (dm.parent_node_id = sp.id))
12
13                         #Join con subsets significativi
14                         INNER JOIN astral.astral_domain AS admn      #domino
15                         ON (admn.node_id = dm.id)
16                           #subset domini blast
17                           INNER JOIN astral.astral_domain_subset_blast_e AS ssb
18                                 ON(ssb.astral_domain_id = admn.id)
19
20                            #selezione sequenza
21                             INNER JOIN astral.astral_seq AS seq    #sequenze
22                                   ON seq.id=admn.seq_id
23
24 WHERE pr.parent_node_id = @id_family     AND   #selezione della famiglia
25       pr.release_id = @rel_id            AND   #controllo della release famiglia
26       #Selezione E-value
27       ssb.blast_log10_e = @e_value       AND
28       (admn.style_id = 1 OR admn.style_id=3) AND
29       ssb.release_id = @rel_id           AND   #Selezione release
30       admn.source_id = 2;                      #Selezione sequenze SEQRES
```

Figura 3-12: Query per la selezione dei domini di una data famiglia per sottoinsiemi E-value

In figura Figura 3-12 vi è l'interrogazione per selezionare i domini di una famiglia appartenenti a un dato sotto-insieme significativo BLAST E-value.

Nelle righe 17-18 vi è il join con la tabella astral_domain_subset_blast_e contenente i sotto-insiemi significativi BLAST E-value.

Selezione delle sequenze della super-famiglia di un famiglia, escluse le appartenenti alla famiglia in oggetto

Si introduce un'interrogazione SQL per selezionare tutte le sequenze della super-famiglia di una data famiglia fornita come parametro @id_famiglia, e si escludono dai risultati le sequenze appartenenti alla famiglia stessa. Questa interrogazione è utile in quanto può essere utilizzata per ottenere l'insieme degli esempi di training positivi per il rilevamento delle omologie remote. In Figura 3-13 il codice di tale interrogazione.

```
01 set @id_family                := 1076961;
02 set @pct_id          := 90;
03 set @rel_id          := 13;
04
05 SELECT admn.id ,admn.sid, admn.style_id, dm.sccs, fa.id AS family_id, seq.seq
06         FROM ((((astral.scop_node AS qfa                    #query family
07               INNER JOIN astral.scop_node AS fa             #family
08               ON fa.parent_node_id = qfa.parent_node_id)    #si prendono famiglie
09                                                             #con stessa superfamiglia
10                  INNER JOIN astral.scop_node AS pr          #protein domain
11                  ON pr.parent_node_id = fa.id)
12                     INNER JOIN astral.scop_node AS sp       #specie
13                     ON sp.parent_node_id = pr.id)
14                        INNER JOIN astral.scop_node AS dm    #pdb domain
15                        ON dm.parent_node_id = sp.id)
16
17              #Join con subsets significativi
18            INNER JOIN astral.astral_domain AS admn         #domino
19            ON (admn.node_id = dm.id)
20                INNER JOIN astral.astral_domain_subset_id AS ss  #subset dei domini
21               ON (ss.astral_domain_id = admn.id)
22
23                   #selezione sequenza
24                   INNER JOIN astral.astral_seq AS seq     #sequenze
25                   ON seq.id=admn.seq_id
26
27 WHERE qfa.id= @id_family AND fa.id <> @id_family AND  #Selezione le altre famiglie
28          qfa.release_id = @rel_id                AND  #controllo release famiglia
29          #condizioni subset significativi
30          ss.pct_identical = @pct_id              AND
31         (admn.style_id = 1 OR admn.style_id=3)   AND
32         ss.release_id = @rel_id                  AND  #Selezione release
33         admn.source_id = 2;                           #Selezione sequenze
SEQRES
```

Figura 3-13: Selezione sequenze super-famiglia

Nella riga 06 si seleziona la famiglia d'interrogazione di ID @id_family dalla tabella scop_node. Successivamente si selezionano in riga 07 tutte le famiglie della stessa super-

famiglia di @id_family. In riga 27 infine si esclude la famiglia di interrogazione da tale insieme con la clausola fa.id <> @id_family. Le altre righe da 10 a 33 sono simili a quelle introdotte in 0 e servono a selezionare i domini PDB delle famiglie selezionate.

Selezione delle sequenze appartenenti ai fold complementari al fold di una data famiglia

Nella Figura 3-14 si presenta l'interrogazione per selezionare tutte le sequenze appartenenti ai fold diversi dal fold di una data famiglia @id_family.

```
01 set @id_family       := 1076961;
02set @pct_id            := 90;
03set @rel_id            := 13;
04
05 SELECT admn.id ,admn.sid, admn.style_id, dm.sccs, fa.id AS family_id, seq.seq
06          FROM ((((((astral.scop_node AS qfa                #query family
07                  #query super family , qsf.parent_node_id è il fold delle famiglia
08                  INNER JOIN astral.scop_node AS qsf
09                  ON qfa.parent_node_id = qsf.id)           #seleziono super family in sf
10                    INNER JOIN astral.scop_node AS sf
11                     #seleziono superfamiglie di un  altro fold
12                    ON sf.parent_node_id <> qsf.parent_node_id)
13                      INNER JOIN astral.scop_node AS fa       #family
14                      ON fa.parent_node_id = sf.id)
15                        INNER JOIN astral.scop_node AS pr         #protein domain
16                        ON pr.parent_node_id = fa.id)
17                            INNER JOIN astral.scop_node AS sp     #specie
18                            ON sp.parent_node_id = pr.id)
19                              INNER JOIN astral.scop_node AS dm   #pdb domain
20                              ON dm.parent_node_id = sp.id)
21
22           #Join con subsets significativi
23           INNER JOIN astral.astral_domain AS admn            #domino
24           ON (admn.node_id = dm.id)
25                INNER JOIN astral.astral_domain_subset_id AS ss    #subset dei domini
26               ON (ss.astral_domain_id = admn.id)
27
28                        #selezione sequenza
29                        INNER JOIN astral.astral_seq AS seq     #sequenze
30                        ON seq.id=admn.seq_id
31
32  #Selezione nodi di livello 4, cioè superfamiglie
33  WHERE sf.level_id = 4                          AND
34          qfa.id = @id_family                    AND
35          qfa.release_id = @rel_id               AND   #controllo release famiglia
36          #condizioni subset significativi
37          ss.pct_identical = @pct_id                   AND
38          (admn.style_id = 1 OR admn.style_id=3)       AND
39          ss.release_id = @rel_id                      AND   #Seleziono release
40          admn.source_id = 2;                                #Selezione sequenze
```

Figura 3-14: Selezione domini dei fold complementari a un fold di una data famiglia

Questa interrogazione è utile poiché può essere utilizzata per ottenere l'insieme degli esempi di test e train negativi per il problema del rilevamento delle omologie remote.

In riga 06 si seleziona la famiglia oggetto d'interrogazione, mentre in riga 09 si ottiene la superfamiglia della famiglia selezionata. Infine da tale superfamiglia si estrae il fold in riga 12 con qsf.parent_node_id. In riga 10 si selezionano tutte le superfamiglie, filtrano i nodi di superfamiglia grazie alla condizione in riga 33. Successivamente dal loro insieme si escludono tutte le superfamiglie appartenenti al fold della famiglia @id_family, tramite la condizione di riga 12. La rimanente parte dell'interrogazione (righe 13-40) è simile a quella introdotta in 0.

3.6 Sottoinsiemi significativi reperibili in rete

3.6.1 SCOP 1.37 per test Fisher Kernel

Nell'articolo [23] è introdotto un metodo per classificare le proteine mediante SVM. Esso utilizza un sotto-insieme delle famiglie del database SCOP 1.37, reperibile all'indirizzo [25]. Tale database è costituito da file di testo in formato FASTA contenenti i domini PDB e le relative sequenze. Vi sono 33 famiglie SCOP, per ognuna delle quali vi sono file per gli esempi di training positivi e negativi e gli esempi di test positivi e negativi. Nella radice del database vi sono le cartelle che dividono le famiglie in base agli identificativi sccs dei fold, e in ognuna di esse vi sono le cartelle delle famiglie di nome pari al sccs di famiglia. Le cartelle per i fold contengono gli esempi di test e training negativi, essi sono divisi in due insiemi chiamati fold0 e fold1. La divisione fold1 è l'inverso di fold0, cioè gli esempi di test in fold0 sono gli esempi di train in fold1 . Infine le cartelle nelle famiglie contengono gli esempi di test e train positivi.

Per esempio, nella cartella fold 3.25:

- 3.25.1.1/ - Contiene i file per i membri della famiglia 3.25.1.1.
- 3.25.1.3/ - Contiene i file per i membri della famiglia 3.25.1.3.
- 3.25.fold0.neg-train.seq - fold0 division: contiene le sequenze di train negative
- 3.25.fold0.neg-test.seq - fold0 division: contiene le sequenze test negative.
- 3.25.fold1.neg-train.seq - contiene le sequenze di train negative, è un collegamento a 3.25.fold0.neg-test.seq.
- 3.25.fold1.neg-test.seq - fold1 contiene le sequenze test negative, è un collegamento a 3.25.fold0.neg-train.seq.

3.6.2 SCOP 1.53 per test Mismatch Kernel

Nell'articolo [2] viene utilizzato un particolare sottoinsieme della versione SCOP 1.53 per effettuare i test di classificazione di proteine mediante SVM. Tale database è reperibile all'indirizzo [26].

Il database è stato elaborato e messo nel formato del database presentato in 3.6.2 in modo che sia più facilmente accessibile ed utilizzabile via software. Il codice che crea il database in tale formato, a partire dal formato originale fornito in [26] è incluso nel software utilizzato per i test, introdotto nel capitolo 6. Esso è nella classe scopDbExplorer/Scop153Parser.cpp, alla quale prima dell'elaborazione si forniscono i file che descrivono il dataset e la cartella per il database prodotto.

L'output produce solo file di sequenze d'insieme fold0, in quanto in SCOP 1.53 non è prevista l'inversione del train e test set negativi per effettuare i test.

4 Approximate Mismatch kernel adattivo con componenti pesati

La maggior parte degli approcci per il problema della classificazione delle proteine, introdotti in 1.1.2, sono ***generativi***. Cioè le metodologie implicano la costruzione di un modello per una singola famiglia di proteine e poi valutano ogni sequenza candidata per vedere quanto essa si adatta al modello. Se l'adattamento supera un certo punteggio di soglia, allora la sequenza è classificata come appartenente alla famiglia.

Gli ***approcci discriminativi*** invece gestiscono le sequenze di proteine come insiemi di esempi etichettati positivamente se appartengono ad una data famiglia, negativamente altrimenti. Successivamente un algoritmo di apprendimento automatico, come quelli visti in 1.2, cerca di imparare a discriminare tra gli esempi delle differenti classi. Sia gli esempi positivi che negativi sono usati in tale approccio, mentre gli approcci generativi possono solo usare esempi di training positivi. Tra gli approcci discriminativi più di successo per il rilevamento dell'omologia remota tra proteine, si citano metodi basati su Hidden Markov Model [27], su conteggio di parole esatte (Spectrum Kernel) [15] e su conteggio di parole con mismatch (Mismatch kernel) [15].

In particolare questi ultimi due sono computazionalmente molto più efficienti del primo, penalizzato dal calcolo HMM. Esso richiede tempo quadratico nella lunghezza delle sequenze mentre lo spectrum kernel richiede tempo lineare, e il mismatch kernel richiede tempo proporzionale alla lunghezza della stringa e a $m^k l^k$, con m lunghezza delle tuple, k il numero massimo di mismatch ammessi e l dimensione dell'alfabeto.

Nel seguito s'introduce la notazione utilizzata e si descrivono i due kernel sovracitati, che sono all'origine dei metodi proposti in questa tesi. Successivamente si descrive il problema del rilevamento delle omologie remote tra proteine tramite SVM, che è stato utilizzato per valutare le loro prestazioni, riportate nel seguito.

Infine si introduce un nuovo tipo di kernel, chiamato ***Weighted Approximate Mismatch Kernel***, che costituisce un'evoluzione del mismatch kernel. Esso presenta alcune modifiche che lo rendono più preciso del mismatch kernel per molti test di omologia remota, inoltre il suo tempo di calcolo è indipendente dalla lunghezza delle m-tuple (risoluzione m) e dal numero massimo k di mismatch consentiti, dipendendo solo dalla lunghezza delle stringhe A e B ad esso in ingresso. Ciò costituisce un netto miglioramento nei tempi di calcolo rispetto al Mismatch Kernel, che peggiora notevolmente all'aumentare di m e k.

4.1 Notazione utilizzata

Dato un alfabeto $\sum$, siano $A = a_1a_2...a_{l_A}$ e $B = b_1b_2...b_{l_B}$ due sequenze con caratteri appartenenti all'alfabeto $\sum$. Si indichi come $L = |\sum|$ la cardinalità dell'alfabeto. Sia m la lunghezza delle tuple considerate e k il massimo numero di mismatch ammessi. Sia $n_A(w,k)$ il numero di occorrenze della stringa w in A con esattamente k mismatch. Sia $N_A(w,k) = \sum_{i=0}^{k} n_A(w,i)$ il numero di occorrenze di w in A con al massimo k mismatch. Siano $n_B(w,k)$ e $N_B(w,k)$ definite in modo analogo.

Sia T l'insieme di tutte le possibili stringhe di lunghezza m (m-mer o m-tuple) sull'alfabeto $\sum$, cioè $T = \sum^m$, con $M = L^m$ si ha

$$T = \{w_1, w_2, \ldots, w_M\}$$

Con $w_1 < w_2 < \ldots < w_M$, cioè le tuple sono in ordine lessicografico crescente.

Sia T_A l'insieme delle stringhe di lunghezza m, $w \in \sum^m$, che occorrono solo in A.

Sia T_B l'insieme delle stringhe di lunghezza m, $w \in \sum^m$, che occorrono solo in B. Sia $T_{A \cap B}$ l'insieme delle stringhe di lunghezza m, $w \in \sum^m$, che occorrono sia in A che in B.

Le tuple in A possono essere anche indicate come $w_A(i,m)$ per $1 \leq i \leq l_A - m + 1$, dove i indica l'indice iniziale in A della tupla w e $l_A - m + 1$ è il numero totale di tuple. L'insieme delle tuple in A di lunghezza m, distinte per posizione, è $S_A = \{w_A(0,m), w_A(1,m), w_A(2,m), ..., w_A(l_A - m + 1,m)\}$.

Le tuple in B possono essere anche indicate come $w_B(i,m)$ per $1 \leq i \leq l_B - m + 1$, dove i indica l'indice iniziale in B della tupla w e $l_B - m + 1$ è il numero totale di tuple. L'insieme delle tuple in A di lunghezza m, distinte per posizione, è $S_B = \{w_B(0,m), w_B(1,m), w_B(2,m), ..., w_B(l_B - m + 1,m)\}$.

4.2 Mismatch kernel

Il mismatch kernel è basato su una feature map che porta a uno spazio vettoriale caratterizzato da tutte le possibili sotto-sequenze di amminoacidi di lunghezza fissata m. Ogni m-tupla in una sequenza di input contribuisce a tutte le features che differiscono da essa per al massimo k mismatch. Esso aggiunge l'idea biologicamente importante del mismatching al spectrum kernel. E' stato presentato in [15].

Il mismatch kernel è stato testato in contesti di rilevamento delle omologie remote tramite SVM, ma può essere usato anche in altri problemi di classificazione basati sulle sequenze.

Il Mismatch kernel MK può essere definito come:

$$MK = \sum_{w \in T} N_A(w,k)\, N_B(w,k)$$

Il kernel può essere riscritto come $MK = K_A + K_B + K_I + K_O$, dove:

$$K_A = \sum_{w \in T_A} N_A(w,k)\, N_B(w,k)$$

$$K_B = \sum_{w \in T_B} N_A(w,k)\, N_B(w,k)$$

$$K_I = \sum_{w \in T_{A \cap B}} N_A(w,k)\, N_B(w,k)$$

$$K_O = \sum_{w \in \{T - T_{A \cap B} - T_A - T_B\}} N_A(w,k)\, N_B(w,k)$$

Cioè è la somma dei componenti associati alle m-tuple che, in ordine, appartengono solo ad A, solo a B, sia ad A e B e infine che non appartengono né ad A né a B.

Il contributo di una singola stringa w può essere scritto come:

$$CS(w,k) = N_A(w,k) N_B(w,k) = \sum_{i=0}^{k} \sum_{j=0}^{k} n_A(w,i)\, n_B(w,j)$$

Il kernel MK può anche essere riscritto come:

$$MK = \sum_{w \in T} CS(w,k) = \sum_{i=0}^{k} \sum_{j=0}^{k} \left(\sum_{w \in T_A} n_A(w,i) n_B(w,j) \right) = \sum_{i=0}^{k} \sum_{j=0}^{k} (C(i,j))$$

Dove ***C(i,j)*** è il ***contributo*** di tutte le stringhe w ϵ T, considerando solo le sue occorrenze con esattamente *i* mismatch in A e con esattamente j mismatch in B.

Tale kernel è stato implementato efficientemente per bassi valori di mismatch k tramite una struttura dati chiamata mismatch tree data structure. Tale struttura è molto efficiente per k = 0,1 ma per valori superiori è risultata molto lenta, in particolare all'aumentare di

m. La complessità computazionale, per il calcolo di un'intera matrice di kerkel di M sequenze è pari a $O(M^2nm^kl^k)$, con n lunghezza massima delle sequenze.

E' stato testato che è utilizzabile efficientemente per k=1 e m>=2, mentre per k=2, già con m=5 risulta lenta. Questo perché da k=1 a k=2 la complessità aumenta di un termine ml. Le performance di calcolo sperimentali saranno analizzate nella sezione 7.2.2.

L'implementazione viene descritta in [15].

4.2.1 Considerazioni sul metodo

Considerando il mismatch kernel per k = 1, il kernel K è costituito dai contributi per i,j=0,1, cioè:

$$MK = \sum_{i=0}^{1}\sum_{j=0}^{1}\big(C(i,j)\big) = C(0,0) + C(0,1) + C(0,1) + C(1,1)$$

E' da notare che il contributo C(1,1) porta del "bias" nel calcolo del kernel. Si consideri una tupla w che può esser presente in A e B, solo in una delle due sequenze o in nessuna delle due. Ad esempio sia w = abc. Nel seguito si dà un voto all'allineamento tra una tupla w_A in A e una tupla w_B in B, confrontando le due sequenze tra caratteri nelle posizioni corrispondenti. Si possono avere i seguenti casi:

1. in A c'è *w* con una mutazione genetica, ad esempio w_A = ab<u>d</u>. In B la mutazione genetica è uguale w_B = abd. In tale caso l'allineamento si può considerare "perfetto".
2. in A c'è *w* con una mutazione genetica. In B la mutazione genetica è nella stessa posizione, con un Amminoacido (AA) funzionalmente equivalente. In tale caso il matching si può considerare "eccellente".
3. in A c'è w con una mutazione genetica. In B la mutazione genetica è nella stessa posizione, con un AA <u>non</u> funzionalmente equivalente. In tale caso le due tuple potrebbero comunque essere relazionate, essendo una delle due la mutazione genetica dell'altra. L'allineamento si può considerare "più che discreto discreto".
4. in A c'è *w* con una mutazione genetica, ad esempio w_A = ab<u>d</u>. In B la mutazione genetica è in una posizione diversa w_B = <u>z</u>bc. Si hanno due mutazioni e si possono avere i seguenti casi:
    a. La posizione della mutazione di *w* in A, ha il carattere, il terzo in questo caso, che è uguale o funzionalmente equivalente a quello di *w*, che è presente anche in B. In tal caso, considerando solo il mismatch per il terzo carattere, il matching si può considerare "eccellente".
        i. La posizione della mutazione di *w* in B, ha il carattere, il primo in questo caso, che è uguale o funzionalmente equivalente a quello di T, che è presente anche in A. In tal caso per il primo carattere l'allineamento è "eccellente". Nel complesso, le tuple in A e B sono funzionalmente identiche. In tale caso il matching totale può considerarsi "eccellente".
        ii. La posizione della mutazione di *w* in B, ha il carattere, il primo nel nostro caso, che <u>non</u> è funzionalmente equivalente a quello di *w*, che è presente anche in A. In tal caso per il primo carattere l'allineamento è "discreto". Il matching totale può considerarsi "discreto".
    b. La posizione della mutazione di *w* in A, ha il carattere, il terzo nel nostro caso, che è *non* funzionalmente equivalente a quello di *w*. Il terzo carattere in B è uguale a quello di *w*. In tal caso, considerando solo il mismatch per il terzo carattere, il matching è "più che discreto".
        i. La posizione della mutazione di *w* in B, ha il carattere, il primo nel nostro caso, che è equivalente o funzionalmente equivalente a quello di *w*, che è presente anche in A. In tal caso per il primo

carattere l'allineamento è "eccellente". In tale caso il matching totale può considerarsi "discreto".

ii. La posizione della mutazione di *w* in B, ha il carattere, il primo nel nostro caso, che non è funzionalmente equivalente a quello di *w*, che è presente anche in A. In tal caso per il primo carattere il matching può considerarsi "discreto". Il matching totale può considerarsi "scarso".

Si può notare che nel caso 4.a.i le mutazioni in A e B possono portare ad una stessa tupla, cioè $w_N = w_A = w_B \neq w$. In tal caso il contributo di tale tupla porta ad un bias nel valore del kernel, in quanto w_N corrisponde ad un'altra tupla di T, cioè ad un'altra dimensione dello spazio delle features, per la quale vi sono corrispondenze in A e B senza mismatch.

4.3 Spectrum Kernel

Lo spectrum kernel è uno string kernel pensato per essere utilizzato con le SVM per la classificazione mediante approccio discriminativo delle proteine. E' stato introdotto in [15]. Esso è molto semplice, efficiente da calcolare e utilizza un approccio discriminativo. Tale kernel è definito come il mismatch kernel per k=0. Cioè non vengono ammessi mismatch nel conteggio delle frequenze delle tuple.

4.4 Rilevamento delle omologie remote

Il problema del rilevamento delle omologie remote (Remote homology detection) è il problema del rilevamento delle omologie tra proteine nel caso di bassa similarità tra sequenze. E' un problema computazionalmente difficile, e non esiste un approccio che funziona bene in tutti i casi [28].

Spectrum kernel e mismatch kernel sono stati testati usando un esperimento progettato da Jakkola et al., descritto in [27] , per il rilevamento delle omologie remote.

Sia F una famiglia SCOP oggetto del test.

L'omologia remota è simulata prendendo come esempi di training positivi le sequenze appartenenti alla superfamiglia di F, escludendo tutte le sequenze che appartengono a F.

dalla sua superfamiglia. Gli esempi di test e training negativi sono scelti all'esterno del fold di F. Infine gli esempi di test positivi sono costituiti dalle sequenze appartenenti ad F.

In Figura 4-1 sono illustrati graficamente gli insiemi per il test.

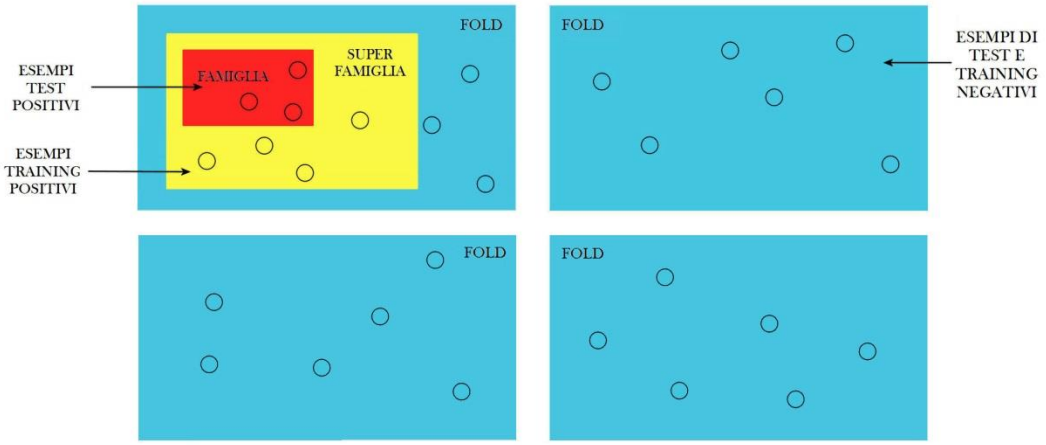


Figura 4-1: Insiemi per il test del rilevamento remoto delle omologie

4.5 Prestazioni dello spectrum kernel e mismatch kernel

4.5.1 Test con database SCOP 1.37

Lo spectrum kernel e il mismatch kernel sono stati inizialmente testati allenando una SVM su un insieme di dati ricavati dal database SCOP 1.37, rispettivamente negli articoli [15] e [15]. Il dataset è attualmente reperibile all'indirizzo [25] ed è descritto nell'articolo [27].

I test sono stati effettati sia per i valori base del kernel, che per il kernel normalizzato, definito come:

$$K_m^{Norm} = \frac{K_m(x,y)}{\sqrt{K_m(x,x)}\sqrt{K_m(y,y)}}$$

Il software SVM utilizzato è **GIST SVM**, reperibile all'indirizzo [29] esso verrà descritto più nel dettaglio in 7.1**.**

Nel seguito si riportano le figure tratte da [15], riportanti le performance dei metodi di classificazione.

La Figura 4-2 mostra le prestazioni del metodo mismatch-SVM relativi a tre metodi di rilevamento delle omologie esistenti, misurati come punteggi ROC. La figura include i risultati per 33 famiglie SCOP, e ciascuna serie corrisponde a un metodo. Ogni curva mostra il numero totale di famiglie per le quali il metodo associato eccede una un

punteggio ROC soglia. Si può vedere che le curve Fisher-SVM e mismatch-SVM sono molto simili e hanno performance migliori di SAM-T98 e molto migliori del metodo PSI-BLAST.

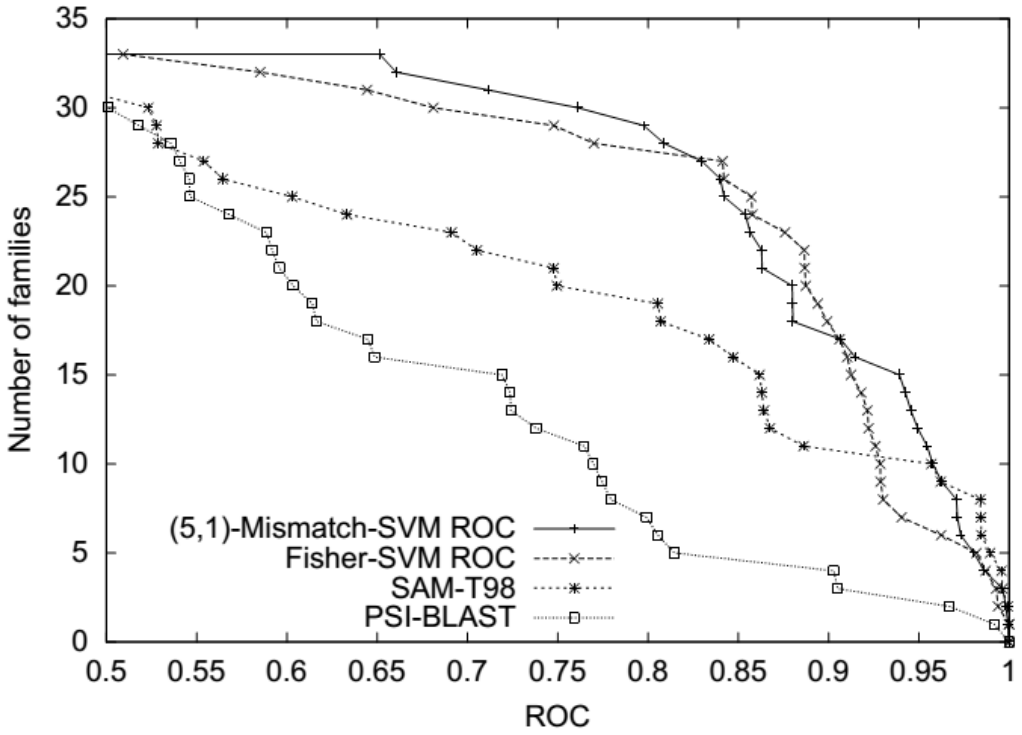


Figura 4-2: Confronto di 4 metodi di rilevamento delle omologie.

La figura Figura 4-3 mostra un confronto famiglia per famiglia delle performance del (5,1)-mismatch kernel rispetto al Fisher-SVM. Nel grafico, i punti cadono approsimativamente equamente sia sopra che sotto la diagonale, indicando piccola differenza nelle performance dei due metodi.

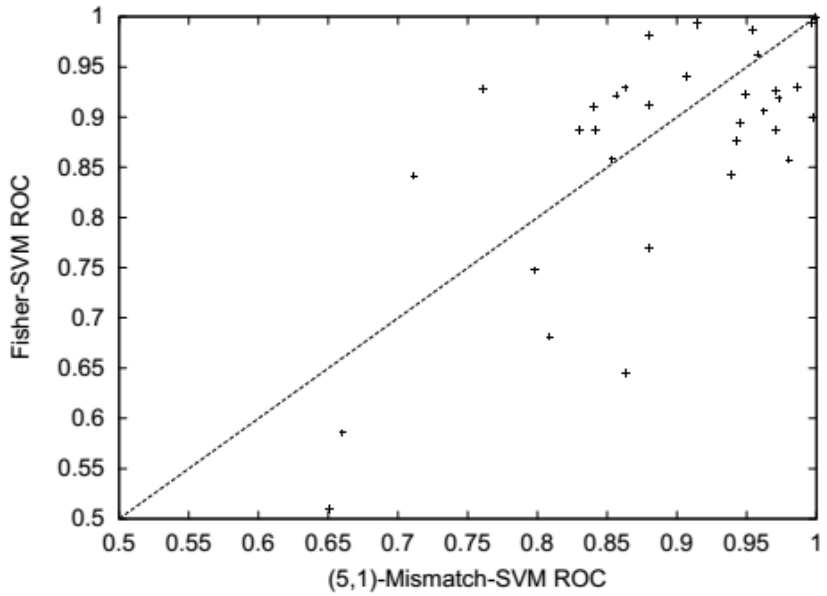


Figura 4-3: Confronto famiglia-per-famiglia tra mismatch kernel e Fisher kernel

La figura ** effettua il confronto famiglia per famiglia tra (5,1)-mismatch kernel e m=3 spectrum kernel. Si può vedere che il mismatch kernel porta dei notevoli miglioramenti rispetto a quest'ultimo, infatti la maggior parte dei punti cadono sotto la diagonale, indicando che i punteggi ROC sono maggiori per il mismatch kernel.

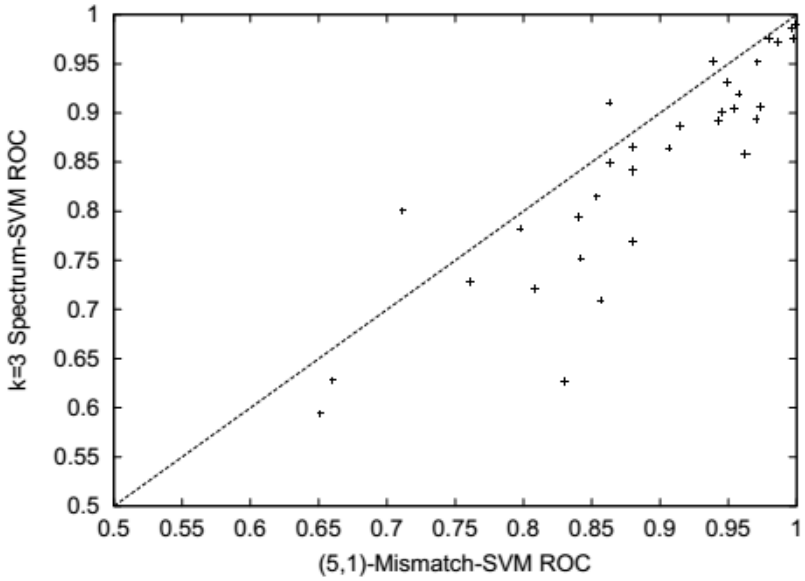


Figura 4-4: Confronto famiglia-per-famiglia tra mismatch kernel e spectrum kernel

4.5.2 Test con database SCOP 1.53

il mismatch kernel è stato successivamente testato allenando l'SVM su un insieme di dati ricavati dal database SCOP 1.53, i test sono descritti nell'articolo [2]. Il dataset è attualmente reperibile all'indirizzo [26] ed è descritto in [30]. Di seguito si riportano brevemente i risultati. Le sequenze per gli esperimenti sono estratte dal database SCOP 1.53 rimuovendo le sequenze simili usando una soglia di E-value di 10^{-25}. Tale procedura porta a 4352 sequenze distinte raggruppate in famiglie, superfamiglie e fold.

In Figura 4-5 vi è il confronto dei metodi di rilevamento dell'omologia remota per il dataset di benchmark ricavato da SCOP 1.53.

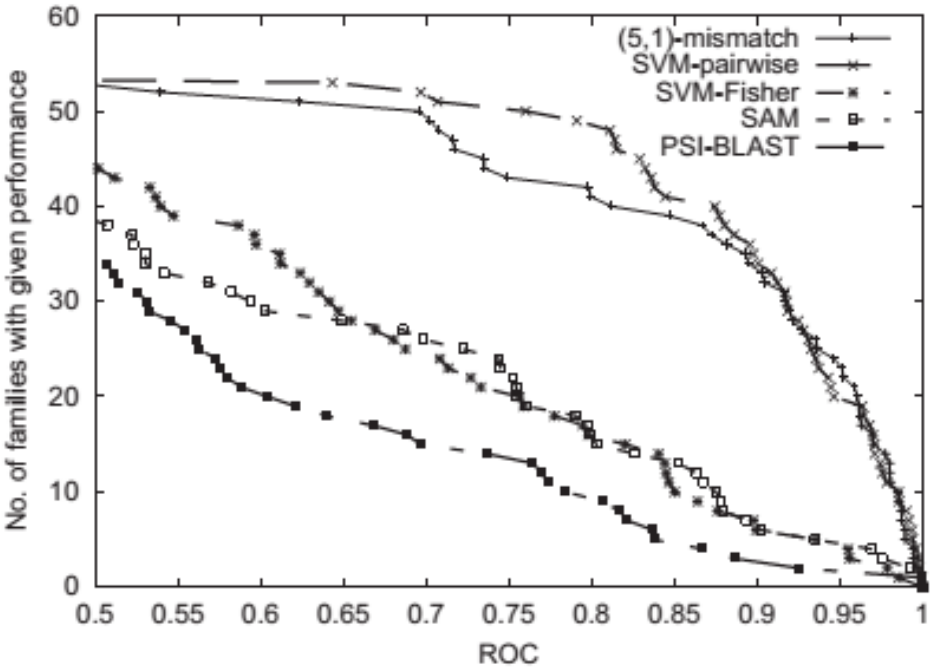


Figura 4-5: Confronto di 5 metodi di rilevamento delle omologie.

Il grafico traccia il numero totale di famiglie per le quali un dato metodo eccede un punteggio ROC di soglia. Ogni serie corrisponde a un metodo di rilevamento dell'omologia remota [2]. Si può osservare che il metodo SVM-pairwise descritto in [30] ha performance molto superiori ai metodi SVM-Fisher, SAM-T98 e PSI-BLAST. Il metodo mismatch kernel con parametri m=5, k=1 invece ha performance confrontabili con SVM-pairwise. Inoltre esso ha il vantaggio di esser molto più veloce da calcolare rispetto a SVM-pairwise. In figura Figura 4-6 vi è il confronto famiglia per famiglia tra il mismatch kernel (5,1) e SVM-pairwise. In essa si può chiaramente vedere che i due metodi hanno performance confrontabili.

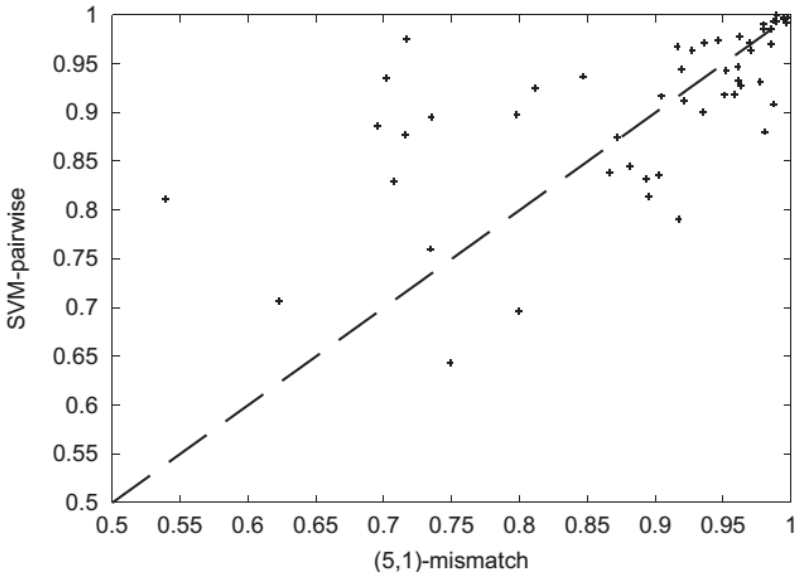


Figura 4-6: Confronto famiglia-per-famiglia tra mismatch-(5,1) e SVM-pairwise

4.6 Approximate Mismatch Kernel

Nel seguito definiamo il kernel ***Approximate Mismatch Kernel***. Esso è definito come un'approssimazione del Mismatch Kernel in quanto non contiene il termine K_O introdotto in 4.2, contributo al valore del kernel dato dalle tuple che non appartengono né ad A né a B.

Il kernel Approximate Mismatch (AMK) può essere definito semplicemente come:

$$AMK = \sum_{w \in T_A \cup T_B \cup T_{A \cap B}} N_A(w,k)\, N_B(w,k)$$

Il kernel può essere riscritto come AMK = K_A + K_B + K_I, dove:

$$K_A = \sum_{w \in T_A} N_A(w,k)\, N_B(w,k)$$

$$K_B = \sum_{w \in T_B} N_A(w,k)\, N_B(w,k)$$

$$K_I = \sum_{w \in T_{A \cap B}} N_A(w,k)\, N_B(w,k)$$

Si può notare che Mismatch Kernel è definito come MK = K_A + K_B + K_I + K_O e possiede il termine K_O , mancante in AMK.

Il contributo di una singola stringa w può essere scritto come:

$$CS(w,k) = N_A(w,k) N_B(w,k) = \sum_{i=0}^{k} \sum_{j=0}^{k} n_A(w,i)\, n_B(w,j)$$

Ad esempio per k= 2 si ottiene:

$n_A(w,0)n_B(w,0) + n_A(w,0)n_B(w,1) + n_A(w,0)n_B(w,2) + n_A(w,1)n_B(w,0) + n_A(w,1)n_B(w,1) + n_A(w,1)n_B(w,2) + n_A(w,2)n_B(w,0) + n_A(w,2)n_B(w,1) + n_A(w,2)n_B(w,2)$

Il kernel può anche essere riscritto come:

$$K_A = \sum_{w \in T_A} CS(w,k) = \sum_{i=0}^{k} \sum_{j=0}^{k} \left(\sum_{w \in T_A} n_A(w,i) n_B(w,j) \right) = \sum_{i=0}^{k} \sum_{j=0}^{k} (C_A(i,j))$$

$$K_B = \sum_{w \in T_B} CS(w,k) = \sum_{i=0}^{k} \sum_{j=0}^{k} \left(\sum_{w \in T_B} n_A(w,j) n_B(w,i) \right) = \sum_{i=0}^{k} \sum_{j=0}^{k} (C_B(i,j))$$

$$K_I = \sum_{w \in T_{A \cap B}} CS(w,k) = \sum_{i=0}^{k} \sum_{j=0}^{k} \left(\sum_{w \in T_{A \cap B}} n_A(w,i) n_B(w,j) \right) = \sum_{i=0}^{k} \sum_{j=0}^{k} (C_I(i,j))$$

Dove $C_A(i,j)$ è il ***contributo*** di tutte le stringhe w ∈ T_A, considerando solo le occorrenze con esattamente *i* mismatch in A e con esattamente *j* mismatch in B. $C_B(i,j)$ è il ***contributo*** di tutte le stringhe $w \in T_B$, considerando solo le occorrenze con esattamente *i* mismatch in B e con esattamente *j* mismatch in A. $C_I(i,j)$ è il ***contributo*** di tutte le stringhe w ∈ $T_{A \cap B}$, considerando solo le occorrenze con esattamente *i* mismatch in A e con esattamente *j* mismatch in B. Nelle formule prestare attenzione all'inversione di indici i, j tra K_A e K_B per i termini n_A e n_B, tale inversione di indici, che nel calcolo del

valore di K_B è superflua, sarà successivamente necessaria e motivata quando si introdurranno i pesi ai vari contributi. In altre parole, per C_A il primo indice si riferisce al numero di mismatch delle stringhe $w \in T_A$ nella sequenza A, sequenza in cui appaiono sicuramente esattamente almeno una volta. Per C_B il primo indice si riferisce al numero di mismatch delle stringhe $w \in T_B$ nella sequenza B, sequenza in cui appaiono sicuramente esattamente almeno una volta. Per $C_{A\cap B}$ il primo indice si riferisce al numero di mismatch delle stringhe $w \in T_{A\cap B}$ nella sequenza A.

4.7 Weighted Approximate Mismatch Kernel

Definiamo ***Weighted Approximate Mismatch Kernel*** (WAMK) come una variante del kernel AMK nel quale si introducono dei pesi per i contributi $C_x(i,j)$.

Tali pesi vengono indicati con *H(i,j)* per i,j = 0,...,k.

Il contributo di una singola stringa $w \in T_A$ al valore del kernel può essere scritto come:

$$C_{WA}(w,k) = \sum_{i=0}^{k}\sum_{j=0}^{k} H(i,j)\big(n_A(w,i)n_B(w,j)\big)$$

Il contributo di una singola stringa $w \in T_B$ al valore del kernel può essere scritto come:

$$C_{WB}(w,k) = \sum_{i=0}^{k}\sum_{j=0}^{k} H(i,j)\big(n_A(w,j)n_B(w,i)\big)$$

Il contributo di una singola stringa $w \in T_{A\cap B}$ al valore del kernel può essere scritto come:

$$C_{WI}(w,k) = \sum_{i=0}^{k}\sum_{j=0}^{k} H(i,j)\big(n_A(w,i)n_B(w,j)\big)$$

Il Weighted Approximate Mismatch (WAMK) può essere definito come $WAMK = K'_A + K'_B + K'_I$, dove:

$$K'_A = \sum_{w\,\epsilon\,T_A} C_{WA}(w,k) = \sum_{i=0}^{k}\sum_{j=0}^{k}\left(\sum_{w\,\epsilon\,T_A} H(i,j)\big(n_A(w,i)n_B(w,j)\big)\right) = \sum_{i=0}^{k}\sum_{j=0}^{k} H(i,j)(C_A(i,j))$$

$$K'_B = \sum_{w\,\epsilon\,T_B} C_{WB}(w,k) = \sum_{i=0}^{k}\sum_{j=0}^{k}\left(\sum_{w\,\epsilon\,T_A} H(i,j)\big(n_A(w,j)n_B(w,i)\big)\right) = \sum_{i=0}^{k}\sum_{j=0}^{k} H(i,j)(C_B(i,j))$$

$$K'_I = \sum_{w\,\epsilon\,T_{A\cap B}} C_{WI}(w,k) = \sum_{i=0}^{k}\sum_{j=0}^{k}\left(\sum_{w\,\epsilon\,T_{A\cap B}} H(i,j)\big(n_A(w,i)n_B(w,j)\big)\right) = \sum_{i=0}^{k}\sum_{j=0}^{k} H(i,j)(C_I(i,j))$$

Il valore finale del kernel è dato da:

$$WAMK = \sum_{i=0}^{k}\sum_{i=0}^{k}\big(H(i,j)(C_I(i,j) + C_A(i,j) + C_B(i,j)\big)$$

L'inversione degli indici nel contributo $C_{WB}(w,k)$ è necessario per garantire la coerenza nell'assegnazione dei pesi. In tale modo presa una qualunque tupla *w*, l'indice i indica sempre il numero di mismatch nella sequenza in cui *w* appare esattamente (senza mismatch) almeno una volta, nel caso in cui *w* appaia o solo in A o solo in B.

4.8 Weighted Approximate Mismatch Kernel con funzione sul numero di mismatch

E' possibile modificare la definizione di kernel WAMK introducendo una funzione nel seguente modo. Nel kernel WAMK il contributo di una stringa w (per il caso di $w \in T_A$) è dato da:

$$C_{WA}(w,k) = \sum_{i=0}^{k}\sum_{j=0}^{k} H(i,j)\big(n_A(w,i)n_B(w,j)\big)$$

I termini $n_A(w,i)$ e $n_B(w,j)$ contribuiscono per un fattore F:

$$F = n_A(w,i) * n_B(w,j)$$

Si sostituisce tale fattore con una funzione di $n_A(w,i)$ e $n_B(w,j)$, cosa che permetterà di definire il kernel in modo più flessibile.

Si consideri la generica funzione ***KFUNCT(v,p)***, dove v e p sono due parametri interi.

Si definisce il kernel ***Function-WAMK*** come $FWAMK = K''_A + K''_B + K''_I$, con:

$$K''_A = \sum_{w \in T_A} C_{FWA}(w,k) = \sum_{i=0}^{k}\sum_{j=0}^{k} H(i,j)\left(\sum_{w \in T_A} KFUNCT\big(n_A(w,i), n_B(w,j)\big)\right) = \sum_{i=0}^{k}\sum_{j=0}^{k} H(i,j)(C_{FA}(i,j))$$

$$K''_B = \sum_{w \in T_B} C_{FWB}(w,k) = \sum_{i=0}^{k}\sum_{j=0}^{k} H(i,j)\left(\sum_{w \in T_A} KFUNCT\big(n_A(w,j), n_B(w,i)\big)\right) = \sum_{i=0}^{k}\sum_{j=0}^{k} H(i,j)(C_{FB}(i,j))$$

$$K''_I = \sum_{w \in T_{A\cap B}} C_{FWI}(w,k) = \\ = \sum_{i=0}^{k}\sum_{j=0}^{k} H(i,j)\left(\sum_{w \in T_{A\cap B}} KFUNCT\big(n_A(w,i), n_B(w,j)\big)\right) = \sum_{i=0}^{k}\sum_{j=0}^{k} H(i,j)(C_{FI}(i,j))$$

Si noti che per:

- H(i,j) = 1, $1 \le i,j \le k$ e KFUNCT($n_A(w,i)$, $n_B(w,j)$) = $n_A(w,i) * n_B(w,j)$ si ottiene l'Approximate Mismatch Kernel.
- KFUNCT($n_A(w,i)$,, $n_B(w,j)$) = $n_A(w,i)$, * $n_B(w,j)$ si ottiene il Weighted Approximate Mismatch Kernel.

4.9 Kernel combinatorio senza ordine

Il kernel combinatorio senza ordine è un FWAMK la cui funzione è definita come:

$$KFUNCT(a,b) = UCOMB(a,b) = \sum_{i=0}^{\min(a,b)} (\max(a,b) - i)$$

Con a = $n_A(w,i)$ e b = $n_B(w,j)$.

Le combinazioni enumerano tutte le possibili corrispondenze senza ordine tra le tuple presenti in A e in B che differiscono da una data tupla *w* per:

- i mismatch in A e j mismatch in B nel caso di tuple in T_A
- i mismatch in B e j mismatch in A nel caso di tuple in T_B
- i mismatch in A e j mismatch in B nel caso di tuple in T_I.

In altri termini, presa una tupla $w \in T_A$, essa appare $n_A(w,i)$ volte con esattamente i mismatch in A e $n_B(w,j)$ volte con esattamente j mismatch in B. Si considerano tutte le possibili corrispondenze tra le $n_A(w,i)$ tuple e le $n_B(w,j)$ tuple, senza tenere conto dell'ordine.

Ad esempio siano A = "aaafghbfghcfgh" e B = "ccfgaaafgbb" , per il mer w = "fgh" si ha $n_A(w,0) = 3$ e $n_B(w,1) = 2$. Le possibili associazioni sono UCOMB(3,2) = 6 e sono elencate in Tabella 4-1.

aaa**fgh**bfghcfgh	cc**fga**aafgbb
aaa**fgh**bfghcfgh	ccfgaaa**fgb**b
aaa**fgh**bfghcfgh	cc**fga**aafgbb
aaa**fgh**bfghcfgh	ccfgaaa**fgb**b
aaafghb**fgh**cfgh	cc**fga**aafgbb
aaafghb**fgh**cfgh	ccfgaaa**fgb**b

Tabella 4-1: Esempio di combinazioni senza ordine

4.10 Kernel combinatorio con ordine

Il kernel combinatorio senza ordine è un FWAMK la cui funzione è definita come:

$$KFUNCT(a,b) = COMB(a,b) = \begin{pmatrix} \max(a,b) \\ \\ \min(a,b) \end{pmatrix}$$

Le combinazioni enumerano tutte le possibili corrispondenze *con* ordine tra le tuple presenti in A e in B che differiscono da una data tupla *w* per:

- i mismatch in A e j mismatch in B nel caso di tuple in T_A
- i mismatch in B e j mismatch in A nel caso di tuple in T_B
- i mismatch in A e j mismatch in B nel caso di tuple in $T_{A\cap B}$.

In altri termini, presa una tupla $w \in T_A$, essa appare $n_A(w,i)$ volte con esattamente i mismatch in A e $n_B(w,j)$ volte con esattamente j mismatch in B. Si considerano tutte le possibili corrispondenze tra le $n_A(w,i)$ tuple e le $n_B(w,j)$ tuple, considerando l'ordine.

Ad esempio siano A = "aaafghbfghcfgh" e B = "ccfgaaafgbb" , per il mer w = "fgh" si ha $n_A(w,0) = 3$ e $n_B(w,1) = 2$. Le possibili associazioni sono COMB(3,2) = 3 e sono elencate in Tabella 4-2.

aaa**fgh**bfghcfgh	cc**fga**aafgbb
aaa**fgh**bfghcfgh	cc**fga**aafgbb
aaafghb**fgh**cfgh	cc**fga**aafgbb

Tabella 4-2: Esempio di combinazioni ordinate

4.11 Normalizzazione

La normalizzazione del valore del kernel è una procedura che è effettuata per portare tutti i possibili valori di kernel calcolati in una scala da 0 a 1 compresi. Limitare l'escursione del valori può essere utile per migliorare in aluni casi le prestazioni dell'SVM.

Normalizzazione del Mismatch Kernel

Il mismatch kernel normalizzato tra due stringhe A e B è definito come il rapporto tra il mismatch kernel per A e B e il prodotto delle radici quadrate dei mismatch kernel calcolati tra A e se stesso e tra B e se stesso.

$$MK^{NORM}(A,B) = \frac{K(A,B)}{\sqrt{K(A,A)}\ \sqrt{K(B,B)}}$$

Normalizzazione degli Approximate Mismatch Kernel

Nel normalizzare i kernel AMK, WAMK, FWAMK la precedente formula non è applicabile. Questo perché nel calcolo del kernel tra due sequenze si considerano le tuple (dimensioni) presenti in A o B, mentre nel calcolo dell'self-kernel di A e quello di B, si considerano solo le tuple presenti in una sequenza. Nel seguito indicheremo uno qualsiasi di questi kernel semplicemente come K.

In altri termini lo feature-space di K(A,B) è dato dalle tuple $w \in T_{A\ U\ B}$, mentre per K(A,A) è dato solo dalle tuple $w \in T_A$ e per K(B,B) solo dalle tuple $w \in T_B$.

Perché la normalizzazione abbia significato, è invece necessario che tutti i kernel calcolati abbiano lo stesso feature-space. Si devono quindi apportare dei termini correttivi ai self-kernel. Per K(A,A) si devono aggiungere le dimensioni $w \in T_B$, mentre per K(B,B) si devono aggiungere le dimensioni $w \in T_A$. Tali termini correttivi vengono calcolati durante l'elaborazione di K(A,B).

La formula per la normalizzazione del kernel è:

$$K^{NORM}(A,B) = \frac{K(A,B)}{\sqrt{K(A,A) + adjA}\ \sqrt{K(B,B) + adjB}}$$

Dove i termini correttivi (adjust) sono:

$$\text{adjA} = \sum_{w\ \epsilon\ T_B} N_A(w,k)\, N_A(w,k)$$

$$\text{adjB} = \sum_{w\ \epsilon\ T_A} N_B(w,k)\, N_B(w,k)$$

.

5 Implementazione dei kernel

Nel capitolo si descrivono gli algoritmi utilizzati per l'implementazione del kernel combinatorio con ordine. Si introduce l'algoritmo di k-difference matching che permette di calcolare in tempo lineare ammortizzato il numero di mismatch tra tutte le possibili m-tuple in una stringa, e una sua modifica che permette di calcolare il numero di mismatch tra tutte le possibili coppie di m-tuple tra due stringhe. In seguito si presenta l'implementazione del kernel.

5.1 Algoritmo di k-difference matching in tempo lineare ammortizzato

L'algoritmo di k-difference matching descritto in [4], permette di calcolare tutte le occorrenze con mismatch per tutte le parole che occorrono in un testo x almeno una volta esattamente. Altri algoritmi conosciuti in letteratura per tale scopo hanno complessità temporale $O(n^2\sqrt{k\log k})$, per parole di lunghezza fissa, $O(n^3\sqrt{k\log k})$ e per parole di lunghezza variabile, mentre tale algoritmo effettua gli stessi compiti in O(nt) e $O(n^2t)$, quindi in tempo lineare ammortizzato per parola, con t pari al numero di parole differenti nel testo.

5.1.1 Descrizione dell'algoritmo

Dato un testo x = $x_1x_2...x_n$ di lunghezza n, denotiamo con *x(s,e)*, s ≤ e, il segmento di testo $x_sx_{s+1}...x_e$. La lunghezza *x(s,e)* di è m = e –s + 1. Per una lughezza fissata m, $\zeta_k(s,m)$ è l'insieme delle posizioni iniziali delle occorrenze della stringa *w = x(s, s+m-1)* in *x* con esattamente *k* mismatch.

$$\zeta_k(s,m) = \{p: d(x(s, s+m-1), x(p,p+m-1)) = k\}$$

L'approccio per risolvere il problema calcola le occorrenze di *x(s+1,s+m)* da quelle di x(s,s+m-1) scorrondo una finestra di lunghezza m lungo il testo. Dato l'insieme $\{\zeta_k(s,m)$, $0 \le k \le m\}$ delle occorrenze con mismatch della parola di lunghezza m che inizia alla posizione s di x, calcoliamo l'insieme $\zeta_k(s+1,m)$ in due passi:

a. Dall'insieme $\zeta_k(s,m)$ calcoliamo l'insieme $\zeta_k(s,m+1)$: per ogni *k* si considerano le posizioni p ∈ $\zeta_k(s,m)$, e si confronta il simbolo x_{p+m} col simbolo x_{s+m}. Se sono uguali *p* appartiene all'insieme $\zeta_k(s,m+1)$, altrimenti appartiene all'insieme $\zeta_{k+1}(s,m+1)$.

b. Dall'insieme appena calcolato $\zeta_k(s,m+1)$, calcoliamo l'insieme $\zeta_k(s+1,m)$: per ogni $p \in \zeta_k(s,m+1)$, si confrontano x_s e x_p. Se sono uguali, allora *p+1* appartiene a $\zeta_k(s+1,m)$, altrimenti appartiene all'insieme di posizioni $\zeta_{k-1}(s+1,m)$.

Esempio

Si mostra il calcolo del numero di mismatch tra due tuple di lunghezza m a partire dalle tuple che immediatamente le precedono.

Si consideri la stringa *x* di lunghezza *n = 11*, sia *m = 3* la lunghezza delle sottostringhe.

Sia *d(x(i-1,m), x(j-1,m)) = k* il numero conosciuto di mismatch tra le due tuple di lunghezza m che iniziano rispettivamente alle posizioni *i-1* e *j-1*. Si vuole calcolare h = d(*x(i,m), x(j,m)).*

x_0	x_1	x_2	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}
		i--------	**---------**	**(i+m-1)**		**j--------**	**---------**	**(j+m-1)**	
	(i-1)----	**---------**	**(i+m-2)**		**(j-1)----**	**---------**	**(i+m-2)**		

Allora $h = k + bool(x_{i+m-1}, x_{j+m-1}) - bool(x_{i-1},x_{j-1})$, dove $bool(a,b) = 0$ se $a = b$, 1 altrimenti.

5.1.2 Pseudocodice

Data la stringa x di lunghezza n, si introduce la matrice simmetrica M di dimensione n x n per memorizzare il contenuto degli insiemi $\zeta_k(s,m)$, per k=0,…,m e s = 1,…,n-m+1. Ciascuna riga *i* (e colonna *j*) è associata alla tupla *x(i,i+m-1).* L'entry $M_{i,j}$ contiene il numero di mismatch tra *x(i,i+m-1) e x(j,j+m-1).*

```
ALGORITHM_KDIFFSMATCH_LIN()

Read input text string x, and string length m
Compute row 1 with classic k-mismatch algorithm
Compute vector A of length n, where A[i] = first occurrence of x(i,i+m-1) in x.
for i = 2 to n-m+1 do
  if A[i] = i then
       M[i,1] = M[1,i]
       for j = 2 to n-m-1 do
            M[i,j] = M[ A[i]-1,j-1]
            If x_{i+m-1} ≠ x_{j+m-1} then
                 M[i,j] = M[i,j] + 1
            If x_{i-1} ≠ x_{j-1} then
            M[i,j] = M[i,j] – 1
```

Per prima cosa si costruisce il vettore *A* di dimensione n in modo tale che *A[j]* contenga gli indici della prima occorrenza della tupla *x(j,j+m-1)* nel testo *x*.

Il calcolo di A può essere fatto in tempo lineare calcolando per prima cosa il suffix-tree di x in tempo O(n) e successivamente utilizzarlo per cercare le occorrenze di ogni m-tupla di x in tempo O(n) [31].

Si procede a calcolare la prima riga della matrice M con un algoritmo di k-mismatch classico in tempo O(nk), successivamente si procede a riempire M riga per riga. Per ogni stringa x(i,i+m-1) associata alla riga i da calcolare, si legge per prima cosa il valore di A[i].

Se A[i] = i allora questa è la prima occorrenza di x(i,i+m-1) e per la riga *i* si calcolano i valori $M_{i,j}$ per j = 1,…,n e si pone $M_{i,0} = M_{0,i}$. $M_{i,j}$ è inizialmente uguale a $M_{i-1,j-1}$, poi se $x_{i+m-1} \neq x_{j+m-1}$ il suo valore è incrementato di un'unità. Infine se $x_{i-1} \neq x_{j-1}$ il suo valore è decrementato di un'unità.

Se A[i] ≠ i allora si sono già calcolate le occorrenze per la parola x(i,i+m-1), quindi si passa alla prossima riga i+1. si prosegue fino alla prima riga j in cui c'e' una parola non ancora analizzata e si recupera la riga di M[A[j-1],*] per poter calcolare la riga j.

Il tempo e lo spazio totale di tale algoritmo è O(tn), dove t è il numero di parole differenti nel testo.

5.1.3 Implementazione efficiente in C++

5.1.3.1 Librerie e strutture dati utilizzate

Per il calcolo di A si può utilizzare un'implementazione in ANSI C del suffix tree descritto in [32], ma a causa dell'overhead causato dalla struttura dati ad albero e la lunghezza relativamente corta delle stringhe (massimo 600 caratteri), si è preferito utilizzare un'implementazione più semplice ma più molto veloce. In essa si scandisce ogni tupla presente nella stringa e si cerca tramite la funzione standard c++ strstr la sua prima occorrenza.

Sono state utilizzate mappe ordinate per memorizzare le frequenze con mismatch di delle tuple presenti nelle stringhe. La struttura dati utilizzata per le mappe ordinate è l'implementazione standard <map> delle librerie standard c++.

5.1.3.2 Codice

La funzione di calcolo k-difference matching per tutte le tuple di lunghezza m in un testo x restituisce una mappa ordinata. Le chiavi rappresentano tutte le possibili m-tuple di x in formato stringa mentre un valore associato a una chiave K è un vettore V di lunghezza m + 1, dove V[i] è il numero di tuple in x che differiscono per esattamente i mismatch da K.

La variabile seqAMap, riga 03 di Figura 5-1, è la mappa che conterrà i dati restituiti dalla funzione secondo il formato specificato. Matrix, riga 06, è una variabile vettore di puntatori a vettori di double corrispondente alla variabile M. La sua dimensione è pari a lenA – m + 1, cioè il numero di righe di M (o equivalentemente il numero di tuple di lunghezza m in x). Il vettore merPosA, riga 09 di Figura 5-1, contiene le posizioni delle prime occorrenze di ogni tupla in seqA. Il ciclo for in riga 11 effettua il calcolo di tale vettore. Per trovare la prima occorrenza di ogni stringa si utilizza l'efficiente funzione strstr, riga 18, delle librerie standard C++. Assieme al calcolo di mersPosA viene inizializzata la variabile matrix. Per ottimizzare il calcolo e lo spazio dati utilizzato, si usano in maniera accurata i puntatori. Una riga i di matrix è definita solo nel caso in A[i] = i (riga 27) , cioè nel caso in cui la tupla nella posizione *x(i,i+m-1)* è alla sua prima occorrenza. Se la tupla in posizione *i* è gia definita precedentemente, con prima posizione pos = A[i], matrix[i] punta al vettore della riga A[i]. In tale modo l'occupazione della matrice in memoria è linearmente ammortizzata, pari a O(tn), con t pari al numero totale di parole differenti nel testo. Inoltre si semplifica la gestione dei riferimenti per le righe.

firstARow, riga 33, memorizza i valori della prima riga di matrix, cioè il numero di mismatch tra la prima tupla e tutte le altre tuple in seqA.

Per il calcolo si utilizza la funzione naiveSingleKdiffMatch, che implementa il metodo naive per il calcolo di tale riga. Tale funzione scandisce ogni tupla della stringa per confrontarla con la tupla passata in ingresso. Si è preferito tale metodo in quanto non introduce overhead dato da strutture dati aggiuntive, e quindi per stringhe relativamente corte è veloce.

Le righe da 41 a 50 di Figura 5-1 inseriscono i dati associati alla prima tupla nella mappa seqAmap e inizializzano la prima riga della matrice.

Il ciclo for nella riga 02 di *Figura 5-2* effettua il calcolo di ogni riga di matrix, nel caso in cui essa si incontri per la prima volta. In riga 11 si copia il valore per l'entry nella prima colonna dalla entry corrispondente nella prima riga. Il ciclo for in 16 effettua il calcolo per ogni elemento della riga secondo l'algoritmo descritto precedentemente.

```
01 map<string,vector<int> >* PizziKernel::allKDiffsMap_Fast(char* seqA, int lenA, int m, int k){
02
03        map<string,vector<int> >* seqAMap = new map<string,vector<int> >();
04
05        //use of pointers for repeated rows
06        double** matrix;
07        matrix = new double*[lenA - m + 1];
08        //Vector for first occurences
09        int* mersPosA = new int[lenA - m + 1];
10
11        for(int i = 0; i <= lenA - m;i++){
12
13                  char* mer = new char[m + 1];
14                  for(int j = 0; j < m; j++){
15                    mer[j] = seqA[i+j];
16                  }
17                  mer[m] = '\0';
18                  char* ref = strstr(seqA,const_cast<char *> (mer) );
19                  int pos = ref - seqA;
20
21                  //initialize of matrix, based on first occurences
22                  if(pos < i){
23                    mersPosA[i] = pos;
24                    matrix[i] = matrix[pos];
25                  }else{
26                    mersPosA[i] = i;
27                    matrix[i] = new double[lenA - m + 1];
28                  }
29                  delete[] mer;
30        }
31
32        //compute first row
33        vector<int>* firstARow = naiveSingleKdiffMatch(seqA, seqA, m);
34        string keyA = "";
35        //read first row key
36        for(int v = 0; v < m; v++){
37          keyA += seqA[v];
38        }
39
40        //default value for map vectors. (Vector of m+1 length)
41        vector<int> mapDefaultVal;
42        mapDefaultVal.resize(m + 1);
43        seqAMap->insert(std::pair<string,vector<int> >(keyA,mapDefaultVal) );
44        map<string,vector<int> >::iterator itPairA = seqAMap->find(keyA);
45
46        //copy of the first row in the matrix.
47        for(unsigned int i = 0; i < firstARow->size() ; i++){
48                  (itPairA->second)[(*firstARow)[i]]++;
49                  matrix[0][i] = (*firstARow)[i];
50        }
```

Figura 5-1: Implementazione C++ algoritmo k-diff match - parte 1

```
01        //for each matrix row
02        for(int i = 1; i <= lenA - m ; i++){
03                  //if it's the first time the mer occur, compute the row
04                  if(mersPosA[i] == i){
05                            //read the row key
06                            keyA = "";
07                            for(int v = 0; v < m; v++){
08                              keyA += seqA[i + v];
09                            }
10
11                            matrix[i][0] = (*firstARow)[i];
12                            vector<int> vals;
13                            vals.resize(m + 1);
14
15                            //for each column
16                            for(int j = 1; j <= lenA - m ; j++){
17                                      //initialize matrix[i,j]
18                                      int val = matrix[i-1][j-1];
19                                      if(seqA[i+m-1] !=  seqA[j+m-1]){
20                                                val++;
21                                      }
22                                      if(seqA[i-1] !=  seqA[j-1]){
23                                                val--;
24                                      }
25                                      matrix[i][j] = val;
26                                      //update mismatch vector
27                                      vals[val]++;
28                            }
29                            //insert in the map the pair computed
30                            seqAMap->insert(std::pair<string,vector<int> >(keyA,vals) );
31                  }
32        }
33        //memory clean
34        delete firstARow;
35        for(int i = 0; i <= lenA - m;i++){
36                  if(mersPosA[i] == i){
37                    delete matrix[i];
38                  }
39        }
40        delete matrix;
41        delete mersPosA;
42
43        return seqAMap;
44 }
```

Figura 5-2: Implementazione C++ algoritmo k-diff match - parte 2

5.2 Algoritmo k-difference matching tra due sequenze

L'algoritmo descritto nel paragrafo precedente si può adattare in modo semplice per trovare tutte le occorrenze con mismatch di tutte le parole di una sequenza *a* in un'altra sequenza *b* e viceversa. Tale algoritmo sarà chiamato cross k-difference matching, ad indicare il matching incrociato tra le tuple di due sequenze.

5.2.1 Descrizione dell'algoritmo

Date due sequenze, a = $a_1a_2\ldots a_n$ di lunghezza n e b = $b_1b_2\ldots b_h$ di lunghezza h, denotiamo con *a(s,e)* , s ≤ e, il segmento di testo $a_sa_{s+1}\ldots a_e$ e con *b(s,e)* , s ≤ e, il segmento di testo $b_sb_{s+1}\ldots b_e$. Per una lughezza fissata m, $\zeta_k(s,m)$ è l'insieme delle posizioni iniziali delle occorrenze della stringa *w = a(s, s+m-1)* in *b* con esattamente *k* mismatch

$$\zeta_k(s,m) = \{p: d(a(s, s+m-1), b(p,p+m-1)) = k\}$$

$\lambda_k(s,m)$ è l'insieme delle posizioni iniziali delle occorrenze della stringa *w = b(s, s+m-1)* in *a* con esattamente *k* mismatch

$$\lambda_k(s,m) = \{p: d(b(s, s+m-1), a(p,p+m-1)) = k\}$$

L'approccio calcola le occorrenze di *a(s+1,s+m)* da quelle di a(s,s+m-1) scorrendo una finestra di lunghezza m lungo la sequenza *a*. Dato l'insieme $\{ \zeta_k(s,m) , 0 \leq k \leq m\}$ delle occorrenze con mismatch della parola di lunghezza m che inizia alla posizione s di a, calcoliamo l'insieme $\zeta_k(s+1,m)$ in due passi, in modo simile a quello utilizzato per il k-difference matching:

a. Dall'insieme $\zeta_k(s,m)$ si calcola l'insieme $\zeta_k(s,m+1)$: per ogni *k* si considerano le posizioni $p \in \zeta_k(s,m)$, e si confronta il simbolo a_{p+m} col simbolo b_{s+m}. Se sono uguali *p* appartiene all'insieme $\zeta_k(s,m+1)$, altrimenti appartiene all'insieme $\zeta_{k+1}(s,m+1)$.

b. Dall'insieme appena calcolato $\zeta_k(s,m+1)$, calcoliamo l'insieme $\zeta_k(s+1,m)$: per ogni $p \in \zeta_k(s,m+1)$, si confrontano a_s e b_p. Se sono uguali, allora *p+1* appartiene a $\zeta_k(s+1,m)$, altrimenti appartiene all'insieme di posizioni $\zeta_{k-1}(s+1,m)$.

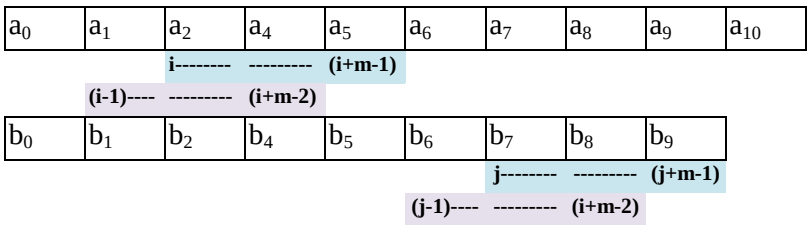
In modo simmetrico è possibile calcolare le occorrenze con mismatch delle tuple di *b* in *a*.

Esempio

Si mostra il calcolo del numero di mismatch tra una tupla di *a* e una di *b* di lunghezza m a partire dalle tuple che immediatamente le precedono nelle rispettive sequenze.

Si considerino le stringhe a di lunghezza *n = 11* e b di lunghezza h = 10, sia *m = 3* la lunghezza delle sottostringhe.

Sia *d(a(i-1,i+m-2), b(j-1,j+m-2)) = k* il numero di mismatch tra le due tuple di lunghezza m che iniziano rispettivamente alle posizioni *i-1* di *a* e *j-1 di* b. Si vuole calcolare h = d(*a(i,i+m-1), b(j,m+m-1)).*



Allora *h = k + bool*(a_{i+m-1}, b_{j+m-1}) – *bool*(a_{i-1},b_{j-1}), dove *bool(a,b) = 0* se *a = b*, *1* altrimenti.

5.2.2 Pseudocodice

Data la stringhe a,b di lunghezza n e h rispettivamente, si introduce la matrice M di dimensione n x h per memorizzare il contenuto degli insiemi $\zeta_k(s,m)$, per k=0,…,m e s = 1,…,n-m+1 e degli insiemi $\lambda_k(s,m)$, per k=0,…,m e s = 1,…,h-m+1. Ciascuna riga *i* è associata alla tupla *a(i,i+m-1)* e ciascuna colonna *j* è associata alla tupla *b(j,j+m-1).* La entry $M_{i,j}$ contiene il numero di mismatch tra *a(i,i+m-1) e b(j,j+m-1).*

```
ALGORITHM_KDIFFSMATCH_LIN()

Read input text sequences a and b, and string length m
Compute row 1 with classic k-mismatch algorithm using a(0,m-1) and sequence b
Compute column 1 with classic k-mismatch algorithm using b(0,m-1) and sequence a
Compute vector A of length n, where A[i] = first occurrence of a(i,i+m-1) in a.
for i = 2 to n-m+1 do
  if A[i] = i then
        for j = 2 to n-m-1 do
              M[i,j] = M[ A[i]-1,j-1]
              If a_{i+m-1} ≠ b_{j+m-1} then
                    M[i,j] = M[i,j] + 1
              If a_{i-1} ≠ b_{j-1} then
              M[i,j] = M[i,j] – 1
```

L'algoritmo è simile a quello presentato in 5.1.2. Si può notare il calcolo aggiuntivo delle prima colonna, dovuto alla mancanza di simmetria in M, e i confronti tra gli elementi delle tuple di a e b.

Il tempo e lo spazio totale impiegato dall'algoritmo è pari a O(km), dove k è il numero di tuple distinte in a.

5.2.3 Implementazione efficiente in C++

Il metodo allcross_KDiffs effettua il calcolo della matrice M secondo l'algoritmo descritto in *5.2.2*. Riceve in ingresso le sequenze A e B, la lunghezza delle tuple m, e il numero di mismatch k. I vettori mersPosA e mersPosB (righe 11,13 Figura 5-3) contengono le posizioni delle prime occorrenze di ogni tupla in seqA e seqB rispettivamente. Come per l'implementazione dell'algoritmo k-difference matching la matrice dei mismatch M è gestita tramite puntatori, e una riga i è inizializzata solo se la tupla *a(i,i+m-1)* è alla sua prima occorrenza nella sequenza. Dalle righe 16 a 55 di *Figura* 5-3 si inizializzano mersPosA, mersPosB e M. Notare in particolare alle righe 31 e 33 l'inizializzazione di M, effettuata in modo simile a quanto illustrato in *5.1.3.2.*

crsSeqAMap è una mappa non ordinata le cui chiavi rappresentano tutte le possibili m-tuple di seqA in formato stringa. Il valore associato ad una chiave K, tupla di A, è un vettore V di lunghezza m + 1, dove V[i] è il numero di tuple nella sequenza B che differiscono per esattamente i mismatch da K. crsSeqBMap contiene in modo simile le informazioni sulle tuple di B che sono in A con mismatch. Tali mappe sono restituite in un vettore, crsSeqAMap all'indice 0, crsSeqBMap all'indice 1.

In riga 66, Figura 5-3 si effettua il calcolo della prima riga di M. La funzione naiveSingleKdiffMatch calcola il numero di mismatch tra la prima tupla di seqA e ogni tupla di B.

Il ciclo for tra le righe 12-22 di Figura 5-4 copia in M il contenuto della prima riga calcolata e calcola il vettore del numero di mismatch associato alla prima tupla di A per l'inserimento nella mappa crsSeqAMap. Il ciclo for tra le righe 39-43 effettua la medesima procedura per la prima colonna di M.

Nella riga 02 di Figura 5-5 vi è il ciclo principale che effettua il calcolo di M una riga alla volta.

```
01 vector<unordered_map<string,vector<int> > > allcross_KDiffs(const char* inSeqA,int lenA,const char*
02 inSeqB, int lenB, int m, int k){
03
04          char* seqA = const_cast<char*>(inSeqA);
05          char* seqB = const_cast<char*>(inSeqB);
06
07          //mismatch matrix
08          double** M = new double*[lenA - m + 1];
09
10          //Vector for first occurences of A
11          int* mersPosA = new int[lenA - m + 1];
12          //Vector for first occurences of B
13          int* mersPosB = new int[lenB - m + 1];
14
15          //build mersPosA and M
16          for(int i = 0; i <= lenA - m;i++){
17                    char* mer = new char[m + 1];
18                    for(int j = 0; j < m; j++){
19                      mer[j] = seqA[i+j];
20                    }
21                    mer[m] = '\0';
22
23                    char* ref = strstr(seqA,const_cast<char *> (mer) );
24                    int pos = ref - seqA;
25
26                    //initialize of cross matrix, based on first occurrences
27                    if(pos < i){
28                      mersPosA[i] = pos;
29                      //use of pointers for repeated rows
30                      M[i] = M[pos];
31                    }else{
32                      mersPosA[i] = i;
33                      M[i] = new double[lenB - m + 1];
34                    }
35                    delete[] mer;
36          }
37
38          //build merPosB
39          for(int i = 0; i <= lenB - m;i++){
40                    char* mer = new char[m + 1];
41                    for(int j = 0; j < m; j++){
42                      mer[j] = seqB[i+j];
43                    }
44                    mer[m] = '\0';
45
46                    char* ref = strstr(seqB,const_cast<char *> (mer) );
47                    int pos = ref - seqB;
48
49                    if(pos < i){
50                      mersPosB[i] = pos;
51                    }else{
52                      mersPosB[i] = i;
53                    }
54                    delete[] mer;
55          }
56
57          //cross mer mismatch infos maps
58          unordered_map<string,vector<int> > crsSeqAMap;
59          unordered_map<string,vector<int> > crsSeqBMap;
60
61          //default value for map vectors. (Vector of m+1 length)
62          vector<int> mapDefaultVal;
63          mapDefaultVal.resize(m + 1);
64
65          //--> compute M first row
66          vector<int>* firstRow = PizziKernel::naiveSingleKdiffMatch(seqB, seqA, m);
67
68          string keyA = "";
69          //read A key
70          for(int v = 0; v < m; v++){
71            keyA += seqA[v];
72          }
```

Figura 5-3: Algoritmo k-difference matching tra due sequenze - parte 1

```
01      //A map iterator
02      unordered_map<string,vector<int> >::iterator crsItPairA;
03
04      vector<int> aFirstMerVect;
05      aFirstMerVect.resize(m + 1);
06
07      string keyB = "";
08      //B map iterator
09      unordered_map<string,vector<int> >::iterator crsItPairB;
10
11      //--> first row analisys
12      for(unsigned int i = 0; i < firstRow->size() ; i++){
13              keyB = "";
14              //read B key (column)
15              for(int v = 0; v < m; v++){
16                keyB += seqB[i + v];
17              }
18
19              //update value for A
20              aFirstMerVect[(*firstRow)[i]]++;
21              M[0][i] = (*firstRow)[i];
22      }
23      //clean firstRow memory
24      delete firstRow;
25
26      crsSeqAMap.insert(std::pair<string,vector<int> >(keyA,aFirstMerVect) );
27      //--> first column computing
28      vector<int>* firstColumn = PizziKernel::naiveSingleKdiffMatch(seqA, seqB, m);
29
30      //First column  matrix inizialization
31      keyB = "";
32      for(int v = 0; v < m; v++){
33        keyB += seqB[v];
34      }
35
36      vector<int> bFirstMerVect;
37      bFirstMerVect.resize(m + 1);
38
39      for(int i = 1; i <= lenA - m ; i++){
40              M[i][0] = (*firstColumn)[i];
41              //update value for B
42              bFirstMerVect[(*firstColumn)[i]]++;
43      }
44
45      crsSeqBMap.insert(std::pair<string,vector<int> >(keyB,bFirstMerVect) );
46      //clean firstColumn memory
47      delete firstColumn;
```

Figura 5-4: Algoritmo k-difference matching tra due sequenze - parte 2

E' molto simile all'algoritmo illustrato in 5.1.3.2 eccetto che per le righe da 20 a 27 che, in questo caso, effettuano il confronto tra gli elementi delle due sequenze invece che tra gli elementi di una stessa sequenza. In tale ciclo vengono anche aggiornate le coppie chiave-valore della mappa di A (righe 14, 27, 30). Tale ciclo ha complessità computazionale O(kh), dove k è il numero di tuple di lunghezza m distinte di seqA e h è la lunghezza di seqB.

Successivamente nelle righe 35-50 di *Figura 5-5* si analizza ogni colonna j di M associata ad un mer di seqB che appare per la prima volta nella posizione j della sequenza seqB. Ciè viene fatto in tempo O(nl), dove n è il numero di tuple in seqA e l è il numero di tuple distinte in seqB.

Nell'indice h di valsVect si memorizzano il numero di tuple in A che differiscono per esattamente h mismatch dal mer *b(j,j+m-1)*. Successivamente si inserisce tale vettore, associato alla chiave *b(j,j+m-1)* nella mappa crsSeqBMap.

Nelle righe 62-66 di Figura 5-5 si inseriscono le due mappe in un vettore, che viene restituito dalla funzione. La complessità computazionale del metodo è O(kh + nl). Nel caso peggiore pari a O(nh). L'occupazione in memoria è pari a O(kh), dovuta al contributo determinante della matrice M.

```
01          //for each matrix row
02          for(int i = 1; i <= lenA - m ; i++){
03                      //if it's the first time the mer occur in A, compute the row
04                      if(mersPosA[i] == i){
05                                  //read the row key
06                                  keyA = "";
07                                  for(int v = 0; v < m; v++){
08                                    keyA += seqA[i + v];
09                                  }
10
11                                  vector<int> merVect;
12                                  merVect.resize(m + 1);
13                                  //set value related to first column item
14                                  merVect[M[i][0]]++;
15
16                                  //for each column
17                                  for(int j = 1; j <= lenB - m ; j++){
18
19                                              M[i][j] = M[i-1][j-1];
20                                              if(seqA[i+m-1] !=  seqB[j+m-1]){
21                                                          M[i][j] = M[i][j] + 1;
22                                              }
23                                              if(seqA[i-1] !=  seqB[j-1]){
24                                                          M[i][j] = M[i][j] - 1;
25                                              }
26                                              //aggiorno valore per A
27                                              merVect[M[i][j]]++;
28                                  }
29                                  //insert in the map the pair computed
30                                  crsSeqAMap.insert(std::pair<string,vector<int> >(keyA,merVect) );
31                      }
32          }
33          //update mer infos of B, for each column (number of columns
34          //equal to n° of mers in B)
35          for(int j = 0; j <= lenB - m ; j++){
36                      if(mersPosB[j] == j){
37                                  keyB = "";
38                                  //leggo chiave B
39                                  for(int v = 0; v < m; v++){
40                                    keyB += seqB[j + v];
41                                  }
42                                  vector<int> valsVect;
43                                  valsVect.resize(m + 1);
44                                  //for each row
45                                  for(int i = 0; i <= lenA - m ; i++){
46                                              valsVect[M[i][j]]++;
47                                  }
48                                  crsSeqBMap.insert(std::pair<string,vector<int> >(keyB,valsVect) );
49                      }
50          }
51
52          //Free the memory
53          delete[] seqA; delete[] seqB;
54          for(int i = 0; i <= lenA - m; i++){
55                      if(mersPosA[i] == i){
56                                  delete M[i];
57                      }
58          }
59          delete M;  delete mersPosA; delete mersPosB;
60
61          //return vector
62          vector<unordered_map<string,vector<int> > > retVec;
63          //on first index A map
64          retVec.push_back(crsSeqAMap);
65          //on second index B map
66          retVec.push_back(crsSeqBMap);
67
68          return  retVec;
69 }
```

Figura 5-5: Algoritmo k-difference matching tra due sequenze - parte 3

5.3 Algoritmo Kernel WAMK

Nel seguito si riporta l'algoritmo per il calcolo del Kernel WAMK introdotto in 4.8. Esso utilizza gli algoritmi introdotti in 5.1 e 5.2.

Il metodo computerKernelFast riceve in ingresso le due sequenze inSeqA e inSeqB e i parametri lunghezza delle tuple m e numero massimo di mismatch. Le variabili seqAMap e seqBMap memorizzano le mappe calcolate dalla funzione *allKDiffsMap_Fast* rispettivamente sulla sequenza A e sulla sequenza B. Esse sono mappe ordinate secondo l'ordine lessicografico delle loro chiavi.

```
01      KernelResult computeKernelFast(const char* inSeqA,int lenA,const char* inSeqB,
02                                                              int lenB, int m, int k){
03
04              //cast away constantness
05              char* seqA = const_cast<char*>(inSeqA);
06              char* seqB = const_cast<char*>(inSeqB);
07
08              //cross values map
09              unordered_map<string,vector<int> > crsSeqAMap;
10              unordered_map<string,vector<int> > crsSeqBMap;
11
12              //single maps
13              map<string,vector<int> >* seqAMap;
14              map<string,vector<int> >* seqBMap;
15
16              vector<int> mapDefaultVal;
17              mapDefaultVal.resize(m + 1);
18              //*********************************************************************
19              // --->   Compute A map
20              seqAMap = PizziKernel::allKDiffsMap_Fast(seqA, lenA, m, k);
21              //*********************************************************************
22              // --->   Compute B map
23              seqBMap = PizziKernel::allKDiffsMap_Fast(seqB, lenB, m, k);
24
25              string keyA = "";
26              //read A key
27              for(int v = 0; v < m; v++){
28                keyA += seqA[v];
29              }
30
31              unordered_map<string,vector<int> >::iterator crsItPairA;
32
33              string keyB = "";
34              unordered_map<string,vector<int> >::iterator crsItPairB;
35
36              vector<unordered_map<string,vector<int> > > crossMaps;
37              crossMaps = allcross_KDiffs(inSeqA, lenA, inSeqB, lenB, m, k);
38
39              crsSeqAMap = crossMaps[0];
40              crsSeqBMap = crossMaps[1];
41
42              double kernelVal = 0;
43              //adjustment values
44              double adjX = 0; double adjY = 0;
45              //Max k = 2
46              double prests[3][3];
47              prests[0][0] = 1;       prests[0][1] = 1;       prests[0][2] = 0.6;
48              prests[1][0] = 1;       prests[1][1] = 0.3;     prests[1][2] = 0.55;
49              prests[2][0] = 0.6;     prests[2][1] = 0.55;    prests[2][2] = 0;
50
51              map<string, vector<int> >::iterator itOrdPairA;
52              map<string, vector<int> >::iterator itOrdPairB;
53
54              //*********************************************************************
55              //----------COMPUTE KERNEL--------------------
56              itOrdPairA = seqAMap->begin();
57              itOrdPairB = seqBMap->begin();
```

Figura 5-6: computerKernelFast - parte 1

Le variabili crsSeqAMap e crsSeqBMap memorizzano invece le mappe contenenti le informazioni di k-difference matching tra le sequenze A e B calcolate tramite

allCross_KDiffs nella riga 37. La matrice prests in riga 46 memorizza i pesi dei contributi del kernel, e corrisponde alla matrice H introdotta in 4.7.

Il calcolo del kernel viene effettuato nel ciclo while (riga 01) di figura Figura 5-7. Nel ciclo si scandiscono in modo ordinato le mappe seqAMap e seqBMap mediante gli iteratori itOrdPairA e itOrdPairB. Si confrontano le chiavi correnti degli iteratori delle mappe in riga $w \in T_{A \cup B}$.

Se le due tuple sono uguali (riga 03) allora la chiave w in oggetto è presente sia in A che in B e si ha $w \in T_{A \cap B}$. Si procede allora nelle righe aa-bb nel calcolo di $C_{WI}(w,k)$.

```
01          while(itOrdPairA != seqAMap->end() && itOrdPairB != seqBMap->end()){
02              //compare A mer with B mer
03              int compVal = (itOrdPairA->first).compare(itOrdPairB->first);
04              //if the mer is in common for both strings A, B
05              // multiply the number of occurencies in one string
06              //for the number in the other one
07              if(compVal == 0){
08                double sum  = 0;
09                for(int i = 0; i <= k; i++){
10                      for(int j = 0; j <= k; j++){
11                          sum += prests[i][j] * calcComb(itOrdPairA->second[i],
12                                  itOrdPairB->second[j]);
13                      }
14                }
15                //--------------------------------
16                kernelVal += sum;
17                //increments A and B iterators
18                itOrdPairA++;
19                itOrdPairB++;
20              //A key is lower than B key, so the
21              //mer is only in A.
22              }else if(compVal < 0){
23                 double sum  = 0;
24                 keyA = itOrdPairA->first;
25                 //search the pair (key, freqs count) in A cross map
26                 crsItPairA = crsSeqAMap.find(keyA);
27                 for(int i = 0; i <= k; i++){
28                      for(int j = 0; j <= k; j++){
29                          int sum  += prests[i][j] * calcComb(itOrdPairA->second[i],
30                                  crsItPairA->second[j]);
31                          adjY += prests[i][j] * calcComb(crsItPairA->second[i],
32                                          crsItPairA->second[j]);
33                      }
34                 }
35                 //--------------------------------
36                 kernelVal += sum;
37                 //increment only the A iterator
38                 itOrdPairA++;
39
40              }else{
41                  double sum  = 0;
42                  keyB = itOrdPairB->first;
43                  crsItPairB = crsSeqBMap.find(keyB);
44
45                  for(int i = 0; i <= k; i++){
46                      for(int j = 0; j <= k; j++){
47                        int sum += prests[i][j] * calcComb(itOrdPairB ->second[i],
48                                crsItPairB ->second[j]);
49                        adjX      += prests[i][j] * calcComb(crsItPairB->second[i],
50                                          crsItPairB->second[j]);
51                      }
52                  }
53                  //--------------------------------
54
55                  kernelVal += sum;
56                  itOrdPairB++;
57              }
58          }
```

Figura 5-7: computerKernelFast - parte 2

In riga 36 si aggiunge il valore calcolato al valore finale del kernel.

Se la tupla in A è minore della tupla in B (riga 22), allora la tupla w in A è presente solo in tale sequenza, cioè $w \in T_A$. Nelle righe 37-34 si calcola $C_{WA}(w,k)$.

Infine se la tupla in A è maggiore della tupla in B (riga 22), allora la tupla w in B è presente solo in tale sequenza, cioè $w \in T_B$. Nelle righe 37-34 si calcola $C_{WB}(w,k)$.

```
01
02                  //process remaining A mers
03                  while(itOrdPairA != seqAMap->end()){
04                          double sum  = 0;
05
06                          keyA = itOrdPairA->first;
07                          crsItPairA = crsSeqAMap.find(keyA);
08
09                          for(int i = 0; i <= k; i++){
10                              for(int j = 0; j <= k; j++){
11                                  int sum += prests[i][j] * calcComb(itOrdPairA->second[i],
12                                                  crsItPairA->second[j]);
13                                      adjY  += prests[i][j] * calcComb(crsItPairA ->second[i],
14                                                                  crsItPairA->second[j]);
15
16                              }
17                          }
18                          //-------------------------------
19                          kernelVal += sum;
20                          itOrdPairA++;
21                  }
22
23                  //process remaining B mers
24                  while(itOrdPairB != seqBMap->end()){
25                          double sum  = 0;
26                          keyB = itOrdPairB->first;
27                          crsItPairB = crsSeqBMap.find(keyB);
28                          for(int i = 0; i <= k; i++){
29                              for(int j = 0; j <= k; j++){
30                                  int sum           += prests[i][j] * calcComb(itOrdPairB ->second[i],
31                                                      crsItPairB ->second[j]);
32                                  adjX              += prests[i][j] * calcComb(crsItPairB ->second[i],
33                                                                      crsItPairB ->second[j])
34                              }
35                          }
36                          //-------------------------------
37
38                          kernelVal += sum;
39                          itOrdPairB++;
40                  }
41
42                  //build result object
43                  KernelResult result(kernelVal,adjX,adjY);
44                  //-----------------------------------------------------------------------------
45                  //--> Memory clean
46
47                  delete[] seqA; delete[] seqB;
48
49                  delete seqAMap;
50                  delete seqBMap;
51
52                  return  result;
53          }
```

Figura 5-8: computerKernelFast - parte 3

Infine in Figura 5-8 vi sono due cicli while (righe 03 e 24) per il calcolo delle tuple rimanenti in A o in B. Si noti che nelle righe 32 e 50 di Figura 5-7 e nelle righe 14 e 33 di Figura 5-8 sono calcolati anche i termini correttivi introdotti in Normalizzazione degli Approximate Mismatch Kernel per il calcolo del kernel normalizzato.

In riga 43 si costruisce l'oggetto da restituire nel risultato del metodo, contenente il valore del kernel e i termini correttivi adjX, adjY.

5.4 Ottimizzazione dell'algoritmo WAMK

L'algoritmo descritto nel paragrafo precedente si può migliorare. Si può vedere che esso chiama due volte la funzione *allKDiffsMap_Fast* (righe 20, 21 di Figura 5-6), una volta per il calcolo della per la mappa che descrive i mismatch di A, e un'altra volta per il calcolo della mappa che descrive i mismatch di B. Successivamente (righe 37 di Figura 5-6) viene chiamata la funzione *allCross_KDiffs* per il calcolo delle mappe che descrivono i mismatch incrocati tra le tuple di A e B.

Si può osservare che la funzione *allCross_KDiffs* calcola i vettori *merPosA*, *merPosB* per le prime occorrenze delle tuple in A e B. Tale calcolo è effettuato separatamente anche dalla funzione *allKDiffsMap_Fast* chiamata separatamente per le stringhe A e B.

Il calcolo del WAMK si può in primo luogo migliorare calcolando solo una volta tali vettori.

La funzione *allKDiffsMap_Fast* effetua per la stringa A un ciclo for di complessità O(rn), con r numero di m-tuple distinte in A e n lunghezza di A, mentre per la stringa B effettua un ciclo for di complessità O(lh) , con l numero di m-tuple distinte in B e h lunghezza di B.

La funzione *allCross_KDiffs* effettua, come precedentemente descritto, un ciclo for di complessità O(rh) ed un altro di complessità O(nl).

Per migliorare le prestazioni del calcolo del kernel si possono unire tra loro tali cicli for chiamando un unico ciclo for in un singolo metodo risparmiando tempo di esecuzione. Implementando tutto in un unico metodo si può inoltre risparmiare il tempo speso per le tre chiamate alle funzioni e per la gestione di strutture dati aggiuntive nei tre metodi separati.

Si usa quindi un unico ciclo for per calcolare le matrici di mismatch per A, B e A e B congiunti.

Per sfruttare al meglio l'ammortizzamento lineare, alle righe del ciclo for si fa corrispondere la sequenza tra A e B più lunga. Infatti più lunga è una stringa e più è probabile che al suo interno vi siano m-tuple ripetute.

Chiamiamo quindi:

- A' = stringa_lunghezza_max(A,B)
- B'= stringa_lunghezza_min(A,B)

L'indice i per il ciclo for delle righe parte da 2 fino a len(A')-m. Per poter calcolare la matrice dei mismatch per B all'interno di tale ciclo si definisce una variabile, *bIndex* che tiene conto dell'indice di riga per B. Infatti, una riga i per la matrice di mismatch di A e A e B incrociati è calcolata solo se A(i,i+m-1) è alla prima occorrenza. Per usare l'ammortizzamento lineare in B è necessario tenere conto separatamente delle prime occorrenze. Può accadere, ad esempio, che A abbia molte tuple ripetute e quindi il numero di righe calcolate sia molto inferiore al numero di righe in B. Quindi, alla fine del ciclo for per le righe, è necessario avviare un altro ciclo per calcolare le righe rimanenti in B.

All'interno del ciclo for per le righe si deve effettuare un ciclo for di indice j da 1 a lunghezza(B') - m. In esso si calcolano i valori per la matrice di mismatch congiunta, i valori per la matrice di B e i primi lunghezza(B')-m valori per la matrice di mismatch di A. In seguito si esegue un altro ciclo for per i rimanenti (lunghezza(A') – lunghezza(B')) valori di A.

Si mostra in figura Figura 5-9 il concetto illustrato.

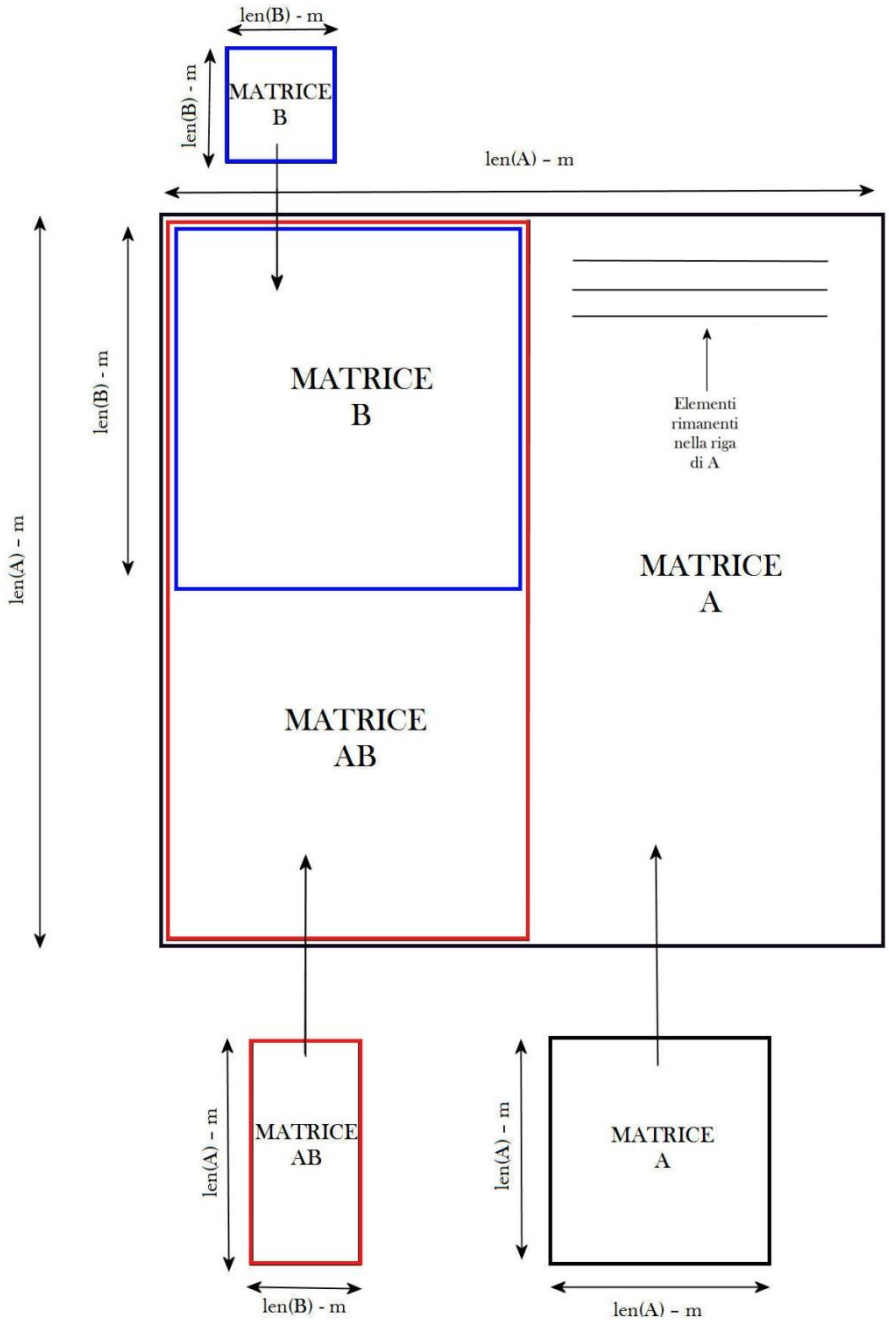


Figura 5-9: Idea alla base dell'ottimizzazione per il calcolo del AMK

Nelle figure rimanenti del capitolo è riportato l'algoritmo ottimizzato.

```
KernelResult computeKernelFastOptimization(const char* inSeqA,int inLenA,const char*
                                                    inSeqB, int inLenB, int m, int k){

        char* seqA;
        char* seqB;
        int lenA, lenB;
        //si mette per le righe della matrice congiunta sempre la sequenza di
        //lunghezza maggiore (per sfruttare al meglio il possibile ammortizzamento lineare)
        if(inLenA > inLenB){
          lenA = inLenA; lenB = inLenB;
          seqA = const_cast<char*>(inSeqA);
          seqB = const_cast<char*>(inSeqB);
        //inverto le sequenze
        }else{
          lenA = inLenB; lenB = inLenA;
          seqA = const_cast<char*>(inSeqB);
          seqB = const_cast<char*>(inSeqA);
        }
        double** M = new double*[lenA - m + 1];
        //*****************************************************************
        // --->   Mappe singole di A e B
        map<string,vector<int> >* seqAMap = new map<string,vector<int> >();;
        map<string,vector<int> >* seqBMap = new map<string,vector<int> >();;
        // --->   Matrice per A e B
        //use of pointers for repeated rows
        double** matrixA;
        matrixA = new double*[lenA - m + 1];
        double** matrixB;
        matrixB = new double*[lenB - m + 1];
        //-----------------------------------------------------------------------
        //Vettori per tuple ripetute
        int* mersPosA = new int[lenA - m + 1];
        int* mersPosB = new int[lenB - m + 1];

        for(int i = 0; i <= lenA - m;i++){
                char* mer = new char[m + 1];
                for(int j = 0; j < m; j++){
                  mer[j] = seqA[i+j];
                }
                mer[m] = '\0';
                char* ref = strstr(seqA,const_cast<char *> (mer) );
                int pos = ref - seqA;

                if(pos < i){
                  mersPosA[i] = pos;
                  M[i] = M[pos];
                  //inizializzazione matrice A
                  matrixA[i] = matrixA[pos];
                }else{
                  mersPosA[i] = i;
                  M[i] = new double[lenB - m + 1];
                  //inizializzazione matrice A
                  matrixA[i] = new double[lenA - m + 1];
                }

                delete[] mer;
        }
```

Figura 5-10: Ottimizzazione calcolo AMK – parte 1

```
for(int i = 0; i <= lenB - m;i++){
		char* mer = new char[m + 1];
		for(int j = 0; j < m; j++){
		  mer[j] = seqB[i+j];
		}
		mer[m] = '\0';
		char* ref = strstr(seqB,const_cast<char *> (mer) );
		int pos = ref - seqB;
		if(pos < i){
		  mersPosB[i] = pos;
		  matrixB[i] = matrixB[pos];
		}else{
		  mersPosB[i] = i;
		  matrixB[i] = new double[lenB - m + 1];
		}
		delete[] mer;
}
//default value for map vectors. (Vector of m+1 length)
vector<int> mapDefaultVal;
mapDefaultVal.resize(m + 1);
string keyA = "";
string keyB = "";
//************************************************************************
//compute first row of A
vector<int>* firstARow =naiveSingleKdiffMatch(seqA, seqA, m);
keyA = "";
//read first row key
for(int v = 0; v < m; v++){
  keyA += seqA[v];
}
seqAMap->insert(std::pair<string,vector<int> >(keyA,mapDefaultVal) );
map<string,vector<int> >::iterator itPairA = seqAMap->find(keyA);
//--> copy of the first row in the matrix.
for(unsigned int i = 0; i < firstARow->size() ; i++){
		(itPairA->second)[(*firstARow)[i]]++;
		matrixA[0][i] = (*firstARow)[i];
}
//************************************************************************
//compute first row of B
vector<int>* firstBRow =naiveSingleKdiffMatch(seqB, seqB, m);
keyB = "";
//read first row key
for(int v = 0; v < m; v++){
  keyB += seqB[v];
}
seqBMap->insert(std::pair<string,vector<int> >(keyB,mapDefaultVal) );
map<string,vector<int> >::iterator itPairB = seqBMap->find(keyB);
//--> copy of the first row in the matrix.
for(unsigned int i = 0; i < firstBRow->size() ; i++){
		(itPairB->second)[(*firstBRow)[i]]++;
		matrixB[0][i] = (*firstBRow)[i];
}
//------------------------------------------------------------------------
//---------calcolo valore kernel-----------------------------------------

//mappe per valori congiunti
unordered_map<string,vector<int> > crsSeqAMap;
unordered_map<string,vector<int> > crsSeqBMap;
//-----------------------------------------------------------------
//--> calcolo della prima riga congiunta
vector<int>* firstRow =naiveSingleKdiffMatch(seqB, seqA, m);
keyA = "";
//leggo chiave A
for(int v = 0; v < m; v++){
  keyA += seqA[v];
}
```

Figura 5-11: Ottimizzazione calcolo AMK – parte 2

```
unordered_map<string,vector<int> >::iterator crsItPairA;
vector<int> aFirstMerVect;
aFirstMerVect.resize(m + 1);
unordered_map<string,vector<int> >::iterator crsItPairB;

//-----------------------------------------------
//--> analisi prima riga
for(unsigned int i = 0; i < firstRow->size() ; i++){
		keyB = "";
		//leggo chiave B
		for(int v = 0; v < m; v++){
		  keyB += seqB[i + v];
		}
		//aggiorno valore per A
		aFirstMerVect[(*firstRow)[i]]++;
		M[0][i] = (*firstRow)[i];
}
delete firstRow;

crsSeqAMap.insert(std::pair<string,vector<int> >(keyA,aFirstMerVect) );
crsItPairA = crsSeqAMap.find(keyA);
//-----------------------------------------------------------
//--> analisi prima colonna
vector<int>* firstColumn =naiveSingleKdiffMatch(seqA, seqB, m);
//Inizializzazione matrice e prima colonna
keyB = "";
for(int v = 0; v < m; v++){
  keyB += seqB[v];
}
vector<int> bFirstMerVect;
bFirstMerVect.resize(m + 1);

for(int i = 1; i <= lenA - m ; i++){
		M[i][0] = (*firstColumn)[i];
		//aggiorno valore per B
		bFirstMerVect[(*firstColumn)[i]]++;

}

crsSeqBMap.insert(std::pair<string,vector<int> >(keyB,bFirstMerVect) );
//liberazione memoria
delete firstColumn;
int bIndex = 1;

//-------------------------------------------------------------------------------
// Ciclo principale di calcolo matrici A,B e AB
//-------------------------------------------------------------------------------
for(int i = 1; i <= lenA - m ; i++){
		//se è la prima occorrenza del mer in A
		if(mersPosA[i] == i){

				//leggo chiave A
				keyA = "";
				for(int v = 0; v < m; v++){
				  keyA += seqA[i + v];
				}

				//@@ per matrice incrociata AB
				vector<int> merVect;
				merVect.resize(m + 1);
				//imposto valore associato alla prima colonna nel vettore
				merVect[M[i][0]]++;

				//@@ per matrice A
				matrixA[i][0] = (*firstARow)[i];
				vector<int> valsA;
				valsA.resize(m + 1);
```

Figura 5-12: Ottimizzazione calcolo AMK – parte 3

```
valsA[matrixA[i][0]]++;
//*********************************************************
//@@ per matrice B
vector<int> valsB;
valsB.resize(m + 1);

bool bCondition =  (bIndex  <= (lenB - m) && mersPosB[bIndex] == bIndex);

if(bCondition){
        //read the row key
        keyB = "";
        for(int v = 0; v < m; v++){
          keyB += seqB[bIndex + v];
        }

        matrixB[bIndex][0] = (*firstBRow)[bIndex];

        valsB[matrixB[bIndex][0]]++;
}
//*********************************************************
//per ogni colonna della matrice incrociata
for(int j = 1; j <= lenB - m ; j++){

        //@@ Matrice congiunta
        int val = M[i-1][j-1];

        if(seqA[i+m-1] !=  seqB[j+m-1]){
                val++;
        }
        if(seqA[i-1] !=  seqB[j-1]){
                val--;
        }
        M[i][j] = val;
        //aggiorno valore per A
        merVect[val]++;
        //@@ Matrice di A
        int aVal = matrixA[i-1][j-1];
        if(seqA[i+m-1] !=  seqA[j+m-1]){
                aVal++;
        }
        if(seqA[i-1] !=  seqA[j-1]){
                aVal--;
        }
        matrixA[i][j] = aVal;
        //update mismatch vector of A
        valsA[aVal]++;
        //*********************************************************
        //per B
        if(bCondition){
                //initialize matrix[i,j]
                int bVal = matrixB[bIndex-1][j-1];
                if(seqB[bIndex+m-1] !=  seqB[j+m-1]){
                        bVal++;
                }
                if(seqB[bIndex-1] !=  seqB[j-1]){
                        bVal--;
                }
                matrixB[bIndex][j] = bVal;
                //update mismatch vector
                valsB[bVal]++;
        }
        //*********************************************************
}
```

Figura 5-13: Ottimizzazione calcolo AMK – parte 4

```
                        //per ogni colonna rimanente di A
                        for(int j = lenB - m + 1; j <= lenA - m ; j++){
                                //initialize matrix[i,j]
                                int aVal = matrixA[i-1][j-1];

                                if(seqA[i+m-1] !=  seqA[j+m-1]){
                                        aVal++;
                                }
                                if(seqA[i-1] !=  seqA[j-1]){
                                        aVal--;
                                }
                                matrixA[i][j] = aVal;
                                //update mismatch vector
                                valsA[aVal]++;
                        }
                        //insert in the map the pair computed
                        seqAMap->insert(std::pair<string,vector<int> >(keyA,valsA) );
                        crsSeqAMap.insert(std::pair<string,vector<int> >(keyA,merVect) );
                        //*********************************************************
                        if(bIndex  <= (lenB - m)){
                                if(mersPosB[bIndex] == bIndex){
                                        //insert in the map the pair computed
                                        seqBMap->insert(std::pair<string,vector<int> >(keyB,valsB));
                                }
                                //incremento indice riga di b
                                bIndex++;
                        }//*********************************************************
                }
        }
        //*********************************************************
        //analisi righe rimanenti di B
        for(; bIndex <= lenB - m ; bIndex++){
                if(mersPosB[bIndex] == bIndex){
                        //read the row key
                        keyB = "";
                        for(int v = 0; v < m; v++){
                          keyB += seqB[bIndex + v];
                        }
                        vector<int> valsB;
                        valsB.resize(m + 1);
                        matrixB[bIndex][0] = (*firstBRow)[bIndex];
                        //aggiornamento valore per la prima colonna
                        valsB[matrixB[bIndex][0]]++;
                        for(int j = 1; j <= lenB - m ; j++){
                                //initialize matrix[i,j]
                                int val = matrixB[bIndex-1][j-1];
                                if(seqB[bIndex+m-1] !=  seqB[j+m-1]){
                                        val++;
                                }
                                if(seqB[bIndex-1] !=  seqB[j-1]){
                                        val--;
                                }
                                matrixB[bIndex][j] = val;
                                //update mismatch vector
                                valsB[val]++;
                        }
                        seqBMap->insert(std::pair<string,vector<int> >(keyB,valsB) );
                }
        }
        //*********************************************************
        //Aggiornamento info mer di B per ogni colonna (j indice colonna)
        for(int j = 0; j <= lenB - m ; j++){
                //se è la colonnna associata alla prima occorrenza del mer j in B
                if(mersPosB[j] == j){
                        keyB = "";
                        //leggo chiave B
                        for(int v = 0; v < m; v++){
                          keyB += seqB[j + v];
                        }
                        vector<int> valsVect;
                        valsVect.resize(m + 1);
```

Figura 5-14: Ottimizzazione calcolo AMK – parte 5

```
        //per ogni riga
                        for(int i = 0; i <= lenA - m ; i++){
                                valsVect[M[i][j]]++;
                        }
                        crsSeqBMap.insert(std::pair<string,vector<int> >(keyB,valsVect) );
                }
        }
        double kernelVal = 0;
        double adjX = 0;
        double adjY = 0;
//Massimo k = 4
double prests[5][5];

prests[0][0] = 1;prests[0][1] = 1;prests[0][2] = 1;prests[0][3] = 1;prests[0][4] = 1;
prests[1][0] = 1;prests[1][1] = 1;prests[1][2] = 1;prests[1][3] = 1;prests[1][4] = 1;
prests[2][0] = 1;prests[2][1] = 1;prests[2][2] = 1;prests[2][3] = 1;prests[2][4] = 1;
prests[3][0] = 1;prests[3][1] = 1;prests[3][2] = 1;prests[3][3] = 1;prests[3][4] = 1;
prests[4][0] = 1;prests[4][1] = 1;prests[4][2] = 1;prests[4][3] = 1;prests[4][4] = 1;

        map<string, vector<int> >::iterator itOrdPairA;
        map<string, vector<int> >::iterator itOrdPairB;

        //*************************************************************
        //*************************************************************
        //----------CALCOLO VALORE KERNEL----------------------
        itOrdPairA = seqAMap->begin();
        itOrdPairB = seqBMap->begin();

        while(itOrdPairA != seqAMap->end() && itOrdPairB != seqBMap->end()){
                int compVal = (itOrdPairA->first).compare(itOrdPairB->first);
                //se si ha una entry con la stessa parola per entrambe le liste allora
                //semplicemente moltiplico il numero di occorrenze tra loro
                if(compVal == 0){
                  double sum  = 0;
                  for(int i = 0; i <= k; i++){
                          for(int j = 0; j <= k; j++){
                                  sum += prests[i][j] *calcComb(itOrdPairA->second[i],
                                                                  itOrdPairB->second[j]);
                          }
                  }

                  //--------------------------------
                  kernelVal += sum;
                  itOrdPairA++;
                  itOrdPairB++;
                //la chiave di A è minore di quella di B e non è presente in esso.
                }else if(compVal < 0){

                   double sum  = 0;
                   keyA = itOrdPairA->first;
                          crsItPairA = crsSeqAMap.find(keyA);
                   for(int i = 0; i <= k; i++){
                          for(int j = 0; j <= k; j++){
                                  int val = prests[i][j] *calcComb(itOrdPairA->second[i],
                                                                  crsItPairA->second[j]);
                                  sum  += val;
                                  adjY +=calcComb(crsItPairA->second[j],
                                          crsItPairA->second[j]);
                          }
                   }
                   //--------------------------------
                   kernelVal += sum;
                   itOrdPairA++;
                }else{
                          double sum  = 0;
                          keyB = itOrdPairB->first;
                          crsItPairB = crsSeqBMap.find(keyB);
```

Figura 5-15: Ottimizzazione calcolo AMK – parte 6

```
                for(int i = 0; i <= k; i++){
                        for(int j = 0; j <= k; j++){
                                int val       = prests[i][j] *calcComb(itOrdPairB->second[i],
                                                                        crsItPairB->second[j]);
                                sum         += val;
                                adjX        +=calcComb(crsItPairB->second[j],crsItPairB->second[j]);
                        }
                }
                //--------------------------------
                kernelVal += sum;
                itOrdPairB++;
            }
}

while(itOrdPairA != seqAMap->end()){
            double sum  = 0;
            keyA = itOrdPairA->first;
            crsItPairA = crsSeqAMap.find(keyA);
            for(int i = 0; i <= k; i++){
                        for(int j = 0; j <= k; j++){
                          int val = prests[i][j] *calcComb(itOrdPairA->second[i],crsItPairA->second[j]);
                          sum  += val;
                          adjY +=calcComb(crsItPairA->second[j],crsItPairA->second[j]);

                        }
            }
            //--------------------------------
            kernelVal += sum;
            itOrdPairA++;
}

while(itOrdPairB != seqBMap->end()){
            double sum  = 0;

            keyB = itOrdPairB->first;
            crsItPairB = crsSeqBMap.find(keyB);
            for(int i = 0; i <= k; i++){
                        for(int j = 0; j <= k; j++){
                        int val       = prests[i][j] *calcComb(itOrdPairB->second[i],crsItPairB->second[j]);
                        sum         += val;
                        adjX        +=calcComb(crsItPairB->second[j],crsItPairB->second[j]);

                        }
            }
            //--------------------------------
            kernelVal += sum;
            itOrdPairB++;
}

KernelResult result(kernelVal,adjX,adjY);
//-----------------------------------------------------------------------
//DD--> liberazione memoria

//A memory clean
delete firstARow;
for(int i = 0; i <= lenA - m;i++){
            if(mersPosA[i] == i){
              delete matrixA[i];
            }
}
```

Figura 5-16: Ottimizzazione calcolo AMK – parte 6

```
        delete matrixA;
        //B memory clean
        delete firstBRow;

        for(int i = 0; i <= lenB - m;i++){
                if(mersPosB[i] == i){
                  delete matrixB[i];
                }
        }

        delete matrixB;
        delete seqAMap; delete seqBMap;

        //liberazione di M
        for(int i = 0; i <= lenA - m; i++){
                if(mersPosA[i] == i){
                        delete M[i];
                }
        }
        delete M;
        delete mersPosA; delete mersPosB;

        return  result;
}
```

Figura 5-17: Ottimizzazione calcolo AMK – parte 7

6 Struttura del software per l'esecuzione dei TEST

Nel seguito si descrive la struttura del software utilizzato per effettuare i test di rilevazione dell'omologia remota. Esso permette di utilizzare sia i database sotto forma di file di testo introdotti in 4.5.1e 4.5.2, che il database MySql SCOP 1.75B, impostando il training set e il test set grazie alle interrogazioni introdotte in 3.5.

In Figura 6-1 vi è la struttura della directory principale del codice sorgente del software. La cartella principale è proteinSvmClassifier, contenente il file main-test.cpp che ha al suo interno il metodo main per l'avvio del software.

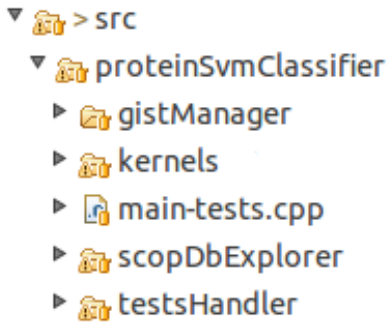


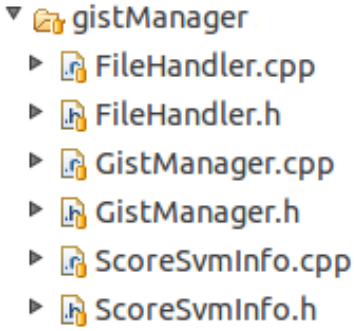
Figura 6-1: Cartelle principali

Essa inoltre contiene le quattro cartelle corrispondenti alle sezioni in cui è diviso il programma. **gistManager** contiene le classi che gestiscono l'interfaccia dell'applicativo col software GIST SVM, esse automatizzano completamente il processo di test eseguendo il training dell'SVM, il test e infine scrivendo i risultati ottenuti (confusion matrix e valori ROC). **kernels** contiene le classi per il calcolo degli string kernels.

scopDbExplorer contiene le classi che gestiscono le interfacce tra il software e i database SCOP, siano essi su file o su database MySql. Infine **testHandler** contiene le classi necessarie all'esecuzione dei test.

6.1 Sezione gistManager

Tale sezione contiene le classi necessarie a gestire il software GistManager e per gestire i file e cartelle su disco in con semplicità.



6.1.1 Classe FileHandler

E' una classe di utilità contenente metodi per la scrittura e l'analisi di file e dati su disco. I suoi metodi principali sono:

static string **getDateTimeString**(): Restituisce una stringa nel formato anno-mese-giorno_Ora:Min:Sec .

static string **getWorkingDirPath**():Restituisce percorso working directory.
static bool **createDirectory**(string dir_root_path, string dir_name): Crea la cartella dir_name dentro la cartella di percorso assoluto dir_root_path. Se l'operazione non va a buon fine restituisce false.

static bool **deleteDirectory**(string dir_root_path, string dir_name): Cancella la cartella dir_name dentro la cartella dir_root_path. Se l'operazione non va a buon fine restituisce false.

static bool **fileExist**(string file_path): Verifica l'esistenza del file di percorso assoluto file_path

static bool **dir_Exist**(string dir_path): Verifica l'esistenza del file di percorso assoluto file_path

static void **printDirectoryContent**(string dir_path): Scrive su stdout il contenuto della directory dir_path

static void **printDirectoryPermissions**(string dir_path): Scrive su stdout i permessi della directory dir_path

static bool **writeStringOnFile**(string dir_path, string file_name, string& content): Crea il file di nome file_name all'interno della cartella di percorso assoluto dir_path e scrive al suo interno la stringa content. Restituisce true se l'operazione va a buon fine, false altrimenti.

static bool **appendStringOnFile**(string dir_path, string file_name, string& content): Aggiunge al termine del file di nome file_name all'interno della cartella di percorso assoluto dir_path e scrive al suo interno la stringa content. Restituisce true se l'operazione va a buon fine.

static vector<string>* **getFileRows**(string file_path): Analizza riga per riga il contenuto di un file di testo di percorso file_path e restiuisce un vettore contenente le righe del file sotto forma di oggetti string.

6.1.2 Classe GistManager

La classe funziona da interfaccia per interagire col software Gist SVM.

void **createLabelsFile**(vector<long long>* labels, vector<int>* classes, string fileName, string comment): Crea nella cartella workingDir un file di nome fileName. Nell'intestazione scrive la stringa comment, preceduta da '#'. Nella seconda riga scrive la stringa "angle \t --\n". Le righe successive sono nel formato etichetta , classe d'appartenenza: "labels[i] \t classes[i] \n"

void **createSelfKernelFile**(vector<long long>* labels, vector<double>* values, string fileName):
Crea nella cartella workingDir un file di nome fileName. Esso è nel formato necessario a Gist per gli auto-valori di kernel. Nella prima riga inserisce il commento #vuoto. Nella seconda riga scrive la stringa "angle \t --\n". Le righe successive sono nel formato etichetta , valore: "labels[i] \t classes[i] \n"

void **createKernelMatrixFile**(vector<long long>* rowsLabels, vector<long long>* columnsLabels,int** matrix, string fileName, string comment): Crea nella cartella workingDir un file di nome fileName. Esso è nel formato di input di GIST per le kernel matrix. La prima riga contiene la stringa comment, preceduta dal carattere '#'. La seconda riga contiene come primo valore "angle \t", seguito dai valori columnsLabels separati da tab (\t).
Le righe successive sono nel formato:
rowsLabels[i] \t matrix[i][0] \t matrix[i][1] \t ... \t matrix[i][columnsLabels->size()-1]

void **createKernelMatrixFile**(vector<long long>* rowsLabels, vector<long long>* columnsLabels,double** matrix, string fileName, string comment): Crea nella cartella workingDir un file di nome fileName. Esso è nel formato di input di GIST per le kernel matrix. La prima riga contiene la stringa comment, preceduta dal carattere '#'. La seconda riga contiene come primo valore "angle \t", seguito dai valori columnsLabels separati da tab (\t). Le righe successive sono nel formato:
rowsLabels[i] \t matrix[i][0] \t matrix[i][1] \t ... \t matrix[i][columnsLabels->size()-1]

void **appendToKernelMatrixFile**(vector<long long>* rowsLabels,double** matrix, int columns, string fileName): Aggiunge al file di nome fileName nella working directory le righe nel formato:
rowsLabels[i] \t matrix[i][0] \t matrix[i][1] \t ... \t matrix[i][columnsLabels->size()-1]

void **exec_gistTrainSvm**(string outputFileName, string kernelMatrix_fileName, string class_fileName, bool normalize): Esegue il programma gist-train-svm, passando in ingresso come parametri:

- matrix
- constant 0
- nonormalize: se normalize è impostato a false
- train kernelMatrix_fileName
- class class_fileName

Scrive nel file outputFileName l'output del programma.

void **exec_gistClassify**(string outputFileName, string weights_fileName, string kernelMatrix_fileName): Esegue il programma gist-classify, passando in ingresso come parametri:

- learned: kernelMatrix_fileName
- test: weights_fileName

scrive nel file outputFileName l'output del programma.

void **exec_gistClassifyNormalized**(string outputFileName, string weights_fileName, string kernelMatrix_fileName, string selfTrain_fileName, string selfTest_fileName):
Esegue il programma gist-classify, passando in ingresso come parametri

- selftrain: selfTrain_fileName
- selftest : selfTest_fileName
- learned: kernelMatrix_fileName
- test: weights_fileName

Scrive nel file outputFileName l'output del programma.

void **exec_gistScoreSvm**(string outputFileName, string testLabels_fileName, string testPredict_fileName, string weights_fileName): Esegue lo script gist-score-svm, passando in ingresso come parametri:

- test testLabels_fileName testPredict_fileName weights_fileName

Scrive nel file outputFileName l'output dello script.

bool **setGistBinDir**(string gistBinPath): Imposta la cartella file binari di Gist SVM. Restituisce falso se la directory impostata è errata.

ScoreSvmInfo* **scoreSvm_outputAnalyzer**(vector<string>* file_rows): Analizza l'output del file scoreSvm. Le righe file_rows sono le righe del file dato in output dallo script gist-score-svm. Restituisce in output un oggetto di tipo ScoreSvmInfo che contiene le informazioni del file in formato gestibile dal software in maniera agevole.

bool **setWorkingDir**(string dir_path): Imposta la directory di lavoro per la classe. Restituisce true se la directory è stata impostata correttamente

6.1.3 classe ScoreSvmInfo

Contiene la descrizione dell'output dello script gist-score-svm. E' costituito da metodi getter/setter per i valori significativi in output dallo script.

int **getNegSupVectNum**() const; void **setNegSupVectNum**(int negSupVectNum): imposta/restituisce il numero di vettori di supporto negativi

int **getNegTrainSetDim**() const; void **setNegTrainSetDim**(int negTrainSetDim): imposta/restituisce la dimensione dell'insieme di train negativo.

int **getPosSupVectNum**() const; void **setPosSupVectNum**(int posSupVectNum): imposta/ restituisce la dimensione del numero di vettori di supporto positivi.

int **getPosTrainSetDim**() const; void **setPosTrainSetDim**(int posTrainSetDim): imposta/ restituisce la dimensione dell'insieme di trianing positivo.

int **getTestFn**() const; void **setTestFn**(int testFn): imposta/restituisce il numero di esempi di test falsi negativi.

int **getTestFp**() const; void **setTestFp**(int testFp): imposta/restituisce il numero di esempi di test falsi positivi.

int **getTestTn**() const; void **setTestTn**(int testTn): imposta/restituisce il numero di esempi di test veri negativi.

int **getTestTp**() const; void **setTestTp**(int testTp): imposta/restituisce il numero di esempi di test veri positivi.

float **getTestRoc**() const; void **setTestRoc**(float testRoc): imposta/restituisce il valore ROC per il test

int **getTrainFn**() const; void **setTrainFn**(int trainFn): imposta/restituisce il numero di esempi di train falsi negativi.

int **getTrainFp**() const; void **setTrainFp**(int trainFp): imposta/restituisce il numero di esempi di train falsi positivi.

int **getTrainTn**() const; void **setTrainTn**(int trainTn): imposta/restituisce il numero di esempi di train true negativi.

int **getTrainTp**() const; void **setTrainTp**(int trainTp): imposta/restituisce il numero di esempi di train true positive.

float **getTrainRoc**() const: void **setTrainRoc**(float trainRoc): imposta/restituisce il valore ROC per il training

6.2 Sezione scopDBExplorer

6.2.1 Classe ProteinDomain

ProteinDomain contiene le informazioni associate a un dominio PDB, quali il suo identificativo e la sua sequenza di amminoacidi.
I suoi attributi privati sono:

- long long astral_domain_id: contiene l'id del dominio nella tabella astral_domain
- const char* astral_sid: contiene il sid astral (ad es. 'd1dlwa_'), ha al massimo 8 caratteri
- long long scop_family_id: contiene l'id del nodo in scop_node rappresentante la famiglia scop contenente il dominio
- seq_style source_type: è lo stile della sequenza
- const char* sccs: Contiene l'sccs del dominio(ad es: a.1.1.1)
- const char* seq: Memorizza la sequenza di caratteri della proteina, nota: la stringa più lunga è di 4128 caratteri
- int seq_length; memorizza la lunghezza di seq

Essi sono impostabili attraverso il costruttore:
ProteinDomain(long long astral_domain_id, const char* astral_sid, long long scop_family_id, seq_style source_type, const char* sccs, const char* seq, int seq_length);

e sono accessibili con metodi getter, ad esempio getSeqLength() per ottenere la lunghezza e getScopFamilyId() per ottenere l'id della famiglia.

6.2.2 Classe Scop153Parser

Tale classe effettua il parsing dei file che descrivono il database 1.53 (introdotto in 3.6.2) e reperibile all'indirizzo [26] al fine di convertirlo nel formato descritto in 4.5.1, l'unica differenza è nei file di train e test negativi, che vengono creati per ogni famiglia, quindi memorizzati all'interno della cartella di una famiglia e non nella cartella superiore di fold.

Il suo unico metodo pubblico è:

void **parseAndConvertDB**(string tableFile, string seqsFile, string baseDir);

i suoi parametri sono:

- tableFile: percorso assoluto del file all_seq_fam_membership.txt
- seqsFile: percorso assoluto del file: scop.1.53.1e-25.fasta

6.2.3 Classe ScopDbQueries

Tale classe serve solo da contenitore per le interrogazioni SQL necessarie per ottenere dal database MySql di SCOP 1.75B i data set necessari per il test di rilevazione omologia remota. Tali interrogazioni sono descritte in 3.5. Le query sono parametrizzate, riferirsi direttamente al codice sorgente per approfondimenti. Perché tali query funzionino lo schema del database deve chiamarsi ASTRAL.

6.2.4 Classe ScopDbExplorer

Tale classe è un interfaccia al database MySql SCOP 1.75B.

void **setMysqlPort**(short port): Imposta la porta TCP del server MySql.

int **setMySqlAddress**(short a,short b,short c, short d): Imposta l'indirizzo IPv4 del server Mysql, condizioni: 0 <= a,b,c,d <= 255. Restituisce -1 se l'indirizzo non è impostato correttamente,0 altrimenti.

int **setMySqlAddress**(ip::address ip): Imposta l'indirizzo IPv4 del server Mysql. Parametro in ingresso di tipo boost asio ip address. Restituisce -1 se l'indirizzo non è impostato correttamente, 0 altrimenti.

int **setMySqlAddress**(string ip): Imposta l'indirizzo IPv4 del server Mysql. Parametro in ingresso di tipo stringa. Restituisce -1 se l'indirizzo non è impostato correttamente,0 altrimenti.

void **setMySqlUser**(string user): Imposta il nome utente per accedere al database.

void **setMySqlPassword**(string pwd): Imposta la password per accedere al database.

void **connectToMysql**() throw (SQLException): Effettua la connessione al database MySql. In caso di errore di connessione, ad esempio per password errata o database non raggiungibile, lancia un eccezzione di tipo SQLException.

void **disconnectFromMysql**():Effettua la disconnessione dal database.

vector<ProteinDomain*>* **getFamily_DomainsPctSs**(unsigned long long familyId,short release, short pct_identical) throw (SQLException): fornisce le sequenze di una famiglia appartenenti al sotto-insieme %ID di percentuale pct_identical.

vector<ProteinDomain*>* **getFamily_DomainsEvalueSs**(unsigned long long familyId,short release, double e_value) throw (SQLException): fornisce le sequenze di una famiglia appartenenti al sotto-insieme BLAST E-value di valore pct_e_value.

vector<ProteinDomain*>* **getNotInF_SFDomainsPctSs**(unsigned long long familyId,short release, short pct_identical) throw (SQLException): fornisce le sequenze di una superfamiglia di una data famiglia, eccetto quelle appartenenti a quest'ultima. Le ottiene dal sotto-insieme %ID di percentuale pct_identical.

vector<ProteinDomain*>* **getNotInF_SFDomainsEvalueSs**(unsigned long long familyId,short release, double e_value) throw (SQLException): fornisce le sequenze di una superfamiglia di una data famiglia, eccetto quelle appartenenti a quest'ultima. Le ottiene dal sotto-insieme BLAST E-value di valore pct_e_value.

vector<ProteinDomain*>* **getNotInSameFoldOfF_DomainsPctSs**(unsigned long long familyId,short release, short pct_identical) throw (SQLException): fornisce le sequenze appartenenti ai fold complementari al fold di una data famiglia. Le ottiene dal sotto-insieme significativo %ID di percentuale pct_identical.

vector<ProteinDomain*>* **getNotInSameFoldOfF_DomainsEvalueSs**(unsigned long long familyId,short release, double e_value) throw (SQLException): fornisce le sequenze appartenenti ai fold complementari al fold di una data famiglia. Le ottiene dal sotto-insieme significativo BLAST E-value di valore pct_e_value.

6.2.5 Classe ScopFileExplorer

Tale classe è un interfaccia al database ricavato da SCOP 1.37 descritto in 3.6.1 e al database ricavato da SCOP 1.53 descritto in 3.6.2 e creato mediante 6.2.2.
Gli identificativi delle famiglie sono passati in formato numerico. Sono costituiti da 7 cifre e identificano gli sccs nel seguente modo:

cOOssFF

dove:

- cc: id numerico della classe
- OO: id numerico della fold
- ss: id numerico della superfamily

- FF: id numerico delle family

static int **getSccsClass**(int familySccsId): Dato un numero in formato cOOssFF restituisce c

static int **getSccsFold**(int familySccsId): Dato un numero in formato cOOssFF restituisce OO

static int **getSccsSuperFamily**(int familySccsId): Dato un numero in formato cOOssFF restituisce ss

static int **getSccsFamily**(int familySccsId): Dato un numero in formato cOOssFF restituisce FF

static string **getFoldSccs**(int familySccsId): Dato un numero in formato cOOssFF restituisce c.OO

static string **getFamilySccs**(int familySccsId): Dato un numero in formato cOOssFF restituisce ss.FF

static string **toLowerCase**(string s): converte s in caratteri tutti minuscoli

bool **setfilesDbDir**(string dirPath): Imposta la cartella dove si trova il database

vector<string> **getFamilyFiles**(int familySccsId): Restituisce un vettore con i percorsi dei file che contengono le sequenze di una famiglia (test set positivo)

vector<string> **getSuperFamilyFiles**(int familySccsId): Restituisce un vettore con i percorsi dei file che contengono le sequenze di una superfamiglia (train set positivo)

vector<string> **getNegativeTestFiles**(int familySccsId, bool use0_1,bool useInnerFiles): Restituisce un vettore con i percorsi dei file che contengono le sequenze di test negative.
use0_1: false si su usano gli insiemi fold0, true sei si usano fold1
useInnerFiles: se true cerca file con gli esempi negativi dentro la cartella della famiglia, anzichè in quella del fold. (per SCOP 1.53)

vector<string> **getNegativeTrainFiles**(int familySccsId, bool use0_1,bool useInnerFiles): Restituisce un vettore con i percorsi dei file che contengono le sequenze di train negative.
use0_1: false si su usano gli insiemi fold0, true sei si usano fold1
useInnerFiles: se true cerca file con gli esempi negativi dentro la cartella della famiglia, anzichè in quella del fold. (per SCOP 1.53)

static vector<string>* **readSequencesFile**(string filePath): Legge le sequenze di proteine contenute nel file di percorso filePath in formato FASTA e le restituisce in un vettore.Le sequenze di checksum contenute nei file sono ignorate.

vector<string>* **readFirstSuperFamilyFile**(int familySccsId): Restituisce un vettore con le sequenze di superfamiglia contenute nel file il cui percorso è nell'indice 0 del vettore restituito da getFamilyFiles(int familySccsId)

vector<string>* **readFirstFamilyFile**(int familySccsId): Restituisce un vettore con le sequenze di superfamiglia contenute nel file il cui percorso è nell'indice 0 del vettore restituito da getSuperFamilyFiles(int familySccsId)

vector<string>* **readFirstNegativeTrainFile**(int familySccsId, bool use0_1): Restituisce un vettore con le sequenze di superfamiglia contenute nel file il cui percorso è nell'indice 0 del vettore restituito da getNegativeTestFiles(int familySccsId, bool use0_1,bool useInnerFiles)

vector<string>* **readFirstNegativeTestFile**(int familySccsId, bool use0_1): Restituisce un vettore con le sequenze di superfamiglia contenute nel file il cui percorso è nell'indice 0 del vettore restituito da getNegativeTrainFiles(int familySccsId, bool use0_1,bool useInnerFiles)

6.3 Sezione kernels

In tale sezione vi sono le classi contenenti le funzioni per il calcolo dei kernel.

6.3.1 Classe MismatchKernel

Tale classe contiene i metodi per il calcolo del mismatch kernel e delle kernel matrix mediante mismatch kernel.

static KernelResult **computeKernel**(const char* seqA,int lenA,const char* seqB, int lenB, int merSize, int kdiff): Calcola il mismatch kernel tra le due sequenze seqA e seqB, con lunghezza di stringa merSize e numero massimo di mismatch kDiff.

static void **computeKernelMatrix**(vector<const char*>* seqs, int merSize, int kdiff, double** kernelMatrix): Calcola il kernel tra tutte le possibili coppie di sequenze nel vettore seqs. I valori vengono memorizzati in kernelMatrix. Tale parametro deve essere una matrice di dimensioni seqs.size() x seqs.size() con tutti i valori inizializzati a 0. merSize è la lunghezza delle tuple. kDiff è il numero di massimo di mismatch che il kernel ammette.

static void **computeRCKernelMatrix**(vector<const char*>* rowSeqs,vector<const char*>* colSeqs, int merSize, int kdiff, double** kernelMatrix): Calcola il kernel tra tutte le possibili coppie di sequenze tra quelle del vettore rowSeqs e quelle del vettore colSeqs. I valori vengono memorizzati in kernelMatrix. Tale parametro deve essere una matrice di dimensioni rowSeqs.size() x colSeqs.size() con tutti i valori inizializzati a 0.

6.3.2 Classe AMKKernel

Contiene i metodi per il calcolo dei kernel definiti in 4.6, 4.7, 4.8, 4.9 e 4.10.

static KernelResult **computeKernelFast**(const char* seqA,int lenA,const char* seqB, int lenB, int merSize, int kdiff): calcola il kernel combinatorio con ordine tra le due sequenze seqA e seqB impostando i pesi dei contributi tutti a 1. L'algoritmo è simile a quello introdotto in 5.3, ma è ottimizzato in modo da ottenere una diminuzione del tempo di esecuzione considerevole rispetto a quest'ultimo.

static vector<int>* **naiveSingleKdiffMatch**(char* inText, char* pattern, int m): per ogni tupla di lunghezza m nella sequenza inText calcola il numero di mismatch con la m-tupla pattern. Resituisce un vettore v di lunghezza len(inText) - m + 1. v[i]: numero di mismatch tra inText[i...i+m-1] e pattern

static map<string,vector<int> >* **allKDiffsMap_Fast**(char* seqA, int lenA, int m, int k): è l'algoritmo introdotto in 5.1.

La classe *AMKKernelDev* contiene altri metodi utili, non direttamente utilizzati per il calcolo dei kernel (ad esempio con il metodo computerKernelFast). I più significativi sono:

vector<unordered_map<string,vector<int> > > **allcross_KDiffs**(const char* inSeqA,int lenA,const char* inSeqB, int lenB, int m, int k): è il metodo descritto in 5.2.

KernelResult **computeKernelFast** (const char* inSeqA,int lenA,const char* inSeqB,int lenB, int m, int k): è l'algoritmo descritto in 5.3.

6.3.3 Classe KernelResult

Tale classe memorizza il valore calcolato da una funzione di kernel e i termini correttivi, descritti in 0.

KernelResult(double kernelVal,double adjX,double adjY): costruttore. KernelVal è il valore del kernel, adjX e adjY i termini correttivi. I valori sono accessibili direttamente tramite gli attributi: kernelVal, adjX e adjY.

6.4 Sezione testHandler

La sezione testHandler contiene le classi necessarie a gestire i test per le famiglie SCOP dei database SCOP 1.37, 1.53 e 1.75B. Vi sono due classi principali, RhdFamilyTest e SerialTestHandler, e una classe d'utilità RhdTestsInfos.

6.4.1 Classe RhdTestsInfos

La classe descrive i risultati di un test. Contiene un oggetto ScoreSvmInfo per memorizzare il valore di ROC per il test set, i valori TP, TN, FP, FN ecc e inoltre memorizza il tempo di esecuzione impiegato dall'algoritmo per calcolare le matrici di kernel di train e test, essa è costituita da un costruttore e due metodi accessori:

RhdTestInfos(ScoreSvmInfo* info, long matricesTime): il costruttore riceve in ingresso un oggetto di tipo ScoreSvmInfo e il tempo di calcolo matricesTime

long **getKernelMatricesTime**() const: Restituisce il tempo di calcolo

ScoreSvmInfo* **getScoreSvmInfo**():Restituisce l'oggetto ScoreSvmInfo contenente le performance del test.

6.4.2 Classe RhdFamilyTest

Data una particolare famiglia in ingresso, effettua il test di rilevamento di omologia remota per tale famiglia. Essa è costituita da un costruttore e un insieme di metodi per l'impostazione dei parametri di test e da metodi per l'esecuzione dei test. Il primo insieme di metodi è:

RhdFamilyTest(long long familyId, string baseDir, bool useExistingFolder, string gistBin): Costruttore di base. Imposta l'identificativo della famiglia e la directory di base per i tests. La classe funziona in due modalità:

1. Crea la cartella con i file di test, calcola le kernel matrix e i label file.
2. Legge il contenuto di una cartella di test già esistente e permette di eseguire o ri-eseguire i test in tale cartella.

La modalità si seleziona col parametro useExistingFolder.

- Se true: Nel caso in cui la cartella non sia esistente o non contenga il file descrittivo ritorna eccezione.
- Se false: Se la cartella esiste la cancella.

bool **setReleaseId**(short releaseId): Permette di impostare la release SCOP su cui eseguire i test se si utilizza il database MySql SCOP 1.75B. La release può essere 1-13. Valido solo se useExistingFolder = false

bool **setSequencesPct**(int pct): Imposta come sotto-insieme significativo da cui ottenere le sequenze il %ID di livello pct. Valido solo se useExistingFolder = false

bool **setSequencesEvalue**(double e_value): Imposta come sotto-insieme significativo da cui ottenere le sequenze quello di e-value pari a e_value. Valido solo se useExistingFolder = false

bool **setNormalizeKernel**(bool val): Fornire in ingresso true se si vuole che il kernel venga normalizzato.

bool **setgestNormalizeKernel**(bool val): Fornire in ingresso true se si vuole che il kernel venga normalizzato da Gist SVM.

bool **setKernelType**(kernel_type kType): Imposta il tipo di funzione di kernel su cui effettuare i test

bool **setMerSize**(int size): Imposta la dimensione dei mer delle sequenze per il kernel oggetto di test.

bool **setKdiff**(int kNum): Imposta il numero massimo di mismatch ammessi dal kernel oggetto di test.

bool **setNegativeSetSize**(int size): Imposta la dimensione totale degli esempi negativi.

bool **setTestExamples_selectionStrategy**(testExamples_selectionStrategy selStrategy): Imposta la strategia di selezione degli esempi di test negativi dagli esempi negativi ottenuti mediante interrogazione da database SCOP 1.75B

I metodi per l'esecuzione dei test sono:

bool **prepareGistExecution**(ScopDbExplorer& explorer): Crea le matrici kernel di train e test scrivendole su file, preparando l'esecuzione dei programmi GIST per il train, test e score. Utilizza come sorgente dati il database MySql SCOP 1.75B.

bool **prepareGistExecution**(short releaseId, bool byPct, int pct, double e_value, bool normalizeKernel, bool gestNormalizeKernel, ScopDbExplorer& explorer, testExamples_selectionStrategy selStrategy, kernel_type kType): Invoca il metodo prepareGistExecution(ScopDbExplorer& explorer) impostando prima i parametri di configurazione passati in ingresso.

bool **prepareGistExecution_FileSource**(string filesDir, bool invert_TrainTestNegative): Crea le matrici kernel di train e test scrivendole su file, preparando l'esecuzione dei programmi GISTper il train, test e score. Utilizza come sorgente dati il database sotto forma di file di testo (SCOP 1.37 e 1.53)

RhdTestInfos* **execAllGistClassification**(): Sui file di supporto a GIST e le kernel matrix calcolate esegue in ordine:

- gist-train-svm
- gist-classify
- gist-score-svm

e restiuisce un oggetto RhdTestInfos contenente i risultati dell'output di gist-score-svm.

6.4.3 Classe SerialTestHandler

Tale classe legge le impostazioni per i test di rilevazione di omologia remota da effettuare da un file di configurazione formattato in modo specifico e il cui formato verrà in seguito descritto. Successivamente esegue in modo seriale test nell'ordine passato in ingresso nell'elenco delle famiglie contenute nel file.

SerialTestsHandler(string testsFile, string workingDir): In ingresso si forniscono i parametri:

- testsFile: percorso assoluto del file contenente la configurazione per eseguire i test
- workingDir: percorso assoluto della cartella base in cui creare la cartella per i test da eseguire

bool **setWorkingDirectory**(string directory): imposta il percorso della cartella di lavoro

void **setGistDirectory**(string gistBin): imposta il percorso degli eseguibili di GIST SVM

void **executeTests**(): Esegue i test in base alla configurazione fornita. Per ogni famiglia di test crea una cartella all'interno di workingDir nominata con l'identificativo della famiglia e al suo interno fa eseguire alla classe RhdTestInfos il test per tale famiglia. Inoltre copia in ogni directory di test il file testsFile nominandolo "test.info". All'interno di workingDir inoltre crea il file "tests.results" contenente alcuni parametri di test e i risultati principali per ogni test di famiglia effettuato.

6.5 Schema di funzionamento dei tests

6.5.1 File di configurazione

Il file di configurazione viene passato in ingresso tramite parametro e contiene le proprietà necessarie a SerialTestHandler per eseguire i test. Le proprietà sono nel formato:

proprietà:valore

e devono essere specificate esattamente nell'ordine in cui vengono presentate nel seguito. Esse sono:

- families:id0, id1, .., idn → contiene gli id delle famiglie su cui effettuare i test
- releaseId:1-13 → se si utilizza il database MySql di SCOP, specifica quale release usare per il dataset tra quelle disponibili nella tabella astral.scop_release
- byPct:0,1 → se si utilizza il database MySql di SCOP, se 1 imposta come sotto-insiemi significativi da cui ottenere le sequenze gli insiemi di tipo %ID. Se 0 imposta i sotto-insiemi di tipo E-value.
- pct:valore → se byPct è impostato a 1, viene usato come valore %ID per il sotto-insieme significativo.
- e_Value:valore → se byPct è impostato a 0, viene usato come valore E-value per il sotto-insieme significativo.
- normalizeKernel:0,1 → se impostato a 1, il software effettua la normalizzazione del kernel.

```
families:3320113,7030502
releaseId:13
byPct:1
pct:90
e_value:-25
normalizeKernel:0
gestNormalizeKernel:0
selectionStrategy:0
kernelType:0
merSize:12
kDiff:3
negativeSetSize:1000
useDbSource:0
mySql_addr:192.168.150.20
mySql_user:root
mySql_pwd:password
gistBin:/opt/libs/gist/bin
scopFileDir:/opt/scop_1.53_db/db
invertNegativeSet:0
```

Figura 6-2: Esempio di file di configurazione

- gestNormalizeKernel:0,1 → se impostato a 1 il software lascia a Gist la normalizzazione delle kernel matrix.
- selectionStrategy:0,1 → Imposta la strategia di selezione degli esempi di test negativi dagli esempi negativi ottenuti mediante interrogazione da database SCOP 1.75B. Le strategie disponibili sono:
  - *0 →SECTIONS_FRACTION:* l'insieme degli esempi negativi viene diviso in esempi di train negativi TRN e esempi di test negativi TSN nelle stesse proporzioni tra gli esempi di test positivi TSP e gli esempi di train positivi TRP. Cioè se TRP/TSP = k , e l'insieme degli esempi negativi è uguale a M = TRN + TSN, TRN e TSN vengono scelti in modo tale che TRN/TSN = k.
  - 1→ TRAIN_800: indipendentemente da M > 800, TRN viene impostato a 800.
- kernel_type:0,1 → Imposta il tipo di kernel con cui effettuare i test. I tipi disponibili sono:
  - 0 → Kernel combinatorio con ordine
  - 1 → Mismatch Kernel
- merSize:valore → è la dimensione delle tuple per i mismatch kernel utilizzati.
- kDiff:value → è il numero massimo di mismatch ammessi per i mismatch kernel utilizzati.
- negativeSetSize:valore → se si utilizza il database MySql di SCOP, è il numero di esempi negativi totale ottenuto dal database.
- useDbSource:0,1 → Se 1 si utilizza come sorgente il database MySql di SCOP, se 0 si utilizza il database su file di testo.
- mySql_addr:indirizzo_ip → si specifica l'indirizzo IP del server MySql contenente il database SCOP.
- mySql_user:user → specifica il nome utente con cui effettuare la connessione al database MySql.
- mySql_pwd:password → specifica la password dell'utente con cui si effettua la connessione al database MySql.
- gistBin:percorso_assoluto → percorso della cartella contenente gli eseguibili di GIST SVM.
- scopFileDir_percorso_assoluto → percorso del database SCOP su file di testo.
- invertNegativeSet:0,1 → per il sotto-insieme SCOP 1.37 se impostato a 0 utilizza fold0, altrimenti fold1.

In Figura 6-2 vi è un esempio di file di configurazione.

6.5.2 Esecuzione del programma

Il software è fornito con codice sorgente ed è possibile compilarlo mediante il tool di compilazione automatica *make* [33].

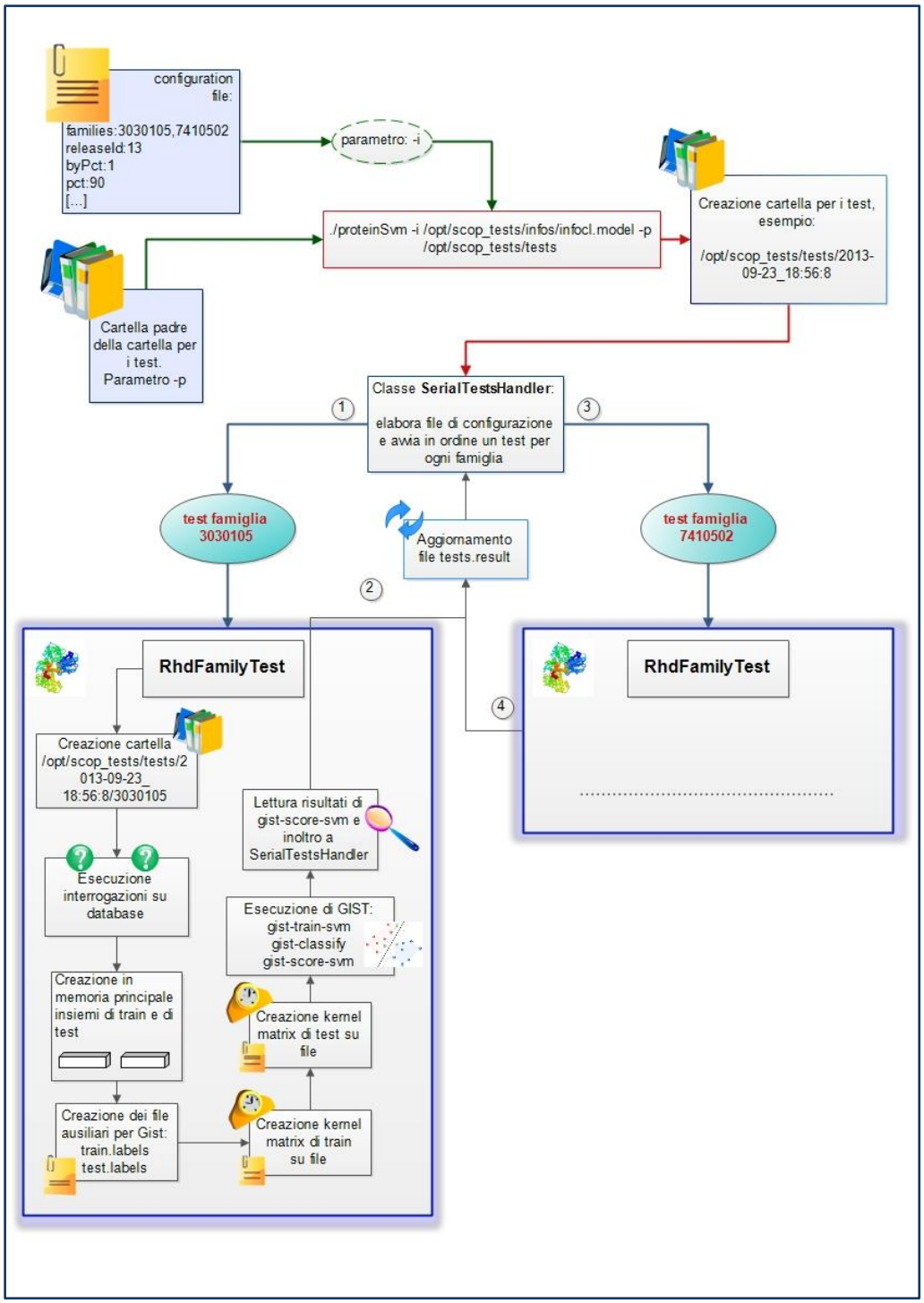


Figura 6-3: Schema esecuzione test

All'interno della directory principale *Release* eseguire nell'ordine:

- make clean
- make all

per compilare il programma. Verrà creato l'eseguibile *proteinSvm*. I parametri per la sua esecuzione sono:

- -i percorso_assoluto: specifica la posizione del file di configurazione
- -p percorso_assoluto: specifica la directory di base per i test

Esempio di esecuzione:

```
./proteinSvm -i /opt/SVM/scop_tests/infos/infocl.model -p /opt/SVM/scop_tests/tests
```

L'esecuzione creerà all'interno della cartella specificata con il parametro –p la cartella di lavoro per i test, nominata nel seguente modo:

aaaa-MM-gg_hh:mm:ss

dove *aaaa* è l'anno, *MM* il mese, *gg* il giorno, *hh* l'ora, *mm* i minuti e *ss* i secondi.

All'interno della cartella di lavoro viene creata una cartella per ogni test di una famiglia, nominata con l'ID di tale famiglia e inoltre un file contenente i risultati dei test. In Figura 6-3 vi è uno schema che descrive il funzionamento del software per l'esecuzione dei test.

7 Risultati dei test

In questo capitolo si descrivono i test effettuati per valutare le prestazioni del kernel combinatorio con ordine.

L'insieme di dati utilizzato per i test è il sottoinsieme del database SCOP 1.53 utilizzato anche in [2]. Esso contiene i data set per effettuare le prove su 54 famiglie significative.

Il tipo di test effettuato per ogni famiglia è di rilevamento dell'omologia remota, descritto in 4.4 .

Per ogni coppia di parametri, m = lunghezza della tupla, k = numero di mismatch scelti, si effettuano i 54 test. Per ogni test si calcolano i valori di area sotto la curva ROC e i valori True Positive, False Positive, True Negative e False Positive. In seguito si calcola per i 54 test effettuati il valore di area ROC medio e si traccia il grafico della curva di frequenza cumulativa ROC. Tale grafico ha sulle ascisse i valori ROC (da 0 a 1) e sulle ordinate il numero di famiglie (da 1 a 52). Un punto (x,y) sulla curva indica il numero di famiglie (y) che hanno valore di ROC sopra una certa soglia (x).

Tale curva è tanto migliore quanto più tende all'angolo in alto a destra del grafico, in quanto ciò sta a significare che un elevato numero di test ha dato risultati di area ROC tendenti a uno. Idealmente se tutti i test dessero risultato di area ROC pari a 1, curva sarebbe una linea verticale che intercetta l'ascisse sul valore 1.

7.1 GIST SVM

Il software di classificazione utilizzato per i test è GIST SVM, sviluppato appositamente per la valutazione dei test di rilevamento delle omologie remote utilizzando come classificatore le macchine a supporto vettoriale. Gist è reperibile all'indirizzo [29].

Nel dettaglio GIST contiene strumenti per la classificazione mediante SVM e l'analisi dei componenti principali dei kernel.

Gli strumenti applicativi utilizzati nella trattazione sono:

- **gist-train-svm**: allena l'SVM basandosi sugli esempi di train dati in ingresso. Vi sono diverse modalità di utilizzo dell'applicativo. Nella più comune si fornisce su un file l'elenco delle coppie ‹identificativo esempi, classe di appartenenza› e su un altro i vettori che descrivono gli esempi, selezionando poi mediante parametri il tipo di kernel da utilizzare. Nella modalità utilizzata per i test effettuati, si fornisce in ingresso al classificatore un file contenente l'elenco delle coppie ‹identificativo esempio, classe di appartenenza› e su un altro file la kernel matrix. La matrice ha n+1 righe e n+1 colonne, dove n è il numero di esempi di training. I valori sono separati dal carattere di tabulazione '\t'. La prima riga e la prima colonna contengono gli identificativi degli esempi di training, con l'elemento di indice <prima riga, prima colonna> contenente un valore a scelta come segnaposto, non utilizzato per i calcoli.
- **gist-classify:** applica a una SVM allenata tramite gist-train-svm un dataset di test, e produce la classificazione sotto forma di file contenente <id esempio, classe predetta>. Nella modalità utilizzata per i test effettuati, si fornisce in ingresso al classificatore un file contenente la kernel matrix di test. Tale matrice ha n+1 righe e m+1 colonne, dove n è il numero di esempi di test e m è il numero di esempi di train. La prima riga e la prima colonna contengono gli identificativi degli esempi. Alle coordinate (i,j) della matrice vi è il valore del kernel K(i,j).
- **gist-score-svm**: valuta le statistiche di prestazioni dall'output di gist-train-svm e gist-classify. In ingresso si fornisce l'output di gist-classify.

7.1.1 Parametri utilizzati per i test

Nel seguito si descrivono i parametri utilizzati per l'esecuzione dei tre strumenti descritti.

gist-train-svm

La riga di commando utilizzata per avviare il programma è:

```
./gist-train-svm -nonormalize -matrix -constant 0 -train ./train.mtx -class ./train.labels >
./train.weights
```

Descrizione dei parametri:

- -matrix: specifica che viene utilizzata la kernel matrix
- -constant 0: specifica che ai valori del kernel non viene aggiunta alcuna costante (costante impostata a 0).
- -train ./train.mtx: specifica il file contenente la kernel matrix di training, con i valori di similarita' tra sequenze della superfamiglia della famiglia analizzata.
- -./train.labels: è il file contenente l'elenco di coppie <identificativo esempio, classe esempio>.
- > ./train.weights: è il file di output contenente i pesi calcolati per l'SVM

Il programma viene avviato in modalità <matrix> passando in ingresso la kernel matrix di train calcolata dal software proteinSvm.

Vengono utilizzati tutti i parametri di default per il training dell'SVM, eccetto per la costante additiva che è impostata a 0 per i valori del kernel. Questo perché i valori di kernel vengono calcolati da un software esterno.

gist-classify

La riga di commando utilizzata per avviare il programma è:

```
./gist-classify -selftrain ./selfTrain.vec -selftest ./selfTest.vec -learned ./train.weights -test ./test.mtx >
./test.predict
```

Descrizione dei parametri:

- -selftrain ./selfTrain.vec: contiene il vettore degli auto valori del kernel per gli esempi di train (K(x,x)).
- -selftest ./selfTest.vec contiene il vettore degli auto valori del kernel per gli esempi di test (K(x,x)).
- -learned ./train.weights: è l'output del software gist-train-svm
- -test ./test.mtx: è il file contenente la kernel matrix di test, con i valori di similarità tra gli esempi di test e training.
- ./test.predict: è l'output del classificatore. Contiene le classi predette per gli esempi di test e i valori discriminativi per la classificazione.

Da notare che il programma viene avviato in modalità <matrix> passando in ingresso la kernel matrix di train calcolata dal software proteinSvm.

gist-score-svm

La riga di commando utilizzata per avviare il programma è:

```
./gist-score-svm -test  ./test.labels ./test.predict ./train.weights > ./score-svm
```

Descrizione dei parametri:

- ./test.labels: è il file contenente l'elenco di coppie <identificativo esempio di test, classe esempio reale>.
- ./test.predict: è l'output del programma gist-classify
- ./train.weights: è l'output di gist-train-svm. Contiene I parametri che definiscono il modello utilizzato per l'SVM.

- ./score-svm: è l'output dello script. Contiene le informazioni riguardanti la qualità del test.

7.2 Hardware utilizzato per i test

Per effettuare i test sono stati utilizzati tre diversi sistemi, due High Performance Computer e un sistema desktop multicore.

P770 AACSE System:

- 3 drawer (unità blade): ognuno con due processori P7 da 16 core, in totale 48 core P7
- Prestazioni singolo core: 32 Gigaflops
- 640GB di RAM
- 16 TB di spazio disco (SAN storage)
- Sistema operativo: Suse Linux Enterprise 11 SP1

Intel i5 3350P:

- 20GB di ram
- Prestazioni singolo core: 118 Gigaflops
- Sistema operativo: Ubuntu 12.10

Cluster blade:

- Sistema operativo: Sun grid engine

Il sistema Power7 utilizza LoadLeveler di IBM [34] per l'esecuzione dei job. In figura Figura 7-1 è riportato lo script di esecuzione utilizzato per avviare proteinSvm tramite esso:

```
#!/bin/sh
# .. put nothing before this header
#@ job_name = proteinSVM.$(jobid)
#@ job_type = serial
#@ environment = COPY_ALL
#@ output = /home/barilaro/SVM/scop_tests/outputs/$(jobid).out
#@ error =  /home/barilaro/SVM/scop_tests/outputs/$(jobid).err
#@ notify_user = youremail@domain.tdl
#@ notification = error
#@ class = long
#@ initialdir = /home/barilaro/SVM
#@ executable = /home/barilaro/SVM/app/Release/proteinSvm
#@ arguments = -i /home/barilaro/SVM/scop_tests/infos/infocl.model -p
/home/barilaro/SVM/scop_tests/tests
#@ queue
```

Figura 7-1: Script LoadLeveler

Il comando per eseguire tale script è:

utente@power7l: llsubmit nomescript

Per informazioni riguardo LoadLeveler consultare le relative guide fornite da IBM.

Per il sistema Sun Grid Engine invece lo script viene caricato per la schedulazione tramite il comando qsub. Esempio:

qsub -cwd -v LD_LIBRARY_PATH=/opt /libs/boost/lib -b y ./proteinSvm –i /opt/scop_tests/infos/infocl.model -p /opt/SVM/scop_tests/tests

Il comando è seguito dal parametro –v che specifica le variabili d'ambiente, nel nostro caso LD_LIBRARY_PATH che specifica dove è situata la libreria Boost utilizzata, il parametro –b che specifica che l'applicativo passato è un programma binario, -cwd che specifica che gli output vengono messi nella cartella corrente e a seguire è specificato il comando da schedulare. Nel nostro caso proteinSvm.

7.2.1 Performance di un singolo core per i diversi sistemi

Si effettua il test del kernel combinatorio per due diverse famiglie sui tre diversi sistemi utilizzati, al fine di confrontarne i tempi di esecuzione. Uno è il test per la famiglia che richiede il minor tempo di esecuzione, l'altro è quello per la famiglia che richiede il maggior tempo. La combinazione di parametri utilizzata è (9,2).

TEMPI CALCOLO singolo core nei diversi SISTEMI			
famiglia	i5 3550P [s]	Power7 [s]	Blade [s]
2440102	1099	3504	2280
3320111	12496	41846	28120

Tabella 7-1: Confronto prestazioni dei tre sistemi

In Tabella 7-1 sono riportati i tempi di calcolo per gli stessi test sulle tre piattaforme. Si può vedere che il tempo di calcolo del Power7 è quasi il quadruplo del tempo di calcolo del i5 3550P. Il sistem Blade invece impiega circa il doppio del tempo rispetto al sistema i5 3350P.

7.2.2 Performance del mismatch kernel al crescere del parametro k

Per il Mismatch Kernel sono stati effettuati alcuni test per i parametri (m=5,k=1) e (m=5,k=2) al fine di valutare sperimentalmente le differenze nei tempi di calcolo al crescere di k.

In tabella Tabella 7-2 sono riportati i risultati di tali test.

TEMPI CALCOLO MISMATCH KERNEL			
family Id:	**A=(5,1)[s]**	**B=(5,2)[s]**	**B/A**
1450102	198	21855	110.3788
1040101	168	14361	85.48214
1040102	220	21752	98.87273
1270101	189	17966	95.0582
1270102	176	15628	88.79545
1040103	211	21587	102.3081
1410105	155	11767	75.91613
1410102	222	22936	103.3153
1360102	214	21707	101.4346
1360105	134	8083	60.3209
2090102	175	15468	88.38857
2560102	178	15903	89.3427
2090103	217	22704	104.6267
2440102	111	5251	47.30631
2090104	191	18215	95.36649
2280101	140	9571	68.36429
		Media B/A:	88.45484

Tabella 7-2: tempi calcolo MK

Si può vedere che per k=2 il tempo di calcolo è in media 88 volte superiore a quello per k=1.

7.2.3 Performance del approximate mismatch kernel al crescere del parametro k

Per l' Approximate Mismatch Kernel sono stati effettuati alcuni test per i parametri (m=7,k=1), (m=7,k=2) , (11,1), (11,2) e (11,3) al fine di valutare sperimentalmente le differenze nei tempi di calcolo al crescere di k.

TEMPI CALCOLO AMK				
family id:	(7,2)[s]	(11,3)[s]	(11,2)[s]	(11,1)[s]
2090102	7853	7971	7996	7866
2560102	8174	8308	8270	8187
2090103	11697	11872	11819	11934
2440102	1113	1124	1117	1131
2090104	9536	9664	9599	9730

Tabella 7-3: Tempi calcolo AMK

In Tabella 7-3 si può vedere che il tempo di calcolo è praticamente costante al variare dei parametri. Ciò verifica sperimentalmente che l'algoritmo utilizzato non dipende dai parametri (m,k).

7.3 Risultati dei test

Gli esperimenti svolti avevano l'obiettivo di studiare le prestazioni della classificazione di string kernel con mismatch al variare dei parametri di lunghezza delle tuple e di numero di mismatch ammessi.

Per quanto riguarda il mismatch kernel, secondo quanto riportato in [2], i valori migliori di classificazione si hanno per (m=5,k=1). Questo kernel non e' pero' particolarmente scalabile. I test sono stati effettuati quindi solo utilizzando kernel approssimati. In particolare si riportano i risultati relativi al kernel con combinazioni ordinate, che tra quelli testati ha presentato le migliori prestazioni. Per completezza si sono riportati anche i risultati del mismatch kernel (m=5,k=1) calcolati con gli stessi parametri con cui sono stati calcolati i risultati dei kernel approssimati. Tali parametri sono quelli di default di Gist, che non garantiscono prestazioni ottimali (infatti, rispetto ai risultati pubblicati in [2], in questi esperimenti il mismatch kernel ha riportato prestazioni inferiori). Tuttavia, essendo l'interesse principale lo studio di classificatori basati su stringhe con mismatch al variare di m e k non ci si e' focalizzati sul setting ottimale dei parametri ma sull'andamento della bonta' della classificazione.

Il kernel combinatorio con ordine è stato analizzato per i valori di mismatch k=1, 2, 3 variando per ciascuno di essi il parametro m al fine di capirne l'andamento della bonta' della classificazione. Nel seguito gli insiemi di test sono descritti per valore di k. Infine nell'ultima sezione del capitolo vengono confrontati i migliori risultati ottenuti per i tre valori del parametro k.

7.3.1 Risultati per k=1

Per k=1 sono stati effettuati i test per i valori di m da 5 a 9. In Figura 7-2 sono riportate le curve di frequenza cumulativa per tali esperimenti.

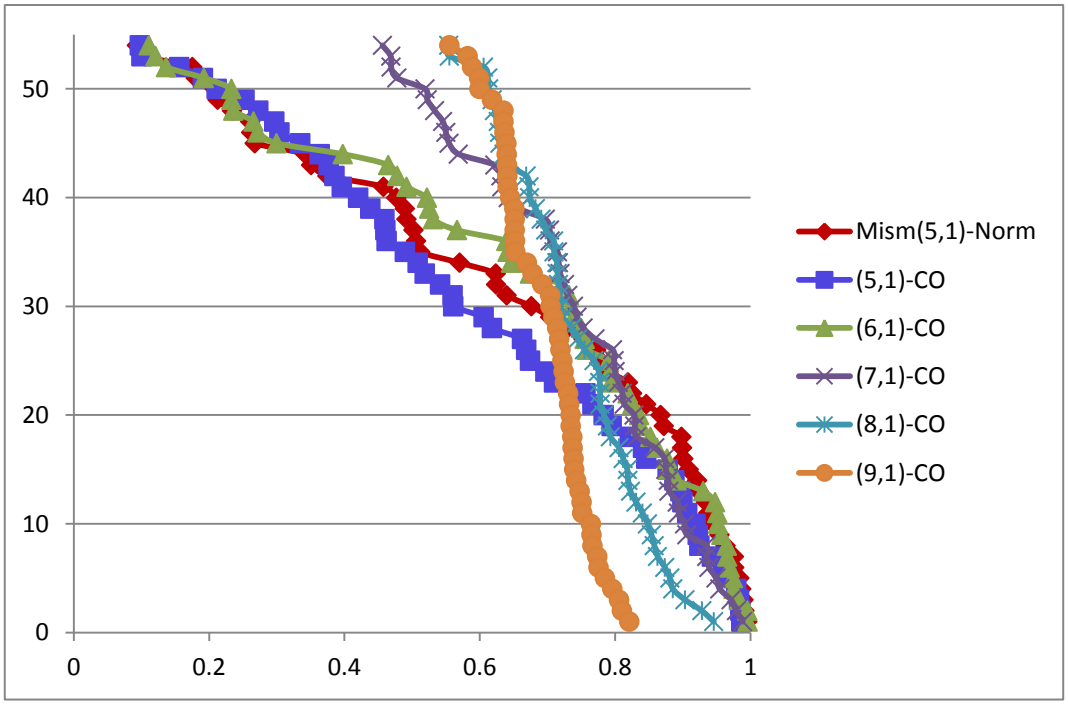


Figura 7-2: Curve di frequenza per k=1

In rosso (rombi) vi è la curva per il Mismatch Kernel (5,1) normalizzato. Si può vedere che la curva blu (quadrati) per la combinazione di parametri (5,1) del kernel combinatorio è sotto alla curva rossa per la maggior parte dei valori ROC. Ciò significa che i test hanno dato risultati inferiori al mismatch kernel. La curva verde (triangoli) per la combinazione (6,1) invece è leggermente superiore alla curva rossa per la maggior parte dei valori di ROC. Questo significa che per tale coppia di parametri il kernel ha performance leggermente maggiori. Per valori di m maggiori a 6 si può vedere che le curve iniziano a comportarsi in maniera totalmente diversa dai casi precedenti. Esse infatti aumentano l'angolo di inclinazione con l'asse delle ascisse. La curva viola (simboli x) per i parametri (7,1) ha un valore di ROC minimo decisamente superiore a quello dei test precedenti pari a 0.4561. La maggior pendenza rispetto ai casi precedenti è da interpretare come ad un più rapido decrescere del numero di test che sono sopra ad una determinata soglia ROC. Diminuisce lo score delle famiglie con score più alto, ma aumenta lo score delle famiglie che lo avevano basso. Si può vedere che per la curva azzurra (simboli x barrato) per (8,1) il valore minimo di ROC aumenta ulteriormente e risulta pari a 0.5541 e la pendenza aumenta leggermente. Infine per la curva arancione (pallini) per (9,1) il valore di ROC minimo aumenta solo leggermente rispetto al caso precedente e risulta 0.5551 ma la pendenza aumenta in modo considerevole. Tale curva quindi risulta significativamente inferiore alla curva (8,1). In Tabella 7-4 sono riportati i valori di ROC medi calcolati per ogni gruppo di test sulle 54 famiglie divisi per tipo di parametri.

Valori ROC medi	
Paremetri di test	***Area ROC***
mism(5,1)-norm:	0.6572
(5,1)-CO:	0.6193
(6,1)-CO:	0.6758
(7,1)-CO:	**0.7499**
(8,1)-CO:	0.745
(9,1)-CO:	0.6996

Tabella 7-4: Valori ROC medi per k=1

Il migliore valore medio si ottiene per la combinazione di parametri (7,1), il secondo miglior valore si ha per (8,1) e differisce dal primo per circa un 0.4%.

Per poter capire meglio la differenza tra i due classificatori, è necessario analizzare i valori TP,FP,TN,FP dei test effettuati. Sono stati calcolati i valori di True Negative Rate medi e i valori di True Positive Rate [rif. 1.2.4], riportati in Tabella 7-5.

Parametri test	TPR medio	TNR medio	(TPR+TNR)/2	S.Q.M. TPR	S.Q.M. TNR
mism(5,1)-norm	0.00282213	1	0.50141	0.01605	0
(7,1)-CO	0.095646	0.975164	0.5354	**0.1951**	**0.0488**
(8,1)-CO	0.536541	0.731588	**0.6341**	0.3195	0.1077

Tabella 7-5: Statistiche per k=1

Si può notare che per (7,1) il valore di TPR rate è molto basso, mentre il TNR rate è molto alto. Ciò significa che il classificatore rileva particolarmente bene gli esempi di test negativi, ma fallisce nella maggior parte dei casi a rilevare quelli positivi.

Per (8,1) invece si ha un netto incremento del valore di TPR medio mentre il TNR descresce rispetto al caso precedente. Ciò significa che il classificatore migliora nettamente nel rilevare gli esempi positivi, a discapito di un decremento degli esempi negativi rilevati.

La media tra i rate positivi e negativi è maggiore per (8,1) e si sceglie tale coppia di valori come la migliore per k=1. Da notare tuttavia che lo scarto quadratico medio è superiore per tale combinazione rispetto a (7,1).

7.3.2 Risultati per k=2

Per k=1 sono stati effettuati i test per i valori di m da 6 a 12. In Figura 7-3Figura 7-2 sono riportate le curve di frequenza cumulativa per tali esperimenti.

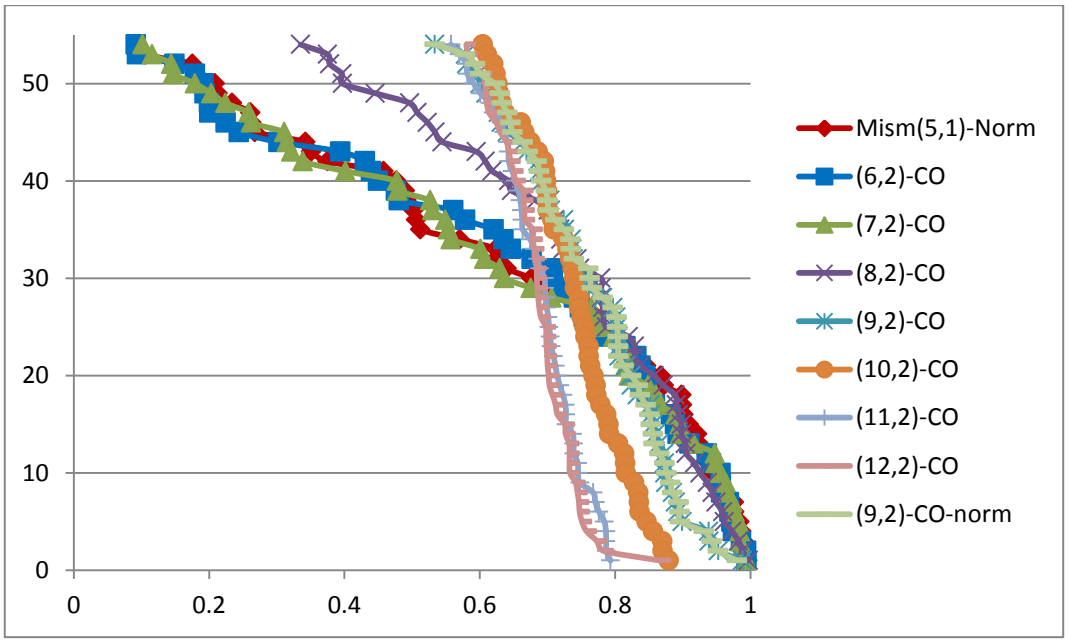


Figura 7-3: Curve di frequenza per k=2

Possiamo vedere che per le coppie (6,2) e (7,2) le curve si comportano in modo simile alla curva del Mismatch Kernel (5,1) normalizzato. Per valori di m>7 il comportamento delle curve cambia in modo significativo. Al crescere di m la pendenza aumenta

progressivamente insieme al valore di ROC minimo. Dal grafico si può osservare che le migliori curve si hanno per le coppie (8,2), (9,2) e (10,2). In Tabella 7-6 sono riportati i valori di ROC medi.

Valori di ROC medi	
Parametri di test	**Area ROC**
misn(5,1)-norm:	0.6572
(6,2)-CO:	0.6561
(7,2)-CO:	0.6491
(8,2)-CO:	0.747
(9,2)-CO:	**0.7687**
(10,2)-CO:	0.7423
(11,2)-CO:	0.6909
(12,2)-CO:	0.6872

Tabella 7-6: Valori ROC medi per k=2

Come ci si può aspettare i migliori valori di ROC medi si hanno per le tre coppie sopra citate: (8,2), (9,2) e (10,2). Il valore più alto si ha per la coppia (9,2). Per gli altri due si ha una differenza percentuale poco significativa di circa il 2%. Si sono calcolati anche in questo caso i valori di TPR e TNR medi, riportati in Tabella 7-7.

Parametri test	TPR medio	TNR medio	(TPR+TNR)/2	S.Q.M. TPR	S.Q.M. TNR
(8,2)-CO	0.050444	0.993603	0.52202	0.1406	0.01673
(9,2)-CO	0.268351	0.907791	0.58807	0.31934	0.09391
(10,2)-CO	0.797492	0.571041	**0.68427**	0.1970	0.0874
(11,2)-CO	**1**	0.359975	0.67998	**0**	**0.0450**

Tabella 7-7: Statistiche per k=2

La migliore media tra TRP e TNR medi si ha per la combinazione (10,2). Inoltre si può vedere che per (10,2) la varianza per i TPR e TNR non è eccessivamente elevata rispetto a (9,2) e non è molto differente dalle varianze per (8,2). Tale combinazione è scelta come combinazione migliore per k=2.

E' da notare che la combinazione (11,2) rileva sempre in modo corretto tutti gli esempi veri positivi. Essa è da tenere in considerazione se è necessario rilevare tutti gli esempi positivi, anche al costo di rilevare anche un elevato numero di falsi positivi.

■ Normalizzazione di (9,2)

Si è effettuata la normalizzazione del kernel con parametri (9,2), cioè la combinazione con il più elevato valore di ROC per comprendere se con essa si ha una qualche variazione nelle prestazioni del classificatore.

Il valore ROC medio rilevato è:

0.770267

Esso non è significativamente superiore al valore per (9,2) non normalizzato. Le statistiche sono:

- TPR = 0.2710
- TNR = 0.9079
- S.Q.M. TPR = 0.3188
- S.Q.M. TNR = 0.0938

Leggermente superiori ai valori rilevati per (9,2). In Figura 7-3 si può osservare la curva delle frequenze cumulative per (9,2) normalizzato.

7.3.3 Risultati per k=3

Per k=1 sono stati effettuati i test per i valori di m da 11 a 13. In Figura 7-4 Figura 7-3Figura 7-2 sono riportate le curve di frequenza cumulativa per tali esperimenti.

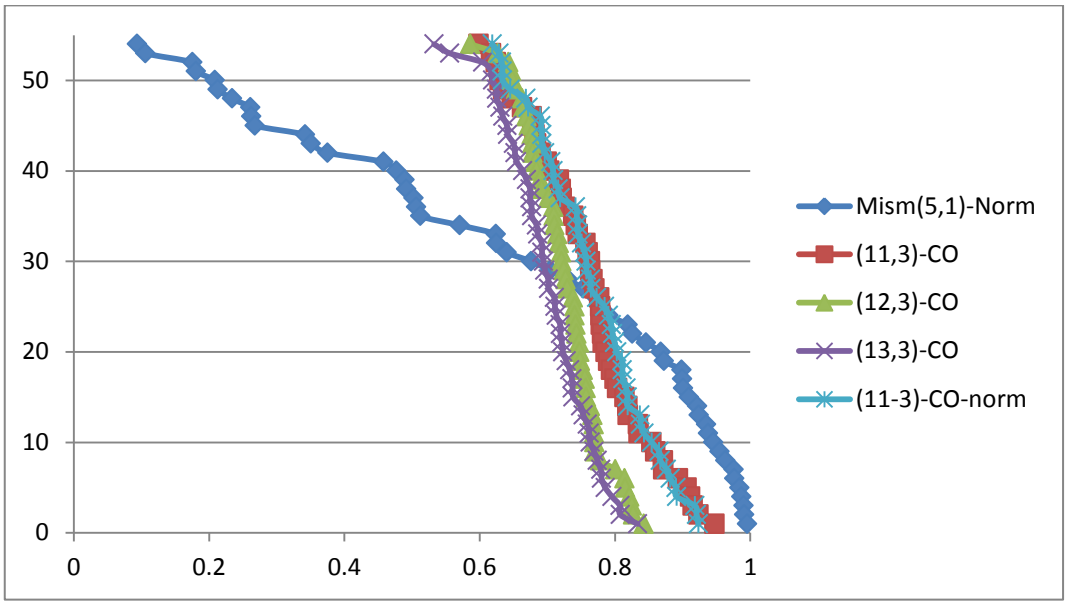


Figura 7-4: Curve di frequenza per k=3

In Tabella 7-8 sono riportati i valori di area ROC media per tali test.

Valori di ROC medi	
Parametri di test	**Area ROC**
mism(5,1)-norm:	0.6572
(11,3)-CO:	**0.7642**
(12,3)-CO:	0.7267
(13,3)-CO:	0.7014

Tabella 7-8: Valori di ROC medi per k=3

Il migliore valore medio si ha per i parametri (11,3). Si può tuttavia notare che non differisce dal valore (12,3) solo di circa il 4%.

Si sono calcolati anche in questo caso i valori di TPR e TNR medi, riportati in Tabella 7-9.

Parametri test	TPR medio	TNR medio	(TPR+TNR)/2	S.Q.M. TPR	S.Q.M. TNR
(11,3)-CO	0.584364	0.737961	0.6612	0.2912	0.1076
(12,3)-CO	0.944822	0.469212	**0.7070**	**0.0938**	**0.0655**

Tabella 7-9: Statistiche per k=3

Si può osservare che per (12,3) il il classificatore riesce a rilevare quasi la totalità degli esempi di test positivi, tuttavia a discapito degli esempi negativi. In media ne vengono rilevati solo il 47%. Il miglior valore medio si ottiene per (12,3), abbinato ai più bassi valori di varianza tra le due combinazioni. Nonostante ciò è da notare che il miglior trade-off tra la media dei TPR e la media dei FPR si ottiene per la combinazione (11,3). Infatti i TP e FP medi rilevati è superiore al 58%, vedere Tabella 7-9.

■ **Normalizzazione di (11,3)**

Si è effettuata la normalizzazione del kernel con parametri (11,3), cioè la combinazione con il più elevato valore di ROC per k=3 per comprendere se con essa si ha una qualche variazione nelle prestazioni del classificatore.

Il valore ROC medio rilevato è:

0.768784

Esso non è significativamente superiore al valore per (11,3) non normalizzato. Le statistiche sono:

- TPR = 0.582477
- TNR = 0.738204
- S.Q.M. TPR = 0.2914
- S.Q.M. TNR = 0.1077

Leggermente superiori ai valori rilevati per (11,3). In si Figura 7-4 può osservare la curva delle frequenze cumulative per (11,3) normalizzato.

7.3.4 Confronto tra i migliori risultati per k=1,2,3

I valori migliori di area ROC si ottengono per le combinazioni (7,1), (10,2), (11,3) e (12,3). Ognuna di esse, come analizzato nel paragrafo precedente, ha differenti proprietà specifiche.

Valori di ROC medi	
Parametri di test	**Area ROC**
mism(5,1)-norm:	0.6572
(7,1)-CO:	0.7499
(10,2)-CO:	0.7423
(11,3)-CO:	0.7642
(12,3)-CO:	0.7267

Tabella 7-10: migliori valori ROC

Nello specifico se al classificatore è necessario rilevare il maggior numero possibile di valori positivi, è da scegliere (12,3). La combinazione (7,1) invece classifica correttamente quasi tutti gli esempi negativi, ma non rileva molti esempi realmente positivi, cosa non utile nel caso di datase sbilanciato.

Le combinazioni (10,2) e (11,3) invece sono a metà via tra le due precedenti e sono da preferire in generale, in quanto permettono un discreto rilevamento degli esempi positivi, senza compromettere eccessivamente la capacità del classificatore di rilevare anche gli esempi negativi. In Tabella 7-11 sono riassunte le statistiche per tali parametri.

Parametri test	TPR medio	TNR medio	(TPR+TNR)/2	S.Q.M. TPR	S.Q.M. TNR
(7,1)-CO	0.095646	**0.975164**	0.5354	0.1951	0.0488
(10,2)-CO	**0.797492**	**0.571041**	**0.68427**	0.1970	0.0874
(11,3)-CO	**0.584364**	**0.737961**	**0.6612**	0.2912	0.1076
(12,3)-CO	**0.944822**	0.469212	**0.7070**	0.0938	0.0655

Tabella 7-11: Statistiche per le migliori combinazioni

In Appendice sono riportati i risultati dei test effettuati per tali parametri.

8 Conclusioni

Si è visto come l'Approximate Mismatch Kernel e la sua generalizzazione WAMK abilitino lo studio in maniera più approfondita degli string kernels. Questo soprattutto grazie ad un algoritmo scalabile che ne esegue il calcolo.

Si è potuto studiare come cambia la classificazione al variare di un gran numero di coppie di parametri (m,k). Si è verificato che per ottenure una buona classificazione aumentando m, , bisogna aumentare il numero di mismatch ammessi. Inoltre dai risultati è emerso che combinazioni di parametri con valori di ROC medio simili possono avere comportamenti tra loro molto diverso in termini di TPR e TNR.

Dai risultati è inoltre emerso che la ricerca sugli string kernel può essere ulteriormente approfondita e tali kernel migliorati. In particolar modo grazie all'algoritmo implementato e alla generalizzazione effettuata si possono condurre in maniera agevole studi aggiuntivi sulla natura e comportamento degli string kernel.

Bibliografia

[1] J. Chandonia, N. S.Walker, L. Conte, P. Kowhl, M. Levitt e S. E.Brenner, «ASTRAL compendium enhancements» *Nucleic Acids Research,* vol. 30, n. 1, pp. 260-263, 2002.

[2] C. S. Leslie, E. Eskin, A. Cohen, J. Weston e W. S. Noble, «Mismatch string kernels for discriminative protein classification» *Bioinformatics,* vol. 20, n. 4, pp. 467-476, 2004.

[3] Intel Corporation, «Processori» 8 Maggio 2012. [Online]. Available: http://www.intel.com/support/it/processors/sb/cs-017346.htm. [Consultato in Ottobre 2013].

[4] C. Pizzi, «K-difference matching in amortized linear time for all the words in a text» *Theoretical Computer Science,* vol. 410, Issues 8-10, pp. 983-987, 2009.

[5] A. Ferro, «Introduzione alla bioinformatica - 2. Traduzione e proteine» 11-03-2010. [Online]. Available: http://ferrolab.dmi.unict.it/introduzione_bioinformatica.html [Consultato in Maggio 2013].

[6] D. W.Mount, in *Bioinformatics: Sequence and Genome Analysis*, Cold Spring Harbor Laboratory press, pp. 382-398.

[7] F. Sambo, «Intelligenza Artificiale» 2009/2010. [Online]. Available: http://www.dei.unipd.it/~sambofra/Apprendimento_Automatico_e_Reti_Neurali-0910.pdf [Consultato in luglio 2013].

[8] V. Kumar, P.-N. Tan e M. Steinbach, «Introduction to Data Mining» Pearson International Edition, 2006, pp. 256-276.

[9] D. W.Mount, in *Bioinformatics: Sequence and Genome Analysis*, Cold Spring Harbor Laboratory Press, pp. 2-14.

[10] S. Needleman e C. Wunsch, «A general method applicable to the search for similarities in the amino acid sequence of two proteins» *Journal of Molecular Biology,* vol. 48, n. 3, pp. 443-453, 1970.

[11] D. Lipman e W. Pearson, «Rapid and sensitive protein similarity searches» *Science,* n. 227, pp. 1435-1441, 1985.

[12] S. Altschul, W. Gish, W. Miller, E. Myers e D. Lipman, «Basic Local Alignment Search Tool» *Journal of Molecular Biology,* n. Oct 5., pp. 403-410, 1990.

[13] S. Vinga e J. Almeida, «Alignment-free sequence comparison - a review» *Bioinformatics,* vol. 19, n. 4, pp. 513-523, 2004.

[14] C. Leslie, E. Eskin e W. S. Noble, «The spectrum kernel: a string kernel for SVM protein classification» *Pacific Symposium on Biocomputing,* n. 7, pp. 566-575, 2002.

[15] C. Leslie, E. Eskin, J. Weston e W. S. Noble, «Mismatch String Kernels for SVM Protein Classification» *Bioinformatics,* n. 20, pp. 467-476, 2004.

[16] R. Kuang, E. Ie, K. Wang, K. Wang, M. Siddiqi, Y. Freund e C. Leslie, «Profile-based string kernels for remote homology detection and motif extraction» *Journal of Computational Biology*, *June* 2005,*3(3), pp. 527-550*.

[17] T. Hubbard, B. Ailey, S. Brenner, A. Murzin e C. Chothia, «SCOP, Structural Classification of Proteins database: applications to evaluation of the effectiveness of sequence alignment methods and statistics of protein structural data»

Crystallography Journals, vol. 54, pp. 1147-1154, 1998.

[18] «BealBio» [Online]. Available: http://bealbio.wikispaces.com/Period+3+ch+5+Q. [Consultato in settembre 2013].

[19] S. E. Brenner, P. Koehl e M. Levitt, «The Astral compendium for protein structure and sequence analysis» *Nucleic Acids Research*, vol. 28, n. 1, pp. 254-256, 2000.

[20] J. Chandonia, N. S.Walker, L. Conte, P. Kowhl, M. Levitt e S. E.Brenner, «The ASTRAL Compendium in 2004» *Nucleic Acid Research*, vol. 32, pp. 189-192, 2004.

[21] A. G. Murzin e J-C. Chandonia, «Structural Classification of Proteins and ASTRAL release 1.75B (January 2013)» Gennaio 2013. [Online]. Available: http://http://scop.berkeley.edu/. [Consultato in Settembre 2013].

[22] «Pfam» Sanger Istitute [Online]. Available: http://pfam.sanger.ac.uk/. [Consultato in Settembre 2013].

[23] S. Jakkola e D. Haussler, «Exploiting generative models in discriminative classifiers» *Conference on Advances in neural information processing systems II*, pp. 487-493, 1999.

[24] T. Jaakkola, M. Diekhans e D. Haussler, «-Eploiting generative models in discriminative classifiers- Supplementary data» 1999. [Online]. Available: http://compbio.soe.ucsc.edu/discriminative/. [Consultato in agosto 2013].

[25] L. Liao e W.-S. Noble, «Supplementary data» 2002, [Online]. Available: http://noble.gs.washington.edu/proj/svm-pairwise/. [Consultato in Settembre 2013].

[26] T. Jakkola, M. Diekhans e D. Haussler, «A discriminative framework for detecting remote protein homologies» *Journal of Computational Biology*, 2000.

[27] B. Asa e B. Douglas, «Remote homology detection: a motif based approach» *Bioinformatics*, vol. 19, n. suppl. 1, p. i26, 2003.

[28] W. Noble e P. Pavlidis, «GIST» Columbia University, 2002. [Online]. Available: http://www.chibi.ubc.ca/gist/. [Consultato in Settembre 2013].

[29] L. Liao e W.S. Noble, «Combining Pairwise Sequence Similarity and Support Vector Machines for Detecting Remote Protein Evolutionary and Structural Relationships» *Journal of computational biology*, vol. 10, n. 6, 2003.

[30] D. Gusfield, «Algorithms on Strings, Trees, and Sequences» Cambridge University Press, USA, 1997, pp. 59,123.

[31] Y. Shlomo, «ANSI C implementation of a Suffix Tree» 2002, 2003. [Online]. Available: http://mila.cs.technion.ac.il/~yona/suffix_tree/. [Consultato in Agosto 2013].

[32] Free Software Foundation, «GNU Make» GNU, [Online]. Available: http://www.gnu.org/software/make/. [Consultato in settembre 2013].

[33] IBM, «Tivoli Workload Scheduler LoadLeveler» [Online]. Available: http://www-01.ibm.com/software/tivoli/products/scheduler-loadleveler/.

[34] G. Vriend, «Molecules of life - Amino Acids» Centre for Molecular and Biomolecular Informatics, [Online]. Available: http://swift.cmbi.ru.nl/gv/students/mtom/AMAC_1.html. [Consultato in agosto 2013].

[35] «SRS@EMBL-EBI» European Molecular Biology Laboratory, [Online]. Available: http://srs.ebi.ac.uk. [Consultato in Settembre 2013].

[36] «PubMed» National Center for Biotechnology Information, [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed. [Consultato in Settembre 2013].

[37] P.-N. Tan, M. Steinbach e V. Kumar, «Introduction to Data Mining» Pearson International Edition, 2006, pp. 294-301.

[38] M. Sewell, «Structural Risk Minimization» [Online]. Available: http://www.svms.org/srm/. [Consultato in Agosto 2013].

Indice delle figure

RINGRAZIAMENTI

I miei più sentiti ringraziamenti alla Dott.ssa **Cinzia Pizzi** per avermi indirizzato e seguito nello sviluppo di questa tesi, per i preziosi momenti creativi e di confronto, per il vivo interesse nell'argomento trattato e per la sua grande disponibilità e gentilezza. Veramente un immenso Grazie!

Grazie ad **Aurora** che in tutti questi anni di Università mi hai sopportato e supportato. Grazie per avermi con (quasi) pazienza ascoltato in decine di ripassi pre-esame e per alcune figure di questa tesi. Nonché grazie per tutto l'aiuto in questi anni.

Grazie al dott. **Fabio Buttari** per la grande fiducia ed attenzione riposta in me, per aver sempre stimolato il mio interesse informatico e per avermi fatto crescere professionalmente. In questi anni di collaborazione ho imparato molte cose che non dimenticherò e che mi saranno sempre utili in futuro.

Grazie a **Michele B.**, mio caro amico che in tutti questi anni mi è sempre stato vicino e con cui ho passato e passerò bei momenti. Quando c'è bisogno di un tecnico, sei "the man for the job"! Grazie per i mille aiuti con giardino, impianti elettrici che vanno misteriosamente a fuoco, caminetto, cellulari ecc.!

Grazie a **Sonja** per la tua *speciale* amicizia e per i moltissimi momenti e belle esperienze condivise assieme. Volemosse bene! Un "grazie" simbolico anche per tutti i momenti in cui mi hai fatto impazzire!

Grazie a **Sanja** per la tua *Q.* amicizia, per i discorsi impossibilmente creativi che ogni volta facciamo e per i bei momenti trascorsi assieme! Volemosse Q. bene.

Grazie a **Tony** per il sostegno datomi in tutti questi anni e per le mille foto tutte uguali che regolarmente mi mostri.

Grazie ad **Alessandro**, per la preziosa amicizia. Averti avuto come collega all'università è sempre stato un onore. Il progetto di IA, il progetto di CP e le relative notti insonni e il progetto di qualità rimarranno indimenticabili. Sarà ancora più un onore averti come collega anche in futuro. Grazie anche per avere sopportato tutti i miei scherzi durante l'università e aver reso il tempo più allegro.

Grazie a **Federica**, carissima amica sempre vicina e indispensabile.

Grazie a **Chiara**, molto gentile e disponibile, presenza discreta ma costante.

Grazie a **Cristina C.** per i tuoi consigli ed per il tuo affetto.

Grazie a **Corina**, sempre pronta ad ascoltarmi.

Grazie a **Sebastian,** buon geniale amico e allo stesso tempo uomo dalle prodi fatiche.

Grazie a **Lucia** per la tua amicizia. Anche se a volte ci si è persi di vista spero rimarremo più spesso in contatto. Tieni a mente: splende ogni giorno il sole!

Grazie ad **Alberto**, che nonostante la tua presenza sfuggente vuoi constantemente essere toccato, e a **Federico**, che hai incollato il tetto della casetta prima di metterci dentro led e sensori e per la tua eco-compatibilità con i tronchi. E' stato piacevole passare gli ultimi anni universitari con voi, siete e sarete buoni amici.

Grazie a **Leonardo**, per tutte le buonissime carbonare preparate in tutti questi anni e per le fotinie.

Grazie a **Francesca M.T.** per la tua amicizia ed immensa simpatia.

Un grazie a parte ad **Eleonora "Mai Mai Mai"**, anche se con me sei più cattiva dei giudici di X-Factor.

Grazie a **Letizia**, preziosa uditrice di storie e gossip vari.

Grazie a **Xtina M&M**, con cui condivido giornalmente allegre stupidaggini.

Grazie a **Gianluca**, **Gloria** & **Alvise** e **Rossella** buoni amici ed ex- colleghi universitari.

Grazie a **Serena** per la simpatia. Spero che in futuro le tue nebulose origini Scandinave saranno svelate.

Grazie a **Francesco** per la tua burlesca amicizia e ad **Eleonora**, oltre che per l'amicizia, anche per la profonda capacità di gestire le "No!" people.

Grazie ad **"fatty" Eva** per essermi vicino e avermi 'stressato' in tutti questi anni.

Grazie a **Zsófia**, per l'affetto e la grande fiducia che riponi in me.

Grazie ad **Anastasia**. Anche se dopo molti anni che ti conosco faccio ancora, a volte, difficoltà a capirti è un piacere conoscerti. Grazie anche per ascoltarmi nei momenti difficili.

Grazie a **Livia**, per il grande interesse ed affetto nei miei confronti.

APPENDICE

Nella tabella seguente si riportano le dimensioni dei dataset utilizzati per i test sulle 54 famiglie del database SCOP 1.53, divisi per numero di esempi negativi e numero di esempi positivi.

DESCRIZIONE DATASET BASATO SU SCOP 1.53				
	Positive set		**Negative set**	
ID famiglia	train	test	train	test
2.52.01.02	12	5	3060	1275
1.27.01.01	12	6	2890	1444
1.27.01.02	10	8	2408	1926
7.41.05.01	10	9	2241	2016
7.41.05.02	10	9	2241	2016
2.56.01.02	11	8	2509	1824
7.03.05.02	12	9	2330	1746
2.05.01.01	13	11	2345	1983
2.05.01.03	14	10	2525	1803
7.39.01.03	13	14	2083	2242
7.39.01.02	20	7	3204	1121
3.01.08.01	19	8	3002	1263
3.01.08.03	17	10	2686	1579
3.03.01.02	22	7	3280	1043
3.03.01.05	13	16	1938	2385
2.09.01.03	26	5	3625	696
2.09.01.04	21	10	2928	1393
2.09.01.02	17	14	2370	1951
2.38.04.03	24	11	2946	1349
2.38.04.01	30	5	3682	613
2.38.04.05	26	9	3191	1104
1.36.01.05	10	26	1199	3117
1.36.01.02	29	7	3477	839
3.42.01.01	29	10	3208	1105
3.42.01.05	26	13	2876	1437
3.42.01.08	34	5	3761	552
1.45.01.02	33	6	3650	663
1.41.01.05	17	25	1744	2563
1.41.01.02	36	6	3692	615
7.03.06.01	33	9	3203	873
7.03.06.02	16	26	1553	2523
7.03.06.04	37	5	3591	485
1.04.01.01	26	23	2256	1994
1.04.01.02	41	8	3557	693

1.04.01.03	40	9	3470	780
3.32.01.08	40	11	3374	927
3.32.01.11	46	5	3880	421
3.32.01.13	43	8	3627	674
3.32.01.01	42	9	3542	759
3.02.01.02	37	16	3002	1297
3.02.01.04	46	7	3732	567
3.02.01.05	46	7	3732	567
3.02.01.07	48	5	3894	405
3.02.01.03	44	9	3569	730
3.02.01.06	48	5	3894	405
2.28.01.01	18	44	1246	3044
2.28.01.03	56	6	3875	415
7.03.10.01	11	95	423	3653
2.01.01.03	113	8	3895	275
2.01.01.05	94	27	3240	930
2.01.01.04	88	33	3033	1137
2.01.01.01	90	31	3102	1068
2.01.01.02	99	22	3412	758
2.44.01.02	11	140	307	3894

Sono di seguito riportati i risultati dei test del Kernel combinatorio con ordine per le combinazioni di parametri più significative rilevate.

	m= 7 k=1	**m= 10 k=2**	**m= 11 k=3**	**m= 12 k=3**
family Id:	1450102	1450102	1450102	1450102
time	24800	54854	39668	28316
Train ROC	1	1	1	1
FP	11	258	133	345
FN	6	0	0	0
TP	0	6	6	6
TN	652	405	530	318
Test ROC	0.90271	0.85621	0.92358	0.72448
family Id:	1040101	1040101	1040101	1040101
time	12951	11476	23810	25826
Train ROC	1	1	1	1
FP	28	890	567	1083
FN	22	3	6	1
TP	1	20	17	22
TN	1966	1104	1427	911
Test ROC	0.87968	0.72986	0.77888	0.68272
family Id:	1040102	1040102	1040102	1040102
time	28720	18934	43036	36463
Train ROC	1	1	1	1

FP	7	264	144	343
FN	8	1	1	0
TP	0	7	7	8
TN	686	429	549	350
Test ROC	0.9715	0.83424	0.89322	0.74856
family Id:	1270101	1270101	1270101	1270101
time	21698	14982	31094	18755
Train ROC	1	1	1	1
FP	31	592	339	746
FN	6	3	4	0
TP	0	3	2	6
TN	1413	852	1105	698
Test ROC	0.69702	0.68594	0.6901	0.7096
family Id:	1270102	1270102	1270102	1270102
time	18917	46791	26891	26527
Train ROC	1	1	1	1
FP	39	859	535	1049
FN	7	2	3	0
TP	1	6	5	8
TN	1887	1067	1391	877
Test ROC	0.83178	0.69457	0.76298	0.64862
family Id:	1040103	1040103	1040103	1040103
time	26252	18538	39075	37180
Train ROC	1	1	1	1
FP	8	286	132	378
FN	9	1	1	0
TP	0	8	8	9
TN	772	494	648	402
Test ROC	0.87322	0.75499	0.87108	0.75712
family Id:	1410105	1410105	1410105	1410105
time	13796	42369	18845	20619
Train ROC	1	1	1	1
FP	76	1328	923	1534
FN	22	1	3	0
TP	3	24	22	25
TN	2487	1235	1640	1029
Test ROC	0.73032	0.74664	0.81881	0.75265
family Id:	1410102	1410102	1410102	1410102
time	28230	19454	39945	27915
Train ROC	1	1	1	1
FP	7	235	130	308
FN	4	1	1	0
TP	2	5	5	6
TN	608	380	485	307
Test ROC	0.82927	0.8355	0.91491	0.72954
family Id:	1360102	1360102	1360102	1360102
time	27636	57058	50345	37762

Train ROC	1	1	1	1
FP	8	322	184	422
FN	7	2	4	0
TP	0	5	3	7
TN	831	517	655	417
Test ROC	0.82837	0.79023	0.76349	0.74187
family Id:	1360105	1360105	1360105	1360105
time	7869	19840	13400	10530
Train ROC	1	1	1	1
FP	152	1795	1392	2006
FN	22	0	6	0
TP	4	26	20	26
TN	2965	1322	1725	1111
Test ROC	0.72534	0.76096	0.70312	0.69157
family Id:	2090102	2090102	2090102	2090102
time	16580	36237	15817	28228
Train ROC	1	1	1	1
FP	33	892	552	1076
FN	14	8	8	3
TP	0	6	6	11
TN	1918	1059	1399	875
Test ROC	0.62235	0.637	0.67764	0.70909
family Id:	2560102	2560102	2560102	2560102
time	16544	38103	15747	22054
Train ROC	1	1	1	1
FP	27	794	487	965
FN	8	1	6	1
TP	0	7	2	7
TN	1797	1030	1337	859
Test ROC	0.70429	0.6751	0.74226	0.75541
family Id:	2090103	2090103	2090103	2090103
time	37046	62652	11919	38777
Train ROC	1	1	1	1
FP	5	271	149	345
FN	5	2	4	0
TP	0	3	1	5
TN	691	425	547	351
Test ROC	0.5454	0.62328	0.71724	0.74713
family Id:	2440102	2440102	2440102	2440102
time	2959	4850	1155	3465
Train ROC	1	1	1	1
FP	1210	3056	2807	3102
FN	103	0	0	0
TP	37	140	140	140
TN	2684	838	1087	792
Test ROC	0.715	0.60456	0.59835	0.58613
family Id:	2090104	2090104	2090104	2090104

time	22328	47460	19051	33178
Train ROC	1	1	1	1
FP	14	590	323	714
FN	10	2	4	0
TP	0	8	6	10
TN	1379	803	1070	679
Test ROC	0.64228	0.74085	0.77933	0.74731
family Id:	2280101	2280101	2280101	2280101
time	9858	19371	8482	14717
Train ROC	1	1	1	1
FP	172	1787	1395	1954
FN	39	1	2	0
TP	5	43	42	44
TN	2872	1257	1649	1090
Test ROC	0.753	0.69659	0.72142	0.66456
family Id:	2280103	2280103	2280103	2280103
time	27058	67407	26263	41706
Train ROC	1	1	1	1
FP	6	150	75	198
FN	6	1	2	0
TP	0	5	4	6
TN	409	265	340	217
Test ROC	0.79639	0.73333	0.83494	0.77028
family Id:	2380401	2380401	2380401	2380401
time	24030	19099	23870	39697
Train ROC	1	1	1	1
FP	7	218	119	297
FN	5	3	3	2
TP	0	2	2	3
TN	606	395	494	316
Test ROC	0.73899	0.69788	0.72594	0.67798
family Id:	2380403	2380403	2380403	2380403
time	23565	43881	20057	35896
Train ROC	1	1	1	1
FP	18	586	335	715
FN	11	0	2	0
TP	0	11	9	11
TN	1331	763	1014	634
Test ROC	0.81084	0.77162	0.81838	0.64223
family Id:	2010101	2010101	2010101	2010101
time	24574	16444	21422	37210
Train ROC	1	1	1	1
FP	23	420	246	530
FN	28	2	7	0
TP	3	29	24	31
TN	1045	648	822	538
Test ROC	0.85877	0.77854	0.85411	0.71487

family Id:	2010102	2010102	2010102	2010102
time	35549	69029	23291	39591
Train ROC	1	1	1	1
FP	6	280	150	374
FN	20	1	9	0
TP	2	21	13	22
TN	752	478	608	384
Test ROC	0.8938	0.84673	0.83401	0.7434
family Id:	2010103	2010103	2010103	2010103
time	36416	25551	25787	42054
Train ROC	1	1	1	1
FP	2	103	59	142
FN	8	0	2	0
TP	0	8	6	8
TN	273	172	216	133
Test ROC	0.89227	0.82818	0.85955	0.74045
family Id:	2010104	2010104	2010104	2010104
time	23628	52991	25083	36164
Train ROC	1	1	1	1
FP	16	446	258	575
FN	29	6	11	0
TP	4	27	22	33
TN	1121	691	879	562
Test ROC	0.87511	0.76331	0.81392	0.76696
family Id:	2380405	2380405	2380405	2380405
time	23861	20991	25123	38420
Train ROC	1	1	1	1
FP	6	429	242	561
FN	9	2	4	0
TP	0	7	5	9
TN	1098	675	862	543
Test ROC	0.77063	0.73973	0.79952	0.76963
family Id:	2010105	2010105	2010105	2010105
time	33037	55229	27167	42837
Train ROC	1	1	1	1
FP	7	361	198	451
FN	27	8	14	0
TP	0	19	13	27
TN	923	569	732	479
Test ROC	0.79873	0.73728	0.75954	0.8268
family Id:	2050101	2050101	2050101	2050101
time	23248	15613	19153	23985
Train ROC	1	1	1	1
FP	32	889	553	1079
FN	10	0	2	0
TP	1	11	9	11
TN	1951	1094	1430	904

Test ROC	0.74401	0.77252	0.78196	0.65356
family Id:	2050103	2050103	2050103	2050103
time	25435	43367	21680	30059
Train ROC	1	1	1	1
FP	39	789	484	950
FN	10	4	7	2
TP	0	6	3	8
TN	1764	1014	1319	853
Test ROC	0.71891	0.66073	0.68375	0.7208
family Id:	2520102	2520102	2520102	2520102
time	30554	19872	26108	25159
Train ROC	1	1	1	1
FP	13	506	284	644
FN	5	1	3	0
TP	0	4	2	5
TN	1262	769	991	631
Test ROC	0.80518	0.7611	0.7942	0.82525
family Id:	3010801	3010801	3010801	3010801
time	28179	46336	23084	36077
Train ROC	1	1	1	1
FP	10	531	310	660
FN	8	3	7	0
TP	0	5	1	8
TN	1253	732	953	603
Test ROC	0.5191	0.70022	0.62381	0.73783
family Id:	3010803	3010803	3010803	3010803
time	25478	37373	20820	28618
Train ROC	1	1	1	1
FP	21	705	457	850
FN	10	1	6	1
TP	0	9	4	9
TN	1558	874	1122	729
Test ROC	0.53325	0.76833	0.69937	0.72058
family Id:	3420101	3420101	3420101	3420101
time	29577	51235	27237	38633
Train ROC	1	1	1	1
FP	18	436	247	563
FN	9	1	4	1
TP	1	9	6	9
TN	1087	669	858	542
Test ROC	0.80009	0.83629	0.80326	0.76914
family Id:	3420105	3420105	3420105	3420105
time	28193	21329	24589	35139
Train ROC	1	1	1	1
FP	19	582	365	736
FN	13	3	7	1
TP	0	10	6	12

TN	1418	855	1072	701
Test ROC	0.81361	0.79118	0.79011	0.77833
family Id:	3020102	3020102	3020102	3020102
time	23713	47738	23779	42135
Train ROC	1	1	1	1
FP	19	524	297	673
FN	16	8	14	1
TP	0	8	2	15
TN	1278	773	1000	624
Test ROC	0.45615	0.63232	0.62929	0.71709
family Id:	3420108	3420108	3420108	3420108
time	27697	27137	28267	44759
Train ROC	1	1	1	1
FP	8	207	112	271
FN	5	1	3	1
TP	0	4	2	4
TN	544	345	440	281
Test ROC	0.71667	0.87029	0.76957	0.76884
family Id:	3020103	3020103	3020103	3020103
time	40864	74989	30601	49005
Train ROC	1	1	1	1
FP	9	289	159	359
FN	9	1	5	1
TP	0	8	4	8
TN	721	441	571	371
Test ROC	0.63272	0.80472	0.77808	0.71065
family Id:	3320101	3320101	3320101	3320101
time	41131	28531	28541	42054
Train ROC	1	1	1	1
FP	11	298	164	380
FN	9	5	7	1
TP	0	4	2	8
TN	748	461	595	379
Test ROC	0.54999	0.6286	0.68116	0.74016
family Id:	3320108	3320108	3320108	3320108
time	29634	51490	26574	36368
Train ROC	1	1	1	1
FP	13	352	192	457
FN	11	2	7	2
TP	0	9	4	9
TN	914	575	735	470
Test ROC	0.63146	0.81593	0.7566	0.68402
family Id:	3020105	3020105	3020105	3020105
time	33804	23211	28409	40438
Train ROC	1	1	1	1
FP	2	203	122	263
FN	7	3	5	0

TP	0	4	2	7
TN	565	364	445	304
Test ROC	0.70345	0.78685	0.73822	0.76291
family Id:	3020106	3020106	3020106	3020106
time	45686	61853	41643	67398
Train ROC	1	1	1	1
FP	2	144	77	196
FN	5	0	4	1
TP	0	5	1	4
TN	403	261	328	209
Test ROC	0.55457	0.86963	0.72	0.67259
family Id:	3030102	3030102	3030102	3030102
time	38037	24262	29210	55911
Train ROC	1	1	1	1
FP	18	427	245	538
FN	7	4	5	2
TP	0	3	2	5
TN	1025	616	798	505
Test ROC	0.47747	0.62608	0.64772	0.67813
family Id:	3020107	3020107	3020107	3020107
time	52156	74645	41455	42222
Train ROC	1	1	1	1
FP	3	138	69	183
FN	5	1	4	0
TP	0	4	1	5
TN	402	267	336	222
Test ROC	0.52247	0.75062	0.67704	0.82123
family Id:	3030105	3030105	3030105	3030105
time	28967	14644	19301	21474
Train ROC	1	1	1	1
FP	70	1161	792	1350
FN	16	5	10	1
TP	0	11	6	15
TN	2315	1224	1593	1035
Test ROC	0.56858	0.70928	0.6298	0.70165
family Id:	3020104	3020104	3020104	3020104
time	49338	65028	29835	29757
Train ROC	1	1	1	1
FP	8	196	90	263
FN	7	5	6	0
TP	0	2	1	7
TN	559	371	477	304
Test ROC	0.65734	0.62006	0.61678	0.81532
family Id:	3320111	3320111	3320111	3320111
time	30108	24079	32008	30477
Train ROC	1	1	1	1
FP	4	154	80	204

FN	5	2	3	1
TP	0	3	2	4
TN	417	267	341	217
Test ROC	0.46841	0.75629	0.74489	0.66271
family Id:	3320113	3320113	3320113	3320113
time	18363	56085	16016	18038
Train ROC	1	1	1	1
FP	7	247	142	330
FN	8	4	7	2
TP	0	4	1	6
TN	667	427	532	344
Test ROC	0.46903	0.6658	0.63501	0.67674
family Id:	7030502	7030502	7030502	7030502
time	19109	19185	8860	7898
Train ROC	1	1	1	1
FP	22	820	501	992
FN	3	0	1	0
TP	6	9	8	9
TN	1724	926	1245	754
Test ROC	0.97938	0.74691	0.78611	0.66921
family Id:	7031001	7031001	7031001	7031001
time	3562	3373	1805	2292
Train ROC	1	1	1	1
FP	752	2719	2450	2817
FN	7	0	0	0
TP	88	95	95	95
TN	2901	934	1203	836
Test ROC	0.87875	0.61051	0.66286	0.62679
family Id:	7030601	7030601	7030601	7030601
time	28404	28441	14482	16180
Train ROC	1	1	1	1
FP	7	352	172	452
FN	5	1	1	0
TP	4	8	8	9
TN	866	521	701	421
Test ROC	0.9356	0.81341	0.91129	0.84243
family Id:	7030602	7030602	7030602	7030602
time	21378	13288	6736	7984
Train ROC	1	1	1	1
FP	98	1379	968	1576
FN	8	0	0	0
TP	18	26	26	26
TN	2425	1144	1555	947
Test ROC	0.93485	0.70773	0.78105	0.68937
family Id:	7030604	7030604	7030604	7030604
time	45553	31132	15527	15554
Train ROC	1	1	1	1

FP	4	179	81	244
FN	4	0	0	0
TP	1	5	5	5
TN	481	306	404	241
Test ROC	0.98845	0.87918	0.90598	0.81402
family Id:	7390102	7390102	7390102	7390102
time	18627	25231	13613	15953
Train ROC	1	1	1	1
FP	10	478	278	590
FN	6	0	0	0
TP	1	7	7	7
TN	1111	643	843	531
Test ROC	0.93883	0.69963	0.94711	0.72716
family Id:	7390103	7390103	7390103	7390103
time	13286	14137	8761	7922
Train ROC	1	1	1	1
FP	63	1117	749	1319
FN	9	2	2	0
TP	5	12	12	14
TN	2179	1125	1493	923
Test ROC	0.95476	0.72439	0.77657	0.64668
family Id:	7410501	7390103	7410501	7410501
time	11723	13501	9575	11327
Train ROC	1	1	1	1
FP	55	1117	619	1150
FN	8	2	2	0
TP	1	12	7	9
TN	1961	1125	1397	866
Test ROC	0.94929	0.69963	0.76681	0.75755
family Id:	7410502	7410502	7410502	7410502
time	11458	19182	10200	10010
Train ROC	1	1	1	1
FP	45	940	609	1127
FN	9	0	0	0
TP	0	9	9	9
TN	1971	1076	1407	889
Test ROC	0.90625	0.81481	0.87164	0.80065