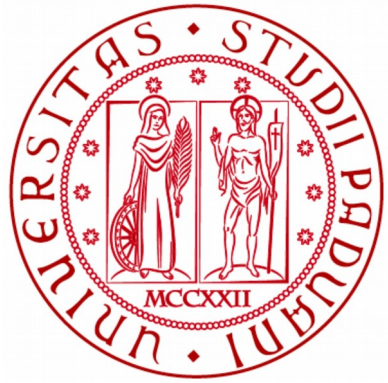

Università degli Studi di Padova
Dipartimento di Matematica "Tullio Levi-Civita"
Corso di Laurea in Informatica

Applicazione di tecniche di intelligenza artificiale per la verifica di programmi gestionali

Laureando
Alessandro Pol

Relatore
Prof. Tullio Vardanega

Anno Accademico 2016/2017

Sommario

In questo documento presento l’esperienza di stage che ho svolto presso Zucchetti SpA in collaborazione con l’Università di Padova.

Il lavoro è suddiviso in quattro capitoli. Nel primo capitolo è descritta l’azienda, il contesto lavorativo ed i prodotti/servizi erogati.

Nel secondo capitolo è presentato il progetto dello stage, analizzando la proposta fatta dall’azienda ospitante e le motivazioni che mi hanno spinto ad accettarla.

La spiegazione tecnica su quanto prodotto, esplicata nel terzo capitolo, espone l’analisi del problema e illustra le soluzioni tecnologiche adottate.

Infine, nell’ultimo capitolo, sono presenti le considerazioni finali e personali su quanto fatto ed imparato.

Indice

1 Azienda

1.1 Zucchetti SpA

Fig. 1.1: Logo Zucchetti SpA
Fonte: www.zucchetti.it

Zucchetti è stata fondata da Domenico Zucchetti a Lodi nel 1977 per fornire applicativi per l'automatizzazione delle procedure contabili aziendali. L'azienda poi crebbe e decise di estendere i propri obiettivi commerciali cominciando a trattare anche dispositivi *hardware*.
Con questa scelta iniziò non solo ad offrire soluzioni interne ed amministrative alle aziende, ma anche prodotti che potessero essere usati dalle stesse per produrre guadagno o ridurre i costi di gestione. Uno dei più importanti vantaggi competitivi per i clienti, risultava quello di avere un unico referente in grado di rispondere alle diverse esigenze di carattere informatico nelle varie fasi di lavoro:

- pre-vendita: comprensione necessità delle cliente e proposta delle soluzioni più adeguate;
- post-vendita: installazione del programma e assistenza tecnica post vendita;
- formazione del personale per utilizzare al meglio tutte le potenzialità dei prodotti installati;
- aggiornamento continuo sulle tematiche legate al mondo amministrativo, contabile e fiscale.

Grazie al grande successo ottenuto ed alla sempre più crescente notorietà, cominciò a stringere rapporti e rilevare altre aziende nel contesto dell'IT, riuscendo cosi ad approcciare nuovi mercati, migliorare i prodotti interni ed espandere il proprio

marchio. Tale strategia venne applicata non solo in Italia ma anche all'estero, in paesi come Stati Uniti, Brasile e Cina, con l'apertura di sedi e filiali.

Al giorno d'oggi l'azienda è una realtà consolidata e riconosciuta nel campo dell'IT italiano ed estero e conta più di 135.000 clienti, 900 partner, 3.350 addetti con un fatturato e popolarità in costante crescita.

1.2 Contesto lavorativo

Alla nascita dell'azienda, l'Italia era verso la fine boom economico degli anni 60' ed i computer, grazie ad aziende come Olivetti, cominciavano ad entrare negli uffici aziendali e negli ambienti accademici. Zucchetti quindi è una delle prime realtà italiane che offre soluzioni informatiche ad enti che cercano migliorare i propri meccanismi di gestione aziendali attraverso l'*Information Tecnology (IT)*. Lo scopo di Zucchetti era di aiutare la conversione al digitale delle aziende italiane, fornendo servizi e prodotti per la creazione e gestione di documenti elettronici cosi da migliorare l'accesso alle informazioni in tempo reale.

Da queste basi Zucchetti comincia la propria crescita fino ai giorni nostri ed è possibile affermare che tale sviluppo va di pari passo e nella stessa direzione con quello della tecnologia informatica. In quasi quarant'anni l'industria dell'IT ha drasticamente cambiato la vita dell'uomo sia in ambito lavorativo che sociale. Tale cambiamento ha offerto numerose opportunità di business in svariati ambiti, dove Zucchetti è riuscita ad inserirsi e proporre soluzioni *software* ed *hardware* per differenti contesti sociali e lavorativi.

A dimostrare tale successo è il variegato pacchetto clienti dell'azienda, composto da aziende che operano in mercati e settori anche molto diversi tra loro. Sono presenti sia aziende storiche leader nel proprio settore (Toyota per l'automobilismo, Fininvest per le telecomunicazione) che realtà più giovani emerse ultimamente e in forte sviluppo: Il Fatto Quotidiano nel giornalismo, BBC nel settore bancario e Snai nel mondo delle scommesse sportive.

Grazie alla politica di partnership ed acquisizioni, l'azienda è riuscita ad allargare il proprio raggio d'azione in diversi campi di ricerca e produzione. Tra i vari temi sono da notare l'*Internet of Things*, l'intelligenza artificiale e la robotica.
Fondamentale per lo sviluppo di questo settore, l'entrata nel gruppo di Zucchetti Centro Sistemi (ZCS), azienda nata come produttrice di *software* che velocemente si afferma nei mercati internazionali della robotica e automazione. ZCS è una realtà fortemente dinamica proiettata alla realizzazione di prodotti competitivi, innovativi e performanti.

1.3 Prodotti

La completezza dell'offerta di Zucchetti spazia dalle soluzioni *software* gestionali alle soluzioni *hardware* a servizi innovativi progettati e realizzati per soddisfare le esigenze di un variegato pacchetto clienti:

- aziende: Zucchetti è in grado di assistere sia realtà di piccole dimensioni che società strutturate;
- professionisti: commercialisti, consulenti del lavoro ed appartenenti a studi professionali;
- pubblica amministrazione: dai piccoli comuni alle società pubbliche.

L'esperienza maturata, affiancando realtà in differenti settori, ha permesso al gruppo di sviluppare negli anni *software* gestionali e servizi di qualità sempre maggiore, garantendo all'utente la migliore soluzione presente sul mercato.

Vengono costruiti sistemi sia per la gestione automatizzata di flussi di lavoro per aziende, aziende sanitarie, lavanderie e palestre che robot destinati all'utilizzo in campo agricolo (per la raccolta olive ed uva) o ricreativo (gestori di piscine).
L'azienda fornisce i propri servizi anche al settore della ristorazione (gestionali per hotel ed alberghi) ed al più innovativo settore della stampa in 3D. Riuscire ad approcciare questo mercato è fondamentale per l'azienda, poiché la stampa in 3D, utilizzando il laser e scansionando strutture tridimensionali di materiali differenti,

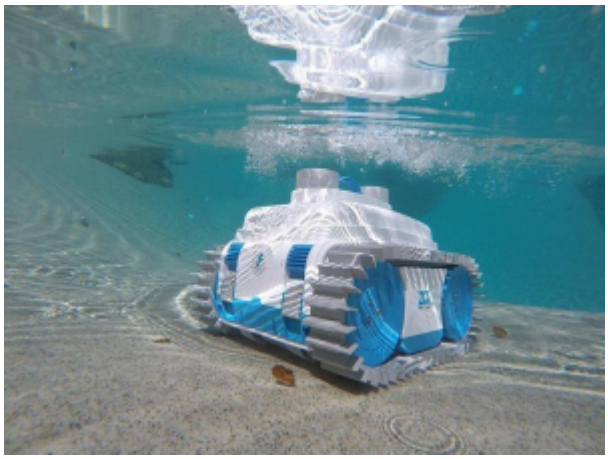

Fig. 1.2: NEMH2O
Fonte: www.nemorobot.it

trova applicazione in svariati settori industriali.

Tra i prodotti offerti dall'azienda, uno dei più importanti è il *framework Infinity*, un set di tools che offre la possibilità di creare applicazioni *web-based*. La suite permette di gestire in maniera integrata numerosi processi aziendali: logistica, magazzino, acquisti, contabilità, amministrazione, vendite, marketing, risorse umane ecc.

Il grande pregio di questo strumento è quello di permettere all'azienda di approcciare sia il mercato B2B che quello B2C.
Nel primo caso fornisce alle *software house* uno strumento per lo sviluppo di ERP personalizzabili in maniera semplice, intuitiva e sicura che rispettano i principi dell'ingegneria del *software*.
Nel secondo caso, grazie all'elevata usabilità del prodotto costruito, permette all'utente finale di utilizzare un *software* adatto al proprio contesto lavorativo e modificabile in base alle esigenze e alle necessità proprie dell'azienda. Uno dei vantaggi principali per il cliente finale è la possibilità di intervenire sul prodotto da remoto ed in tempi rapidi.

1.4 Tecnologie

Questo *framework*, sviluppato principalmente nella sede di Padova, ha il suo ambiente di applicazione di servizi nel web. Le tecnologie usate da Zucchetti sono da ricercare tra gli strumenti che permettono di scrivere applicazioni *web-based*.

Vengono dunque utilizzati principalmente tre linguaggi, ognuno per uno specifico compito del partner architetturale MVC: Java, Javascript ed SQL.

Javascript è usato principalmente per l'applicazioni web lato *client*. Oltre a fornire strumenti per la creazione di UI dinamiche ed interattive, al giorno d'oggi Javascript permette di scrivere anche codice lato server grazie al *framework* NodeJS.

Il *controller* invece è implementato usando Java: in questo linguaggio vengono sviluppate tutte le funzioni che permettono lo scambio e l'elaborazione dell'informazione tra utente e database.

Infine, i database in cui opera *Infinity* sono del tipo relazionale, dunque viene usato SQL come linguaggio principale per l'accesso e la conservazione dei dati.

1.5 Futuro ed innovazione

Come ha dimostrato nel corso degli anni Zucchetti fa dell'innovazione il proprio obiettivo principale. Svariati campi di interesse dell'azienda richiedono una forte

base di studio e ricerca, componenti essenziali per effettuare una corretta analisi dei problemi.

Lo dimostrano i finanziamenti di progetti su temi contemporanei e dal futuro certo in cui Zucchetti ha deciso di puntare. IoT, robotica, stampa3D, intelligenza artificiale sono alcuni degli esempi in cui Zucchetti ha deciso di concentrare i propri sforzi certa che, come in passato, tali investimenti potranno portare benefici e guadagni.

Da sottolineare un progetto avviato dell'azienda chiamato Accademia Zucchetti che si rivolge a imprese e dipendenti fornendo una formazione tecnica, applicativa e legislativa.

Questo progetto è in linea con lo spirito dell'azienda, da sempre orientata al cliente, a capire le sue necessità e ad affiancarlo nella formazione e consulenza, in un contesto in continua evoluzione e in costante aggiornamento come quello delle tecnologie e dell'informatica.

Fig. 1.3: Logo Accademia Zucchetti
Fonte: www.apogeo.it

2 Stage

2.1 Zucchetti e Università di Padova

Tra Zucchetti e l'Università di Padova esiste un rapporto consolidato nel corso del tempo. Tale collaborazione è dovuta ai diversi benefici che i vari attori coinvolti traggono dal progetto: l'università, l'azienda e lo studente.

Dal punto di vista universitario, grazie a Zucchetti, il dipartimento riesce a fornire agli studenti esempi di problemi inerenti al mondo del lavoro. Nel corso di Ingegneria del *Software*, per esempio, lo scopo è simulare una *start-up* informatica e spesso viene presentato un progetto di Zucchetti.

Fig. 2.1: Logo StageIT 2017
Fonte: www.progettogiovani.pd.it/stage-it-2017/

L'azienda, durante lo svolgimento del progetto, effettua colloqui con gruppi di lavoro, fornendo un' analisi dettagliata e instaurando una continua comunicazione con gli studenti. A progetto terminato, spesso il tutor aziendale è presente alla presentazione finale durante la quale esprime la propria opinione sull'elaborato.

Dal punto di vista dell'azienda invece, l'università offre diverse opportunità. Grazie all'attività di stage, può effettuare degli studi di fattibilità di strumenti d'interesse o istruire gli studenti per una futura assunzione. In questo modo riesce ad avviare un progetto con costi relativamente bassi e valutare gli elementi che hanno influenzato in maniera positiva e negativa l'elaborato.

Infine, lo studente che partecipa può effettuare i suoi primi passi nel mondo del lavoro. In questo modo può apprendere il significato del lavoro in team, le dinamiche aziendali e la complessità che sta dietro alla progettazione e realizzazione di un *software*.

A dimostrare quanto l'azienda creda nei progetti di stage e li supporti ci sono i numerosi premi vinti per il migliore stage, riconoscimento conferito annualmente nell'ambito dell'iniziativa STAGE-IT.

2.2 Descrizione proposta di stage

2.2.1 Contesto e motivazioni

Zucchetti proponeva quattro progetti ed ognuno puntava a creare miglioramenti o strumenti aggiuntivi per *Infinity*.

ll programma *Infinity*, tra i diversi servizi, offre all'utente la possibilità di creare complesse procedure di elaborazione in modalità *drag and drop*. L'utente usa dei *widget* che rappresentano variabili, chiamate al database e funzioni (esistenti o personalizzate) per creare un flusso di elaborazione. Il programma viene poi codificato ed eseguito assieme alle altre funzioni al momento richiesto.

```
1  fun(x)
2  {
3    if x==10 then
4      y=x-10
5    else
6      y=|x|+1
7
8    if y>=0 then
9      y=(y+1)/x
10   else
11     y=x/y
12 }
```

Fig 2.2: Esempio di istruzione (n. 11) non eseguibile

In tale processo, ed in generale con qualsiasi linguaggio di programmazione, solitamente il programmatore inserisce all'interno del codice delle strutture di controllo (*if-then-else*, *while*, *case-of*) che, in base alle condizioni delle variabili dichiarate, decidono se eseguire un blocco di codice, quando o per quante volte.

In funzioni complesse in cui sono presenti strutture di controllo annidate, è facile che l'unione di tali condizioni porti ad espressioni logico-aritmetiche contraddittorie ed impossibili da soddisfare. Un esempio è la funzione nella figura 2.2 in cui è facile notare come l'istruzione alla riga 11 non verrà mai eseguita in quanto *y* sarà sempre maggiore o uguale a zero. Tali errori sono difficili da identificare e possono creare comportamenti scorretti ed imprevedibili dei componenti dell'architettura.

Una possibile soluzione a questo problema è data dall'analisi statica, nello specifico dall'esecuzione simbolica: il codice viene eseguito non operando sul valore effettivo dell'*input* ma su quello simbolico.

In particolare, ad ogni istruzione viene associata un'espressione chiamata *path*

condition (*PC*).
La *PC* è un'espressione booleana formata dall'unione delle condizioni iniziali, le operazioni aritmetiche e le condizioni delle strutture di controllo collegate alla singola istruzione. Relativamente alle strutture dei controllo, la *path condition* viene duplicata: una sarà composta dalla condizione che rende vero lo *statement* e continuerà eseguendo il blocco di codice; l'altra, invece, conterrà la negazione della stessa e continuerà l'esecuzione passando all'istruzione successiva del blocco.
La *PC*, composta da formule logiche ed algebriche formate con i simboli delle variaibili, esprime la condizione che i dati devono soddisfare per eseguire l'istruzione.

```
1 fun(x)
2 {
3   y;
4   if x<0||x>10then
5      y=-x
6   else
7     y=x
8   return y
9 }
```

Fig 2.3: Funzione rappresentata nell'albero di esecuzione in fig 2.4

Tulle le possibili combinazioni della *PC*, dalla prima istruzione fino all'ultima, formano l'albero di esecuzione. Tale grafo orientato rappresenta i possibili flussi di elaborazione della procedura, vincolati dalle proprietà descritte nella *PC*. In presenza di costrutti iterativi i cammini possono essere infiniti.

Un esempio di albero di esecuzione, è quello riportato in fig. 2.4 e sviluppato partendo dal codice della funzione in fig. 2.3.

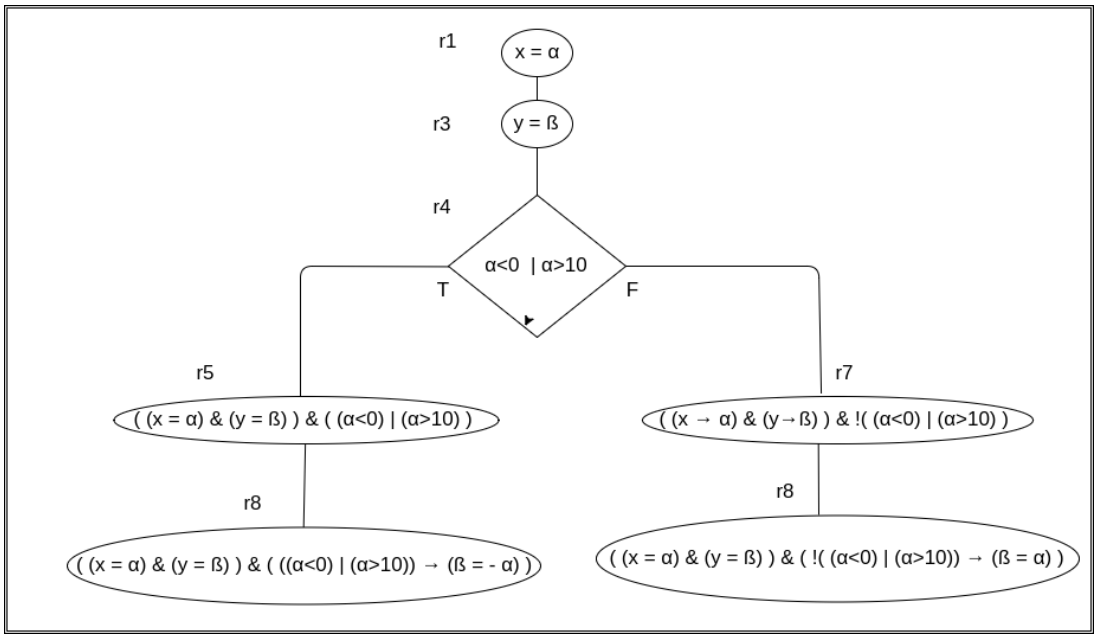

Fig 2.4: Albero di esecuzione della funzione in fig. 2.3

Si noti come l'albero mostri ogni possibile cammino di elaborazione, terminando in foglie che contengono la condizione dell'*output* prodotto.

Tramite l'albero di esecuzione è possibile ricavare diverse informazioni sia di natura funzionale che di dati trattati.
In generale è un problema indecidibile, ma in alcuni casi è possibile provare se esistono valori per cui la *PC* sia vera o falsa usando strumenti come dimostratori di teoremi e programmazione lineare.

Nel caso sia non soddisfacibile significa che non esiste *input* tale per cui quel blocco di codice venga eseguito. Se ogni diramazione che punta alla medesima riga di codice contiene una formula non soddisfacibile, significa che quella porzione di codice non verrà mai eseguita. Per la funzione in figura 2.1 per esempio, le *PC* dei nodi etichettati con r11 sarebbero insoddisfabili su tutto l'albero d'esecuzione.

L'albero inoltre fornisce precise informazioni sull'*input* e sull'*output* della funzione. Tali dati possono essere utili per testare il comportamento della procedura in tutti i possibili stati di esecuzione. E' possibile confrontare le diverse espressioni di *output* prodotte da ogni funzione, potendo cosi estendere la verifica non solo ad una singola funzione, ma a tutti i componenti del programma e le loro interazioni.

2.2.2 Proposta di stage

Partendo da questa premessa, di seguito è la proposta di stage originale:

Realizzazione di un verificatore automatico con tecniche di intelligenza artificiale, controllo di correttezza e l'ottimizzazione di programmi a partire da specifiche di alto livello.

Lo scopo dello stage dunque era la creazione di un set di strumenti che, mediante l'uso di tecniche di IA, potesse valutare la qualità del codice prodotto da *Infinity* secondo specifiche di alto livello. In dettaglio, tali programmi dovevano individuare se nei codici sorgenti fossero presenti condizioni affinché una procedura si venisse a trovare in uno stato di errore logico o invalido. Nel contesto degli ERP prodotti da *Infinity* vengono svolte principalmente quattro funzioni principali: data entry, controllo dati, archiviazione ed elaborazione.

2.2.3 Vincoli di progetto

L'azienda ha imposto due vincoli molto significativi nello sviluppo del progetto: la programmazione funzionale e l'intelligenza artificiale.

2.2.3.1 Programmazione funzionale

La logica di implementazione del *software* doveva rispettare i paradigmi della programmazione funzionale. Questo perché tali linguaggi permettono di produrre *software* che possono essere facilmente eseguiti su elaboratori che sfruttano il calcolo parallelo e distribuito.

La necessità di utilizzare questo tipo di paradigma deriva dall'esigenza di far fronte ad alcuni problemi riguardanti le architetture degli elaboratori.

Nello specifico, a causa delle dimensione dei componenti e dalla poca resistenza dei materiali agli sbalzi termini, ad oggi non è stato ancora possibile aumentare in maniera sostanziale la velocità di elaborazione dei processori, che di fatto rimane sui livelli di inizio secolo.

Per aumentare le prestazioni degli elaboratori si è deciso quindi di fornire le schede madri di più processori in modo che potessero distribuire il carico di lavoro. Nella maggior parte dei casi però sono pochi gli elaboratori che sfruttano a pieno le architetture *multicore.* Questo avviene principalmente per tre motivi:

- Problematiche connesse all'*hardware*: anche se il sistema operativo vede *n* processori fisici, spesso questi condividono la stessa area memoria e *bus*

diminuendo, cosi, considerevolmente la capacità di lavorare parallelamente in modo indipendente;

- Problematiche connesse ai sistemi operativi moderni che non sono ancora implementati al meglio per eseguire programmi su più processori.
- Problematiche legate al *software*. La maggior parte dei *software* in circolazione infatti sono sviluppati con i paradigmi di programmazione imperativa che favoriscono la dipendenza tra dati e processi. Nello specifico le funzioni effettuano *side-effect* modificando variabili al di fuori del proprio *scoping*, interferendo con l'ambiente esterno, creando problemi di sincronizzazione e integrità dei dati tra processi.

A differenza della programmazione imperativa, nella programmazione funzionale, invece, ogni variabile è di sola lettura. Questo perché ogni valore non viene creato modificando lo stato del programma, ma costruendo nuovi valori partendo dai precedenti. Grazie alle variabili immutabili si ottiene la trasparenza referenziale delle funzioni: ogni istruzione diventa automaticamente *thread-safe* eliminando il bisogno di sincronizzare l'accesso ad un oggetto mutabile con i relativi rischi e problematiche. Con tali basi diventa possibile distribuire il lavoro tra diverse unità di elaborazione aumentando la stabilità, la correttezza dei programmi e risolvendo i problemi legati alla velocità di elaborazione dei processori.

Per sviluppare un *software* con queste caratteristiche e che rispondesse ai parametri aziendali, ho applicato le seguenti regole basandosi sui principi della programmazione funzionale:

- Un programma è una funzione, definita in termini di altre funzioni;
- Le funzioni sono pure, cioè funzioni senza effetti collaterali;
- Le funzioni sono *first class object*, cioè trattate come un qualsiasi tipo di dato; nello specifico possono essere passate come parametri ad una funzione, assegnate ad una variabile e restituite come valore.
- Vengono usati variabili immutabili e non esiste la variazione di stato. Le funzioni hanno una rappresentazione matematica ossia associano a ogni elemento del dominio un unico elemento nel codominio;
- La ricorsione è la principale struttura di controllo;
- Liste e derivati (alberi binari, *n*-alberi) devono essere le principali strutture dati, cioè collezioni con struttura interna ricorsiva.

La scelta di applicare il paradigma funzionale comporta la rinuncia di concetti caratteristici della programmazione imperativa quali ciclo, assegnazione di variabili e stato della memoria.
L'obiettivo primario di un programmatore è quello di disegnare una funzione espressa in termini matematici che può fare uso di un certo numero di funzioni ausiliarie per risolvere un dato problema. In generale, si ottiene un programma corretto se:

- ciascuna funzione è stata implementa correttamente;
- le relazioni tra tutte le funzioni coinvolte sono corrette.

Characteristic	Imperative approach	Functional approach
Programmer focus	How to perform tasks (algorithms) and how to track changes in state.	What information is desired and what transformations are required
State changes	Important.	Non-existent
Order of execution	Important.	Low Importance.
Primary flow control	Loops, conditionals, and function (method) calls	Function calls, including recursion
Primary manipulation unit	Instances of structures or classes	Function as first-class objects and data collections.

Fig 2.5: Differenze approccio imperativo e funzionale
Fonte: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/functional-programming-vs-imperative-programming

2.2.3.2 Intelligenza artificiale

Il secondo vincolo riguarda la progettazione del sistema. L'azienda ha chiesto di implementare un *software* che utilizzasse metodi di intelligenza artificiale. Ciò significa che il programma doveva essere istruito e reagire secondo la propria esperienza per applicare l'azione corretta.

Nel campo dell'intelligenza artificiale, un agente è un sistema che percepisce il proprio ambiente attraverso i sensori ed agisce su di esso medianti attuatori eseguendo la giusta operazione. Viene usato il termine percezione per indicare gli *input* percepiti dall'agente dall'ambiente in un dato istante. L'insieme di percezioni in un intervallo di tempo viene definita sequenza percettiva ed è strettamente legata alle azioni di reazioni eseguite dall'agente.
In termini matematici l'agente è definito come una funzione che mappa ogni

possibile sequenza di percezioni su un’azione che l'agente è in grado di eseguire.

Esistono differenti tipologie di agenti che si distinguono per la capacità di interagire con l'ambiente esterno, la quantità di dati percepiti e dalle tecniche usate per l’apprendimento ed il ragionamento.

L’azienda ha richiesto lo sviluppo di un agente semplice basato sulla conoscenza. Il componente più importante di tali agenti è la base di conoscenza (*Knoweledge base*) costituita da un insieme di formule espresse in un determinato linguaggio. L’agente semplice ha il compito di accedere alla conoscenza e reagire in base alle istruzioni in essa contenuta.
L’agente doveva possedere conoscenza di tipo sia dichiarativo che operativo. Nello specifico doveva avere conoscenze che definissero le entità dell’ambiente esterno e che consentissero di agire in base alle percezioni ricevute.

Agenti con queste caratteristiche sono chiamati i sistemi esperti, ossia sistemi intelligenti specializzati in compiti diversi, grazie alla vasta base di conoscenza di cui sono dotati. Utilizzano la propria base di conoscenza per generare soluzioni utilizzando regole e procedimenti di derivazione.

La base di conoscenza deve prevedere meccanismi di accesso per eseguire operazioni di scrittura e lettura da parte dell’utente.
La motivazione alla base di tale scelta è da ricercarsi nell’esigenza di estendere le funzioni del *software* aggiungendo nuove regole senza dover modificare drasticamente il programma. Tale aspetto è fondamentale in campi come la logica e l’aritmetica in quanto spesso viene richiesto di applicare regole di applicazione sui determinati *input* tenendo in considerazione priorità delle operazioni e condizioni d’utilizzo.

2.2.4 Aspettative ed obiettivi

Data la quantità di obiettivi e vincoli del progetto proposto ed i possibili casi di sviluppo, assieme al tutor, si è deciso non tanto di creare un *software* che facesse tutto il lavoro, ma un insieme di strumenti che eseguissero parte delle operazioni richieste a livello globale, rispettando i vincoli riguardanti l’intelligenza artificiale e la programmazione funzionale.

Oltre a cercare nuovi possibili metodi di analisi del codice sorgente prodotto, il tutor voleva capire le difficoltà ed i vantaggi nello scrivere programmi di intelligenza artificiale usando i paradigmi della programmazione funzionale. L’obiettivo primario non era principalmente la costruzione di un *debugger* simbolico, progetto

che avrebbe richiesto tempo e risorse non adatte per un progetto di stage, ma dimostrare che usando i principi della programmazione funzionale è possibile creare programmi intelligenti, sicuri e performanti in grado di sfruttare a pieno le risorse *hardware* degli elaboratori moderni. Tali risultati sarebbero serviti, successivamente, a considerare l'uso di nuove tecnologie e metodologie di sviluppo differenti da quelli usati al giorno d'oggi dall'azienda.

Ho sviluppato due strumenti: un semplificatore di espressioni ed un dimostratore di teoremi.
Il semplificatore è in due versioni simili negli obbiettivi ma differenti nella struttura. Nella prima ho applicato i paradigmi di programmazione funzionale per semplificare espressioni logico-algebriche. Nella seconda, in aggiunta al vincolo precedente, ho usato tecniche di IA per semplificare formule in logica proposizionale. Il motivo risiedeva nella necessità di prendere confidenza con la programmazione funzionale ed il linguaggio usato e, soprattutto, per valutare le differenze tra un algoritmo classico ed uno intelligente.
Per quanto riguarda il dimostratore di teoremi, l'azienda ha imposto due obbiettivi, uno massimo ed uno minimo. Il minimo era l'implementazione di un dimostratore di teoremi per la logica delle proposizioni; quello massimo, invece, per la logica dei predicati.

Infine, l'azienda ha chiesto di analizzare, tramite tecniche di *machine learning*, i dati riguardanti i processi di sviluppo di Zucchetti. Tali dati provenienti dalle *repository* sono stati raccolti attraverso strumenti di monitoraggio.
Si volevano utilizzare algoritmi di *machine learning* per ricavare informazioni riguardanti gli errori commessi nelle fasi di analisi, progettazione e codifica che avevano portato alla modifica del codice. L'azienda era interessata a conoscere i vantaggi e gli svantaggi dell'usare algoritmi di classificazione differenti valutando la qualità dei risultati ed il tempo di esecuzione. Usando tali tecnologie, in futuro sarà in grado di produrre strumenti capaci di migliorare la conoscenza dei programmatori attraverso l'analisi degli errori commessi durante la fase di codifica.
Per effettuare questa analisi ho usato il programma Orage, un *toolkit opensource* che mette a disposizione differenti algoritmi per l'elaborazione di dati grezzi e di visualizzazione delle informazioni ricavate.

2.3 Piano di lavoro

Nel corso dei colloqui con il tutor aziendale è stato steso il piano di lavoro per comprendere la quantità di lavoro, le tempistiche e gli obbiettivi da raggiungere. Lo

stage ha avuto una durata di otto settimane. Assieme al tutor abbiamo scelto di svolgere le seguenti attività:

Settimana 1: Studio teoria e tecnologie

- Introduzione concetti di intelligenza artificiale, logica e linguaggio Lisp.

Settimana 2: Semplificatore di espressioni

- Risultati attesi: un programma che semplificasse le espressioni logiche che nascono dall'interpretazione simbolica di una procedura scritta con i sistemi interni Zucchetti. Le espressioni dovevano contenere solo termini semplici (operazioni relazionali) unite da operazioni logiche and, or e not in forma normale congiunta o disgiunta.

Settimana 3: Creazione sistema di produzione

- Risultati attesi: il programma di semplificazione doveva essere tradotto in regole interpretate da una funzione che le vedeva come suoi dati.

Settimana 4: Test e documentazione

- Risultati attesi: suite di test per validare i risultati delle settimane precedenti. Verifica dei tempi di esecuzione dei programmi in contesti d'uso reale. Documentazione degli algoritmi sviluppati.

Settimana 5: Implementazione dimostratore di Teoremi

- Risultati attesi: algoritmo di unificazione, programma di dimostrazione di teoremi utilizzando la tecnica dei *tableaux* semantici; risultato mimino: dimostratore di teoremi della logica delle proposizioni, risultato massimo: dimostratore di teoremi della logica dei predicati.

Settimana 6: Debugger simbolico

- Risultato atteso: debugger simbolico per i programmi sviluppati in Zucchetti, lo stage avrebbe dovuto fornire gli elementi descritti dai risultati precedenti e il personale della Zucchetti li avrebbe inseriti nei propri sistemi.

Settimana 7: Machine Learning

- Risultato atteso: procedura stand-alone che ricevendo i dati di maturità del codice indicasse quali dei problemi rilevati con i sistemi sviluppati nelle settimane precedenti fossero quelli da verificare per primi.

Settimana 8: Test e documentazione

- Risultati attesi: suite di test per validare i risultati delle settimane precedenti. Verifica dei tempi di esecuzione dei programmi in contesti d'uso reale. Documentazione degli algoritmi sviluppati.

2.4 Motivazioni personali

Le ragioni che mi hanno spinto a scegliere questo stage sono diverse, ma in particolare le seguenti hanno influenzato la mia scelta.

Prima di tutto l'intelligenza artificiale. Tra le tante branche dell'informatica l'IA è sicuramente la materia di studio più affrontata al momento tanto che grosse aziende

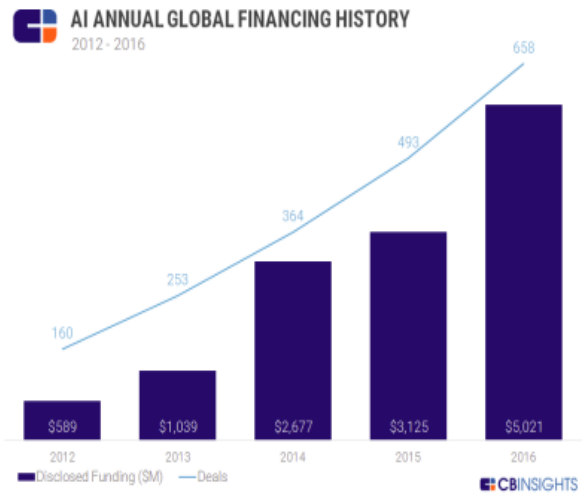

Fig. 2.6: Finanziamenti globali in dollari a progetti di intelligenza artificiale 2012-2016
Fonte: www.techemergence.com/venture-investments-in-artificial-intelligence-trends-in-2016-and-beyond/

informatiche stanno investendo massivamente in differenti ambienti d'applicazione. È chiaro quindi che l'informatico moderno dovrà avere, oltre le conoscenze basi, anche nozioni sui relativi problemi e soluzioni che tale disciplina richiede. La mia scelta va anche oltre l'aspetto lavorativo in quanto ritengo la materia molto produttiva ed affascinante per filosofia di pensiero, la logica di base e soluzioni architetturali ed algoritmiche adottate. La scelta di usare i paradigmi della programmazione funzionale è stato un ulteriore fattore che mi ha spinto ad accettare

la proposta. Oltre l'evoluzione del *hardware*, in questo periodo si nota anche un cambio di tendenza dei linguaggi di programmazione sia di uso comune che moderni.
Diversi linguaggi infatti hanno cominciato ad approcciare ai paradigmi della programmazione funzionale. Degli esempi sono Javascript e Scala, che danno la possibilità di trattare le funzioni come elemento di prim'ordine del programma e facilitano un stile di programmazione basato su set di funzioni. Inoltre, dati i problemi delle architetture attuali e le risposte fornite dalla programmazione funzionale, diventa molto importante uscire da un ambito accademico con delle conoscenze a riguardo. Sono pochi infatti i programmatori a saper programmare con tali vincoli data la difficoltà nell'apprendere ed applicare certi principi una volta abituati alla metodologia imperativa.

In secondo luogo l'azienda è stato un fattore rilevante nella scelta. Cercavo un'azienda che fosse organizzata, conosciuta nel mondo dell'IT e propensa alla ricerca. In Zucchetti ho trovato la risposta alle mie esigenze. Gli incontri effettuati con il tutor dell'azienda Gregorio Piccoli, in cui esprimeva entusiasmo e future applicazioni del progetto, mi hanno convinto ad accettare la proposta, certo che alla fine, avrei incrementato la mia conoscenza nel mondo del lavoro e dell'intelligenza artificiale.

3 Progetto

3.1 Modello di sviluppo

L'azienda non ha imposto un particolare modello di sviluppo del *software*. La metodologia di lavoro usata ha diverse aspetti in comune con il modello evolutivo. Il motivo è dovuto alle diverse fasi di analisi, progettazione, codifica e verifica, avvenute durante la produzione di versioni sempre più evolute. Ho apprezzato tale metodologia nel gestire situazioni in cui i requisiti fondamentali erano ben noti al contrario di quelli più specifici. Grazie alla creazione di prototipi, ho potuto avere a disposizione una base concreta, per verificare il corretto procedimento di sviluppo e la pianificazione delle versioni successive.

Per sviluppare i vari componenti, ho eseguito ciclicamente le seguenti fasi, fino ad ottenere un risultato che rispettasse gli obiettivi imposti dall'azienda:

- Analisi preliminare: raccolta requisiti e componenti di massima dell'architettura. Identificazione obiettivi principali ed attività necessarie per il loro raggiungimento;
- Analisi e progettazione: analisi e la progettazione in dettaglio raffinando i requisiti e specificando le funzionalità dei componenti.
- Realizzazione e verifica: viene effettuata la codifica e l'integrazione dei nuovi componenti. Vengono effettuati test di unità e regressione;
- Approvazione: confronto con il tutor riguardo la nuova versione prodotta. Accettazione del prodotto o studio delle caratteristiche della versione successiva.

Con tale metodo ho potuto identificare le attività più complesse e concentrare le risorse nell'effettuare analisi ed approfondimenti. In questo modo ho ottenuto un ciclo incrementale che ha portato al raggiungimento degli obbiettivi prefissati rispettando le *baseline* di progetto.
La continua verifica del prototipo ha permesso di ricevere un costante *feedback* da parte dell'azienda ed di implementare le funzionalità più importanti.

3.2 Analisi dei requisiti

L'azienda ha chiesto lo sviluppo di due strumenti principali in differenti versioni: un semplificatore logico algebrico ed un dimostratore di teoremi
Durante lo stage sono state effettuate diverse lezioni riguardanti la logica, l'intelligenza artificiale ed esempi di programmazione funzionale. Da questi incontri ho ricavato la maggior parte dei requisiti, descrivendo problemi e possibili soluzioni allo stato dell'arte.

Ho raccolto le seguenti tipologie di requisiti:

- Requisiti funzionali: descrivono le funzionalità del sistema;
- Requisiti qualitativi: descrivono una caratteristica qualitativa *software* (efficienza, manutenibilità);
- Requisiti di vincolo: requisiti che esprimono vincoli tecnologici e di dominio fissati;

L'azienda ha poi richiesto un ulteriore suddivisione, assegnando priorità differenti alle caratteristiche principali degli strumenti sviluppati:

- Requisiti obbligatori: requisito necessario richiesto dall'azienda;
- Requisiti desiderabile: requisito che porta un evidente valore aggiunto ma non essenziale per il raggiungimento dell'obiettivo;

Al termine dello stage, ho raccolto il seguente numero di requisiti:

Tipologia	Numero
Funzionali	47
Vincolo	6
Qualità	2

Tab 3.1: Tipo e numero requisiti raccolti

Tipologia	Numero
Obbligatori	42
Desiderabili	13

Tab 3.2: Priorità e numero di requisti raccolti

I requisiti sono per la maggior parte funzionali in quanto ogni strumento esegue un insieme di funzioni specifiche per l'elaborazione e l'analisi dei dati. I requisiti desiderabili riguardano il dimostratore di teoremi predicativo.

3.3 Teoria e strumenti implementati

Gli strumenti implementati usano l'albero di esecuzione prodotto dall'analisi statica, per ricavare informazioni sull'integrità dei dati ed il loro accesso alle operazioni

richieste. A tale scopo ho sviluppato degli strumenti che riducono ai minimi termini un espressione logica e derivino o deducono il valore di verità usando regole di inferenza.

L'azienda ha richiesto lo sviluppo dei seguenti applicativi:

- Semplificatore logico algebrico: riceve in *input* una formula logica o aritmetica. Il programma restituisce un'espressione equivalente in formato NNF nel caso dell'espressione logica o semplificata ai minimi termini nel caso di espressione algebrica;
- Sistema a regole di produzione: utilizza specifiche di alto livello espresse tramite regole di produzione, per descrivere regole di semplificazione e derivazione di espressioni logiche.
- Semplificatore logico AI: programma che riceve in *input* un'espressione in logica proposizionale e restituisce un'espressione equivalente in forma NNF utilizzando il sistema di produzione.
- Dimostratore di teoremi per la logica proposizionale: algoritmo che riceve in *input* una formula logica proposizionale e restituisce il corrispettivo valore di verità. Nello specifico il programma deve dimostrare che l'espressione è una tautologia; in caso contrario il programma deve restituire i valori per cui l'espressione è falsa.
- Dimostratore di teoremi per la logica predicativa: simile al precedente ma le espressioni elaborate fanno parte del linguaggio formale predicativo.

L'azienda ha imposto come vincolo che le espressioni trattate siano in notazione polacca. In questo modo si semplificano notevolmente le operazioni di lettura e scrittura dei dati.

3.3.1 Semplificatore logico algebrico

Dato un albero di esecuzione, la lunghezza e complessità della *path condition*, è proporzionale al numero di strutture di controllo ed il numero di condizioni espresse al loro interno. L'azienda ha chiesto un semplificatore in quanto il tempo di elaborazione di un'espressione cresce con l'aumentare del numero e tipologia di operatori da analizzare. È essenziale dunque semplificare tali espressioni trasformando una formula in una equivalente ma ottimizzata per il calcolo e l'analisi.

Tale strumento accetta formule contenenti operatori ed operandi sia logici che algebrici. Nel caso la formula sia di tipo logico, viene portata in NNF (*Negation Normal Form*) e raccolti eventuali operatori comuni strettamente annidati. Una formula logica viene detta in forma normale negativa se gli unici simboli di negazione occorrono davanti alle variabili logiche. Per effettuare tale semplificazione, ho utilizzato le seguenti regole di equivalenza:

- Legge di De Morgan congiuntiva: ¬ (∧ a b) = (∨ ¬a ¬b)
- Legge di De Morgan disgiuntiva: ¬ (∨ a b) = (∧ ¬a ¬b)
- Negazione implicazione: ¬ (→ a b) = (∨ ¬a b)
- Regola di doppia negazione: (¬¬a) = a

Nel seguente esempio, è presente una procedura di semplificazione. Il procedimento elimina il simbolo ¬ che precede ogni formula contenente un operatore, distribuendo la negazione ai componenti al suo interno. Tale operazione viene effettuata ciclicamente fino a raggiungere una forma in cui l'operatore di negazione è presente solo davanti alle variabili.

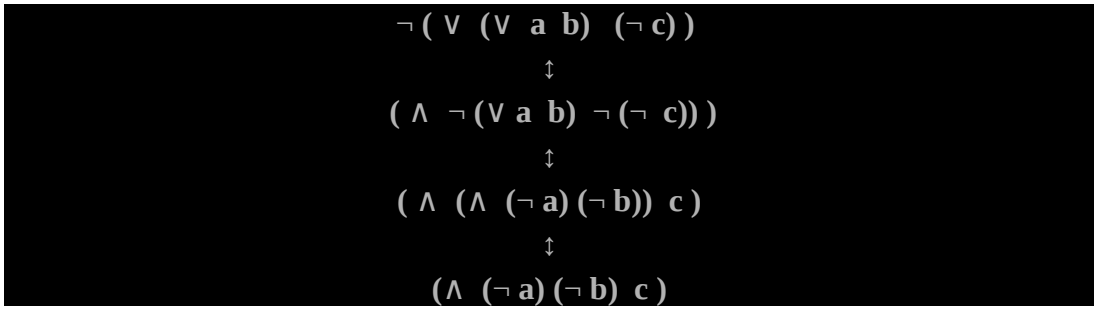

Fig.3.1: Esempio di semplificazione logica in forma NNF

Come è possibile notare, l'espressione ha una complessità minore rispetto all'originale pur mantenendo lo stesso valore di verità. Oltre applicare le leggi di De Morgan di congiunzione disgiunzione, nell'ultima forma, viene raccolto l'operatore AND. In questo modo la valutazione non comprenderà più l'analisi di sei operatori logici, ma sarà ridotta alla valutazione del connettivo di congiunzione applicato a quattro proposizioni.

Nel caso in cui l'espressione sia di tipo algebrico, formata da numeri e monomi, viene semplificata eseguendo le operazioni interne. Ad ogni ciclo, vengono eseguite le operazioni annidate distinguendo la parte numerica da quella letterale. Nel

seguente esempio si noti come vengono svolti i calcoli separando la parte numerica da quella formata da monomi.

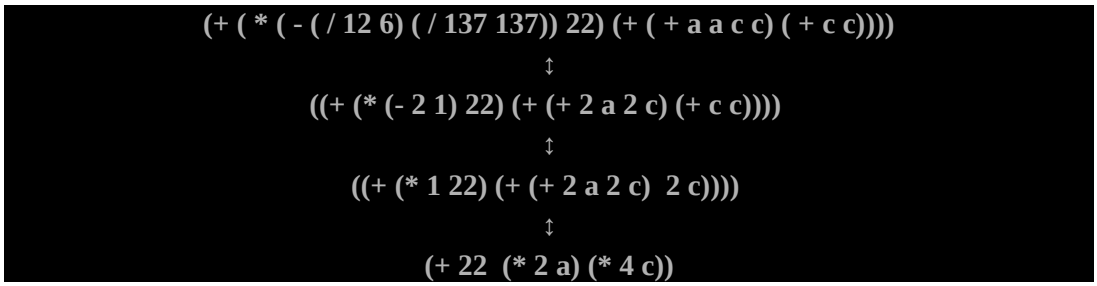

Fig 3.2: Esempio di semplificazione aritmetica

Il semplificatore è stato inoltre una buona esercitazione per apprendere ed applicare i principi della programmazione funzionale. Ho appreso tecniche per lo sviluppo di funzioni correlate, gestione di variabili immutabili e cicli di controllo ricorsivi. Per quest'ultimi sono state utili le esercitazioni sulle le liste elaborate in modo ricorsivo, effettuate durante il corso di Programmazione 1 della triennale.

3.3.2 Production rule system

Tale sistema implementa il concetto di sistema esperto rendendo i successivi algoritmi sviluppati intelligenti. Tradizionalmente un programma è definito in base alla sequenza di istruzioni, che definiscono le capacità dell'algoritmo. Per esempio, nel semplificatore precedente, la conoscenza è implicita nelle istruzioni che descrivono il processo computazionale. Nei sistemi esperti invece l'esperienza non è espressa nel codice ma memorizzata in forma simbolica nella base di conoscenza: una struttura di dati modificabile dall'utente. Con i sistemi esperti, il calcolatore diventa più simile all'uomo perché cerca di imitarlo nel modo in cui seleziona e svolge le operazioni.

I programmi che ho implementato durante lo stage, sono stati sviluppati usando tale componente. Nello specifico, usano un componente intelligente costituito principalmente da due parti: la parte di conoscenza e la parte di calcolo.

Nel progetto la conoscenza è espressa mediante regole di inferenza logica, applicate per semplificare o derivare la verità logica di un enunciato. Il sistema consulta la propria base di conoscenza per identificare il tipo di *input* e restituire l'operazione richiesta. In questo modo le conoscenze di dominio e la loro esecuzione sono separate dall'implementazione del programma. Inoltre è possibile aggiornare il comportamento di un programma senza effettuare modifiche laboriose.

In generale, un sistema esperto è composto da:

- Base di conoscenza (KB): Contiene informazioni riguardo la tipologia di dati e le istruzioni riguardo le operazioni da applicare;
- Motore inferenziale: Contiene funzionalità di analisi dell'*input*, elaborazione dei dati, capacità di lettura ed applicazione delle regole presenti nella KB;

Nel progetto la base di conoscenza è costituita da regole di produzione composte da una premessa (*IF*) e da una parte conseguente (*THEN*). Al verificarsi della condizione presente nel campo *IF* il sistema esegue le operazioni descritte nel campo *THEN*.
Il riconoscimento della condizione *IF* avviene attraverso *pattern matching* prendendo come riferimento l'operatore e gli operandi dell'espressione. Di seguito un esempio di regola di produzione usata per semplificare una formula contenente una doppia negazione. La funzione di *pattern matching* riconosce i pattern presenti nel campo IF della regola 1 e restituisce il solo operando (*?x*) come specificato dal campo *THEN*.

	if	then
1.	(¬ (¬ ?x))	?x

Fig. 3.3: Esempio di regola di produzione presente all'interno della base di conoscenza

I prossimi applicativi sono basati su questo modello. Ogni programma ha una base di conoscenza specifica per il proprio obiettivo e sarà fornito di un motore referenziale adatto al trattato di dati ed operazioni richieste.

3.3.3 Semplificatore logico proposizionale AI

Tale strumento ha funzionalità simili al semplificatore precedente ma presenta sostanziali differenze a livello architetturale ed operativo. In questo caso il programma non usa funzioni che riconoscono l'*input* ed eseguono le relative operazioni richieste, ma usa il sistema di produzione per determinare quando ed in che modo semplificare l'espressione.

Il programma esegue ciclicamente tre operazioni:

1. Lettura e riconoscimento pattern presenti nella formula;
2. Ricerca e selezione della regola corrispondente al pattern trovato;

3. Applicazione delle istruzioni specificate nella regola ricavata.

L'algoritmo termina nel momento in cui l'espressione non è più riducibile.
Lo sviluppo di tale applicazione ha permesso di apprendere il funzionamento del sistema esperto e sfruttare le potenzialità. sfruttare le potenzialità. Tale fatto è dimostrato dalla considerevole diminuzione del numero di componenti rispetto al semplificatore precedente.

3.3.4 Dimostratore di teoremi logica proposizionale

Nel corso del tempo sono state sviluppate diverse tecniche per dimostrare la validità di un enunciato logico. Nel corso di Logica frequentato all'università, ho studiato le tabelle di verità ed il calcolo dei sequenti. Tali procedimenti però, hanno dei difetti a livello di efficienza, in quanto possono portare ad implementare algoritmi con complessità computazionale elevata. Nelle tabelle di verità per esempio, il calcolo del valore di verità ha complessità esponenziale; per un enunciato con n componenti, richiede l'esame di 2^n casi.

L'azienda ha proposto di implementare il dimostratore di teoremi con la tecnica del *tableau* semantico. Con tale metodo, data una formula proposizionale P, si ricerca sistematicamente una refutazione di ¬P applicando opportune regole di derivazione sui connettivi logici. Per dimostrare la che l'enunciato P è una tautologia, si prova a derivare una contraddizione, dall'assunzione che esista una valutazione booleana v:P → {T,F}, rispetto alla quale P risulti falso.

α-rule		
α	**α_1**	**α_2**
$\neg\neg x$	x	-
$x_1 \wedge x_2$	x_1	x_2
$\neg(x_1 \vee x_2)$	$\neg x_1$	$\neg x_2$
$\neg(x_1 \rightarrow x_2)$	x_1	$\neg x_2$

β-rule		
β	**β_1**	**β_2**
$x_1 \vee x_2$	x_1	x_2
$\neg(x_1 \wedge x_2)$	$\neg x_1$	$\neg x_2$
$x_1 \rightarrow x_2$	$\neg x_1$	x_2

$$\frac{\alpha}{\alpha 1 \\ [\alpha 2]}(\alpha - rule) \qquad \frac{\beta}{\beta 1 | \beta 2}(\beta - rule)$$

Fig. 3.4: Regole d'inferenza per il tableau semantico proposizionale

Per eseguire tale processo ho applicato regole simili alle seguenti, basate sulla semantica dei connettivi. Data una preposizione P assunta come vera, contente le formule α e β allora:

- se P è nella forma ¬¬α, allora α è vera;
- se P è nella forma α∧β, allora α e β sono entrambi vere;

- se P è nella forma ¬(α∧β), allora almeno una tra ¬α e ¬β è vera;
- se P è nella forma α→β, allora almeno una tra ¬α e β è vera;
- se P è nella forma ¬(α→β), allora almeno una tra α e ¬β è vera.

Il processo finisce nel momento in cui si ottiene una contraddizione, cioè una formula e la sua negazione sono entrambe vere.

I *tableaux* semantici sono grafi ad albero costituiti da nodi contenenti formule logiche inferite da quelle precedenti; il primo nodo contiene l'enunciato che deve essere confutato.

Le regole di inferenza utilizzate sono elencate in figura 3.4: Le α-regole sono dette regole congiuntive mentre le β-regole sono chiamate regole disgiuntive.

Data un enunciato α composto da un connettivo logico e due formule α_1 e α_2 : l'applicazione di una regola congiunta, aggiunge le formule α_1 e α_2 in fondo a ciascun ramo che contiene α. L'applicazione di una regola disgiuntiva β invece, crea una ramificazione nell'albero: ciascun ramo presente, viene espanso aggiungendo i due nodi β_1 e β_2 come figli dell'ultimo nodo.

Data una proposizione P1 contenuta in una formula logica P, un ramo *r* di un *tableau* è chiuso se contiene le forme complementari P1 e ¬P1. In caso contrario viene definito aperto. Un *tableau* è definito chiuso se tutti i suoi rami sono chiusi, altrimenti viene definito aperto. Un *tableau* si dice completo se ogni nodo contiene solo singole proposizioni, ed eventuali negazioni, senza connettivi logici.

Dato un albero completo costruito partendo da una formula proposizionale P, tale enunciato è:

- Tautologia: se esiste un *tableau* chiuso per P;
- Opinione: se nel *tableau* sono presenti rami aperti;
- Paradosso: se ogni ramo del *tableau* è aperto.

Di seguito in fig. 3.5, è presente un esempio di procedura per la derivazione della formula $(p \vee (q \wedge r)) \rightarrow ((p \vee q) \wedge (p \vee r))$ applicando le regole sopra elencate.

Il primo nodo di ciascun albero contiene la formula negata da valutare. Ad ogni passo viene applicata una regola di inferenza in base all'operatore con priorità maggiore del nodo scelto. Tale procedimento è applicato ciclicamente e per ogni ramo fino a rendere l'albero completo. Per capire come sono state applicate le

regole, a sinistra di ogni nodo è presente un identificativo che indica il numero del nodo; a destra tra parentesi, è indicato il nodo da cui proviene la formula.
Si noti come nel nodo 1, venga applicata una regola congiuntiva e come sono posizionati gli operandi sotto il nodo in esame. Nell'espressione 3 invece, l'albero subisce una ramificazione come previsto dalle regole di disgiunzione. Si noti come i rami vengono poi trattati indipendentemente, sviluppando l'albero fino alla conclusione del processo.
Secondo le definizioni sopra elencate, l'enunciato iniziale è una tautologia, in quanto l'albero è completo ed i nodi interni di ogni ramificazione contengono i valori *p* e *r* con i rispettivi complementari ¬*p* e ¬*r*.

Il metodo dei *tableaux* semantici per la logica proposizionale si conclude sempre dopo un numero finito di passi. Questa caratteristica lo rende particolarmente adatto all'automatizzazione. Ogni regola di espansione infatti genera formule più semplici della formula espansa; cosi facendo, al contrario del metodi dei sequenti, ad ogni passo la formula diminuirà di complessità fino ad ottenere nodi formati da singole proposizioni.

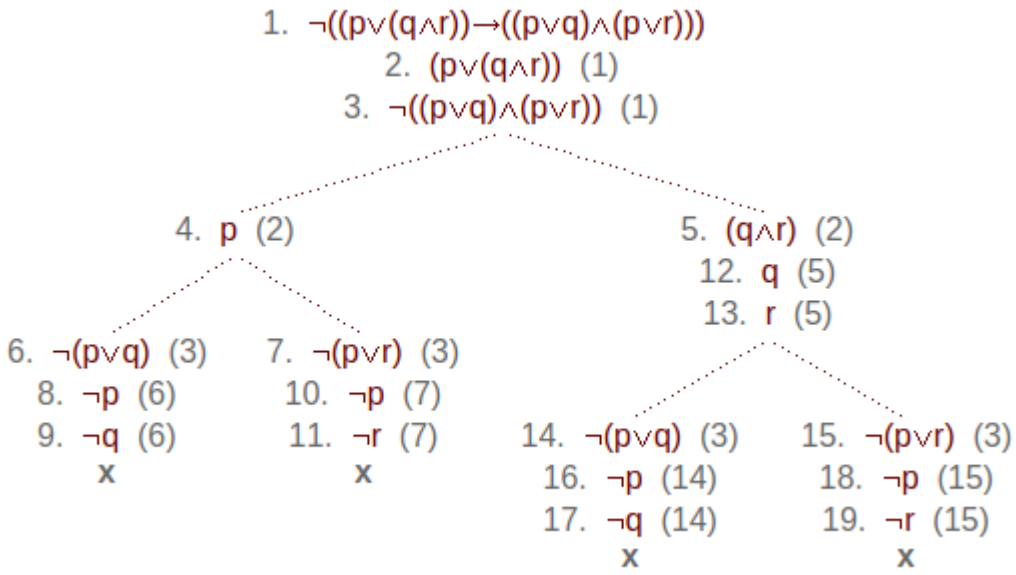

Fig 3.5: Esempio di derivazione mediante tableau semantico per la logica proposizionale
Fonte: http://www.umsu.de/logik/trees/

L'azienda ha voluto implementare questo metodo in quanto, nel caso proporzionale, permette sempre di stabilire la verità di una formula e quindi di terminare. I talbeaux semantici inoltre possono essere facilmente estesi mediante nuove regole,

permettendo così generalizzazioni a teorie più espressive. Tramite l'uso di regole è possibile imporre dei criteri di precedenza sull'ordine in cui vengono eseguite le operazioni, scegliendo quali applicare e sviluppare cosi differenti alberi di derivazione.

3.3.5 Dimostratore di teoremi logica predicativa

Nel linguaggio predicativo, vengono trattate formule composte da predicati, variabili, costanti e quantificatori universale (∀) ed esistenziale (∃). Un termine è un ente che rappresenta variabili e costanti di un determinato dominio. I predicati invece servono a descrivere proprietà dei termini. Il procedimento per la costruzione del *tableau* semantico è simile al precedente in quanto vengono usate le stesse regole per i connettivi della logica proposizionale. La differenza sostanziale sta nel tipo di dati trattati (predicati e termini) e l'aggiunta delle regole di inferenza riguardanti i quantificatori.

α-rule		
α	**α_1**	**α_2**
∀ x A(x)	**∀ x A(x)**	**A(c)**
¬∃ x A(x)	**¬∃ x A(x)**	**¬A(c)**
∃ x A(x)	**A(c)**	-
¬∀x A(x)	**¬A(c)**	

Fig 3.6: Regole d'inferenza per tableau semantico predicativo

L'applicazione della regola congiuntiva contente il quantificazione universale ∀ (e la relativa negazione ¬∃), ha l'effetto di istanziare un nuovo predicato, composto da una costante presente nel linguaggio in esame. Inoltre, rispetto alle restanti regole, la formula derivata viene inclusa nei risultati dell'applicazione. Tale fatto è dovuto alla semantica del quantificatore universale ∀: con il termine *quantificatore* si indica quanto è grande l'estensione in cui è valido un predicato. Per *universale* invece, si intende che tale estensione è sempre totale. È necessario dunque riportare tale formula nel processo di derivazione, in quanto sarà sempre valida.

Di seguito è presente un albero di derivazione per il linguaggio formale predicativo. I nodi che contengono il quantificatore universale non sono riportati nei nodi

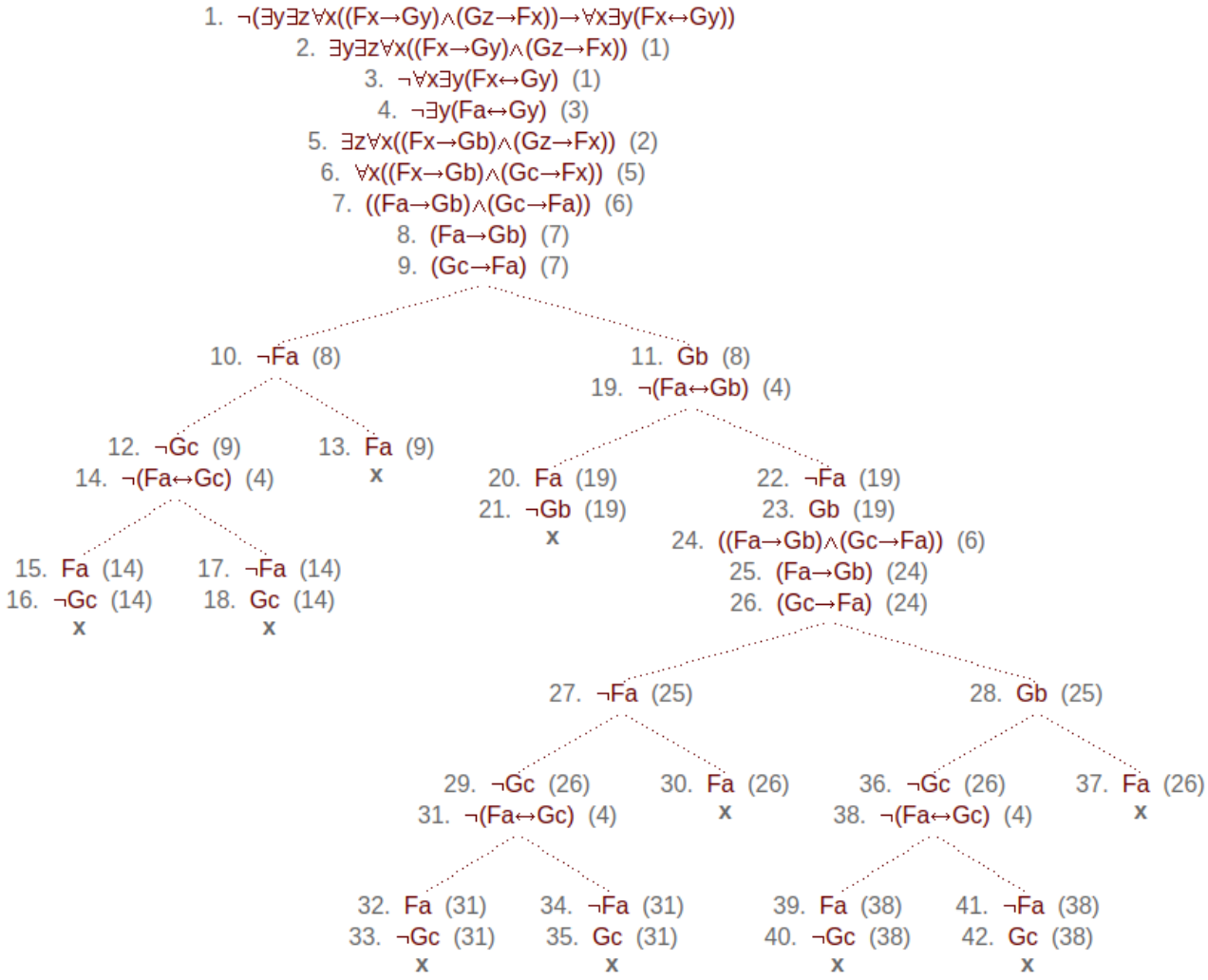

Fig 3.5: Esempio di derivazione mediante tableau semantico per la logica predicativa

Fonte: http://www.umsu.de/logik/trees/

successi ma vengono comunque usati durante l'intero procedimento. Si noti nell'esempio, come l'applicazione delle regole 4 e 6 viene ripetuta in tutte le ramificazioni dell'albero. In questo modo, si cerca di istanziare un predicato che possa chiudere l'albero e terminare il processo.

Grazie ai quantificatori universali, è possibile descrivere insiemi composti da un numero indefinito di elementi. L'introduzione di insiemi infiniti, porta però allo sviluppo di derivazioni che potrebbero non terminare dovendo considerare un numero indefinito di elementi.
La derivazione di una regola con quantificatore universale (e relativa negazione) produce due formule di cui una è la copia dell'originale.

Questo implica che tali regole possono essere sviluppate un numero infinito di volte, eliminando la proprietà di albero completo presente nel *tableau* proposizionale. In presenza di rami aperti, l'algoritmo potrebbe non terminare e sviluppare infiniti predicati senza trovare una coppia complementare.

Tale interpretazione rende necessari diversi cambiamenti all'algoritmo. Il programma deve possedere funzionalità per poter scegliere quali e per quante volte applicare una regola con quantificatore. In questo modo si cerca di sviluppare regole che possano restituire elementi complementari per il ramo in esame. Nel caso non si riesca a chiudere l'albero, si dovrà controllare di non entrare in cicli infiniti ed arrestare lo sviluppo dell'albero.

3.3.6 Lisp

Come linguaggio di programmazione l'azienda ha proposto di usare Lisp (ListProcessor). Ideato nel 1958 da John McCarthy, è uno dei primi linguaggi ad aderire ai principi della programmazione funzionale. Tale linguaggio ideato al MIT principalmente per applicazioni riguardanti l'intelligenza artificiale, è adatto alla gestione e manipolazione di espressioni simboliche. Nel corso degli anni sono stati sviluppati differenti dialetti rispetto al linguaggio originale. La versione più diffusa è il Common Lisp di cui esistono svariate implementazioni libere.

*Fig 3.8:*Alien Lisp Mascot
http://www.lisperati.com/logo.html

LISP si basa su poche ma chiare regole sintattiche in cui la principale è il concetto di lista. Oltre ad essere adatto per il trattamento di tali strutture dati, lo stesso codice di LISP è espresso sotto forma di lista. In questo modo è facile scrivere codice che a sua volta generi o modifichi codice LISP.

Il codice prodotto può essere sia interpretato che compilato. Questa duplice possibilità di esecuzione permette di testare rapidamente porzioni di codice senza dover compilare ed eseguire l'intero programma.

LISP utilizza la notazione prefissa valutando le espressioni da sinistra a destra. Un'

espressione è anch'essa una lista, cioè una sequenza di simboli separati da spazi e delimitata da parentesi tonde:

*(**command** op1 op2 opn)*

Nonostante sia un linguaggio datato, in LISP sono presenti caratteristiche che altri linguaggi di programmazione moderni hanno adottato. Tra queste si ricorda:

- *Dynamic redefinition*: possibilità di modifica del codice senza interrompere l'esecuzione;
- *Dynamic Typing*: le variabili non richiedono dichiarazione ne definizione di tipo;
- *Garbage Collection*: gestione automatica dell'allocazione e liberazione della memoria.

Differenti programmi sono stati scritti in LISP. Il più famoso è certamente EMACS, il primo *software* libero prodotto da Richard Stallman.

3.3.7 Machine Learning: algoritmi di classificazione

Per effettuare l'analisi ho usato il programma Orange ed un set di dati fornito dall'azienda. Tale *software* offre diversi strumenti per analisi ed elaborazione di dati tramite algoritmi di *data mining* e *machine learning*. Il programma inoltre fornisce numerose funzionalità per produrre grafici e statistiche riguardo i risultati prodotti e l'efficienza ed efficacia degli algoritmi predittivi

Orange è rilasciato sotto licenza GPL, sviluppato inizialmente in C++ come *framework* per programmi di *machine learning*. Successivamente sono stati implementati funzionalità sia in Python che C++ aggiungendo un'interfaccia grafica ed un set di strumenti sempre più ampi ed innovativi. Il programma è usato in campi di biomedicina, bioinformatica, ricerca genomica ed istruzione. In quest'ultimo è molto apprezzato per l'insegnamento degli algoritmi e test di nuove tecniche di analisi.

È un linguaggio di programmazione visuale che consente la programmazione tramite manipolazione grafica degli elementi. Il flusso di elaborazione avviene utilizzando *widget* che rappresentano componenti computazionali ed eseguono la maggior parte del lavoro, compreso lo scambio di informazioni (fig 3.9). La transizione dell'informazione avviene passando l'*output* delle rispettive funzionalità; in questo modo è possibile creare flussi di elaborazioni semplici ed intuitivi. Sono presenti

componenti per la lettura dati, elaborazioni pre-processo, creazione grafici, algoritmi di *clustering* e molto altro.

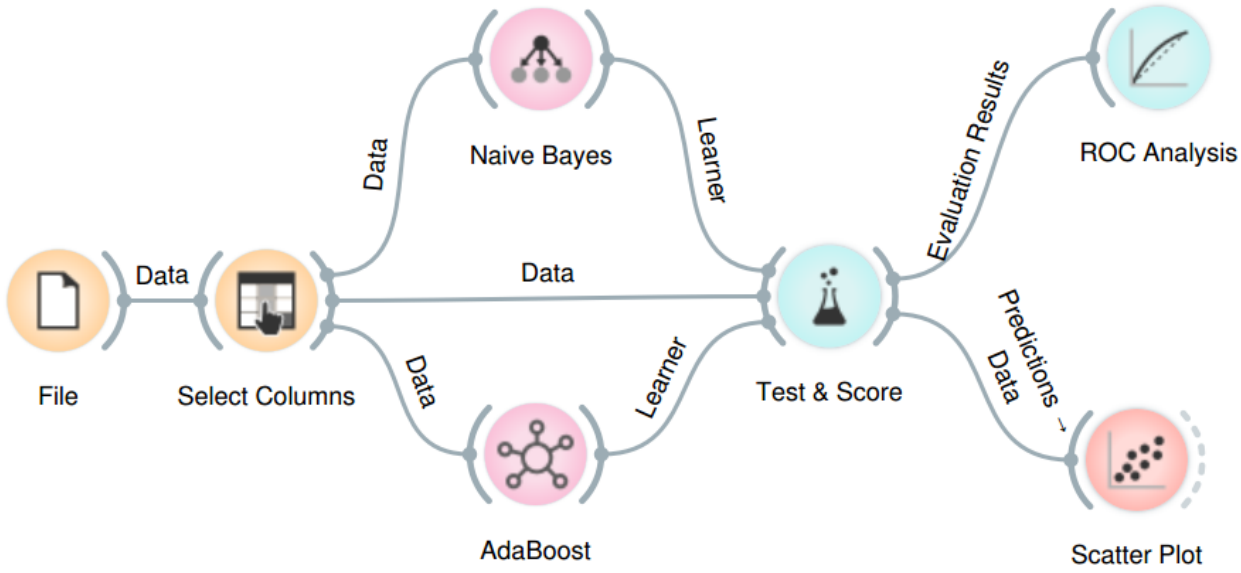

Fig. 3.9: Esempio flusso elaborazione in Orange.

Fonte: Programma Orange.

L'obiettivo dell'azienda, era di capire le funzionalità e potenzialità degli algoritmi di classificazione supervisionati presenti in Orage.

Con classificazione, si intende il problema di identificare la classe di appartenenza di un'entità, sulla base di una conoscenza composta da esempi d'apprendimento. Dal punto di vista matematico un classificatore è una funzione che mappa un elemento di un dominio, su un insieme composto da classi con proprietà definite. Con il termine supervisionato, si indica la capacità di tali algoritmi di ricavare le caratteristiche dei diversi elementi presenti nel *training set*, e suddividerli in categorie.

Orange fornisce diversi tipi di classificatori. Di seguito una breve descrizione a livello generale di come operano gli algoritmi che ho utilizzato per effettuare l'analisi:

- Classificatore Bayesiano: basato sull'applicazione del teorema di Bayes e della probabilità condizionata. Basa le predizioni, calcolando la probabilità di un evento A sapendo che un evento B è verificato.

- *K-nearest neighbor (k-NN)*: Tale algoritmo è un utilizzato per riconoscere la classe d'appartenenza basandosi sulle caratteristiche degli oggetti vicini a quello considerato.
- Albero di decisione: suddivide lo spazio di elementi in aree con la stessa classe. Viene formato un albero in cui le foglie rappresentano le classi e le ramificazioni l'insieme delle proprietà che portano a quelle scelte.
- *Random forest:* si compone da molti alberi di decisione e restituisce la risposta in base alla predizione prodotta dagli alberi interni.
- *Support Vector Machine (SVM):* tale metodo costruisce un iperpiano di separazione che massimizza la distanza tra gli elementi di classi diverse. In questo modo è possibile definire un modello che separa gli elementi di due classi. La sua applicazione viene apprezzata in analisi in cui i dati sono ricchi di caratteristiche.
- *Adaboost*: metodo di classificazione che si basa sulla combinazione di più classificatori di base.

Per addestrare gli algoritmi, ho utilizzato un *training set* contenente informazioni ricavate da statistiche aziendali. Ogni voce è composta dalle seguenti caratteristiche:

- *Category*: Tipologia di errore:*grapich*, *security*, *correctnes*, performance;
- *Rank*: valore compreso tra 1 e 20 che indica la gravità dell'errore; 1 rappresenta il valore di gravità più elevato;
- Data: data della modifica;
- *Author*: nome utente del programmatore che ha effettuato la modifica.

Per la valutazione dei classificatori, l'azienda ha richiesto come obiettivo di prevedere il nome del programmatore in base alle caratteristiche di una correzione di errore avvenuta nelle *repository*.

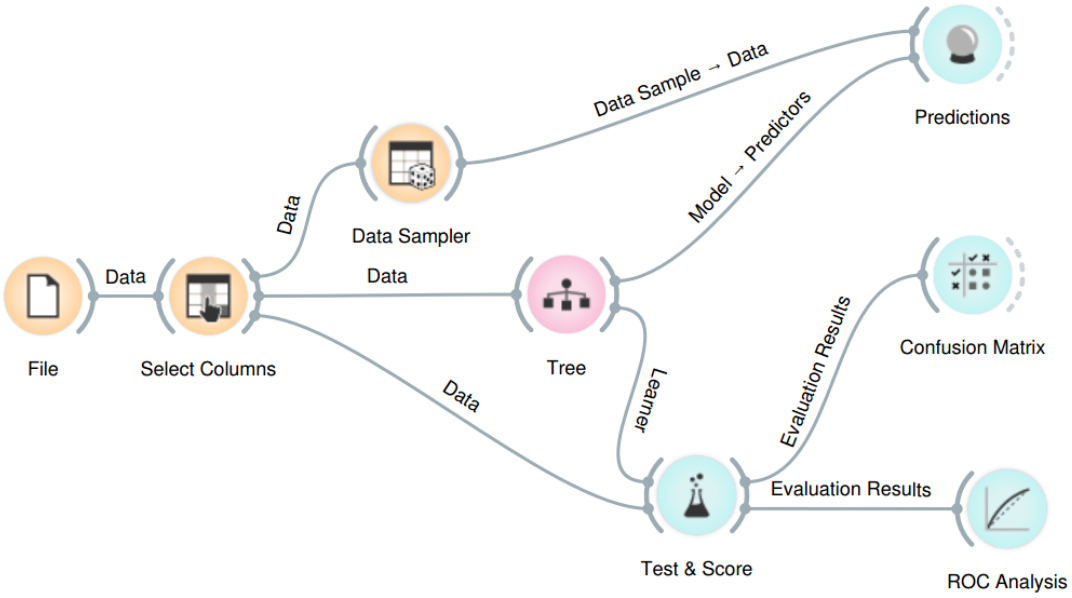

Fig 3.10: Esempio di analisi di un classificatore.
Fonte: Programma Orange

Ho eseguito tale analisi componendo simili flussi di elaborazione (fig. 3.10):

- *File*: i dati sono estratti dal *training set* mediante il *widget* denominato *file;*
- *Select columns:* Il risultato è raccolto e filtrato in base alle caratteristiche di valutazione e quelle da predire;
- *DataSampler:* suddivide i dati presenti all'interno del *training set*: il 70% è usato per istruire l'algoritmo, i restanti 30% per valutare le predizioni dei classificatori.
- *Tree*: classificatore usato per effettuare l'analisi (albero di decisione)

I risultati prodotti dal classificatore vengono successivamente valutati dai *widget* riportati in figura 3.10 con sfondo azzurro e che saranno descritti nel capitolo 3.5.2.

Sono d'interesse principale due tipi di informazioni: *precision che* indica la misura di esattezza e la *recall* che mostra la completezza di tale risultato.
Tali parametri sono fondamentali nello studio degli algoritmi di *machine learning*, in quanto definiscono il numero di termini vero positivo, vero negativo, falso positivo e falso negativo usati per valutare la classificazione di un oggetto.

3.4 Progettazione

Di seguito è descritta l'architettura del *software* che ho implementato. Ho effettuato la progettazione di ogni strumento alla fine dell'analisi dei requisiti per ogni ciclo di sviluppo del programma.

Per ogni applicativo ho stilato un elenco di attività che il programma doveva eseguire per raggiungere l'obiettivo. Come i processi *software*, ho diviso ogni attività in compiti eseguibili da un singola funzione, nel rispetto del principio di singola responsabilità. .

Tale suddivisione in compiti ha semplificato lo sviluppo di un architettura che aderisse ai principi della programmazione funzionale. Un programma scritto con tale paradigma infatti, prevede l'esecuzione di una serie funzioni principali le quali a loro volta chiamano funzioni ausiliare per eseguire compiti specifici. Una volta identificate le attività principali ed i relativi compiti, ho potuto costruzione di un set di funzioni in modo naturale ed intuitivo. Cosi facendo inoltre, ho ottenuto un'architettura coesa e modulare, aperta alla modifica e che offre numerosi vantaggi in fase di codifica e di test.

3.4.1 Architettura software

Nella progettazione degli strumenti implementati, l'unico *design pattern* che ho utilizzato è stato il *facade*. L'inutilizzo dei *design pattern* è dovuto al differente stile di progettazione dei programmi funzionali. I design-pattern esistenti e studiati nei corso di studi universitari, offrono soluzioni comuni per problemi riguardanti la scomposizione di un sistema in oggetti e la gestione delle relative dipendenze. Utilizzando i paradigmi funzionali, cioè non oggetti ma funzioni e variabili immutabili, a parità di problemi, le soluzioni adottate sono completamente differenti.

Ogni programma è formato da set di funzioni raggruppati in un componente. Ognuno contiene la funzione d'avvio dove è possibile inserire l'*input* ed avviare l'elaborazione.

Sono presenti i seguenti componenti:

- ALEsimplifier (Aritmetich Logic Expression): contiene le funzioni per implementare il semplificatore logico algebrico;
- Production_rule_system: sistema che implementa il sistema esperto. Nello specifico sono presenti le base di conoscenza di ogni algoritmo ed un motore inferenziale per fornire l'*output* richiesto;

- AI_Logic_Simplifier: contiene il componente che utilizza la conoscenza del sistema esperto, per semplificare una formula in logica proposizionale;
- Propositional_*tableaux*: funzioni che implementano il dimostratore di teoremi per la logica proporzionale mediante *tableaux* semantico;
- Predicate_*tableaux*: funzioni che implementano il dimostratore di teoremi per la logica predicativa mediante *tableaux* semantico

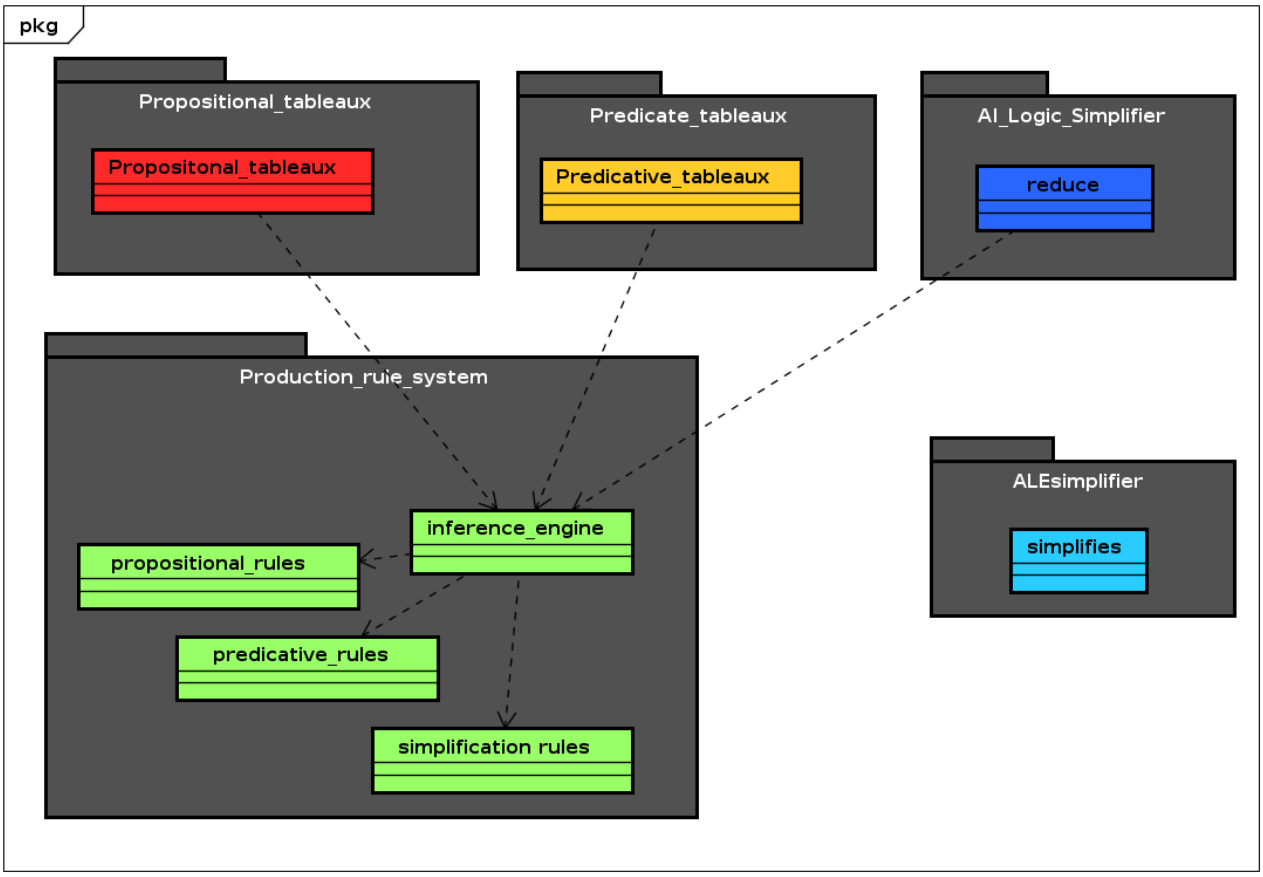

Fig. 3.11: Architettura dei componenti implementati

3.4.2 Caratteristiche architettura software

A fine progettazione sono stati raccolte le seguenti metriche che indicano la buona riuscita dell'architettura

La maggior parte delle metriche elencate, riguardano la fase di codifica di un progetto. Ho riportato comunque tali informazioni in quanto, senza una buona progettazione architetturale, risulta difficile ottenere codifiche che rispettano determinati livelli di accettazione. Inoltre, in una architettura basata su paradigmi

funzionali, tali metriche assumono un significato diverso da un programma orientato agli oggetti. In questo modo ogni funzione è valutata sia nella composizione interna che nel modo in cui collabora con l'ambiente esterno. Nei seguenti dati è possibile ricavare le qualità dell'architettura quali la semplicità, *information hiding*, coesione e manutenibilità.

3.4.2.1 Fan-in e fan-out

Il *fan-in* di un singolo modulo *software* indica quanti altri moduli lo utilizzano durante la loro esecuzione. Consiste in un indice numerico che cresce ogni qualvolta viene individuato un modulo che durante la sua esecuzione utilizza il modulo in questione. Permette quindi di stabilire il livello di riuso implementato.

Fan-In	
Livello ottimale	> 1
Livello accettazione	> 2

Tab 3.3 : Livello ottimale e di accettazione Fan-in

Il *fan-out* di un singolo modulo *software* indica quanti altri moduli vengono utilizzati durante la sua esecuzione. Consiste in un indice numerico che incrementa ogni qualvolta viene individuato un modulo che viene utilizzato durante l'esecuzione del modulo in questione. Permette quindi di stabilire il livello di accoppiamento implementato.

Fan-out	
Intervallo ottimale	0-1
Intervallo accettazione	0-5

Tab 3.4 : Intervallo ottimale e di accettazione Fan-out

Ho raccolto i seguenti dati per i componenti sviluppati. Si noti come ogni modulo abbia un valore di *fan-in* pari a 1. Questo dato indica fedelmente come i paradigmi funzionali sviluppano l'elaborazione. Ogni funzione è invocata da un sola entità che a sua volta ne richiama altre per eseguire il compito. Al termine dell'ultima funzione, restituirà il risultato che provocherà un effetto a catena nelle restanti funzioni attive. Il basso livello di accoppiamento è dovuto all'alta coesione presente tra i moduli sviluppati.

Componente	Fan-In(media)	Fan-Out(media)
Semplificatore logico algebrico	1	2
Sistema a regole di produzione	1	2
Semplificatore logico AI	1	1
Dimostratore di teoremi per la logica proposizionale	1	2
Dimostratore di teoremi per la logica predicativa	1	2

Tab 3.5 : Valori in media di *fan-in* e *fan-out* ricavati dalle funzioni dei rispettivi componenti

3.4.2.2 Numero di parametri per metodo

Metrica che indica il numero di parametri formali di una funzione. Un alto numero di parametri comporta una maggior facilità nel commettere errori durante lo sviluppo e riempie velocemente la memoria in caso di chiamate ricorsive della funzione. Quest'ultimo aspetto è fondamentale, in quanto in ogni programma implementato, la ricorsione è l'unica struttura di controllo presente.

Numero di parametri per metodo	
Livello ottimale	0-5
Livello accettazione	0-3

Tab 3.6 : Intervallo ottimale e di accettazione di parametri per metodo

Tale valore è importante, in quanto i parametri sono le uniche variabili a disposizione e permettono lo scambio di informazione tra le funzioni. Grazie ad un basso numero di parametri si ottiene un'architettura comprensibile formata da componenti ben descritti dall'*input* e dai valori restituiti. L'immutabilità dei parametri fornisce altre caratteristiche all'architettura come l'affidabilità e la robustezza.

Componente	Media numero Parametri
Semplificatore logico algebrico	1
Sistema a regole di produzione	2
Semplificatore logico AI	1

Dimostratore di teoremi per la logica proposizionale	2
Dimostratore di teoremi per la logica predicativa	2

Tab 3.7 : Componente e corrispettivo valore medio di parametri delle funzioni presenti al suo interno

3.4.2.3 Complessità ciclomatica

La complessità ciclomatica è una metrica che indica il numero di cammini linearmente indipendenti che attraversano il grafo di controllo di flusso. In questo grafo i nodi rappresentano gruppi di istruzioni atomici, mentre gli archi rappresentano gruppi di istruzioni eseguibili consecutivamente. Questo indice fornisce un limite superiore sul numero di test da effettuare per garantire un *test* coverage completo.

Complessità ciclomatica	
Livello ottimale	0 – 9
Livello accettazione	0 – 5

Tab 3.8 : Intervallo ottimale e di accettazione di complessità ciclomatica

Tale metrica indica quanto una funzione è complessa e dunque quanto sia adatta ad operazioni di manutenzione ed estensione. Nell'architettura progettata, ogni funzione è composta mediamente da 3 flussi. Tale aspetto rende l'architettura adatta alla verifica e all'individuazione di malfunzionamenti mediante test di unità e regressione. Anche in questo caso, l'utilizzo di componenti modulari, ha permesso di ottenere un livello basso di complessità.

Componente	**Numero Medio Cicli**
Semplificatore logico algebrico	4
Sistema a regole di produzione	3
Semplificatore logico AI	3
Dimostratore di teoremi per la logica proposizionale	4
Dimostratore di teoremi per la logica predicativa	5

Tab 3.9: Media del numero di cammini linearmente indipendenti di ogni funzione suddivisi per componente

3.4.2.4 Linee di codice

Metrica che indica la dimensione di una funzione basandosi sul numero di linee del codice sorgente. Rappresenta il livello di comprensibilità dei componenti implementati e quanto rispettino il principio di singola responsabilità.

Linee di codice	
Livello ottimale	< 10
Livello accettazione	< 15

Tab 3.10 : Livello ottimale e di accettazione di linee di codice

Grazie al linguaggio Lisp, che favorisce un stile basato su funzioni semplici ed essenziali, ho ottenuto un numero molto basso di istruzioni per metodo.

Componente	**Linee di codice**
Semplificatore logico algebrico	6
Sistema a regole di produzione	5
Semplificatore logico AI	4
Dimostratore di teoremi per la logica proposizionale	5
Dimostratore di teoremi per la logica predicativa	7

Tab 3.11: Numero medio di linee di codice che compongono le funzioni

3.5 Verifica

3.5.1 Analisi dinamica

3.5.1.1 Test di unità

Ho sottoposto ogni componente dell'architettura a singoli e specifici test per verificare il corretto funzionamento. Per ogni unità ho definito differenti insiemi di dati in modo da poter eseguire e verificare tutte i possibili flussi di elaborazione. Grazie alla struttura semplice ed essenziale dei programmi scritti con linguaggi funzionali, tale controllo è stato semplificato potendo studiare una funzione in ogni suo comportamento. In questo modo i valori di *stantment coverage* e *branch coverage* dei test effettuati sono spesso pari al 100%.

Test di unità		
Applicativo	**Test effettuati**	**Test passati**
Semplificatore logico algebrico	16	16
Sistema a regole di produzione	6	6
Semplificatore logico AI	3	3
Dimostratore di teoremi per la logica proposizionale	11	11
Dimostratore di teoremi per la logica predicativa	13	13

Tab 3.12: Componenti e relativo numero di test di unità

3.5.1.2 Test d'integrazione

I test d'integrazione verificano il corretto funzionamento di componenti correlati tra loro. Tale verifica è stata eseguita in modalità *bottom-up.* Nello specifico, ho testato ogni nuovo set di funzioni, assemblando ciascun componente uno alla volta con quelli già testati in precedenza. In questo modo ho potuto individuare precisamente la presenza di eventuali errori presenti nelle funzioni da integrare. Ho eseguito tale procedimento fino a assemblare tutte le componenti del programma.

Test d'integrazione		
Componente	**Test effettuati**	**Test passati**
Semplificatore logico algebrico	17	17
Sistema a regole di produzione	7	7
Semplificatore logico AI	4	4
Dimostratore di teoremi per la logica proposizionale	12	12
Dimostratore di teoremi per la logica predicativa	14	14

Tab 3.13: Componenti e relativo numero di test d'integrazione

3.5.1.3 Test di regressione

Per ogni strumento implementato, ho sviluppato una batteria di test in modo da poter verificare il corretto comportamento dei componenti nell'insieme e per verificare la presenza di errori dovuti a modifiche del codice.

Test di regressione		
Componente	**Test effettuati**	**Test passati**
Semplificatore logico algebrico	24	24
Sistema a regole di produzione	10	10
Semplificatore logico AI	29	29
Dimostratore di teoremi per la logica proposizionale	14	14
Dimostratore di teoremi per la logica predicativa	15	15

Tab 3.14: Componenti e relativo numero di test di regressione

3.5.1.4 Test di sistema

Tali test verificano il corretto funzionamento dei programmi in base ai requisiti stabiliti in fase di analisi.

Test di sistema		
Applicativo	**Numero requisiti**	**Requisiti Soddisfatti**
Semplificatore logico algebrico	12	12
Sistema a regole di produzione	6	6
Semplificatore logico AI	12	12
Dimostratore di teoremi per la logica proposizionale	9	9
Dimostratore di teoremi per la logica predicativa	12	12

Tab 3.15: Componenti e numero di test di sistema

3.5.2 Machine learning

Di seguito sono presentati alcuni risultati dei test riguardanti i classificatori. La valutazione di un modello, è una delle fasi principali nel processo di analisi dei dati.

Oltre all'accuratezza dell'algoritmo è necessario considerare ulteriori metriche in modo da ottenere una visione completa della valutazione. In uno scenario di classificazione di un elemento, l'elaborazione ha solo due risultati possibili: vero o falso. In tali test vengono raccolti i seguenti risultati:

- Veri positivi (VP): istanze positive, stimate correttamente
- Veri negativi (VN): istanze negative, stimate correttamente
- Falsi positivi (FP): istanze positive stimate in modo negativo
- Falsi negative (FN): istanze negative stimate positivamente

Di seguito sono descritti i *widget* e le rispettive funzionalità, usati per valutare le caratteristiche dell'elaborato.

3.5.2.1 Prediction

L'accuratezza è la percentuale delle istanze classificate correttamente. In genere è la prima metrica che viene osservata quando si valuta un classificatore. Questo *widget* presenta i risultati di predizione (fig. 3.12). L'algoritmo elenca il 30% dei dati del trainig set come *input* e li confronta con la predizione dell'algoritmo.

	Naive Bayes	Tree	kNN	kNN	Random Forest	SVM	AdaBoost	author	rank	category	date
277	zucale	zucale	zucale	zucale	zucale	zucale	zucale	zucale	20.000	PERFORMANCE	2007-09-14
278	biaand	perfil	broand	perfil	perfil	biaand	perfil	perfil	20.000	PERFORMANCE	2015-01-27
279	broand	perfil	broand	perfil	monmir	biaand	perfil	perfil	15.000	SECURITY	2015-01-27
280	fiaole	avefed	avefed	avefed	olienr	perfil	avefed	avefed	20.000	PERFORMANCE	2010-01-22
281	broand	avefed	avefed	avefed	olienr	broand	avefed	avefed	15.000	SECURITY	2010-01-22
282	broand	avefed	avefed	avefed	avefed	chigia	avefed	avefed	11.000	CORRECTNESS	2010-01-22
283	perfil	avefed	avefed	avefed	broand	perfil	avefed	avefed	18.000	PERFORMANCE	2010-01-22
284	fiaole	avefed	avefed	avefed	avefed	fiaole	avefed	avefed	19.000	STYLE	2010-01-22
285	fiaole	broand	broand	broand	perfil	broand	perfil	perfil	19.000	STYLE	2010-05-04

Fig. 3.12: Confronto tra il risultato di un classificatore (sx) e l'effettivo valore (dx)
Fonte: Programma Orange

A sinistra ogni colonna contiene il risultato predetto dei classificatori usati nel processo. Si noti come gli algoritmi hanno prodotto predizioni diverse in differenti casi. Tale fatto spiega il motivo della varietà di classificatori a disposizione, in quanto ognuno è specializzato in base al tipo di predizione e le caratteristiche dei dati.
La colonna "*author*" rappresenta l'elemento da predire e indica la risposta corretta. A destra sono presenti i dati contenuti nel *traning set.* Tali dati specificano il tipo di correzione effettuata (*category*), la gravità dell'errore corretto (*rank*) e la data che avvenuta la correzione (*date*).

3.5.2.2 Test & Score

Tale wiget testa gli algoritmi direttamente dal *training set*. Offre diverse possibilità di suddivisione e selezione dei dati di addestramento. La scelta varia in base alla varianza dei dati ed alla precisione della predizione. Vengono restituiti diversi risultati tra cui il valore di *precision* e *racall*.
Un valore di *precision* pari a 1.0 indica che ogni oggetto che è stato etichettato come appartenente ad una classe C, vi appartiene davvero. Un valore di *recall* pari ad 1.0 significa che ogni oggetto di classe C è stato etichettato come appartenente ad essa

Method	Precision	Recall
AdaBoost	0.839	0.894
Naive Bayes	0.000	0.000
kNN	0.769	0.863
Tree	0.837	0.901
Random Forest	0.759	0.802

Fig. 3.13: Valori forniti dal widget Test&Score
Fonte: Programma Orange

3.5.2.3 Confusion matrix

Ogni colonna della matrice rappresenta i valori predetti, mentre ogni riga rappresenta i valori reali. Attraverso questa matrice è osservabile se vi è "confusione" nella classificazione di diverse classi. Nello specifico vengono visualizzati i valori di VP/FP e FN/VN.

	ansmic	attani	avefed	bardom	basluc	biaand	broand	chigia
ansmic	5	0	0	0	0	4	1	0
attani	0	89	0	1	0	5	9	0
avefed	0	0	26	0	0	0	12	0
bardom	0	1	0	71	0	14	31	0
basluc	0	0	0	0	192	0	0	27
biaand	1	5	0	10	0	606	43	0
broand	0	10	4	38	0	37	1496	0
chigia	0	0	0	0	10	0	0	252

Fig. 3.14 Esempio di Confusion matrix
Fonte: Orange

3.5.2.4 ROC analysis

Il *widget Roc Analysis* visualizza il grafico contenente la curva ROC. Tale curva rappresenta il rapporto tra falsi-positivi e veri-positivi rilevati dai classificatori al variare dell'elemento predetto. Più la curva segue il bordo sinistro e poi quello superiore, maggiore è il livello di accuratezza. In fig. 3.15, le differenti linee tracciate nel grafico rappresentano le curve espresse dai classificatori.

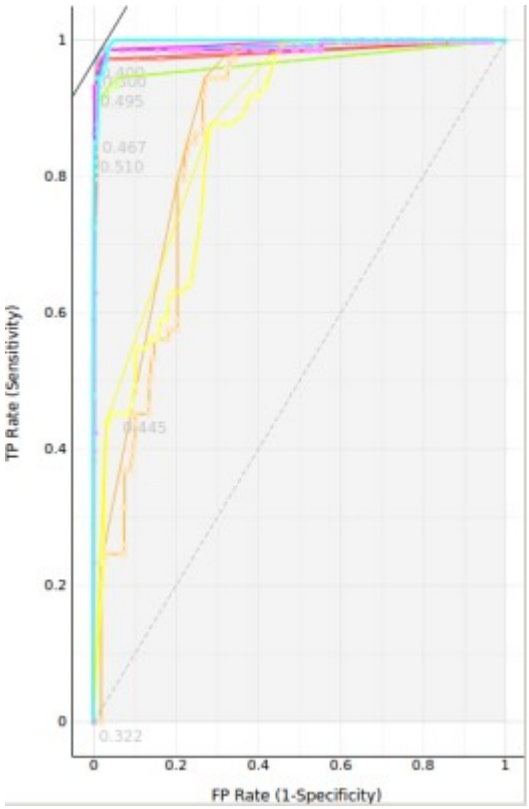

Fig. 3.15 Grafico prodotto dal widget Roc Analysis
Fonte: Programma Orage

4 Conclusioni

Nella stesura del piano di lavoro concordato con il tutor aziendale abbiamo fissato gli obiettivi minimi ed opzionali degli applicativi. Riassumendo brevemente, gli obiettivi principali prevedevano:

- lo sviluppo di un semplificatore logico-algebrico;
- lo sviluppo di un sistema esperto a regole di produzione;
- lo sviluppo di un semplificatore logico-algebrico con metodi di IA;
- lo sviluppo di un dimostratore di teoremi per la logica proporzionale.

In sede di valutazione, è importante ricordare anche i vincoli imposti dall'azienda nello sviluppo del progetto:

- Sviluppo di algoritmi che utilizzassero metodi di intelligenza artificiale;
- Sviluppo di *software* che aderissero ai principi della programmazione funzionale.

In linea generale, sulla base dei requisiti e dei vincoli e delle aspettative fissate all'inizio dello stage, ritengo di aver raggiunto gli obiettivi fissati portando a termine un lavoro completo e soddisfacente.

Anche il tutor aziendale ha espresso una grande soddisfazione per il lavoro svolto e i risultati raggiunti, nonostante un'iniziale incertezza data dal peso sperimentale del progetto. In particolare, a preoccupare il tutor era la complessità delle nozioni teoriche da apprendere e la successiva difficoltà di applicazione dei principi di programmazione funzionale.

4.1 Obiettivi

Come già evidenziato, in linea di massima gli obiettivi prefissati, sia quelli dell'azienda che i miei, sono stati raggiunti.

4.1.1 Il programma

4.1.1.1 Dimostratore di teoremi proposizionale

Il Dimostratore di Teoremi Proposizionali è lo strumento più completo tra quelli sviluppati. Il programma fornisce il risultato corretto per qualsiasi proposizione logica, fornendo risultati specifici in caso di opinione.

Tra i vari test che ho effettuato, uno dei più significativi prevedeva l'utilizzo delle espressioni logiche presenti negli appelli d'esame di logica dell'Università. Il programma forniva sempre la soluzione corretta e a dimostrazione la bontà del procedimento, il programma inoltre stampava a video le regole applicate durante il processo di inferenza.

In *fig. 4.1* è riportato un esempio di esecuzione del dimostratore di teoremi per la logica proposizionale, che ci offre una chiara visione del comportamento dell'algoritmo.

Nella prima riga è riportata la funzione d'avvio con parametro l'espressione logica da valutare. A seguire, vengono elencate tutte le regole di inferenza applicate e, infine, il risultato, nel caso specifico un'*opinione*.

```
[2]>  (tableau '(and  (->  (and (not (-> s p)) (-> n (not p))) (not (and (and (not p) (not s)) (not n))) ) (-> (->
(or s p) n) (and (and (not p) (not s)) (not n)) )))

RUNNING>>

RULE- "NOT ( A AND B)"
RULE- "(NOT ( A1 -> A2))"
RULE- "(NOT ( A1 -> A2))"
RULE- "(A1 & A2)"
RULE- "(NOT ( A -> B))"
RULE- "(NOT (NOT A))"
RULE- "NOT ( A AND B)"
RULE- "(NOT ( A1 -> A2))"
RULE- "(NOT (A1 OR A2))"
RULE- "NOT ( A AND B)"
RULE- "(NOT (NOT A))"
RULE- "(NOT ( A -> B))"
RULE- "NOT ( A AND B)"
RULE- "(NOT (NOT A))"
RULE- "(NOT (NOT A))"
RULE- "(NOT (NOT A))"
RULE- "(A1 & A2)"
RULE- "(NOT (NOT A))"
RULE- "(NOT (NOT A))"
RULE- "(A1 & A2)"
RULE- "(A1 & A2)"

|---------------------------- O P I N I O N E ----------------------------|
|espressione falsa per:|

(S P =0 & N =1)
(N P =1)
(N S =1)
(N =1)


>>COMPLETE
NIL
[3]>
```

Fig 3.15: Esempio esecuzione del dimostratore di teoremi implementato

A dimostrazione della correttezza del procedimento e del risultato, in figura 4.2 è esposta la tabella di verità.

Truth table:

n	p	s	$(\neg (s \Rightarrow p) \wedge (n \Rightarrow \neg p) \Rightarrow \neg (\neg p \wedge \neg s \wedge \neg n)) \wedge ((s \vee p \Rightarrow n) \Rightarrow \neg p \wedge \neg s \wedge \neg n)$
T	T	T	F
T	T	F	F
T	F	T	F
T	F	F	F
F	T	T	T
F	T	F	T
F	F	T	T
F	F	F	T

Fig 4.2: Tabella di verità per l'espressione analizzata in fig.4.1

4.1.1.2 Dimostratore di teoremi predicativi

Nello sviluppo del dimostratore di teoremi predicativi i requisiti da soddisfare, al contrario dei precedenti, erano tutti opzionali.

Il programma che ho ottenuto, tuttavia, non è completo, anche se fornisce una solida base di partenza per ulteriori sviluppi futuri.

Non ho potuto concludere il lavoro per due ordini di motivi:

- da una parte la durata limitata dello stage che non ha permesso di approfondire in modo adeguato il problema: la maggior parte del tempo che ho dedicato a questo progetto, è stato impiegato per scrivere le procedure di manipolazione di operandi composti da predicati, variabili e costanti;
- dall'altra ho riscontrato delle difficoltà nell'implementazione del modulo che avrebbe dovuto selezionare la costante corretta per chiudere l'albero.

All'ultima versione implementata, l'algoritmo riesce ad accettare solo formule formate da un solo simbolo di variabile e funzioni che accettano più parametri. Più precisamente, l'algoritmo non fornisce nessuna elaborazione per formule di questo tipo:

```
(and    (∃ y (∀x (→ A(x)  B(y))))  D(x,z) )
```

Se la formula è composta da una singola variabile, invece, l'algoritmo termina ed accetta formule come la seguente:

(and (∃ y (∀x (→ A(x) B(x)))) D(x))

Se l'enunciato è una tautologia, il programma stampa a video tale informazione.Nel resto dei casi, il programma prova a sviluppare i predicati contenenti quantificatori universali. Se alla fine dei tentativi l'albero non risulta chiuso, l'algoritmo termina senza specificare il risultato , come si può vedere in figura 4.3.

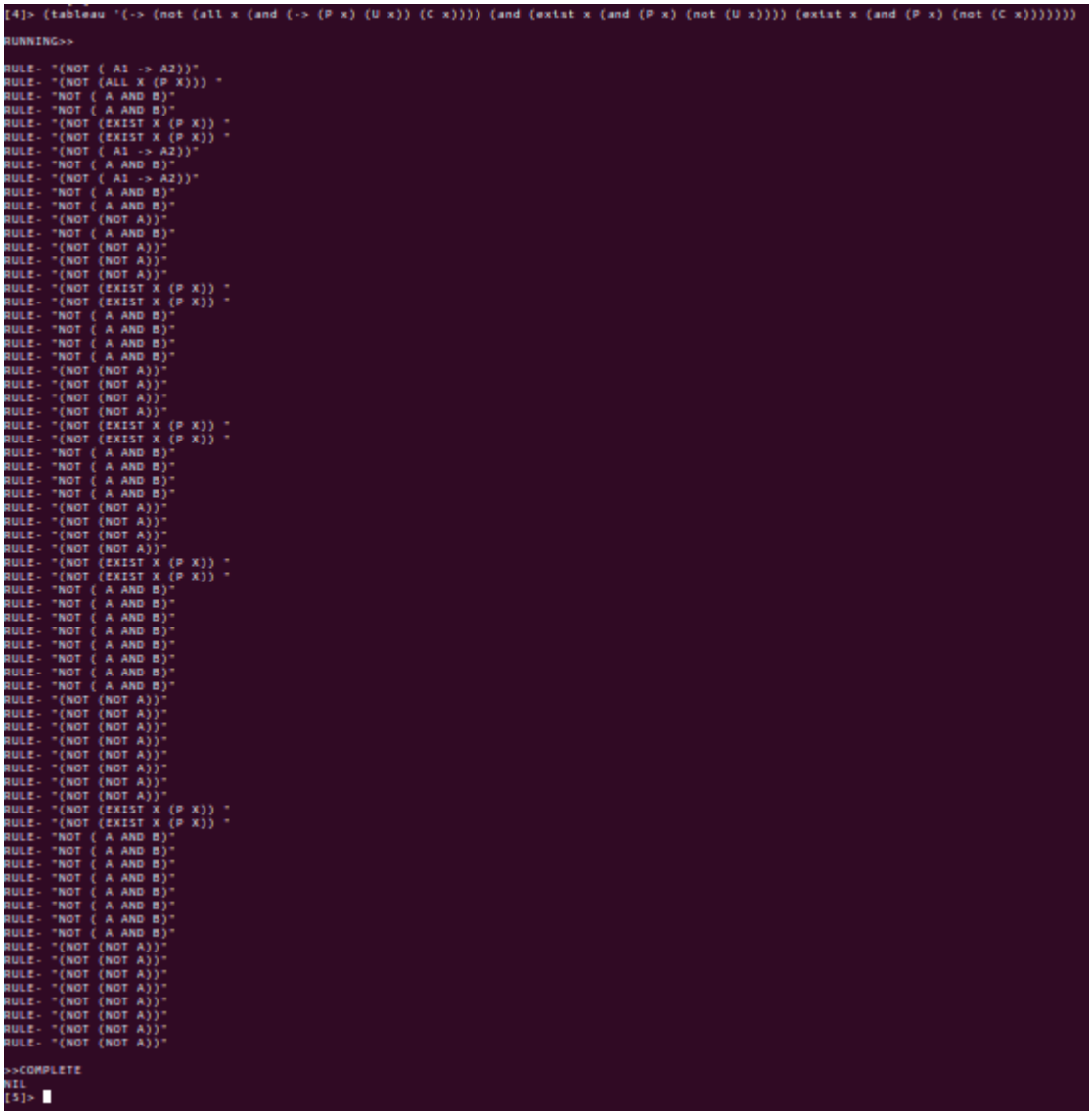

Fig 4.3: Esempio esecuzione del dimostratore di teoremi implementato

Anche se non sono riuscito a completare lo sviluppo del programma, con il tutor abbiamo analizzato e studiato le possibilità per completare l'algoritmo anche per funzioni complesse.

Una di queste, in particolare, riguarda la base di Herbrand. L’idea è di creare un insieme di espressioni finito, sostituendo le variabili dei predicati con tutte le combinazioni di costanti presenti nel linguaggio. In questo modo è possibile ottenere un insieme di formule proposizionali risolvibili in tempo finito.

Anche se il numero di espressioni è esponenziale al numero di costanti, grazie ai principi adottati, è possibile distribuire il lavoro su più processori sfruttando il calcolo parallelo. Tale potenza di calcolo è possibile reperirla anche nelle GPU moderne, sempre più spesso utilizzate per algoritmi di intelligenza artificiale. Con un numero elevato di espressioni risolte, sarà possibile addestrare dei classificatori a riconoscere le caratteristiche delle formule ed usarle come regole per i dimostratori di teoremi.

4.1.2 L'azienda

L'obiettivo dichiarato dell'azienda era riuscire a creare un sistema che fosse in grado di semplificare un'espressione logica e ricavare il valore di verità.

Ho pienamente raggiunto questo obiettivo, sviluppando applicativi che rispettano non solo le caratteristiche funzionali, ma anche le proprietà tecnologiche richieste. Infatti l'applicativo può essere aggiornato costantemente attraverso l'inserimento di nuove regole nella base di conoscenza del sistema esperto sviluppato.

Successivamente al mio periodo in azienda, è iniziata la fase di test con l'integrazione nei sistemi Zucchetti e l'inserimento di quanto prodotto al fine di identificare errori logici commessi dai programmatori al momento della creazione di *routine* all'interno del programma *Infinity*.

Se i test daranno esito positivo, sarà possibile ridurre i tempi di *debugging* con un notevole vantaggio competitivo per l'azienda.

Da un lato i programmatori saranno in grado di individuare e correggere in tempi brevi gli errori di sviluppo con un guadagno in termini di tempo di realizzazione. Dall'altro, sarà ugualmente più rapida l'individuazione e la risoluzione di problemi che si possono verificare durante l'utilizzo del programma da parte dell'utente finale.

4.1.3 Obiettivi personali

Quando ho scelto il progetto di stage, mi ero prefissato degli obiettivi che intendevo raggiungere durante lo svolgimento. In particolare approcciare ai metodi di intelligenza artificiale, approfondire meglio la programmazione funzionale e essere inserito in una realtà aziendale strutturata e all'avanguardia.

Al termine dello stage posso dire di aver raggiunto i miei obiettivi e di essere soddisfatto del percorso svolto.

Senza dubbio lo stage mi ha dato la possibilità di conoscere una realtà importante e strutturata, all'avanguardia nel mondo delle tecnologie come Zucchetti. Ho appreso un metodo di lavoro nello sviluppo dei programmi che sicuramente mi potrà essere utile nel mio futuro non solo lavorativo, ma anche formativo.

Per quanto riguarda l'intelligenza artificiale, lo stage mi ha permesso di avvicinarmi a questa branchia dell'informatica in maniera pratica e dandomi una visione d'insieme della materia e delle possibili applicazioni. Tre i punti fondamentali che ho potuto analizzare:

- introduzione ai concetti: attraverso la formazione teorica in affiancamento al tutor aziendale, sono riuscito a comprendere il contesto generale di analisi, progettazione e utilizzo dei metodi di intelligenza artificiale;
- comprensione di cosa sia un'entità intelligente: ho approfondito e compreso le differenze che contraddistinguono un'applicazione scritta con i principi classici di programmazione da una scritta con i metodi di intelligenza artificiale (sviluppo del semplificatore in due versioni: una standard e una “intelligente”);
- come rendere un'entità intelligente: lo stage mi ha permesso non solo di approfondire gli aspetti teorici, ma anche di metterli in pratica realizzando un'applicazione concreta e funzionante come il sistema esperto.

Passando alla programmazione funzionale, l'obiettivo era di comprendere cosa significasse in concreto questo tipo di programmazione e come fosse possibile utilizzarla in fase di progettazione e codifica.

Come per le applicazione legate all'intelligenza artificiale, ad un primo periodo di formazione teorica in affiancamento è seguito un periodo di applicazione pratica che mi ha permesso di utilizzare i paradigmi di programmazione funzionale (semplificatore di teoremi in versione standard).

L'utilizzo pratico di questo tipo di programmazione mi ha permesso di comprendere al meglio anche i vantaggi e gli svantaggi di questi linguaggi e, per contrapposizione, i vantaggi e gli svantaggi della programmazione imperativa, sia in fase di progettazione che di codifica e applicazione.

4.2 Bilancio formativo

Lo stage per me ha rappresentato anche un valido strumento per analizzare a che livello fossero le mie conoscenze teorico/pratiche e in che modo gli argomenti affrontati durante il percorso di studi potessero trovare applicazione in campo lavorativo.

La stesura del piano di lavoro mi ha permesso di identificare le aree di conoscenza già acquisite e quelle che, invece, avrei dovuto apprendere nel corso dello stage stesso. Alcuni degli argomenti che ho affrontato in azienda (IA, apprendimento automatico, programmazione funzionale) fanno riferimento a corsi d'esame della Corso di Laurea Magistrale, questo però è stato uno stimolo per me poiché volevo fortemente lavorare su questi temi anche in ottica di formazione e lavoro futuro.

4.2.1 Formazione universitaria

Il corso di Laurea triennale in Informatica mi ha sicuramente fornito le basi necessarie per poter comprendere e sviluppare il progetto di stage, anche se, gli argomenti che ho affrontato sono trattati e approfonditi durante il Corso di Laurea Magistrale.

Ho studiato specifici campi di applicazione dell’informatica sia negli aspetti teorici che pratici. Svolgendo il progetto ho apprezzato la possibilità di applicare concretamente nozioni teoriche apprese durante la triennale e in molti casi di approfondirle.

La difficoltà maggiore è stata dall’intensa e rapida fase di studio ed applicazione avvenuta nel periodo di stage. Durante i corsi universitari, solitamente i progetti vengono svolti dopo la spiegazione teorica e questo sicuramente è un vantaggio per lo studente che parte con delle nozioni definite e chiare. Durante lo stage, invece, il processo è stato inverso. Il progetto si è svolto quasi contemporaneamente con la fase di apprendimento teorico e non riguardava un solo argomento. Ho dovuto fare un lavoro di ricerca su più argomenti, capire le varie interazioni e solo successivamente ho potuto applicarle operativamente.

Sviluppare progetti multidisciplinari nei vari corsi, già durante il periodo universitario, potrebbe essere un buon modo per abituare da subito gli studenti alle metodologie di lavoro che incontreranno in azienda.

La difficoltà maggiore che ho riscontrato, è stata l'applicazione de principi della programmazione funzionale in quanto il corso viene affrontato solo alla magistrale. A conclusione del progetto, credo che questo corso potrebbe portare molti benefici agli studenti se inserito nella triennale. Tali nozioni, infatti, mi hanno fornito una visione totalmente differente della programmazione e mi hanno permesso di sviluppare capacità analitiche differenti da quelle apprese con lo studio. Tale differenza d'approccio mi ha permesso di approfondire, di apprendere e sviluppare nuove metodologie sia in fase di progettazione che di codifica. Inoltre mi ha fatto apprezzare le differenze rispetto ai linguaggi imperativi e mi ha dato la possibilità di confrontare le varie metodologie esistenti ed i relativi vantaggi e svantaggi nella loro applicazione.

Posso dire dunque che il percorso di studi mi ha fornito delle buone e solide basi teoriche che mi hanno permesso di affrontare e portare a termine il progetto positivamente. Il merito dallo stage, invece, è stato quello di avermi mostrato un approccio più aziendale ai problemi e una metodologia di lavoro e di ricerca diversa ma complementare.

4.2.2 Esperienza in azienda

Durante il periodo di stage ho avuto modo di affrontare argomenti e tematiche non trattate in aula, sia dal punto di vista teorico, per avere le nozioni base da cui partire, che pratico, dovendole successivamente applicare in fase di codifica.

Sicuramente grande rilievo ha avuto il tempo utilizzato per la ricerca, effettuata con il tutor o in autonomia, per comprendere meglio e approfondire i temi affrontati.

I due argomenti principali su cui ho focalizzato la mia ricerca sono stati quelli collegati ai vincoli che l'azienda aveva posto per lo sviluppo del programma: la programmazione funzionale e l'intelligenza artificiale (sistemi esperti).

In particolare ho approfondito la conoscenza dei dimostratori di teoremi mediante *Tableaux* semantici e del linguaggio di programmazione necessario, ovvero il LISP.

Da ultimo, anche se per un periodo di tempo limitato, ho avuto modo di affrontare il tema del *Machine Learning* e in particolare gli strumenti come i classificatori (utilizzati, come già esposto, nel programma *Orange*).

In conclusione, mi sento di affermare che nonostante gli argomenti trattati durante il periodo di stage non fossero stati affrontati nel mio percorso di studi, la base teorica e pratica acquisita nei tre anni di Università mi ha permesso di approcciare il piano di lavoro e gestirlo in maniera completa e positiva.

Uno dei principali pregi del mio percorso di studi è quello di avermi fornito un metodo di studio e di lavoro efficace che mi ha permesso di affrontare e risolvere problemi anche senza avere una base teorica approfondita.

4.3 Conclusioni

Il bilancio finale del mio periodo di stage è sicuramente positivo: superate le difficoltà iniziali rappresentate dall'inserimento in un contesto lavorativo nuovo e a tratti chiuso e dalla complessità degli argomenti che mi sono trovato a studiare e applicare, ho vissuto in maniera costruttiva questa esperienza.

Durante i due mesi passati in azienda, ho acquisito nuove conoscenze: ho imparato nuovi linguaggi e paradigmi di programmazione, ho approfondito argomenti per me nuovi sia dal punto di vista teorico che pratico, ho appreso nuovi metodi di analisi, progettazione e sviluppo anche da un punto di vista di applicazione aziendale, non solo di ricerca pura.

Il percorso strutturato con il tutor aziendale è stato utile per acquisire nuove abilità soprattutto nella fase di approccio, analisi e studio di argomenti nuovi o problemi da risolvere. Al termine dello stage sento di aver accresciuto le mie competenze in questo campo, in particolare nella scrittura di programmi basati su metodi di intelligenza artificiale e scritti usando i paradigmi della programmazione funzionale

Molto importante per la buona riuscita del progetto è stato il rapporto con il tutor aziendale che mi ha seguito molto nella parte iniziale di ricerca e approfondimento degli argomenti che avremmo poi affrontato.