# Sistema di diagnostica del diabete

Gruppo di lavoro

- Martina Blanco, 758272, m.blanco@studenti.uniba.it

Link repo su GitHub

AA 2024-25

Sommario

# 1. Introduzione

Il **diabete** è una patologia caratterizzata da un eccesso di glucosio (zucchero) nel sangue, noto come iperglicemia. Questa condizione può derivare da una produzione insufficiente di insulina o da una sua azione inadeguata. **L'insulina** è l'ormone fondamentale per la regolazione dei livelli di glucosio. Le due forme più diffuse di diabete sono il tipo 1 (caratterizzato dall'assenza di secrezione insulinica) e il tipo 2 (dovuto a una ridotta sensibilità all'insulina da parte di fegato, muscolo e tessuto adiposo e/o a una ridotta secrezione di insulina dal pancreas). Per **diagnosticare il diabete**, ho sviluppato un programma che si avvale **di due moduli principali**:

- applicazione di algoritmi di apprendimento supervisionato per la diagnostica del diabete.
- un agente intelligente (rule-based) che interagisce con l'utente e utilizza le sue risposte per la diagnosi del diabete.

# 2. Knowledge base

Un sistema esperto è un programma capace di collegare un insieme di fatti a un insieme di regole, eseguendo azioni specifiche in base alle corrispondenze trovate.
Dopo un'accurata ricerca e vari tentativi, ho appreso e utilizzato la libreria Python "Experta", che mi ha permesso di realizzare questo sistema esperto. Vediamo ora come funziona.

## 2.1 Fatti

1. La classe **Fact** è una sottoclasse di **dict**.

```
>>> f = Fact(a=1, b=2)
>>> f['a']
1
```

2. Un **Fact**, a differenza di un normale **dict**, non mantiene un ordine interno degli elementi.

```
>>> Fact(a=1, b=2)
Fact(b=2, a=1)
```

3. È possibile creare un **Fact** anche senza chiavi, utilizzando solo valori. In questo caso, **Fact** assegnerà automaticamente un indice numerico ai valori.

```
>>> f = Fact('x', 'y', 'z')
>>> f[0]
'x'
```

4. I valori autonumerici possono essere combinati con i valori-chiave, ma è importante dichiarare prima i valori autonumerici.

```
>>> f = Fact('x', 'y', 'z', a=1, b=2)
>>> f[1]
'y'
>>> f['b']
2
```

5. È possibile sottoclassare **Fact** per definire tipi di dati personalizzati o estenderli con funzionalità specifiche.

```
class Alert(Fact):
    """The alert level."""
    pass

class Alert(Fact):
    """The alert level."""
    pass

class Status(Fact):
    """The system status."""
    pass

f1 = Alert('red')
f2 = Status('critical')
```

6. I campi dei fatti possono essere automaticamente validati definendoli tramite Field. Questa funzionalità sfrutta la libreria Schema internamente per la convalida dei dati. Inoltre, un campo può essere dichiarato come obbligatorio o con un valore predefinito.

## 2.2 Regole

In Experta, una **regola** è un *callable* dichiarato con Rule.
Le regole si compongono di due parti: la LHS (parte sinistra) e la RHS (parte destra).

- La LHS descrive, tramite l'uso di **pattern**, le condizioni che devono essere soddisfatte affinché la regola venga eseguita (o **attivata**).
- La RHS è l'insieme delle azioni che vengono eseguite quando la regola viene attivata.

Perché un Fact corrisponda a un Pattern, tutte le restrizioni del pattern devono essere True quando il Fact viene valutato rispetto ad esso.

```python
class MyFact(Fact):
    pass

@Rule(MyFact())  # This is the LHS
def match_with_every_myfact():
    """This rule will match with every instance of `MyFact`."""
    # This is the RHS
    pass

@Rule(Fact('animal', family='felinae'))
def match_with_cats():
    """
    Match with every `Fact` which:

      * f[0] == 'animal'
      * f['family'] == 'felinae'

    """
    print("Meow!")
```

È possibile utilizzare operatori logici per esprimere condizioni LHS complesse.

```python
@Rule(
    AND(
        OR(User('admin'),
           User('root')),
        NOT(Fact('drop-privileges'))
    )
)
def the_user_has_power():
    """
    The user is a privileged one and we are not dropping privileges.

    """
    enable_superpowers()
```

Affinché una regola sia operativa, deve essere un metodo di una sottoclasse di KnowledgeEngine.

## 2.3 Fatti contro Modelli

La distinzione tra fatti e modelli è sottile. I modelli sono, di fatto, solo fatti che contengono **elementi condizionali del modello** anziché dati standard. Vengono utilizzati esclusivamente nella LHS di una regola. Se il contenuto di un pattern non viene fornito come PCE (Pattern Conditional Element), Experta lo includerà automaticamente come LiteralPCE. Inoltre, è importante notare che non è possibile dichiarare un Fact contenente un **PCE**; tentare di farlo genererà un'eccezione.

```
>>> ke = KnowledgeEngine()
>>> ke.declare(Fact(L("hi")))
Traceback (most recent call last):
  File "<ipython-input-4-b36cff89278d>", line 1, in <module>
    ke.declare(Fact(L('hi')))
  File "/home/experta/experta/engine.py", line 210, in declare
    self.__declare(*facts)
  File "/home/experta/experta/engine.py", line 191, in __declare
    "Declared facts cannot contain conditional elements")
TypeError: Declared facts cannot contain conditional elements
```

## 2.4 DefFacts

Nella maggior parte dei casi, i sistemi esperti richiedono che un insieme di fatti sia già presente per poter funzionare correttamente. Questo è lo scopo di **DefFacts**.

```
@DefFacts()
def needed_data():
    yield Fact(best_color="red")
    yield Fact(best_body="medium")
    yield Fact(best_sweetness="dry")
```

Tutti i **DefFacts** all'interno di un **KnowledgeEngine** verranno richiamati ogni volta che viene eseguito il metodo reset.

## 2.5 Motore della conoscenza

Qui è dove avviene la vera e propria magia. Il primo passo è estendere la classe **KnowledgeEngine** e dichiarare i suoi metodi con Rule. Successivamente, si crea un'istanza, la si popola con i fatti e infine la si esegue.

```
from experta import *

class Greetings(KnowledgeEngine):
    @DefFacts()
    def _initial_action(self):
        yield Fact(action="greet")

    @Rule(Fact(action='greet'),
          NOT(Fact(name=W())))
    def ask_name(self):
        self.declare(Fact(name=input("What's your name? ")))

    @Rule(Fact(action='greet'),
          NOT(Fact(location=W())))
    def ask_location(self):
        self.declare(Fact(location=input("Where are you? ")))

    @Rule(Fact(action='greet'),
          Fact(name=MATCH.name),
          Fact(location=MATCH.location))
    def greet(self, name, location):
        print("Hi %s! How is the weather in %s?" % (name, location))

engine = Greetings()
engine.reset()  # Prepare the engine for the execution.
engine.run()  # Run it!
```

## 2.6 Gestione dei fatti

Per manipolare l'insieme di fatti conosciuti dal motore, si utilizzano i seguenti metodi.

### 2.6.1 Declare

Questo metodo aggiunge un nuovo fatto all'elenco dei fatti noti al motore.

```
>>> engine = KnowledgeEngine()
>>> engine.reset()
>>> engine.declare(Fact(score=5))
<f-1>
>>> engine.facts
<f-0> InitialFact()
<f-1> Fact(score=5)
```

Nota: Lo stesso fatto non può essere dichiarato due volte, a meno che **fact.duplication** non sia impostato su True.

### 2.6.2 Retract

Questo metodo rimuove un fatto esistente dall'elenco dei fatti.

```
>>> engine.facts
<f-0> InitialFact()
<f-1> Fact(score=5)
<f-2> Fact(color='red')
>>> engine.retract(1)
>>> engine.facts
<f-0> InitialFact()
<f-2> Fact(color='red')
```

### 2.6.3 Modify

Questo metodo ritira uno o più fatti esistenti e ne dichiara uno nuovo con modifiche specifiche, passate come argomenti.

```
>>> engine.facts
<f-0> InitialFact()
<f-1> Fact(color='red')
>>> engine.modify(engine.facts[1], color='yellow', blink=True)
<f-2>
>>> engine.facts
<f-0> InitialFact()
<f-2> Fact(color='yellow', blink=True)
```

### 2.6.4 Duplicate

Questo metodo aggiunge un nuovo fatto all'elenco dei fatti, utilizzandone uno esistente come modello e applicando alcune modifiche.

```
>>> engine.facts
<f-0> InitialFact()
<f-1> Fact(color='red')
>>> engine.duplicate(engine.facts[1], color='yellow', blink=True)
<f-2>
>>> engine.facts
<f-0> InitialFact()
<f-1> Fact(color='red')
<f-2> Fact(color='yellow', blink=True)
```

## 2.7 Procedura di esecuzione del motore

Il processo standard per eseguire un **KnowledgeEngine** è il seguente:

1. La classe KnowledgeEngine deve essere istanziata.

2. Deve essere chiamato il metodo reset():
    - Questo dichiara il fatto speciale *InitialFact*, essenziale per il corretto funzionamento di alcune regole.
    - Dichiara tutti i fatti generati dai metodi con **@DefFacts.**
3. Deve essere chiamato il metodo **run()**, che avvia il ciclo di esecuzione.

## 2.8 Ciclo di esecuzione

A differenza della programmazione convenzionale, dove il programmatore definisce esplicitamente il punto di inizio, fine e la sequenza delle operazioni, con Experta il flusso del programma non richiede una definizione così esplicita. La conoscenza (Regole) e i dati (Fatti) sono separati, e il KnowledgeEngine si occupa di applicare la conoscenza ai dati.
Il ciclo di esecuzione di base è il seguente:

1. Se è stato raggiunto il limite di "spari" della regola, l'esecuzione si interrompe.
2. La regola in cima all'ordine del giorno viene selezionata per l'esecuzione. Se l'ordine del giorno è vuoto, l'esecuzione si interrompe.
3. Vengono eseguite le azioni della RHS della regola selezionata (il metodo viene chiamato). Di conseguenza, le regole possono essere **attivate** o **disattivate**. Le regole attivate (quelle le cui condizioni sono soddisfatte) vengono inserite nell'agenda. La loro posizione nell'agenda è determinata dalla rilevanza della regola e dalla **strategia attuale di risoluzione dei conflitti**. Le regole disattivate vengono rimosse dall'agenda.

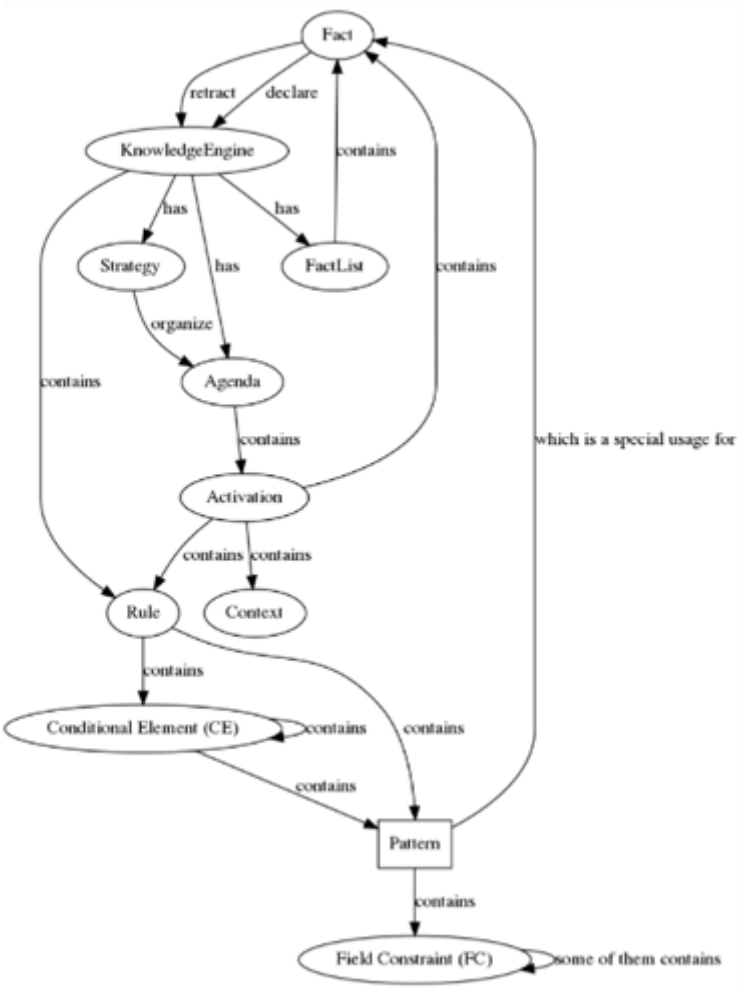
## 2.9 Differenza tra DefFacts e declare

Entrambi i metodi sono usati per dichiarare fatti sull'istanza del motore, ma con le seguenti distinzioni:

- declare aggiunge i fatti direttamente alla memoria di lavoro.
- I generatori dichiarati con DefFacts vengono richiamati dal metodo reset(), e tutti i fatti prodotti vengono aggiunti alla memoria di lavoro utilizzando require.

## 2.10 Diagramma

Il diagramma seguente illustra tutti i componenti del sistema e le relazioni tra di essi.




## 2.11 Esempio di funzionamento del sistema esperto per la diagnosi del diabete

Il nostro sistema esperto opera attraverso un meccanismo basato sul dialogo con l'utente. Durante l'interazione vengono poste una serie di domande mirate, pensate per raccogliere informazioni utili sullo stato di salute della persona. A seconda delle risposte fornite, il sistema genera e registra dei *facts* (cioè affermazioni o proposizioni), che rappresentano la base di conoscenza da cui parte il ragionamento automatico.

Una volta che i fatti sono stati acquisiti, il motore inferenziale del sistema li elabora applicando le regole predefinite, fino a formulare una valutazione finale. In questo modo, l'agente intelligente è in grado di fornire una diagnosi preliminare sulla presenza o meno del diabete, simulando il processo logico che un medico potrebbe adottare di fronte ai dati raccolti.

## 2.12 CSP

All'interno dell'intelligenza artificiale, molti problemi possono essere affrontati come **Problemi di Soddisfacimento di Vincoli** (*Constraint Satisfaction Problem*, CSP). Un CSP si definisce su un insieme finito di variabili (X1, X2, …, Xn), ognuna delle quali può assumere valori da un determinato dominio (D1, D2, …, Dn). Su queste variabili vengono imposti dei vincoli (C1, C2, …, Cn), che limitano le possibili combinazioni di valori ammissibili.

In termini pratici, un vincolo può essere visto come un insieme di condizioni che specificano quali valori le variabili possono assumere contemporaneamente. Si tratta, in sostanza, di un sottoinsieme del prodotto cartesiano dei domini delle variabili coinvolte, e può essere espresso attraverso diverse rappresentazioni: matrici, equazioni, disuguaglianze o relazioni logiche.
Un concetto importante è il **grado di una variabile**, che indica il numero di vincoli a cui essa è soggetta.

La risoluzione di un CSP inizia a partire da un'assegnazione iniziale (che può essere parziale o anche del tutto vuota). Il processo procede poi estendendo progressivamente questa assegnazione, fino ad arrivare a una configurazione completa e coerente, cioè un insieme di valori che soddisfi contemporaneamente tutti i vincoli imposti.

Nel nostro progetto abbiamo utilizzato la libreria Python **constraint**, che ci ha permesso di implementare un CSP semplice ma efficace. L'idea era quella di gestire la disponibilità dei laboratori convenzionati nel caso in cui il sistema diagnostico suggerisse di effettuare ulteriori analisi cliniche.

Il funzionamento si può riassumere nei seguenti passaggi:

- Alla base vi è una sottoclasse di *Problem*, già definita nella libreria, che rappresenta il modello del CSP.
- Successivamente vengono dichiarate esplicitamente le variabili e i rispettivi domini.
- In base alle risposte dell'utente, il sistema valuta se sia necessario prescrivere degli esami aggiuntivi.
- Qualora fosse necessaria una prescrizione, entra in azione il CSP, che si occupa di gestire gli orari disponibili nei laboratori.
- Ad esempio, se un laboratorio per l'analisi dell'insulina è operativo dalle ore 8:00 alle 14:00, il CSP calcola e restituisce le fasce orarie compatibili.
- Infine, l'utente riceve un'indicazione chiara delle finestre temporali in cui può recarsi presso il laboratorio per eseguire gli esami.

### 2.12.1 Un possibile esempio di utilizzo

```
Benvanuto in Diabetes Expert, un sistema esperto per la diagnosi e la cura del diabete di tipo 1
Ti senti molto assetato di solito (sopratutto di notte) ? [si/no]

no
Ti senti molto stanco? [si/no]

si
Stai perdendo peso e massa muscolare? [si/no]

no
Senti prurito? [si/no]

no
Hai la vista offuscata? [si/no]

si
Consumi spesso bevande zuccherate? [si/no]

no
Hai fame costantemente? [si/no]

no
Hai spesso la bocca asciutta? [si/no]

no


Inserisci l'altezza in centimetri


180
Inserisci il peso in kilogrammi


90
Hai fatto l'esame della pressione sanguigna?

si
Inserisci il valore della pressione sanguigna

190
Il valore della pressione e' maggiore o uguale a quella dei diabetici
Potresti avere il diabete, rivolgiti ad un medico
```

```
L'agente ragiona con i seguenti fatti:

<f-0>: InitialFact()
<f-1>: Fact(inizio='si')
<f-2>: Fact(chiedi_sintomi='si')
<f-3>: Fact(molto_stanco='si')
<f-4>: Fact(vista_offuscata='si')
<f-5>: Fact(chiedi_imc='si')
<f-6>: Fact(esami_pressione='si')
<f-7>: Fact(esame_pressione_eseguito='si')
<f-8>: Fact(diagnosi_pressione_diabete='si')
<f-9>: Fact(diagnosi_diabete_incerta='si')
```

In base alle risposte dell'utente date nell'esempio, l'agente ragiona con i seguenti fatti

## 2.13 Ontologie

Nel campo dell'informatica, il termine *ontologia* indica una rappresentazione formale, esplicita e condivisa della conoscenza relativa a un determinato dominio. In altre parole, un'ontologia definisce in modo strutturato i concetti fondamentali di un ambito di interesse e le relazioni che intercorrono tra essi.

Più nello specifico, un'ontologia può essere considerata come una teoria assiomatica espressa tramite logiche descrittive di primo ordine, che consente di rappresentare in maniera rigorosa e non ambigua la conoscenza. Questo approccio è stato introdotto in maniera significativa nell'ambito dell'Intelligenza Artificiale e della rappresentazione della conoscenza per fornire strumenti capaci di organizzare, collegare e formalizzare dati complessi.

Una *ontologia formale* consente infatti di integrare diversi schemi informativi all'interno di un'unica struttura coerente, capace di includere tutte le entità rilevanti di un dominio e le rispettive relazioni. Una volta costruita, può essere utilizzata dai sistemi informatici per molteplici scopi: dal ragionamento automatico alla classificazione, fino all'impiego in tecniche avanzate di problem solving.

Nel progetto da noi realizzato, l'ontologia è stata sviluppata con l'ausilio dello strumento **Protégé**, un ambiente largamente utilizzato per la modellazione di ontologie. La gestione e l'utilizzo dell'ontologia sono stati invece resi possibili tramite la libreria Python **Owlready2**, che consente di importare le strutture ontologiche, leggerle e manipolarle per poi integrarle nei processi di ragionamento automatico.

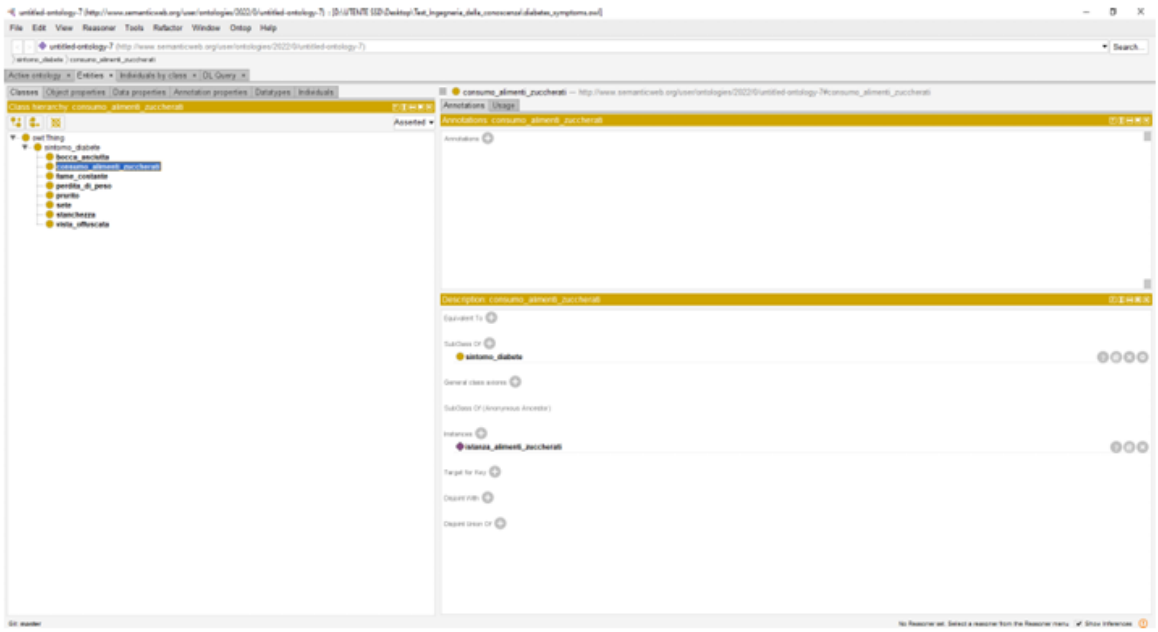


*Figura 1: Tool Protege*

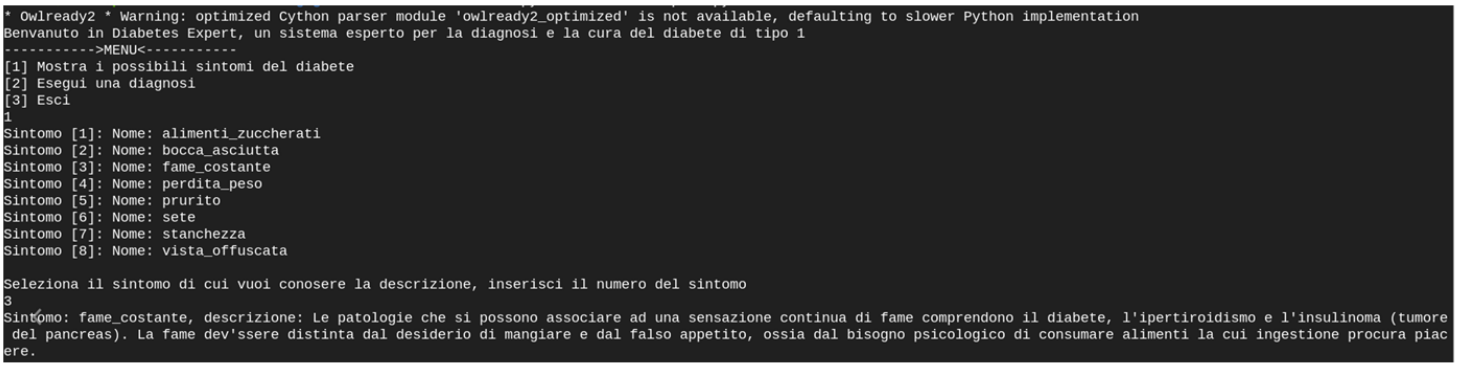



*Figura 2: Dall'ontologia viene creato un dizionario con i sintomi e la relativa descrizione*

# 3. Algoritmi di classificazione

Per la fase di classificazione abbiamo scelto di utilizzare il **dataset Pima Indians Diabetes**, disponibile sulla piattaforma Kaggle (https://www.kaggle.com/uciml/pima-indians-diabetes-database). Questo insieme di dati contiene numerose caratteristiche cliniche associate ai pazienti, ed è ampiamente impiegato come benchmark per problemi di predizione del diabete.

Le principali variabili considerate nel dataset sono:

- Numero di gravidanze (successivamente eliminata in fase di pre-processing);
- Concentrazione di glucosio plasmatico a 2 ore da test di tolleranza al glucosio orale;
- Pressione arteriosa diastolica (mm Hg);
- Spessore della piega cutanea del tricipite (mm);
- Livello di insulina sierica a 2 ore (µU/ml);
- Indice di massa corporea (BMI, calcolato come peso/altezza$^2$);
- Funzione pedigree del diabete (che riflette la predisposizione familiare);
- Età del paziente;
- Variabile target (classe 0 o 1), che indica l'assenza o la presenza di diabete. Dei 768 casi presenti, 268 corrispondono a soggetti con esito positivo.

Per costruire e valutare il nostro sistema abbiamo scelto di implementare tre differenti algoritmi di classificazione supervisionata:

- **Regressione logistica**
- **Albero di decisione**
- **K-Nearest Neighbors (K-NN)**

## 3.1 Albero di decisione

Gli alberi di decisione costituiscono una tecnica semplice ma al tempo stesso estremamente efficace per risolvere problemi di classificazione. Si basano su una struttura ad albero, in cui ogni **nodo interno** corrisponde a una condizione (o domanda) relativa a una caratteristica del dataset. A

seconda del valore assunto da tale caratteristica, si percorre un ramo diverso dell'albero, fino a giungere a una **foglia**, che rappresenta la classe di appartenenza del caso analizzato.

Nella pratica, gli alberi vengono costruiti tramite algoritmi che individuano progressivamente le condizioni più informative, con l'obiettivo di massimizzare la separazione tra le classi. Particolare rilevanza viene spesso attribuita agli **split binari**, in cui ogni nodo suddivide i dati in due possibili ramificazioni, semplificando così il processo decisionale.

## 3.2 Regressione logistica

La **regressione logistica** è un modello statistico ampiamente utilizzato nel machine learning per problemi di classificazione binaria. Si tratta di un algoritmo supervisionato che stima la probabilità di appartenenza a una determinata classe tramite la funzione logistica (o sigmoide), la quale trasforma valori reali in un intervallo compreso tra 0 e 1.

Durante la fase di addestramento, il modello riceve in ingresso un dataset costituito da un insieme di esempi, ciascuno composto da più attributi (X) e da un'etichetta di classe (y). Nel nostro caso, il preprocessing ha previsto la rimozione della feature relativa al numero di gravidanze. Successivamente, i dati vengono organizzati in input features e target features, dove la variabile *Outcome* rappresenta l'etichetta da predire (diabetico/non diabetico).

Il cuore del modello è la combinazione lineare degli attributi:

$$z = W \cdot X = w_1 x_1 + \ldots + w_m x_m$$

Tale valore viene trasformato tramite la funzione sigmoide in una probabilità compresa tra 0 e 1. Ad esempio, fissando una soglia a 0.6, se f(z) > 0.6 il modello classificherà l'individuo come positivo al diabete, altrimenti come negativo.

L'algoritmo stima i pesi WW che massimizzano la probabilità di predizione corretta (log-likelihood). Al termine del training, il modello è in grado di classificare nuovi casi non presenti nei dati di addestramento.

## 3.3 K-Nearest Neighbors (K-NN)

Il **K-NN** è un algoritmo molto intuitivo e si basa sul principio della somiglianza: un oggetto viene classificato in base alle caratteristiche dei suoi *k* vicini più prossimi nello spazio delle feature.

- In **classificazione**, l'oggetto viene assegnato alla classe più frequente tra i suoi *k* vicini. Se ad esempio k=1, il punto viene associato direttamente alla classe del singolo vicino più vicino.
- In **regressione**, invece, il valore predetto è la media dei valori assunti dai *k* vicini.

Una scelta cruciale è quella del valore di *k*: un numero troppo basso rende il modello sensibile al rumore, mentre uno troppo alto può ridurre la capacità di distinguere le classi. Nei problemi binari, è spesso consigliabile scegliere valori dispari di *k* per evitare situazioni di pareggio.

### 3.3.1 Fase di apprendimento

L'algoritmo non richiede una vera fase di addestramento: lo spazio dei dati viene semplicemente organizzato, in modo che sia possibile calcolare le distanze tra punti.

### 3.3.2 Calcolo della distanza

Per misurare la somiglianza tra gli esempi, i dati vengono rappresentati come vettori in uno spazio multidimensionale. La distanza più comunemente usata è quella **euclidea**, ma sono impiegabili anche altre metriche, come la distanza Manhattan o, per dati simbolici, la distanza di Hamming. La scelta della metrica può influenzare in maniera significativa le prestazioni.

### 3.3.3 Fase di classificazione

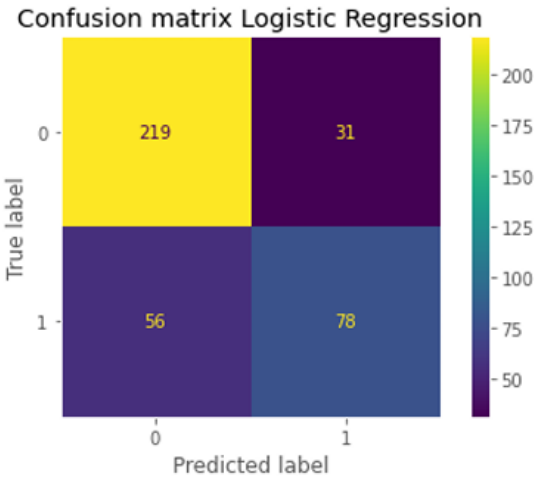
Quando un nuovo esempio deve essere classificato, l'algoritmo individua i suoi *k* vicini più vicini e determina la classe in base alla maggioranza dei voti. In caso di regressione, viene calcolata la media dei valori dei vicini. Per ridurre il rischio che classi più numerose dominino il risultato, è possibile introdurre un **peso basato sulla distanza**, dando maggiore importanza agli esempi più vicini.

## 3.4 Implementazione dei modelli di apprendimento supervisionato in sklearn

Per la fase di implementazione degli algoritmi di classificazione è stata utilizzata la libreria **scikit-learn (sklearn)**, uno degli strumenti più diffusi in Python per il machine learning. Grazie alle numerose classi e funzioni già disponibili, sklearn consente di definire in modo semplice i modelli, addestrarli sui dati e valutarne le prestazioni.

Il flusso di lavoro seguito può essere riassunto nei seguenti passaggi:

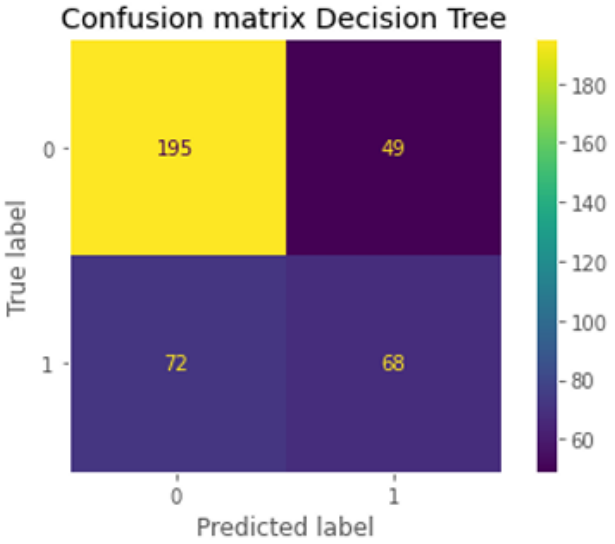
1. All'interno del file *main.py*, per ciascun algoritmo selezionato viene istanziata la relativa classe fornita dalla libreria.
2. Successivamente, il modello viene addestrato sui dati di training e impiegato per effettuare predizioni sui dati di test.
3. Una volta ottenuti i risultati, viene generata la **matrice di confusione**, che permette di osservare nel dettaglio i casi di classificazione corretta e quelli di errore.
4. A partire dalla matrice, vengono calcolate le principali metriche di valutazione: **accuratezza**, **precision**, **recall** e **F1-score**.
5. Tutte queste misure sono state ottenute tramite i metodi messi a disposizione direttamente da sklearn, il che ha reso l'implementazione rapida ed efficace.



```
Logistic Regression metrics
Accurancy : 0.792
Precision : 0.765
Recall : 0.568
F1_precision : 0.652
```

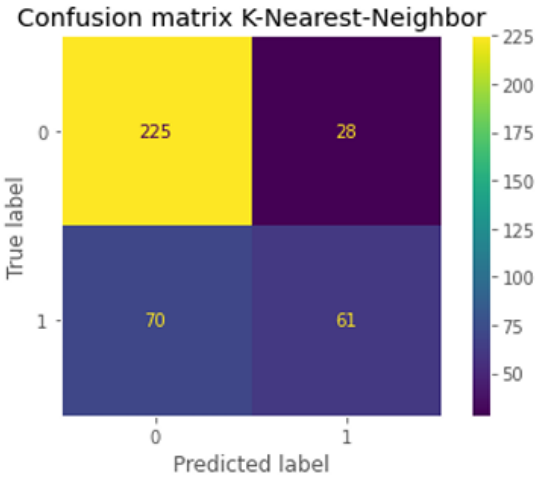
Dall'analisi della matrice di confusione emergono alcune osservazioni: ad esempio, in un primo caso si contano **219 veri positivi** (soggetti correttamente identificati come diabetici) e **78 veri negativi** (soggetti correttamente classificati come non diabetici). Vi sono tuttavia anche **31 falsi positivi** e **56 falsi negativi**, ossia errori di classificazione che il modello non è riuscito a gestire correttamente

In un altro scenario, il numero di veri positivi scende a 195 e i veri negativi a 68, mentre aumentano sia i falsi positivi (49) che i falsi negativi (72). Le metriche corrispondenti sono riportate sempre di seguito:




```
Decision tree metrics
Accurancy : 0.669
Precision : 0.534
Recall : 0.515
F1_precision : 0.524
```

Un ulteriore esperimento evidenzia **225 veri positivi** e **61 veri negativi**, con 28 falsi positivi e 70 falsi negativi. Anche in questo caso le metriche di valutazione riflettono l'andamento complessivo del modello.




```
Knn tree metrics
Accurancy : 0.771
Precision : 0.750
Recall : 0.500
F1_precision : 0.600
```

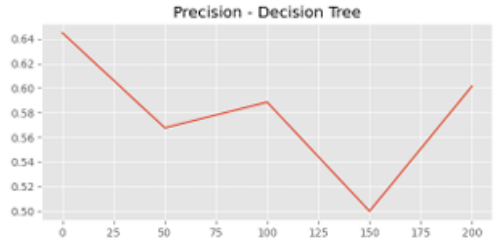
Per comprendere meglio l'impatto dei parametri interni, è stata condotta un'analisi di sensibilità. Nel caso della regressione logistica, ad esempio, è stato variato il numero massimo di iterazioni consentite durante l'addestramento:

- L'**accuratezza** raggiunge valori elevati intorno alle 700 o 1200 iterazioni;
- La **precisione** ottiene i migliori risultati intorno alle 300 iterazioni;
- L'**F1-score** si stabilizza intorno alle 1900 iterazioni.

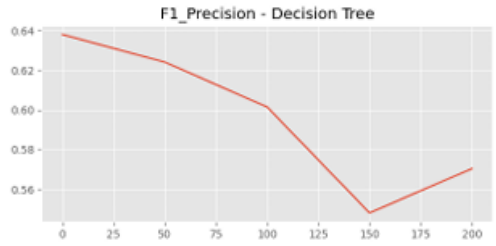




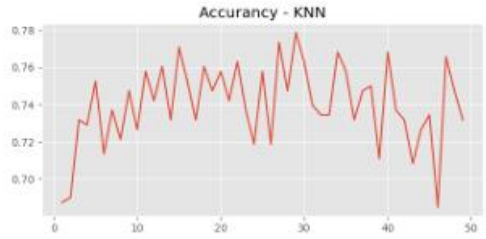
Per quanto riguarda l'albero di decisione, si è osservato che incrementare eccessivamente la profondità (ad esempio fino a 150 livelli) porta a un forte peggioramento delle prestazioni, probabilmente a causa di fenomeni di overfitting.

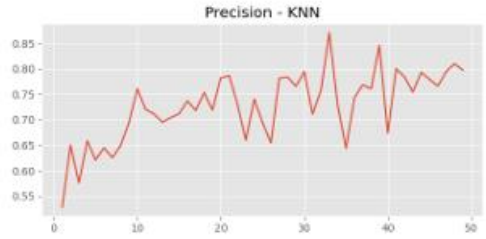



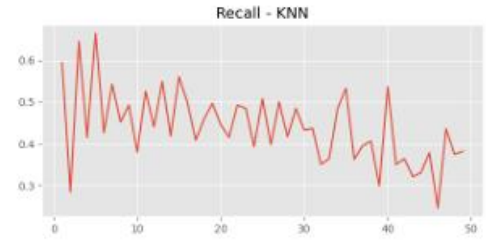


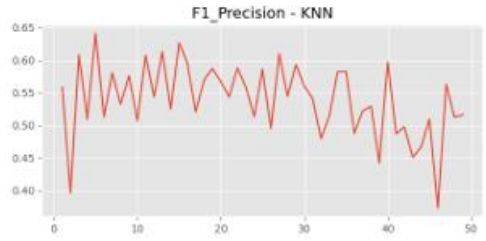



Nel caso del K-NN, il parametro analizzato è stato il numero di vicini (*neighbors*). Le prestazioni risultano piuttosto variabili e meno stabili rispetto agli altri algoritmi, mostrando una sensibilità elevata alla scelta di *k*.

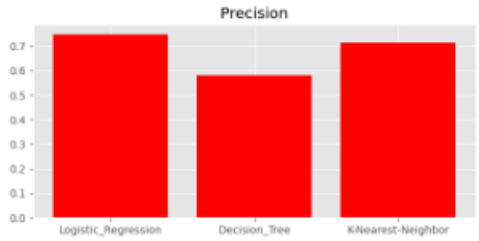


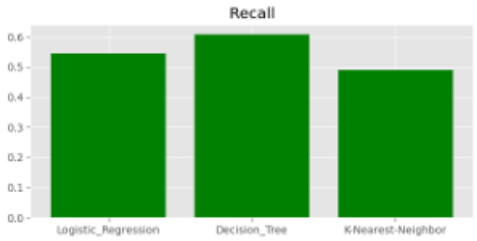



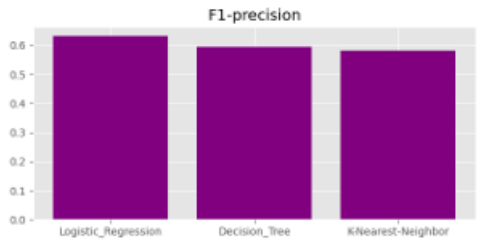



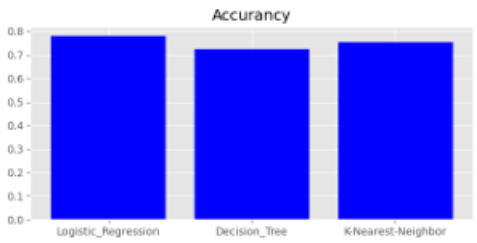




In generale, dal confronto finale emerge che i tre modelli si equivalgono in termini di prestazioni complessive, anche se la **regressione logistica** si distingue leggermente come l'approccio più affidabile e costante nei risultati.




## 4. Conclusioni e Lavoro Futuro

Il progetto sviluppato ha dimostrato come sia possibile integrare approcci diversi di intelligenza artificiale per affrontare un problema complesso come la diagnosi del diabete. Da un lato, l'impiego di un **sistema esperto rule-based** ha permesso di simulare un processo decisionale basato su regole e fatti dichiarati dall'utente; dall'altro, l'utilizzo di **algoritmi di machine learning supervisionato** ha consentito di costruire modelli predittivi capaci di generalizzare a partire dai dati disponibili.

I risultati ottenuti sono complessivamente positivi: le prestazioni dei modelli di classificazione sono state soddisfacenti e, in particolare, la regressione logistica si è rivelata l'algoritmo più stabile ed efficace. Ciò conferma come la combinazione di metodi simbolici e statistici possa offrire un valore aggiunto nel campo delle applicazioni mediche.

Nonostante ciò, il lavoro svolto rappresenta solo un punto di partenza. L'idea alla base del progetto si presta infatti a numerose estensioni e miglioramenti futuri, che potrebbero renderlo un supporto ancora più utile nella pratica clinica.

## 4.1 Limiti del Progetto Attuale

L'analisi condotta ha messo in luce alcuni limiti significativi:

- **Dataset limitato**: il numero di campioni utilizzato non è particolarmente elevato e questo incide sulla capacità di generalizzare del modello.
- **Semplicità delle regole**: le regole implementate nel sistema esperto sono volutamente basilari e non coprono tutte le possibili casistiche cliniche.
- **Assenza di validazione clinica**: le diagnosi prodotte dal sistema non sono state verificate da personale medico qualificato, pertanto il progetto ha solo valore sperimentale.
- **Interfaccia poco intuitiva**: l'interazione avviene principalmente da terminale, rendendo l'utilizzo meno agevole per utenti non tecnici.

Questi aspetti limitano l'applicabilità pratica del sistema e indicano la necessità di ulteriori sviluppi.

## 4.2 Possibili Sviluppi Futuri

Per rendere il sistema più completo e vicino a un impiego reale, si possono ipotizzare diversi miglioramenti:

- **Arricchimento della knowledge base**, inserendo nuove regole cliniche e ampliando l'ontologia con ulteriori concetti e relazioni mediche.
- **Utilizzo di dataset più ampi e diversificati**, preferibilmente provenienti da fonti cliniche certificate, così da migliorare la robustezza dei modelli di machine learning.
- **Introduzione di algoritmi più complessi**, come reti neurali profonde o metodi ensemble (ad esempio Random Forest e Gradient Boosting), che potrebbero incrementare le performance predittive.
- **Sviluppo di un'interfaccia grafica user-friendly**, per permettere anche a utenti non esperti di interagire facilmente con il sistema.
- **Integrazione con strumenti di telemedicina**, che consentirebbero l'impiego del sistema in contesti di monitoraggio a distanza, offrendo un supporto diagnostico continuo ai pazienti.

Tali prospettive dimostrano come il progetto possa evolvere in una direzione sempre più vicina alle esigenze reali della sanità digitale, rappresentando un utile supporto, se opportunamente validato, al lavoro dei professionisti della salute.