Sistema di diagnostica sul diabete

Federico Canistro, Matricola: **723320**
Simone Gramegna, Matricola:**717041**
Email: f.canistro@studenti.uniba.it , s.gramegna5@studenti.uniba.it

Indice

1 Introduzione

Il diabete è una malattia che si caratterizza per la presenza di quantità eccessive di glucosio (zucchero) nel sangue. L'eccesso di glucosio, noto con il termine di iperglicemia, può essere causato da un'insufficiente produzione di insulina o da una sua inadeguata azione; l'insulina è l'ormone che regola il livello di glucosio nel sangue. Le forme più note di diabete sono due: il diabete di tipo 1 (con assenza di secrezione insulinica) e il diabete di tipo 2, conseguente a ridotta sensibilità all'insulina da parte di fegato, muscolo e tessuto adiposo e/o a una ridotta secrezione di insulina da parte del pancreas. Il nostro programma, per diagnosticare il diabete utilizza due moduli:

- L'utilizzo di algoritmi di apprendimento supervisionato per la diagnostica del diabete.

- Un agente inteligente (rule-based) che tiene conto delle risposte dell'utente per la diagnostica del diabete.

2 Knowledge base

Il nostro gruppo ha deciso di creare un sistema esperto in grado di diagnosticare il diabete.

Un sistema esperto è un programma in grado di associare un insieme di **fatti** a un insieme di **regole** a quei fatti, ed eseguire alcune azioni in base alle regole di corrispondenza.

Dopo una lunga ricerca e molti tentativi, abbiamo trovato ed imparato ad usare questa libreria python di nome Experta, che ci ha aiutato a creare il nostro sistema esperto. Vediamo adesso come funziona questo sistema.

2.1 Fatti

1. La classe Fact è una sottoclasse di dict.

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

2. Quindi un Fatto non mantiene un ordine interno di elementi.

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

3. A differenza di dict , puoi creare un Fact senza chiavi (solo valori) e Fact creerà un indice numerico per i tuoi valori.

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

4. Puoi combinare i valori autonumerici con i valori-chiave, ma l'autonumeric deve essere dichiarato prima:

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

5. Puoi sottoclassare Fact per esprimere diversi tipi di dati o estenderli con la tua funzionalità personalizzata.

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

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

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

```
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 convalidati automaticamente per te, se li definisci utilizzando Field . Il campo utilizza la libreria Schema internamente per la convalida dei dati. Inoltre, un campo può essere dichiarato obbligatorio o avere un valore predefinito .

2.2 Regole

In Experta una **regola** è un callable, decorato con Regola .
Le regole hanno due componenti, LHS (lato sinistro) e RHS (lato destro).

- Il LHS descrive (usando **patterns**) le condizioni in base alle quali la regola * dovrebbe essere eseguita (o attivata).
- L' RHS è l'insieme delle azioni da eseguire quando la regola viene attivata.

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

```
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.

```
@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()
```

Perché una **regola** sia utile, deve essere un metodo di una sottoclasse **KnowledgeEngine**.

2.3 Fatti contro Modelli

La differenza tra fatti e modelli è piccola. In effetti, i modelli sono solo fatti che contengono **elementi condizionali del modello** invece di dati regolari. Sono utilizzati solo nella LHS di una regola.
Se non fornisci il contenuto di un pattern come **PCE** , Experta includerà automaticamente il valore in un LiteralPCE.
Inoltre, non puoi dichiarare alcun Fact contenente un **PCE** , se lo fai, riceverai una bella eccezione indietro.

```
>>> 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 Difetti

La maggior parte delle volte i sistemi esperti necessitano di una serie di fatti per essere presenti affinché il sistema funzioni. Questo è lo scopo del decoratore 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 chiamati ogni volta che viene chiamato il metodo reset .

2.5 Motore della conoscenza

Questo è il luogo dove avviene tutta la magia. Il primo passo è farne una sottoclasse e usare Rule per decorare i suoi metodi. Successivamente, puoi crearne un'istanza, popolarlo con i fatti e infine eseguirlo.

```
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!
```

```
$ python greet.py
What's your name? Roberto
Where are you? Madrid
Hi Roberto! How is the weather in Madrid?
```

Figura 2.5.1: Output dell'esempio di sopra

2.6 Gestione dei fatti

I seguenti metodi vengono utilizzati per manipolare l'insieme di fatti di cui il motore è a conoscenza.

2.6.1 Dichiarare

Aggiunge un nuovo fatto all'elenco dei fatti (l'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 Ritrattare

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 Modificare

Ritira alcuni fatti dall'elenco dei fatti e ne dichiara uno nuovo con alcune modifiche. Le modifiche vengono 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 Duplicare

Aggiunge un nuovo fatto all'elenco dei fatti utilizzando un fatto esistente come modello e aggiungendo 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

Questo è il normale processo per eseguire un **KnowledgeEngine**.

1. La classe deve essere istanziata, ovviamente.
2. Il metodo di **ripristino** deve essere chiamato:
   - Questo dichiara il fatto speciale $InitialFact$. Necessario per il corretto funzionamento di alcune regole.
   - Dichiara tutti i fatti prodotti dai metodi decorati con @$DefFacts$.
3. È necessario chiamare il metodo **run**. Questo avvia il ciclo di esecuzione.

2.8 Ciclo di esecuzione

In uno stile di programmazione convenzionale, il punto di partenza, il punto di arresto e la sequenza delle operazioni sono definiti esplicitamente dal programmatore. Con Experta, il flusso del programma non ha bisogno di essere definito in modo così esplicito. La conoscenza (Regole) e i dati (Fatti) sono separati e il KnowledgeEngine viene utilizzato per applicare la conoscenza ai dati.
Il ciclo di esecuzione di base è il seguente:

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

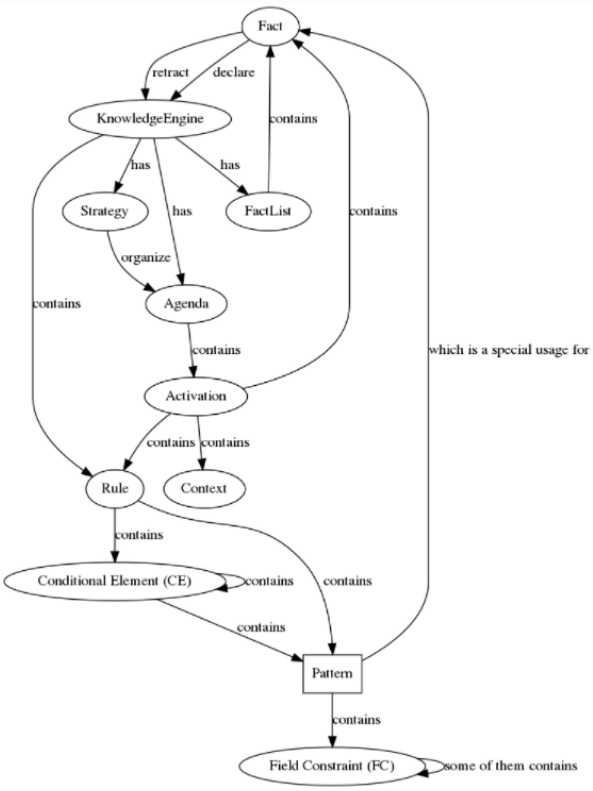
2.9 Differenza tra DefFacts e declare

Entrambi sono usati per dichiarare fatti sull'istanza del motore, ma:

- **declare** aggiunge i fatti direttamente alla memoria di lavoro.
- I generatori dichiarati con $DefFacts$ vengono chiamati dal metodo **reset** e tutti i fatti prodotti vengono aggiunti alla memoria di lavoro utilizzando require .

2.10 Diagramma

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



2.11 Esempio di come funziona il nostro sistema esperto di diagnostica sul diabete

Ecco come funziona il nostro sistema esperto:
Sono poste delle domande all'utente, e in base alla risposta dell'utente, il sistema dichiara dei **facts**, ovvero delle **proposizioni** su cui andrà poi a ragionare, così sarà in grado di dare una diagnosi sul diabete.

2.12 CSP

Molti problemi nell'ambito dell'Intelligenza Artificiale sono classificabili come Problemi di Soddisfacimento di Vincoli (Constraint Satisfaction Problem o CSP); Formalmente, un CSP può essere definito su un insieme finito di variabili $(X_1, X_2, \ldots, X_n)$ i cui valori appartengono a domini finiti di definizione $(D_1, D_2, \ldots, D_n)$ e su un insieme di vincoli (constraints) $(C_1, C_2, \ldots, C_n)$. Un vincolo su un insieme di variabili è una restrizione dei valori che le variabili possono assumere simultaneamente. Concettualmente, un vincolo può essere visto come un insieme che contiene tutti i valori che le variabili possono assumere contemporaneamente: un vincolo tra k variabili $C(X_{i_1}, X_{i_2}, \ldots, X_{i_k})$, è un sottoinsieme del prodotto cartesiano dei domini delle variabili coinvolte $D_{i_1}, D_{i_2}, \ldots, D_{i_k}$ che specifica quali valori delle variabili sono compatibili con le altre. Questo insieme può essere rappresentato in molti modi, per esempio per mezzo di matrici, equazioni, disuguaglianze o relazioni. Si definisce grado della variabile il numero di vincoli a cui è sottoposta.

Un problema di soddisfacimento di vincoli presuppone un assegnamento iniziale, ovvero un insieme di variabili già vincolate. L'assegnamento iniziale può anche essere vuoto. La risoluzione del problema prosegue estendendo l'assegnamento iniziale, ovvero assegnando via via valori alle variabili ancora libere. La soluzione di un CSP è un assegnamento completo e coerente di valori alle variabili (ovvero un assegnamento che soddisfi tutti i vincoli e non lasci variabili libere), ottenuto estendendo l'assegnamento iniziale.

La libreria utilizzata da noi, di nome **constraint**, ci ha permesso di realizzare un semplice CSP (è un caso molto semplice, non si tiene conto di altri utenti) in grado di mostrare la disponibilità degli

studi convenzionati, nel caso in cui il sistema preveda la prescrizione delle analisi.
Il funzionamento è il seguente:

- Alla base vi è una sottoclasse di Problem (classe già definita in **constraint**, che modella un CSP).
- Vengono aggiunte variabili con proprio dominio associato in maniera esplicita.
- In base alle risposte dell'utente, il sistema decide se prescrivere o meno delle analisi.
- Se il sistema decide di prescrivere delle analisi, allora qui entra in gioco il CSP.
- Il CSP è relativo agli orari di apertura del laboratorio delle analisi.
- Per esempio: laboratorio per l'insulina, fa le analisi dalle ore 8 alle 14.
- Il sistema quindi indica i possibili orari a cui l'utente può presentarsi per sottoporsi alle analisi.

2.12.1 Un possibile esempio:

```
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
```

Figura 2.12.1: **Esempio:** Output del programma

```
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')
```

Figura 2.12.2: **In base alle risposte dell'utente** date nell'esempio, l'agente ragiona con i seguenti fatti

2.13 Ontologie

In informatica, un'ontologia è una rappresentazione formale, condivisa ed esplicita di una concettualizzazione di un dominio di interesse. Più nel dettaglio, si tratta di una teoria assiomatica del primo ordine esprimibile in una logica descrittiva.

Il termine ontologia formale è entrato in uso nel campo dell'intelligenza artificiale e della rappresentazione della conoscenza, per descrivere il modo in cui diversi schemi vengono combinati in una struttura dati contenente tutte le entità rilevanti e le loro relazioni in un dominio. I programmi informatici possono poi usare l'ontologia per una varietà di scopi, tra cui il ragionamento induttivo, la classificazione, e svariate tecniche per la risoluzione di problemi. L'ontologia è stata creata mediante il tool **protege** (figura 2.13.1), e la lettura avviene tramite la libreria python **Owlready2** (figura 2.13.2).

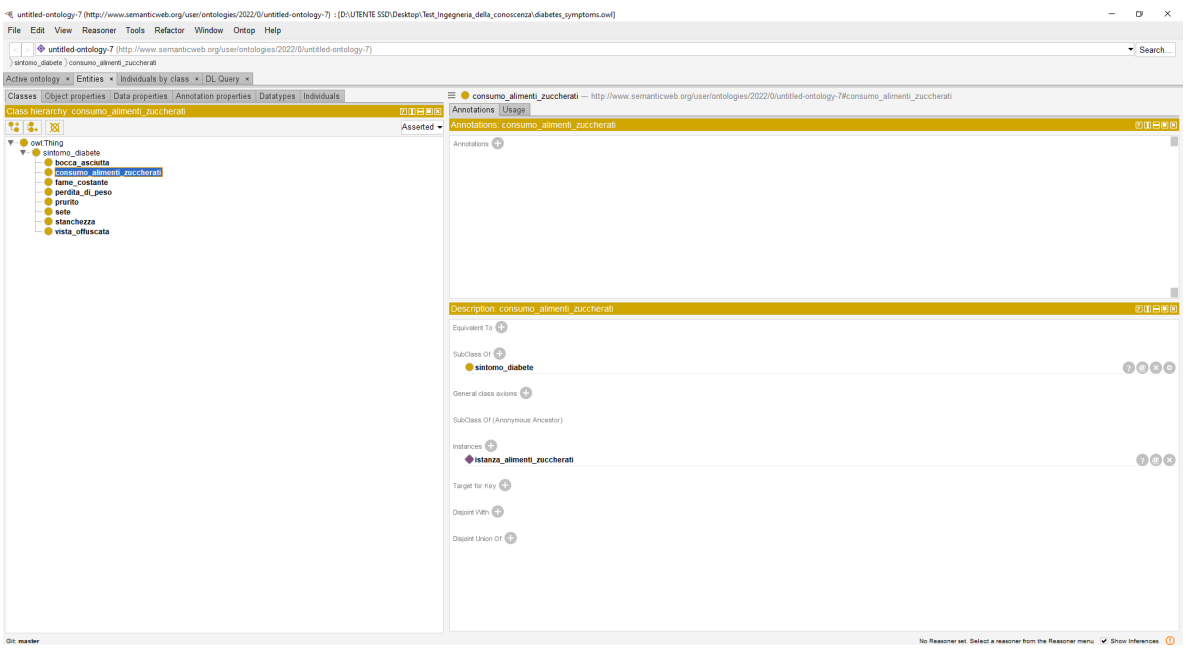


Figura 2.13.1: tool protege

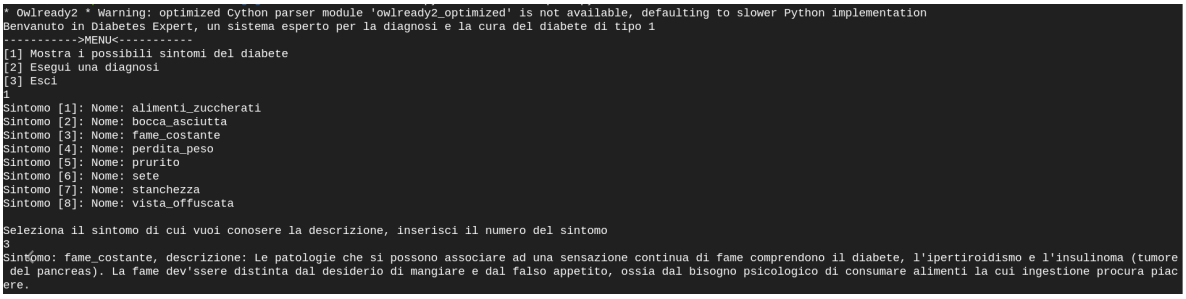


Figura 2.13.2: Dall'ontologia viene creato un dizionario con i sintomi e la relativa descrizione

3 Algoritmi di classificazione

Abbiamo deciso di utilizzare il seguente dataset:
https://www.kaggle.com/uciml/pima-indians-diabetes-database
Il dataset presenta varie features:

- numero di gravidanze (questa feature viene eliminata nella fase di pre-processing).
- Concentrazione di glucosio plasmatico a 2 ore in un test di tolleranza al glucosio orale
- Diastolic blood pressure (mm Hg)
- Spessore della piega cutanea del tricipite (mm)
- Insulina sierica a 2 ore (mu U/ml)
- Body mass index (weight in kg/(height in m)^2)
- Funzione pedigree del diabete
- Età
- Variabile di classe (0 o 1) 268 di 768 sono 1, gli altri sono 0

Gli algoritmi di classificazione da noi implementati, sono i seguenti:

- Regressione logistica
- Albero di decisione
- K-nearest neighbor

3.1 Albero di decisione

Un Albero di Decisione (Decision Tree) è un metodo molto semplice ed efficace per realizzare un classificatore e l'addestramento degli alberi di decisione è una delle tecniche attuali di maggior successo. Un albero di decisione è un albero di classificatori (Decision Stump) dove ogni nodo interno è associato ad una particolare "domanda" su una caratteristica (feature). Da questo nodo dipartono tanti archi quanti sono i possibili valori che la caratteristica può assumere, fino a raggiungere le foglie che indicano la categoria associata alla decisione. Particolare attenzione normalmente è posta per i nodi di decisione binaria.

3.2 Regressione logistica

La regressione logistica è un modello statistico (modello logit) usato negli algoritmi di classificazione del machine learning per ottenere la probabilità di appartenenza a una determinata classe. L'algoritmo di classificazione basato sulla regressione logistica è del tipo ML supervisionato. Si basa sull'utilizzo della funzione logistica (sigmoid) che converte i valori reali in un valore compreso tra 0 e 1.

Nella fase di addestramento l'algoritmo riceve in input un dataset di training composto da N esempi. Ogni esempio è composto da m attributi X e da un'etichetta y che indica la corretta classificazione. Nel nostro caso il dataset viene prima pre-processato creando l'oggetto di classe *diabetes_data*, tramite questa classe si elimina la feature relativa alle gravidanze, successivamente definisce tramite il metodo *get_training_data*(), che restituisce i dati relativi a input features e target features. Nel nostro caso la target feature è una sola, cioè **Outcome**. Questa feature indica se un soggetto è malato o meno di diabete.

L'algoritmo individua una vettore dei pesi W da associare al vettore degli attributi X_m degli esempi, in modo tale da massimizzare la percentuale di risposte corrette (o minimizzare quelle sbagliate). La **combinazione lineare z** dei pesi L per gli attributi X fornisce una risposta del sistema per ogni esempio del training dataset.

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

Nella regressione logistica la combinazione lineare z è l'argomento della funzione logistica che lo traduce in un valore compreso tra 0 e 1.

$$f(z) = [0, 1]$$

Il risultato della funzione logistica è usato come funzione di attivazione dei nodi della rete neurale.

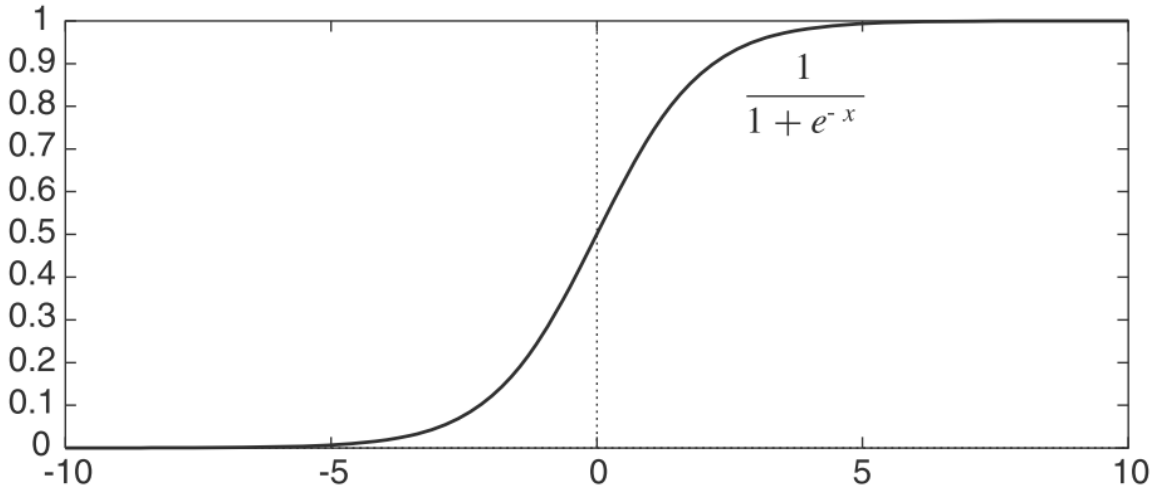


Figure 7.9: The sigmoid or logistic function

Ad esempio, se viene fissato come parametro di soglia il valore 0.6, quando $f(z) > 0.6$ il nodo viene attivato. Viceversa è disattivo.

$$\begin{cases} 1 & if \quad f(z) > 0.6 \\ 0 & else \end{cases}$$

Per scegliere la distribuzione W* migliore l'algoritmo utilizza un'altra funzione di massimizzazione della probabilità L.

$$max(L(W) = \sum_{i=1}^{N} y_i log f(z_{(i)}(x) + (1 - y_i) log(1 - f(z_{(i)}))$$

La sommatoria confronta tutte le risposte del modello con le risposte corrette calcolando la log-probabilità.

Al termine dell'addestramento l'algoritmo produce un modello, utilizzabile per classificare qualsiasi altro esempio non compreso nel training set.

3.3 K-nearest neighbor

Il k-nearest neighbors, abbreviato in K-NN, è un algoritmo utilizzato nel riconoscimento di pattern per la classificazione di oggetti basandosi sulle caratteristiche degli oggetti vicini a quello considerato. In entrambi i casi, l'input è costituito dai k esempi di addestramento più vicini nello spazio delle funzionalità. L'output dipende dall'utilizzo di k-NN per la classificazione o la regressione.

- Nella **classificazione k-NN**, l'output è un'appartenenza a una classe. Un oggetto è classificato da un voto di pluralità dei suoi vicini, con l'oggetto assegnato alla classe più comune tra i suoi k vicini più vicini (k è un numero intero positivo, tipicamente piccolo). Se k = 1, l'oggetto viene semplicemente assegnato alla classe di quel singolo vicino più prossimo.
- **Nella regressione k-NN**, l'output è il valore della proprietà per l'oggetto. Questo valore è la media dei valori di k vicini più vicini.

  Un oggetto è classificato in base alla maggioranza dei voti dei suoi k vicini. k è un intero positivo tipicamente non molto grande. Se k=1 allora l'oggetto viene assegnato alla classe del suo vicino. In un contesto binario in cui sono presenti esclusivamente due classi è opportuno scegliere k dispari per evitare di ritrovarsi in situazioni di parità.

  Questo metodo può essere utilizzato per la tecnica di regressione assegnando all'oggetto la media dei valori dei k oggetti suoi vicini.

  Considerando solo i voti dei k oggetti vicini c'è l'inconveniente dovuto alla predominanza delle classi con più oggetti. In questo caso può risultare utile pesare i contributi dei vicini in modo da dare, nel calcolo della media, maggior importanza in base alla distanza dall'oggetto considerato.

3.3.1 Fase di apprendimento dell'algoritmo

Lo spazio viene partizionato in regioni in base alle posizioni e alle caratteristiche degli oggetti di apprendimento. Questo può essere considerato come l'insieme d'apprendimento per l'algoritmo, anche se esso non è esplicitamente richiesto dalle condizioni iniziali.

3.3.2 Calcolo della distanza

Ai fini del calcolo della distanza gli oggetti sono rappresentati attraverso vettori di posizione in uno spazio multidimensionale. Di solito viene usata la distanza euclidea, ma anche altri tipi di distanza sono ugualmente utilizzabili, ad esempio la distanza Manhattan. Nel caso in cui si debbano manipolare stringhe e non numeri si possono usare altre distanze quali ad esempio la distanza di Hamming. L'algoritmo è sensibile alla struttura locale dei dati.

3.3.3 Fase di classificazione

Un punto (che rappresenta un oggetto) è assegnato alla classe C se questa è la più frequente fra i k esempi più vicini all'oggetto sotto esame, la vicinanza si misura in base alla distanza fra punti. I vicini sono presi da un insieme di oggetti per cui è nota la classificazione corretta. Nel caso della regressione per il calcolo della media (classificazione) si usa il valore della proprietà considerata.

3.4 Implementazione dei modelli di apprendimento supervisionato in sklearn

La libreria da noi scelta per l'implementazione di questi algoritmi è sklearn, che ci ha permesso tramite le classi e i metodi presenti di implementare, allenare e testare i modelli di apprendimento supervisionato.

1. Nel main (main.py), per ogni modello si istanzia la relativa classe.
2. Per ogni modello viene eseguita la predizione.
3. Successivamente viene mostrata la **matrice di confusione** (figura 1).
4. Vengono quindi calcolate le metriche di accuratezza, precision, recall, F1.
5. Il calcolo di questa metrica avviene tramite l'ausilio di metodi messi a disposizione da Sklearn.

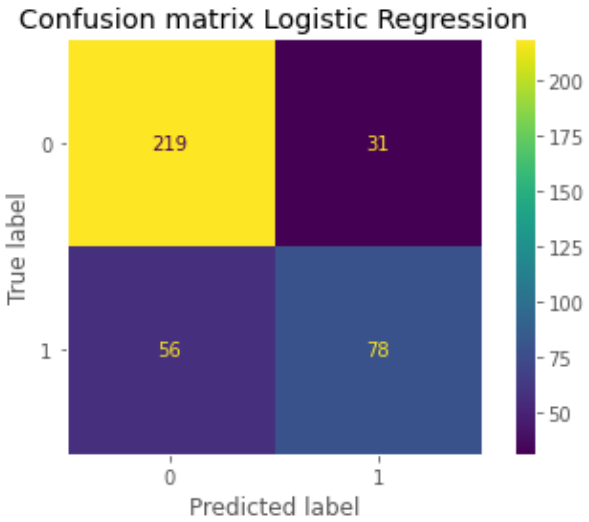


Figura 3.4.1: in questo caso abbiamo 219 casi true-positive, ovvero che appartengono alle persone la cui predizione sul diabete ha dato esito positivo. Abbiamo 78 true negative, quindi il modello ha predetto correttamente che 78 soggetti non presentano il diabete, mentre il false-positive(31) ed il false-negative(56) non sono stati predetti correttamente.

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

Figura 3.4.2: le metriche calcolate in figura (3.4.1) sono le seguenti

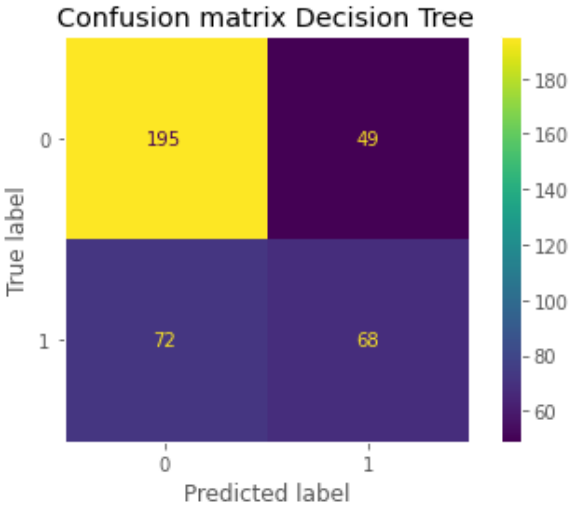


Figura 3.4.3: in questo caso abbiamo 195 casi true-positive, ovvero che appartengono alle persone la cui predizione sul diabete ha dato esito positivo. Abbiamo 68 true negative, quindi il modello ha predetto correttamente che 68 soggetti non presentano il diabete, mentre il false-positive(49) ed il false-negative(72) non sono stati predetti correttamente.

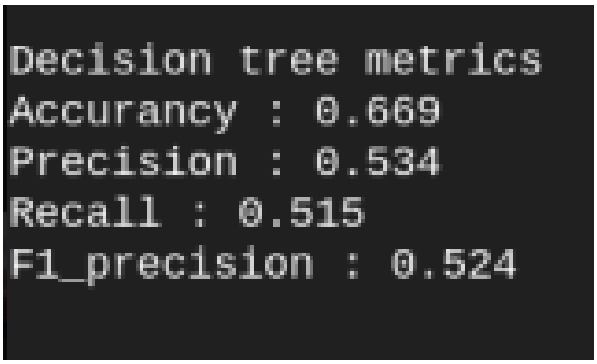


Figura 3.4.4: le metriche calcolate in figura sono le seguenti

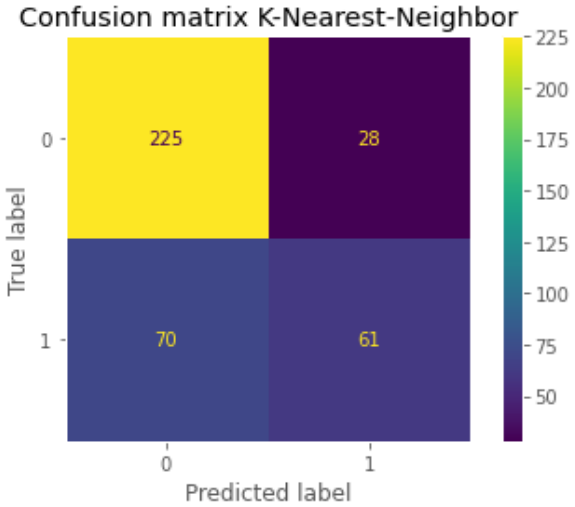


Figura 3.4.5: in questo caso abbiamo 225 casi true-positive, ovvero che appartengono alle persone la cui predizione sul diabete ha dato esito positivo. Abbiamo 61 true negative, quindi il modello ha predetto correttamente che 61 soggetti non presentano il diabete, mentre il false-positive(28) ed il false-negative(70) non sono stati predetti correttamente.

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

Figura 3.4.6: le metriche calcolate in figura sono le seguenti

Figura 3.4.7: analisi delle metriche al variare del parametro preso in input.

In figura 3.4.7, nel caso della regressione logistica, il parametro che viene cambiato è il numero di iterazioni fatte dall'algoritmo di classificazione. Vedendo i valori della regessione logistica, un valore alto dell'accuratezza è ottenuto passando in input circa 700 iterazioni, oppure 1200, la precisione, si ottiene intorno ai 300, la F1 si ottiene intorno alle 1900 iterazioni circa.

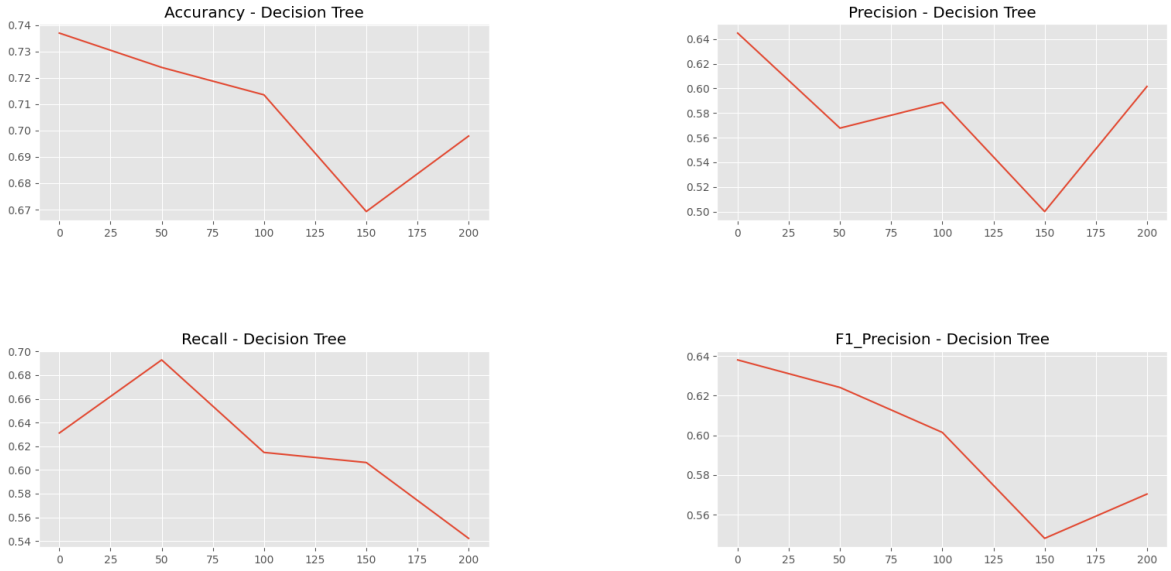


Figura 3.4.8: Nell'albero di decisione invece, varia la profondità. Le performance crollano considerando una profondità pari a 150.

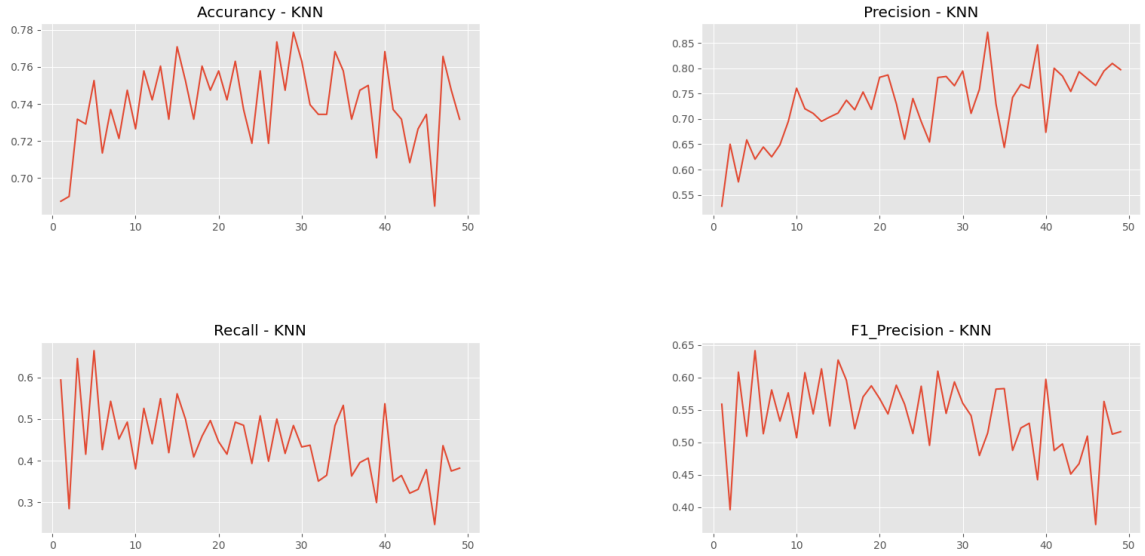


Figura 3.4.9: In questo caso variano i neighbors. Le performance sono molto altalenanti

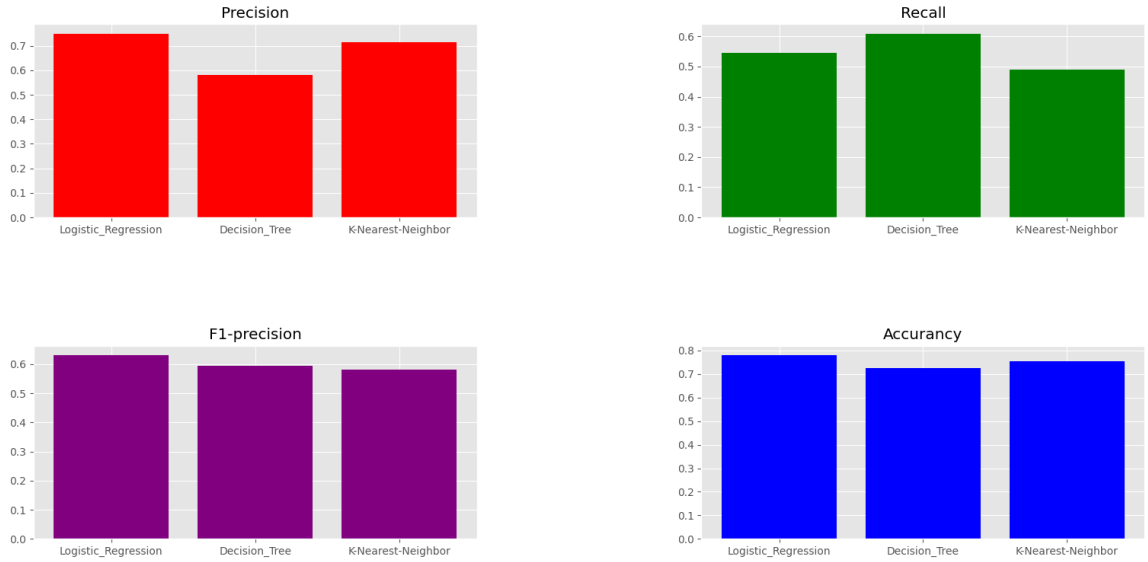


Figura 3.4.10: In generale, come prestazioni i modelli si equivalgono, ma la regressione logistica sembra dare prestazioni leggermente migliori.