# INGEGNERIA DELLA CONOSCENZA

Progetto a cura di Pompeo Francesco Lippolis
(matricola n° 597996)

## Introduzione

Il progetto che ho voluto sviluppare, i cui file sono disponibili sul repository github, prende spunto dalle competizioni online sul Machine Learning come Kaggle, dove si cerca di costruire il modello che meglio predice la classe di appartenenza di esempi, non visti prima, appartenenti ad un dataset dato.
Ho cercato un dataset adeguato al compito, imbattendomi nel sito del Machine Learning Repository dell'Università della California, Irvine dove il dataset sull'insorgenza del diabete ha attirato la mia attenzione in quanto è un problema presente nella mia famiglia.

Ho sviluppato una serie di classificatori addestrati sul dataset:

- Regressione Lineare
- Regressione Logistica
- Albero di decisione
- Rete Neurale
- Support Vector Machine
- Random Forest
- ADAptive BOOSTing
- K-Nearest Neighbors
- Naive Bayes

Li ho successivamente valutati e ho scelto il modello più adeguato al compito, tra quelli sviluppati. Il vincitore è stato utilizzato per sviluppare una semplice applicazione che predice il rischio di insorgenza del diabete.

Questa applicazione tuttavia richiede un carico cognitivo enorme all'utente, che deve conoscere con certezza tutti i valori di ogni feature per fare una predizione accurata.

La struttura dell'Albero di Decisione permette, invece, di fare una predizione conoscendo un numero di feature al massimo pari alla profondità dell'albero.

Ho sviluppato un programma in AILog che utilizza la struttura dell'albero di decisione per determinare la probabilità che un individuo abbia il diabete. Utilizzando atomi askable fa domande all'utente solo sulle condizioni effettivamente necessarie al ragionamento.

Questo programma non supera un problema di fondo: e se l'utente non ha informazioni su una determinata feature? È stata quindi sviluppata un'ultima applicazione, che predice la probabilità del diabete anche in caso di feature non conosciute tramite le capacità di ragionamento probabilistico del primo ordine del linguaggio.

## Software utilizzato

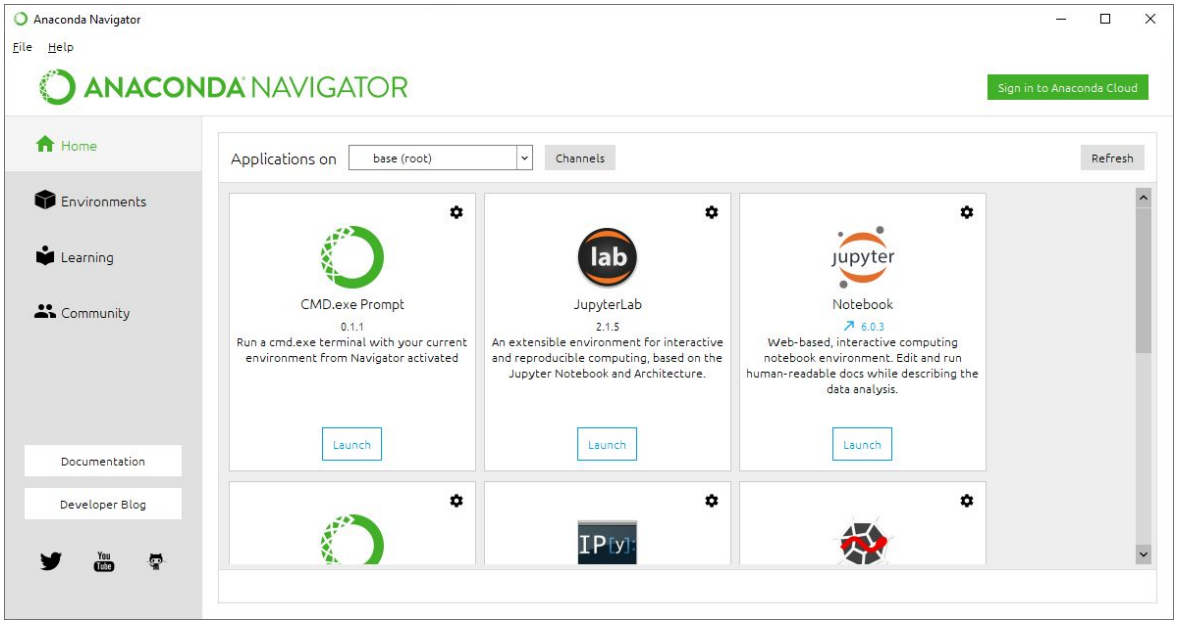
Per sviluppare i vari classificatori, ho usato l'applicazione Jupyter in quanto consente di creare e condividere documenti contenenti codice, equazioni, grafici e testo descrittivo.
Il codice utilizzato è in Python, con alcune istruzioni dell'estensione IPython del linguaggio.
Le librerie sono pandas, numpy, matplotlib, scikit-plot, IPython, scikit learn, tensorflow 2.0 e keras tuner.

L'applicazione che predice il rischio di insorgenza del diabete utilizzando il modello considerato migliore tra quelli sviluppati (file 'applicazione demo.py') è stata sviluppata in Python.
Ho sviluppato la GUI dell'applicazione usando la libreria PySimpleGUI.

Le applicazioni che utilizzano l'Albero di Decisione per fare inferenza (file 'diabetes.ailog' e 'prob_diabetes.ailog' nella cartella AILog) sono state implementate nel linguaggio AILog.
Il meta-interprete AILog (file ailog2.pl nella cartella AILog) necessita dell'interprete Prolog SWI-Prolog.

È possibile visualizzare il lavoro effettuato nello sviluppo dei vari modelli senza installare alcun software: online visualizzando il file nel repository oppure tramite l'export html ICon project.html.
Il notebook contiene codice e risultati dell'esecuzione dello stesso, il tutto brevemente commentato.

Per visionare la versione interattiva del notebook (ICon project.ipynb) è necessario installare Jupyter sul proprio PC. Il modo più semplice è installando la distribuzione Anaconda, che contiene Jupyter, Python e molte delle librerie utilizzate. Una volta installato si potrà far partire il server Jupyter notebook dall'applicazione Anaconda Navigator:



La maggior parte delle librerie sono pre-installate o installabili tramite Anaconda Navigator

Le librerie mancanti possono essere installate tramite il package installer for python (pip):

Pandas: pip install pandas
Numpy: pip install numpy
MatPlotLib: python -m pip install -U matplotlib
Scikit-plot: pip install scikit-plot
IPython: pip install ipython
Scikit learn: pip install -U scikit-learn
Tensorflow 2: pip install tensorflow
Keras Tuner: pip install -U keras-tuner
PySimpleGUI: pip install pysimplegui

# Preparazione del dataset

Per permettere alla maggior parte dei classificatori di poter operare sul dataset, ho effettuato la codifica delle etichette (Maschio, Femmina; Sì, No; Positivi, Negativi) in 0 e 1.
Normalmente etichette di categoria andrebbero codificate con il one-hot encoding; tuttavia, essendo i valori di etichette binari per ogni colonna, ho preferito codificare i valori della colonna come booleano rispetto alla relazione; ad esempio maschio(individuo): 1 se l'individuo è uomo, 0 se l'individuo è donna; questo metodo di descrizione risulta più compatto.
Ho successivamente discretizzato l'età in fasce d'età: ogni individuo è classificato secondo l'età più vicina in decadi. Questo è utile per limitare la presenza di outliers: se nel dataset ogni individuo con, per esempio, 24 anni d'età ha il diabete, un classificatore potrebbe essere tentato di classificare ogni 24enne come positivo, quando ciò è, ovviamente, falso.

Ho poi normalizzato il dataset, portando l'età su valori tra 0 e 1, in quanto determinati modelli si comportano male se addestrati su valori con scale diverse.

## Fase di addestramento

Le fasi di suddivisione del dataset e di addestramento sono state rese riproducibili tramite la variabile random_state, passata come parametro.
Ho suddiviso il dataset in dati di training e di test con proporzione 4:1, considerando che un numero più alto di dati di training avrebbero reso il test set troppo piccolo.

La maggior parte dei modelli è stata addestrata usando un metodo di Cross validation (Lo Stratified K-Fold Cross Validation) per poter mettere a punto i parametri del modello.
L'unica eccezione è il Neural Network, addestrato sempre dividendo il set in training e validation per la messa a punto dei parametri, ma senza uno schema di CV (non supportato direttamente da Keras Tuner).
I parametri generalmente si riferiscono o alla struttura interna del classificatore o a parametri di regolarizzazione. Descrizioni specifiche per ogni modello sono presenti nel notebook.

## Raccolta di metriche

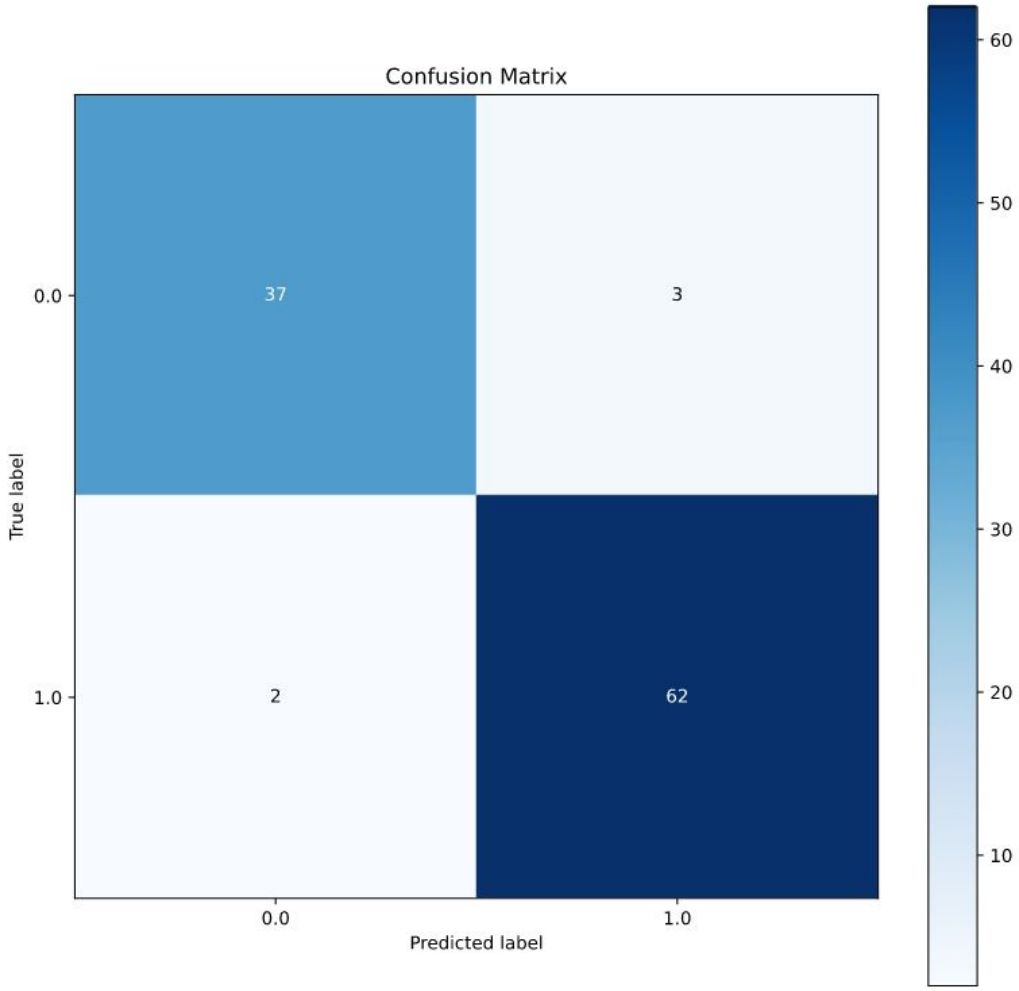
Per ogni modello sono state raccolte le seguenti metriche: sul training set è stata calcolata l'accuratezza (per valutare un eventuale overfitting dei modelli) invece sul test set vengono calcolati accuratezza, precisione e richiamo e i valori necessari a mostrare graficamente la matrice di confusione e la curva di precisione-richiamo.

## Scelta del modello

Ho valutato la performance di ogni modello sul test set (non usato nell'addestramento)

Il modello che ho scelto tra i vari addestrati è la SVM, con le seguenti metriche:

| | Train accuracy | Test accuracy | Precision | Recall |
|---|---|---|---|---|
| **Support Vector Machine** | 99.52 | 95.19 | 95.38 | 96.88 |




L'ho scelto in quanto è il classificatore con la minor presenza di Falsi Negativi (alto richiamo)

Ogni falso negativo corrisponde ad un diabetico a cui non viene diagnosticata la malattia, cosa che vogliamo decisamente evitare.

L'ADAptive BOOSTing presenta lo stesso numero di Falsi Negativi, tuttavia ho scelto la SVM in quanto ha un numero decisamente più basso di Falsi Positivi; una metrica importante, seppur secondaria rispetto ai Falsi Negativi, in quanto un falso positivo corrisponde ad un esame di

accertamento prescritto in più, non grave quanto una malattia non diagnosticata, ma certamente da evitare.

## Applicazioni sviluppate

Ho sviluppato in seguito una serie di semplici applicazioni per la predizione dell'insorgere del diabete.
La prima, in Python, consiste in un semplice form per la raccolta di tutte le feature dell'individuo, classificato in seguito utilizzando la SVM.

Tuttavia questa applicazione richiede un carico cognitivo particolarmente elevato all'utente: l'utente infatti deve sapere l'esatto valore di ogni feature o condizione posseduta dal paziente per poter fare una classificazione corretta.

Ho deciso, quindi, di sfruttare la struttura intrinseca dell'Albero di decisione per permettere all'utente di fare previsioni specificando il valore di un numero limitato di feature (dipendenti dalla profondità del ramo percorso).
Ho sviluppato questa applicazione in AILog, in modo che potesse ragionare direttamente sulla struttura dell'albero e perché l'interprete per il linguaggio AILog, meta-interprete per il Prolog, lo espande con supporto per gli atomi askable: in questo modo l'applicazione interroga l'utente solo sulle condizioni del paziente necessarie al ragionamento.

Le decisioni di progetto sono state due:
La prima riguardava la scelta della query iniziale.
La query più semplice possibile in questa applicazione è ask has_diabetes(Individuo)., dove Individuo nella relazione has_diabetes rappresenta l'individuo che si vuole classificare.

Tuttavia questa rappresentazione ha un problema: può dare solo valori di verità (true o false) alla relazione. Non permette quindi di rappresentare al meglio stime puntuali con incertezza.

Ho scelto, allora, di utilizzare una tecnica comune nei programmi logici, quella di avere una variabile libera nella query, che sarà del tipo:

ask has_diabetes(Individuo, P).

La variabile P rappresenta la probabilità che l'individuo ha il diabete.
Al termine del ragionamento la relazione si unificherà con un fatto, rendendo vera la relazione ed effettuando il binding della variabile P tramite il Most General Unifier calcolato, dando all'utente il valore ricercato di P.

La seconda scelta riguarda il metodo di rappresentazione della struttura dell'albero.
Avrei potuto rappresentare l'albero come una serie di regole, una per ogni ramo, del tipo:

has_diabetes(I, P) ← has(I, polyuria) ∧ has(I, polydipsia) ∧ P=1.

Tuttavia ritengo che questa soluzione non sia opportuna, in quanto richiede al progettista di riscrivere l'intera base di conoscenza per ogni modifica all'albero di decisione, e al motore di inferenza di ragionare ripetutamente sulle stesse condizioni.
Ho optato quindi per rappresentare l'albero tramite simboli di funzione:

bt(Etichetta, Ramo_sinistro, Ramo_destro) e leaf(Etichetta).

Il ragionamento avviene in maniera ricorsiva sulla struttura dell'albero:
Passo base:

has_diabetes(I,P,leaf(P)).

Passo Iterativo, con condizione alla foglia vera:

has_diabetes(I,P,bt(C,_,T)) ← has(I,C) ∧ has_diabetes(I,P,T).

Passo iterativo, con condizione alla foglia falsa:

has_diabetes(I,P,bt(C,F,_)) ← ~ has(I,C) ∧ has_diabetes(I,P,F).

Semanticamente: una persona ha probabilità P di avere il diabete per un albero decisionale se l'albero è costituito dal singolo nodo foglia etichettato con P.
Alternativamente, una persona ha probabilità P di avere il diabete per un albero decisionale se:

- L'albero decisionale possiede sottoalberi
- La condizione indicata dalla foglia è vera (o falsa)
- Il sottoalbero destro (o sinistro) predice similmente la stessa probabilità P.

È possibile testare l'applicazione aprendo l'interprete AILog (file ailog2.pl) dentro SWI-Prolog e caricando il programma con l'istruzione:

load 'diabetes.ailog'.

Un esempio di query effettuabile è:

ask has_diabetes(giovanni,P).

Tuttavia dal testing è sorta una problematica: rispondere alle domande con "unknown" ha lo stesso effetto che rispondere negativamente.
Questo è corretto in accordo con l'assunzione di mondo chiuso e di negazione come fallimento, ma non in linea con le aspettative dell'utente (o del progettista): il sistema non sa gestire l'incertezza.

Per poter gestire l'incertezza è stata sviluppata un'altra applicazione che sfrutta capacità di ragionamento probabilistico del primo ordine del

linguaggio AILog per predire la probabilità di insorgenza del diabete anche in caso di feature non conosciute.

L'albero utilizzato in questa versione è stato annotato con varie probabilità condizionate.
La probabilità annotata sulle foglie è la seguente (la variabile D indica la presenza di diabete):
$P( D = \textit{vero} \mid F_{1j}=v_{1j} \wedge F_{2j}=v_{2j} \wedge \ldots \wedge F_{nj}=v_{nj})$

Dove $F_{ij}$ è l'i-esima feature del j-esimo ramo, e il rispettivo $v_{ij}$ è il valore di verità per la feature i sul ramo j.

La probabilità annotata sul nodo della prima feature è a priori:
$P(F_{1j} = \textit{vero}) = 0.51 \quad P(F_{1j} = \textit{falso}) = 0.49$

La probabilità annotata sul nodo della seconda feature è condizionato alla prima:
$P(F_{2j} = \textit{vero} \mid F_{1j} = \textit{vero}) = 0.76$ $P(F_{2j} = \text{falso} \mid F_{1j} = \textit{vero}) = 0.24$
$P(F_{2j} = \text{vero} \mid F_{1j} = \textit{falso}) = 0.77$ $P(F_{2j} = \text{falso} \mid F_{1j} = \textit{falso}) = 0.23$

E così via.
Per la chain rule:
$P(D = \text{vero} \mid F_{1j}=v_{1j} \wedge \ldots \wedge F_{nj}=v_{nj}) * P(F_{1j}=v_{ij}) * P(F_{2j}=v_{nj} \mid F_{1j}=v_{1j}) * \ldots$
$\ldots * P(F_{nj}=v_{nj} \mid F_{1j}=v_{1j} \wedge \ldots \wedge F_{n-1j}=v_{n-1j}) = P(D = \textit{vero} \wedge F_{1j}=v_{1j} \wedge \ldots \wedge F_{nj}=v_{nj})$

$P(D = \textit{vero}) = P((D = \textit{vero} \wedge F = \textit{vero}) \vee (D = \textit{vero} \wedge F = \textit{falso})) =$
$= P(D = \textit{vero} \wedge F = \textit{vero}) + P(D = \textit{vero} \wedge F = \textit{falso})$

Esteso ad ogni feature di ogni ramo:
$P(D = \textit{vero}) = \sum_j P(D = \text{vero} \wedge F_{1j} = v_{1j} \wedge \ldots \wedge F_{nj} = v_{nj}) =$
$\sum_j (P(D = \text{vero} \mid F_{1j}=v_{1j} \wedge \ldots \wedge F_{nj}=v_{nj}) * P(F_{1j}=v_{ij}) * P(F_{2j}=v_{nj} \mid F_{1j}=v_{1j}) * \ldots$
$\ldots * P(F_{nj}=v_{nj} \mid F_{1j}=v_{1j} \wedge \ldots \wedge F_{n-1j}=v_{n-1j}))$

Si noti che la probabilità di una feature in presenza di osservazione dell'utente diventa:

$P(F_{ij}=v_{ij}|F_{1j}=v_{1j} \wedge \ldots \wedge F_{i-1j}=v_{i-1j} \wedge F_{ij}=obs) = P(F_{ij} = v_{ij} | F_{ij}=obs) \Rightarrow$
1 se obs = $v_{ij}$
0 se obs = $\neg v_{ij}$

È importante notare come il programma AILog non abbia codificato queste equazioni al suo interno, ma arriva ad un risultato analogo solo tramite il ragionamento logico e una interpretazione π che mappa gli atomi su [0,1].

La struttura logica del programma è rimasta pressoché invariata rispetto al programma precedente, tuttavia il predicato has_diabetes perde il termine P in quanto la probabilità diventa parte del valore di verità della relazione. Il predicato has, invece, acquista un termine P per tenere traccia della probabilità condizionata di ogni condizione al nodo.

È possibile testare l'applicazione aprendo l'interprete AILog (file ailog2.pl) dentro SWI-Prolog e caricando il programma con l'istruzione:

load 'prob_diabetes.ailog'.

Un esempio di query effettuabile è:

predict has_diabetes(giovanni).