Università degli Studi di Urbino Carlo Bo

Implementazione di Regressione Lineare e K-Nearest Neighbors in Haskell e Prolog

Corso: Programmazione Logica e Funzionale

17 agosto 2023

Docente
Prof. Marco Bernardo

Corsista
Andrea De Lorenzis, 308024

Indice

1 Specifica del problema

Linear regression e K-Nearest neighbors (KNN) sono due algoritmi che trovano applicazione in diversi campi, tra cui statistica, analisi di dati e intelligenza artificiale. La prima consente di prevedere il valore di una variabile sconosciuta mediante un modello basato su un'equazione lineare. La seconda è una tecnica utilizzata per la classificazione di oggetti basandosi sulle caratteristiche degli oggetti vicini a quello considerato. L'obiettivo di questo progetto è quello di implementare le due tecniche nei linguaggi di programmazione Haskell e Prolog.

2 Analisi del problema

2.1 Dati di ingresso del problema

I dati di ingresso del problema sono inseriti dall'utente e consistono in un insieme di punti bidimensionali per allenare e valutare i modelli. L'utente inizialmente deve fornire un valore numerico per scegliere l'operazione da svolgere (1 - Linear regression, 2 - KNN, 3 - Terminare il programma). Una volta scelta una delle prime due operazioni, all'utente è richiesto di inserire un insieme di punti bidimensionali per allenare il modello. Successivamente, potrà inserire un ulteriore elemento (valore di x oppure punto) da valutare sul modello allenato, ottenendo poi il risultato. In particolare, i dati di ingresso per la regressione lineare sono:

- Un insieme di punti bidimensionali, ognuno dei quali è costituito da una coordinata x e una coordinata y

$$(x_1\ y_1),\ (\ x_2\ y_2)\ \ldots$$

- Un valore di x da valutare sul modello lineare allenato.

I dati di ingresso per il KNN sono:

- Un insieme di punti bidimensionali etichettati, ognuno dei quali è costituito da una coordinata x, una coordinata y e una classe o etichetta in formato testuale che rappresenta la categoria di appartenenza del punto

$$(x_1\ y_1\ \text{label1}),\ (x_2\ y_2\ \text{label2})\ \ldots$$

- Il valore k di vicini da valutare per il punto.
- Il punto di test, senza etichetta, da classificare tramite KNN.

2.2 Dati di uscita del problema

I dati di uscita per la regressione lineare sono:

- Coefficienti della retta di regressione, ossia il valore della pendenza (m) e dell'intercetta (b), i quali definiscono l'equazione della retta.
- Risultati predittivi stimando nuovi valori di y per determinati valori di x sulla base del modello di regressione lineare ottenuto.

Invece, l'operazione di KNN produce in uscita:

- Classificazione del punto di test, al quale verrà assegnata una classe sulla base del numero di occorrenze di quella classe nei vicini.
- I k punti più vicini al punto di test, comprensivi di etichetta, informazione utile al fine di comprendere le ragioni della classificazione svolta sul punto di test.

2.3 Relazioni intercorrenti

Nella regressione lineare si cerca di modellare la relazione tra la variabile indipendente x e la variabile dipendente y attraverso una retta. Pertanto la relazione che intercorre tra i dati di ingresso ed uscita per questa operazione è:

$$y = mx + b$$

dove:

- y è la variabile dipendente (o di output)
- x è la variabile indipendente (o di input)
- m è il coefficiente di pendenza, che rappresenta il cambiamento in y rispetto a una variazione unitaria in x
- b è l'intercetta, che rappresenta il valore di y quando x è uguale a zero

Per il calcolo dei coefficienti m e b vengono utilizzate le seguenti formule:

$$m = \frac{Cov(x, y)}{Var(x)}$$
$$b = \bar{y} - m \cdot \bar{x}$$

dove:

- $Cov(x, y)$ rappresenta la covarianza tra x e y, calcolata come media delle deviazioni dei punti (x, y) rispetto alle loro medie. Si calcola usando questa formula:

$$\text{cov}(x, y) = \frac{1}{n} \sum_{i=1}^{n} (x_i - \bar{x})(y_i - \bar{y})$$

- $Var(x)$ rappresenta la varianza di x, data da:

$$\text{var}(x) = \frac{1}{n} \sum_{i=1}^{n} (x_i - \bar{x})^2$$

- $\bar{x}$ e $\bar{y}$ sono le medie di x e y, rispettivamente.

Dopo aver calcolato i coefficienti m e b, possiamo utilizzare l'equazione della retta per effettuare delle previsioni su nuovi valori di x

$$\hat{y} = mx + b$$

Dove $\hat{y}$ è il valore predetto di y per il nuovo valore di x valutato.

Per quanto riguarda il KNN, questo si basa su una misura di distanza tra i punti nel dataset. Useremo la distanza euclidea tra due punti (x_1, y_2) e (x_2, y_2), data da:

$$Dist(x_1, y_2, x_2, y_2) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Il parametro k influisce sulla classificazione dei punti e rappresenta il numero di vicini che verranno considerati. Un valore di k più piccolo comporta una classificazione più influenzata dai punti vicini, mentre un valore più grande porta ad una classificazione più generale basata su una maggiore diversità dei vicini. Il punto di test viene poi assegnato alla classe di maggioranza presa dai suoi k punti più vicini. Vengono cioè conteggiate le etichette dei punti vicini, e si assegna al punto l'etichetta avente il maggior numero di occorrenze.

3 Progettazione dell'algoritmo

3.1 Scelte di progetto

Gli input per entrambi i programmi sono liste di punti bidimensionali, rappresentati medianti coppie di numeri reali nel caso della regressione lineare, e triple di valori nel caso del KNN (due numeri reali e una stringa). La lista di coppie reali deve essere inserita nel seguente formato:

$$[(x_1, y_1), \ldots, (x_n, y_n)]$$

I punti del KNN invece sono acquisiti in questo formato, contenente anche le etichette usate per rappresentare la classi:

$$[(x_1, y_1, \text{classe}), \ldots, (x_n, y_n, \text{classe})]$$

All'avvio del programma, si è deciso di presentare all'utente un menu principale in cui poter scegliere le diverse operazioni (1: regressione lineare, 2: KNN, 3: uscita). Una volta scelta una delle due operazioni, all'utente viene chiesto di inserire il dataset corrispondente nel corretto formato. Una volta inserito il dataset e svolta l'operazione, l'utente può continuamente valutare nuovi valori/punti sul modello allenato, all'interno di un ciclo. E' il programma che, al termine di ogni valutazione, chiede all'utente se ha intenzione di procedere o se preferisce tornale al menu per svolgere un'altra operazione. Il programma termina quando si sceglie l'opzione numero 3 del menu.

Nel contesto della regressione lineare, una retta verticale non ha molto senso, poichè la sua equazione sarebbe $x = c$, dove c è una costante. Questo significherebbe che x rimane invariato per qualsiasi valore di y, il che è in contrasto con l'idea di relazione tra x e y. Per questo motivo, in entrambi i programmi bisogna controllare che nella prima operazione l'utente non inserisca un dataset che porti ad avere una retta verticale, ossia dove la varianza tra i valori della coordinata x è pari a zero, in quanto questo porterebbe ad avere una divisione per zero. Il linguaggio Haskell provvede già a questa evenienza restituendo il valore speciale 'NaN'. In Prolog questo non è previsto, quindi bisogna controllare in maniera condizionale se avviene o meno una divisione per zero. In ogni caso, l'output del programma nel caso di retta verticale sarà $y = \text{NaN}x + \text{NaN}$ per indicare il fatto che non vi è relazione tra x e y.

3.1.1 Haskell

Ci sono alcune differenze nel modo in cui i dati devono essere passati nei due programmi, in particolare per quanto riguarda l'operazione del KNN. Nel programma Haskell, il dataset etichettato deve essere passato in questo formato:

$$[(x_1, y_1, \text{"classe"}), \ldots, (x_n, y_n, \text{"classe"})]$$

dove cioè l'etichetta della classe è all'interno di virgolette per dichiarare al programma Haskell che si tratta di stringhe.

In entrambi i programmi viene fatta sempre una validazione sugli input inseriti dall'utente. In particolare, nel programma Haskell si segue questo formato generale, laddove non si utilizzi il pattern matching:

```
ask = do
  write_prompt
  input <- getLine
```

```
case readMaybe input :: Maybe <tipo> of
  Just value -> return (Just value)
  Nothing -> do
    write_error
    ask
```

Viene utilizzata la funzione "readMaybe" che converte una stringa contenente un valore di un tipo specifico in un valore del tipo 'Maybe', un tipo di dato che viene utilizzato per rappresentare il concetto di valore opzionale o mancante. In questo caso, la funzione readMaybe tenta di convertire la stringa in un valore del tipo specificato e, se la conversione non riesce, viene stampato un messaggio di errore e viene ripetuta l'interrogazione.

3.1.2 Prolog

Nel programma Prolog, il dataset etichettato deve essere passato in uno di questi due formati:

$$[(x_1, y_1, \text{'Classe'}), \ldots, (x_n, y_n, \text{'Classe'})] \quad \textit{oppure} \quad [(x_1, y_1, \text{classe}), \ldots, (x_n, y_n, \text{classe})]$$

dove cioè l'etichetta della classe è una stringa che inizia con un carattere maiuscolo in apici singoli, per differenziarla dalle variabili in Prolog, oppure un'arbitraria stringa che inizia con una minuscola.

Nel programma Prolog per la validazione degli input si segue questo formato generale:

```
ask(Data) :-
    repeat,
        write(Prompt),
        read(Data),
    valid(Data),
    !.
```

Si è scelto di usare il costrutto "repeat", che viene utilizzato per creare un ciclo infinito che continua fino a quando non si incontra una condizione di uscita (l'utente inserisce un input valido). Quando la validazione fallisce, viene fatto un backtracking per cercare altre alternative e il corpo del repeat viene ripetuto.

3.2 Passi dell'algoritmo

I passi dell'algoritmo sono:

- Visualizzazione del menu principale e acquisizione dall'utente di un valore corrispondente ad un'operazione da eseguire.

  1. Regressione lineare:
     - Lettura del dataset di punti bidimensionali, nel formato descritto.
     - Estrazione delle coordinate x e y dalla lista di punti in due liste separate.
     - Calcolo della media su entrambe le liste di valori di x e y.
     - Calcolo della varianza sui valori di x.
     - Calcolo della covarianza tra le liste di valori di x e y.
     - Applicazione dei risultati parziali nelle formule descritte sopra per trovare pendenza e intercetta della retta.
     - Stampa della retta trovata.
     - Ripetizione dei seguenti punti fino alla scelta di tornare al menu da parte dell'utente:
       * Lettura del nuovo valore di x da valutare sulla retta interpolatrice.
       * Stampa del risultato, ossia del corrispondente valore y trovato.
       * Richiesta se continuare o meno con la valutazione di altri valori di x.
  2. K-nearest neighbors:
     - Lettura del dataset di punti etichettati, nel formato descritto.
     - Lettura del numero di vicini da valutare (k).
     - Ripetizione dei seguenti punti fino alla scelta di tornare al menu da parte dell'utente:
       * Lettura del punto di test da valutare (senza etichetta).
       * Ricerca dei k punti del dataset più vicini al punto di test, tramite la misura di distanza euclidea.
       * Ricerca della classe di maggioranza tra i vicini del punto.
       * Stampa dell'elenco dei k vicini del punto.
       * Stampa dell'etichetta che indica la classe prevista per il punto.
       * Richiesta se continuare o meno con la valutazione di altri punti.
  3. Uscita dal menu: stampa del messaggio di chiusura e terminazione del programma.

4 Implementazione dell'algoritmo

4.1 Haskell

File sorgente operazioni_analisi_dati.hs:

```
{- Programma Haskell per calcolare regressione lineare e k-nearest neighbors -}

{- IMPORTAZIONE DELLE LIBRERIE -}

{- Formattazione e output -}
import Text.Printf (printf)          -- Per formattare le stringhe di output
{- Gestione delle liste -}
import Data.List (sortOn, nub)       -- Per ordinare liste e rimuovere duplicati
import Data.Function (on)            -- Per ordinare in base a una funzione
{- Conversione e lettura -}
import Text.Read (reads, readMaybe)  -- Per leggere input in modo sicuro
{- Controllo dei caratteri -}
import Data.Char (isPrint)           -- Per sapere se un carattere è stampabile

{- ESECUZIONE MAIN -}

{- Punto iniziale del programma -}
main :: IO ()
main = do
  putStrLn "\n"
  stampa_riga_orizzontale
  putStrLn "Progetto del corso di Programmazione Logica e Funzionale"
  putStrLn "Anno 2022/2023"
  putStrLn "Progetto realizzato da: Andrea De Lorenzis\n"
  menu_principale
  stampa_riga_orizzontale

{- Funzione Main contenente il menu di selezione dell'operazione da svolgere -}
menu_principale :: IO ()
menu_principale = do
  putStrLn "\nSelezionare l'operazione da svolgere:"
  putStrLn "1 - Regressione lineare"
  putStrLn "2 - K-Nearest neighbors"
  putStrLn "3 - Esci"

  choice <- getLine
  case choice of
    "1" -> do
      putStrLn ""
      putStrLn "-------- Regressione Lineare --------"
      punti <- leggi_dataset
      putStrLn "\nAdattamento della retta al dataset.."
      let (pendenza, intercetta) = calcola_coefficienti_retta punti
      let pendenza_formattata = printf "%.2f" pendenza
      let intercetta_formattata = printf "%.2f" intercetta
      putStrLn $ "\nRetta interpolatrice: y = " ++ pendenza_formattata
                                                ++ "x + "
                                                ++ intercetta_formattata
      valuta_valori_x (pendenza, intercetta)
      menu_principale
```

```
    "2" -> do
      putStrLn ""
      putStrLn "-------- K-Nearest Neighbors --------"
      dataset <- leggi_dataset_etichettato
      let num_punti = length dataset
      k <- leggi_valore_k num_punti
      valuta_punti k dataset
      menu_principale
    "3" -> putStrLn "\nChiusura del programma."
    _ -> do
      putStrLn "\nScelta incorretta. Seleziona un'opzione valida."
      menu_principale

{- DEFINIZIONE DI STRUTTURE DATI -}

{- Definizione di un tipo per rappresentare punti in uno spazio 2D -}
data Punto = Punto { xCoord :: Double, yCoord :: Double }
  deriving Show

{- Definizione di un tipo per rappresentare i punti 2D etichettati con una classe -}
data PuntoEtichettato = PuntoEtichettato { x :: Double, y :: Double, label :: String }
  deriving Show

{- CALCOLO REGRESSIONE LINEARE -}

{- Funzione che calcola i coefficienti (pendenza e intercetta) della retta
 - interpolatrice -}
calcola_coefficienti_retta :: [Punto] -> (Double, Double)
calcola_coefficienti_retta punti = (pendenza, intercetta)
  where
    xs = map xCoord punti
    ys = map yCoord punti
    pendenza = covarianza xs ys / varianza xs
    intercetta = media ys - pendenza * media xs

{- Funzione che calcola la media di una lista di Double -}
media :: [Double] -> Double
media xs = sum xs / fromIntegral (length xs)

{- Funzione che calcola la varianza di una lista di Double -}
varianza :: [Double] -> Double
varianza xs = sum [(x - m) ** 2 | x <- xs] / fromIntegral (length xs)
  where
    m = media xs

{- Funzione che calcola la covarianza di due liste di Double -}
covarianza :: [Double] -> [Double] -> Double
covarianza xs ys =
  sum [(x - mx) * (y - my) | (x, y) <- zip xs ys] / fromIntegral (length xs)
  where
    mx = media xs
    my = media ys

{- Funzione per ottenere dall'utente una lista di punti nel corretto formato -}
leggi_dataset :: IO [Punto]
leggi_dataset = do
```

```
  putStrLn "\nInserisci i punti del dataset nel formato [(x1,y1), ..., (xn,yn)]:"
  input <- getLine
  let lista_elaborata = readMaybe input :: Maybe [(Double, Double)]
  case lista_elaborata of
    Nothing -> do
      putStrLn $ "\nFormato non valido. Inserisci i punti nel formato " ++
                 " [(x1, y1), ..., (xn, yn)]."
      leggi_dataset
    Just punti -> if length punti < 2
                    then do
                      putStrLn "\nInput invalido. Inserisci almeno due punti."
                      leggi_dataset
                    else return $ converti_tuple_in_punti punti

{- Funzione che converte una lista di coppie di Double in una lista di punti -}
converti_tuple_in_punti :: [(Double, Double)] -> [Punto]
converti_tuple_in_punti = map (\(x, y) -> Punto x y)

{- Funzione che cicla continuamente per valutare nuovi valori di x -}
valuta_valori_x :: (Double, Double) -> IO ()
valuta_valori_x (pendenza, intercetta) = do
  valore <- leggi_valore
  case valore of
    Just x -> do
      let previsione = valuta_valore x (pendenza, intercetta)
      let previsione_formattata = printf "%.2f" previsione
      putStrLn $ "\nPer X = " ++ show x
                              ++ " il valore previsto e' Y = "
                              ++ previsione_formattata
      continua <- richiedi_continuazione
      if continua
        then valuta_valori_x (pendenza, intercetta)
        else return ()
    Nothing -> return ()  -- Ritorna al menu

{- Funzione che ottiene dall'utente un punto da valutare sulla retta ottenuta -}
leggi_valore :: IO (Maybe Double)
leggi_valore = do
  putStrLn "\nInserisci un valore per la coordinata x:"
  input <- getLine
  case readMaybe input of
    Just x -> return (Just x)
    Nothing -> do
      putStrLn "\nInput incorretto. Inserisci un numero valido."
      leggi_valore

{- Funzione che valuta la regressione lineare per il punto -}
valuta_valore :: Double -> (Double, Double) -> Double
valuta_valore x (pendenza, intercetta) = pendenza * x + intercetta

{- CALCOLO K-NEAREST NEIGHBORS -}

{- Funziona che trova i k vicini per un punto dal dataset -}
trova_vicini :: Int -> PuntoEtichettato -> [PuntoEtichettato] -> [PuntoEtichettato]
trova_vicini k punto_test dataset =
  take k $ sortOn (distanza punto_test) dataset
```

```
{- Funzione che calcola la distanza euclidea tra due punti -}
distanza :: PuntoEtichettato -> PuntoEtichettato -> Double
distanza p1 p2 = sqrt $ (x p1 - x p2) ^ 2 + (y p1 - y p2) ^ 2

{- Funzione che trova la classe di maggioranza tra i vicini -}
trova_classe_maggioranza :: [PuntoEtichettato] -> String
trova_classe_maggioranza vicini = fst $ head conteggi_ordinati
  where
    etichette = nub $ map label vicini
    conteggi = map (\label' ->
                     (label', length (filter (\p -> label p == label') vicini))
                   ) etichette
    conteggi_ordinati = reverse $ sortOn snd conteggi

{- Funzione per leggere dall'utente un dataset di punti etichettati -}
leggi_dataset_etichettato :: IO [PuntoEtichettato]
leggi_dataset_etichettato = do
  putStrLn $ "\nInserisci i punti etichettati del dataset nel formato " ++
             "[(x1,y1,\"<classe>\"), ..., (xn,yn,\"<classe>\")]:"
  input <- getLine
  case reads input of
    [(punti, "")] ->
      if all punto_etichettato_valido punti && length punti >= 2
        then return $ converti_triple_in_punti punti
        else do
          putStrLn $ "\nErrore. " ++ messaggio_errore punti
          leggi_dataset_etichettato
    _ -> do
      putStrLn $ "\nErrore. Assicurati che l'input sia nel formato " ++
                 "[(x1,y1,\"<classe>\"), ..., (xn,yn,\"<classe>\")]."
      leggi_dataset_etichettato
  where
    messaggio_errore ps
      | length ps < 2 = "Inserisci almeno due punti."
      | not (all punto_etichettato_valido ps) =
          "Formato non valido. Controlla che ogni " ++
          "punto sia nel formato (x, y, \"<classe>\")."
      | otherwise = ""

{- Converte una lista di triple in una lista di punti etichettati -}
converti_triple_in_punti :: [(Double, Double, String)] -> [PuntoEtichettato]
converti_triple_in_punti = map (\(x, y, l) -> PuntoEtichettato x y l)

{- Funzione per assicurarsi che il carattere sia stampabile
 - (es. non è un carattere di controllo come '\n' -}
punto_etichettato_valido :: (Double, Double, String) -> Bool
punto_etichettato_valido (_, _, c) = all isPrint c

{- Funzione per ottenere dall'utente il valore di k -}
leggi_valore_k :: Int -> IO Int
leggi_valore_k num_punti = do
  putStrLn $ "\nInserisci il valore di k (numero dei vicini, 1-" ++ show num_punti
                                                                  ++ "):"
  input <- getLine
  case readMaybe input of
```

```
    Just k
      | k > 0 && k <= num_punti -> return k
      | otherwise -> do
          putStrLn $ "\nIl valore di k deve essere un "  ++
                     "intero positivo compreso tra 1 e " ++
                                        show num_punti ++ "."
          leggi_valore_k num_punti
    Nothing -> do
      putStrLn "\nInput non valido. Inserisci un numero intero valido."
      leggi_valore_k num_punti

{- Funzione che cicla continuamente per valutare nuovi punti -}
valuta_punti :: Int -> [PuntoEtichettato] -> IO ()
valuta_punti k dataset = do
  punto_test <- leggi_punto
  let vicini = trova_vicini k punto_test dataset
      classe_prevista = trova_classe_maggioranza vicini
  stampa_vicini vicini
  putStrLn $ "\nClasse prevista per il punto: " ++ show classe_prevista
  continua <- richiedi_continuazione
  if continua
    then valuta_punti k dataset
    else return ()

{- Funzione per ottenere dall'utente un punto da testare sul KNN -}
leggi_punto :: IO PuntoEtichettato
leggi_punto = do
  putStrLn "\nInserisci il punto da valutare nel formato '(x, y)':"
  input <- getLine
  case readMaybe input of
    Just (x, y) -> return $ PuntoEtichettato x y "Z"  -- Etichetta non significativa
    Nothing -> do
      putStrLn "\nInput incorretto. Inserisci un punto nel formato '(x, y)'."
      leggi_punto

{- Funzione per stampare i vicini -}
stampa_vicini :: [PuntoEtichettato] -> IO ()
stampa_vicini vicini = do
  putStrLn "\nI vicini del punto sono:"
  mapM_ stampa_vicino vicini
    where
      stampa_vicino (PuntoEtichettato x y lbl) =
        putStrLn $ "Punto: (" ++ show x ++ ", " ++ show y ++ ", " ++ lbl ++ ")"

{- FUNZIONI AUSILIARIE -}

{- Funzione ausiliaria per chiedere all'utente se vuole continuare -}
richiedi_continuazione :: IO Bool
richiedi_continuazione = do
  putStrLn "\nVuoi continuare? (s/n)"
  input <- getLine
  case input of
    "s" -> return True
    "n" -> return False
    _ -> do
      putStrLn $ "\nInput invalido. Inserisci 's' per continuare o 'n' per " ++
```

```
                "ritornare al menu principale"
      richiedi_continuazione

{- Funzione ausiliaria per stampare una riga orizzontale -}
stampa_riga_orizzontale :: IO ()
stampa_riga_orizzontale = putStrLn $ "--------------------------------" ++
                                     "--------------------------------"
```

4.2 Prolog

File sorgente operazioni_analisi_dati.pl:

```
/* Programma Prolog per calcolare regressione lineare e k-nearest neighbors */

% ESECUZIONE MAIN

/* Punto iniziale del programma */
main :-
  nl, nl,
  stampa_riga_orizzontale,
  write('Progetto del corso di Programmazione Logica e Funzionale'), nl,
  write('Anno 2022/2023'), nl,
  write('Progetto realizzato da: Andrea De Lorenzis'), nl, nl,
  menu_principale,
  stampa_riga_orizzontale.

/* Menu di selezione dell'operazione da svolgere */
menu_principale :-
  nl, write('Selezionare l\'operazione da svolgere: '), nl,
  write('1 - Regressione lineare'), nl,
  write('2 - K-Nearest Neighbors'), nl,
  write('3 - Esci'), nl,

  read(Scelta),
  (Scelta = 1 -> esegui_regressione_lineare, menu_principale ;
   Scelta = 2 -> esegui_knn, menu_principale ;
   Scelta = 3 -> nl, write('Chiusura del programma.'), nl ;
   write('Scelta incorretta. Per favore, seleziona una delle opzioni valide.'), nl,
   menu_principale).

% CALCOLO REGRESSIONE LINEARE

/* Predicato per eseguire la regressione lineare */
esegui_regressione_lineare :-
  nl, write('-------- Regressione Lineare --------'), nl,
  leggi_dataset(Punti),
  nl, write('Adattamento della retta al dataset..'), nl,
  calcola_coefficienti_retta(Punti, Pendenza, Intercetta),
  nl,
  (Pendenza = 'undefined' ->  % Caso di retta verticale
   write('Retta interpolatrice: y = NaNx + NaN'), nl
  ;
   format('Retta interpolatrice: y = ~2fx + ~2f~n', [Pendenza, Intercetta])
  ),
  valuta_valori_x(Pendenza, Intercetta).

/* Predicato per leggere una lista di tuple da tastiera */
leggi_dataset(Lista) :-
  repeat,
  nl, write('Inserisci i punti del dataset nel formato: [(x1,y1), ..., (xn,yn)]:'),
  nl, read(Lista),
  (lista_punti_valida(Lista) ->
     !    % Successo: lista valida di punti
     ;
     nl, write('Input invalido. La lista deve contenere almeno due punti '),
```

```
    nl, write('nel formato [(x,y), ...]'), nl,
    fail % Fallimento: formato non valido, ritentare
  ).

/* Predicato per validare una lista di punti 2D */
lista_punti_valida(Lista) :-
  is_list(Lista),
  length(Lista, NumPunti),
  NumPunti >= 2,
  forall(member(Punto, Lista), punto_valido(Punto)).

/* Predicato per controllare che un punto sia nel formato (x,y) */
punto_valido(Punto) :-
  Punto = (X, Y),
  number(X),
  number(Y).

/* Calcola i coefficienti (pendenza e intercetta) della retta interpolatrice */
calcola_coefficienti_retta(Punti, Pendenza, Intercetta) :-
  estrai_coordinate_x(Punti, Xs),
  estrai_coordinate_y(Punti, Ys),
  varianza(Xs, VarX),
  (VarX =:= 0 -> % Se VarX è zero, gestisco il caso di retta verticale
   Pendenza = 'undefined',
   Intercetta = 'undefined'
  ;
   covarianza(Xs, Ys, CovXY),
   Pendenza is CovXY / VarX,
   media(Ys, MediaY),
   media(Xs, MediaX),
   Intercetta is MediaY - Pendenza * MediaX
  ).

/* Estra le coordinate X da una lista di punti */
estrai_coordinate_x([], []).
estrai_coordinate_x([(X, _) | Punti], [X | Xs]) :-
  estrai_coordinate_x(Punti, Xs).

/* Estrae le coordinate Y da una lista di punti */
estrai_coordinate_y([], []).
estrai_coordinate_y([(_, Y) | Punti], [Y | Ys]) :-
  estrai_coordinate_y(Punti, Ys).

/* Calcola la varianza di una lista di numeri */
varianza([], 0).
varianza([X | Xs], Var) :-
  media([X | Xs], Media),
  varianza(Xs, Resto),
  Var is Resto + (X - Media) * (X - Media).

/* Calcola la covarianza di due liste di numeri */
covarianza([], [], 0).
covarianza([X | Xs], [Y | Ys], Cov) :-
  media([X | Xs], MediaX),
  media([Y | Ys], MediaY),
  covarianza(Xs, Ys, Resto),
```

```
  Cov is Resto + (X - MediaX) * (Y - MediaY).

/* Calcola la media di una lista di numeri */
media([], 0).
media([X | Xs], Media) :-
  media(Xs, Resto),
  length(Xs, N),
  Media is (X + N * Resto) / (N + 1).

/* Predicato che cicla continuamente per valutare nuovi valori di x */
valuta_valori_x(Pendenza, Intercetta) :-
  leggi_valore(X),
  valuta_valore(X, Pendenza, Intercetta, Y),
  (Pendenza = 'undefined' -> % Caso di retta verticale
   nl, format('Per X = ~2f il valore previsto e\' Y = NaN', [X]), nl
  ;
   nl, format('Per X = ~2f il valore previsto e\' Y = ~2f', [X, Y]), nl
  ),
  richiedi_continuazione(Scelta),
  (Scelta = 's' -> valuta_valori_x(Pendenza, Intercetta) ; true).

/* Predicato che ottiene dall'utente una coordinata X da valutare sulla
 * retta ottenuta */
leggi_valore(X) :-
  repeat,
  nl, write('Inserisci un valore per la coordinata x: '), nl,
  read(X),
  (number(X) ->
     true  % Successo: X è un numero
     ;
     nl, write('Il valore inserito non e\' un numero.'), nl,
     fail % Fallimento: X non è un numero, ritentare
  ).

/* Predicato che valuta la regressione lineare per il punto */
valuta_valore(X, Pendenza, Intercetta, Y) :-
  (Pendenza = 'undefined' ->  % Caso di retta verticale
   Y is 0  % valore non significativo
  ;
   Y is Pendenza * X + Intercetta
  ).

% CALCOLO K-NEAREST NEIGHBORS

/* Predicato per eseguire il KNN */
esegui_knn :-
  nl, write('-------- K-Nearest Neighbors --------'), nl,
  leggi_dataset_etichettato(Dataset),
  length(Dataset, NumPunti),
  repeat,
  nl, write('Inserisci il valore per k: '), nl,
  read(K),
  (number(K) ->
     K > 0,
    (K =< NumPunti ->
       !
```

```
      ;
      nl, write('Input invalido. Inserisci un intero positivo '),
      write('minore del (o uguale al) numero totale di punti nel dataset'), nl,
      format('(Numero di punti: ~d)', [NumPunti]), nl,
      fail
    )
    ;
    nl, write('Input invalido. Inserisci un intero positivo.'), nl,
    fail
  ),
  valuta_punti(Dataset, K).

/* Predicato per ottenere dall'utente un dataset di punti etichettati con una classe */
leggi_dataset_etichettato(Lista) :-
  repeat,
  nl, write('Inserisci i punti del dataset nel formato: '),
  write('[(x1,y1,<classe>), ..., (xn,yn,<classe>)].'), nl,
  read(Lista),
  (dataset_etichettato_valido(Lista) ->
     !  % Successo: la lista è valida
     ;
     nl, write('Input invalido. La lista deve contenere almeno due punti '),
     write('nel formato [(x,y,<classe>), ...]'), nl,
     fail % Fallimento: formato lista invalido, ritentare
  ).

/* Predicato di validazione per un dataset con etichette */
dataset_etichettato_valido(Lista) :-
  is_list(Lista),
  length(Lista, NumPunti),
  NumPunti >= 2,
  forall(member(Punto, Lista), punto_etichettato_valido(Punto)).

/* Predicato per controllare se un elemento è un punto etichettato valido nel
   formato (X, Y, C) */
punto_etichettato_valido(Punto) :-
  Punto = (X, Y, C),
  number(X),
  number(Y),
  atom(C).

/* Predicato per inserire continuamente punti da valutare con KNN */
valuta_punti(_, 0) :- !.
valuta_punti(Dataset, K) :-
  leggi_punto_etichettato(Punto),
  trova_vicini(K, Punto, Dataset, Vicini),
  nl, format('I ~d vicini del punto sono: ', [K]), nl,
  stampa_vicini(Vicini), nl,
  trova_classe_maggioranza(Vicini, Classe),
  format('La classe prevista per il punto e\': ~w~n', [Classe]),
  richiedi_continuazione(Scelta),
  (Scelta = 's' -> valuta_punti(Dataset, K) ; true).

/* Predicato per leggere un singolo punto da valutare con KNN */
leggi_punto_etichettato(Punto) :-
  repeat,
```

```
  nl, write('Inserisci il valore del punto da testare nel formato (x,y): '), nl,
  read(Punto),
  (punto_valido(Punto) ->
   true  % Successo: punto valido, uscita dal repeat
   ;
   nl, write('Punto non valido. Assicurati di inserire un punto '),
   write('nel formato (x,y).'), nl,
   fail % Fallimento: punto non valido, ritentare
  ).

/* Predicato che trova i k vicini per un punto */
trova_vicini(K, Punto, Dataset, Vicini) :-
  calcola_distanze(Punto, Dataset, Distanze),
  ordina_distanze(Distanze, DistanzeOrdinate),
  prendi(K, DistanzeOrdinate, CoppieVicini),
  converti_coppie_in_vicini(CoppieVicini, Vicini).

/* Predicato per calolcare le distanze tra PuntoTest e ogni altro punto
 * del dataset */
calcola_distanze(_, [], []).
calcola_distanze(PuntoTest, [Punto | Resto], [Dist-Punto | Distanze]) :-
  distanza(Punto, PuntoTest, Dist),
  calcola_distanze(PuntoTest, Resto, Distanze).

/* Predicato che calcola la distanza euclidea tra due punti */
distanza((X1, Y1, _), (X2, Y2), Dist) :-
  Dist is sqrt((X1 - X2) * (X1 - X2) + (Y1 - Y2) * (Y1 - Y2)).

/* Ordina le distanze in ordine crescente */
ordina_distanze(Distanze, DistanzeOrdinate) :-
  keysort(Distanze, DistanzeOrdinate).

/* Prende i primi K elementi di una lista */
prendi(0, _, []).
prendi(K, [X | Xs], [X | Resto]) :-
  K > 0,
  K1 is K - 1,
  prendi(K1, Xs, Resto).

/* Rimuove il valore della distanza dal formato (Dist-(X, Y, Class))
   per ottenere il formato (X, Y, Class) */
converti_coppie_in_vicini([], []).
converti_coppie_in_vicini([_-Vicino | CoppieRestanti], [Vicino | ViciniRestanti]) :-
  converti_coppie_in_vicini(CoppieRestanti, ViciniRestanti).

/* Predicato per stampare i vicini nel formato (x, y, C) */
stampa_vicini([]).
stampa_vicini([(X, Y, C) | Resto]) :-
  format('Punto: (~2f, ~2f, ~w)~n', [X, Y, C]),
  stampa_vicini(Resto).

/* Predicato che trova la classe di maggioranza tra i vicini */
trova_classe_maggioranza(Vicini, ClasseMaggioranza) :-
  conta_occorrenze_classe(Vicini, MapConteggi),
  classe_max_occorrenze(MapConteggi, ClasseMaggioranza).
```

```
/* Predicato per conteggiare le occorrenze di ogni classe nei vicini */
conta_occorrenze_classe(Vicini, MapConteggi) :-
  conta_occorrenze_classe(Vicini, [], MapConteggi).

conta_occorrenze_classe([], MapConteggi, MapConteggi).
conta_occorrenze_classe([(_, _, Classe) | Vicini], MapConteggiParziale, MapConteggi) :-
  aggiorna_map_conteggi(Classe, MapConteggiParziale, NuovaMapConteggi),
  conta_occorrenze_classe(Vicini, NuovaMapConteggi, MapConteggi).

aggiorna_map_conteggi(Classe, [], [(Classe, 1)]).
aggiorna_map_conteggi(Classe,
                      [(Classe, Conteggio) | Resto],
                      [(Classe, NuovoConteggio) | Resto]
                     ) :-
  NuovoConteggio is Conteggio + 1.
aggiorna_map_conteggi(Classe,
                      [(AltraClasse, Conteggio) | Resto],
                      [(AltraClasse, Conteggio) | NuovoResto]
                     ) :-
  Classe \= AltraClasse,
  aggiorna_map_conteggi(Classe, Resto, NuovoResto).

/* Predicato per trovare la classe con il maggior numero di occorrenze */
classe_max_occorrenze([(Classe, Conteggio) | Resto], ClasseMaggioranza) :-
  classe_max_occorrenze(Resto, Classe, Conteggio, ClasseMaggioranza).

classe_max_occorrenze([], ClasseMaggioranza, _, ClasseMaggioranza).
classe_max_occorrenze([(Classe, Conteggio) | Resto],
                      ClasseMax, ConteggioMax, ClasseMaggioranza) :-
  (Conteggio > ConteggioMax ->
   classe_max_occorrenze(Resto, Classe, Conteggio, ClasseMaggioranza)
   ;
   classe_max_occorrenze(Resto, ClasseMax, ConteggioMax, ClasseMaggioranza)
  ).

% FUNZIONI AUSILIARIE

/* Predicato per chiedere all'utente se vuole continuare */
richiedi_continuazione(Scelta) :-
  nl, write('Vuoi continuare? (s/n)'), nl,
  read(SceltaInput),
  (atom(SceltaInput), SceltaInput = 's' -> Scelta = 's' ;
   atom(SceltaInput), SceltaInput = 'n' -> Scelta = 'n' ;
   nl, write('Input incorretto. Inserisci "s" per continuare o "n" per uscire.'),
   nl, richiedi_continuazione(Scelta)).

/* Predicato per stampare una linea orizzontale */
stampa_riga_orizzontale :- stampa_riga_orizzontale(65).

/* Specifica il numero di trattini nella linea orizzontale */
stampa_riga_orizzontale(N) :-
  N > 0,
  write('-'),
  N1 is N - 1,
  stampa_riga_orizzontale(N1).
stampa_riga_orizzontale(0) :- nl.
```

5 Testing del programma

Il programma emette un messaggio di errore e chiede di rieseguire l'inserimento nei seguenti casi:

- All'interno del menu viene scelta un'opzione non presente o un valore che non sia un numero intero.
- All'atto della scelta se continuare o meno con la valutazione dei valori, l'utente inserisce un valore diverso da 's' o 'n'.
- L'utente inserisce il dataset in un formato incorretto, ossia tale che la lista non sia in un formato valido o che i suoi elementi non siano punti validi.
- L'utente inserisce un dataset contenente meno di due punti.
- L'utente inserisce un valore/punto di valutazione in un formato incorretto.
- L'utente inserisce un valore k di vicini che non sia un intero positivo, o il cui valore sia maggiore del numero totale di punti nel dataset.

Di seguito sono riportati dei test significativi per ogni operazione e su entrambi i programmi.

5.1 Testing del programma Haskell

5.1.1 Testing "Regressione lineare"

Test Haskell 1
Dataset: $[(1,2),(2,4),(3,6),(4,8),(5,10)]$
Valore x: 6.0
Retta: $y = 2.00x + 0.00$
Valore y: 12.00

Test Haskell 2
Dataset: $[(1,2),(2,3),(3,7),(4,6),(5,8),(6,9)]$
Valore x: 7.0
Retta: $y = 1.40x + 0.93$
Valore y: 10.73

Test Haskell 3 (Pendenza negativa)
Dataset: $[(-1,3),(0,1),(1,-1),(2,-3),(3,-5)]$
Valore x: 4.0
Retta: $y = -2.00x + 1.00$
Valore y: -7.00

Test Haskell 4 (Retta verticale)
Dataset: $[(2,1),(2,2),(2,3),(2,4),(2,5)]$
Valore x: 2.0
Retta: $y = NaNx + NaN$
Valore y: NaN

Test Haskell 5 (Retta orizzontale)
Dataset: $[(1,2),(2,2),(3,2),(4,2),(5,2)]$
Valore x: 5.0

Retta: $y = 0.00x + 2.00$
Valore y: 2.00

Test Haskell 6 (Numeri decimali)
Dataset: $[(0.1, 0.3), (0.2, 0.6), (0.3, 0.9), (0.4, 1.2), (0.5, 1.5)]$
Valore x: 0.6
Retta: $y = 3.00x + 0.00$
Valore y: 1.80

5.1.2 Testing "K-Nearest neighbors"

Test Haskell 7
Dataset: $[(1, 2, \text{"A"}), (2, 3, \text{"A"}), (2, 4, \text{"A"}), (3, 5, \text{"B"}), (4, 6, \text{"B"}), (5, 7, \text{"B"}), (6, 8, \text{"C"}), (7, 9, \text{"C"})]$
K: 3
Punto di test: $(3, 4)$
Vicini:

- $(2.00, 4.00, A)$
- $(3.00, 5.00, B)$
- $(2.00, 3.00, A)$

Classe prevista: A

Test Haskell 8
Dataset: $[(1, 2, \text{"A"}), (2, 3, \text{"A"}), (2, 4, \text{"A"}), (3, 5, \text{"B"}), (4, 6, \text{"B"}), (5, 7, \text{"B"}), (6, 8, \text{"C"}), (7, 9, \text{"C"})]$
K: 2
Punto di test: $(6, 5)$
Vicini:

- $(4.00, 6.00, B)$
- $(5.00, 7.00, B)$

Classe prevista: B

Test Haskell 9 (Singolo vicino)
Dataset: $[(1, 2, \text{"A"}), (2, 3, \text{"A"}), (2, 4, \text{"A"}), (3, 5, \text{"B"}), (4, 6, \text{"B"}), (5, 7, \text{"B"}), (6, 8, \text{"C"}), (7, 9, \text{"C"})]$
K: 1
Punto di test: $(3.5, 4.5)$
Vicini:

- $(3.00, 5.00, B)$

Classe prevista: B

Test Haskell 10 (Punto di test uguale ad un vicino)
Dataset: $[(1, 2, \text{"A"}), (2, 3, \text{"A"}), (2, 4, \text{"A"}), (3, 5, \text{"B"}), (4, 6, \text{"B"}), (5, 7, \text{"B"}), (6, 8, \text{"C"}), (7, 9, \text{"C"})]$
K: 1
Punto di test: $(2, 4)$
Vicini:

- $(2.00, 4.00, A)$

Classe prevista: A

Test Haskell 11 (Punto di test vicino a 3 punti della stessa classe 'A')
Dataset: $[(1, 2, \text{"A"}), (2, 3, \text{"A"}), (3, 4, \text{"A"}), (4, 5, \text{"B"}), (5, 6, \text{"B"})]$
K: 5
Punto di test: $(3.5, 4.5)$
Vicini:

- $(3.00, 4.00, A)$
- $(4.00, 5.00, B)$
- $(2.00, 3.00, A)$
- $(5.00, 6.00, B)$
- $(1.00, 2.00, A)$

Classe prevista: A

5.2 Testing del programma Prolog

5.2.1 Testing "Regressione lineare"

Test Prolog 1
Dataset: $[(1,2),(2,4),(3,6),(4,8),(5,10)]$
Valore x: 6.0
Retta: $y = 2.00x + 0.00$
Valore y: 12.00

Test Prolog 2
Dataset: $[(1,2),(2,3),(3,7),(4,6),(5,8),(6,9)]$
Valore x: 7.0
Retta: $y = 1.41x + 0.88$
Valore y: 10.78

Test Prolog 3 (Pendenza negativa)
Dataset: $[(-1,3),(0,1),(1,-1),(2,-3),(3,-5)]$
Valore x: 4.0
Retta: $y = -2.00x + 1.00$
Valore y: -7.00

Test Prolog 4 (Retta verticale)
Dataset: $[(2,1),(2,2),(2,3),(2,4),(2,5)]$
Valore x: 2.0
Retta: $y = NaNx + NaN$
Valore y: NaN

Test Prolog 5 (Retta orizzontale)
Dataset: $[(1,2),(2,2),(3,2),(4,2),(5,2)]$
Valore x: 5.0
Retta: $y = 0.00x + 2.00$
Valore y: 2.00

Test Prolog 6 (Numeri decimali)
Dataset: $[(0.1,0.3),(0.2,0.6),(0.3,0.9),(0.4,1.2),(0.5,1.5)]$
Valore x: 0.6
Retta: $y = 3.00x + 0.00$
Valore y: 1.80

5.2.2 Testing "K-Nearest neighbors"

Test Prolog 7
Dataset: $[(1,2,\text{'A'}),(2,3,\text{'A'}),(2,4,\text{'A'}),(3,5,\text{'B'}),(4,6,\text{'B'}),(5,7,\text{'B'}),(6,8,\text{'C'}),(7,9,\text{'C'})]$
K: 3
Punto di test: $(3,4)$
Vicini:

- $(2.00, 4.00, A)$
- $(3.00, 5.00, B)$
- $(2.00, 3.00, A)$

Classe prevista: A

Test Prolog 8
Dataset: $[(1, 2, \text{'A'}), (2, 3, \text{'A'}), (2, 4, \text{'A'}), (3, 5, \text{'B'}), (4, 6, \text{'B'}), (5, 7, \text{'B'}), (6, 8, \text{'C'}), (7, 9, \text{'C'})]$
K: 2
Punto di test: $(6, 5)$
Vicini:

- $(4.00, 6.00, B)$
- $(5.00, 7.00, B)$

Classe prevista: B

Test Prolog 9 (Singolo vicino)
Dataset: $[(1, 2, \text{'A'}), (2, 3, \text{'A'}), (2, 4, \text{'A'}), (3, 5, \text{'B'}), (4, 6, \text{'B'}), (5, 7, \text{'B'}), (6, 8, \text{'C'}), (7, 9, \text{'C'})]$
K: 1
Punto di test: $(3.5, 4.5)$
Vicini:

- $(3.00, 5.00, B)$

Classe prevista: B

Test Prolog 10 (Punto di test uguale ad un vicino)
Dataset: $[(1, 2, \text{'A'}), (2, 3, \text{'A'}), (2, 4, \text{'A'}), (3, 5, \text{'B'}), (4, 6, \text{'B'}), (5, 7, \text{'B'}), (6, 8, \text{'C'}), (7, 9, \text{'C'})]$
K: 1
Punto di test: $(2, 4)$
Vicini:

- $(2.00, 4.00, A)$

Classe prevista: A

Test Prolog 11 (Punto di test vicino a 3 punti della stessa classe 'A')
Dataset: $[(1, 2, \text{'A'}), (2, 3, \text{'A'}), (3, 4, \text{'A'}), (4, 5, \text{'B'}), (5, 6, \text{'B'})]$
K: 5
Punto di test: $(3.5, 4.5)$
Vicini:

- $(3.00, 4.00, A)$
- $(4.00, 5.00, B)$
- $(2.00, 3.00, A)$
- $(5.00, 6.00, B)$
- $(1.00, 2.00, A)$

Classe prevista: A