Università degli Studi di Urbino Carlo Bo

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

Corso: Programmazione Logica e Funzionale

30 agosto 2023

Docente
Prof. Marco Bernardo

Corsista
Andrea De Lorenzis, 308024

Indice

1 Specifica del problema **2**

2 Analisi del problema **3**
2.1 Dati di ingresso del problema . . . 3
2.2 Dati di uscita del problema . . . 3
2.3 Relazioni intercorrenti . . . 3

3 Progettazione dell'algoritmo **5**
3.1 Scelte di progetto . . . 5
3.2 Passi dell'algoritmo . . . 5

4 Implementazione dell'algoritmo **7**
4.1 Haskell . . . 7
4.2 Prolog . . . 14

5 Testing del programma **22**
5.1 Testing del programma Haskell . . . 22
5.1.1 Testing "Regressione lineare" . . . 22
5.1.2 Testing "K-Nearest neighbors" . . . 23
5.2 Testing del programma Prolog . . . 25
5.2.1 Testing "Regressione lineare" . . . 25
5.2.2 Testing "K-Nearest neighbors" . . . 25

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. Scrivere un programma Haskell e un programma Prolog che implementano regressione lineare e KNN.

2 Analisi del problema

2.1 Dati di ingresso del problema

Inizialmente bisogna fornire un valore numerico tra 1 e 2 per scegliere quale delle due operazioni svolgere. Nel caso dell'operazione di regressione lineare, i dati di ingresso sono:

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

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

- Un valore per la coordinata x da usare per prevedere il corrispondente valore di y sulla base del modello lineare allenato

Nel caso dell'operazione di KNN i dati in ingresso sono:

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

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

- 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 per la coordinata y.

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.

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.

Sia per regressione lineare che per KNN devono esserci almeno due punti distinti nel dataset per svolgere l'operazione. Inoltre, eventuali punti duplicati non verranno tollerati in fase d'acquisizione.

3 Progettazione dell'algoritmo

3.1 Scelte di progetto

La lista di coppie reali che rappresentano i punti 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:

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

Inizialmente, all'utente viene presentato un menu principale in cui poter scegliere le diverse operazioni (1: regressione lineare, 2: KNN, 3: uscita). Una volta scelta una delle prime due operazioni, viene chiesto di inserire il dataset corrispondente. Inserito il dataset e svolta l'operazione, l'utente può continuamente fare nuove valutazioni sul modello allenato. Al termine di ogni valutazione, viene chiesto all'utente se ha intenzione di procedere o se preferisce tornare al menu per svolgere un'altra operazione.

Nel contesto della regressione lineare, una retta verticale non ha molto senso, dato che 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. Nonostante ciò i dataset che portino a modellare una retta verticale verranno tollerati in fase di acquisizione, e verrà fornito un output che faccia intuire la mancanza di relazione tra le due variabili (p.e. $y = NaNx + NaN$). Inoltre, sia nel caso della regressione lineare che del KNN non si accettano punti duplicati nel dataset.

3.2 Passi dell'algoritmo

I passi dell'algoritmo sono:

- Visualizzare il menu principale e acquisire il valore che indica l'operazione da svolgere:
  - Caso base: se l'utente sceglie l'opzione "3".
  - Caso generale: se l'utente sceglie l'opzione "1" o "2", viene eseguita la corrispondente operazione e poi si procede ricorsivamente con un'ulteriore scelta.

  1. Regressione lineare:
     - Acquisire l'insieme di punti.
     - Estrarre le coordinate x e y dalla lista di punti in due liste separate.
     - Calcolare la media su entrambe le liste di valori di x e y.
     - Calcolare la varianza sui valori di x.
     - Calcolare la covarianza tra le liste di valori di x e y.
     - Applicare i risultati parziali nelle formule descritte sopra per trovare pendenza e intercetta della retta.
     - Stampare la retta trovata.
     - Valutare nuovi valori per la coordinata x sulla retta trovata:
       * Caso base: se l'utente decide di non continuare, la serie di valutazioni termina.

    - Caso generale: se l'utente decide di continuare, si acquisisce il nuovo valore della coordinata x, lo si valuta sull'equazione della retta, si restituisce il valore previsto di y e si richiede all'utente se ha intenzione di procedere o meno. In caso affermativo, si procede ricorsivamente con un'ulteriore valutazione sul modello.

2. K-nearest neighbors:
   - Acquisire l'insieme di punti etichettati.
   - Acquisire il numero di vicini da valutare (k).
   - Acquisire e classificare nuovi punti:
     - Caso base: se l'utente decide di non continuare, la serie di valutazioni termina.
     - Caso generale: se l'utente decide di continuare, si acquisisce un nuovo punto (senza etichetta), si ricercano i k punti del dataset più vicini al punto acquisito tramite la misura di distanza euclidea, si trova la classe di maggioranza tra quest'ultimi e si stampa l'output, che è dato dall'elenco dei k vicini e dal risultato di classificazione. Infine si richiede all'utente se ha intenzione di procedere o meno. In caso affermativo, si procede ricorsivamente con un'ulteriore valutazione sul modello.
3. Uscita: stampare il messaggio di terminazione.

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 -}

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

{- Definizione di un tipo per rappresentare i punti. -}

data Punto = Punto { xCoord :: Double, yCoord :: Double }
  deriving Show

{- Definizione di un tipo per rappresentare i punti etichettati. -}

data PuntoEtichettato = PuntoEtichettato { x :: Double, y :: Double, label :: String }
  deriving Show

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 che gestisce il menu delle operazioni. -}

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

{- CALCOLO REGRESSIONE LINEARE -}

{- Funzione che calcola i coefficienti (pendenza e intercetta) della retta sul dataset:
   - il suo unico argomento è la lista di punti (dataset). -}

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:
   - il suo unico argomento è la lista di Double. -}

media :: [Double] -> Double
media xs = sum xs / fromIntegral (length xs)

{- Funzione che calcola la varianza di una lista di Double:
   - il suo unico argomento è la 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
   - il primo argomento è la prima delle due liste;
   - il secondo argomento è la seconda delle due liste. -}

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 acquisire una lista di punti nel corretto formato. -}
```

```
leggi_dataset :: IO [Punto]
leggi_dataset = do
  putStrLn "\nInserisci almeno due punti distinti 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 if length (nub punti) /= length punti
        then do
          putStrLn "\nInput invalido. Alcuni punti inseriti non sono distinti."
          leggi_dataset
        else return $ converti_tuple_in_punti punti

{- Funzione che converte una lista di coppie di Double in una lista di punti:
   il suo unico argomento è la lista di coppie. -}

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 della coordinata x
   sulla retta:
   - il suo unico argomento è una coppia di Double (pendenza e intercetta). -}

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 acquisisce un valore per la coordinata x. -}

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 un valore della coordinata x sulla retta:
   - il primo argomento è il valore della coordinata x;
   - il secondo argomento è una coppia di double (pendenza e intercetta). -}

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:
   - il primo argomento è il valore k dei vicini;
   - il secondo argomento è il punto di test;
   - il terzo argomento è una lista di punti etichettati (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:
   - il primo argomento è il primo dei due punti;
   - il secondo argomento è il secondo dei 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:
   - l'unico argomento è l'insieme dei k punti 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 acquisire un dataset di punti etichettati. -}

leggi_dataset_etichettato :: IO [PuntoEtichettato]
leggi_dataset_etichettato = do
  putStrLn $ "\nInserisci almeno due punti distinti ed etichettati nel formato " ++
             "[(x1,y1,\"<classe>\"), ..., (xn,yn,\"<classe>\")]:"
  input <- getLine
  case reads input of
    [(triple, "")] ->
      if all punto_etichettato_valido triple && length triple >= 2 && punti_distinti triple
        then return $ converti_triple_in_punti triple
        else do
          putStrLn $ "\nInput invalido. " ++ messaggio_errore triple
          leggi_dataset_etichettato
    _ -> do
```

```
      putStrLn $ "\nInput invalido. 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>\")."
      | not (punti_distinti ps) = "Tutti i punti devono essere distinti."
      | otherwise = ""

{- Funzione che verifica se tutti i punti in un dataset etichettato sono distinti:
   - il suo unico argomento è la lista di punti etichettati. -}

punti_distinti :: [(Double, Double, String)] -> Bool
punti_distinti ps = length solo_punti == length (nub solo_punti)
  where
    solo_punti = [(x, y) | (x, y, _) <- ps]

{- Funzione che converte una lista di triple in una lista di punti etichettati:
   - l'unico argomento è la lista di triple di valori. -}

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 per il punto etichettato sia stampabile:
   - l'unico argomento è la tripla di valori che rappresentano il punto etichettato.
   Per esempio, non deve essere un carattere di controllo come '\n'. -}

punto_etichettato_valido :: (Double, Double, String) -> Bool
punto_etichettato_valido (_, _, c) = all isPrint c

{- Funzione per acquisire il valore di k:
   - l'unico argomento è il numero di punti nel dataset.
   Il numero dei vicini deve essere infatti minore uguale del numero di punti. -}

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 sul KNN:
   - il primo argomento è il valore k dei vicini;
```

```
   - il secondo argomento è la lista di punti etichettati. -}

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 acquisire un punto di test. -}

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 del punto di test:
   - l'unico argomento è la lista di tutti 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. */

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.

/* Predicato per gestire il menu delle operazioni. */

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 punti da tastiera. */

leggi_dataset(Lista) :-
  nl, write('Inserisci almeno due punti distinti nel formato '),
  write('[(x1,y1), ..., (xn,yn)]: '),
  nl, read(Input),
  (
    % Controllo che la lista sia nel formato corretto
    (\+ is_list(Input)
     ;
```

```
    \+ forall(member(Punto, Input), punto_valido(Punto)))
  ->
    nl, write('Input invalido. Assicurati di inserire una lista nel formato '),
    write('[(x1,y1), ..., (xn,yn)].'), nl,
    leggi_dataset(Lista)
  ;
    % Controllo che ci siano almeno due punti
    length(Input, NumPunti),
    NumPunti < 2
  ->
    nl, write('Input invalido. Inserisci almeno due punti.'), nl,
    leggi_dataset(Lista)
  ;
    % Controllo che tutti i punti siano distinti
    \+ punti_distinti(Input)
  ->
    nl, write('Input invalido. Tutti i punti devono essere distinti.'), nl,
    leggi_dataset(Lista)
  ;
    Lista = Input
 ).


/* Predicato per controllare che un punto sia nel formato (x,y):
   - il suo unico argomento è il punto. */

punto_valido(Punto) :-
  Punto = (X, Y),
  number(X),
  number(Y).

/* Predicato che verifica se tutti i punti sono distinti:
   - il suo unico argomento è la lista di punti. */

punti_distinti([]).
punti_distinti([Testa|Coda]) :-
  \+ member(Testa, Coda),
  punti_distinti(Coda).

/* Predicato che calcola i coefficienti (pendenza e intercetta) della retta:
   - il suo unico argomento è la lista di punti che forma il dataset. */

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
  ).
```

```
/* Predicato che estrae le coordinate X da una lista di punti:
   - il suo unico argomento è la lista. */

estrai_coordinate_x([], []).
estrai_coordinate_x([(X, _) | Punti], [X | Xs]) :-
  estrai_coordinate_x(Punti, Xs).

/* Predicato che estrae le coordinate Y da una lista di punti:
   - il suo unico argomento è la lista. */

estrai_coordinate_y([], []).
estrai_coordinate_y([(_, Y) | Punti], [Y | Ys]) :-
  estrai_coordinate_y(Punti, Ys).

/* Predicato che calcola la varianza di una lista di numeri:
   - il suo unico argomento è la lista. */

varianza([], 0).
varianza([X | Xs], Var) :-
  media([X | Xs], Media),
  varianza(Xs, Resto),
  Var is Resto + (X - Media) * (X - Media).

/* Predicato che calcola la covarianza di due liste di numeri:
   - il primo argomento è la prima delle due liste;
   - il secondo argomento è la seconda delle due liste. */

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).

/* Predicato che calcola la media di una lista di numeri:
   - il suo unico argomento è la lista.*/

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 della coordinata x:
   - il primo valore è la pendenza della retta;
   - il secondo valore è l'intercetta della retta. */

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 acquisisce un valore per la coordinata X. */

leggi_valore(X) :-
  nl, write('Inserisci un valore per la coordinata x: '), nl,
  read(Val),
  (number(Val) ->
   X = Val
   ;
   nl, write('Il valore inserito non e\' un numero.'), nl,
   leggi_valore(X)
  ).

/* Predicato che valuta un valore della coordinata x sulla retta:
  - il primo argomento è il valore della coordinata x;
  - il secondo argomento è la pendenza della retta;
  - il terzo argomento è l'intercetta della retta. */

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),
  leggi_valore_k(NumPunti, K),
  valuta_punti(Dataset, K).

/* Predicato che acquisisce il valore k dei vicini:
   - il suo unico argomento è il numero dei punti nel dataset. */

leggi_valore_k(NumPunti, K) :-
  nl, write('Inserisci il valore per k: '), nl,
  read(Input),
  (number(Input), Input > 0, Input =< NumPunti ->
   K = Input
   ;
   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,
   leggi_valore_k(NumPunti, K)  % Ricorsione: chiede di nuovo all'utente il valore di k
  ).

/* Predicato per acquisire un dataset di punti etichettati. */

leggi_dataset_etichettato(Lista) :-
```

```
  nl, write('Inserisci i punti del dataset nel formato '),
  write('[(x1,y1,<classe>), ..., (xn,yn,<classe>)]: '), nl,
  read(Input),
  (
   % Controllo che la lista sia nel formato corretto
    (\+ is_list(Input)
     ;
     \+ forall(member(Punto, Input), punto_etichettato_valido(Punto)))
   ->
     nl, write('Input invalido. Assicurati di inserire una lista nel formato '),
     write('[(x1,y1,<classe>), ..., (xn,yn,<classe>)].'), nl,
     leggi_dataset_etichettato(Lista)
   ;
     % Controllo che ci siano almeno due punti
     length(Input, NumPunti),
     NumPunti < 2
   ->
     nl, write('Input invalido. Inserisci almeno due punti.'), nl,
     leggi_dataset_etichettato(Lista)
   ;
     % Controllo che tutti i punti siano distinti
     \+ punti_etichettati_distinti(Input)
   ->
     nl, write('Input invalido. Tutti i punti devono essere distinti.'), nl,
     leggi_dataset_etichettato(Lista)
   ;
     Lista = Input
  ).

/* Predicato per validare un punto etichettato:
   - il suo unico argomento è il punto in questione. */

punto_etichettato_valido(Punto) :-
  Punto = (X, Y, C),
  number(X),
  number(Y),
  atom(C).

/* Predicato che verifica se tutti i punti etichettati sono distinti:
   - il suo unico argomento è la lista di punti etichettati. */

punti_etichettati_distinti([]).
punti_etichettati_distinti([(X, Y, _)|Coda]) :-
  \+ member((X, Y, _), Coda),
  punti_etichettati_distinti(Coda).


/* Predicato che cicla continuamente per valutare nuovi punti sul KNN:
   - il primo argomento è la lista di punti etichettati;
   - il secondo argomento è il numero k dei vicini. */

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 che acquisisce un punto di test. */

leggi_punto_etichettato(Punto) :-
  nl, write('Inserisci il valore del punto da testare nel formato (x,y): '), nl,
  read(Input),
  (punto_valido(Input) ->
   Punto = Input
   ;
   nl, write('Punto non valido. Assicurati di inserire un punto '),
   write('nel formato (x,y).'), nl,
   leggi_punto_etichettato(Punto)
  ).

/* Predicato che trova i k vicini per un punto:
   - il primo argomento è il numero k dei vicini;
   - il secondo argomento è il punto di test;
   - il terzo argomento è il dataset di punti etichettati. */

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 che calcola la distanza tra il punto di test e tutti gli altri punti:
   - il primo argomento è il punto di test;
   - il secondo argomento è il dataset di punti.
   Crea un'associazione distanza-punto, annotando ogni punto del dataset con la
   sua distanza dal punto di test. */

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:
   - il primo argomento è il primo dei due punti;
   - il secondo argomento è il secondo dei due punti. */

distanza((X1, Y1, _), (X2, Y2), Dist) :-
  Dist is sqrt((X1 - X2) * (X1 - X2) + (Y1 - Y2) * (Y1 - Y2)).

/* Predicato che ordina le distanze in ordine crescente:
   - il suo unico argomento è la lista di associazioni Dist-Punto */

ordina_distanze(Distanze, DistanzeOrdinate) :-
  keysort(Distanze, DistanzeOrdinate).

/* Predicato che prende i primi K elementi di una lista:
   - il suo unico argomento è la lista summenzionata. */
```

```
prendi(0, _, []).
prendi(K, [X | Xs], [X | Resto]) :-
  K > 0,
  K1 is K - 1,
  prendi(K1, Xs, Resto).

/* Predicato che elimina l'informazione sulla distanza dal formato Dist-Punto
   per ottenere solamente il punto etichettato:
   - il suo unico argomento è la lista di associazioni Dist-Punto */

converti_coppie_in_vicini([], []).
converti_coppie_in_vicini([_-Vicino | CoppieRestanti], [Vicino | ViciniRestanti]) :-
  converti_coppie_in_vicini(CoppieRestanti, ViciniRestanti).

/* Predicato che stampa tutti i vicini:
   - il suo unico argomento è la lista di vicini. */

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:
   - il suo unico argomento è la lista di vicini. */

trova_classe_maggioranza(Vicini, ClasseMaggioranza) :-
  conta_occorrenze_classe(Vicini, MapConteggi),
  classe_max_occorrenze(MapConteggi, ClasseMaggioranza).

/* Predicato che conteggia le occorrenze di ogni classe nei vicini:
   - il primo argomento è la lista di vicini;
   - il secondo argomento è la mappa di conteggi corrente (inizialmente vuota).
   La mappa di conteggi è una lista di coppie, dove ogni coppia è composta da
   una classe e il suo corrispondente conteggio. */

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).

/* Predicato che aggiorna una mappa di conteggi:
   - il primo argomento è la classe di cui aggiornare il conteggio;
   - il secondo argomento è la mappa di conteggi corrente. */

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 che trova la classe con il maggior numero di occorrenze:
   - il suo unico argomento è la mappa dei conteggi. */

classe_max_occorrenze([(Classe, Conteggio) | Resto], ClasseMaggioranza) :-
  classe_max_occorrenze_aux(Resto, Classe, Conteggio, ClasseMaggioranza).

/* Predicato ausiliario usato dal predicato principale classe_max_occorrenze:
   - il primo argomento è la lista di elementi rimanenti nella nella mappa conteggi;
   - il secondo argomento è la classe con conteggio massimo trovata finora;
   - il terzo argomento è il conteggio massimo trovato finora per suddetta classe.
   Se esiste un'altra classe nella mappa con conteggio superiore a ConteggioMax,
   quella diventa la nuova classe di maggioranza.   */

classe_max_occorrenze_aux([], ClasseMaggioranza, _, ClasseMaggioranza).
classe_max_occorrenze_aux([(Classe, Conteggio) | Resto],
                      ClasseMax, ConteggioMax, ClasseMaggioranza) :-
  (Conteggio > ConteggioMax ->
   classe_max_occorrenze_aux(Resto, Classe, Conteggio, ClasseMaggioranza)
   ;
   classe_max_occorrenze_aux(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 che stampa un certo numero di trattini su una linea orizzontale:
   - il suo unico argomento è il numero di trattini.*/

stampa_riga_orizzontale :- stampa_riga_orizzontale(65).

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 dataset dove sono presenti dei punti duplicati.
- L'utente inserisce un valore per la coordinata x che non sia un numero reale.
- L'utente inserisce un punto di test 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