Alberi di Decisione e Potatura

Mihail Teodor Gurzu
5940299

13 aprile 2020

1 Intento

In questo elaborato si implementa in Python un algoritmo per l'apprendimento di Alberi di Decisione utilizzando *l'entropia* come misura dell'impurità, e successivamente un algoritmo per la potatura basato sull'errore sul *validation set*.

Il codice viene poi applicato a tre datasets e vengono ricavati e mostrati i risultati ottenuti (numero dei nodi, precisione sul test set) prima e dopo la potatura.

2 Implementazione

2.1 Alberi di Decisione

Gli Alberi di Decisione ricoprono un ruolo molto importante nell'ambito del *Machine Learning* e vengono utilizzati per costruire dei modelli per vari tipi di problemi come la *regressione* e la *classificazione*.

L'apprendimento di Alberi di Decisione consiste nel costruire un albero di decisione partendo da un dataset di dati in modo tale da poter predire certe caratteristiche sui dati futuri, con l'obiettivo di massimizzare la precisione e minimizzare lo spazio occupato in memoria.

Infatti uno dei principali problemi di questa struttura è che essa aumenta in modo esponenziale con l'aumento degli attributi e dei relativi possibili valori. Bisogna quindi adottare una certa tecnica nella costruzione di tali strutture e quella che è stata adottata in questo elaborato si basa sul concetto di *information gain* utilizzando *l'entropia* come misura delle impurità.

```
def ID3(examples, default):
   ...
    else:
        best = chooseAttr(allAttr, examples, allAttr[last])
        t = Node(best, {})
   ...
        return t
```

Come si vede ID3 usa la funzione *chooseAttr* per scegliere il miglior attributo in base al miglior valore dell' *information gain* che viene mostrato in seguito.

ID3 prende in ingresso l'insieme degli esempi ed un valore di default per l'attributo target e ritorna l'albero come istanza di *Node*.

```
class Node:
  """ seen and pruned attributes are used in the prunning routine"""
  def __init__(self, label, children=None, seen=False, pruned=False):
        self.label = label
        self.children = {} if children == None else children
        self.seen = seen
        self.pruned = pruned
  ...
```

2.1.1 Entropia

L'*entropia* definisce una misura comunemente usata nella teoria dell'informazione che caratterizza l'impurità di una collezione di esempi.

Data per esempio una certa collezione S dove l'attributo target può prendere c differenti valori, l'*entropia* di S relativa a questa classificazione è definita come:

$$Entropy(S) = \sum_{i=1}^{c} -p_i log_2 p_i$$

dove p_i è la proporzione di S appartenente alla classe i.

```
def entropy(examples, attributes, target):
  ...
  i = 0
    for entry in attributes:
        if(target == entry):
            break
        i += 1
    a = attributes[i]
    for entry in examples:
        if((entry[a]) in valfreq):
            valfreq[entry[a]] += 1.0
        else:
            valfreq[entry[a]] = 1.0
    for freq in valfreq.values():
        entrop += (-freq/len(examples)) * math.log(freq / len(examples), 2)
    return entrop
```

2.1.2 Information Gain

L'*Information Gain* di un particolare attributo è la riduzione attesa dell'entropia causata dal partizionare gli esempi relativamente a quel attributo.

Precisamente, l'*information gain* di un attributo A relativo ad una collezione di esempi S è definita come:

$$Gain(S,A) = Entropy(S) - \sum_{v \in Values(A)} \frac{|S_v|}{|S|} Entropy(S)$$

dove *Values(A)* è l'insieme di tutti i possibili valori per l'attributo A e S_v è il sottoinsieme di S per il quale l'attributo A ha valore v.

```
def gain(examples, allAttr, a, target):
    valfreq = {}
    subs_ent = 0.0
    for entry in examples:
        if((entry[a]) in valfreq):
            valfreq[entry[a]] += 1.0
        else:
            valfreq[entry[a]] = 1.0
    for val in valfreq.keys():
        valprob = valfreq[val] / sum(valfreq.values())
        subs = [entry for entry in examples if entry[a] == val]
        subs_ent += valprob * entropy(subs, allAttr, target)
    return(entropy(examples, allAttr, target) - subs_ent)
```

2.2 Potatura

Un' altro problema caratteristico degli Alberi di Decisione riguarda l'*overfitting* dei dati; ciò avviene quando l'Albero di Decisione si specializza troppo sul *training set* a scapito della generalizzazione sul problema in questione. Ciò si manifesta in una precisione altissima sul

training set che cala drasticamente quando consideriamo il *test set*, cioè dati nuovi, mai incontrati prima.

Per ovviare a questo problema, applichiamo la *'potatura'* all'albero con l'intento di despecializzarlo riguardo al *training set* aumentandone la generalizzazione e di conseguenza la precisione sul *test set*.

La tecnica utilizzata in questo elaborato si chiama *REDUCED ERROR PRUNNING* che descriviamo brevemente di seguito.

2.2.1 Reduced Error Prunning

Per applicare questa tecnica, dobbiamo innanzitutto suddividere il nostro *training set* in due parti: il *grow set* utilizzato per creare l'albero e il *validation set*, usato per decidere se potare un sottoalbero avente radice un certo nodo oppure no. Questa tecnica richiede avere a disposizione un numero abbastanza elevato di esempi.

La potatura, quindi consiste nello considerare, uno per uno, ciascun nodo non foglia e sostituire al relativo sottoalbero la più comune classificazione sul *grow set* relativo a quel nodo. Il nodo diventa quindi un candidato per la potatura se la precisione sul *validation set* dell'albero potato non è inferiore a quella dell'albero originale sul *validation set*. Dopo aver applicato questa procedura a tutti i nodi non foglia, si procede con la potatura per il nodo che massimizza l'aumento della precisione sul *validation set*. A questo punto, l'intero procedimento viene ripetuto finché c'è un qualche nodo che si candida per la potatura.

Adesso, prima di far vedere l'implementazione precisiamo che la classe *Data Manager* si occupa di suddividere il dataset nelle tre parti che ci servono così composte:

$$growset(70\%) + validationset(30\%) = trainingset$$

$$trainingset(70\%) + testset(30\%) = dataset$$

Inoltre *Data Manager* si occupa anche dell'introdurre rumore nei dati del *grow set* in base ad una certa percentuale, attraverso il metodo *introduce_ noise()*

Per quanto riguarda l'implementazione della potatura, ne sono state fatte due, dove la prima è quella descritta sopra e che come si può intuire, quando l'albero diventa grande ha un tempo di esecuzione molto grande. La seconda implementazione si differenzia dalla prima col fatto che la procedura pota il primo nodo che incontra e che aumenta la precisione sul validation set (partendo dai nodi a profondità maggiore) e poi ricomincia da capo la ricerca dei nodi da potare, terminando quando non ne trova alcuno che migliori la precisione sul validation set con la conseguenza che il tempo di esecuzione viene così estremamente ridotto. La differenza principale nel codice sta nel metodo *managePrune()*.

Per primo facciamo vedere l'implementazione originale:

```
def _manage_prune(node, validation_set, grow_set):
    pruned_nodes = False
    iteration = True
    tmp_best_tree = copy.deepcopy(node)
    best_tree = node
    best_performance = test(node, validation_set)
    while iteration:
        iteration = False
        new_tree = copy.deepcopy(tmp_best_tree)
        iteration = _prune(new_tree, tmp_best_tree, grow_set, iteration)
        if iteration:
            new_performance = test(new_tree, validation_set)
            if new_performance >= best_performance:
                best_tree = new_tree
                best_performance = new_performance
                pruned_nodes = True
    reset_tree(best_tree)
```

```
    if pruned_nodes:
        best_tree, best_performance, pruned_nodes = _manage_prune(best_tree,
    validation_set, grow_set)
    return best_tree, best_performance, pruned_nodes
```

Come si può notare dal codice, la procedura nel caso peggiore ha un andamento asintotico pari a $O(n^2)$ dove n è il numero dei nodi dell'albero.

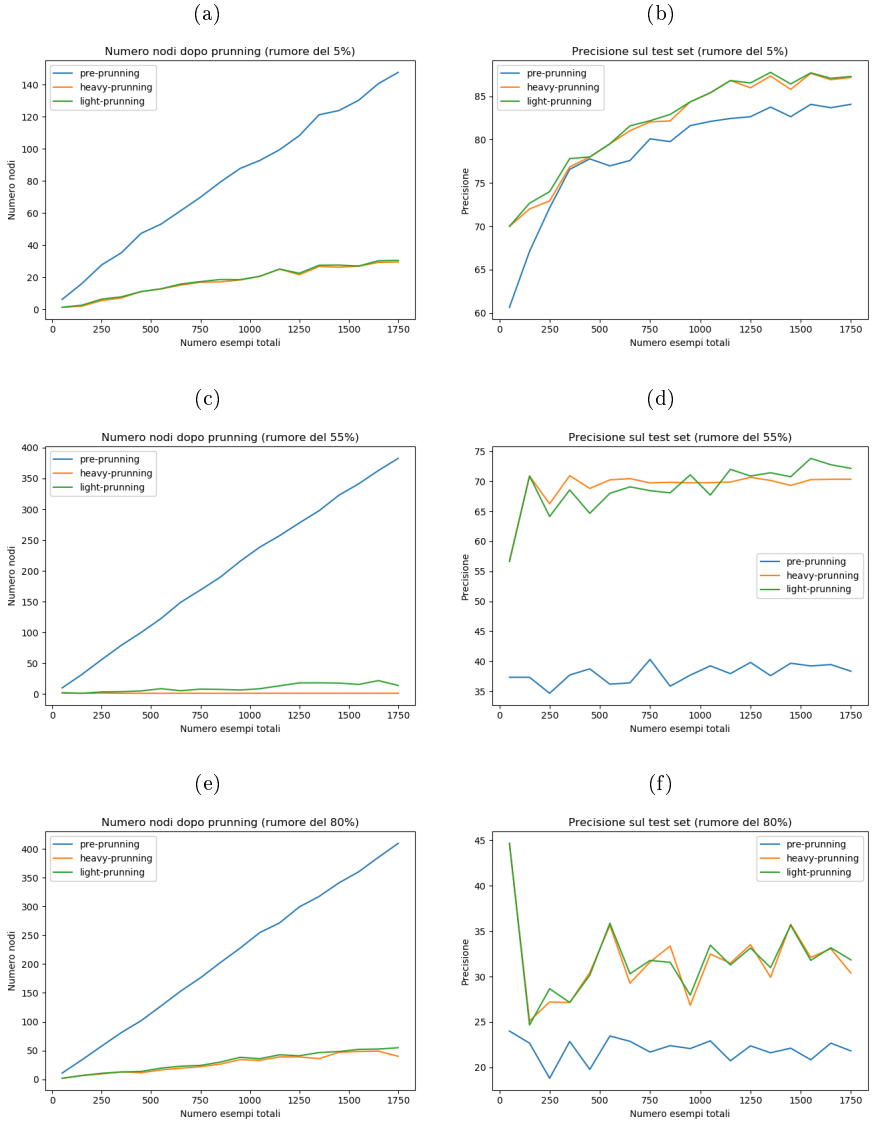
```
def manage_prune(node, validation_set, grow_set):
    nr_pruned_nodes = 0
    iteration = True
    best_tree = node
    best_performance = test(node, validation_set)
    while iteration:
        iteration = False
        new_tree = copy.deepcopy(best_tree)
        iteration = prune(new_tree, best_tree, grow_set, iteration)
        new_performance = test(new_tree, validation_set)
        if iteration and new_performance >= best_performance:
            best_tree = new_tree
            best_performance = new_performance
            nr_pruned_nodes += 1
    return best_tree, best_performance, nr_pruned_nodes
```

Questa versione invece ha un andamento asintotico pari a $\Theta(n)$ con n il numero dei nodi.

Entrambe le implementazioni fanno uso di *prune()*:

```
def prune(node, original_node, examples, pruned):

    if len(node.children) == 0:
        return pruned
    if node.pruned == True:
        return pruned

    if len(examples) == 0:
        node.children = {}
        node.label = mode(examples, T)
        return pruned

    subsets = subset(examples, node.label)

    for child in node.children.keys():
        if pruned == True:
            node.seen = False
            return pruned
        if node.children[child].children != {} and node.children[child].pruned !=
     True:
            node.seen = True
        if child in subsets.keys():
            pruned = prune(node.children[child], original_node.children[child],
    subsets[child], pruned)
        else:
            pruned = prune(node.children[child], original_node.children[child],
    {}, pruned)
    if node.seen:
        node.seen = False
        pruned = True
        return pruned

    node.children = {}
    ex_label = node.label
    node.label = mode(examples, T)
    node.pruned = True
    original_node.pruned = True
    pruned = True
    return pruned
```

Figura 1: heavy prunning è l'implementazione originale, light prunning quella asintoticamente più veloce. I grafici si riferiscono al dataset *cardata*.



37

In Figura **??** è mostrata la differenza tra le due procedure riguardo il numero dei nodi dopo la potatura e la precisione sul test set (i risultati sono stati ottenuti applicando le due procedure allo stesso albero di decisione):

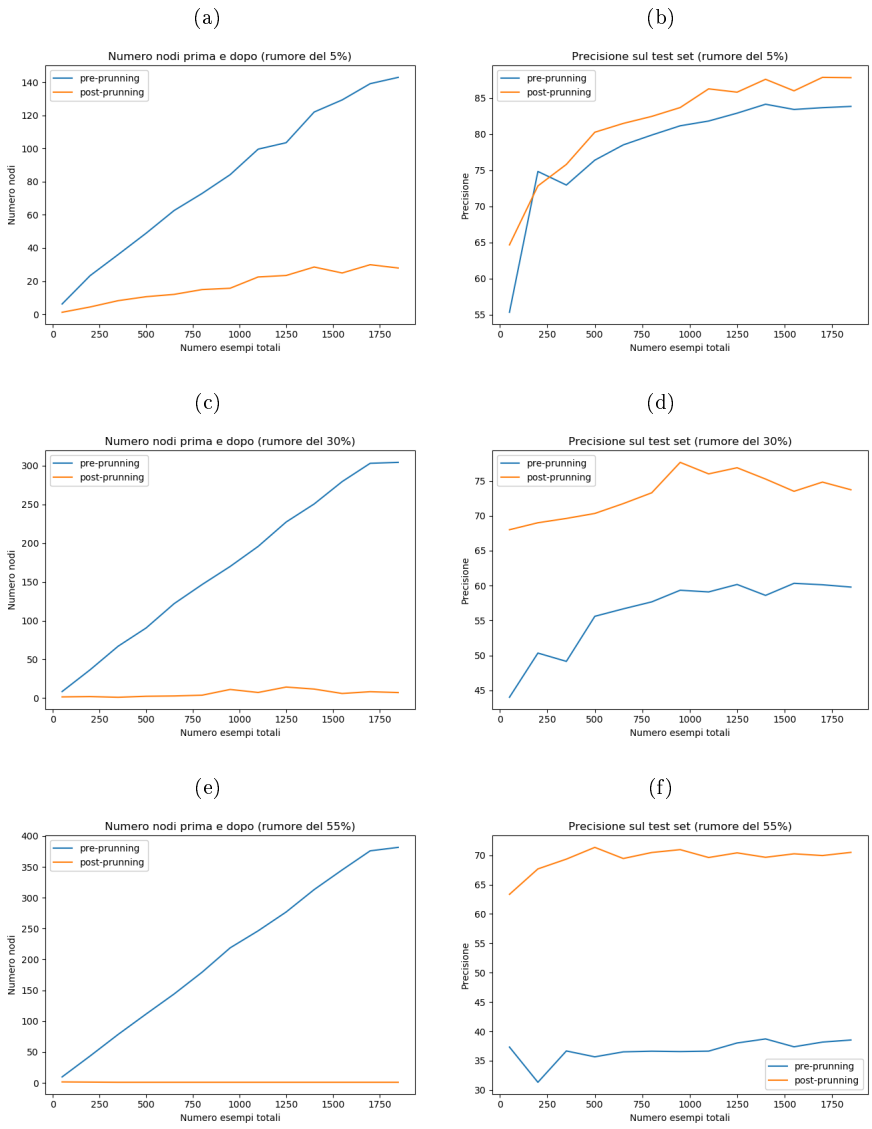
Come si può vedere, le due procedure sono pressoché equivalenti nei risultati ottenuti.

Dato che le differenze tra le due procedure è piccola e dato che l'andamento asintotico dell'implementazione *'light'* è molto minore dell'altra, i risultati degli esperimenti sul dataset *nursery_admission_school* sono stati ottenuti attraverso l'applicazione della procedura *'light'*.

3 Risultati esperimenti

3.1 Cardata

Figura 2: Risultati sul dataset *cardata*.

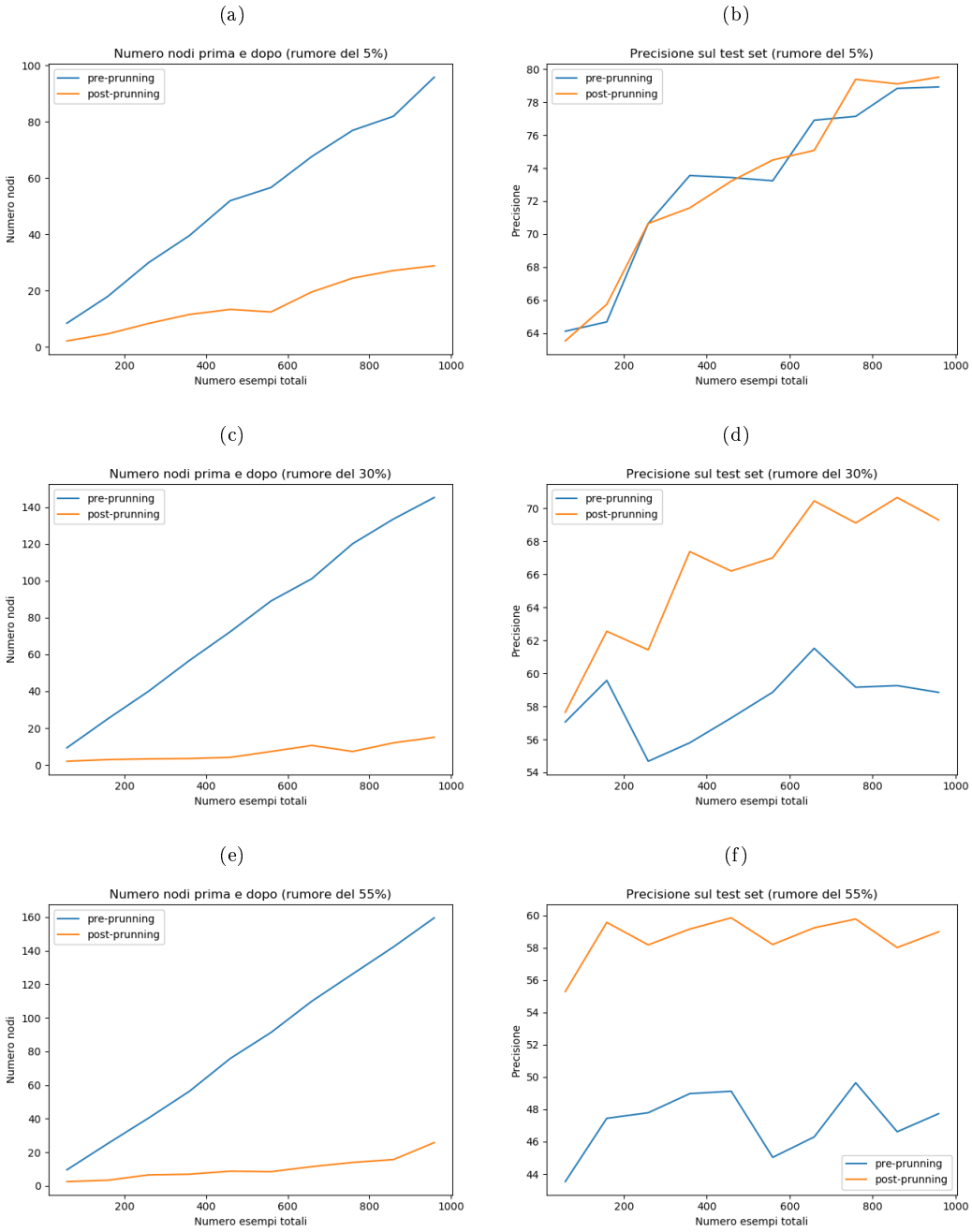


Come si evince dalla Figura **??**, la potatura riduce al minimo il numero dei nodi aumentando la precisione sul test set.

3.2 tic-tac-toe

Nella Figura **??** sono mostrati i risultati sul dataset *tic-tac-toe*.

Figura 3: Risultati sul dataset *tic-tac-toe*.



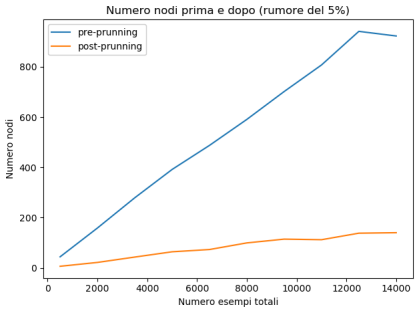
3.3 Nursery School Admission

Dalla Figura **??** notiamo che dopo la potatura il numero dei nodi viene ridotto al minimo, ed anche la precisione sul test set viene sostanzialmente aumentata.

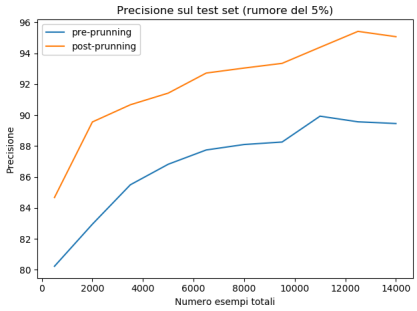
Dai risultati di tutti i dataset si evince che la precisione massima si ha per un insieme di esempi elevato. Più sono gli esempi del dataset iniziale e di conseguenza più sono i nodi

Figura 4: Risultati sul dataset *nursery school admission*.

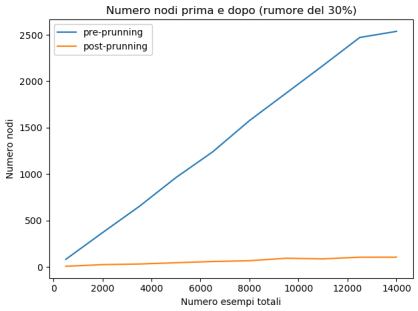
(a)



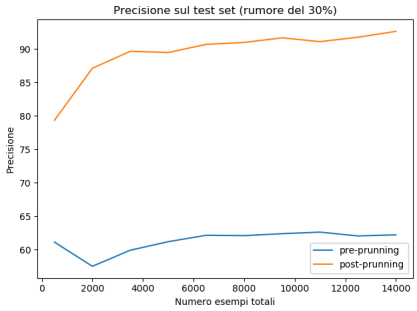
(b)



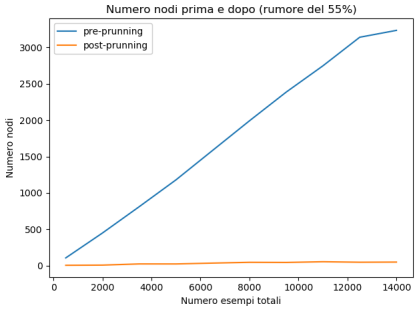
(c)



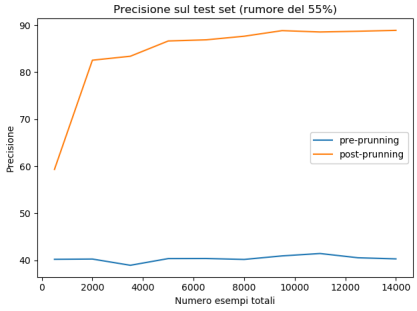
(d)



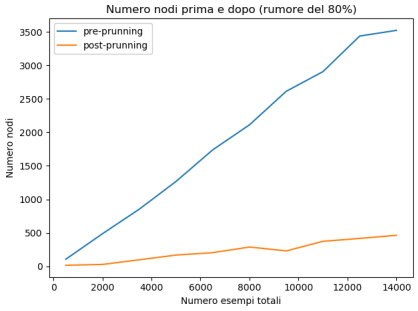
(e)



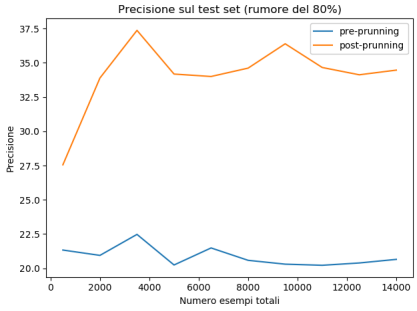
(f)



(g)



(h)



dell'albero risultante, maggiore risulta la precisione sul test set.

4 Come usare il codice

```
def main(dataset):
    '''
    elaborate() function generates a decision tree based on file dataset and then
     pruns it plotting the results (True = light prunning; False = heavy prunning
    )
    compare() function compares the two implementations of prunning (light and
    heavy)
    '''
    elaborate(dataset, 5, 50, 3, True)
    compare(dataset, 5, 50, 3)

main("cardata")
```

Per usare il codice, bisogna passare alla funzione main() il dataset su cui si desidera operare. Il dataset deve avere come prima riga gli attributi e l'attributo target deve occupare l'ultima colonna.

```
def elaborate(dataset, initial_error_rate, initial_examples_nr,
    number_of_iterations, light)
...
def compare(dataset, initial_error_rate, initial_examples_nr,
    number_of_iterations)
```

I parametri delle due funzioni rappresentano:

- *dataset* - denota il dataset su cui operare;
- *initial_error_rate* - denota la percentuale dell'errore da introdurre nel grow set (la percentuale aumenta dello 25% ad ogni iterazione del ciclo principale delle rispettive funzioni);
- *initial_examples_nr* - denota il numero iniziale di esempi presi casualmente dall'intero dataset sul quale costruire i tre dataset(grow set, validation set e test set) usati dall'elaborazione. Il numero di esempi viene aumentato ad ogni iterazione interna di un numero prefissato;
- *number_of_iterations* - denota il numero delle iterazioni su cui fare la media (i grafici mostrati qui sono ottenuti come media su 10 iterazioni);
- *light* - se **True** verrà usata l'implementazione *'light'* della procedura di potatura; se **False** verrà usata quella *'heavy'*;