Random Forest

Marco Miani

Gennaio 2020

1 Implementazione

La struttura per la Random Forest è implementata nel file *"Random_Forest.py"*.

1.1 Foresta

L'oggetto Foresta è semplicemente una lista di alberi, e tiene in memoria alcuni valori utili. L'inizializzazione richiede i parametri:

- ***predicted_feature*** che deve essere una delle features del dataframe, quella che vogliamo classificare.
- ***points*** il dataframe
- ***numero_alberi*** il numero di alberi nella foresta
- ***fraz_conoscenza_albero*** la precentuale di dataframe con cui allenare ogni albero, troppo alta porta ad alberi troppo uguali tra loro, troppo bassa porta ad alberi mal allenati a causa dello scarso numero di datapoints

Sono implementate le funzioni di cui abbiamo bisogno:

- ***.train****(numero_split=1)*
  allena la random forest aggiungendo *numero_split* ulteriori nodi a ciascun albero, di default 1 per verificare passo passo l'aumento delle prestazioni. Notiamo che l'altezza degli alberi è sempre limitata dall'altro dal numero di nodi. Il parametro aggiuntivo *stampa* consente di visualizzare l'andamento del training.
  La complessità è $O(n_{datapoints} * \log(n_{datapoints}) * n_{features} * n_{alberi} * \log(n_{split}))$
- ***.guess****(point)*
  raccoglie i risultati di ciascun albero per la classificazione dell'elemento *point*, restituisce la media dei risultati, una possibile alternativa di facile implementazione era il voto a maggioranza.
  Il parametro aggiuntivo *numero_votanti* permette di far votare solo i primi n alberi, utile per confrontare le performance senza allenare di nuovo tutta la foresta.
  La complessità è $O(n_{alberi} * \log(n_{split}))$

1.2 Albero

L'oggetto Albero è una struttura di albero binario. I punti del dataframe vengono divisi in ciascun nodo tra i due figli in base al valore di una determinata feature del datapoint, maggiore o minore di un valore soglia. Di conseguenza le foglie dell'albero costituiscono una partizione dei punti del dataframe.
L'altezza dell'albero nel worst case è $numero_{split}$ ma in media è $\log(numero_{split})$.
Sono implementate le funzioni:

- ***.train****(numero_split=1)*
  aggiunge *numero_split* ulteriori nodi all'albero, di default 1. Per aggiungere ogni nodo cicla su tutti i nodi foglia, per ognuno calcola lo split ottimale, sceglie quello che porta al miglioramento maggiore nel Gini ed aggiunge il nodo che implementa quello split. Di fatto non ricalcola tutto ogni volta ma riutilizza valori calcolati precedentemente.
  La complessità è $O(n_{datapoints} * \log(n_{datapoints}) * n_{features} * \log(n_{split}))$

- ***.guess****(point)*
  raggiunge il nodo foglia relativo al datapoint *point* partendo dalla radice dell'albero e scegliendo ogni volta il nodo figlio in base al valore del datapoint nella feature *nodo.direzione* rispetto al valore soglia *nodo.x*.
  Restituisce il guess per la classificazione, calcolato in base alle occorrenze di 0 e 1 nei punti del dataset della foglia.
  La complessità è $O(\log(n_{split}))$

1.3 Nodo

L'oggetto Nodo è la struttura base con cui viene costruito l'albero, oltre ai puntatori ai due eventuali figli, contiene il valore ***direzione*** e il valore ***x***, il primo è una stringa tra le features del dataframe, il secondo è un numero.
Inoltre per comodità contiene i puntatori ai datapoints contenuti in quel nodo. Tali datapoints vengono divisi tra i due figli a seconda che il loro valore nella feature *direzione* sia maggiore o minore del valore soglia *x*.
Oltre a questi contiene anche altri valori che è utile memorizzare per non doverli ricalcolare ogni volta, in favore della complessità computazionale. È implementata la funzione:

- ***.split_ottimale****(predicted_feature)*
  cicla su tutte le possibili features e per ciascuna cicla su tutti i possibili valori dei datapoints nella feature corrente. Ogni volta calcola il valore Gini che si otterrebbe con tale split, calcolato come media pesata dei Gini dei figli, chiaramente cacolati relativamente a *predicted_features*.
  Restituisce il valore ottimale, minimo, di Gini ottenuto e il relativo split che lo realizza. Memorizza anche tali valori in modo da non doverli ricalcolare ogni volta.
  La complessità è $O(n_{datapoints} * \log(n_{datapoints}) * n_{features})$

2 Prestazioni

2.1 Titanic Dataframe

Il dataframe Titanic contiene 708 datapoints e 7 features, tra cui *Survived* che è quella che vogliamo classificare.
Per prima cosa, per poter verificare la bontà dei risultati, abbiamo diviso il dataframe in Train Set e Test Set assegnandogli rispettivamente l'80% e il 20% dei datapoints.

Plottiamo l'andamento degli errori, sia in norma 2 che in percentuale, al variare del numero di nodi.
Consideriamo i casi con il numero di alberi che varia in [1,2,4,10,100,1000], per ciascun caso alleniamo gli alberi con una *fraz_conoscenza_alberi* ragionevole in base al numero di alberi.

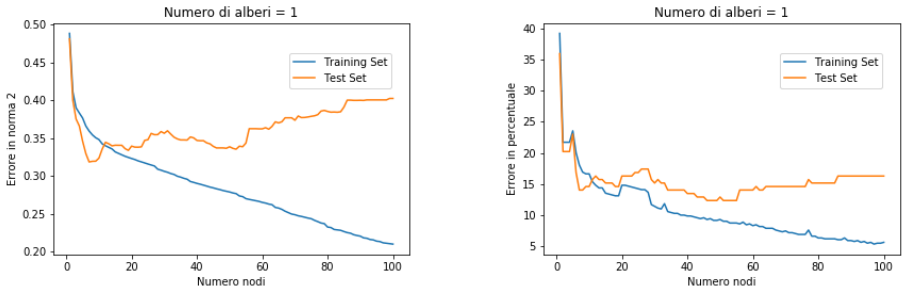


Figure 1: Percentuale dataframe per ciascun albero = 100%

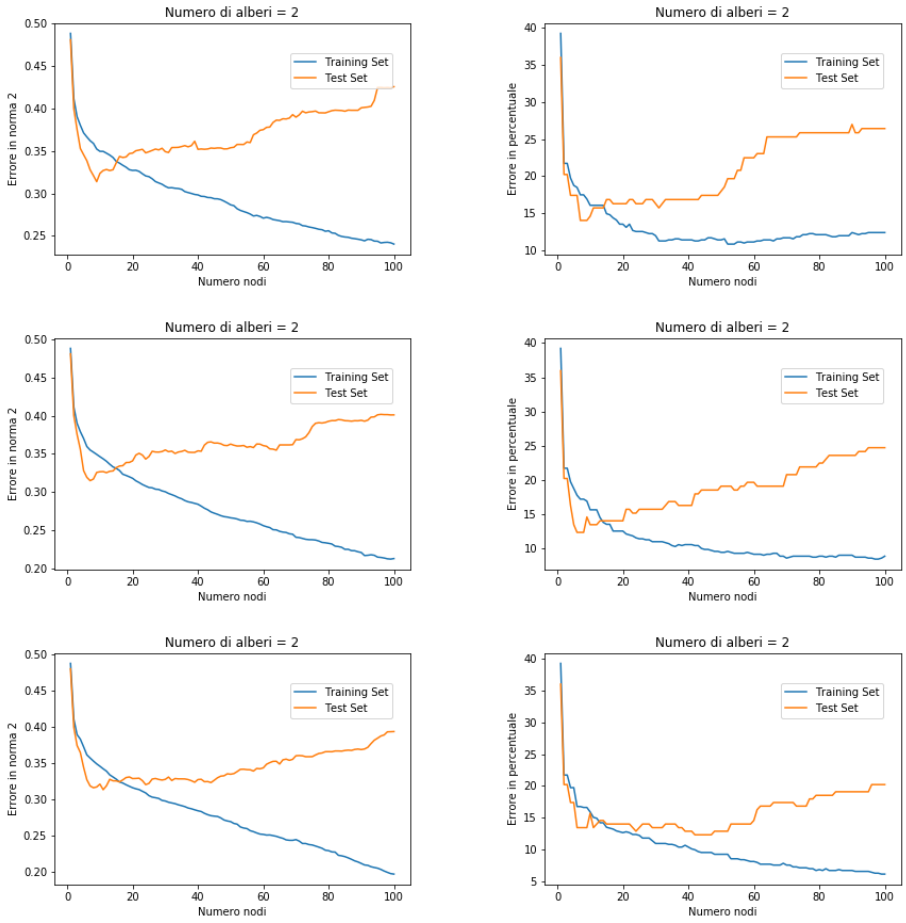


Figure 2: Percentuali dataframe per ciascun albero = [70%, 80%, 90%]

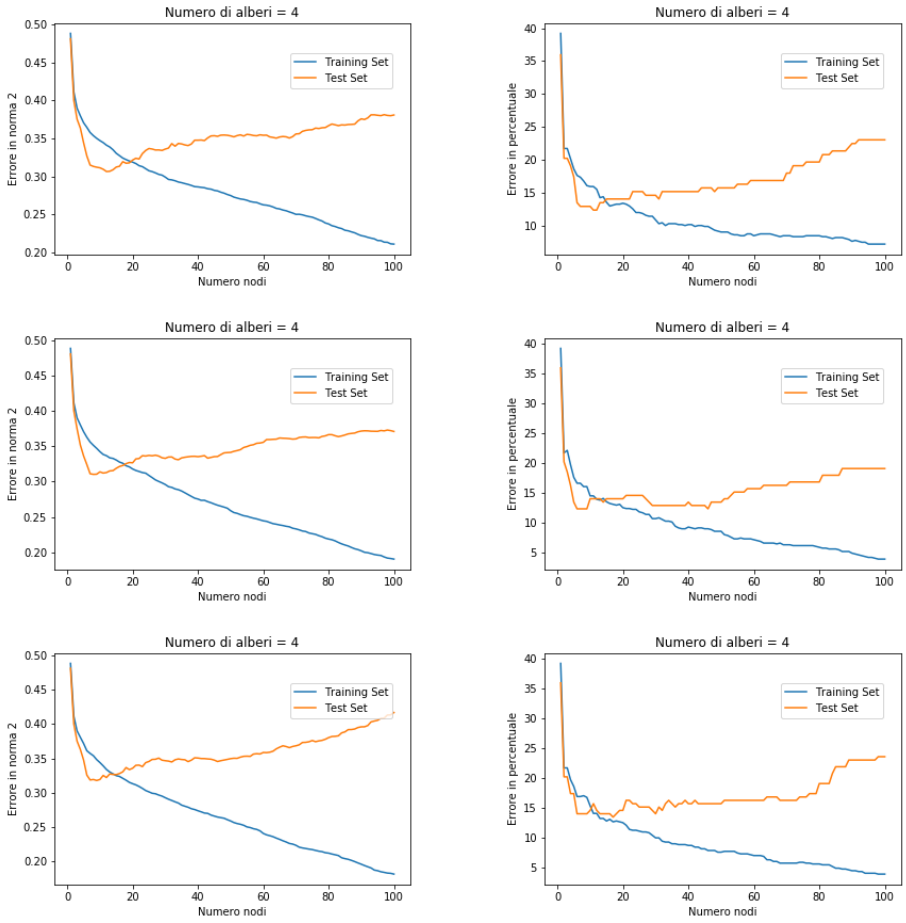


Figure 3: Percentuali dataframe per ciascun albero = [70%, 80%, 90%]

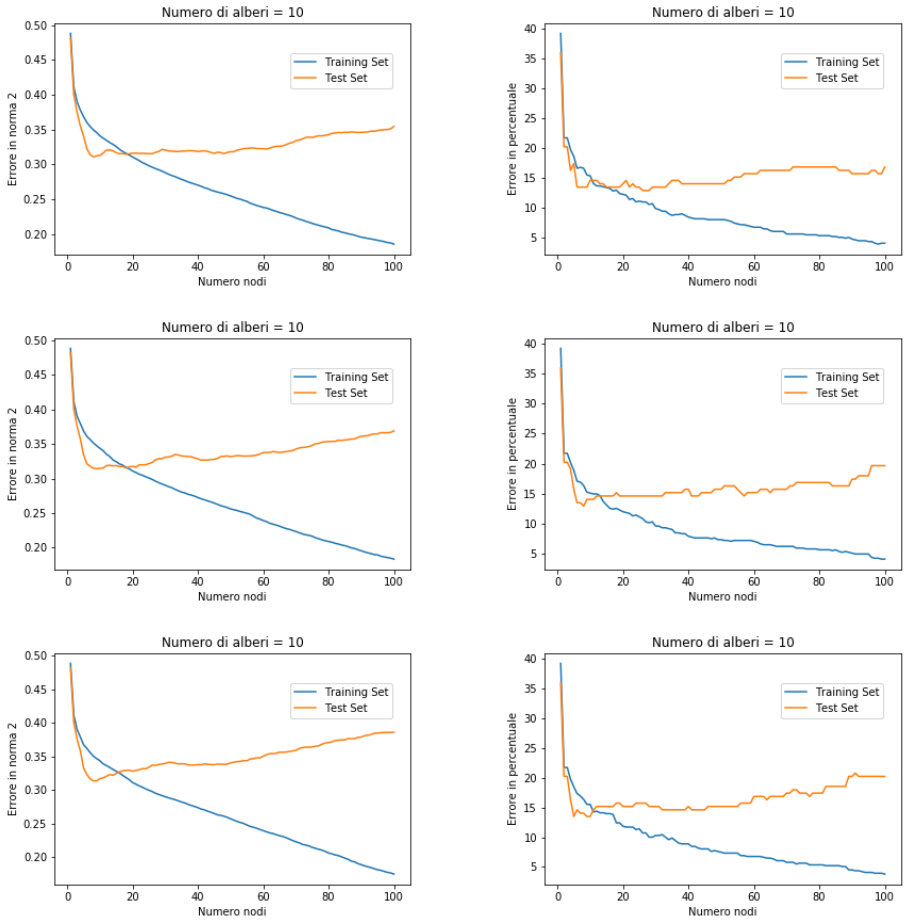


Figure 4: Percentuali dataframe per ciascun albero = [70%, 80%, 90%]

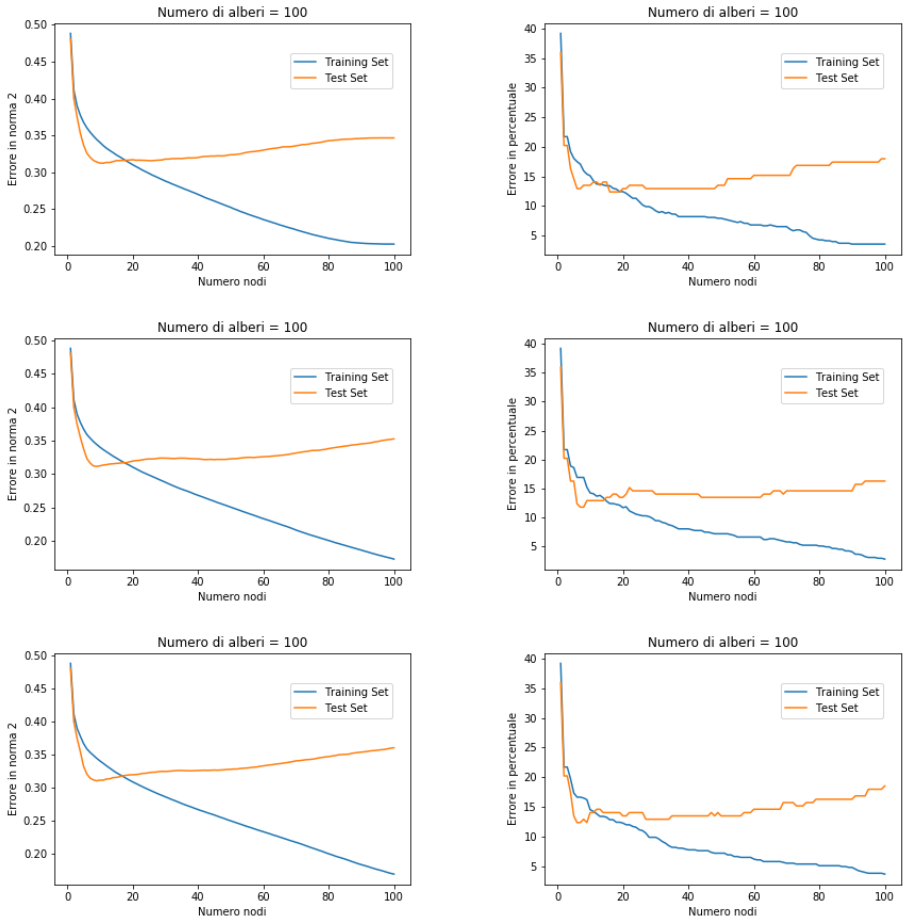


Figure 5: Percentuali dataframe per ciascun albero = [50%, 70%, 80%]

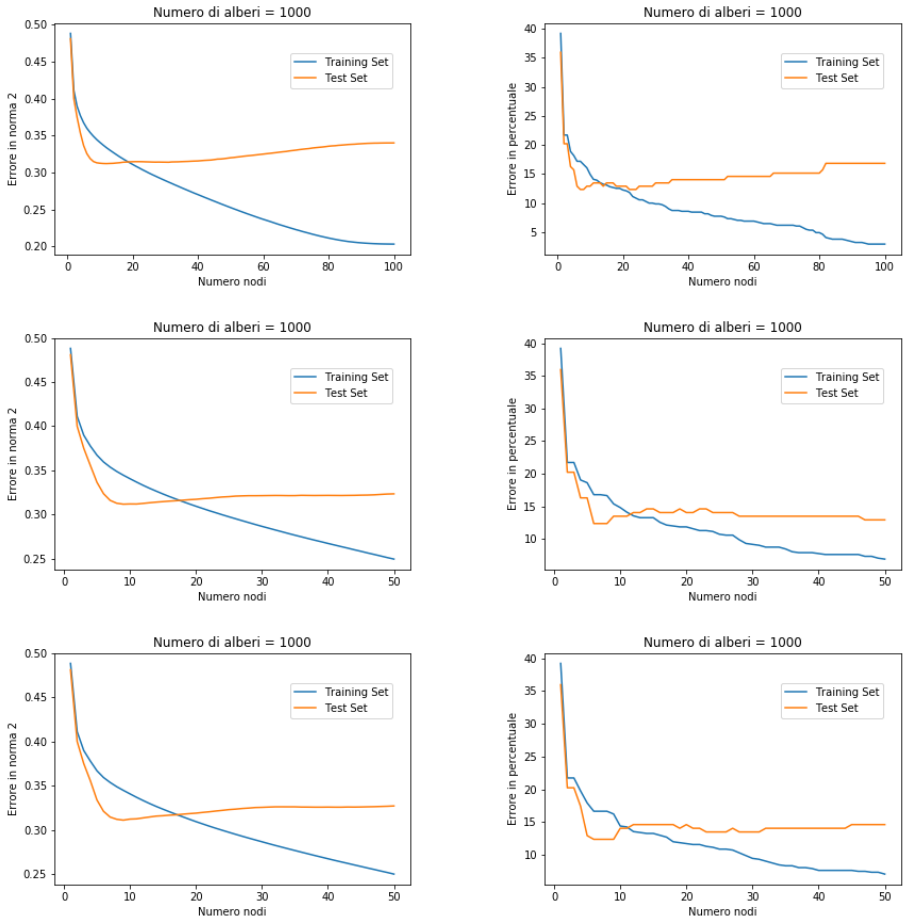
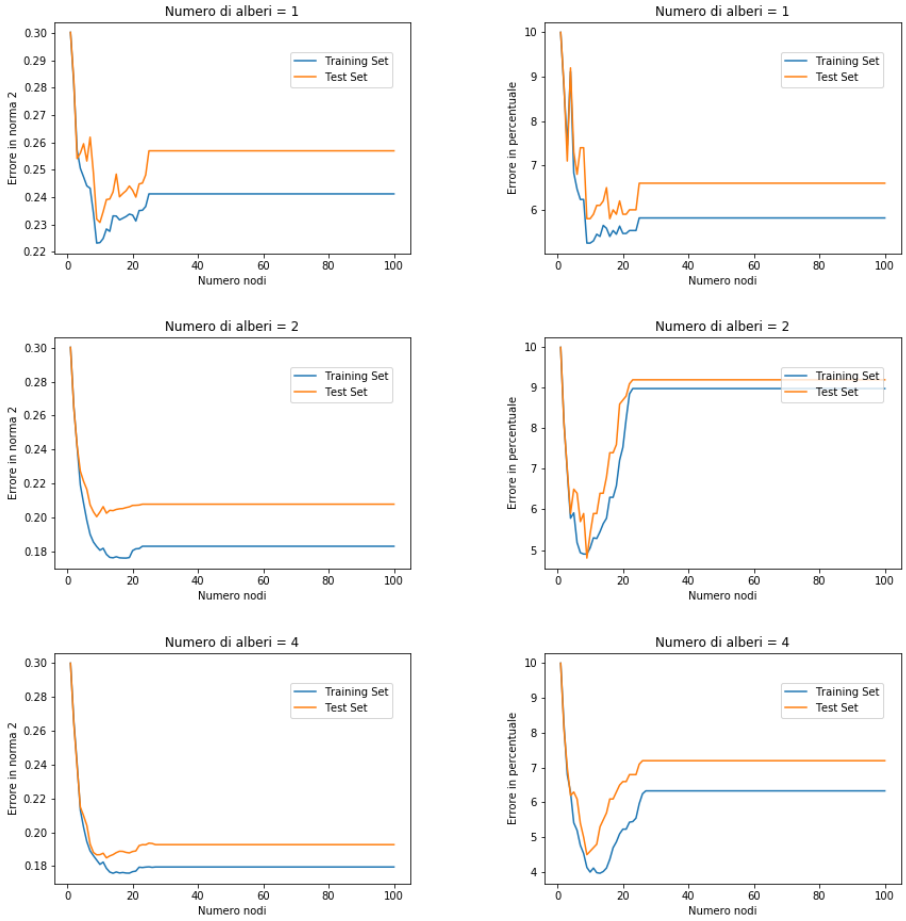


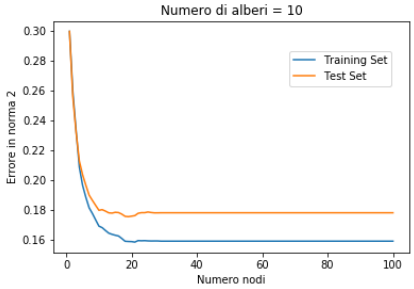
Figure 6: Percentuali dataframe per ciascun albero = [50%, 70%, 80%]

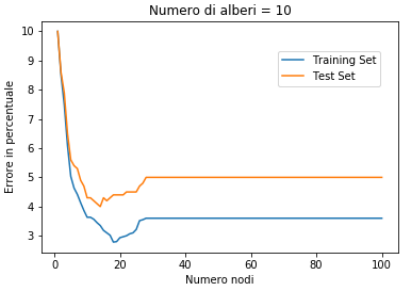
2.1.1 MNIST Dataframe

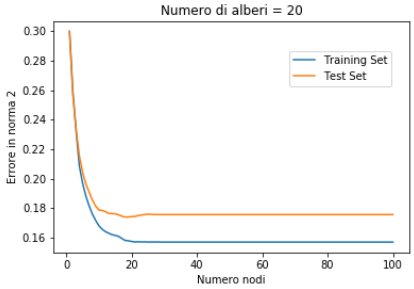
Il dataframe MNIST contiene 785 features, tra cui y che è quella che vogliamo classificare, che rappresenta la cifra nell'immagine. Visto che siamo interessati ad una classificazione binaria poniamo a 1 il valore di y nelle occorrenze di una cifra specifica e a 0 altrimenti. È già diviso in Train Set (*MNISTtrain.csv*) e Test Set (*MNISTtrain.cvs*) che contengono rispettivamente 6000 e 1000 datapoints.

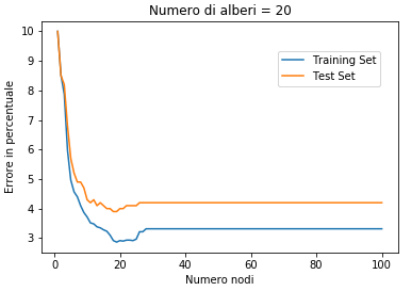
Come prima plottiamo l'andamento degli errori, sia in norma 2 che in percentuale, al variare del numero di nodi.
A causa della maggiore dimensione del dataframe, stavolta consideriamo i casi con il numero di alberi che varia in [1,2,4,10,20,100], il caso con 1000 alberi infatti richiede troppo tempo. A differenza del caso con il dataframe Titanic, sempre a causa della diversa dimensione, fissiamo *fraz_conoscenza_alberi* uguale a 0,1. Ciò significa che ogni albero vede solo il 10% del dataframe, scelta chiaramente in modo casuale.

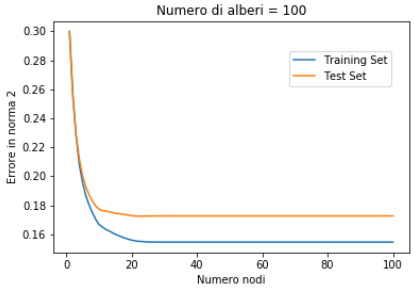


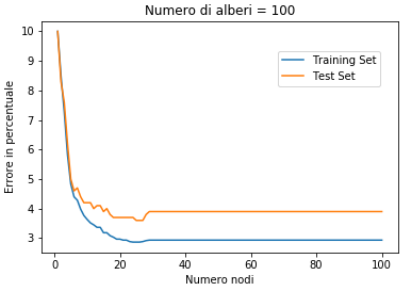

Numero di alberi = 10
Training Set
Test Set
Errore in norma 2
Numero nodi


Numero di alberi = 10
Training Set
Test Set
Errore in percentuale
Numero nodi


Numero di alberi = 20
Training Set
Test Set
Errore in norma 2
Numero nodi


Numero di alberi = 20
Training Set
Test Set
Errore in percentuale
Numero nodi


Numero di alberi = 100
Training Set
Test Set
Errore in norma 2
Numero nodi


Numero di alberi = 100
Training Set
Test Set
Errore in percentuale
Numero nodi