

Progetto IA: RANDOM FOREST RELAZIONE.

Matricola: 5424944 Plesoiu Silviu Robert

PREFAZIONE

Il linguaggio di programmazione scelto per sviluppare il progetto e' stato C# utilizzando il motore grafico Unity.

Motivazioni: Avevo interesse ad imparare il linguaggio C# ed Unity per via di un plug-in,

Open Source del medesimo che si chiama Unity ML-Agents basato sulla PPO (Proximal Policy Optimization).

<https://github.com/Unity-Technologies/ml-agents> (link su github per curiosita')

E' stata implementata anche una GUI per semplificare la riproduzione dei risultati.

Nella relazione, sara' mostrato ogni tanto una parte di codice per fare cercare di fare capire l'essenziale, evitando di spiegare alcune parti meccaniche del codice.

The screenshot shows a user interface for a Random Forest application. At the top, there's a title bar with the text "RandomForest Unifi" in green. On the right side of the title bar is a red button with a white 'X'. Below the title bar, there's a logo of the University of Florence (Universitas Florentina) and a seed input field labeled "Seed: DefaultSeed = 23".

Underneath the title bar, there are two sections for configuration:

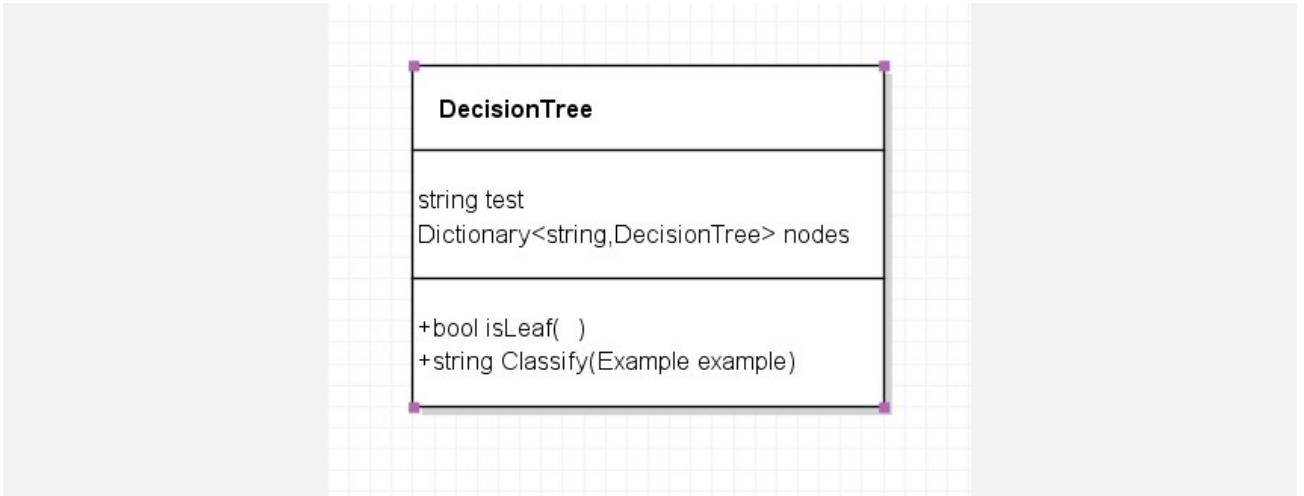
- Measure:** A dropdown menu showing "ENTROPY" with a checked checkbox, and "GINI_INDEX" with an unchecked checkbox.
- Number of Trees:** A slider set to "1". Below the slider are two checkboxes: "OneHotEncoding" and "Single Random Input", both of which are unchecked.

Below these configuration sections is a heading "Select DataSet" in green. Underneath this heading are five dataset options, each enclosed in a light green box:

- GermanCredit Card
- Breast-Cancer
- IonoSphere
- Diabetes
- Vehicle

Each dataset option has a small green tree icon underneath it.

Come prima cosa, visto che RandomForest si basa sul costruire tanti alberi di Decisione senza pruning, costruiti ciascuno su di un bootstrap del dataSet, per poi farli votare scegliendo infine la maggioranza, andiamo a vedere come ho rappresentato il singolo Albero di Decisione.

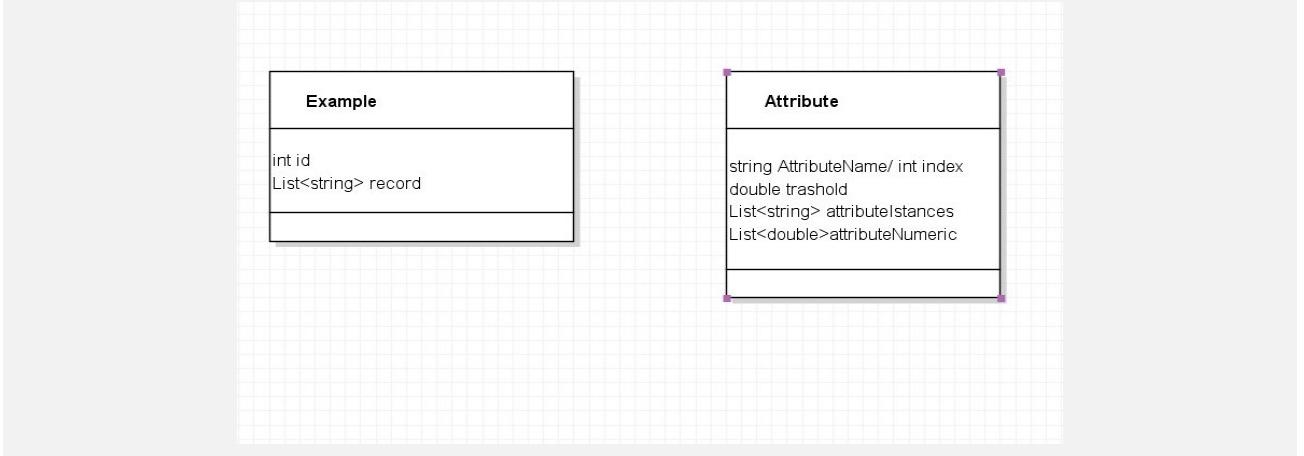


L'albero di Decisione ha un campo stringa test, che rappresenta l'attributoTest scelto per fare un ipotetico split del dataSet durante la fase di costruzione dell'albero.

I nodi vengono rappresentati come un Dizionario<stringa, albero di decisione> (dove il tipo stringa rappresenta una delle istanze dell'attributoTest, nel caso questi sia categorico) ho preferito l'utilizzo di un dizionario ad una lista, per facilitare il tracciamento del corretto path sull'albero quando viene chiamato il metodo Classify(Example example).

Il metodo isLeaf() ritorna true quando il dizionario nodes non contiene alcun elemento, se questo si verifica si deve verificare allo stesso tempo che il campo stringa "test" non contenga piu' un attributoTest ma una delle classiTarget da classificare!

Prima di guardare il metodo Classify(Example example) diamo una occhiata alle classi Example e Attribute.



Ciascun record del dataset viene wrappato da una classe Example. I valori del record vengono messi in una lista di stringhe e ciascuna classe Example ha un id identificativo, questo serve per distinguere record che si ripetono all'interno del Dataset (mi e' tornato utile quando ho testato se il metodo bootstrap mi creava effettivamente dei sample diversi l'uno dall'altro).

La classe Attribute ha due liste (sarebbero dei Set in quanto contengono solo i valori distinti delle istanze di quell'attributo) viste come stringhe e numeriche (in caso l'attributo sia di tipo continuo).

L'AttributeName del singolo attributo corrisponde all'indice di colonna del corrispettivo dataSet*.

Tornando all'albero di decisione vediamo un estratto del metodo Classify. (solo un estratto del codice*)

```
while (!nextNode.IsLeaf())
{
    int index = int.Parse(nextNode.test);
    string element = record[index];

    double parse = 0;
    if (double.TryParse(element, out parse))
    {
        element = nextNode.ChooseLink(element);
    }
    if (nextNode.Nodes.ContainsKey(element))
        { nextNode = nextNode.Nodes[element]; }
```

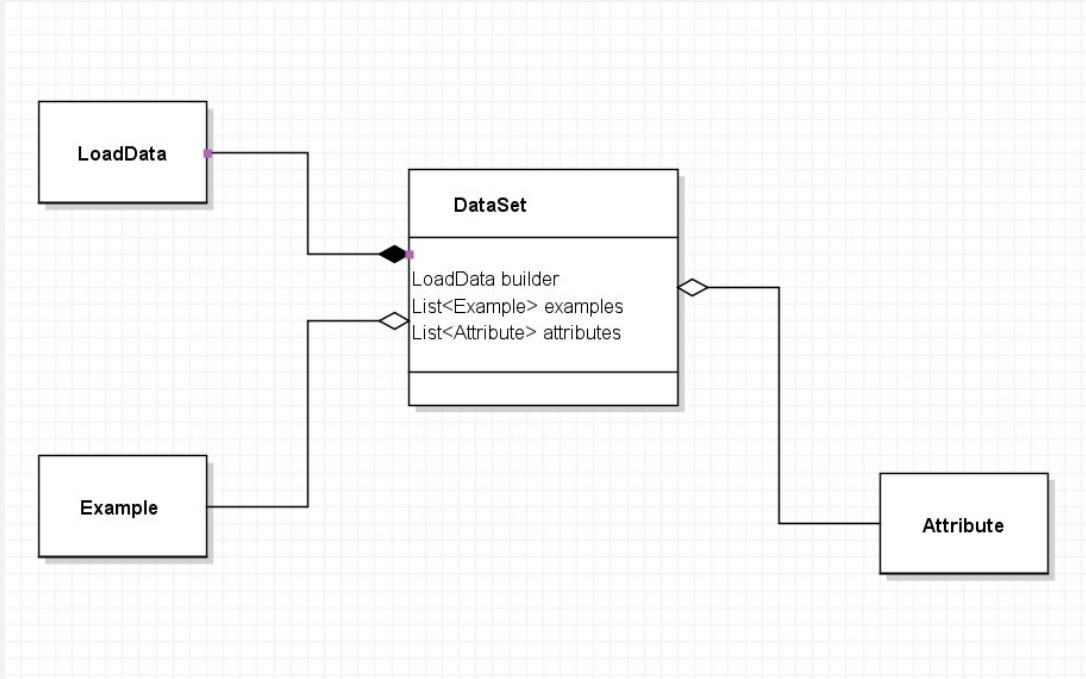
Si fa partire un ciclo while che si ferma solo quando il nodo DecisionTree nextNode e' una foglia.

Per ottenere in O(1) dal record dell'esempio da classificare il corrispondente valore da sottoporre al test del nodo, basta fare un parse del campo test (che ricordiamo corrisponde al nome dell'attributoTest, il quale nome e' un indice di colonna del DataSet).

Se il valore ottenuto e' continuo si utilizza il metodo ChooseLink() che guarda se l'istanza dell'attributo e' maggiore uguale o minore del trashold per vedere a quale prossimo nodo andare (in seguito e' spiegato come viene trovato il trashold e come viene codificato il tutto) altrimenti se il valore e' categorico si ottiene il prossimo nodo semplicemente utilizzando il dizionario nodes, cosi' via finche' l'ultimo nodo e' una foglia.

Estrapolazione dei Dati e creazione della classe DataSet.

(Classi in gioco: DataSet, LoadData, Example, Attribute)



CONVENZIONI: i dataSet utilizzati sono tutti del formato csv e all'interno di essi hanno solo i valori delle istanze degli attributi.

La classe DataSet per costruire i suoi campi "examples" ed "attributes" delega questo compito ad una terza classe LoadData la quale si occupa di estrarre i dati dal file csv attraverso il metodo BuildDataSet:

```

// costruttore
public LoadData(string document, char separator, int target)...

public void BuildDataSet(List<Example>examples, List<Attribute> attributes, int
numberOfAttributes)
{
    setAttributes(attributes, numberOfAttributes);
    TextAsset data_csv = Resources.Load<TextAsset>(document);
    string[] data = data_csv.text.Split(new char[] { '\n' });

    for (int i = 0; i < data.Length-1; i++)
    {
        string[] row = data[i].Split(new char[] { separator }); // ','
        for (int j = 0; j < attributes.Count; j++)
        {
            attributes[j].Add(row[j]);
        }

        Example example = new Example(row,target);
        examples.Add(example);
    }
    foreach(Attribute attr in attributes)
    {
        attr.SetType();
    }
}

```

Il metodo riceve in input due liste vuote, e le riempie estrapolando i dati dal csv nel caso caso di examples.

Quando faccio quel foreach in fondo setto il tipo di ogni attributo (continuo/categorico) con questo metodo SetType().

```

public void SetType()
{
    MyType = numbersInstances.Count > 0 ? MyType = typeOfAttribute.Continuos :
    MyType = typeOfAttribute.Categorical;

}

```

Il tipo degli attributi lo ho realizzato attraverso un Enumerator, ed ogni Attributo ha un campo enum di questo tipo myType.

```
public enum typeOfAttribute { Categorical, Continuos };
```

In questo modo si corre il rischio di dare per scontato che se un attributo ha solo istanze numeriche come un id oppure “0” ed “1” sia di tipo Continuo. Pero’ per i dataSet che ho utilizzato va bene perche’ i dati numeri nei vari csv non erano valori categorici*

DecisionTreeMaker

DecisionTreeMaker e' la classe che si occupa effettivamente di costruire i DecisionTree. Costruttore*

```
public DecisionTreeMaker(DataSet ds, int indexTarget, Measure measure)
{
    count++;
    id = count;
    MyMeasure = measure;
    IndexTargetClass = indexTarget;
}
```

Il costruttore prende una Measure (che potra' essere GINI oppure ENTROPY) che poi utilizzera' come criterio per fare splitting sul DataSet attraverso InformationGain oppure Impurity, un DataSet dal quale prendera' la lista di Example e la lista Attribute che gli servono per costruire l'albero di Decisione e l'indice di colonna della classeTarget da classificare.

Il compito dell'id utilizzato all'interno del costruttore lo spieghero' piu' tardi.

```
public DecisionTree
BuildTree(List<Example>examples, List<Attribute>attributes, List<Example>parentExamples)

function DECISION-TREE-LEARNING(examples, attributes, parent examples)
    returns a tree
    if examples is empty then return PLURALITY-VALUE(parent examples)
    else if all examples have the same classification then return the classification
    else if attributes is empty then return PLURALITY-VALUE(examples)
    else
        A←argmaxa ∈ attributes IMPORTANCE(a, examples)
        tree ← a new decision tree with root test A
        foreach value vk of A do
            exs ←{e : e∈examples and e.A = vk}
            subtree ←DECISION-TREE-LEARNING(exs, attributes -A, examples)
            add a branch to tree with label (A = vk) and subtree subtree
    return tree
```

Lo pesudoCodice presente nel libro di testo, sul quale mi sono basato per implementare il seguente metodo*

Penso che i punti salienti da far vedere sono come abbia implementato il metodo IMPORTANCE() e lo split.

SPLIT

Lo split avviene in maniera diversa a seconda se l'Attributo e' categorico o continuo.

In entrambi i casi si utilizza un metodo sulle collezioni che si chiama `FindAll(Predicate<T> match)` che recupera in una nuova collezione, in questo caso una lista di `Example`, tutti gli elementi che corrispondono alle condizioni definite dal predicato specificato (In ogni caso il predicato viene passato sotto forma di Lambda-Expression).

In caso di Attributo categorico, utilizziamo il metodo `Contains(string s, int index)` della classe `Example`, che controlla data una stringa ed un indice in input se la stringa e' presente nel campo `Record` dell'Esempio.

```
public bool Contains(string s, int index)
{
    return record[index].Equals(s);
}
```

```
private List<Example> SplitDataSet(List<Example> examples, string vk, int attribute)
{
    return examples.FindAll(x => x.Contains(vk,attribute));
}
```

Altrimenti in caso di attributo continuo, si utilizza il metodo `PassTheTest()` sempre della classe `Example`:

```
public bool PassTheTest(string attribute, double test)
{
    int attributeL = attribute.Length;
    int index = int.Parse(attribute);
    string s = record[index];
    double num;
    if (!double.TryParse(s, out num)) Debug.Log(s);
    if (double.Parse(s, CultureInfo.InvariantCulture.InvariantCulture) > test) return true;
    return false;
}
```

```
private List<Example> SplitDataSetForContinuesValues(List<Example> examples, Attribute attr)
{
    return examples.FindAll(x => x.PassTheTest(attr.AttributeName,attr.Trashold));
}
```

```
private List<Example> SplitDataSetForContinuesValues_TestNotPassed(List<Example> examples,
Attribute attr)
{
    return examples.FindAll(x => !x.PassTheTest(attr.AttributeName, attr.Trashold));
}
```

IMPORTANCE

Nel Paper ogni singolo albero di Random Forest quando va a scegliere l'attributo migliore sul quale fare lo split, non lo fa sull'intero set di attributi ma su un sottoinsieme F.

F assume due valori F=1 ed F =int(log2 M+1), dove M e' la dimensione del set di attributi al momento precedente di fare lo split.

Il compito di scegliere gli attributi che appartengono al sottoinsieme F degli attributi e' svolto dal metodo Selection. Questa scelta deve essere fatta in maniera randomica. Nel paper si creavano foreste di 100 alberi. Per assicurarmi di avere alberi il piu' diverso possibile tra loro (ed anche per poter replicare i risultati), viene utilizzato come seed l'id del DecisionTreeMaker, generando cosi' sequenze di elementi di F diverse per ogni albero, questo seed verra' riutilizzato anche in seguito*

```
public List<Attribute> Selection(List<Attribute> attributes)
{
    System.Random rand = new System.Random(id);
    List<Attribute> a = new List<Attribute>();
    int F = (int)Math.Log(attributes.Count, 2)+1;
    for(int i = 0; i < F; i++)
    {
        int index = rand.Next(attributes.Count);
        if (!a.Contains(attributes[index])) { a.Add(attributes[index]); } else
        {
            i--;
        }
    }
    return a;
}
```

```
private Attribute ImportanceInformationGain(List<Attribute> attributes, List<Example> examples)
{
    List<string> targetList = TargetList(examples);
    double greatestGain = 0.0;
    Attribute attributeWithBestGain = attributes[0];
    foreach (Attribute attr in attributes)
    {
        double gain = attr.CalculateInformationGain(examples, targetList);
        if (gain > greatestGain)
        {
            greatestGain = gain;
            attributeWithBestGain = attr;
        }
    }
    return attributeWithBestGain;
}
```

Nel codice potete trovare anche la versione con Importancelmpurity() ma mi limito a spiegare solo il funzionamento della variante con informationGain*

Essenzialmente IMPORTANCE fa una sorta di "cerca il massimo in un vettore.." rappresentato dall'Attributo con l'informationGain piu' alto.

In sintesi calcolo InformationGain/Impurity

Il compito di calcolare l'informationGain (oppure Impurita' del nodo) e' delegato al singolo Attributo.

Se l'attributo e' di tipo categorico calcola direttamente l'informationGain che si avrebbe a splittare su di esso altrimenti, se di tipo continuo, delega questo compito ad una classe chiamata "TrasholdFinder", che calcola per la classe Attributo l'informationGain e setta per esso anche un trasholder che trova nel medesimo modo.

Fa un sort delle istanzeNumeriche dell'attributo. Calcola i middlePoints di questi valori e per ciascun middePoint ne calcola l'informationGain. Il middlePoint con l'informationGain piu' alto diventa il trashold effettivo dell'Attributo in caso venga scelto per fare lo split.

Per ottenere le varie occorrenze dei middlePoint (o delle istanze per gli attributi categorici) si utilizza lo stesso metodo FindAll(Predicato) citato prima per lo split sugli esempi e si restituisce la dimensione della lista che ha soddisfatto il predicato, quindi FindAll(Predicato).Count.

Un esempio*

```
occurrenceVk = examples.FindAll(x => x.Contains(vk,Index) && x.ContainsTarget(target)).Count;  
//occurrenceVk = examples.AsParallel().Count(x => x.Contains(vk,index) &&  
x.ContainsTarget(target));
```

La riga di codice a commento e' una alternativa valida a fare FindAll().Count.

AsParallel() come si intuisce dal nome lavora in parallelo. Essendo i dataSet utilizzati non giganteschi FindAll() va piu' che bene. Anzi paradossalmente mi ci impiega qualche secondo in meno dovuti al fatto che AsParallel() debba fare un bel po' di chiamate di setUp prima di partire, e come detto prima non essendo i dataSet giganteschi il gioco non vale la candela. Ma già' su DataSet di 10'000 esempi AsParallel() dovrebbe dare improvment notevoli in termini di efficienza.

DataSet, TrainingSet e TestingSet, Imputation

Nel paper Breiman suggerisce di prendere in maniera del tutto casuale il 90% del DataSet da utilizzare come TrainingSet ed utilizzarne il rimanente come TestingSet.

Questo compito e' affidato alla classe DataSet. Che setta i suoi campi trainingSet e testingSet attraverso i metodi

```
public void TrainingSet()
{
    List<int> indexes = new List<int>();
    System.Random rand = new System.Random(SettingGui.seed);
    trainingSet = new List<Example>();
    int myBoyTrainer = (int) Math.Round((double) 9/10*examples.Count);
    for (int i = 0; i < myBoyTrainer; i++)
    {
        int index = rand.Next(examples.Count);
        if (!indexes.Contains(index)) {
            indexes.Add(index);
            Example example = Examples[index];
            trainingSet.Add(Examples[index]);
        } else
        {
            i--;
        }
    }
}
public void TestingSet()
{
    testingSet = new List<Example>();
    List<int> indexesT = indexes(trainingSet);
    for(int i = 0; i < examples.Count; i++)
    {
        if (!indexesT.Contains(i))
        {
            testingSet.Add(examples[i]);
        }
    }
}
```

Per ripetere i risultati viene fissato il seed. Il seed viene passato dall'utente attraverso l'interfaccia grafica. Di default il seed e' "23".

IMPUTAZIONE

```
public void Imputation(string missSymbol, int attribute)
{
    Attribute attr = Attributes[attribute];
    string replacementSample = attr.MyType==typeOfAttribute.Categorical ?
        replacementSample=ValueMaxFrequency(attr, missSymbol):
        replacementSample=Mean(attr).ToString();
    examples.ForEach(example => { if (example.Contains(missSymbol,attribute))
        example.Replace(replacementSample, attribute); });
}
```

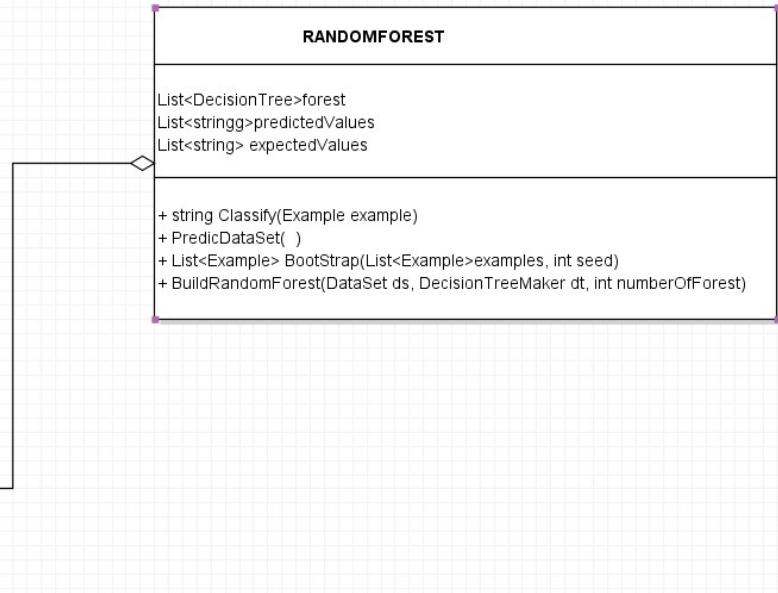
Un altro compito del DataSet e' quello di fare l'imputazione in caso di missing Values all'interno dei suoi esempi. Di solito i missingValues vengono indicati con il simbolo "?".

Se l'attributo che presenta missingValues e' di tipo categorico sostituiamo il missingvalue con la moda delle istanze dell'attributo, altrimenti si fa la media.

RANDOMFOREST

//Costruttore

```
public RANDOMFOREST(DataSet ds, DecisionTreeMaker dt, int numberOfTrees)...
```



Iniziamo a far vedere il metodo BuildRandomForest()

```
public void BuildRandomForest(DataSet ds, DecisionTreeMaker dt, int numberOfForest)
{
    TargetList = dt.TargetList(trainingSet);
    expectedValues = dt.ExpectedValues(testingSet);
    List<DecisionTreeMaker> builder = new List<DecisionTreeMaker>();
    for (int i = 0; i < numberOfForest; i++)
    {
        DecisionTreeMaker dtt = new DecisionTreeMaker(ds, dt.IndexTargetClass,
            ... dt.MyMeasure);
        builder.Add(dtt);
    }
    builder.AsParallel().ForAll(x=> {
        List<Attribute> attributes = new List<Attribute>(ds.Attributes);
        List<Example> examples = BootStrap(trainingSet,x.Id);
        forest.Add(x.BuildTree(examples, attributes, null)); } );
}
```

Un vantaggio di RandomForest sugli Alberi di Decisione e' che va a cercare il miglior attributo sul quale fare lo split su un sottoinsieme degli attributi e quindi ci mette meno a costruire il singolo alberello. Quindi per mantenere questo vantaggio si deve parallelizzare l'operazione di costruzione della foresta.

A seconda del numero di alberi della foresta creo lo stesso ammontare di DecisionTreeMaker aggiungendolo alla list<DecisionTreeMaker>builder ed in parallelo, ciascun DecisionTreeMaker appartenente a "builder", su un diverso BootStrap del trainingSet, crea un alberello da aggiungere alla foresta.

BootStrap

```
public List<Example> BootStrap(List<Example>examples, int seed)
{
    System.Random rand = new System.Random(seed);
    List<Example> result = new List<Example>();
    for(int i = 0; i < examples.Count; i++)
    {
        int randomIndex = rand.Next(examples.Count);
        Example example = examples[randomIndex];
        result.Add(example);
    }
    //printIndexes(result);
    return result;
}
```

Per poter ripetere i risultati ogni bootStrap prende come seed l'id del DecisionTreeMaker, garantendo così di poter ripetere risultati, ma anche di creare sequenze che andranno a finire nel Bootstrap diverse tra loro.

Classify

```
public string Classify(Example example)
{
    List<string> predictionSet = new List<string>();
    foreach(DecisionTree tree in forest)
    {
        string predictedValue = tree.Classify(example);
        predictionSet.Add(predictedValue);
    }
    return MajorityVote(predictionSet);
}

public string MajorityVote(List<string>votes)
{
    int max = 0;
    string result = TargetList[0];
    foreach(string target in TargetList)
    {
        int occurence = votes.FindAll(x => x.Contains(target)).Count;
        if (occurence > max)
        {
            max = occurence;
            result = target;
        }
    }
    return result;
}
```

In sintesi. I campi PredictedValues ed ExpectedValues della classe RANDOMFOREST una volta settati vengono passati come parametri ad una classe chiamata ConfusionMatrix che ha il compito di calcolare Accuratezza e Percentuale di errori che compie RANDOMFOREST a classificare il TestingSet.

```
public ConfusionMatrix(List<string> predicted, List<string> expected, List<string>
targetList)
{
    Matrix = BuildConfusionMatrix(predicted, expected, targetList);
}
```

```
public double Accuracy()
{
    return DiagonalSum()/Total();
}
```

```
public double ErrorRate()
{
    return 1 - Accuracy();
}
```

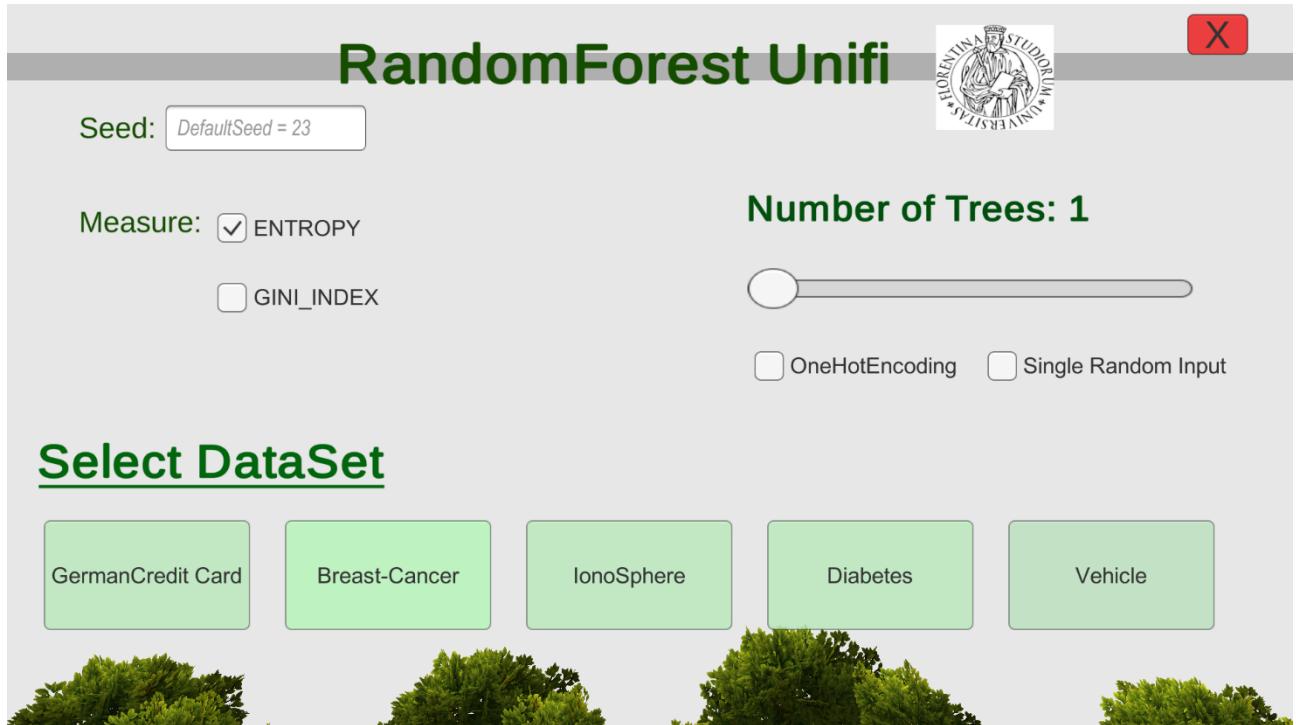
A questo punto per facilitare le cose tutto il codice visto fino ad ora precedente viene wrappato dentro una classe che si chiama PerformanceClassification.

```
public PerformanceClassification(string document, char separator, int attributes,
                                int targetIndex, Measure measure)
{
    Ds = new DataSet(document, separator, attributes, targetIndex);
    Ds.Attributes.Remove(Ds.Attributes[targetIndex]);
    dm = new DecisionTreeMaker(Ds, Ds.Target, measure);
}
```

```
public void PerformanceRandomForest(int forest)
{
    RANDOMFOREST rm = new RANDOMFOREST(Ds, dm, forest);
    c = new ConfusionMatrix(rm.PredictedValues, rm.ExpectedValues, rm.TargetList);
}
```

“forest” indica il numero degli alberi che vogliamo costruire*

GUI



Per facilitare la visione dei risultati, ho implementato una piccola Gui.

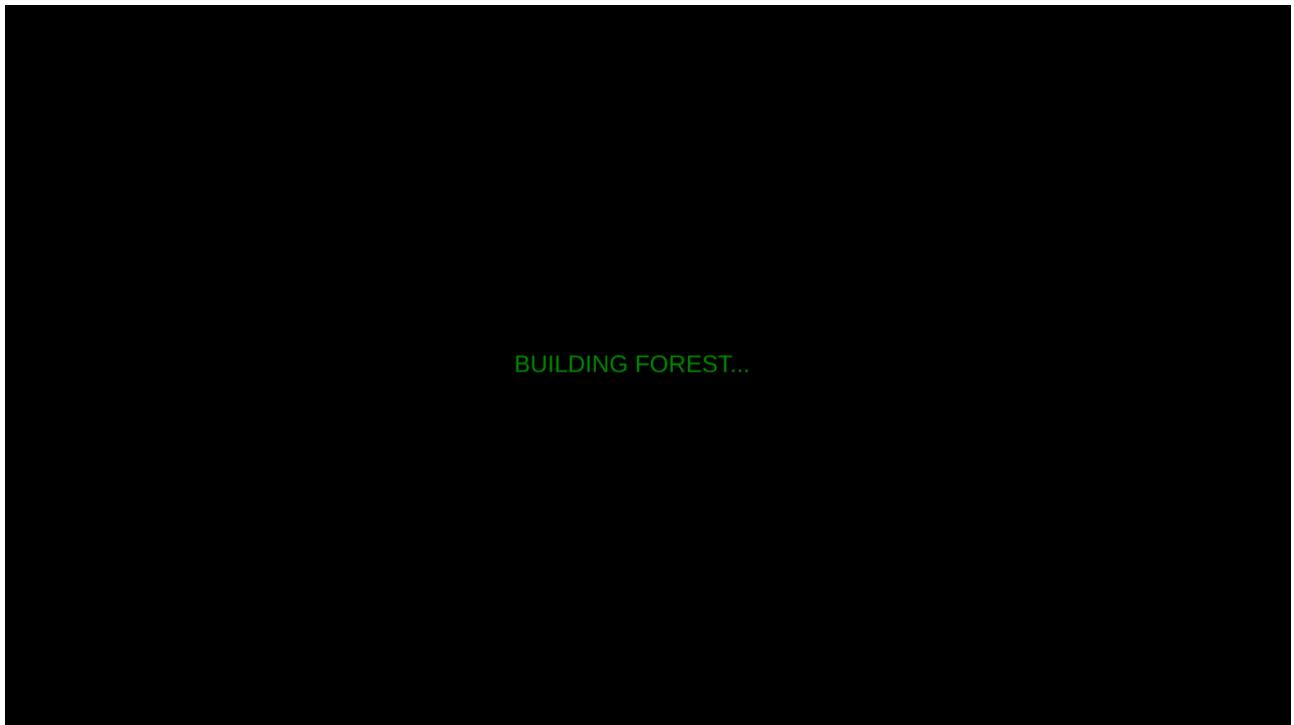
Come settaggio si puo' scegliere se usare come misura l'entropia oppure il gini_index, il numero di alberi, se fare OneHotEncoding e SingleInputRandom(ad ogni split scegliere un attributo randomico su cui fare lo split) e cosa piu' importante impostare un Seed che determinera' il modo in cui traingSet e testingSet vengono scelti (viene sempre scelto randomicamente il 90% del DataSet iniziale). A parte la scelta di questo seed ricordo che nella costruzione di RandomForest, ogni scelta randomica (dalla costruzione del Bootstrap alla scelta radomica del sottoinsieme di attributi scelti per fare lo split) e' lockata ed utilizza come seed l'id del DecisionTreeMaker che diciamo ha "fabbricato" il singolo DecisionTree*

Tutti i settaggi corrispondono a variabili statiche. Questo perche' era il modo piu' semplice senza dover entrare nei particolari di Unity che comportano serializzazione ecc, di passare valori attraverso scene diverse (ad ogni cambio scena tutti gli oggetti in scena vengono distrutti, e quindi bisogna preservare il settaggio).

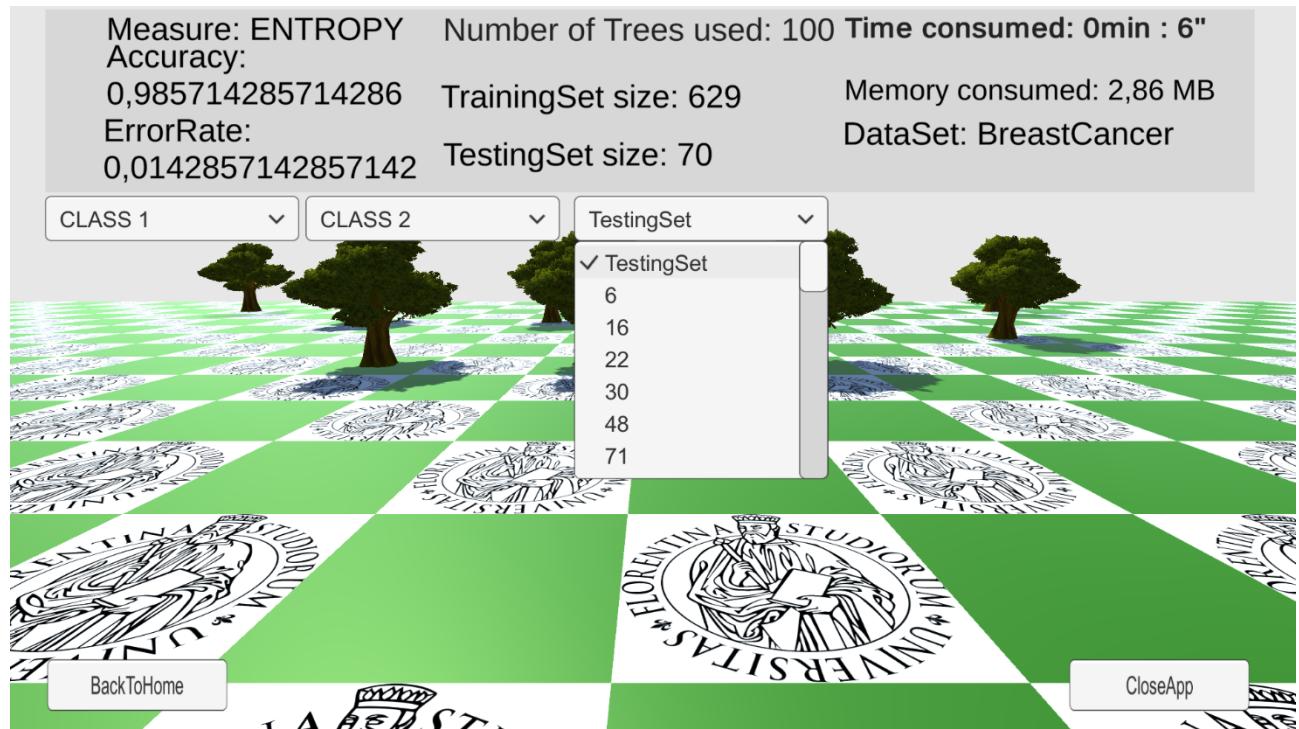
```
public static int dataSet;  
public static bool single_RandomInput=false;  
public static bool oneHot = false;  
  
public static int measure=1;  
public static int forestDimension;  
  
public static int seed=23;  
  
public void SetDataSetGerman()  
{  
    dataSet = 2;  
    StartCoroutine(LoadingScene());  
}
```

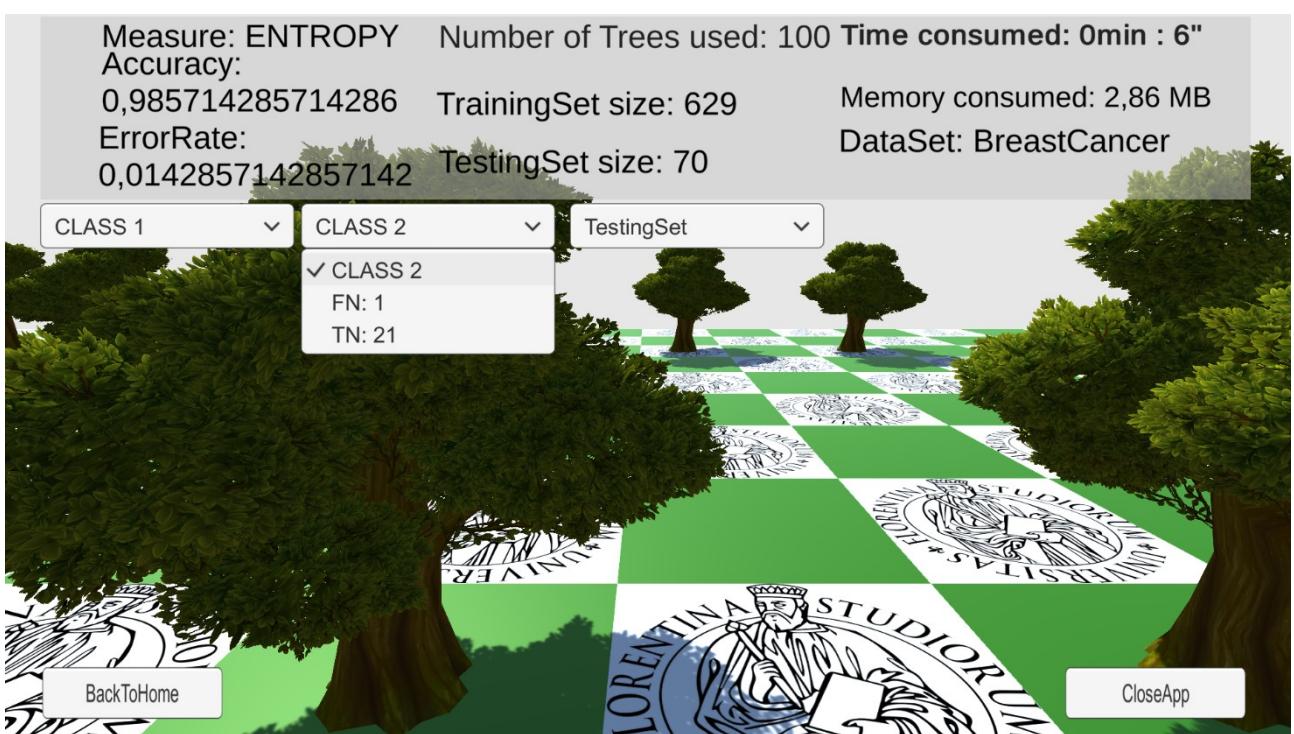
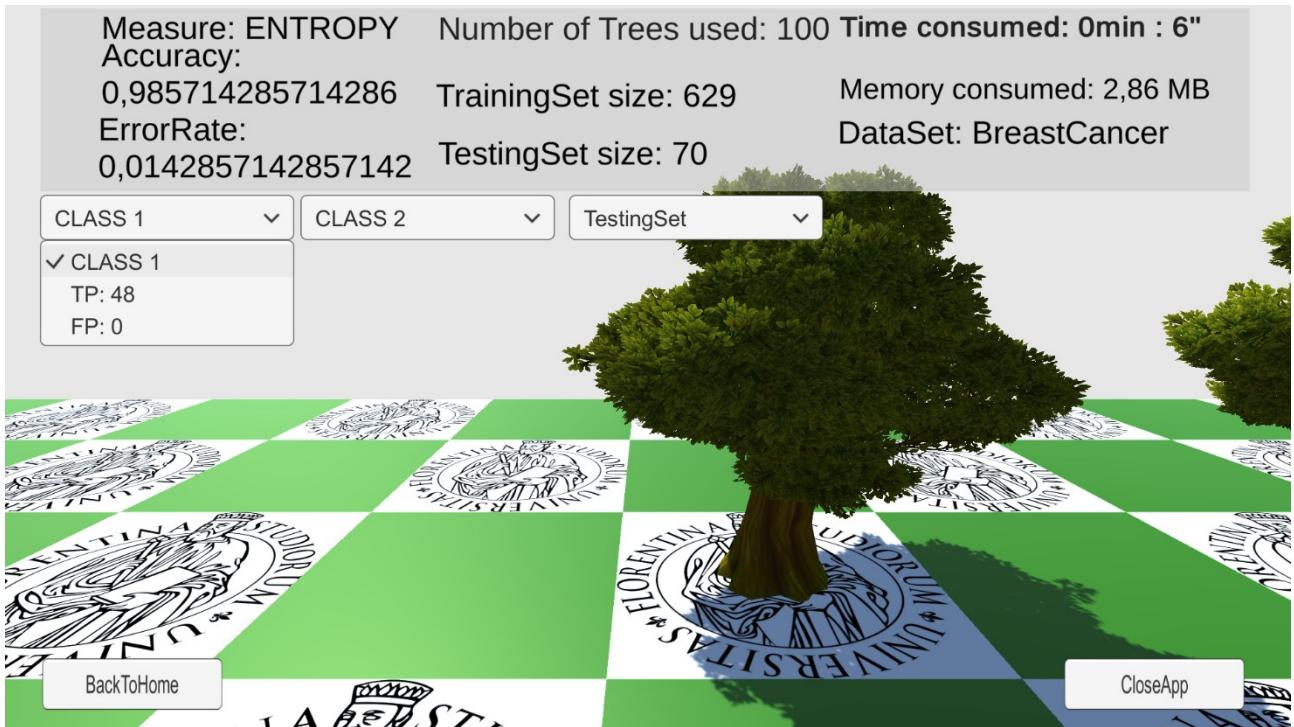
Un estratto del codice che si trova nella classe SettingGui*

Una volta settato il tutto, basta clikkare sul nome del DataSet che vogliamo testare e partira' una schermata di caricamento.



Se tutto e' andato a buonfine ci ritroveremo in questa schermata.





In breve, se si clikka su Class1 possiamo vedere quanti sono i Falsi Positivi, Veri Positivi.

Se si clikka su classe 2 quanti sono i Falsi Negativi e Veri Negativi e su TestingSet invece i record del dataSet che sono finiti nel TesitingSet (ovviamente questi cambieranno a seconda del seed scelto).

Se si clikka sul bottone BackToHome torniamo alla scena di Settaggio, mentre se si clikka su CloseApp chiudiamo l'applicazione.

In breve il codice che si puo' trovare nella classe Test*

```
void Awake()
{
    DateTime _start=DateTime.Now;
    int forest = setNumberOfTrees();
    int dataSet = SettingGui.dataSet;
    Measure measure = setMeasure();
    switch (dataSet)
    {
        case 1:
            performanceIONO(forest,measure);
            break;
        case 2:
            performanceGerman(forest,measure);
            break;
        case 3:
            performanceBreastCancer(forest,measure);
            break;
        case 4:
            performanceVehicle(forest,measure);
            break;
        case 5:
            performanceDiabetes(forest,measure);
            break;
    }
    TimeSpan elapsed = DateTime.Now - _start;
    time.text = "Time consumed: " + elapsed.Minutes +"min : "+elapsed.Seconds+ "sec";
    confusion = p.C;
    setAccuracyErrorRate();
    ConfusionMatrixDropDown();
    SetTestingExamples();
    string memoria = GetAllocatedMemory();
    memoryConsumed.text ="Memory consumed: "+ memoria;
}
```

```
public void LoadHome()
{
    SettingGui.single_RandomInput = false;
    SettingGui.oneHot = false;
    SettingGui.measure = 1;
    SettingGui.seed = 23;
    DecisionTree.unseenValues = 0;
    Example.count = -1;
    DecisionTreeMaker.count = -1;
    SceneManager.LoadScene(0);
}
```

LoadHome() si occupa di ricaricare la scena iniziale e risettare i vari settaggi come default.

RISULTATI

Table 1. Data set summary.

Data set	Train size	Test size	Inputs	Classes
Glass	214	—	9	6
Breast cancer	699	—	9	2
Diabetes	768	—	8	2
Sonar	208	—	60	2
Vowel	990	—	10	11
Ionosphere	351	—	34	2
Vehicle	846	—	18	4
Soybean	685	—	35	19
German credit	1000	—	24	2
Image	2310	—	19	7
Ecoli	336	—	7	8
Votes	435	—	16	2
Liver	345	—	6	2
Letters	15000	5000	16	26
Sat-images	4435	2000	36	6
Zip-code	7291	2007	256	10
Waveform	300	3000	21	3
Twonorm	300	3000	20	2
Threenorm	300	3000	20	2
Ringnorm	300	3000	20	2

I dataSet che mi sono scelto sono quelli evidenziati.

Ad eccezione di Vehicle, tutti i DataSet sono booleani (non so se si possa dire cosi', cioe' hanno solo due tipi di classi).

La scelta e' stata la seguente: Breast cancer presentava dei missingValues e quindi si prestava bene per implementare l'imputazione vista in classe.

German Credit Card presentava molte variabili categoriche e quindi si prestava bene per implementare opzionalmente One Hot Encoding (ero curioso).

Vehicle perche' ha classi multivariate. IonoSphere e Diabetes solo perche' avevano due tipi di classi.

Table 2. Test set errors (%).

Data set	Adaboost	Selection	Forest-RI single input	One tree
Glass	22.0	20.6	21.2	36.9
Breast cancer	3.2	2.9	2.7	6.3
Diabetes	26.6	24.2	24.3	33.1
Sonar	15.6	15.9	18.0	31.7
Vowel	4.1	3.4	3.3	30.4
Ionosphere	6.4	7.1	7.5	12.7
Vehicle	23.2	25.8	26.4	33.1
German credit	23.5	24.4	26.2	33.3
Image	1.6	2.1	2.7	6.4
Ecoli	14.8	12.8	13.0	24.5
Votes	4.8	4.1	4.6	7.4
Liver	30.7	25.1	24.7	40.6
Letters	3.4	3.5	4.7	19.8
Sat-images	8.8	8.6	10.5	17.2
Zip-code	6.2	6.3	7.8	20.6
Waveform	17.8	17.2	17.3	34.0
Twonorm	4.9	3.9	3.9	24.7
Threenorm	18.8	17.5	17.5	38.4
Ringnorm	6.9	4.9	4.9	25.7

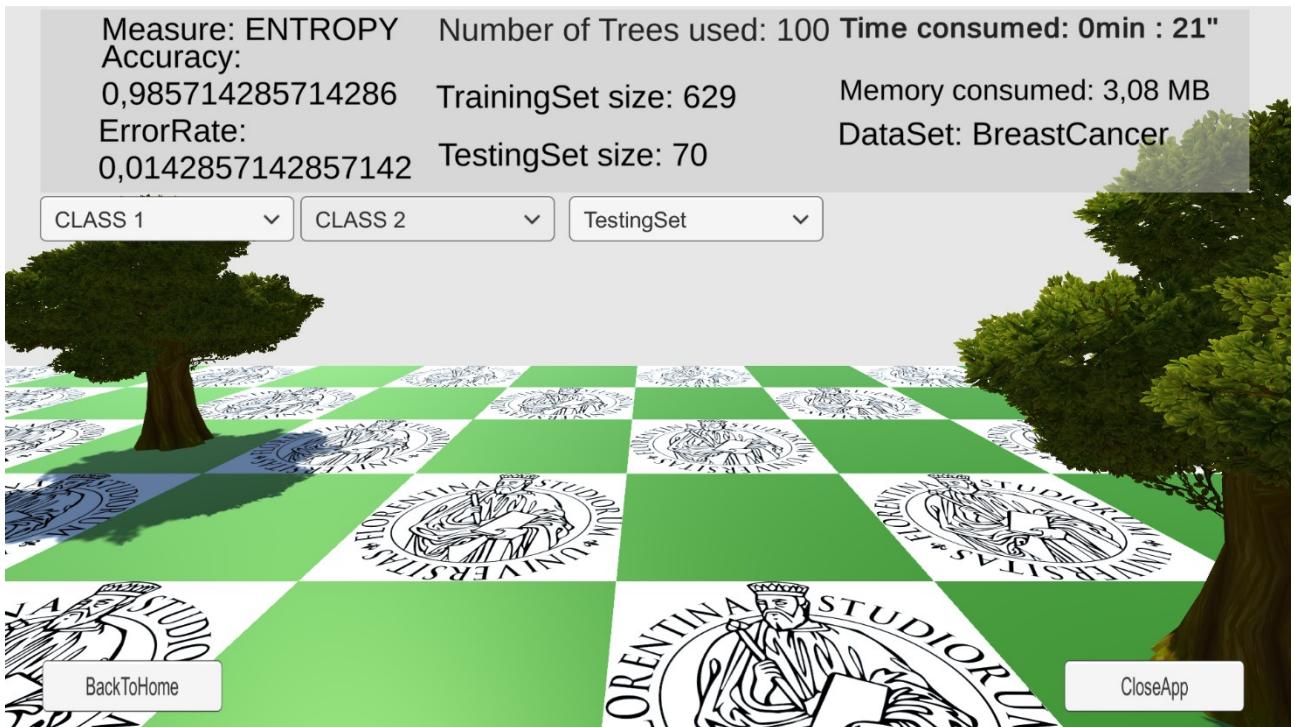
I set errors riportati in figura sono una media di 100 lanci dei set errors di RandomForest.

Io mi sono limitato a mostrare i risultati per 4 seed. Che sono "23", "42", "5" e "101" e vedere se piu' o meno i valori ricadevano in un intorno di quelli del paper.

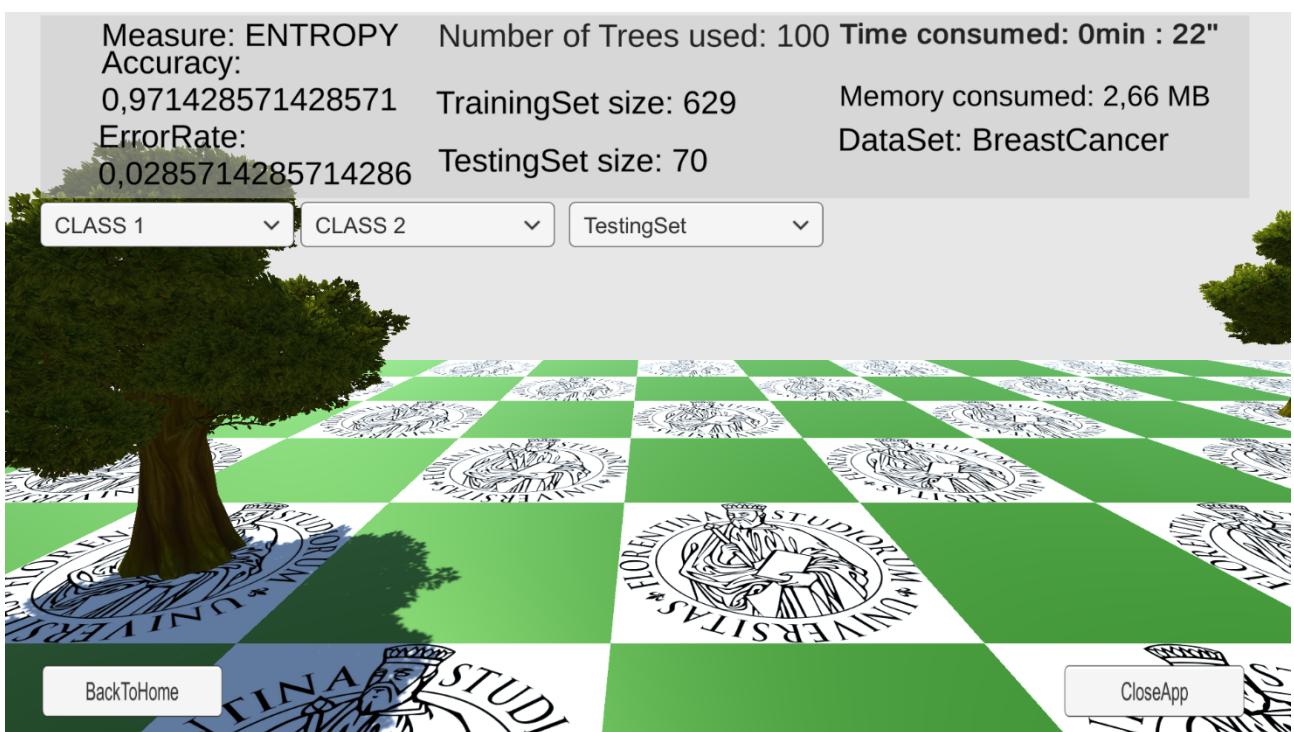
L'obiettivo e' Classification.

BreastCancer SetError Paper= 2,9%

Seed = 23



Seed = 42



Seed = 5

Measure: ENTROPY Number of Trees used: 100 Time consumed: 0min : 22"
Accuracy:
0,942857142857143 TrainingSet size: 629 Memory consumed: 2,98 MB
ErrorRate:
0,0571428571428572 TestingSet size: 70 DataSet: BreastCancer

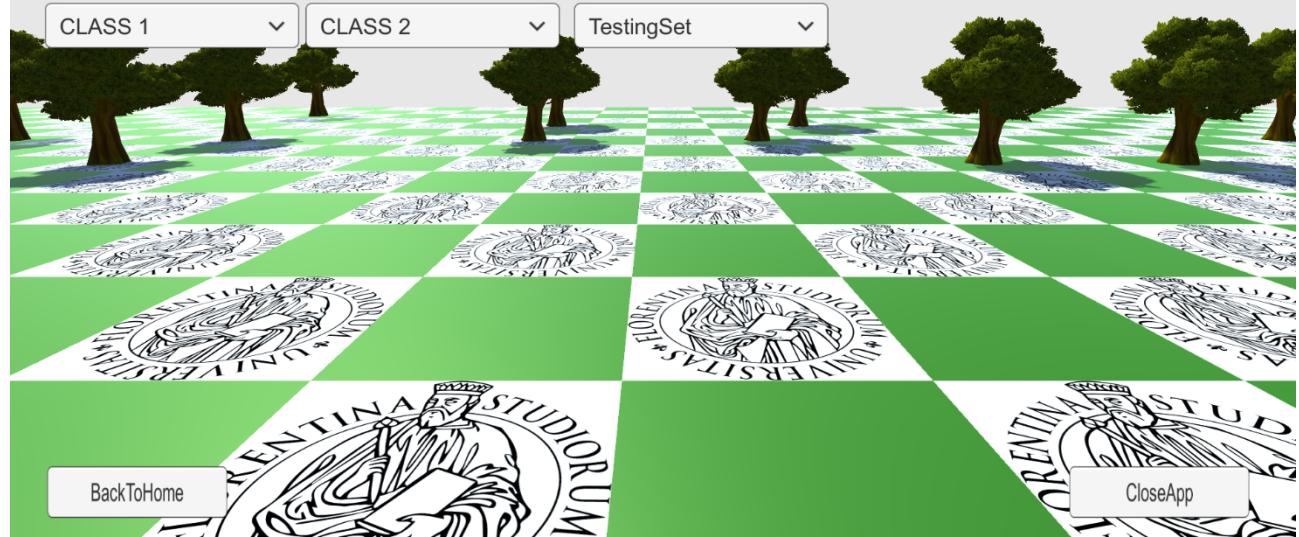
CLASS 1 ▾ CLASS 2 ▾ TestingSet ▾



Seed = 101

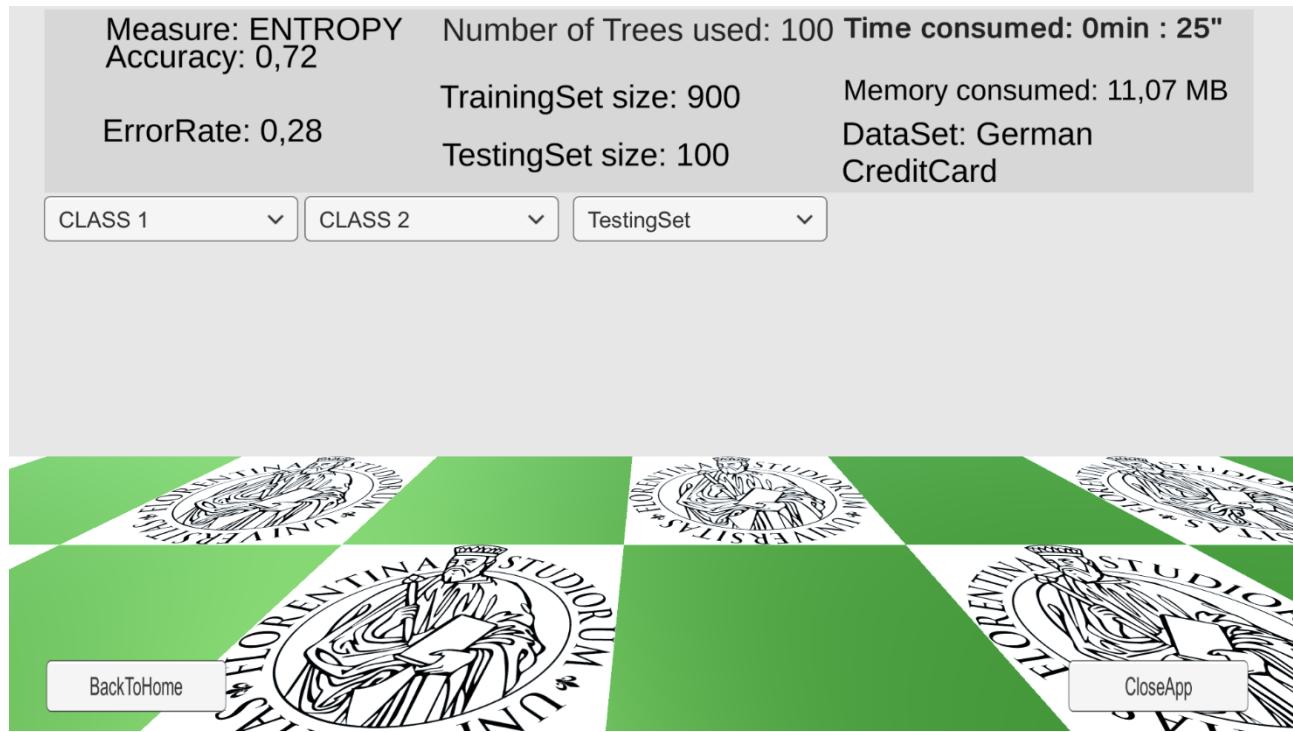
Measure: ENTROPY Number of Trees used: 100 Time consumed: 0min : 23"
Accuracy:
0,971428571428571 TrainingSet size: 629 Memory consumed: 2,79 MB
ErrorRate:
0,0285714285714286 TestingSet size: 70 DataSet: BreastCancer

CLASS 1 ▾ CLASS 2 ▾ TestingSet ▾

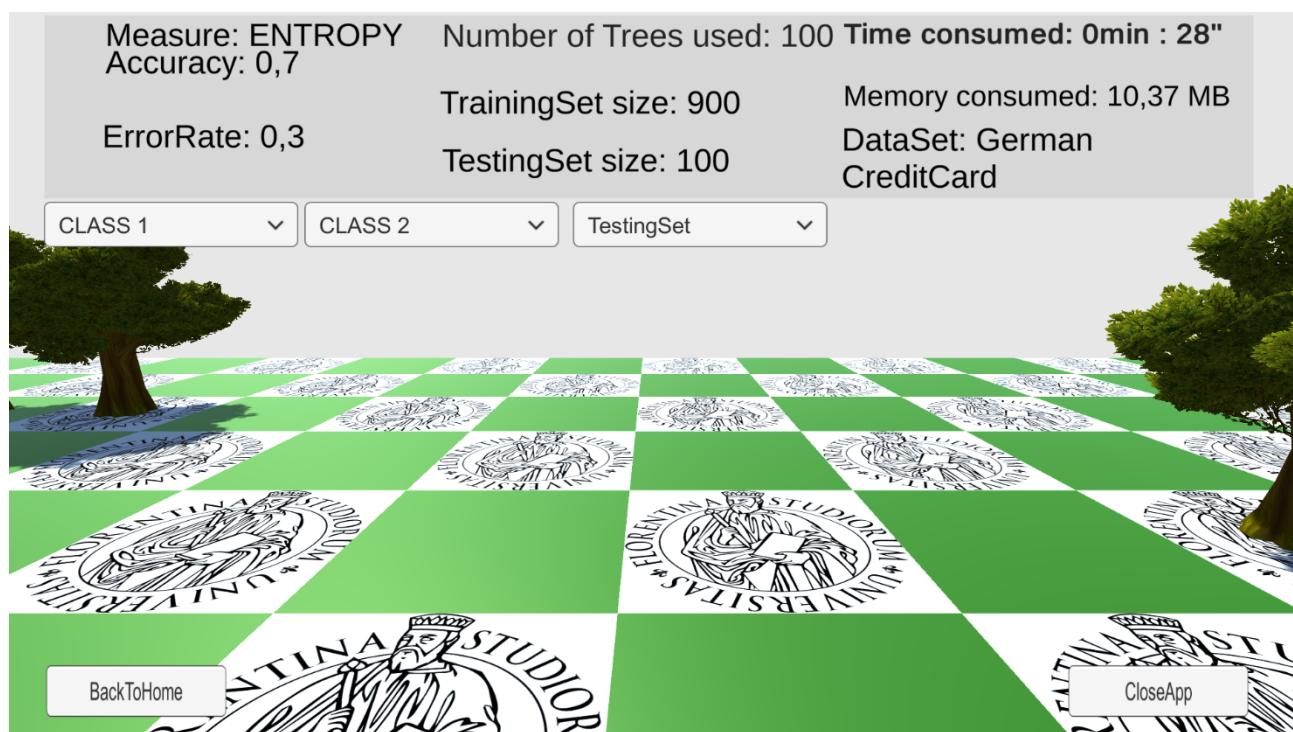


German Credit-Card Set
Error Paper= 24,4%

Seed = 23



Seed = 42



Seed = 5

Measure: ENTROPY Number of Trees used: 100 Time consumed: 0min : 25"
Accuracy: 0,79
ErrorRate: 0,21
TrainingSet size: 900
TestingSet size: 100
Memory consumed: 11,22 MB
DataSet: German CreditCard

CLASS 1 CLASS 2 TestingSet

Seed = 101

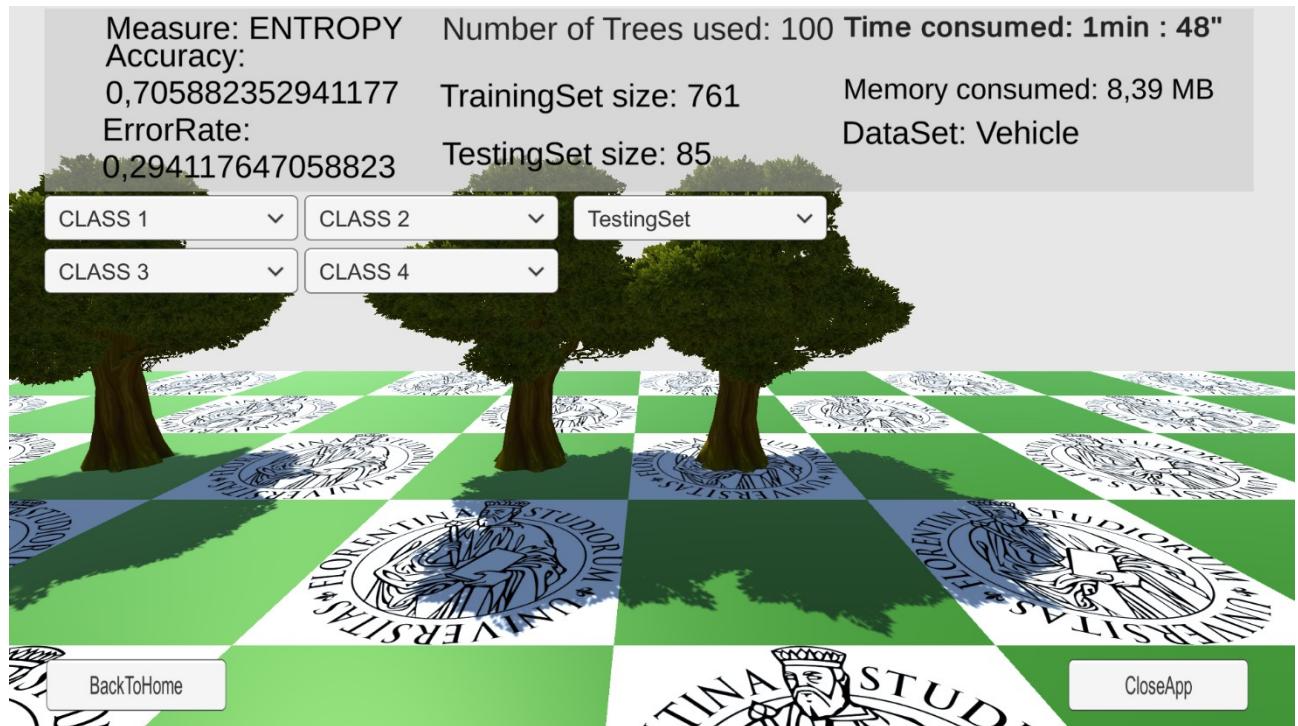
Measure: ENTROPY Number of Trees used: 100 Time consumed: 0min : 25"
Accuracy: 0,71 TrainingSet size: 900 Memory consumed: 10,26 MB
ErrorRate: 0,29 TestingSet size: 100 DataSet: German CreditCard

CLASS 1 CLASS 2 TestingSet

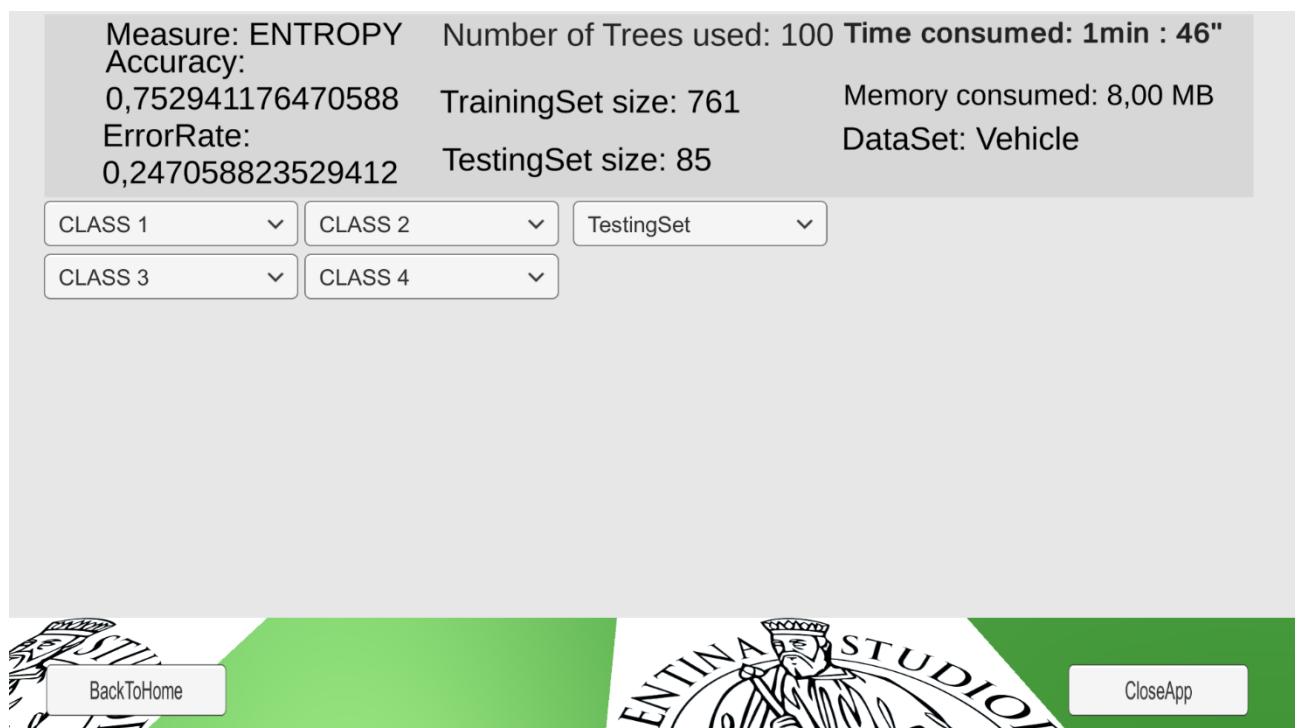
BackToHome CloseApp

Vehicle SetError Paper= 25,8%

Seed = 23



Seed = 42



Seed = 5

Measure: ENTROPY Number of Trees used: 100 Time consumed: 1min : 50"
Accuracy:
0,752941176470588 TrainingSet size: 761 Memory consumed: 6,80 MB
ErrorRate:
0,247058823529412 TestingSet size: 85 DataSet: Vehicle

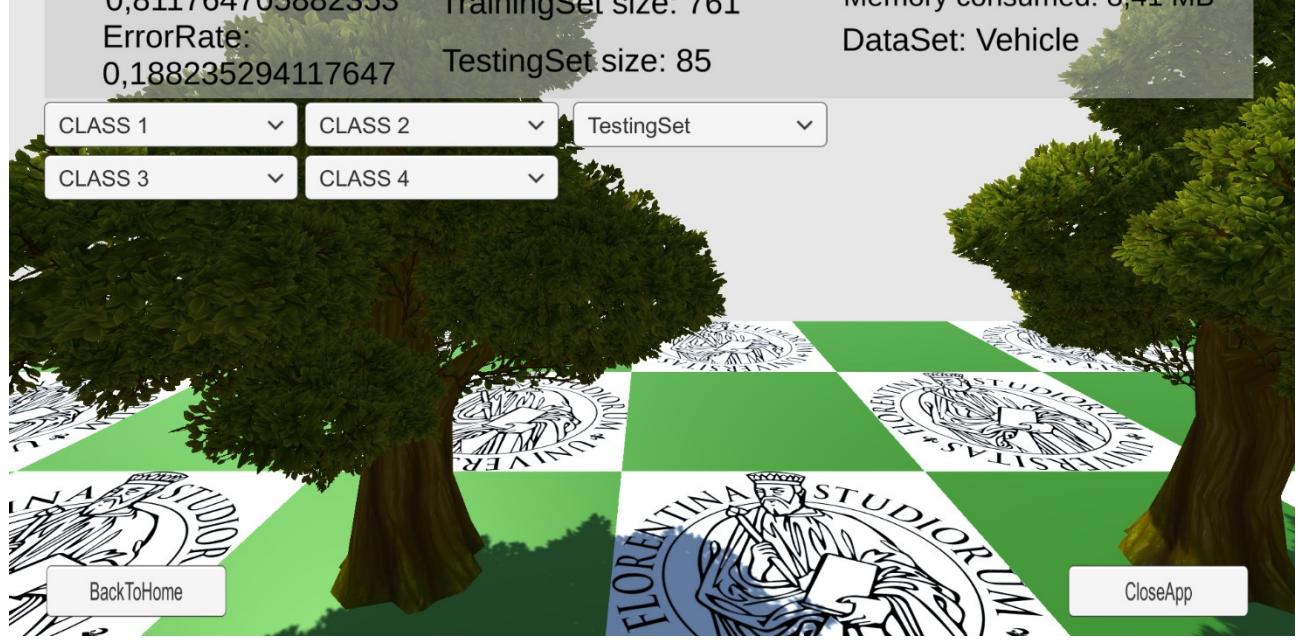
CLASS 1	▼	CLASS 2	▼	TestingSet	▼
CLASS 3	▼	CLASS 4	▼		



Seed = 101

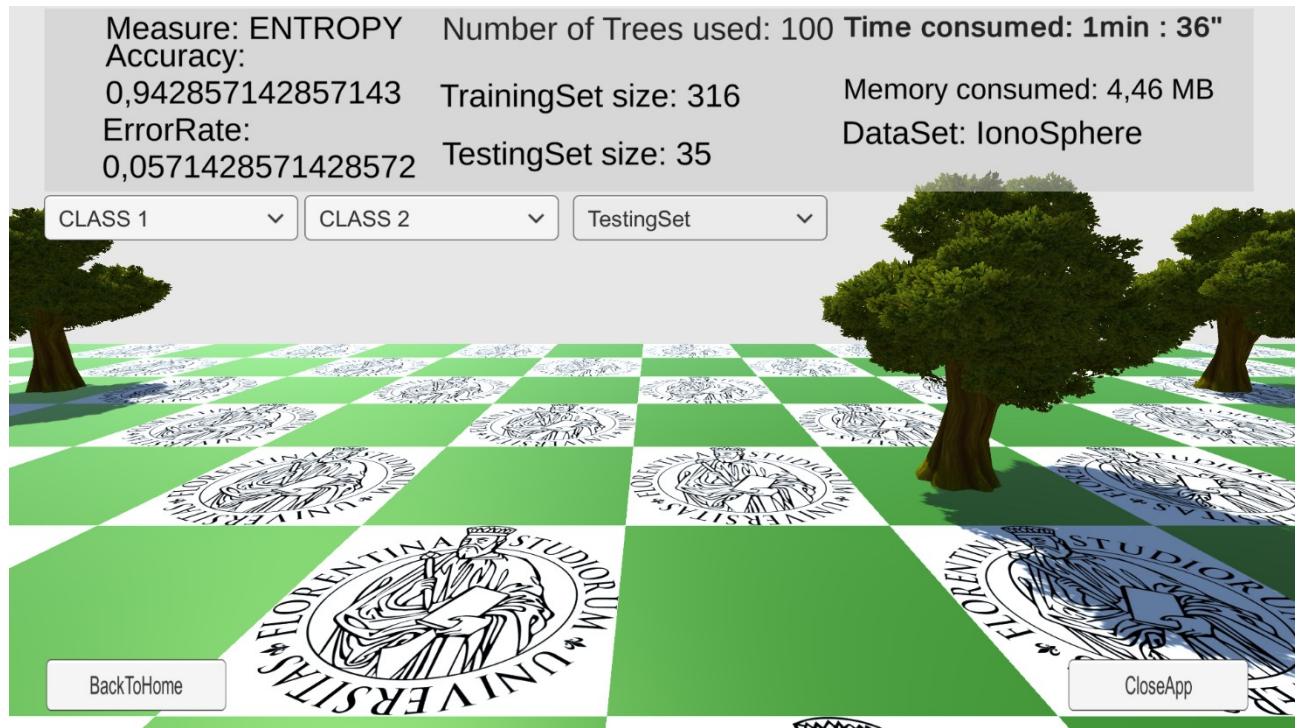
Measure: ENTROPY Number of Trees used: 100 Time consumed: 1min : 48"
Accuracy:
0,811764705882353 TrainingSet size: 761 Memory consumed: 8,41 MB
ErrorRate:
0,188235294117647 TestingSet size: 85 DataSet: Vehicle

CLASS 1	▼	CLASS 2	▼	TestingSet	▼
CLASS 3	▼	CLASS 4	▼		

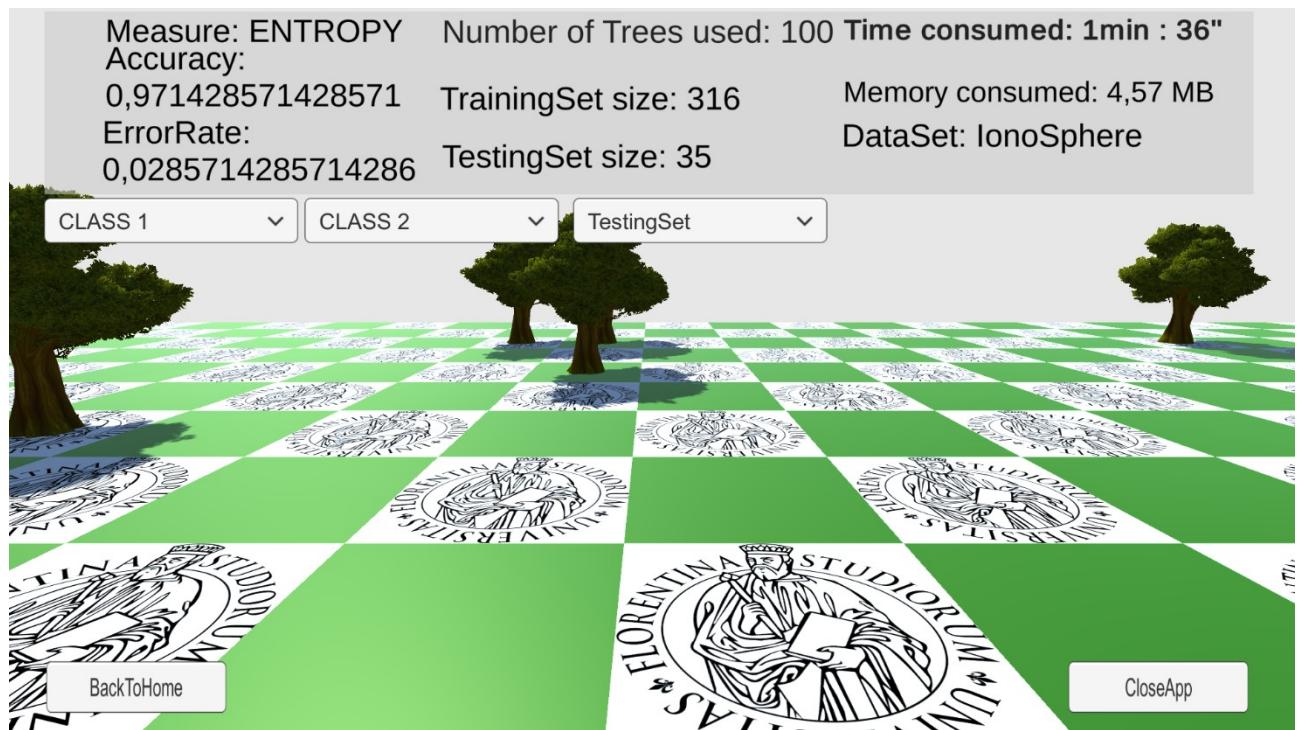


IonoSphere SetError Paper= 7,1%

Seed = 23



Seed = 42



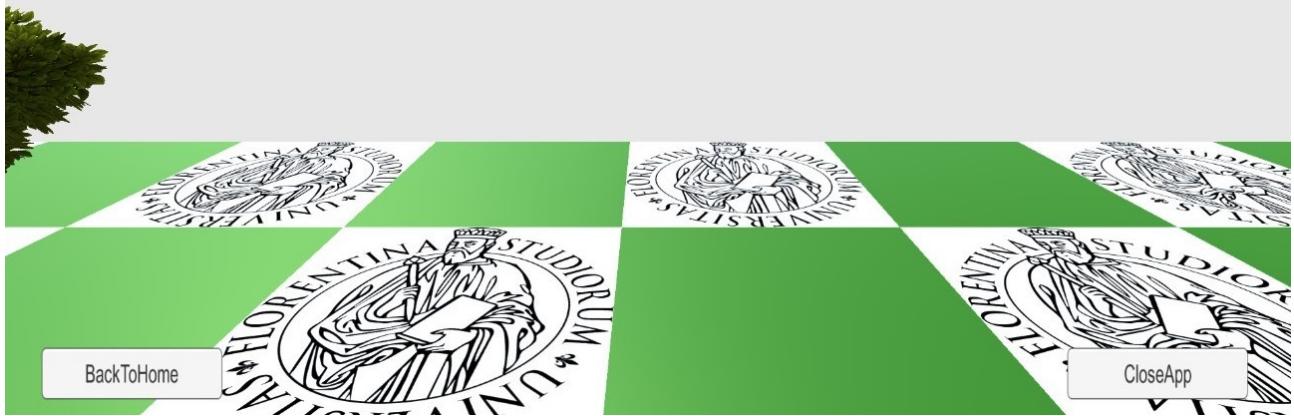
Seed = 5

Measure: ENTROPY Number of Trees used: 100 Time consumed: 1min : 28"
Accuracy:
0,828571428571429 TrainingSet size: 316 Memory consumed: 4,37 MB
ErrorRate:
0,171428571428571 TestingSet size: 35 DataSet: IonoSphere

CLASS 1

CLASS 2

TestingSet



Seed = 101

Measure: ENTROPY Number of Trees used: 100 Time consumed: 1min : 35"
Accuracy: 1 TrainingSet size: 316 Memory consumed: 4,25 MB
ErrorRate: 0 TestingSet size: 35 DataSet: IonoSphere

CLASS 1

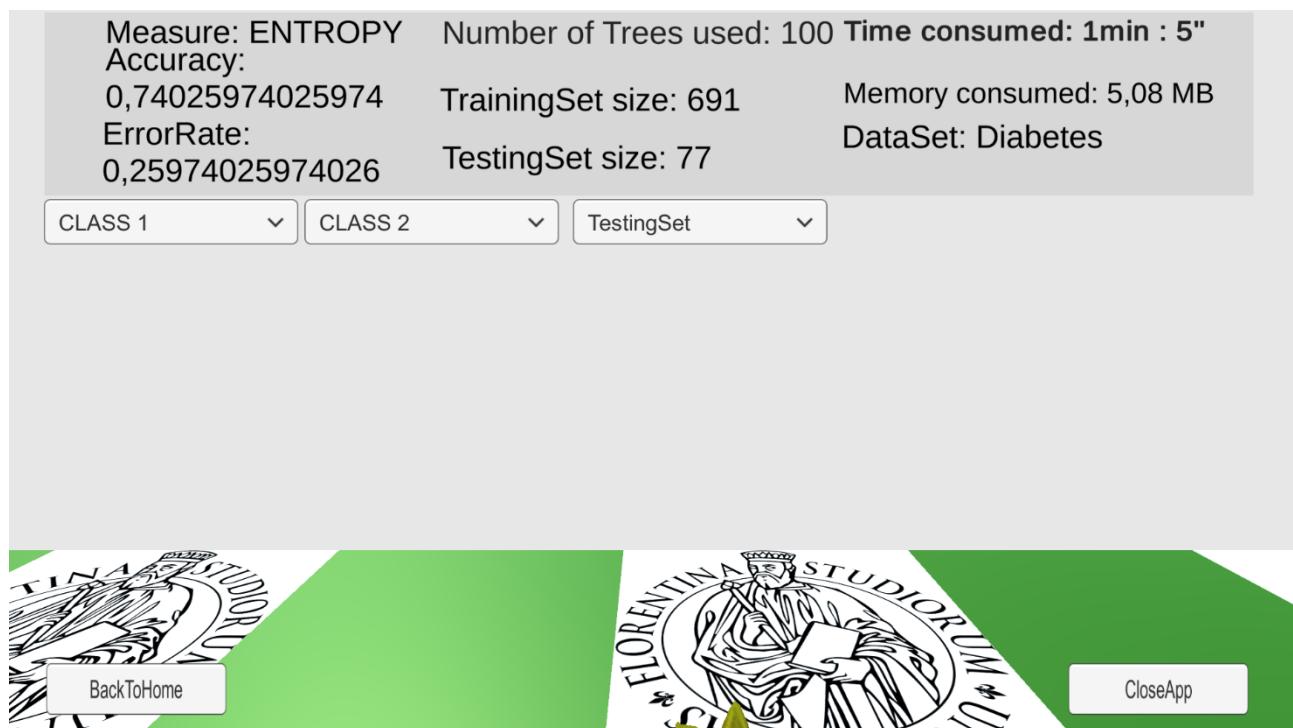
CLASS 2

TestingSet

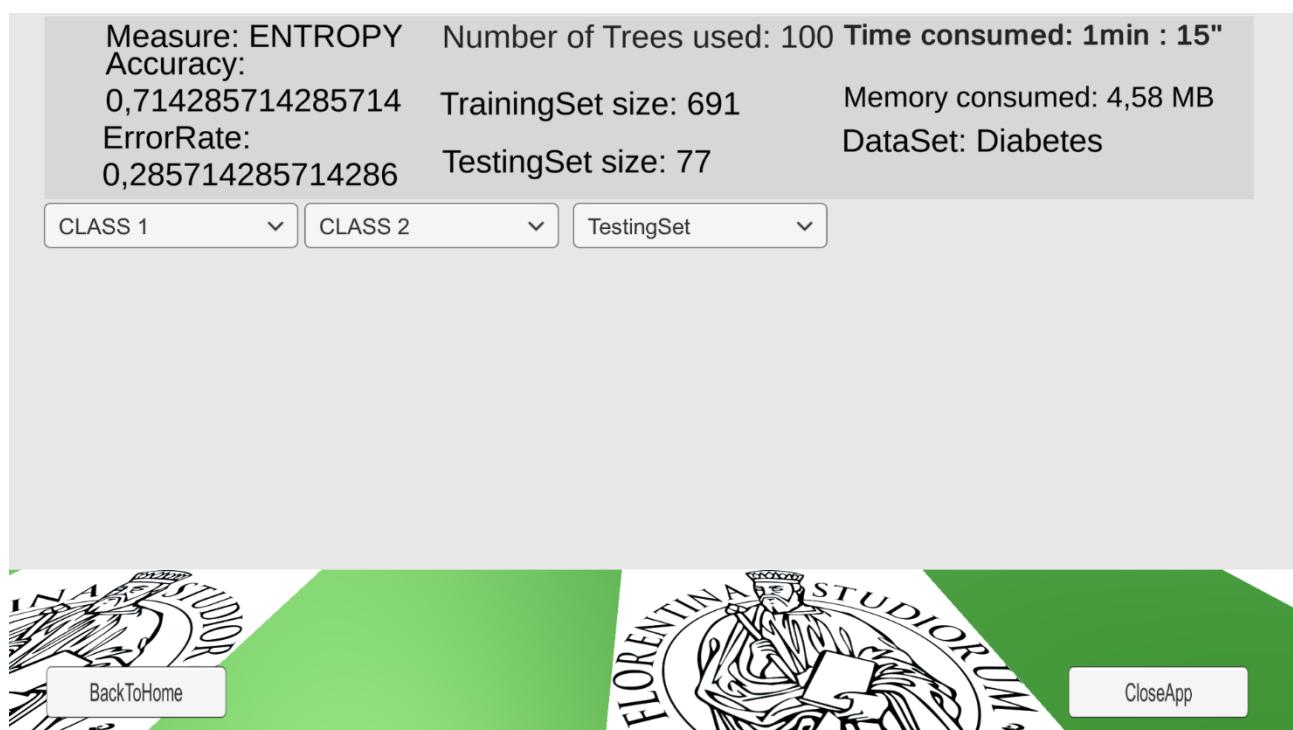


Diabetes SetError Paper= 24,2%

Seed = 23



Seed = 42



Seed = 5

Measure: ENTROPY Number of Trees used: 100 **Time consumed: 1min : 16"**
Accuracy:
0,74025974025974 TrainingSet size: 691 Memory consumed: 4,31 MB
ErrorRate:
0,25974025974026 TestingSet size: 77 DataSet: Diabetes

CLASS 1 CLASS 2 TestingSet



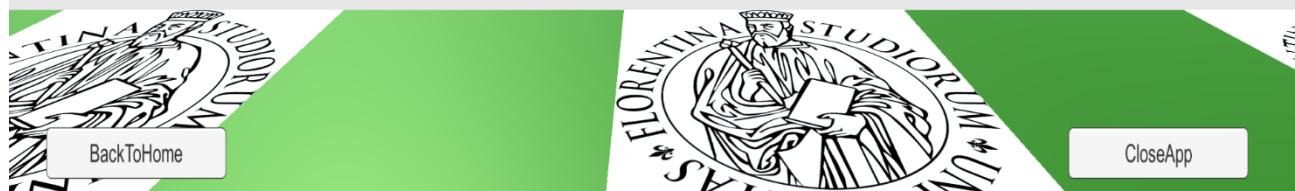
BackToHome

CloseApp

Seed = 101

Measure: ENTROPY Number of Trees used: 100 **Time consumed: 1min : 23"**
Accuracy:
0,818181818181818 TrainingSet size: 691 Memory consumed: 4,52 MB
ErrorRate:
0,181818181818182 TestingSet size: 77 DataSet: Diabetes

CLASS 1 CLASS 2 TestingSet



BackToHome

CloseApp

CONCLUSIONI

Per permettere la scelta tra Gini, Entropia e gli altri vari settaggi, essendomi concentrato nel rendere le cose user Friendly a livello di Gui e cercare di replicare i risultati del paper, nel codice non manca “Spaghetti Code”. Se tornassi indietro implementerei quasi tutto il codice abusando del pattern “Template” per permetterne l'estendibilita' del codice. Ma purtroppo e' stata un po' una corsa contro il tempo..

In conclusione se siete arrivati a leggere fin qui vi ringrazio di cuore della lettura.

-Fine-