UNIVERSITÀ DEGLI STUDI ROMA TRE
DIPARTIMENTO DI INFORMATICA E AUTOMAZIONE

Tutorial Weka

Data: *03/03/2008*

Claudio Biancalana
claudio.biancalana@dia.uniroma3.it
http://www.dia.uniroma3.it/~biancal/

Francesco Saverio Profiti
s.profiti@gmail.com

WEKA: l'approccio intelligente all'esplorazione dei dati

I parte: Introduzione al framework

In questo tutorial analizzeremo il workbench Weka: una collezione di algoritmi e tecniche di preprocessamento allo stato dell'arte del machine learning. Weka è stato disegnato per testare velocemente metodologie esistenti su insiemi di dati diversi, in modo flessibile; prevede infatti il supporto per tutto il processo sperimentativo del data mining: la preparazione dei dati di input, la valutazione statistica degli schemi di apprendimento, la visualizzazione grafica dei dati di input e del risultato dell'apprendimento.
Esamineremo gli ambienti operativi che mette a disposizione il framework e valuteremo un semplice caso di studio.

1. Che cosa è Weka?

Il **Machine Learning** (o **Apprendimento Automatico**) è il settore della Computer Science che studia gli algoritmi capaci di emulare le modalità di ragionare tipiche dell'uomo: riconoscere, decidere, scegliere, ovvero apprendere ed estrarre informazioni su un determinato problema esaminando una serie di esempi ad esso relativi.
Il crescente interesse per il data mining deriva dalla confluenza di tre grandi fenomeni tecnologici e scientifici:

- La diffusione di strumenti per la raccolta e organizzazione di grandi volumi di dati, anche via rete.
- Lo sviluppo di algoritmi più robusti ed efficienti per l'analisi dei dati.
- La disponibilità a basso costo della necessaria potenza di calcolo richiesta dai metodi di analisi dei dati.

L'esperienza mostra che non c'è un singolo schema di machine learning per tutti i problemi di data mining. La tecnica algoritmica di apprendimento universale è una fantasia idealistica: il data mining è una scienza sperimentale.
Il workbench Weka [2] è una collezione di algoritmi e tecniche di preprocessamento allo stato dell'arte del machine learning. E' stato disegnato per testare velocemente metodologie esistenti su insiemi di dati diversi, in modo flessibile; prevede infatti il supporto per tutto il processo sperimentativo del data mining: la preparazione dei dati di input, la valutazione statistica degli schemi di apprendimento, la visualizzazione grafica dei dati di input e del risultato dell'apprendimento.
Weka è stato sviluppato dall'università Waikato in Nuova Zelanda, e il nome sta per **W**aikato **E**nvironment for **K**nowledge **A**nalysis.

2. Installazione ed uso di Weka

Weka [2] è scritto in Java, quindi si può utilizzare su qualunque sistema operativo dotato di un ambiente di esecuzione Java. Se il JRE (Java Runtime Environment) è già installato, basta scaricare il solo programma di installazione Weka ed eseguirlo. Altrimenti, la cosa più conveniente è scaricare il programma di installazione Weka + JRE che installa entrambi gli ambienti in una volta sola.
Una volta lanciato Weka possiamo scegliere tra 4 diversi ambienti operativi:

- ***SimpleCLI***
  - E' un ambiente a linea di comando, da usare per invocare direttamente le varie classi Java di cui Weka è composto. Tutto quello che si può fare dalla SimpleCLI è possibile farlo anche da un ambiente a linea di comando come il "prompt di DOS" di Windows o la shell di Unix.

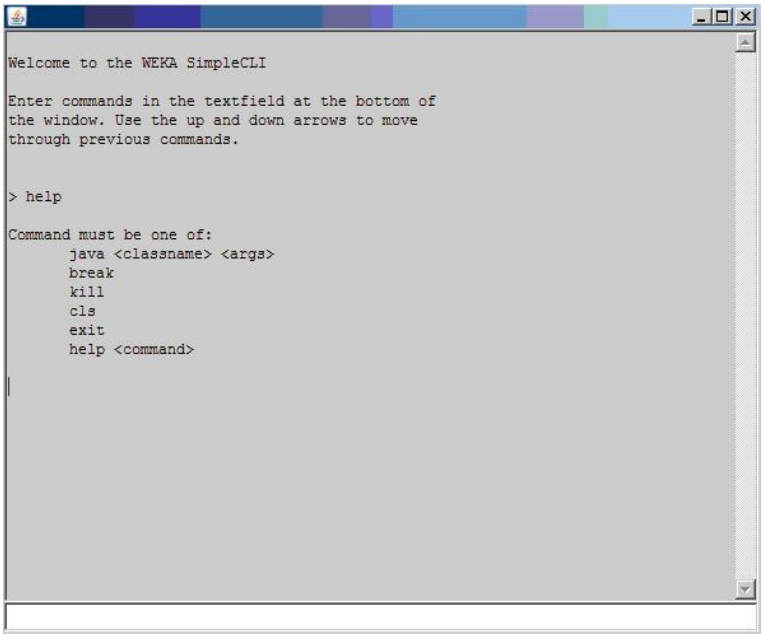

Figura 1. SimpleCLI

- ***Explorer***
  - E' l'ambiente che utilizzeremo più spesso. Con esso si possono caricare degli insiemi di dati, visualizzare in modo grafico la disposizione degli attributi, effettuare una serie di operazioni preliminari di preparazione, ed eseguire algoritmi di classificazione, clustering, selezione di attributi e determinazione di regole associative.
    Per gli attributi nominali abbiamo l'elenco dei possibili valori e, per ognuno di essi, il numero di istanze con quel valore. Interessante anche il conteggio del numero di istanze in cui l'attributo manca e del numero di valori che appaiono una sola volta.
    Per gli attributi numerici, abbiamo le informazioni sul valore massimo, minimo, media e deviazioni standard, oltre alle solite informazioni su numero di valori diverse, numero di valori unici e numero di istanze col valore mancante.

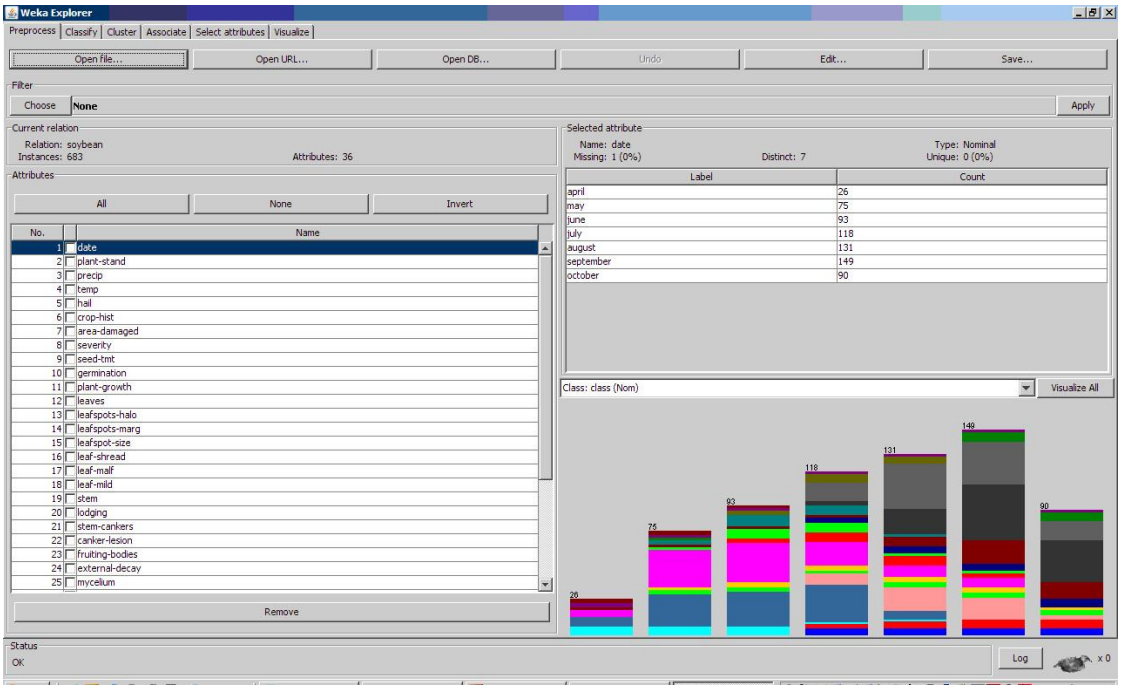

Figura 2. Explorer

- ***Experimenter***
  - E' un ambiente che consente di impostare una serie di analisi, su vari insiemi di dati e con vari algoritmi, ed eseguirle alla fine tutte insieme. E' possibile in questo modo confrontare vari tipi di algoritmi, e determinare qual è il più adatto a uno specifico insieme di dati.

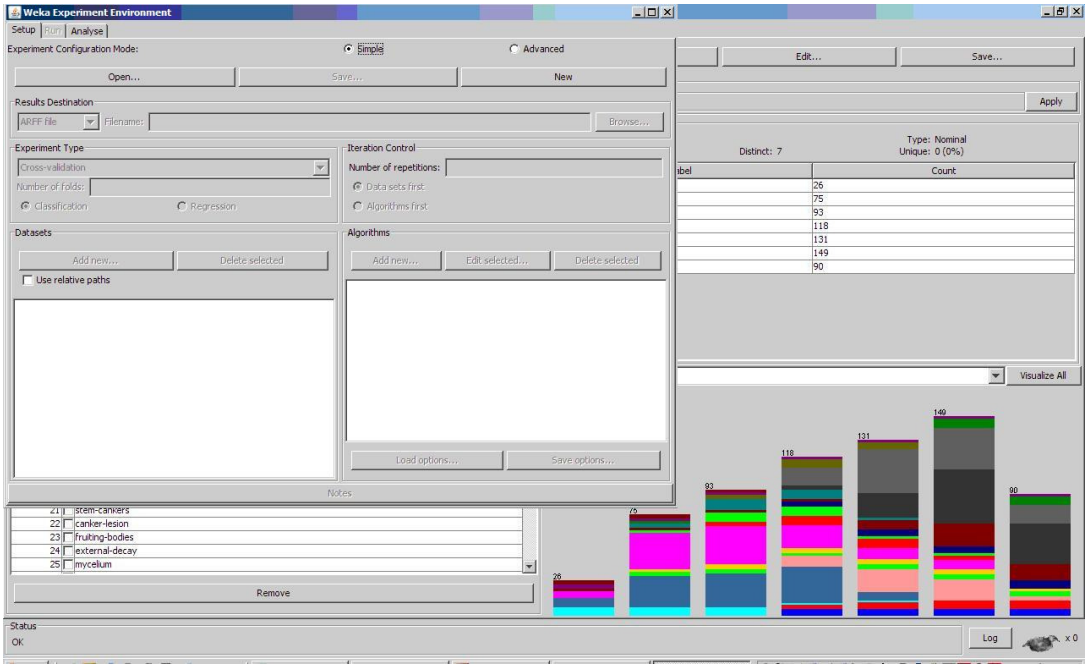

Figura 3. Experimenter

- ***KnowledgeFlow***
  - Una variante dell'explorer, in cui le operazioni da eseguire esprimono in un ambiente grafico, disegnando un diagramma che esprime il "flusso della conoscenza". E' possibile selezionare varie componenti come sorgenti di dati, filtri, algoritmi di classificazione e collegarli tra loro in un diagramma tipicamente detto "data-flow".

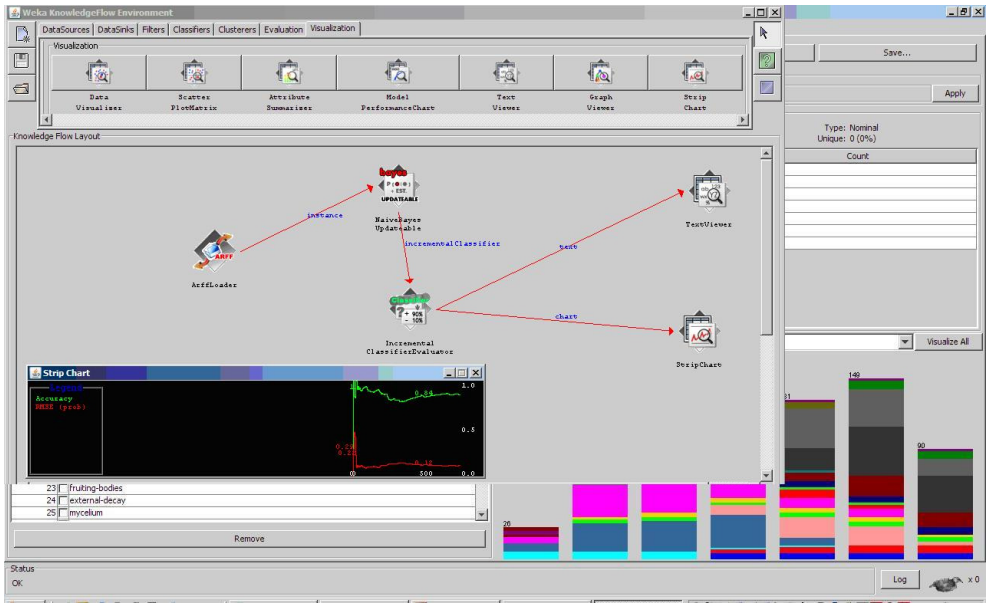

Figura 4. Knowledge Flow

3. Un semplice esempio

Weka concentra il suo interesse principalmente sugli algoritmi di classificazione e sui filtri per la pre-elaborazione dei dataset (entrambi presenti in gran numero), ma include anche l'implementazione di un algoritmo per la scoperta di regole di associazione (l'algoritmo Apriori, il più noto) e di qualche algoritmo per il clustering e per la regressione.
Tra i principali schemi (learner) figurano:

- Apriori
- C4.5 decision tree
- Cobweb and Classit clusterer
- Decision table majority classifier
- EM (estimation maximization) clusterer
- HyperPipe classifier
- K-nearest neighbour classifier
- K* classifier
- Id3 decision tree classifier
- Linear regression
- Locally-weighted regression
- Logistic regression
- Naive Bayes classifier
- Neural Network back propagation classifier
- PART decision list
- PRISM classifier
- 1R classifier
- Voting feature interval classifier
- Voted perceptron

Il package `weka.filters` offre un utile supporto alla pre-elaborazione dei dati (data preprocessing). Questo package contiene numerose classi Java in grado di operare trasformazioni sull'insieme dei dati, come ad esempio la rimozione o l'aggiunta di attributi, il rimescolamento del dataset, la rimozione di determinate tuple, ed altro ancora.
I filtri possono suddividersi in supervisionati e non supervisionati: nel primo caso fanno uso dell'informazione aggiuntiva dovuta al conoscere l'attributo target, mentre nel secondo, trattano tutti gli attributi nello stesso modo.
Un'ulteriore suddivisione è in filtri d'attributo e filtri di tupla: nel primo caso operano sugli attributi, mentre nel secondo operano sulle tuple.
WEKA inoltre contiene dei Meta-learning schemes, ossia metodi che non implementano direttamente un algoritmo per data mining, ma consentono di migliorare le performance dei learner.

Supponiamo di avere alcuni dati da analizzare e di volerci costruire un sistema di inferenza. Prima di tutto abbiamo la necessità di preparare i dati da caricare tramite l'Explorer, in seguito dobbiamo selezionare un metodo di costruzione del classificatore, ed infine interpretarne l'output.

3.1. *Preparare i dati*

Il formato di analisi (nativo) di weka è il formato ARFF (**A**ttribute-**R**elation **F**ile **F**ormat): si tratta di un formato per un file testo, utilizzato per memorizzare dati in database.
Prepariamo il file dei dati seguendo queste semplice struttura:

- **Intestazione** (definizione degli attributi)

```
@relation <nome del dataset>
@attribute <nome attr> {<val1>, <val2>, …,<valn>}
   oppure
@attribute <nome attr> real
```

  Dove l'ultimo attributo indica la classe.

- **Dati**
  - E' costituita dal tag `@data` seguito dalle descrizioni degli esempi, una su ogni riga (terminata da a capo)
  - Ogni esempio è descritto dalla lista dei valori per ciascun attributo separati da virgole
  - Ogni valore corrisponde all'attributo che si trova nella stessa posizione dell'intestazione

- Le dichiarazioni `@relation`, `@attribute` e `@data` sono case insensitive.

Segue un'istanza della struttura descritta:

```
@RELATION iris

@ATTRIBUTE sepallength  NUMERIC
@ATTRIBUTE sepalwidth   NUMERIC
@ATTRIBUTE petallength  NUMERIC
@ATTRIBUTE petalwidth   NUMERIC
@ATTRIBUTE class        {Iris-setosa,Iris-versicolor,Iris-virginica}

@DATA
5.1,3.5,1.4,0.2,Iris-setosa
4.9,3.0,1.4,0.2,Iris-setosa
4.7,3.2,1.3,0.2,Iris-setosa
4.6,3.1,1.5,0.2,Iris-versicolor
5.0,3.6,1.4,0.2,Iris-versicolor
5.4,3.9,1.7,0.4,Iris-versicolor
4.6,3.4,1.4,0.3,Iris-virginica
5.0,3.4,1.5,0.2,Iris-virginica
4.4,2.9,1.4,0.2,Iris-virginica
4.9,3.1,1.5,0.1,Iris-virginica
```

Dall'Explorer di Weka, tutte le operazioni di pre-elaborazioni si possono eseguire dalla scheda "Pre-process".

3.2. *Caricare i dati nell'Explorer*

Carichiamo i dati nell'Explorer ed analizziamoli con tecniche di machine learning: cliccando su "open file" carichiamo il file ARFF di esempio create precedentemente "iris.arff". A questo punto l'interfaccia mostra alcune informazioni fondamentali del dataset (vedi figura 5): 10 istanze e 5 attributi. Un istogramma in basso a destra mostra la frequenza dei valori della classe (attributo `class`) corrispondentemente al valore dell'attributo selezionato. Per gli attributi numerici l'explorer fornisce i valori di minimo, massimo, media e deviazione standard. È possibile ispezionare i dati caricati attraverso la funzionalità di editing (pulsante edit) al fine di cercare uno specifico valore per modificare/cancellare il suo corrispondente attributo o istanza.

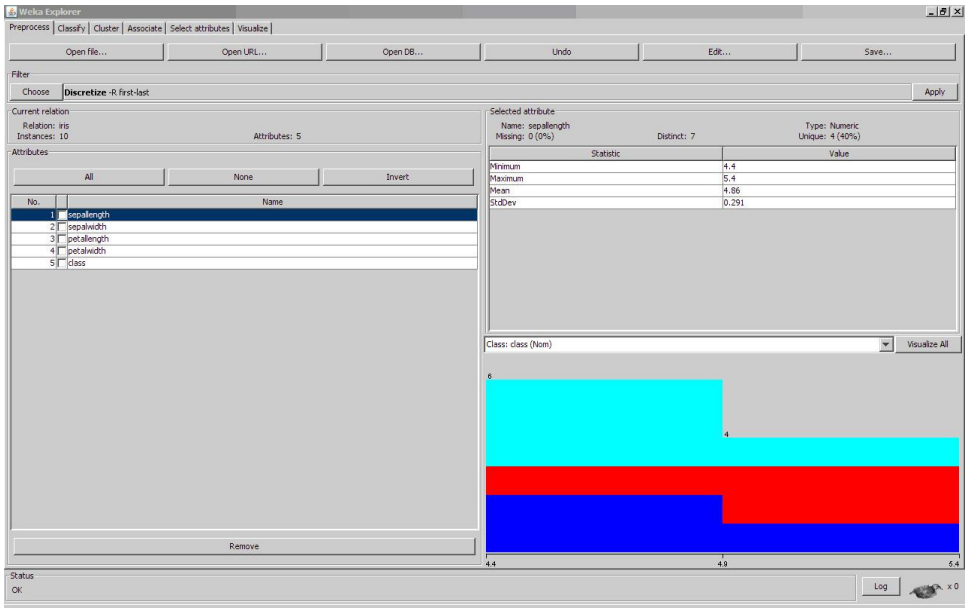

Figura 5. Explorer: dataset iris.arff

3.3. *Costruire un classificatore*

Analizziamo a titolo esemplificativo e del tutto generico un learner costituito da un albero di decisione: nello specifico Weka integra al suo interno l'algoritmo J48.

Che cos'è un albero di decisione? Tentiamo di dare una breve descrizione:
gli alberi di decisione costituiscono il modo più semplice di classificare degli "oggetti" in un numero finito di classi. Essi vengono costruiti suddividendo ripetutamente i record in sottoinsiemi omogenei rispetto alla variabile risposta. La suddivisione produce una gerarchia ad albero, dove i sottoinsiemi (di record) vengono chiamati nodi e, quelli finali, foglie. In particolare, i nodi sono etichettati con il nome degli attributi, gli archi (i rami dell'albero) sono etichettati con i possibili valori dell'attributo soprastante, mentre le foglie dell'albero sono etichettate con le differenti modalità dell'attributo classe che descrivono le classi di appartenenza. Un oggetto è classificato seguendo un percorso lungo l'albero che porti dalla radice ad una foglia. I percorsi sono rappresentati dai rami dell'albero che forniscono una serie di regole.
Anche se gli alberi di decisione forniscono una rappresentazione compatta di una procedura di classificazione, spesso risulta difficile spiegare, ad esempio a persone non esperte, una struttura del tipo di quella presentata in figura 6, quindi si preferisce utilizzare una rappresentazione equivalente, ma più intuitiva, con l'ausilio di "regole di classificazione", che possono essere ricavate facilmente dall'albero e che nel caso dell'esempio in questione potrebbero essere espresse nel modo seguente:

```
IF petallength  > 1.4 then Iris-versicolor
ELSE IF sepallength  > 4.6 then Iris-setosa
ELSE Iris-virginica
```

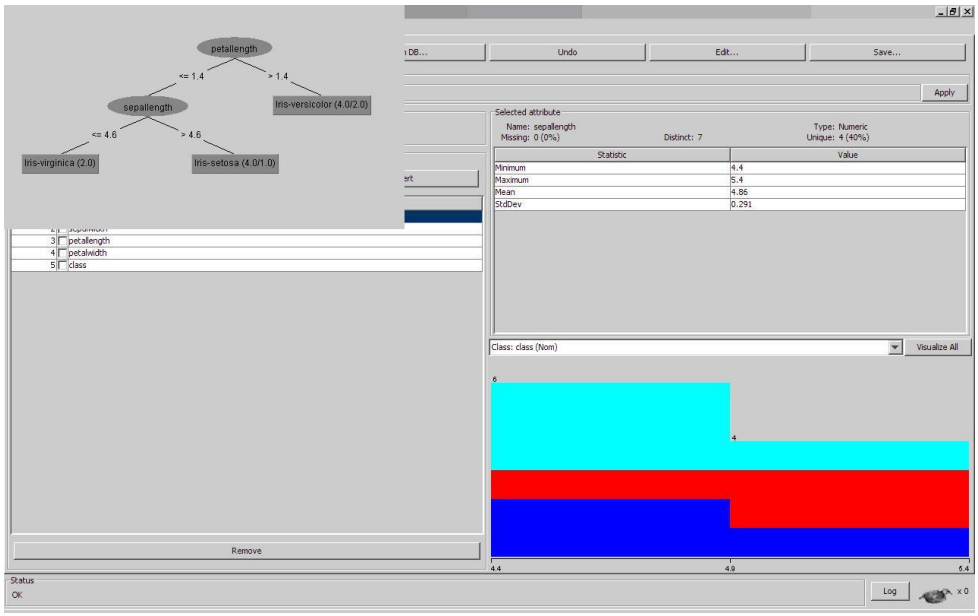

Figura 6. Albero di decisione generato dal dataset creato

Cliccando sul tab "classify" è sufficiente selezionare il classificatore attraverso il pulsante "choose" (nella sezione trees troviamo l'algoritmo J48).
Cliccando sul bottone "start", il classificatore lavorerà fintanto che il piccolo uccello in basso a destra si muove; produrrà infine l'output che esamineremo nel prossimo paragrafo.

3.4. *Esaminare l'output*

L'output generato da Weka per il dataset col classificatore J48. Nella parte iniziale troviamo un riassunto del dataset, e la metodologia con il quale il classificatore viene valutato (nel caso di default è la 10-fold-cross validation). Subito dopo troviamo l'albero di decisione generato in formato testuale.

```
=== Run information ===

Scheme:       weka.classifiers.trees.J48 -C 0.25 -M 2
```

```
Relation:     iris
Instances:    10
Attributes:   5
              sepallength
              sepalwidth
              petallength
              petalwidth
              class
Test mode:    10-fold cross-validation

=== Classifier model (full training set) ===

J48 pruned tree
------------------

petallength <= 1.4
|   sepallength <= 4.6: Iris-virginica (2.0)
|   sepallength > 4.6: Iris-setosa (4.0/1.0)
petallength > 1.4: Iris-versicolor (4.0/2.0)

Number of Leaves  :     3
Size of the tree :      5
```

L'output fornisce, come risultato finale, uno strumento noto in letteratura come "matrice di confusione" che può essere molto interessante per misurare le potenzialità del modello generato. Associare ad ogni modello una matrice di confusione permette di scegliere il modello migliore che non necessariamente coincide con quello avente il tasso di accuratezza maggiore.
Essa è una matrice di dimensione kxk, con k numero di classi nella quale sulle colonne si hanno il numero reale di records appartenenti a ciascuna classe e sulle righe il numero previsto di records appartenenti ad una data classe.

```
=== Confusion Matrix ===

 a b c   <-- classified as
 2 0 0 | a = Iris-setosa
 0 2 2 | b = Iris-versicolor
 0 0 1 | c = Iris-virginica
```

In questo modo i valori presenti sulla diagonale principale sono quelli che rappresentano il numero di casi classificati correttamente dall'algoritmo, mentre ogni valore fuori dalla diagonale principale rappresenta un errore di classificazione.

Il parte: Preprocessamento e classificazione

In questa seconda parte esamineremo il processo di estrazione della conoscenza, noto in letteratura col nome di Knowledge Discovery to Data (KDD), analizzando come questo viene implementato nel framework Weka. Daremo una breve overview delle metodologie di preprocessamento e classificazione presenti all'interno del framework fornendo un semplice esempio pratico di come queste possano essere utilizzate all'interno del nostro codice java.

4. Il processo per l'estrazione di conoscenza dai dati

I sistemi classici di memorizzazione dei dati i DBMS (DataBase Management System) offrono un'ottima possibilità di memorizzare ed accedere ai dati con sicurezza, efficienza e velocità ma non permettono un'analisi per l'estrazione di informazioni utili come supporto alle decisioni. Solo negli ultimi anni sono apparse tecnologie in grado di svolgere questo compito, grazie al contributo di diverse discipline tra cui: l'intelligenza artificiale, reti neurali, statistica, apprendimento automatico.
Queste tecnologie, applicate alle basi di dati, sono conosciute con i seguenti termini: knowledge discovery, knowledge extraction, data archaeology, Data Mining[1][4].
Questi termini descrivono, almeno in parte, il concetto di knowledge discovery in databases (KDD) inteso come un processo per l'estrazione di informazioni, o meglio conoscenze, da grosse quantità di dati. Ecco una definizione comunemente usata[4]:

Per data mining si intende il processo di selezione, esplorazione, e modellazione di grandi masse di dati, al fine di scoprire regolarità o relazioni non note a priori, e allo scopo di ottenere un risultato chiaro e utile al proprietario del database.

Il processo per il KDD è costituito da un numero di stadi interattivi che manipolano e trasformano i dati per riuscire ad estrarre delle informazioni utili. Nel 1996 Usama Fayyad, Piatetsky-Shapiro e Smyth identificarono cinque passi nel processo per il KDD che di seguito elenchiamo con particolare attenzione allo stadio del Data Mining:

- *Selezione*
  - i dati grezzi vengono segmentati e selezionati secondo alcunin criteri al fine di pervenire ad un sottoinsieme di dati, che rappresentano il target data o dati obiettivo.

- *Preprocessamento*
  - la fase di pulizia dei dati (data cleaning) che prevede l'eliminazione dei possibili errori e la decisione dei meccanismi di comportamento in caso di dati mancanti.

- *Trasformazione*
  - quando i dati provengono da fonti diverse, è necessario effettuare una loro riconfigurazione al fine di garantirne la consistenza.

- *DataMining*
  - Apprendimento mediante classificazione. Questo schema di apprendimento parte da un insieme ben definito di esempi di classificazione per casi noti, dai quali ci si aspetta di dedurre un modo per classificare esempi non noti.

- *Interpretazione*
  - Il sistema di astrazione per classificazione crea dei pattern, ovvero dei modelli, che possono costituire un valido supporto alle decisioni. Non basta però interpretare i risultati attraverso dei grafici che visualizzano l'output della classificazione, ma occorre valutare questi modelli e cioé capire in che misura questi possono essere utili.

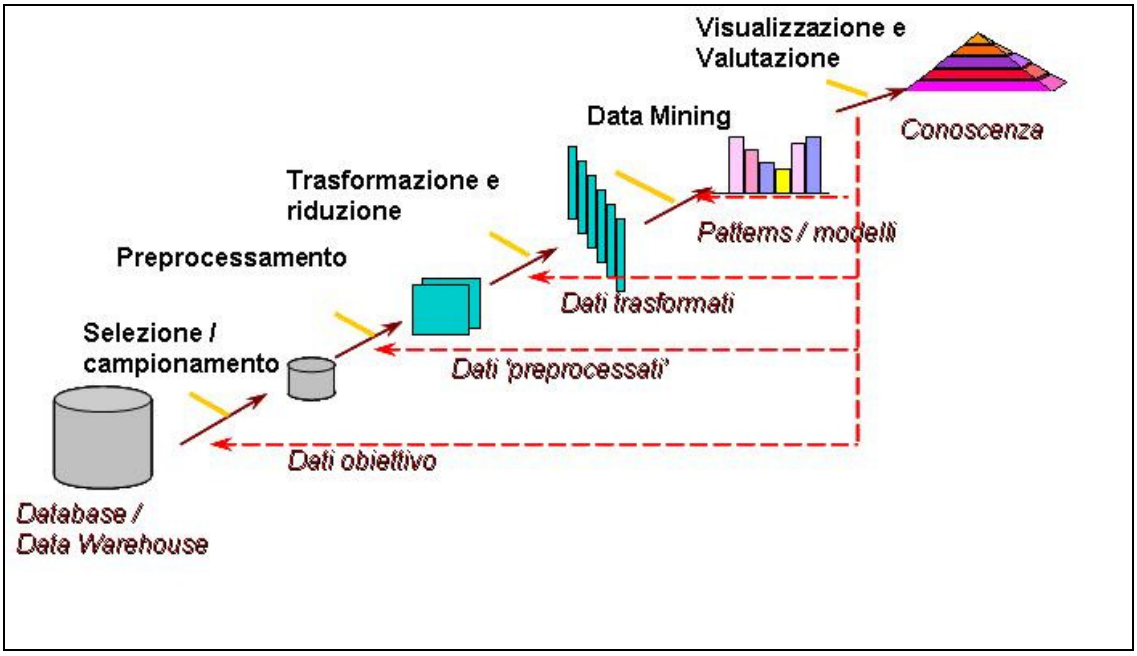

Figura 7. Processo Estrazione di conoscenza

In questo tutorial esploreremo con l'aiuto di Weka[2][3] il processo di selezione, preprocessamento e classificazione.

5. Algoritmi per la selezione

Esistono vari modalità di selezione degli attributi. Alcuni considerano un attributo alla volta e determinano la misura della sua significatività in base alla capacità di discriminare una classe da un'altra. Altri considerano invece una collezione di attributi e ne valutano l'efficacia complessiva.
Possiamo prendere come set di dati zoo.arff (fornito come esempio nel framework) e scegliere come valutatore per gli attributi il meotodo InfoGainAttributeEval che calcola, per ogni attributo, il guadagno di informazione.
Ricordiamo che

$$\mathbf{InfoGain(Attr)=H(Class) - H(Class|Attr)}$$
$$\mathbf{GainRatio(Attr)=InfoGain(Attr)/H(Attr)}$$

```
    === Attribute Selection on all input data ===

Search Method:
    Attribute ranking.

Attribute Evaluator (supervised, Class (nominal): 18 type):
    Information Gain Ranking Filter

Ranked attributes:
 2.3906    1 animal
 1.3108   14 legs
 0.9743    5 milk
 0.8657    9 toothed
 0.8301    4 eggs
 0.7907    2 hair
 0.7179    3 feathers
 0.6762   10 backbone
 0.6145   11 breathes
 0.5005   15 tail
 0.4697    6 airborne
 0.4666   13 fins
 0.3895    7 aquatic
```

```
 0.3085  17 catsize
 0.1331  12 venomous
 0.0934   8 predator
 0.0507  16 domestic

Selected attributes: 1,14,5,9,4,2,3,10,11,15,6,13,7,17,12,8,16 : 17
```

Per ogni attributo viene visualizzato un punteggio, in questo caso il gudadagno di informazione, a partire dall'attributo che ha il guadagno maggiore fino a quello che ha il guadagno minore. Normalmente tutti gli attributi vengono selezionati, indipendentemente dal valore del punteggio. Tuttavia, modificando i parametri del metodo di ricerca Ranker, è possibile cambiare il comportamento

- **numToSelect**: se viene impostato al valore n positivo, indica che si vogliono selezionare solo i primi n attributi in ordine di rilevanza. Se impostasto a un valore negativo, la selezione avviene in base al valore di soglia di cui si parla qui sotto.

- **threshold**: se numToSelect è negativo, vengono selezionati gli attributi con ranking superiore a questa soglia.

6. Algoritmi di preprocessamento

In Weka sono disponibili numerosi algoritmi di filtraggio; questi sono accessibili attraverso l'Explorer, il Knowledge Flow e l'Experimenter. Tutti i filtri trasformano in il set di dati in qualche modo. Scegliendo il filtro col bottone "choose" dall'interfaccia dell'explorer weka mostra 2 macrotipologie di filtri:

- Supervisionati
  - Esiste un attributo speciale, l'attributo classe, che viene usato per guidare le operazioni di filtraggio.
- Non Supervisionati (il processo di training avviene in modo automatico)
  - Tratta gli attributi allo stesso modo

Esempi di filtri supervisionati presenti in Weka:

- *AddCluster*: aggiunge un nuovo attributo che rappresenta la classe assegnata ad ogni istanza da un algoritmo di clustering
- *Discretize*: discretizza un attributo con il metodo dell'equi-widht o equi-depth binning
- *Normalize*: normalizza col metodo min-max, restringendo tutti gli attributi numerici all'intervallo 0-1.
- *Numeric Transform*: applica una generica funziona matematica a determinati attributi.
- *Replace Missing Values*: rimpiazza tutti i valori mancanti con la moda dell'attributo (se si tratta di attributi nominali) o la media (per attributi numerici)
- *Standardize*: normalizza col metodo z-score tutti gli attributi numerici
- *Resample*: campionamento semplice con rimpiazzamento dei dati

Esempi di filtri non supervisionati presenti in Weka:

- *Discretize*: discretizza gli attributi convertendoli da numerici a nominali
- *Resample*: campionamento con rimpiazzamento dei dati
  - può funzionare come il metodo non supervisionato oppure è in grado di effettuare il campionamento in modo che l'attributo classe abbia una distribuzione uniforme. Ciò avviene settando a 1.0 il valore dell'attributo biasToUniformClass.

7. Algoritmi per la classificazione

Nel pannello "Classify" dell'explorer è possibile selezionare un algoritmo di classificazione divisi per:

- *Bayesian Classifier* (classificatori basati sulla stima bayesiana)
- *Trees* (classificatori basati su alberi di decisione)
- *Rules* (classificatori che generano regole di decisione per discriminare le classi del dominio)
- *Functions* (classificatori che possono che possono assumere una formulazione matematica)
- *Lazy classifier* (classificatori che apprendono sullo spazio delle istanze, fornendo la classe solo al momento della classificazione (approccio pigro))
- *Miscellaneous* (classificatori non apparenenti a nessuno dei punti elencati)

Analizziamo un insieme di dati zoo.arff con i vari metodi di classificazione conosciuti.

Prima di effettuare le varie analisi, discretizziamo il set di dati usando il filtro supervisionato *Discretize*. In questo modo possiamo utilizzare anche algoritmi che funzionano solo su dati discreti.

Classificatore ZeroR
Viene selezionata come predizione la classe più frequente nel set di dati. Equivale a un albero di classificazione di altezza 0.
Accuratezza = 41%

Notare che per ZeroR (e per tutti gli algoritmi di classificazione) è possibile ottenere uno scatter plot, ma con in più l'indicazione se l'istanza è classificata correttamente oppure no.
Istanza classificate correttamente sono marcate con '+', mentre le istanze classificate non correttamente sono classificate con un quadratino

J48 e ID3
Genera un albero di classificazione.
Accuratezza: 89%

Naive Bayes
Utilizza il metodo bayesiano con incremento dei contatori di 1, in modo da evitare i problemi che sorgono con probabilità nulle.
Accuratezza: 93%

BayesNet
È un metodo per apprendere reti bayesiane. La struttura della rete può essere fissata o può essere appresa dall'algoritmo, e si possono modificare sia l'euristica usata per la ricerca della struttura della rete, sia il metodo usato per stimare le probabilità condizionate.
Accuratezza: 95%

IDk
Implementa il metodo "k-NN". E' possibile selzionare il valore di k e se si vuole che le istanze siano pesate a seconda della distanza.
Accuratezza:

- *K=1 ➔ 96%*
- *K=5 ➔ 93%*

8. La struttura di weka

Abbiamo spiegato come invocare il filtraggio (preprocessamento/selezione) e gli schemi di apprendimento con l'interfaccia Explorer, per capire a fondo come lavorano gli algoritmi all'interno di weka dobbiamo esaminare la sua struttura.

Il package weka.core è il nucleo del framework; le classi chiave di weka.core sono:

- Attribute
  - Rappresenta un attributo. Contiene il nome dell'attributo, il tipo e nel caso di attributo nominale, i suoi possibili valori.
- Instance
  - Contiene il valore dell'attributo di una particolare istanza;
- Instances
  - Un oggetto di una classe Instances contiene un insieme ordinato di istanze, in altre parole un dataset.

Il package weka.classifiers contiene gli algoritmi di classificazione e predizione numerica. La classe più importante contenuta nel package è Classifier, la quale definisce la struttura generale di ogni schema di classificazione o di predizione numerica. Classifier contiene tre metodi ereditati in ogni classe che la estende:

- buildClassifier(Instances instances)
  - genera/addestra il classificatore
- classifyInstance(Instance instance)
  - classifica un'instanza attraverso il metodo distributionForInstance
- distributionForInstance(Instance instance)
  - calcola le probabilità di appartenenza dell'istanza alle classi definite

A titolo di esempio mostriamo i metodi del classificatore *DecisionStump*:

void	**buildClassifier**(Instances instances)
double[]	**distributionForInstance**(Instance instance)
java.lang.String	**globalInfo**()
static void	**main**(java.lang.String[] argv)
java.lang.String	**toSource**(java.lang.String className)
java.lang.String	**toString**()

Notiamo che vengono sovrascritti i metodi buildClassifier e distributionForInstance e che il metodo classifiyInstance di Classifier, utilizza distributionForInstance.

Per capire a fondo come utilizzare uno dei classificatori utilizzando le librerie messe a disposizione (weka.jar dalla root della directory di installazione) direttamente da codice java, mostriamo un frammento di codice: ipotizziamo di avere delle stringhe che rappresentano il DNA umano, e di voler identificare (dopo un training adeguato) due classi, che abbiamo logicamente rappresentato come N e notN.

```
...

public class BioClassifier implements Serializable
{
	private Instances m_Data = null;
	private StringToWordVector m_Filter = new StringToWordVector();
	private Classifier m_Classifier = null;
	private boolean m_UpToDate;

	private static final String N = "N";
	private static final String NOTN = "notN";

...
```

Nella dichiarazione delle variabili di istanza notiamo il filtro non supervisionato `StringToWordVector`. Tale filtro svolge la funzione di trasformazione della stringa in un vettore di occorrenze di item identici presenti negli attributi analizzati.
Con le costanti N e notN identifichiamo le 2 possibili classi di appartenenza.

```
...

	public BioClassifier() throws Exception
	{
		m_Classifier = new Id3();

		String nameOfDataset = "BioInformaticsProblem";
		FastVector attributes = new FastVector(2);
		attributes.addElement(new Attribute("dna", (FastVector)null));
		FastVector classValues = new FastVector(2);
		classValues.addElement(N);
		classValues.addElement(NOTN);
		attributes.addElement(new Attribute("Class", classValues));

		// ipotizziamo 100 elementi nel vettore
		m_Data = new Instances(nameOfDataset, attributes, 100);

		m_Data.setClassIndex(m_Data.numAttributes() - 1);
	}
...
```

Nel costrutture inizializziamo il classificatore con un albero di decisione Id3.

Notiamo inoltre che la penultima istruzione identifica il numero di attributi del pattern valorizzato a 100.

```
...

	public void trainN(String dna) throws Exception
	{
		train(dna,N);
	}

	public void trainNotN(String dna) throws Exception
	{
		train(dna,NOTN);
	}

	private void train(String dna, String classValue) throws Exception
	{
		Instance instance = makeInstance(dna, m_Data);
		instance.setClassValue(classValue);
		m_Data.add(instance);
		m_UpToDate = false;
	}

...
```

Il metodo train aggiunge semplicemente un istanza, con la relative classe di appartenenza, al set di dati da analizzare.

```
...

	public String classify(String dna) throws Exception
	{
		if (m_Data.numInstances() == 0)
		{
			throw new Exception("No classifier available.");
		}

		if (!m_UpToDate)
		{
			m_Filter.setInputFormat(m_Data);
			Instances filteredData  = Filter.useFilter(m_Data, m_Filter);
			m_Classifier.buildClassifier(filteredData);
			m_UpToDate = true;
		}

		Instances testset = m_Data.stringFreeStructure();
		Instance instance = makeInstance(dna, testset);
		m_Filter.input(instance);
		Instance filteredInstance = m_Filter.output();
		double predicted = m_Classifier.classifyInstance(filteredInstance);

		return (m_Data.classAttribute().value((int)predicted));
	}

	private Instance makeInstance(String text, Instances data)
	{
		Instance instance = new Instance(2);
		Attribute messageAtt = data.attribute("dna");
		instance.setValue(messageAtt, messageAtt.addStringValue(text));
		instance.setDataset(data);
		return instance;
	}
```

. . .

Durante la prima classificazione, il metodo classify, addestra lo schema di apprendimento e restituisce la stringa della classe ritrovata classificazione.

9. Alberi di decisione

Un albero di decisione (o "decision tree") accetta come input un oggetto od una situazione descritta da un insieme di proprietà e restituisce in output una "decisione" costituita da un valore booleano. In realtà, è possibile anche implementare alberi di decisione in grado di rappresentare un range di output più ampio, ma in questa trattazione ci limiteremo, per semplicità, ai decision tree booleani [1][4].

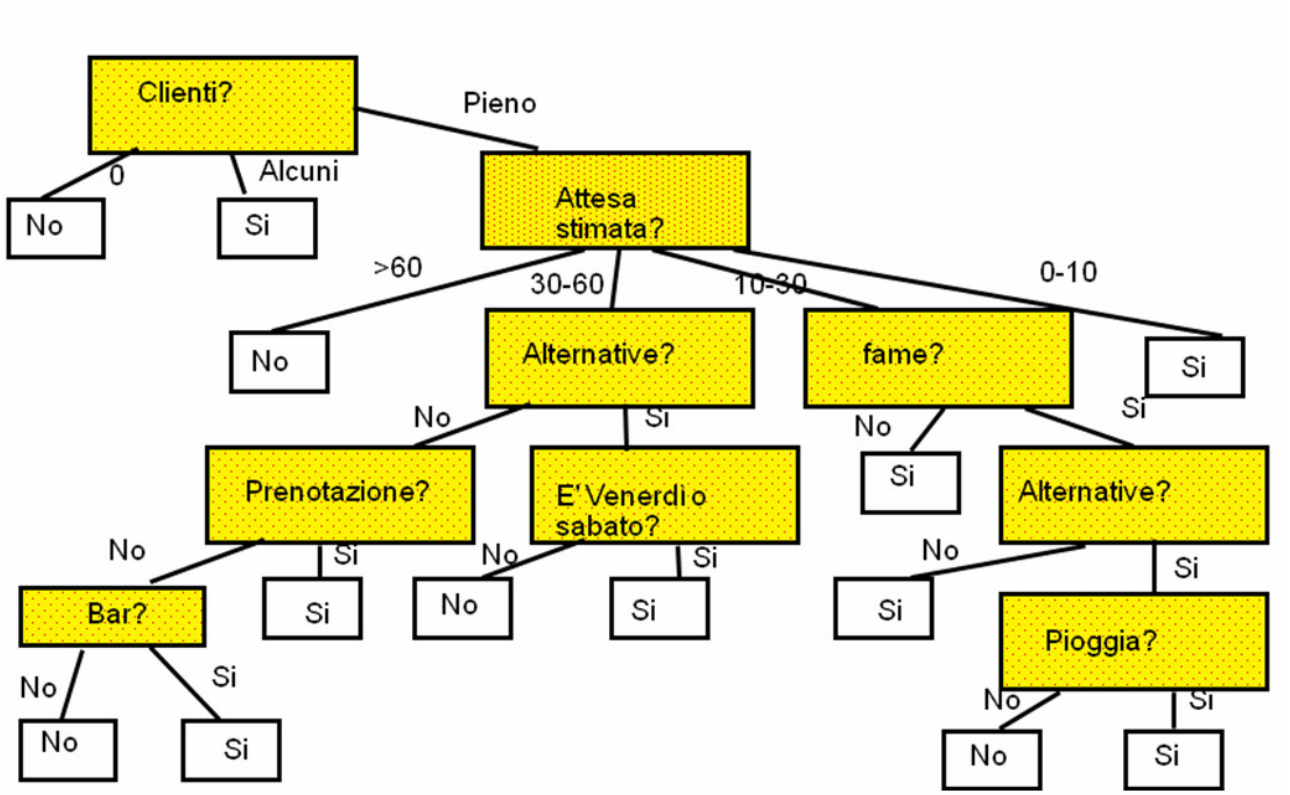

Figura 8. Esempio di albero di decisione: al ristorante... aspettare o meno?

Ogni nodo interno dell'albero corrisponde al test del valore di una delle proprietà, e i nodi dell'albero sono etichettati con i possibili esiti di tale test (Nel nostro caso booleano solo "true" e "false"). Ogni nodo foglia dell'albero è etichettato con il valore da riportare nel caso in cui sia raggiunta la foglia. Purtroppo gli alberi di decisione sono fortemente limitati: infatti, a causa della loro natura essenzialmente proposizionale, sono implicitamente condannati a riferirsi ad un unico oggetto. Se vogliamo rappresentare con un albero di decisione l'algoritmo che ci porterà a scegliere o meno un paio di scarpe sulla base del prezzo più basso e ci sono 200 modelli diversi di scarpe tra cui scegliere, sarà necessario creare un albero di decisione di grandissima profondità, poiché si dovranno confrontare tutti i prezzi con tutti gli altri. Per alcune particolari funzioni questo è un grosso problema: le più famose sono la funzione di parità, che restituisce "true" solo se un numero pari di input è "true" (1), e la funzione di maggioranza, che restituisce "true" più della metà degli input sono "true". In entrambi i casi sarà necessario un albero con un numero di foglie esponenziale con base 2 rispetto alla dimensione dell'input (n).
Quante funzioni ci sono in questo insieme? Esattamente quante sono le volte di verità che siamo in grado di scrivere, poiché una funzione binaria è completamente definita dalla sua tavola di verità. Poiché l'input di ognuna di queste funzioni è composto da n attributi, ognuna delle tavole di verità avrà righe (tutte le combinazioni possibili su n bit). Se sono necessari bit per definire una funzione, ciò vuol dire che ci sono differenti funzioni su n attributi.

9.1. *Il problema: creare nuova informazione*

Finora abbiamo creato il nostro albero di decisione, ma non abbiamo ancora detto nulla su come affrontare problemi che il nostro povero agente non ha mai dovuto affrontare. Fin qui, in realtà, abbiamo utilizzato i decision tree semplicemente come una struttura di dati nella quale memorizzare l'algoritmo che il nostro programma deve eseguire per fornire in output un valore di verità.
L'obiettivo è quello di estrapolare un pattern dalle nostre osservazioni, in modo da poter far fronte a problemi cui non ci siamo mai trovati davanti. Un pattern, in realtà, non è altro che un modo coinciso per rappresentare un gran numero di casi. Un esempio banale di un pattern utilizzato per rappresentare un concetto in modo coinciso è fornito da una banale espressione regolare.

Una funzione è un altro esempio di metodo per la rappresentazione in modo fortemente coinciso di un concetto: ad esempio la funzione x+1 è un modo molto conciso per rappresentare la corrispondenza tra infinite coppie di numeri (tutti i valori possibili per la variabile dipendente ed i rispettivi valori della variabile indipendente).
Dobbiamo quindi trovare un modo per reperire un albero decisionale piccolo e che "funzioni" con tutti i possibili input.

Dato un insieme di dati, ci sono solo poche ipotesi semplici che possono spiegare tali rilevazioni, e molte ipotesi complesse. Ma poiché le ipotesi semplici sono molte di meno, è meno probabile che ci sia un'ipotesi semplice in grado di spiegare tutti i dati sperimentali non essendo l'ipotesi corretta. Da ciò possiamo concludere che è più probabile che una ipotesi semplice consistente con tutte le rilevazioni sia corretta rispetto ad un'ipotesi complessa con le medesime caratteristiche.
Sfortunatamente, trovare il più piccolo albero di decisioni possibile è un problema intrattabile, mentre trovarne uno dei più piccoli è possibile mediante alcune semplici euristiche (ad esempio l'information gain descritto nel seguito).

10. Un albero di decisione in Weka

Per capire come estendere il framework Weka[2][3] con algoritmi personalizzati di classificazione, analizzeremo lo schema di apprendimento basato su alberi di decisone e lo integreremo all'interno del workbench.

La classe che definisce il nuovo schema di classificazione deve estendere la classe Classifier:

```
public class MyDecisionTree extends Classifier
{
      private Id3[] m_Successors;
      private Attribute m_Attribute;
      private double m_ClassValue;
      private double[] m_Distribution;
      private Attribute m_ClassAttribute;

      public String globalInfo()
      {
            return "Classe per costruire un albero di decisione"
      }
...
}
```

Il primo metodo in MyDecisionTree è globalInfo(): mostra semplicemente la stringa che viene visualizzata nell'interfaccia grafica di Weka quando lo schema di classificazione viene selezionato.

10.1. *buildClassifier()*

Il metodo buildClassifier() costruisce il classificatore da un insieme di training. In prima battuta controlliamo i valori degli attributi passati in training: gli attributi devono essere nominali e tutti valorizzati. Successivamente facciamo una copia del training set e chiamiamo un metodo da weka.core.Instances per cancellare tutte le istanze (instances) con valori di classe mancanti (infatti non possono far parte dell'insieme di apprendimento). Infine chiamiamo il metodo makeTree(), il quale costruisce l'albero di decisione ricorsivamente.

```
...
public void buildClassifier(Instances data) throws Exception
{
      if (!data.classAttribute().isNominal())
      {
            throw new UnsupportedClassTypeException
                  ("Id3: le classi devono essere nominali.");
      }

      Enumeration enumAtt = data.enumerateAttributes();
      while (enumAtt.hasMoreElements())
      {
            if (!((Attribute) enumAtt.nextElement()).isNominal())
            {
```

```
                throw new UnsupportedAttributeTypeException
                      ("Id3: gli attributi devono essere nominali.");
            }
      }

      Enumeration enum = data.enumerateInstances();
      while (enum.hasMoreElements())
      {
            if (((Instance) enum.nextElement()).hasMissingValue())
            {
                  throw new NoSupportForMissingValuesException
                        ("Id3: non devono esserci valori mancati.");
            }
      }

      data = new Instances(data);
      data.deleteWithMissingClass();
      makeTree(data);
}
...
```

10.2. *makeTree()*

Il primo passo in makeTree() è controllare se il dataset è vuoto, se lo è viene creata una foglia settando mAttribute a null. Il valore della classe mClassValue assegnato alla foglia è settato come assente e la probabilità stimata per ogni classe del dataset in mDistribution è inizializzata a 0. Se le istanze di training sono presenti, il metodo makeTree() trova gli attributi che contengono l'information gain più alto.
Il metodo computeInfoGain() calcola l'information gain di ogni attributo e lo memorizza in un array (vedi sottoparagrafo successivo).
In ottica ricorsiva, il metodo makeTree() costruisce un array di oggetti MyDecisionTree, uno per ogni valore dell'attributo e chiama makeTree() su ognuno passandogli in input il relativo dataset.

```
...
private void makeTree(Instances data) throws Exception
{
      // Controlla se non ci sono istanze che raggiungono questo nodo
      if (data.numInstances() == 0)
      {
            mAttribute = null;
            mClassValue = Instance.missingValue();
            mDistribution = new double[data.numClasses()];
      }

      // Calcola il valore di Information Gain
      double[] infoGains = new double[data.numAttributes()];
      Enumeration attEnum = data.enumerateAttributes();
      while (attEnum.hasMoreElements())
      {
            Attribute att = (Attribute) attEnum.nextElement();
            infoGains[att.index()] = computeInfoGain(data, att);
      }
      mAttribute = data.attribute(Utils.maxIndex(infoGains));

      // Crea una foglia se l'Information Gain è pari a 0
      // visita i successori altrimenti

      if (Utils.eq(infoGains[mAttribute.index()], 0))
      {
            mAttribute = null;
            mDistribution = new double[data.numClasses()];
```

```
            Enumeration instEnum = data.enumerateInstances();

            while (instEnum.hasMoreElements())
            {
                  Instance inst = (Instance) instEnum.nextElement();
                  m_Distribution[(int) inst.classValue()]++;
            }

            Utils.normalize(m_Distribution);
            mClassValue = Utils.maxIndex(m_Distribution);
            mClassAttribute = data.classAttribute();
      } else {
            Instances[] splitData = splitData(data, mAttribute);
            mSuccessors = new Id3[mAttribute.numValues()];

            for (int j = 0; j < mAttribute.numValues(); j++)
            {
                  mSuccessors[j] = new Id3();
                  mSuccessors[j].makeTree(splitData[j]);
            }
     }
}
...
```

10.3. *computeInfoGain()*

Il metodo computeInfoGain() calcola il guadagno di informazione (IG) associato all'attributo in esame attraverso il calcolo dell'entropia di informazione (E):

$$IG(S,A) = E(S) - \sum_{y \in Valori(A)} \frac{|S_v|}{|S| \times E(S_v)}$$

Il metodo splitData divide il dataset in sottoinsiemi mentre computeEntropy calcola l'entropia di informazione delle istanze passate in input.

```
private double computeInfoGain(Instances data, Attribute att) throws Exception
{
      double infoGain = computeEntropy(data);
      Instances[] splitData = splitData(data, att);
      for (int j = 0; j < att.numValues(); j++)
      {
            if (splitData[j].numInstances() > 0)
            {
                  infoGain -= ((double) splitData[j].numInstances() /
                              (double) data.numInstances()) *
                                       computeEntropy(splitData[j]);
            }
      }
      return infoGain;
}
```

10.4. *classifyInstance()*

Finora abbiamo esaminato come viene costruito un albero di decisione, ora vediamo come utilizzare tale struttura per classificare un'istanza.
Ogni classificatore deve implementare un metodo classifyInstance() oppure un metodo distributionForInstance(). La classe Classifier contiene un'implementazione di default per entrmbi metodi:

- se la classe è nominale, la predizione ricade sulla classe con il valore massimo di probabilità (tale calcolo avviene mediante l'utilizzo dell'array restituito dal metodo distributionForInstance())
- se la classe è numerica, distributionForInstance() restituisce un array composto da un solo valore che classifiInstance() estrae e restituisce per la predizione.

```
public double[] distributionForInstance(Instance instance) throws
NoSupportForMissingValuesException
{
      if (instance.hasMissingValue())
      {
            throw new NoSupportForMissingValuesException
                  ("Id3: non sono accettati valori assenti.");
      }

      if (mAttribute == null)
      {
            return mDistribution;
      } else {
            return mSuccessors[(int) instance.value(mAttribute)];
            distributionForInstance(instance);
      }
}

...

public double classifyInstance(Instance instance) throws
NoSupportForMissingValuesException
{
      if (instance.hasMissingValue())
      {
            throw new NoSupportForMissingValuesException
                  ("Id3: non sono accettati valori assenti.");
      }
      if (m_Attribute == null)
      {
            return m_ClassValue;
      } else {
            return m_Successors[(int) instance.value(m_Attribute)];
            classifyInstance(instance);
      }
}
```

IV parte: Distribuire il processo di classificazione

Realizzare un sistema per distribuire il processo di classificazione su più calcolatori con il framewok weka è semplice e flessibile. Analizziamo step by step come possiamo far collaborare un pool di calcolatori per sperimentare gli algoritmi del processo di Knowledge Discovery in Data (KDD) su un insieme di dati predefinito.

11. *Experimenter: il tool di Weka per automatizzare le sperimentazioni*

Un sistema di Knowledge Discovery in Data (KDD [1][4]) è composto da un insieme di componenti che tutte insieme possono identificare ed estrarre relazioni dai dati memorizzati nella base di dati che siano nuove, utili ed interessanti.

Dato un Training set e delle classi definite dall'utente, un sistema di Data Mining può costruire molte descrizioni. Alcune di queste sono più corrette di altre, cioè alcune di esse classificano meglio gli esempi sconosciuti. Una volta definita una misura di qualità di una descrizione, la costruzione di una descrizione può essere espressa come un problema di ricerca:

trovare la migliore descrizione nell'insieme delle descrizioni costruibili

Per questo scopo Weka[2][3] mette a disposizione l'Experimenter: si tratta di una versione batch dell'Explorer, e consente di impostare veri e propri esperimenti di Data Mining.
Ad esempio, è possibile effettuare una serie di analisi su vari insiemi di dati e con svariati algoritmi, ed eseguirle alla fine tutte insieme, facilitando un confronto tra i vari tipi di algoritmi, per determinare qual è il più adatto a risolvere uno specifico problema.

12. Distribuire il processo su più macchine

Una caratteristica notevole dell'Experimenter di Weka è che può splittare un esperimento e distribuirlo su più processi. Questa caratteristica è fornita per gli utenti avanzati di Weka ed è disponibile scegliendo "Advanced" dal pannello Weka Experiment Environment.

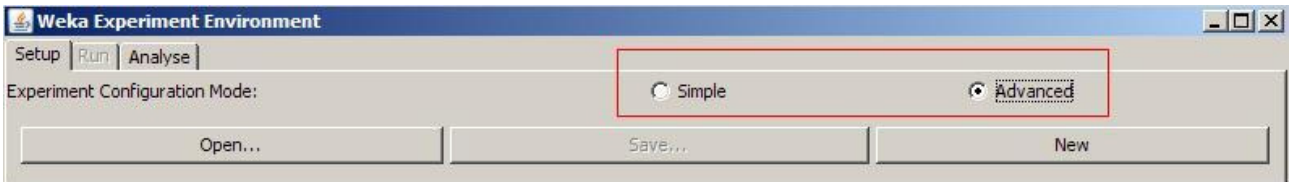

Figura 9. Dettaglio Experimenter

Alcuni utenti evitano di lavorare con questo pannello regolando l'esperimento sulla versione semplice e commutando alla versione avanzata per distribuirla. Tuttavia, la distribuzione di un esperimento è una caratteristica avanzata ed è spesso difficile da realizzare.
La distribuzione dell'esperimento funziona al meglio quando i risultati sono trasmessi ad una base di dati centrale: selezionando la base di dati con JDBC per definire la destinazione dei risultati dal pannello dell'interfaccia dell'experimenter.
Alternativamente, potreste conservare su ogni host i relativi risultati per poi fonderli in seguito.
Per distribuire un esperimento, ogni host deve avere Java installato, deve possedere i corretti grant sui dataset di riferimento e deve far girare il pacchetto weka.experiment (compreso nel workbench Weka).
Per iniziare un esperimento "a distanza" su una macchina ospite, è necessario copiare il file `remoteExperimentServer.jar` dalla distribuzione di Weka su una directory dell'host.
"Spacchettare" il file con: `java xvf remoteExperimentServer.jar`

Si espande a due file: `remoteEngine.jar`, un jar eseguibile che contiene il server dell'experimente, e `remote.policy`.

Il file `remote.policy` assegna il permesso all'engine remoto per effettuare alcune operazioni, come il collegamento alle porte o l'accesso alle directory.
Deve essere editato per specificare i percorsi corretti in alcuni dei permessi; ciò è evidente quando esaminate la file:

```
/*
 * Necessary permissions for remote engines
 *
```

```
 * Example setup for user John Doe:
 * - home directory:
 *   /home/johndoe
 * - location of datasets:
 *   /home/johndoe/datasets/
 * - location of weka.jar:
 *   /home/johndoe/weka/weka.jar
 * - location of additional jars (e.g., for database access):
 *   /home/johndoe/jars
 * - remote engine directory (policy, start scripts, etc.):
 *   /home/johndoe/remote_engine
 *
 * Start Experimenter in directory /home/johndoe/remote_engine:
 *   java \
 *     -classpath /home/johndoe/jars/<db.jar>:/home/johndoe/weka/weka.jar \
 *     -Djava.rmi.server.codebase=file:/home/johndoe/weka/weka.jar \
 *     weka.gui.experiment.Experimenter
 *
 * Start remote engine on remote machine:
 * - cd to /home/johndoe/remote_engine
 * - start engine
 *   java -Xmx256m \
 *     -classpath remoteEngine.jar:/home/johndoe/jars/<db.jar> \
 *     -Djava.security.policy=remote.policy \
 *     weka.experiment.RemoteEngine &
 *
 * Note:
 * replace <db.jar> with actual jar filename, e.g., mysql.jar
 *
 * Version: $Revision: 1.1.2.1 $
 */

grant {

  // allow the remote engine to replace the context class loader.
  // This enables the unloading of types from the remote engine
  permission java.lang.RuntimePermission
    "setContextClassLoader";

  // file permission for data sets
  permission java.io.FilePermission
    "/home/johndoe/datasets/-", "read";

  // file permissions for downloading classes from client file url's
  permission java.io.FilePermission
    "/home/johndoe/-", "read";
  permission java.io.FilePermission
    "/home/johndoe/weka/weka.jar", "read";

  // connect to or accept connections from unprivaleged ports and the http port
  permission java.net.SocketPermission
    "*:1024-65535", "connect,accept";
  permission java.net.SocketPermission
    "*:80", "connect";

  // file permission to load server classes from remoteEngine.jar.
  // only needed if RemoteEngine_Skel.class/_Stub.class are going to
  // be downloaded by clients (ie, if these files are not already in the
  // client's classpath). Normally this doesn't need to be changed.
  permission java.io.FilePermission
```

```
    "/home/johndoe/remote_engine/-", "read";

  // read write for properties
  permission java.util.PropertyPermission
    "*", "read,write";

};
```

Di default, il file specifica la porta 80 su http da qualunque parte del web, può comunque essere impostato un URL specifico: è sufficiente scommentare l'esempio ed inserire il path appropriato. I remote engines per funzionare correttamente hanno la necessità di accedere al dataset usato nell'esperimento (prima riga del file `remote.policy`).

I percorsi ai dataset sono specificati nell'experimenter (cioè, il client) e gli stessi percorsi devono essere applicabili nel contesto dei remote engines.

L'Experimenter mette a disposizione una comoda interfaccia grafica per facilitare l'inserimento dei path relativi: è sufficiente selezionare la casella "Use relative paths" indicata nel pannello di setup dell'experimenter.

Figura 10. Dettaglio Experimenter

Per avviare il server del remote engine, scrivere:

```
java  -classpath remoteEngine.jar:<path_to_any_jdbc_drivers>
      -Djava.security.policy=remote.policy weka.experiment.RemoteEngine
```

dalla directory che contiene `remoteEngine.jar`.
Se tutto va bene comparirà questo messaggio (o qualcosa di simile):

```
Host name : biancalana@dia.uniroma3.it
RemoteEngine exception: Connection refused tohost:
ml.cs.waikato.ac.nz; nested exception is:
java.net.ConnectException: Connection refused
Attempting to start rmi registry...
RemoteEngine bound in RMI registry
```

Malgrado le apparenze iniziali, questa è una buona notizia! Il collegamento è stato rifiutato perché nessun RMI registry stava funzionando su quel server e quindi il remote engine ne ha iniziato uno.
Ripetere il processo su tutti gli host tenendo conto che non ha senso avviare più di un remote engine sullo stessa macchina!
Avviare l'experimenter scrivendo:

```
java -Djava.rmi.server.codebase=<URL_for_weka_code>
weka.gui.experiment.Experimenter
```

L'URL specifica dove i remote engine possono trovare il codice da poter eseguire, se l'URL denota una directory anziché un file jar, deve terminare con il separatore di path (ad esempio "/").
Il pannello avanzato di setup dell'experimenter contiene un piccolo riquadro che determina se un esperimento deve essere distribuito oppure no, normalmente disabilitato.

Figura 11. Dettaglio Experimenter

Per distribuire l'esperimento cliccare sul check-box nominato "hosts": comparirà una finestra popup per inserire le macchine sulle quali distribuire l'esperimento, i nomi degli host devono essere fully qualified (per esempio, biancalana.dia.uniroma3.it). Una volta inseriti gli host è sufficiente utilizzare l'Explorer nel modo usuale.
Quando l'esperimento è avviato usando il pannello principale dell'explorer, viene visualizzato il progresso dei sotto-esperimenti sui vari host, con i relativi messaggi di warning.
Il task della distribuzione dell'esperimento gestisce la suddivisione in sotto-esperimenti che RMI trasmette agli host.
Gli esperimenti sono gestiti attraverso lo split del dataset: ovviamente non possono essere più ospiti che dataset. Ogni sotto-esperimento è autonomo: l'experimenter applica tutti gli schemi precalcolati ad un singolo dataset.

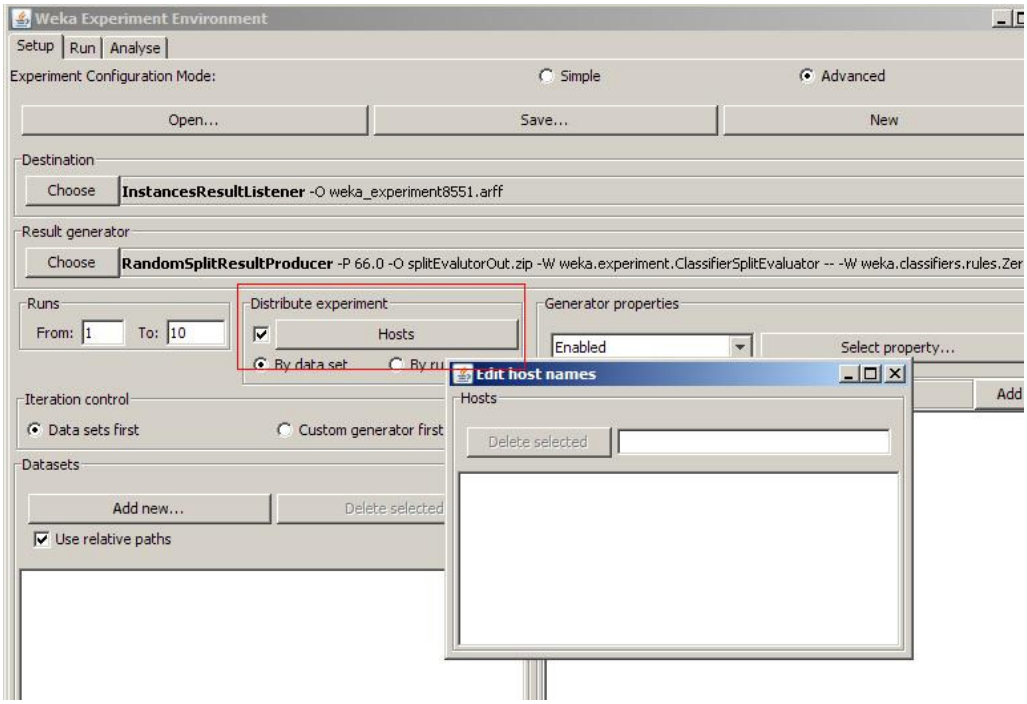

Figura 12. Dettaglio Experimenter

13. Bibliografia

- [1] "Data Mining: Practical Machine Learning Tools and Techniques with Java Implementation", I. Witten & E. Frank, Morgan Kauffman 2005.
- [2] Weka Home Site: http://www.cs.waikato.ac.nz/ml/weka/
- [3] Weka Sourceforge Site: http://sourceforge.net/projects/weka/
- [4] "Data Mining – Metodi informatici, statistici e applicazioni" Paolo Giudici, McGraw-Hill 2005.