Introduzione all'Intelligenza Artificiale
Supporto per le Esercitazioni

Dipartimento di Informatica – Università di Pisa

Corso di Laurea in Informatica

Anno Accademico 2016 – 2017
e successivi

Claudio Gallicchio, Ph.D.
Assistant Professor
Computational Intelligence and Machine Learning Group
Dipartimento di Informatica - Università di Pisa
Largo Bruno Pontecorvo 3, 56127 Pisa, Italy

web: www.di.unipi.it/~gallicch
email: gallicch@di.unipi.it

Docenti del corso
Prof. Alessio Micheli
Prof. Maria Simi

Pagina web del corso:
elearning.di.unipi.it (IIA)

Table of Contents

Indice delle Tabelle

Informazioni su questo documento

Questo documento contiene una guida all'utilizzo delle funzionalità fornite nell'implementazione in Python degli algoritmi e modelli di supporto ai temi dell'Intelligenza Artificiale e sue applicazioni, nell'ambito del corso "Introduzione all'Intelligenza Artificiale", corso di Laurea in Informatica, anno accademico 2016/2017.

Il codice descritto in questo documento rappresenta un'implementazione di alcune funzionalità e di alcuni algoritmi trattati durante il corso, con specifico riferimento alla parte riguardante l'Introduzione all'Apprendimento Automatico.

Note Ulteriori:

Il codice descritto in questo documento è stato ottenuto a partire dal codice disponibile all'indirizzo:
https://github.com/aimacode/aima-python
la cui autorizzazione all'uso è concessa secondo quanto descritto nella licenza consultabile all'indirizzo
https://github.com/aimacode/aima-python/blob/master/LICENSE
consultabile anche all'indirizzo:
https://elearning.di.unipi.it/course/view.php?id=91
nella sezione "Codice Python".

Tutte le classi e funzioni descritte in questo documento sono fornite all'interno dei file `learning.py` e `demo_learnig.py`.

Note su Python
Note sull'installazione e utilizzo di Python sono disponibili su elearning di IIA
all'interno della Sezione 1 del documento "Documentazione-ProblemSolving-IIA-python_vxx.pdf"

1. Rappresentazione di un dataset: la classe DataSet

Un dataset è rappresentato mediante la classe `DataSet`. Data una istanza `d` di `DataSet`, i suoi principali attributi sono i seguenti:

- `d.examples`: una lista di esempi, in cui ogni esempio è rappresentato come una lista di valori di attributi;
- `d.inputs`: una lista di posizioni corrispondenti agli attributi di input in ogni esempio nel dataset;
- `d.target`: l'indice corrispondente all'attributo target in ogni esempio (un solo attributo target in ogni esempio); nota che per default d.target corrisponde all'ultima posizione in ciascun esempio.

Costruire un dataset

Il modo più semplice per costruire un dataset consiste nell'importarlo dal un file in formato .csv, sfruttando le funzionalità messe a disposizione dalla classe `DataSet`. In particolare, è sufficiente invocare il costruttore di classe specificando tramite l'argomento `name` il nome del file .csv da cui caricare gli esempi del dataset. Tramite l'argomento `directory_dataset` è anche possibile specificare la directory in cui si trova il file da caricare (per default è la sotto-directory '`datasets`' della directory in cui si trova il file `learning.py`).

Ad esempio, la seguente istruzione carica il dataset Iris:

```
ds = DataSet(name="iris", directory_dataset = "datasets")
```

All'interno del file .csv indicato, gli esempi sono specificati seguendo un semplice formato in base al quale:

- ogni linea del file corrisponde ad un esempio,
- gli attributi di ciascun esempio sono riportati uno dopo l'altro, separati da virgole (secondo lo stesso ordine fissato per tutti gli esempi nel dataset).

In Figura 1 è riportato un estratto del file iris.csv, come esempio rappresentativo del formato di dati utilizzato.

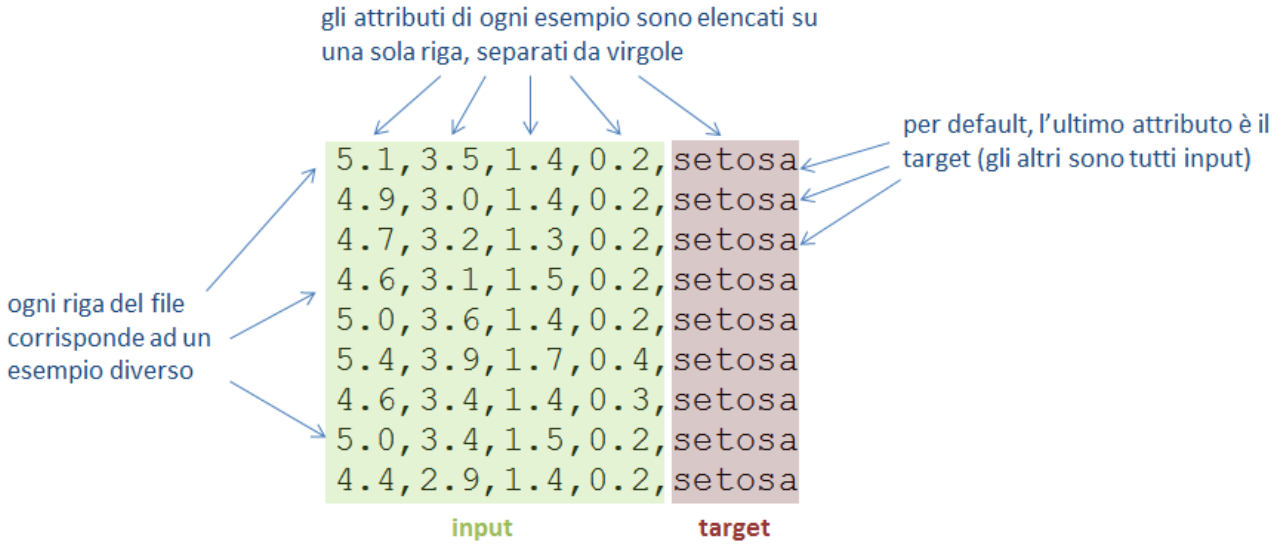

Figura 1 Esempio di formato .csv per il caricamento di un dataset da file.

Esistono diverse alternative al caricamento da file in formato .csv per la costruzione di un dataset. Tra queste, le più semplici sono elencate di seguito:

- costruzione incrementale degli esempi nel dataset, usando il metodo `add_example`;
- impostazione manuale degli attributi `d.examples`, `d.inputs` e `d.target`.

Manipolazione di un dataset

All'interno della classe DataSet, sono definite alcune funzionalità aggiuntive per la manipolazione degli esempi all'interno di un dataset. Tra queste, alcune tra le funzioni più utili sono elencate di seguito:

- `classes_to_numbers`: che converte etichette categoriche in valori numerici interi;
- `remove_examples`: elimina dal dataset gli esempi che contengono un determinato valore tra gli attributi.

Ulteriori informazioni circa le funzionalità messe a disposizione dalla classe DataSet sono descritte mediante commenti nel codice della classe stessa (a cui il lettore interessato è rimandato).

Dataset di esempio

All'interno della sotto-cartella `'dataset'` sono presenti alcuni dataset in formato .csv, forniti a titolo di esempio per le funzionalità dei modelli di apprendimento implementati. Di seguito sono brevemente elencati i file .csv comprendenti i dataset e alcune loro caratteristiche principali (si ricordi che in tutti i dataset il target è monodimensionale):

- `iris.csv`
  - classificazione a 3 classi
  - 150 esempi (50 per ciascuna classe)
  - 4 attributi di input, usati per rappresentare caratteristiche osservabili del fiore
  - il target è la classe che specifica il tipo di iris;
- `restaurant.csv`
  - classificazione binaria
  - 12 esempi (6 per ciascuna classe)
  - gli esempi corrispondono ai dati riportati in Figura 18.3 del libro AIMA- Russell Norvig (a cui il lettore è rimandato per una descrizione degli attributi di input e di target);
- `simplefit.csv`
  - regressione
  - 94 esempi
  - 1 attributo di input (a valori continui)
- `abalone.csv`
  - regressione
  - 4177 esempi
  - 8 attributi di input, usati per rappresentare alcune caratteristiche osservabili della conchiglia di una particolare varietà di mollusco
  - Il target è un numero intero che indica il numero di anelli sulla conchiglia
- `bodyfat.csv`
  - regressione
  - 252 esempi

- 13 attributi di input, usati per descrivere caratteristiche misurabili circa il corpo di una persona (come ad es. Età, peso, altezza, circonferenza del petto, etc.)
- Il target è un numero reale che rappresenta la percentuale di grasso corporeo

Ulteriori dataset sono disponibili all'indirizzo web:
https://github.com/aimacode/aima-data

2. Misure d'errore

Le funzioni implementate per valutare l'efficacia dell'apprendimento nei modelli considerati sono brevemente elencate qui di seguito:

- `err_ratio`
  Calcola la proporzione di esempi che non sono correttamente predetti (assumendo che un esempio sia correttamente predetto solo se l'output è identico al target). E' una funzione adatta al caso di classificazione con etichette categoriche (rappresentazione dell'output a etichette).
- `accuracy_binary`
  Calcola l'accuratezza di un predittore per un problema di classificazione binaria. Questa funzione assume che il target sia un valore nell'insieme {0,1}, e che l'output sia un numero reale (che viene discretizzato in {0,1} all'interno della funzione, usando come valore soglia 0.5).
- `misclassification_loss`
  Calcola l'errore di classificazione per un problema di classificazione binaria. Il valore restituito è quindi `1-accuracy_binary`.
- `mse_loss`
  Calcola l'errore quadratico medio per un problema di regressione.

3. Valutazione della performance del modello

Le principali funzioni fornite per l'implementazione delle metodologie di base per valutare la performance del modello di apprendimento sono brevemente elencate qui di seguito:

- `hold_out`
  Funzione usata per valutare la performance del modello di apprendimento secondo uno schema di hold-out cross validation. Il dataset in ingresso viene separato in insiemi di training, validation e test secondo una proporzione che è specificata dai parametri in ingresso. In output restituisce la una lista contenente, nell'ordine, la performance del modello sui dati di training, validation e test. Nel caso il modello di apprendimento usato sia il modello lineare, restituisce anche il vettore dei pesi addestrati e l'errore quadratico medio sui dati di training per tutte le epoche.
- `cross_validation` (per model selection)
  Funzione usata per valutare la performance del modello di apprendimento secondo uno schema di k-fold cross validation Il dataset in ingresso viene separato in k parti, e per ognuna delle fold solo una delle parti viene usata come validation set, lasciando gli altri esempi nel training set. In output restituisce la una lista contenente, nell'ordine, la performance del modello sui dati di training e validation (mediati sulle k fold).

4. Modello Lineare

In questa sezione viene brevemente descritto l'algoritmo per l'addestramento di un modello lineare.
Nello specifico, è presentata una implementazione in Python del solo algoritmo di training, fornita come strumento basilare e a solo scopo illustrativo. L'invocazione e l'uso appropriato è demandato a funzioni ausiliari e alla loro invocazione, vedi la sotto-sezione *Demo d'uso*.
L'uso del codice qui fornito è quindi da intendere con alcune avvertenze (warning). In particolare:

- il controllo di fine algoritmo non ha un criterio di uscita significativo (vi è qui semplicemente un ciclo `for` sul numero di epoche, fissate da utente, con default uguale a 100);
- non vi sono strumenti per il plot della learning curve (errore ad ogni epoca) in genere utili per la gestione e analisi di tool di Machine Learning;
- le misure di performance sono limiate al Mean Square Error (MSE) e all'accuracy (per classificazione);
- la gestione della cross validation è implementata in modo primitivo;
- non ci sono ottimizzazioni nell'uso delle strutture dati, nell'uso delle matrici, ecc. ;
- l'uso degli indici per scegliere le variabili di input o di target (come descritto in Sezione 1 a proposito dell'organizzazione del dataset, è una possibilità tra le molte);
- non è implementata alcuna tecnica di regolarizzazione;
- è possibile avere un solo attributo target.

L'algoritmo di addestramento per il modello lineare è implementato nella funzione LinearLearner, il cui codice è riportato in Tabella Tabella 1. La funzione richiede in ingresso:

- `dataset`: un oggetto della classe `DataSet`, contenente i soli esempi di training (il training set)
- `learning_rate`: il learning rate (default value = 0.01), usato nella regola per l'aggiornamento dei pesi del modello (eta a lezione);
- `epochs`: il numero massimo di epoche di addestramento (default value = 100).

In output la funzione restituisce una lista contente, nell'ordine:

- il predittore, da usare in fase di predizione con funzione predict;
- una lista contenente i pesi addestrati del modello (solitamente indicati con **w**);
- l'errore quadratico medio (MSE) di training al variare delle epoche.

Il codice relativo all'implementazione dell'algoritmo di apprendimento per un modello lineare è riportato in Tabella 1. Si noti che l'output del modello lineare è un numero reale (un valore continuo), quindi nel caso di problemi di classificazione per calcolare l'uscita di classificazione e valutare la performance del classificatore occorre usare la funzione `accuracy_binary` che applica una soglia all'output in modo da discretizzarne il valore (Linear Threshold Unit).

```
def LinearLearner(dataset, learning_rate=0.01, epochs=100):
    """Addestra un modello lineare.
       In input richiede:
        dataset: oggetto della classe DataSet che contiene gli esempi
                 per il training e le funzioni per la gestione del data set
        learning_rate: eta a lezione (default 0.01)
        epochs: numero massimo di epoche di training (default 100)
       In output restituisce una lista contenente (nell'ordine):
        il predittore (per uso funzione predict, uso post training)
        i parametri addestrati (w)
        l'errore quadratico medio (MSE) di training al variare delle epoche
    """
    idx_i = dataset.inputs # indici delle variabili di input
    #nota: idx_i e' una lista degli indici nei dati di ogni esempio
    #   come [0,1,2,3], o [2,5,7]
    idx_t = dataset.target # indice della variabile usata per target
```

```
                            #(variabile target y a lezione)
    examples = dataset.examples #lista di tutti gli esempi
    #nota: ogni esempio corrisponde a una lista contenente sia le variabili
    # di input sia il target
    num_examples = len(examples) # numero di esempi nel dataset (l a lezione)
    input_dim = len(idx_i) # dimensione dell'input (n a lezione)
    #inizializza i pesi del modello in modo casuale
    #da una distribuzione uniforme tra -0.5 e 0.5
    w = [random.uniform(-0.5, 0.5) for _ in range(len(idx_i) + 1)]

    trainingError = [] # inizializza la lista che conterra' l'MSE ad ogni epoca di tr.
    for epoch in range(epochs):
        err = []
        inputs = [] #conterra' l'input di ogni esempio nel dataset
        #considera tutti gli esempi, prepara il calcolo gradiente
        for example in examples:
            x = [1] + [example[i] for i in idx_i] #input (lista locale dei
                                                  #valori delle variabili di input)
                                                  #e aggiunge l'input bias unitario
            inputs.append(x) #aggiunge l'esempio attuale alla lista
            out = dotproduct(w, x) #output y del modello
            t = example[idx_t] #target (fu y a lezione)
            err.append(t - out)
        for i in range(len(w)): #calcolo il DeltaW
            delta_wi = 0
            for p in range(num_examples):
                x_pi = inputs[p][i] #componente i del pattern p
                delta_wi += err[p] * x_pi
            #stiamo tenendo la costante 2 come a lezione,
            #ma dividendo per il numero di esempi (l) , ossia una LMS
            delta_wi = 2 * (delta_wi /num_examples)
            w[i] = w[i] + learning_rate * delta_wi
        # calcola l'MSE per questa epoca e lo aggiunge alla lista
        trainingError.append(sum(e*e for e in err)/len(err))

    def predict(example):
        # funzione usata per calcolare l'output del modello
        # per l'esempio specificato in input
        x = [1] + example
        return dotproduct(w, x)

    return predict, w, trainingError
```

Tabella 1 Addestramento di un modello lineare di apprendimento .

Demo d'uso per il Modello

Nel file `demo_linear.py` è fornita l'implementazione di alcune funzioni di esempio che mostrano i passi necessari al caricamento del dataset e all'addestramento del modello lineare. Allo scopo di fornire esempi di funzionalità relativi a diversi casi, ogni funzione è specifica per un diverso dataset (quindi la natura del problema di apprendimento da affrontare cambia a seconda dei casi).

Qui di seguito illustriamo alcune delle funzioni implementate in `demo_linear.py`. Il lettore è rimandato ai commenti nel codice in `demo_linear.py` per ulteriori dettagli.

Addestramento del modello lineare: semplice problema di regressione

In Tabella 2 è riportato il codice di esempio per l'addestramento del modello lineare su un task di regressione in cui input e output hanno entrambi dimensione 1.

Il dataset 'simplefit' viene caricato tramite chiamata al costruttore della classe DataSet. Quindi, viene chiamata la funzione `cross_validation`, specificando `LinearLearner` come modello di apprendimento, il dataset caricato, l'errore quadratico medio come funzione d'errore, il numero di fold e i

valori scelti per learning rate e massimo numero di epoche. Infine viene visualizzato l'errore in training e validation ottenuto dal modello (mediato sul numero di fold). Ad esempio, questo processo può essere ripetuto con diversi valori degli iper-parametri (per esempio in questo caso il learning rate) per effettuare una model selection (non c'è quindi un test set).

```
def demo_ll_regression1():
    # modello lineare applicato al task di regressione 'simplefit'
    d = DataSet(name="simplefit")
    n_folds = 5 # 5-fold cross validation
    cv_performance = cross_validation(LinearLearner, d, loss = mse_loss,
                                      k = n_folds, learning_rate = 0.01, epochs = 100)


    print('Performance del modello lineare sul task di regressione "simplefit"')
    print('TR MSE = %s -- VL MSE = %s' % (cv_performance[0],cv_performance[1]))
```

Tabella 2 Esempio di addestramento del modello lineare per un semplice problema di regressione

Visualizzazione dei parametri addestrati e dell'errore di training

In Tabella 3 è riportato il codice di esempio che mostra come sia possibile accedere a informazioni come il vettore dei pesi del modello dopo l'addestramento, e l'andamento dell'errore di training all'aumentare del numero delle epoche (utile per verificare il fitting del modello sui dati di training nel setting specificato dai parametri, tra cui ad esempio il learning rate). In questo caso si applica uno schema di cross-validation di tipo hold out con Training (TR), Validation (VL) e Test (TS) set.

```
def demo_ll_regression_w_err():
    # modello lineare applicato al task di regressione 'simplefit'
    d = DataSet(name="simplefit")
       # il metodo per l'hold out restituisce
       # nelle ultime due posizioni
       # una lista contenente il vettore dei pesi addestrati
       # e una lista contenente l'errore sul training set durante le
       # epoche di training
    ho_data = hold_out(LinearLearner, d, loss = mse_loss, propTr = 0.70, propVl = 0.20,
                                    learning_rate = 0.01, epochs = 1000)

    print('Performance del modello lineare sul task di regressione "simplefit"')
    print('TR MSE = %s -- VL MSE = %s -- TS MSE = %s' % (ho_data[0],

        ho_data[1],ho_data[2]))
    print('I parametri del modello lineare appresi sul training set: %s ' % ho_data[3])
    print('MSE sul training set al variare delle epoche: %s' % ho_data[4])
```

Tabella 3 Esempio di visualizzazione dei parametri addestrati e dell'errore di training per il modello lineare.

In questo semplice esempio, il vettore dei parametri appresi dal modello contiene solo due elementi: il primo è il peso associato all'input bias e il secondo è il peso associato all'attributo di input (c'è un solo attributo di input nel caso del dataset 'simplefit'). Provando a fare un plot (con uno strumento software esterno, come ad esempio Matlab) gli esempi nel dataset (asse X: input, asse Y: target output) e nella stessa figura l'iper-piano separatore appreso dal modello (una linea su un piano, in questo caso) si ottiene un utile esempio per valutare l'efficacia dell'apprendimento.

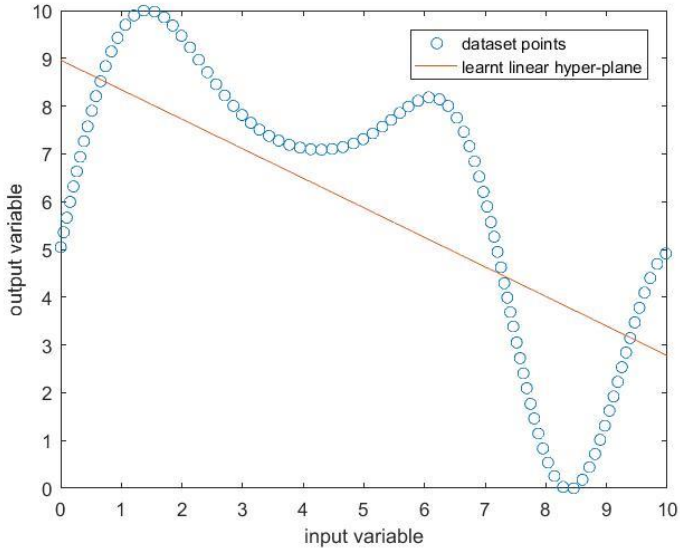

Figura 2 Esempio di iper-piano appreso dal modello lineare sul dataset 'simplefit'.

In Figura 2 è mostrato un plot ottenuto a partire dall'output (ho_data[3]) generato dal codice in Tabella 3. In Figura 3 viene inoltre mostrato un plot, ottenuto con Matlab a partire dall'output (ho_data[4]) del codice in Tabella 3, raffigurante l'andamento dell'errore quadratico medio in training all'aumentare del numero di epoche.

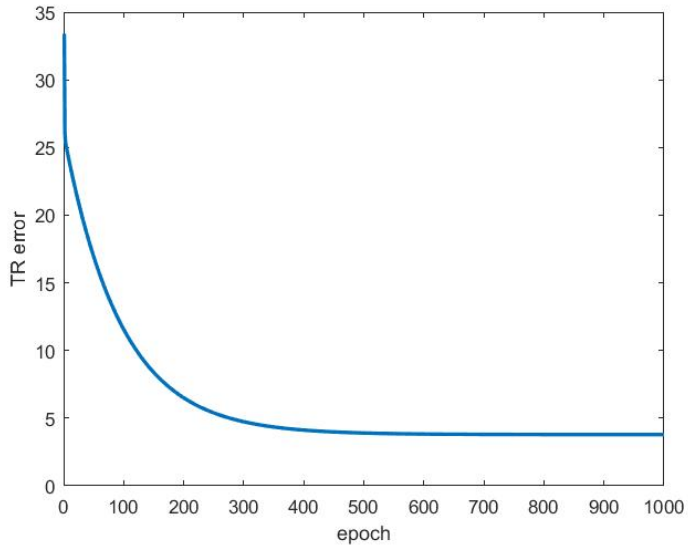

Figura 3 Esempio di learning curve del modello lineare (solo l'errore sul training set è visualizzato in questo plot).

Ulteriori esempi di applicazione del modello lineare a problemi di natura differente (con riferimento ai dataset di esempio elencati in Sezione 1) sono riportati in `demo_linear.py`.

5. Albero di Decisione

In questa sezione viene brevemente descritto il codice per l'implementazione dell'algoritmo di addestramento di un albero di decisione, come discusso nel libro AIMA- Russel-Norvig (con particolare riferimento alla Figura 18.5). L'algoritmo di addestramento per un albero di decisione è implementato mediante la funzione DecisionTreeLearner, riportato in Tabella 4. L'algoritmo richiede in input un oggetto di classe DataSet e restituisce in output il predittore che può essere utilizzato con sintassi simile a quella vista per il modello lineare. Ad esempio, possono essere usate in modo simile le funzioni per hold out e k-fold cross validation (in questo caso però non vanno specificati i valori di learning rate e numero massimo di epoche, e piu in generale questa versione elementare non prevede iper-parametri, ne tecniche di pruning, etc.). Si faccia inoltre attenzione al fatto che l'algoritmo qui fornito è adatto solo al caso di dataset con attributi discreti (o categorici) e si usa la loss `err_ratio` nella chiamata (`loss = err_ratio`).

```
def DecisionTreeLearner(dataset):
    """Addestra un albero di decisione.
       In input richiede:
        dataset: oggetto della classe DataSet che contiene gli esempi
                 per il training e le funzioni per la gestione del data set
       In output restituisce il predittore
       (per uso post training)
    """

    target, values = dataset.target, dataset.values

    def decision_tree_learning(examples, attrs, parent_examples=()):
        if len(examples) == 0:
            return plurality_value(parent_examples)
        elif all_same_class(examples):
            return DecisionLeaf(examples[0][target])
        elif len(attrs) == 0:
            return plurality_value(examples)
        else:
            A = choose_attribute(attrs, examples)
            tree = DecisionFork(A, dataset.attrnames[A])
            for (v_k, exs) in split_by(A, examples):
                subtree = decision_tree_learning(
                    exs, removeall(A, attrs), examples)
                tree.add(v_k, subtree)
            return tree

    def plurality_value(examples):
        """Return the most popular target value for this set of examples.
        (If target is binary, this is the majority; otherwise plurality.)"""
        popular = argmax_random_tie(values[target],
                                    key=lambda v: count(target, v, examples))
        return DecisionLeaf(popular)

    def count(attr, val, examples):
        """Count the number of examples that have attr = val."""
        return sum(e[attr] == val for e in examples)

    def all_same_class(examples):
        """Are all these examples in the same target class?"""
        class0 = examples[0][target]
        return all(e[target] == class0 for e in examples)

    def choose_attribute(attrs, examples):
        """Choose the attribute with the highest information gain."""
        return argmax_random_tie(attrs,
                                 key=lambda a: information_gain(a, examples))

    def information_gain(attr, examples):
        """Return the expected reduction in entropy from splitting by attr."""
        def I(examples):
```

```
            return information_content([count(target, v, examples)
                                        for v in values[target]])
        N = float(len(examples))
        remainder = sum((len(examples_i) / N) * I(examples_i)
                        for (v, examples_i) in split_by(attr, examples))
        return I(examples) - remainder

    def split_by(attr, examples):
        """Return a list of (val, examples) pairs for each val of attr."""
        return [(v, [e for e in examples if e[attr] == v])
                for v in values[attr]]

    return decision_tree_learning(dataset.examples, dataset.inputs)


def information_content(values):
    """Number of bits to represent the probability distribution in values."""
    probabilities = normalize(removeall(0, values))
    return sum(-p * math.log2(p) for p in probabilities)

class DecisionFork:
    """A fork of a decision tree holds an attribute to test, and a dict
    of branches, one for each of the attribute's values."""

    def __init__(self, attr, attrname=None, branches=None):
        """Initialize by saying what attribute this node tests."""
        self.attr = attr
        self.attrname = attrname or attr
        self.branches = branches or {}

    def __call__(self, example):
        """Given an example, classify it using the attribute and the branches."""
        attrvalue = example[self.attr]
        return self.branches[attrvalue](example)

    def add(self, val, subtree):
        """Add a branch.  If self.attr = val, go to the given subtree."""
        self.branches[val] = subtree

    def display(self, indent=0):
        name = self.attrname
        print('Test', name)
        for (val, subtree) in self.branches.items():
            print(' ' * 4 * indent, name, '=', val, '==>', end=' ')
            subtree.display(indent + 1)

    def __repr__(self):
        return ('DecisionFork({0!r}, {1!r}, {2!r})'
                .format(self.attr, self.attrname, self.branches))


class DecisionLeaf:
    """A leaf of a decision tree holds just a result."""

    def __init__(self, result):
        self.result = result

    def __call__(self, example):
        return self.result

    def display(self, indent=0):
        print('RESULT =', self.result)

    def __repr__(self):
        return repr(self.result)
```

Tabella 4 Addestramento di un semplice albero di decisione.

6. K-Nearest Neighbor

In questa sezione viene descritto il codice relativo all'implementazione dell'algoritmo k-Nearest Neighbor. L'algoritmo viene fornito in tre varianti, riportate in Tabella 5:

- `NearestNeighborClassifier`
  da usare per problemi di classificazione binaria con etichette in {0,1};
- `NearestNeighborClassifier_cat`
  da usare per problemi di classificazione con etichette di classe categoriche;
- `NearestNeighborRegressor`
  da usare per problemi di regressione.

In tutti i tre casi, in input è richiesto il dataset contenente gli esempi di riferimento (tra i quali cercare i vicini) e il valore di k che rappresenta il numero di vicini da considerare per ogni esempio (con default 1, cioè 1-NN). In output viene restituito il predittore, il cui uso segue la stessa sintassi descritta per il modello lineare, con la differenza che in questo caso nell'invocare le funzionalità per hold out e cross –fold validation occorre specificare il valore di k (e non il learning rate e il numero di epoche). La loss dipende dalle varianti sopra riportate.

```
def NearestNeighborClassifier(dataset, k=1):
    """Implementazione dell'algoritmo k-NN per problemi di classificazione binaria.
       In input richiede:
        . il dadataset, oggetto di classe DataSet
        . il numero di vicini da considerare
       In output restituisce il predittore.
       Nota: l'output del predittore è calcolato come la media
             del target dei k elementi piu' vicini nel dataset.
             Quindi l'output e' la classe piu' frequente tra i k vicini."""
    def predict(example):
        """Cerca i k elementi più vicini ad example e calcola la media
           dell'output."""
        # heapq.nsmallest restituisce una lista degli 'n' elementi
        # piu' vicini ad example, usando come distanza
        # la funzione dataset.distance, che di default e'
        # la distanza Euclidea.
        neighbors = heapq.nsmallest(k,
                                    ((dataset.distance(e,
                                      example), e) for e in dataset.examples))

        # restituisce il valore target maggiormente presente in neighbors
        # discretizzando la media dei valori degli elementi piu' vicini
        return (mean(e[dataset.target] for (d, e) in neighbors)>0.5)
    return predict

def NearestNeighborClassifier_cat(dataset, k=1):
    """Implementazione dell'algoritmo k-NN per problemi di classificazione con etichetta
       categorica.
       In input richiede:
        . il dadataset, oggetto di classe DataSet
        . il numero di vicini da considerare
       In output restituisce il predittore.
       Nota: l'output del predittore è calcolato come la moda
             del target dei k elementi piu' vicini nel dataset.
             Quindi l'output e' l'etichetta (categorica)
             della classe piu' frequente tra i k vicini."""

    def predict(example):
        """Cerca i k elementi più vicini ad example e applica un majority vote
           sull'output."""
        # heapq.nsmallest restituisce una lista degli 'n' elementi
        # piu' vicini ad example, usando come distanza
        # la funzione dataset.distance, che di default e'
        # la distanza Euclidea.
```

```
        neighbors = heapq.nsmallest(k, ((dataset.distance(e, example),
                                         e) for e in dataset.examples))

        # mode restituisce il valore target maggiormente presente in neighbors
        return mode(e[dataset.target] for (d, e) in neighbors)
    return predict

def NearestNeighborRegressor(dataset, k=1):
    """Implementazione dell'algoritmo k-NN per problemi di regressione.
       In input richiede:
        . il dadataset, oggetto di classe DataSet
        . il numero di vicini da considerare
       In output restituisce il predittore.
       Nota: l'output del predittore è calcolato come la media
             del target dei k elementi piu' vicini nel dataset.."""
    def predict(example):
        """Cerca i k elementi più vicini ad example e calcola la media
           dell'output."""
        # heapq.nsmallest restituisce una lista degli 'n' elementi
        # piu' vicini ad example, usando come distanza
        # la funzione dataset.distance, che di default e'
        # la distanza Euclidea.
        neighbors = heapq.nsmallest(k, ((dataset.distance(e, example), e)
                                        for e in dataset.examples))

        # calcola l'output come la media dell'output dei vicini
        return mean(e[dataset.target] for (d, e) in neighbors)
    return predict
```

Tabella 5 Implementazioni dell'algoritmo k-nearest neighbor.