Machine Learning

Università Roma Tre
Dipartimento di Ingegneria

Anno Accademico 2021 - 2022

Esercitazione: Classificatore Bayesiano (Ex 13)

Sommario

- ...

Scikit-learn: Classificatori Naive Bayes

- Un approccio di classificazione molto veloce nell'addestramento, che non richiede che il training set sia caricato interamente in memoria, anche se a volte mostrano performance peggiori rispetto agli approcci lineare (es. LogisticRegression e LinearSVC).
- *Naive* perché basato sull'assunzione che le feature siano indipendenti dal punto di vista statistico, spesso inesatta.
  - Es. un problema cardiovascolare può dipendere dal colesterolo, peso, livelli di diabete, etc; se presenti contemporaneamente possono aumentarne il rischio, ma l'approccio naive le valuta singolarmente.
- Si ricavano i parametri del modello analizzando le features singolarmente, e collezionando statistiche per ogni feature per ogni classe.
- Ricavare la classe più verosimile (con più alta probabilità *a posteriori*) si ottiene mediante il *Teorema di Bayes*.
  - L'approccio naive (indipendenza tra features) ci porta a non interpretare la probabilità in output poiché risulta essere una approssimazione troppo grossolana rispetto a quella reale.

Classificatori Naive Bayes: pregi e difetti

- Semplice implementazione (basata sulle occorrenze)
- Può funzionare anche su dataset piccoli
- È veloce e richiede poca memoria
- Gestiste il caso di valori mancanti nei dati
- Poco sensibile a dati rumorosi

- L'assunzione dell'indipendenza statistica è raramente soddisfatta; il modello non considera le dipendenze tra features
- I dati nel continuo devono essere spesso rielaborati (es. binning)
- Non raggiunge prestazioni ottimali rispetto ad altri approcci
- Non supporta l'*online learning*: occorre riaddestrare il modello in presenza di nuovi dati.
- Non funziona correttamente se i dati nel test set non sono presenti nel training.

Scikit-learn: Classificatori Naive Bayes

- Ci sono vari classificatori implementati in Scikit-learn:
  - GaussianNB: adatto a dati nel continuo
  - CategoricalNB: features discrete distribuite su categorie predefinite
  - BernoulliNB: assume dati binari
  - MultinomialNB: assume feature che accumulano valori (es. frequenza)
  - ComplementNB: variazione del Multinomial per correggere alcune assunzioni sui dati.
- BernoulliNB e MultinomialNB sono spesso usati per dati testuali.
  - Per dataset di training molto grandi e sparsi si può usare il parametro *partial_fit* che riduce la richiesta di memoria.
- È una valida alternativa a *logistic regression* e *decision trees*.

Scikit-learn: BernoulliNB

- Conteggia quante volte una feature non è pari 0 per ogni classe.
- Ad esempi, 4 istanze con 4 feature binarie ciascuna. La 1a e 3a istanza hanno classe '0', mentre la 2a e 4a hanno classe '1'.

```
X = np.array([[0, 1, 0, 1],
[1, 0, 1, 1],
[0, 0, 0, 1],
[1, 0, 1, 0]])
y = np.array([0, 1, 0, 1])
```

- Effettuando il conteggio per entrambe le classi si ha:

```
counts = {}
for label in np.unique(y):
    # iterate over each class
    # count (sum) entries of 1 per feature
    counts[label] = X[y == label].sum(axis=0)
print("Feature counts:\n{}".format(counts))
```

```
Feature counts:
{0: array([0, 1, 0, 2]), 1: array([2, 0, 2, 1])}
```

Scikit-learn: MultinomialNB e GaussianNB

- *MultinomialNB* tiene conto del valor medio per ogni feature per ogni classe. *GaussianNB* ricava valor medio e varianza.
- La predizione su una istanza è ricavata valutando tutte le classi e scegliendo quella ottimale.
- MultinomialNB e BernoulliNB hanno un singolo parametro *alpha*, che determina la complessità del modello. Ai dati sono aggiunti *alpha* istanze virtuali che hanno valori positivi per tutte le features. Questo genera uno "smoothing" sulle statistiche calcolate.
  - Valori elevati di *alpha* creano smoothing elevati e modelli meno complessi.
- *GaussianNB* è più adatto a dataset con molte features. *MultinomialNB* è migliore rispetto a *BernoulliNB* con dataset con un numero elevato di features diverse da 0 (es. grandi documenti testuali).

Naive Bayes classifier da zero

- Proviamo a fare l'implementazione del classificatore
  - Step 1: Separate By Class.
  - Step 2: Summarize Dataset.
  - Step 3: Summarize Data By Class.
  - Step 4: Gaussian Probability Density Function.
  - Step 5: Class Probabilities

- Immaginiamo di impiegare il dataset *Iris*:
  - lunghezza e larghezza sepalo (reali)
  - lunghezza e larghezza petalo (reali)
  - classe di appartenenza = {Iris-setosa, Iris-versicolor, Iris-virginica}

Naive Bayes classifier: step 1

- Calcoliamo la probabilità di appartenenza di una istanza ad una certa classe.
- Separiamo i dati in ingresso in base alla classe di appartenenza.

```
# Split the dataset by class values
# Restituisce un dizionario classe -> lista di istanze
# Funziona per ogni dataset il cui ultimo valore è la classe di appartenenza
def separate_by_class(dataset):
    separated = dict()
    for i in range(len(dataset)):
        vector = dataset[i]
        class_value = vector[-1]   # ultimo valore
        if (class_value not in separated):
            separated[class_value] = list()
        separated[class_value].append(vector)
    return separated
```

Naive Bayes classifier: step 1

```
# Iris dataset
dataset = [[3.393533211,2.331273381,0],
    [3.110073483,1.781539638,0],
    [1.343808831,3.368360954,0],
    [3.582294042,4.67917911,0],
    [2.280362439,2.866990263,0],
    [7.423436942,4.696522875,1],
    [5.745051997,3.533989803,1],
    [9.172168622,2.511101045,1],
    [7.792783481,3.424088941,1],
    [7.939820817,0.791637231,1]]
separated = separate_by_class(dataset)
for label in separated:
    print(label)
    for row in separated[label]:
        print(row)
```

```
0
[3.393533211, 2.331273381, 0]
[3.110073483, 1.781539638, 0]
[1.343808831, 3.368360954, 0]
[3.582294042, 4.67917911, 0]
[2.280362439, 2.866990263, 0]
1
[7.423436942, 4.696522875, 1]
[5.745051997, 3.533989803, 1]
[9.172168622, 2.511101045, 1]
[7.792783481, 3.424088941, 1]
[7.939820817, 0.791637231, 1]
```

Naive Bayes classifier: step 2

- Per ogni dataset ricaviamo 2 statistiche: media e deviazione standard.
  - La media può essere ricavata così: $\mu = sum(x)/n * count(x)$

dove x è la lista dei valori (o colonna) sui cui stiamo stimando la media.

```
# Calculate the mean of a list of numbers
def mean(numbers):
    return sum(numbers)/float(len(numbers))
```

- Per la deviazione standard σ si ha: $sqrt(\Sigma_i(x_i - \mu(x))^2 / N-1)$

```
from math import sqrt

# Calculate the standard deviation of a list of numbers
def stdev(numbers):
    avg = mean(numbers)
    variance = sum([(x-avg)**2 for x in numbers]) / float(len(numbers)-1)
    return sqrt(variance)
```

- Media e deviazione standard devono essere calcolate per ogni feature e considerando tutte le istanze.

Naive Bayes classifier: step 2

- Media e deviazione standard devono essere calcolate per ogni feature e considerando tutte le istanze.
- La funzione *zip(*...)* separa le colonne del dataset e restituisce una tupla per ogni colonna contenente i relativi valori delle features.

```
def summarize_dataset(dataset):
    summaries=[(mean(column),stdev(column),len(column)) for column in zip(*dataset)]
    del(summaries[-1])
    return summaries
```

- Ad esempio:

```
dataset = [[3.393533211,2.331273381,0],
    [3.110073483,1.781539638,0],
    [1.343808831,3.368360954,0],
    [3.582294042,4.67917911,0],
    [2.280362439,2.866990263,0],
    [7.423436942,4.696522875,1],
    [5.745051997,3.533989803,1],
    [9.172168622,2.511101045,1],
    [7.792783481,3.424088941,1],
    [7.939820817,0.791637231,1]]
summary = summarize_dataset(dataset)
print(summary)
```

```
> [(5.178333386499999, 2.7665845055177263, 10), (2.9984683241, 1.218556343617447, 10)]
```

Naive Bayes classifier: step 3

- Vogliamo ricavare le statistiche per ogni classe (o label). Sfruttiamo la funzione *separate_by_class()* definita in precedenza:

```
def summarize_by_class(dataset):
    separated = separate_by_class(dataset)
    summaries = dict()
    for class_value, rows in separated.items():
        summaries[class_value] = summarize_dataset(rows)
    return summaries
```

- Ad esempio:

```
dataset = [[3.393533211,2.331273381,0],
    [3.110073483,1.781539638,0],
    [1.343808831,3.368360954,0],
    [3.582294042,4.67917911,0],
    [2.280362439,2.866990263,0],
    [7.423436942,4.696522875,1],
    [5.745051997,3.533989803,1],
    [9.172168622,2.511101045,1],
    [7.792783481,3.424088941,1],
    [7.939820817,0.791637231,1]]
separated = separate_by_class(dataset)
for label in separated:
    print(label)
    for row in separated[label]:
        print(row)
```

```
>>>
0
(2.7420144012, 0.9265683289298018, 5)
(3.0054686692, 1.1073295894898725, 5)
1
(7.6146523718, 1.2344321550313704, 5)
(2.9914679790000003, 1.4541931384601618, 5)
```

Naive Bayes classifier: step 4

- Assumiamo che la probabilità che un certo valore *x* osservato sia funzione da una distribuzione gaussiana, descritta interamente dai due valori: media e deviazione standard.

- La funzione di densità di probabilità sarà così ricavata (vedi lezione; la y corrisponde alla media):

$$p(x_i|y_j) = \frac{1}{\sqrt{2\pi\sigma_j^2}} e^{-\frac{(x_i-\mu_j)^2}{2\sigma_j^2}}$$

f(x) = (1 / sqrt(2 * PI) * Σ) * exp(-((x-*μ*)^2 / (2 * Σ^2)))

- Dove Σ è la matrice di covarianza (con d =1 coincide con la varianza).

- Esercizio: definire la funzione *calculate_probability(x, mean, stdev)* per il calcolo della densità di probabilità.

Naive Bayes classifier: step 4

- Esercizio: definire la funzione calculate_probability(x, mean, stdev) per il calcolo della densità di probabilità.

```
from math import sqrt
from math import pi
from math import exp

def calculate_probability(x, mean, stdev):
    exponent = exp(-((x-mean)**2 / (2 * stdev**2 )))
    return (1 / (sqrt(2 * pi) * stdev)) * exponent
```

```
print(calculate_probability(1.0, 1.0, 1.0))
print(calculate_probability(2.0, 1.0, 1.0))
print(calculate_probability(0.0, 1.0, 1.0))
```

```
> 0.398942280401432
> 0.24197072451914337
> 0.24197072451914337
```

- Notare come per x=1, e media e varianza pari a 1, l'apice della campana assume valore 0.39. Per x=2 e x=0, e medesime statistiche, il valore è 0.24.

Naive Bayes classifier: step 5

- Ora impieghiamo le statistiche ricavate dal training data per nuovi dati. La stima delle probabilità viene stimata per ogni classe.
  - P(class|data) = P(X|class) * P(class)
- Attenzione: Avendo eliminato la frazione, il risultato non è strettamente una probabilità.
- Vogliamo massimizzare tale valore, ovvero prendere la classe con valore di probabilità massimo.
- L'approccio naive implica l'indipendenza, es:
  - P(class=0|X1,X2) = P(X1|class=0) * P(X2|class=0) * P(class=0)

Naive Bayes classifier: step 5

- Esercizio: definire *calculate_class_probabilities()* che prende in input le statistiche restituite da *summarize_by_class()* e valuta la probabilità per una certa istanza data sempre in input.

- Esempio:

```
# Test calculating class probabilities
dataset = [[3.393533211,2.331273381,0],
  [3.110073483,1.781539638,0],
  [1.343808831,3.368360954,0],
  [3.582294042,4.67917911,0],
  [2.280362439,2.866990263,0],
  [7.423436942,4.696522875,1],
  [5.745051997,3.533989803,1],
  [9.172168622,2.511101045,1],
  [7.792783481,3.424088941,1],
  [7.939820817,0.791637231,1]]
summaries = summarize_by_class(dataset)
probabilities = calculate_class_probabilities(summaries, dataset[0])
print(probabilities)

> {0: 0.05032427673372075, 1: 0.0001155771837994576 5}
```

Naive Bayes classifier: step 5

- Esercizio: definire *calculate_class_probabilities()* che prende in input le statistiche restituite da *summarize_by_class()* e valuta la probabilità per una certa istanza data sempre in input.
  - Calcola il numero totale di istanze a partire dalle statistiche passate come parametro.
  - Valuta il valore P(class) come frazione tra il numero di istanze per una classe e il numero di istanze nel dataset
  - Stima la probabilità per ogni valore in input impiegando la funzione densità di probabilità, e le statistiche per ogni colonna associata ad una certa classe. Le probabilità saranno moltiplicate se associate alla stessa classe.
  - Il processo sarà ripetuto per ogni classe nel dataset.
  - Restituire un dizionario classe->probabilità

Naive Bayes classifier: step 5

- Esercizio: definire *calculate_class_probabilities()* che prende in input le statistiche restituite da *summarize_by_class()* e valuta la probabilità per una certa istanza data sempre in input.

```
def calculate_class_probabilities(summaries, row):
  # numero totale di istanze di training
  total_rows = sum([summaries[label][0][2] for label in summaries])
  # output
  probabilities = dict()
  # per ogni chiave (classe) e valore (istanze di quella classe)
  for class_value, class_summaries in summaries.items():
     # probabilità calcolata in base alle frequenze
     probabilities[class_value] = summaries[class_value][0][2]/float(total_rows)
     # per ogni istanza in summaries associata ad una classe
     for i in range(len(class_summaries)):
        # ricava le statistiche di quella classe
        mean, stdev, count = class_summaries[i]
        # aggiorna la probabilità per quella classe
        probabilities[class_value] *= calculate_probability(row[i], mean, stdev)
  return probabilities
```

Naive Bayes classifier: esercitazione

- Considerare il dataset Kaggle Adult income dataset:
  - https://www.kaggle.com/datasets/wenruliu/adult-income-dataset
  - http://www.cs.toronto.edu/~delve/data/adult/adultDetail.html
- Contiene 16 colonne:
  - Target filed: Income
    - -- The income is divide into two classes: <=50K and >50K
  - Number of attributes: 14
    - -- These are the demographics and other features to describe a person
- Analizza il dataset passo passo seguendo le considerazioni su:
  - https://www.kaggle.com/code/prashant111/naive-bayes-classifier-in-python/notebook
- Applica l'algoritmo Naive Bayes classifier per i suddetto dataset.
- Nota: alcuni attributi potrebbero dover essere normalizzati oppure convertiti in valori numerici.

Naive Bayes classifier: esercitazione

- Alcune funzioni di supporto:

```
# Load a CSV file
def load_csv(filename):
  dataset = list()
  with open(filename, 'r') as file:
    csv_reader = reader(file)
    for row in csv_reader:
      if not row:
        continue
      dataset.append(row)
  return dataset
```

```
# Convert string column to float
def str_column_to_float(dataset, column):
  for row in dataset:
    row[column] = float(row[column].strip())

# Convert string column to integer
def str_column_to_int(dataset, column):
  class_values = [row[column] for row in dataset]
  unique = set(class_values)
  lookup = dict()
  for i, value in enumerate(unique):
    lookup[value] = i
  for row in dataset:
    row[column] = lookup[row[column]]
  return lookup
```

Testi di Riferimento

- Andreas C. Müller, Sarah Guido. *Introduction to Machine Learning with Python: A Guide for Data Scientists*. O'Reilly Media 2016
- Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media 2017
- Tutorial https://machinelearningmastery.com/naive-bayes-classifier-scratch-python/
- Dataset:
  - https://www.kaggle.com/datasets/wenruliu/adult-income-dataset
  - http://www.cs.toronto.edu/~delve/data/adult/adultDetail.html