Наивный Байесовский классификатор в 25 строк кода

* [Python](https://habrahabr.ru/hub/python/)

Наивный Байесовский классификатор один из самых простых из алгоритмов классификации. Тем не менее, очень часто он работает не хуже, а то и лучше более сложных алгоритмов. Здесь я хочу поделиться кодом и описанием того, как это все работает.   
  
И так, для примера возьму задачу определения пола по имени. Конечно, чтобы определить пол можно создать большой список имен с метками пола. Но этот список в любом случае будет неполон. Для того чтобы решить эту проблему, можно «натренировать» модель по маркированным именам.   
Если интересует, прошу .

Немного теории

Пусть у нас есть строка текста O. Кроме того, имеются классы С, к одному из которых мы должны отнести строку. Нам необходимо найти такой класс с, при котором его вероятность для данной строки была бы максимальна. Математически это записывается так:  
  
image  
  
Вычислить P(C|O) сложно. Но можно воспользоваться теоремой Байеса и перейти к косвенным вероятностям:  
  
image  
  
Так как мы ищем максимум от функции, то знаменатель нас не интересует (он в данном случае константа). Кроме того, нужно взглянуть на строку O. Обычно, нет смысла работать со всей строкой. Намного эффективней выделить из нее определенные признаки (features). Таким образом формула примет вид:  
  
image  
  
Знаменатель нас не интересует. Числитель же можно переписать так.   
  
  
  
Но это опять сложно. Здесь включаем «наивное» предположение о том, что переменные O зависят только от класса C, и не зависят друг от друга. Это сильно упрощение, но зачастую это работает. Числитель примет вид:  
  
image  
  
Финальная формула примет вид:  
  
image (1)  
  
Т.е. все что нужно сделать, это вычислить вероятности P( C ) и P(O|C). Вычисление этих параметров и называется тренировкой классификатора.

Код

Ниже — код на питоне. Содержит всего две функции: одна для тренировки (подсчета параметров формулы), другая для классификации (непосредственный расчет формулы).

from \_\_future\_\_ import division

from collections import defaultdict

from math import log

def train(samples):

classes, freq = defaultdict(lambda:0), defaultdict(lambda:0)

for feats, label in samples:

classes[label] += 1 *# count classes frequencies*

for feat in feats:

freq[label, feat] += 1 *# count features frequencies*

for label, feat in freq: *# normalize features frequencies*

freq[label, feat] /= classes[label]

for c in classes: *# normalize classes frequencies*

classes[c] /= len(samples)

return classes, freq *# return P(C) and P(O|C)*

def classify(classifier, feats):

classes, prob = classifier

return min(classes.keys(), *# calculate argmin(-log(C|O))*

key = lambda cl: -log(classes[cl]) + \

sum(-log(prob.get((cl,feat), 10\*\*(-7))) for feat in feats))

В функции *train* первые пять строк производят подсчет количества классов C, а также частоту появления фич O и С в одном семпле. Вторая часть метода просто нормирует эти частоты. Таким образом на выходе получаются вероятности P© и P(O|C).  
  
В функции *classify* происходит поиск наиболее вероятного класса. Единственное отличие от формулы (1) в том, что я заменяю произведение вероятностей на сумму логарифмов, взятых с отрицательным знаком, и вычисляю не argmax, а argmin. Переход к логарифмам — распространненный прием чтобы избежать слишком маленьких чисел, которые могли бы получится при произведении вероятностей.  
Число 10(^-7), которое подставляется в логарифм, это способ избежать нуля в аргументе логарифма (т.к. он будет иначе он будет неопределен).   
  
Чтобы натренировать классификатор возьмем размеченный список мужских и женских имен и воспользуемся этим кодом:

def get\_features(sample): return (sample[-1],) *# get last letter*

samples = (line.decode('utf-8').split() for line in open('names.txt'))

features = [(get\_features(feat), label) for feat, label in samples]

classifier = train(features)

print 'gender: ', classify(classifier, get\_features(u'Аглафья'))

Файл 'names.txt' можно скачать [здесь](http://dl.dropbox.com/u/27900527/names.txt).  
  
В качестве фич я выбрал последнюю букву имени (см функцию get\_features). Работает неплохо, но для рабочего варианта лучше использовать схему посложнее. К примеру, выбрать первую букву имени и две последних. К примеру, вот так:

def get\_features(sample): return (

'll: %s' % sample[-1], *# get last letter*

'fl: %s' % sample[0], *# get first letter*

'sl: %s' % sample[1], *# get second letter*

)

Алгоритм можно использовать для произвольного числа классов. К примеру, можно попробовать построить классификатор текстов по эмоциональной окраске. 

Тесты

Я протестировал классификатор на части исходного корпуса с именами. Точность составила 96%. Это не блестящий результат, но для многих задач вполне достаточно.