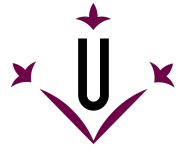
Naive Bayes Classifiers

Ramon Béjar Torres

Data mining - Master on Computer Science



Universitat
de Lleida

We present in this unit one of the most basic, and yet powerfull and efficient, machine learning systems for building automatic classifiers for different tasks, when we have a set of *multivariate objects* sampled (obtained) from some data source, and we assume that they follow certain (unknown) multivariate distribution. The system we want to learn is a classifier:

- As input, we have one object from that data source for which we know only the values for *some of its variables* (that we call the observable attributes)
- The goal is to infer the value of some other of its variables.

To build such system, we will follow a **supervised machine learning** approach, so we assume that we have available a data set of objects from that data source with **all** the variable values known.

The basic tool we use to infer the value for a unknown variable from a set of known variables is the Bayes theorem. Consider first a distribution with two variables A and B. If $P(A, B)$ is the joint probability distribution (probability of obtaining particular values for A and B for an object sampled from the distribution), we can factorize that distribution using conditional probability in two different ways:

$$P(A, B) = P(A|B)P(B) = P(B|A)P(A)$$

So, if we **do not know** $P(A, B)$, but we know one of these two factorizations, we can make questions about conditional probabilities in the following way. Suppose that for an object we know the value of B, and we know (**or we have learned estimations of**) the factors $P(B|A)$, $P(A)$ and $P(B)$. Then, we have that:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

But this same basic principle is equally valid if B is not a single variable, but a larger set of known variables from the object: $E_1, E_2, \ldots, E_n$ (that we also call the **evidence**), and A could also be a subset of unknown variables, although in typical Naive Bayesian models there is only one unknown variable A. So, we want to compute:

$$P(A = a_j | E_1, E_2, \ldots, E_n) = \frac{P(E_1, E_2, \ldots, E_n | A = a_j) P(A = a_j)}{P(E_1, E_2, \ldots, E_n)}$$

Observe that for each possible value a_j for A, in all the expressions the denominator will be equal. Actually, from probability theory we have that the denominator is equal to the sum of all the numerators with different values for A:

$$P(E_1, E_2, \ldots, E_n) = \sum_{a_j} P(E_1, E_2, \ldots, E_n | A = a_j) P(A = a_j)$$

So, if we compute the numerator for all the possible values for A, we do not need to explicitely compute the denominator, to know which value for A is more probable.

So, let's turn our attention to the numerator, and let's say that our target query is *proportional* to the numerator, given that all the denominators will be equal for all the possible values of A:

$$P(A|E_1, E_2, \ldots, E_n) \propto P(E_1, E_2, \ldots, E_n|A)P(A)$$

Now, in that factorization, to compute $P(E_1, E_2, \ldots, E_n|A)$, is when we are going to use one more assumption in our model (and from this assumption comes the name *naive* for our class of bayes models). If we assume that all the E_i variables are independent between them (but they depend on the value of A), we can factorize that expression as product of simpler conditional probability expressions:

$$P(A|E_1, E_2, \ldots, E_n) \propto P(E_1|A)P(E_2|A)\ldots P(E_n|A)P(A)$$

Then, if we can learn (or estimate) the values of such factors, we can build a classifier that given the known attributes E_i from an object, infers the most probable value for the unknown variable A. How do we learn such factors ? Assume we have a data set of m objects from the data source with all the variable values known:

$$\begin{matrix} A_{11} & E_{11} & E_{12} & \dots & E_{1n} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ A_{m1} & E_{m1} & E_{m2} & \dots & E_{mn} \end{matrix}$$

From this data set, we can estimate any factor $P(E_j|A)$ and $P(A)$, although the exact expressions depend on some assumptions about the class of random variables that best represent their behaviour.

But the important point is that they can efficiently and *independently* be estimated computing some statistics from our sample data set of m objects. That is:

- Estimate the probability $P(E_i|A = a_j)$ (for any possible value for attribute E_i and value a_j for A) from the data set of m objects
- Estimate the probability $P(A = a_j)$ also from the data set of m objects.

This makes naive bayes classifiers the most efficient class of probabilistic models to be trained in a distributed computation environment.

Finally, if we want to compute the exact probabilities for each $P(A = a_j|E)$, we can do it using the expression:

$$P(A = a_j|E) = \frac{P(E_1|A = a_j) \dots P(E_n|A = a_j)P(A = a_j)}{\sum_{a_{j'}} = P(E_1|A = a_{j'}) \dots P(E_n|A = a_{j'})P(A = a_{j'})}$$

That is, sum up all the above mentioned numerators expressions to have the value of the denominator.

Naive Bayes Models in Spark

Let's begin with a very simple example, where the goal is to learn a bayes model to predict the value for the first variable in each object from the known value of the other three variables.

In [3]:
```
sampledata = [ [0,1, 0, 0], [0,2, 0, 0], [0,3, 0, 0], [0,4, 0, 0],
               [1,0, 2, 0], [1,0, 3, 0], [1,0, 4, 0], [1,0, 5, 0],
               [2,0, 1, 1], [2,0, 1, 2], [2,0, 0, 3], [2,0, 0, 4]  ]
```

Before proceeding to learn a model for querying the first variable from the others, we have to think about the class of random variables we assume they follow. The spark.mllib library supports two kinds of random variables for Naive Bayes models:

- *Bernoulli* variables. Appropiate if our variables are indicator variables: domain {0,1}.
- *Multinomial* variables. Appropiate if our variables have a bigger range of values: {0, 1, ... , n}

By default, spark assumes multinomial variables, but we can change it when we train our naive bayes model.

To learn a naive bayes model, like happens with other machine learning models in spark, we have to transform our data as a set of LabeledPoint objects, where the label represents the variable for which we want to learn a prediction model and the rest of the variables represent the attributes (features) of the labeled point.

In [4]:
```
bayes1RDD = sc.parallelize( sampledata ).\
         map( lambda sample :  LabeledPoint(sample[0], Vectors.dense( sample[1
:] ) ) )
modelbayes1 = NaiveBayes.train(bayes1RDD)

# Make prediction and test accuracy on the training set.
predictionAndLabel = bayes1RDD.map(lambda p: (modelbayes1.predict(p.features),
p.label))
accuracy = 1.0 * predictionAndLabel.filter(lambda pandl: pandl[0] == pandl[1]).
count() / bayes1RDD.count()
print ( "Accuracy of the model obtained on training data : ", accuracy, "\n")

# Let's query (predict) the most probable value for the first variable in a new
instance:
tobj = Vectors.dense([0,1,4])
print ("Most Probable value for ", tobj, " : ", modelbayes1.predict(tobj))
```

```
Accuracy of the model obtained on training data :  1.0

Most Probable value for  [0.0,1.0,4.0]  :  2.0
```

In the naive bayes model learned, the information is stored in the following members:

- labels – list of labels.
- pi – log of class prior probabilities, whose dimension is C, number of labels.
- theta – log of class conditional probabilities, a matrix of C rows and D columns, where D is the number of features.

We can access to this information to make any other operations with the model, apart of predicting target variable values with the predict() function.

In particular, for the case of mutinomial variables, these parameters give us the following information:

1. labels: the set of different values for the target variable
2. pi: Is the set of (logarithm of) probabilities : $[log(P(C_k)) \mid k \in C]$
3. theta: Is the set of (logarithm of) conditional probabilities: $[\theta_{k,j} = log(P(x_j = 1|C_k)) \mid k \in C, j \in D]$. Then, with these parameters the log probability $log(P(x_j = m \mid C_k))$ is equal to $m\ \theta_{k,j}$ given that:
$$P(x_j = m|C_k) = P(x_j = 1|C_k)^m$$
in a multinomial random variable

All these probabilities are estimations computed from our data set of points.

Starting in spark 3.0, naive bayes models, when using spark data frames API, can also use Gaussian variables (so for continuous domains). In that case, the estimations of conditional probabilities are based on parameter estimations for Gaussian variables:

$$P(x_j = v \mid C_k) = \frac{1}{\sqrt{2\pi\sigma_{k,j}^2}} e^{-\frac{(v-\mu_{k,j})^2}{2\sigma_{k,j}^2}}$$

where $\mu_{k,j}$ and $\sigma_{k,j}$ represent the mean and the standard deviation of the gaussian variable associated with attribute j and when the class label is k.

Text Classification with Naive Bayes Classifiers

Next, we are going to turn our attention on the topic of text classification. This is a basic ingredient, for example, in tradicional SPAM classifiers, or for example in news aggregation systems that need to filter huge numbers of news artices by the topics of interest for particular users.

It turns out that the Naive Bayes model can give quite good results in many settings, although nowadays models based on Support Vector Machines show better accuracy in such learning tasks (and in many others). However, as we have said, training a Naive Bayes model can be done quite efficiently, so Naive Bayes models are still a good choice if we have to rebuild our learned model as new data is available.

In [5]:
```
#
# Meaning of labels:
#
#   label 0.0 :  soccer  text
#   label 1.0 :  politics  text
#   label 2.0 :  cinema    text
#
training_rawdocs = sc.parallelize([
    {"text": "We will give our best and we have a lot of games - the Europa Lea
gue and Premier League and cups - and I think as time goes , we will be better
 and better .", "label": 0.0},
    {"text": "Spanish league has very spensive soccer players . Messi is the be
st . Ronaldo sucks .", "label": 0.0},
    {"text": "Champions league where the best european players can be found . S
occer matches can be dangerous for referees .", "label": 0.0},
    {"text": "Spanish Soccer is all about Barcelona and Madrid ", "label" : 0.0
},
    {"text": "Spanish Soccer is all about politics .", "label": 1.0},
    {"text": "Every spanish politician loves some spanish soccer team", "label"
: 1.0},
    {"text": "Spanish political parties can be divided in left , centre and rig
ht parties .", "label": 1.0},
    {"text": "The political power in Spain resides in Madrid .", "label": 1.0},
    {"text": "Woody Allen movies are not for every possible spectator , like yo
u can say about Arnold Schwarzenegger movies .", "label": 2.0},
    {"text": "Star Wars and Disney is a weird union but it is not George Lucas
 business anymore .", "label": 2.0},
    {"text": "Nouvelle vague movies can be quite hard to watch .", "label": 2.0
},
    {"text": "François Truffaut made a movie about the novel Fahrenheit 451 .",
"label": 2.0}])
```

A very basic feature model for text classification

For starters, let's consider a very basic model for text classification. From each text, we will extract the set of different words it contains, and assign a different identifier to each different word from our entire set of *corpus* documents (the set of documents we have correctly classified and we want to use them for learning a good classifier).

In [6]:
```
# First, compute set of diferent words as the feature set
#
docstofeaturesetRDD = training_rawdocs.map( lambda doc: \
                         { "words": list(set(doc["text"].split())), "label" : do
c["label"] } )
docstofeaturesetRDD.cache()
print ("First doc: ", docstofeaturesetRDD.take(1)[0])

# Beware, the EXPENSIVE (shuffle) operation comes here : distinct() !
dictionaryRDD = docstofeaturesetRDD.flatMap( lambda doc : doc["words"] ).distin
ct()

worddict = dictionaryRDD.collect()
sizedict = len(worddict)
print (" Dictionary size: ",sizedict, "\n")
```

```
First doc:  {'words': ['we', 'better', '-', 'our', 'goes', 'the', ',', 'of',
'as', '.', 'games', 'League', 'Premier', 'have', 'and', 'cups', 'I', 'a', 'giv
e', 'Europa', 'We', 'best', 'time', 'think', 'lot', 'be', 'will'], 'label': 0.
0}
 Dictionary size:  107
```

In [7]:
```
# Then, map each document to Labelled points with the feature set vector
#
# Observe that the parameter wdict represents the whole set of distinct words
# This will be HIGHLY inefficient if the size of wdict is too big, and it makes
# problems when distributing it as a parameter trough the workers of the
# computation cluster used by Spark
def mapDocsToLabelledPoints(seqofDocs,wdict,sizedict):
    for DocFeatureSet in seqofDocs:
        features = list()
        values = list()
        for w in DocFeatureSet["words"]:
            features.append(wdict.index(w))
            values.append(1.0)
        yield  LabeledPoint( DocFeatureSet["label"] ,\
                             Vectors.sparse(sizedict, sorted(features), values )
)
```

In [8]:
```
docsToLabelledPoints = docstofeaturesetRDD.mapPartitions( lambda seqofDocs : \
                        mapDocsToLabelledPoints(seqofDocs,worddict,sizedict) )

print (docsToLabelledPoints.take(1)[0])
```

```
(0.0,(107,[0,1,12,13,25,26,27,28,40,41,42,43,44,45,51,52,53,54,55,69,70,71,72,
84,94,95,96],[1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,
1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0]))
```

We use sparse vectors to represent the documents, as we assume that any document will contain only a small fraction of the total set of distinct words

Let's train a Naive Bayes model with our basic feature model for our data set of documents.

In [9]:
```
model = NaiveBayes.train(docsToLabelledPoints)
# Pair predicted and actual document labels
labels_and_preds = docsToLabelledPoints.map(lambda p: \
                        {  "predicted" : model.predict(p.features), "actual" : p
.label } )
labels_and_preds.cache()
for landp in labels_and_preds.collect():
  print (landp)
print ("\n")

accuracy = 1.0 * labels_and_preds.filter(lambda x : x['predicted'] == x['actua
l']).count() / labels_and_preds.count()
print (" Accuracy (fraction of correctly classified documents) : ", accuracy)
```

```
{'predicted': 0.0, 'actual': 0.0}
{'predicted': 0.0, 'actual': 0.0}
{'predicted': 0.0, 'actual': 0.0}
{'predicted': 0.0, 'actual': 0.0}
{'predicted': 1.0, 'actual': 1.0}
{'predicted': 1.0, 'actual': 1.0}
{'predicted': 1.0, 'actual': 1.0}
{'predicted': 1.0, 'actual': 1.0}
{'predicted': 2.0, 'actual': 2.0}
{'predicted': 2.0, 'actual': 2.0}
{'predicted': 2.0, 'actual': 2.0}
{'predicted': 2.0, 'actual': 2.0}


 Accuracy (fraction of correctly classified documents) :  1.0
```

Extracting term frequency information and hashing for fast feature extraction

The TF-IDF score for words

The previous feature model to represent text documents can be good enough in some domains, but the fact that gives the same relevance to each single word appearing in a document can be somehow missleading in some cases.

For example, some words can appear with high frequency in many documents, like articles and prepositions, but this does not mean that they provide any useful information about the topic of the document. By contrast, text documents talking about football will probably include many ocurrences of words like football, ball and player, much more frequently than in other documents of different classes.

So, it seems that it makes sense to include the information about how frequent is a word with respect to the relative frequency of the world in the entire document set. One common approach to measure that information is the TF-IDF word score. This score value is computed as the product of two values. The term frequency (TF) of the word i in the document j (but relative to the maximum frequency of any word in the document) is computed as:

$$TF(i,j) = \frac{f_{i,j}}{max_k\ f_{k,j}}$$

The inverse document frequency (IDF), that is the inverse of the frequency of documents that contain the word, but in logarithmic scale, is computed as:

$$IDF(i,N) = \log_2(N/n_i)$$

So, given a word i and a document j from our corpus of N documents, the product:

$$TF-IDF(i,j,N) = TF(i,j) \times IDF(i,N)$$

will be high for word i, document j if:

1. Word i appears frequently in document j
2. But word i appears in few documents (n_i is small)

In text classification we may also substract the words that are commonly found in ANY class of documents (like articles and prepositions) before measuring the really relevant words. These common words are called stop words. However, observe that for some classes of documents an unusual high frequency of certain stop words could be signaling some specific feature about the style of the document.

Hashing words for efficient mapping of words to feature indexes

In order to improve the performance of our naive bayes model for text classification, there is another aspect we could improve. Observe that we work with a collection of base words (that in our case have been extracted from the corpus of documents), and that this base set is indexed to assign to each different word a different index value, that will be used to represent documents as feature vectors.

But the problem is that the time needed to build the set of base words (that we also call dictionary) and then later use it in each document to find the index of each word in the document increases linearly with the size of the dictionary (at least with a dictionary of words stored as a sequence or an array of words), and if we consider big realistic dictionaries with hundreds (or thousands) of words and we have a huge collection of documents to extract their feature vectors, this can be quite expensive to compute.

So, hash functions come to our rescue. Assume we have a hash function with a number of buckets large enough to encode all the different words we think that our entire corpus of documents have (or at least to encode **most of the words** we are interested in encoding).

So, instead of keeping a list of different words, we simply map every word with our hash function to get the associated index with the world. Of course, we know that hash functions have **collisions**, so that means that with this representation certain words will be assigned to the same index, so their frequency counts will be added up and we will loose some information.

But with good hash functions, with a large enough number of buckets, the real impact of these collisions can be minimized. Spark has an implementation of TF-IDF based scoring that uses this hashing trick to assign indexes to words.

For more information about how spark uses TF-IDF based scores for words and hash functions for fast feature representation and extraction, take a look at this documentation page:

http://spark.apache.org/docs/3.0.0/mllib-feature-extraction.html#tf-idf (http://spark.apache.org/docs/3.0.0/mllib-feature-extraction.html#tf-idf)

Naive Bayes with TF-IDF scoring of words in Spark

In [10]:
```
# Import the needed object classes, HashingTF and IDF
#
from pyspark.mllib.feature import HashingTF, IDF
```

In [11]:
```
labels = training_rawdocs.map(
    lambda doc: doc["label"],  # Standard Python dict access
    preservesPartitioning=True # This is obsolete, but we want to explicitely i
ndicate it.
)
# Get the TF score of each word, but represent each word with a hash code
# with hash range of values equal to parameter numFeatures (50)
tfDocs = HashingTF(numFeatures=50).transform(
    training_rawdocs.map(lambda doc: doc["text"].split(),
    preservesPartitioning=True)).cache()
```

In [12]:
```
# Compute the TF-IDF score as the final feature value for each hashed term
# We first compute the IDF information from the TF information, and then
# combine the TF and IDF information to get the final TFIDF score again
# as a RDD
idf = IDF().fit(tfDocs)
tfidf = idf.transform(tfDocs)
print( "First tf-idf vector: ", tfidf.take(1) )

# Combine labels with TF-IDF feature based version of documents using zip
# to finally build set of LabeledPoints to train NaiveBayes model
# the RDD1.zip(RDD2) transformation creates pairs (a,b) where a comes
# from RDD1 and b from RDD2
training2 = labels.zip(tfidf).map(lambda x: LabeledPoint(x[0], x[1]))

print (" training set, first doc: ", training2.take(1))
```

```
First tf-idf vector:  [SparseVector(50, {0: 1.4663, 1: 2.4275, 4: 1.8718, 9:
1.4663, 10: 1.4663, 11: 3.7436, 12: 0.1671, 13: 1.1787, 17: 1.5464, 21: 1.911,
22: 4.399, 23: 0.619, 28: 1.911, 30: 1.1787, 31: 0.9555, 37: 2.9327, 39: 2.866
5, 42: 1.8718, 45: 1.4663, 46: 1.1787, 47: 3.7436, 48: 0.7732})]
 training set, first doc:  [LabeledPoint(0.0, (50,[0,1,4,9,10,11,12,13,17,21,2
2,23,28,30,31,37,39,42,45,46,47,48],[1.466337068793427,2.427539078908504,1.871
8021769015913,1.466337068793427,1.466337068793427,3.7436043538031827,0.1670540
8466316624,1.1786549963416462,1.5463797764669633,1.9110228900548727,4.39901120
6380281,0.6190392084062235,1.9110228900548727,1.1786549963416462,0.95551144502
74363,2.932674137586854,2.866534335082309,1.8718021769015913,1.46633706879342
7,1.1786549963416462,3.7436043538031827,0.7731898882334817]))]
```

Let's train a Naive Bayes model with this TF-IDF based feature model for our data set of documents.

In [13]:
```
# Train and check
model2 = NaiveBayes.train(training2)

# model2.predict(RDDofTFIDFdocs) gives RDD with the predictions for all the doc
s
labels_and_preds2 = labels.zip(model2.predict(tfidf)).map(
    lambda x: {"actual": x[0], "predicted": float(x[1])})

labels_and_preds2.cache()

for landp in labels_and_preds2.collect():
  print (landp)

# Compute fraction of sucessfully classified docs
accuracy2 = 1.0 * labels_and_preds2.filter(lambda x : x['predicted'] == x['actu
al']).count() / labels_and_preds2.count()
print ("\n Accuracy (fraction of correctly classified documents) : ", accuracy2
)
```

```
{'actual': 0.0, 'predicted': 0.0}
{'actual': 0.0, 'predicted': 0.0}
{'actual': 0.0, 'predicted': 0.0}
{'actual': 0.0, 'predicted': 0.0}
{'actual': 1.0, 'predicted': 1.0}
{'actual': 1.0, 'predicted': 1.0}
{'actual': 1.0, 'predicted': 1.0}
{'actual': 1.0, 'predicted': 1.0}
{'actual': 2.0, 'predicted': 2.0}
{'actual': 2.0, 'predicted': 2.0}
{'actual': 2.0, 'predicted': 2.0}
{'actual': 2.0, 'predicted': 2.0}

 Accuracy (fraction of correctly classified documents) :  1.0
```

Small exercise: Repeat the previous naive bayes model training example but with smaller number of features. Observe that the number of different words in a our sample document set is 107, so with the current number of features (50) we are actually merging the information for many words. How much small can the feature set be without reducing too much the accuracy of the classifier obtained ?

Exercise: SPAM/HAM classification

Consider a collection of messages, where some of them are SPAM messages and some others not (HAM). Train a naive bayes model using the TF-IDF score with hashing representation of words, trying different values for the numFeatures parameter. Compare the accuracy obtained with each numFeatures value. Use the collection of SPAM/HAM messages that you can find at the following repository:

https://archive.ics.uci.edu/ml/datasets/sms+spam+collection (https://archive.ics.uci.edu/ml/datasets/sms+spam+collection)

You may need to apply some transformations/cleaning to get the data set in the appropiate format to be used with the spark naive bayes model learner.