# Naive Bayes

- Nearest neighbor approaches are called **lazy learners** because each time an instance is classified, we have to comb through the entire training set.
- Bayesian models are called **eager learners.** When given a training set, it builds the model, and then to classify instances, uses this internal model. Eager learners *tend to classify instances faster than lazy learners*

The ability to make probablistic predictions along with the fact that they are eager learners are two big advantages of Bayesian methods.

**P(h|d)** means the probability of "h" happening given some condition "D" (posterior probability)

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

- **P(h):** prior probability
- **P(h|d):** posterior probability
- **P(D) and P(D|h):** will be needed for Bayes

## Bayes Theorem

$$P(h|D) = \frac{P(D|h)P(h)}{P(D)}$$

You can **think of classes as hypotheses**- once we do this, we pick the class (hypothesis) with the highest probability.

**Maximum a posteriori hypothesis:** pick the hypothesis with the highest probability.

The core idea behind Bayesian probability here is illustrated in the example below:

- There is a form of cancer that affects 0.3% of people
- There is a test that identifies whether you have the cancer or not with 98% accuracy.
- **IF A RANDOM PERSON TESTS AS POSITIVE, THEY ARE STILL NOT THAT LIKELY TO HAVE THE CANCER!** In fact, it is far more likely they do not. I cannot stress this enough.

## Naive Bayes

Take each P(D|h), multiply them together, then **multiply by P(h)**. Whichever hypothesis yields the higher Bayesian probability can be classified as our prediction.

In [117]:

```
import math

class BayesClassifier:
    def __init__(self, bucketPrefix, testBucketNumber, dataFormat):
        counts = {}
        classes = {}
        total = 0

        #same stuff as before with the buckets
        self.format = dataFormat.strip().split('\t')
        self.prior = {}
        self.conditional = {}

        numericValues = {}
        totals = {}

        self.ssd = {}
        self.means = {}

        hasNums = False

        for i in range(1,11):
            if i != testBucketNumber:
                filename = "%s-%02i" % (bucketPrefix, i)
                f = open(filename)
                lines = f.readlines()
                f.close()
                for line in lines:
                    fields = line.strip().split('\t')
                    ignore = []
                    vector = []
                    nums = []
                    for i in range(len(fields)):
                        if self.format[i] == 'num':
                            hasNums = True
                            nums.append(float(fields[i]))
                        elif self.format[i] == 'attr':
                            vector.append(fields[i])
                        elif self.format[i] == 'comment':
                            ignore.append(fields[i])
                        elif self.format[i] == 'class':
                            category = fields[i]
                    total += 1
                    classes.setdefault(category, 0)
                    counts.setdefault(category,{})
                    classes[category] += 1
                    totals.setdefault(category, {})
                    numericValues.setdefault(category, {})
                    col = 0
                    for columnValue in vector:
                        #count 'em up
                        col += 1
                        counts[category].setdefault(col ,{})
                        counts[category][col].setdefault(columnValue, 0)
                        counts[category][col][columnValue]+= 1

                    col = 0
```

```
                    for columnValue in nums:
                        col += 1
                        totals[category].setdefault(col, 0)
                        totals[category][col] += columnValue
                        numericValues[category].setdefault(col, [])
                        numericValues[category][col].append(columnValue)

        #calculate probability for each class
        for (category, count) in classes.iteritems():
            self.prior[category] = count / total

        for (category, columns) in counts.iteritems():
            self.conditional.setdefault(category, {})
            for (col, valueCounts) in columns.iteritems():
                self.conditional[category].setdefault(col, {})
                for (attrValue, count) in valueCounts.items():
                    #calculate the probability of CONDITION / CLASS
                    self.conditional[category][col][attrValue] = count /
 classes[category]

        #EXTRA stuff we need to do if the classifier has to deal with nu
mbers
        #namely, we have to generate statistics for each column (mean, s
sd)
        if hasNums:
            for(category, columns) in totals.iteritems():
                self.means.setdefault(category,{})
                for(col, cTotal) in columns.iteritems():
                    self.means[category][col]= cTotal /
classes[category]

            for(category, columns) in numericValues.iteritems():
                self.ssd.setdefault(category, {})
                for (col, values) in columns.iteritems():
                    sumsd = 0
                    tmean = self.means[category][col]
                    for val in values:
                        sumsd += (val - tmean) ** 2
                    columns[col] = 0
                    self.ssd[category][col] = (sumsd /
(classes[category] - 1)) ** (0.5)

    def classify(self, itemVec, numVector):
        results = []
        for (category, prior) in self.prior.items():
            prob = prior
            col = 1
            if len(itemVec) > 0:
                #how do the categorical elements factor into the probabi
lity?
                for attrValue in itemVec:
                    if not attrValue in self.conditional[category][col]:
                        prob = 0
                    else:
                        prob = prob * self.conditional[category][col][at
trValue]
                    col += 1
```

```
            #how the continuous attributes factor into the probability
            if len(numVector) > 0:
                col = 1
                for x in numVector:
                    #calculate probability that value exists at that poi
nt on the normal
                    #distrobution based on sample-generated heuristics
                    mean = self.means[category][col]
                    ssd = self.ssd[category][col]
                    e = math.pow(math.e, -(x-mean) ** 2 / (2 * ssd **
2))
                    prob = prob * ((1.0 / math.sqrt(2 * math.pi))*e)
                    col += 1
            results.append((prob, category))
        #return hypothesis with maximum probability
        return(max(results)[1])

c = BayesClassifier("iHealth/i", 10, "attr\tattr\tattr\tattr\tclass")
print(c.classify(['health', 'moderate', 'moderate', 'yes'], []))

#WOOOHHH! it works. NOW LETS LOAD SOME POLITICAL DATA!

d = BayesClassifier("house-votes/hv", 10,
"class\tattr\tattr\tattr\tattr\tattr\tattr\tattr\tattr\tattr\tattr\tattr\
ttr\tattr\tattr\tattr\tattr")
print(d.classify(['yes','no','yes','yes','no','yes','yes','yes','no','ye
s','no','yes','no','yes','no','no'], []))

e = BayesClassifier("pimaSmall/pimaSmall", 1,
"num\tnum\tnum\tnum\tnum\tnum\tnum\tnum\tclass")
print(e.classify([], [3,78,50,32,88,31,0.248,26]))
```

```
i500
republican
1
```

### Problem with Naive Bayes

If there is an event that **hasn't occurred yet** (like "Democrat votes no for Healthcare" or something), and it is in the attribute set of an item we are trying to classify, it will **shoot the probability down to zero no matter what!** In many cases, this is illogical.

## Estimating probabilities

Probabilities in Naive Bayes are **estimates** of true probabilities. How can we fix this?

### Fixing this

Our formula now for each probability is shown below:

$$P(x|y) = \frac{n_c}{n}$$

This has some drawbacks, namely if something hasn't occured before, the entire value goes to zero.

$$P(x|y) = \frac{n_c + mp}{n + m}$$

**m** is a constant called the equivalent sample size. Many methods to calculate it, but for starters, we can use the numbers of classes. **p** is the prior probability of an event.

**m** does not have to remain constant! If there are 3 options (easy, medium, hard), m would be 3 and p would be 1/3.

# Numbers

So far we have been working with discrete categories instead of continuous. This makes sense- 100 is closer than 105, but a Saxophonist cannot be said to be closer to a Pianist. **How to use Bayesian methods for topics that fit less nicely into discrete categories?**

### Method 1: Making Categories

Split numbers into into intervals, with each interval as a distinct class.

### Method 2: Gaussian Distrobutions

### Population standard deviation and sample standard deviation

**Sample standard deviation** is the same as that for **population standard deviation** except with **n-1** in the denominator.

$$P(x_i|y_i) = \frac{1}{\sqrt{2\pi}\sigma_{i,j}} e^{\frac{-(x_i-\mu_{i,j})^2}{2\sigma_{i,j}^2}}$$

This formula gives the probability that x is in class y given its value, and assuming the values are scattered over a standard distrobution.

# Pros and Cons of Naive Bayes vs. kNN

**Pros** for Naive Bayes:

- **Simple** to implement!
- Needs **less** training data than many other methods
- Performs quick operations, and gives results fast.

Main **con** for Naive Bayes: cannot learn interactions among features. (e.g. I like food with cheese and cream but not both)

**Pros** for kNN:

- Also **simple** to implement
- Does not assume data has a particular structure
- **Large amount of memory** needed to store training set

**k Nearest Neighbors** makes sense when we have huge amounts of training data. Used in image classification, recommender systems, etc.

**NAIVE BAYES WORKS ON INDEPENDENT PROBABILITIES, BUT MOST REAL-WORLD ATTRIBUTES AREN'T INDEPENDENT!** This is what makes it "Naive"- we are assuming independence.