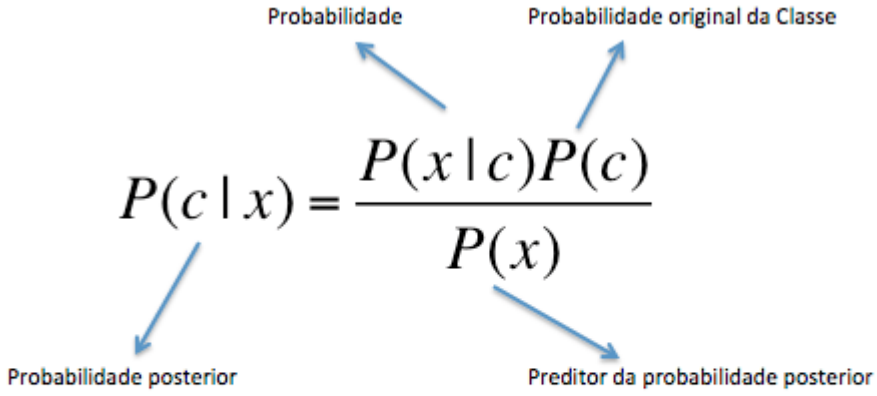
Naive Bayes

O que é o algoritmo Naive Bayes?

É uma técnica de classificação baseado no teorema de Bayes com uma suposição de independência entre os preditores. Em termos simples, um classificador Naive Bayes assume que a presença de uma característica particular não está relacionada com a presença de qualquer outra característica.

Por exemplo, um fruto pode ser considerado como uma maçã se é vermelho, redondo, e tiver um certo diâmetro. Mesmo que essas características dependam umas das outras, todas contribuem de forma independente para a probabilidade de que este fruto é uma maçã e é por isso que é conhecido como 'Naive' (ingênuo).

Teorema de Bayes fornece uma forma de calcular a probabilidade posterior P (C | X) a partir de P (C), P (x) e P (X | c). Veja a equação abaixo:



$$P(c \mid X) = P(x_1 \mid c) x P(x_2 \mid c) x ... x P(x_n \mid c) x P(c)$$

Acima,

- (c | x) é a probabilidade posterior da classe (c, alvo) dada preditor (x, atributos).
- (c) é a probabilidade original da classe.
- (x | c) é a probabilidade que representa a probabilidade de preditor dada a classe.
- (x) é a probabilidade original do preditor.

Como o algoritmo Naive Bayes funciona?

Vamos entender isso usando um exemplo. Abaixo eu tenho um conjunto de dados de treinamento de clima e da correspondente variável-alvo 'Play' (sugerindo possibilidades de jogar). Agora, precisamos classificar se os jogadores vão jogar ou não com base na

condição meteorológica.
Vamos seguir os passos abaixo para realizar a operação.

- Passo 1: Converter o conjunto de dados em uma tabela de frequência
- Passo 2: Criar tabela de Probabilidade ao encontrar as probabilidades de tempo Nublado = 0,29 e probabilidade de jogar = 0,64.

TEMPO	"PLAY"
Sol	Não
Nublado	Sim
Chuva	Sim
Sol	Sim
Sol	Sim
Nublado	Sim
Chuva	Não
Chuva	Não
Sol	Sim
Chuva	Sim
Sol	Não
Nublado	Sim
Nublado	Sim
Chuva	Não

Tabela de frequência		
Clima	Não	Sim
Nublado	0	4
Sol	3	2
Chuva	2	3
Total	5	9

Tabela de probabilidade				
Clima	Não	Sim		
Nublado	0	4	=4/14	0,29
Sol	3	2	=5/14	0,36
Chuva	2	3	=5/14	0,36
Total	5	9		
	=5/14	=9/14		
	0,36	0,64		

- Passo 3: Agora, use a equação Bayesiana Naive para calcular a probabilidade posterior para cada classe. A classe com maior probabilidade posterior é o resultado da previsão.

Problema: Qual a probabilidade de haver jogo sabendo-se que o tempo está ensolarado?

Podemos resolver isso usando o terorema de Bayes.

P (Sim |Ensolarado) = P (Ensolarado | Sim) * P (Sim) / P (Ensolarado)

Aqui temos P (Ensolarado | Sim) = 2/9 = 0,22, P (Ensolarado) = 5/14 = 0,36, P (Sim) = 9/14 = 0,64

Agora, P (Sim | Ensolarado) = 0,22 * 0,64 / 0,36 = 0,39, ou **39% de chances de haver jogo**.

Agora considere que você foi desafiado a usar a ciência de dados para descobrir uma maneira de filtrar mensagens de spam.

Um Filtro de Spam Sofisticado

Imagine que temos um vocabulário de muitas palavras **W1, ..., Wn**. Para levar isso para o reino da teoria da probabilidade, escreveremos **Xi** para o evento "uma mensagem contém a palavra **Wi**".

Imagine também que encontramos um **P(Xi|S)** como probabilidade de uma mensagem **spam** conter a i-ésima palavra e uma estimativa similar **P(Xi|nao S)** para probabilidade de

uma mensagem **não spam** contér a i-ésima palavra.

A chave para **Naive Bayes** é a suposição de que as presenças (ou ausências) de cada palavra são **independentes** umas das outras, na definição de uma mensagem ser spam ou não. Intuitivamente, essa suposição significa que saber se uma certa mensagem de spam contém a palavra "viagra" não fornece informações sobre se essa mesma mensagem contém a palavra "rolex".

Naive Bayes nos permite calcular a probabilidade de uma classe (spam, não spam) simplesmente multiplicando as estimativas de probabilidade individuais para cada palavra do vocabulário. Na prática, você geralmente deseja evitar a multiplicação de várias probabilidades, para evitar um problema chamado **underflow**, em que os computadores não lidam bem com números de ponto flutuante muito próximos de zero. Relembrando da álgebra que $\log(ab) = \log a + \log b$ e que $\exp(\log x) = x$ usualmente computamos como P1...Pn como equivalente $\exp(\log(p_1) + \cdots + \log(p_n))$

Lembre-se, usamos operação com log para evitar o problema de underflow ao lidar com multiplicação das probabilidades.

O único desafio que resta é encontrar as estimativas para **P(Xi|S)** e **P(Xi|Não S)**, as probabilidades de que uma mensagem de spam (ou mensagem não spam) conter a palavra *Wi*. Se tivermos um número razoável de mensagens de "treinamento" rotuladas como spam e não-spam, uma primeira tentativa óbvia é estimar simplesmente como a fração de mensagens de spam que contêm palavras *Wi*.

No entanto, isso causa um grande problema. Imagine que em nosso conjunto de treinamento a palavra "dados" do vocabulário só ocorre em mensagens não-spam. Então nós estimamos **P("dados"|S)=0**. O resultado é que nosso classificador Naive Bayes sempre atribuiria **probabilidade 0 (zero)** de spam a qualquer mensagem contendo a palavra "dados", até mesmo uma mensagem como "dados sobre relógios autênticos e rolex". Para evitar esse problema, geralmente usamos algum tipo de suavizador. Em particular, escolheremos um **pseudo contador k** e estimaremos a probabilidade de ver a i-ésima palavra em um spam como:

P(Xi|S) = (*k* + No spams contendo *Wi*) / (2*k* + No spams)

Similarmente para **P(Xi|Nao S)**. Isto é, ao computar as probabilidades de spam para a inésima palavra, assumimos que também vimos *k* spams adicionais contendo a palavra e *k* spams adicionais que não contêm a palavra.

Por exemplo, se a palavra "dado" ocorrer em 0/98 mensagens de spam, e se *k* for 1, estimamos como 1/100 = 0,01, o que permite ao nosso classificador atribuir ainda alguma probabilidade de spam diferente de zero a mensagens que contenham a palavra "dado".

Implementação

Agora temos todas as peças que precisamos para construir nosso classificador. Primeiro, vamos criar uma função simples para quebrar (*tokenize*) as mensagens em palavras distintas. Mas antes, vamos:

- converteremos cada mensagem em minúscula;
- use *re.findall()* para extrair "palavras" consistindo de letras, números e apóstrofos;
- e finalmente use set() para obter apenas as palavras distintas

In [31]:
```
def tokenize(message):
    message = message.lower()                    # convert to lowercase
    all_words = re.findall("[a-z0-9']+", message) # extract the words
    return set(all_words)                        # remove duplicates
```

In [39]:
```
re.findall("[a-z0-9']+", "Life Insurance - Why Pay More?".lower())
```

Out[39]:
```
['life', 'insurance', 'why', 'pay', 'more']
```

Nossa segunda função contará as palavras em um conjunto de treinamento rotulado de mensagens. Teremos que retornar um dicionário cujas chaves sejam palavras e cujos valores sejam listas de dois elementos [spam_count, non_spam_count] correspondendo a quantas vezes vimos essa palavra em mensagens de spam e não-spam:

In [32]:
```
def count_words(training_set):
    """training set consists of pairs (message, is_spam)"""
    counts = defaultdict(lambda: [0, 0])
    for message, is_spam in training_set:
        for word in tokenize(message):
            counts[word][0 if is_spam else 1] += 1
    return counts
```

Nosso próximo passo é transformar essas contagens em probabilidades estimadas usando a **suavização** descrita anteriormente. Nossa função retornará uma **lista de triplas** contendo cada palavra, a probabilidade de ver essa palavra em uma mensagem de spam e a probabilidade de ver essa palavra em uma mensagem não-spam:

In [40]:
```
def word_probabilities(counts, total_spams, total_non_spams, k=0.5):
    """turn the word_counts into a list of triplets w, p(w | spam) and p(w |
    return [(w,
             (spam + k) / (total_spams + 2 * k),
             (non_spam + k) / (total_non_spams + 2 * k))
             for w, (spam, non_spam) in counts.items()]
```

A última parte é usar essas probabilidades de palavras para atribuir probabilidades às mensagens:

In [41]:
```
import math
def spam_probability(word_probs, message):
    message_words = tokenize(message)
    log_prob_if_spam = log_prob_if_not_spam = 0.0

    # iterate through each word in our vocabulary
    for word, prob_if_spam, prob_if_not_spam in word_probs:

        # if *word* appears in the message, add the log probability of seeing
        if word in message_words:
            log_prob_if_spam += math.log(prob_if_spam)
            log_prob_if_not_spam += math.log(prob_if_not_spam)

        # if *word* doesn't appear in the message add the log probability of
        # which is log(1 - probability of seeing it)
        else:
            log_prob_if_spam += math.log(1.0 - prob_if_spam)
            log_prob_if_not_spam += math.log(1.0 - prob_if_not_spam)
    prob_if_spam = math.exp(log_prob_if_spam)
    prob_if_not_spam = math.exp(log_prob_if_not_spam)
    return prob_if_spam / (prob_if_spam + prob_if_not_spam)
```

Podemos juntar tudo isso em nosso Classificador Naive Bayes:

In [42]:
```
class NaiveBayesClassifier:
    def __init__(self, k=0.5):
        self.k = k
        self.word_probs = []

    def train(self, training_set):
        # count spam and non-spam messages
        num_spams = len([is_spam
                         for message, is_spam in training_set
                         if is_spam])
        num_non_spams = len(training_set) - num_spams

        # run training data through our "pipeline"
        word_counts = count_words(training_set)
        self.word_probs = word_probabilities(word_counts,
                                             num_spams,
                                             num_non_spams,
                                             self.k)

    def classify(self, message):
        return spam_probability(self.word_probs, message)
```

Testando nosso modelo

Um bom conjunto de dados (antigo, de certo modo) é o corpus público do **SpamAssassin**.
http://spamassassin.apache.org/old/publiccorpus/
Veremos os arquivos prefixados com 20021010.

Depois de extrair os dados você deve ter três pastas: **spam**, **easy_ham** e **hard_ham**. Cada pasta contém muitos e-mails, cada um contido em um único arquivo. Para manter as coisas simples, vamos ver as linhas de **assunto** de cada e-mail.
Como identificamos a linha de assunto? Olhando através dos arquivos, todos parecem começar com **"Subject:"** (Assunto). Então, vamos procurar por isso:

In [43]:
```
import glob, re
path = r"./spamassassin/*/*"
data = []
# glob.glob returns every filename that matches the wildcarded path
for fn in glob.glob(path):
    is_spam = True if "ham" not in fn else False
    with open(fn,'r', encoding="utf8", errors='ignore') as file:
        for line in file:
            if line.startswith("Subject:"):
                # remove the leading "Subject: " and keep what's left
                subject = re.sub(r"^Subject: ", "", line).strip()
                data.append((subject, is_spam))
```

In [44]:
```
len(data)
```

Out[44]: 3423

In [63]:
```
data[-5:-1]
```

Out[63]:
```
[('See your Company sales sky rocket.                 4611', True),
 ('Hit the Road with CNA', True),
 ('$10 a hour for watching e-mmercials! No joke!', True),
 ('Make a Fortune On eBay                           24772', True)]
```

Agora podemos dividir os dados em dados de treinamento e dados de teste e, em seguida, estamos prontos para executar em um classificador:

In [64]:
```
import random
from collections import defaultdict

def split_data(data, prob):
    """divide os dados en frações [prob, 1 - prob]"""
    results = [], []
    for row in data:
        results[0 if random.random() < prob else 1].append(row)
    return results
```

In [65]:
```
random.seed(0)
train_data, test_data = split_data(data, 0.75)
```

In [66]:
```
print('Train data size=', len(train_data),
      'Test data size=', len(test_data))
```

```
Train data size= 2547 Test data size= 876
```

In [49]:
```
# criando e treinando um classificador
classifier = NaiveBayesClassifier()
classifier.train(train_data)
```

In [50]:
```
print(classifier.classify("Life Insurance - Why Pay More?"))
print(classifier.classify("This week: Deck, Tex-Edit Plus, Boom"))
print(classifier.classify("Data Science Class"))
```

```
0.8159892501280752
0.07834054533624166
0.005007833873548498
```

E então podemos verificar como nosso modelo faz:

In [51]:
```
from collections import Counter
import math

# triplets (subject, actual is_spam, predicted spam probability)
classified = [(subject, is_spam, classifier.classify(subject))
              for subject, is_spam in test_data]

# assume that spam_probability > 0.5 corresponds to spam prediction
# and count the combinations of (actual is_spam, predicted is_spam)
counts = Counter((is_spam, spam_probability > 0.5)
                 for _, is_spam, spam_probability in classified)
```

In [52]:
```
#print(classified)
print(counts)
```

```
Counter({(False, False): 704, (True, True): 101, (True, False): 38, (False, T
rue): 33})
```

Resultado:

- 101 verdadeiros positivos (spam classificado como "spam")
- 33 falsos positivos (ham classificado como "spam")
- 704 negativos verdadeiros (ham classificado como "ham")
- 38 falsos negativos (spam classificado como "ham") .

Isso significa que nossa **precisão = 101 / (101 + 33) = 75,3%**, e nosso **recall = 101 / (101 + 38) = 72,6%**, que não são números ruins para um modelo tão simples.

Também é interessante ver os mais erroneamente classificados:

In [67]:
```
def precision(tp, fp, fn, tn):
    return tp / (tp + fp)

def recall(tp, fp, fn, tn):
    return tp / (tp + fn)

def f1_score(tp, fp, fn, tn):
    p = precision(tp, fp, fn, tn)
    r = recall(tp, fp, fn, tn)
    return 2 * p * r / (p + r)
```

In [69]:
```
print("precision = ", precision(101, 33, 38, 704))
print("recall = ", recall(101, 33, 38, 704))
print("f1-score = ", f1_score(101, 33, 38, 704))
```

```
precision =  0.753731343283582
recall =  0.7266187050359713
f1-score =  0.73992673992674
```

In [54]:
```
classified[:1]
```

Out[54]:
```
[('Re: New Sequences Window', False, 1.7755526669480555e-05)]
```

In [55]:
```
# sort by spam_probability from smallest to largest
classified.sort(key=lambda row: row[2])

# the highest predicted spam probabilities among the non-spams
spammiest_hams_f = filter(lambda row: not row[1], classified)
spammiest_hams = list(spammiest_hams_f)[-5:]

# the lowest predicted spam probabilities among the actual spams
hammiest_spams_f = filter(lambda row: row[1], classified)
hammiest_spams = list(hammiest_spams_f)[:5]
```

In [56]:
```
print(spammiest_hams)
```

```
[('Attn programmers: support offered [FLOSS-Sarai Initiative]', False, 0.9759
651625533735), ('2000+ year old Greek computer reinterpreted', False, 0.98377
51869158617), ('What to look for in your next smart phone (Tech Update)', Fal
se, 0.9900202987412373), ('[ILUG-Social] Re: Important - reenactor insurance
needed', False, 0.9995417850188201), ('[ILUG-Social] Re: Important - reenacto
r insurance needed', False, 0.9995417850188201)]
```

In [30]:
```
print(hammiest_spams)
```

```
[('I was so scared... my very first DP', True, 5.552557717017996e-05), ('Re:
Hi', True, 0.0009985164836998836), ('*****SPAM*****', True, 0.002184205081521
465), ('http://www.efi.ie/', True, 0.007967272221780357), ('Outstanding Oppor
tunities for "Premier Producers"', True, 0.00857958728638082)]
```

Como poderíamos obter um melhor desempenho? Uma maneira óbvia seria obter mais dados para treinar. Há várias maneiras de melhorar o modelo também. Aqui estão algumas possibilidades que você pode tentar:

- Veja o conteúdo da mensagem, não apenas a linha de assunto. Você precisa ter cuidado ao lidar com os cabeçalhos das mensagens.
- Nosso classificador leva em consideração todas as palavras que aparecem no conjunto de treinamento, até mesmo palavras que aparecem apenas uma vez. Modifique o classificador para aceitar um limiar min_count opcional e ignorar os tokens que não aparecem pelo menos muitas vezes.
- O tokenizador não tem noção de palavras semelhantes (por exemplo, "cheap" e "cheapest"). Modifique o classificador para obter uma função stemmer opcional que converta palavras em classes de equivalência de palavras.

Por exemplo, uma função stemmer realmente simples pode ser:

In [70]:
```
def drop_final_s(word):
    return re.sub("s$", "", word)
```

In [71]:
```
drop_final_s('hands')
```

Out[71]: 'hand'

Criar uma boa função stemmer é difícil. As pessoas freqüentemente usam o Porter Stemmer.
https://tartarus.org/martin/PorterStemmer/

SKLearn

A biblioteca Scikit-Learn possui uma implementação desse algoritmo que fizemos:
https://scikit-learn.org/stable/modules/naive_bayes.html#bernoulli-naive-bayes

In [ ]: