Наивный Байесовский классификатор.

В этом ноутбуке мы разберём один из самых простых методов классификации - наивный байесовский классификатор. Однако в некоторых задачах он работает даже лучше других, более сложных моделей. В любом случае, наивный байесовский классификатор содержит в себе важные теоретические идеи, поэтому с ним в любом случае полезно ознакомиться.

Применение при детекции спама.

Данные для решения задачи детекции спама можно сделать следующим образом: взять набор размеченных текстовых сообщений, часть которых размечена как спам, а остальные - как не спам, зафиксировать словарь (самый простой вариант - взять все слова, встречающиеся в наборе текстовых сообщений) и преобразовать текстовые данные в целочисленные, посчитав для каждого слова из словаря, встречается ли оно в данном сообщении. А на этих данных уже можно обучить наивный байесовский классификатор.

При реализации класса для наивного байесовского классификатора надо помнить один очень важный на практике момент: произведение вероятностей большого количества чисел может очень быстро сравняться с нулем при вычислении на компьютере, так как компьютеру может не хватить вычислительной точности. Поэтому при реализации стоит испльзовать логарифмы вероятностей.

Применим наивный байесовский классификатор к конкретному датасету https://archive.ics.uci.edu/ml/datasets/SMS+Spam+Collection (https://archive.ics.uci.edu/ml/datasets/SMS+Spam+Collection).

In [1]:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score, f1_score
from sklearn.model_selection import train_test_split
```

1. Чтение данных.

In [32]:

```
labels = []
messages = []

with open('SMSSpamCollection.txt', 'r') as fin:
    for line in fin:
        cur_label, cur_message = line.split('\t')
        labels.append(cur_label)
        messages.append(cur_message)
```

In [33]:

```
raw_df = pd.DataFrame()
raw_df['data'] = messages
raw_df['label'] = labels
raw_df.head()
```

Out[33]:

	data	label
0	Go until jurong point, crazy.. Available only ...	ham
1	Ok lar... Joking wif u oni...\n	ham
2	Free entry in 2 a wkly comp to win FA Cup fina...	spam
3	U dun say so early hor... U c already then say...	ham
4	Nah I don't think he goes to usf, he lives aro...	ham

В датасете метки бывают 2 видов:

- `ham` - означает, что сообщение - не спам,
- `spam` - означает, что сообщение - спам.

2. Предобработка данных.

Очевидно, что сразу в таком виде нельзя передавать данные наивному байесовскому классификатору. Их надо привести к численному виду.

Столбец `label` привести к численному виду можно очень просто.

In [34]:

```
raw_df['label'] = (raw_df['label'] == 'spam') * 1
raw_df.head()
```

Out[34]:

	data	label
0	Go until jurong point, crazy.. Available only ...	0
1	Ok lar... Joking wif u oni...\n	0
2	Free entry in 2 a wkly comp to win FA Cup fina...	1
3	U dun say so early hor... U c already then say...	0
4	Nah I don't think he goes to usf, he lives aro...	0

Для преобразования текстовых сообщений воспользуемся `CountVectorizer` , работающий по принципу мешка слов (bag of words). Он имеет следующие гиперпараметры:

- `max_df` -- максимальная доля сообщений, в которых может встречатся слово из словаря (такой параметр может быть полезен для борьбы со стоп-словами). То есть в словарь не включаются слишком частые слова.
- `min_df` -- минимальная доля сообщений, в которых может встречатся слово из словаря. То есть в словарь не включаются слишком редкие слова.

- `max_features` -- максимамальное возможное число слов в словаре (берётся `max_features` наиболее частых слов).
- `stop_words` -- можно просто взять и задать стоп-слова, которые не будут добавлены в словарь ни при каких обстоятельствах.

In [35]:

```
from sklearn.feature_extraction.text import CountVectorizer
```

In [36]:

```
vectorizer = CountVectorizer(min_df=0.03)
transformed_data = vectorizer.fit_transform(messages).toarray()
```

Напечатаем весь мешок слов и их количество:

In [37]:

```
print(len(vectorizer.get_feature_names()))
print(vectorizer.get_feature_names())
```

```
66
['all', 'am', 'and', 'are', 'at', 'be', 'but', 'call', 'can', 'come',
'day', 'do', 'for', 'free', 'from', 'get', 'go', 'good', 'got', 'gt',
'have', 'he', 'how', 'if', 'in', 'is', 'it', 'its', 'just', 'know', 'l
ike', 'll', 'love', 'lt', 'me', 'my', 'no', 'not', 'now', 'of', 'ok',
'on', 'only', 'or', 'out', 'send', 'so', 'text', 'that', 'the', 'the
n', 'there', 'this', 'time', 'to', 'up', 'ur', 'want', 'was', 'we', 'w
hat', 'when', 'will', 'with', 'you', 'your']
```

Посмотрим на преобразованные данные

In [38]:

```
print(transformed_data[:10])
```

```
[[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
0 0
  0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0
  0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
0 0
  0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 3 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0
  0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0
0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 1 0 0 0 0
0 0
  1 0 1 0 1 0 0 0 0 1 0 0 0 0 0 1 0 0 2 1 0 0 0 0 0 0 0 0 1 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 2 0 0 0
2 1
  0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0]
 [1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 3]
 [0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0
  0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 1 0]
 [0 0 0 0 0 0 0 1 0 0 0 0 1 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0
  0 0 0 0 0 1 0 1 0 0 0 0 0 2 0 0 0 0 2 0 0 0 0 0 0 0 0 1 0 1]]
```

3. Классификатор

В библиотеке `sklearn` имеются следующие реализации наивного байесовского классификатора:

1) `BernoulliNB` - байесовский классификатор для данных, в которых все признаки являются бинарными.

2) `MultinomialNB` - байесовский классификатор для данных, в которых все признаки являются дискретными.

3) `GaussianNB` - байесовский классификатор для вещественных данных, каждый из признаков которых имеет нормальное распределение.

Первые два метода имеют следующие гиперпараметры:

- `alpha` -- коэффициент сглаживания Лапласа или Линдсона, при фиксированном значении `alpha` условные плотности будут записаны следующим образом:
$$P(X_j = x_j | Y = y) = \frac{\#\{i : Y_i = y \ \& \ X_{ij} = x_j\} + \alpha}{\#\{i : Y_i = y\} + \alpha d},$$
где d -- количество признаков в датасете. При `alpha=0` сглаживания не происходит и получаются стандартные формулы для условных вероятностей;
- `prior` -- арпиорные вероятности принадлежности каждому из классов. Кроме того, можно не только задавать априорное распределение руками, но и воспользоваться функцией `fit_prior` ,

которая восстановит априорное распределение по переданным в функцию данным.

В нашей текущей задаче для признаков, описывающих количество вхождений каждого слова из словаря в сообщение, логично использовать `MultinomialNB`. Однако после мы сравним точность предсказаний `MultinomialNB` с точностью предсказаний `BernoulliNB` для бинарных признаков (каждый признак является индикатором того, присутствует ли данное слово из словаря в сообщении).

In [39]:

```
from sklearn.naive_bayes import MultinomialNB

multinomial_nb = MultinomialNB()
```

In [40]:

```
transformed_data
```

Out[40]:

```
array([[0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       ...,
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 1, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0]])
```

Как обычно, разделим данные на обучающую выборку и на тестовую.

In [41]:

```
X_train, X_test, y_train, y_test \
    = train_test_split(transformed_data, raw_df['label'],
                       random_state=42)
```

Обучаем модель и смотрим качество на тестовой выборке

In [42]:

```
multinomial_nb.fit(X_train, y_train)
predictions = multinomial_nb.predict(X_test)

print('accuracy:', accuracy_score(predictions, y_test))
print('f1 score:', f1_score(predictions, y_test))
```

```
accuracy: 0.9440459110473458
f1 score: 0.7845303867403315
```

Результат получился весьма неплохой.

А теперь посмотрим, как с этой же задачей справится наивный байесовский классификатор на бинарных данных.

In [43]:

```
X_train = (X_train > 0) * 1
X_test = (X_test > 0) * 1
```

```
[[0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 ...
 [0 0 0 ... 0 0 0]
 [0 0 1 ... 0 0 0]
 [0 0 0 ... 0 0 0]]
```

In [44]:

```
from sklearn.naive_bayes import BernoulliNB

bernoulli_nb = BernoulliNB()
```

In [45]:

```
bernoulli_nb.fit(X_train, y_train)
predictions = bernoulli_nb.predict(X_test)

print('accuracy:', accuracy_score(predictions, y_test))
print('f1 score:', f1_score(predictions, y_test))
```

```
accuracy: 0.9497847919655668
f1 score: 0.8055555555555556
```

Вывод.

Результат получился достаточно неожиданный. Наивный байесовский классификатор, обученный на бинаризованных данных показал более высокую точность классификации.

4. Больший размер словаря

А теперь посмотрим, что будет, если мы возьмём другое количество слов для словаря.

It's gonna be huge!

In [48]:

```
huge_vectorizer = CountVectorizer()
huge_data = huge_vectorizer.fit_transform(messages).toarray()
print(huge_data.shape)
```

```
(5574, 8713)
```

In [49]:

```
X_train, X_test, y_train, y_test = train_test_split(
    huge_data, raw_df['label'], random_state=42
)
```

In [50]:

```
multinomial_nb.fit(X_train, y_train)
predictions = multinomial_nb.predict(X_test)

print('accuracy:', accuracy_score(predictions, y_test))
print('f1 score:', f1_score(predictions, y_test))
```

```
accuracy: 0.9849354375896701
f1 score: 0.9454545454545454
```

In [51]:

```
bernoulli_nb.fit(X_train, y_train)
predictions = bernoulli_nb.predict(X_test)

print('accuracy:', accuracy_score(predictions, y_test))
print('f1 score:', f1_score(predictions, y_test))
```

```
accuracy: 0.9806312769010043
f1 score: 0.9247910863509748
```

Вывод.

От увеличения количества рассматриваемых слов в данном случае точность предсказаний возрасла как для наивного байесовского классификатора над категориальными признаками, так и для классификатора над бинарными признаками.

Бонусная часть

1. Рассмотрите различные способы построения словаря для классификации, например, установив некоторые параметры класса `CountVectorizer` .
2. Попытайтесь улучшить точность классификации (на тестовой выборке) наивного байесовского классификатор за счёт изменения гиперпаметров классификатора.
3. Решите задачу детекции спама при помощи некоторого другого известного классификатора: логистической регрессии, kNN и сравните точность предсказаний с наивным байесовским классификатором.