Vectorization, Multinomial Naive Bayes Classifier and Evaluation

Summary: Machine learning with text using Machine Learning with Text - Vectorization, Multinomial Naive Bayes Classifier and Evaluation

Topics

1. Model building in scikit-learn (refresher)
2. Representing text as numerical data
3. Reading a text-based dataset into pandas
4. Vectorizing our dataset
5. Building and evaluating a model
6. Comparing models
7. Examining a model for further insight
8. Practicing this workflow on another dataset
9. Tuning the vectorizer (discussion)

This guide is derived from Data School's Machine Learning with Text in scikit-learn session with my own additional notes so you can refer to them and they should be self-sufficient to guide you through.

1. Model building in scikit-learn (refresher)

In [1]:
```
# load the iris dataset as an example
from sklearn.datasets import load_iris
iris = load_iris()
```

In [2]:
```
# store the feature matrix (X) and response vector (y)

# uppercase X because it's an m x n matrix
X = iris.data

# lowercase y because it's a m x 1 vector
y = iris.target
```

- **"Features"**
  - Also known as predictors, inputs, or attributes
- **"Response"**
  - Also known as the target, label, or output
- **"Observations"**
  - Also known as samples, instances, or records

In [3]:
```
# check the shapes of X and y
print('X dimensionality', X.shape)
print('y dimensionality', y.shape)
```

```
X dimensionality (150, 4)
y dimensionality (150,)
```

In [4]:
```
# examine the first 5 rows of the feature matrix (including the feature names)
import pandas as pd
data = pd.DataFrame(X, columns=iris.feature_names)
data.head()
```

Out[4]:

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)
0	5.1	3.5	1.4	0.2
1	4.9	3.0	1.4	0.2
2	4.7	3.2	1.3	0.2
3	4.6	3.1	1.5	0.2
4	5.0	3.6	1.4	0.2

In [5]:
```
# examine the response vector
# this is a classification problem where you've 3 categories 0, 1, and 2
print(y)
```

```
[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 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2]
```

In order to **build a model**

1. Features must be **numeric**
    - Machine Learning models conduct mathematical operations so this is necessary
2. Every observation must have the **same features in the same order**
    - Rows must have features with the same order for meaningful comparison

In [6]:
```
## 4 STEP MODELLING

# 1. import the class
from sklearn.neighbors import KNeighborsClassifier

# 2. instantiate the model (with the default parameters)
knn = KNeighborsClassifier()

# 3. fit the model with data (occurs in-place)
knn.fit(X, y)
```

Out[6]:
```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowsk
i',
           metric_params=None, n_jobs=1, n_neighbors=5, p=2,
           weights='uniform')
```

In order to **make a prediction**, the new observation must have the **same features as the training observations**, both in number and meaning.

In [7]:
```
# 4. predict the response for a new observation
# here you pass in 4 features, the number of features that have been le
arned
knn.predict([3, 5, 4, 2])
```

```
/Users/ritchieng/anaconda3/envs/py3k/lib/python3.5/site-packages/sklear
n/utils/validation.py:386: DeprecationWarning: Passing 1d arrays as dat
a is deprecated in 0.17 and willraise ValueError in 0.19. Reshape your
data either using X.reshape(-1, 1) if your data has a single feature or
X.reshape(1, -1) if it contains a single sample.
  DeprecationWarning)
```

Out[7]:
```
array([1])
```

2. Representing text as numerical data

In [8]:
```
# example text for model training (SMS messages)
simple_train = ['call you tonight', 'Call me a cab', 'please call me..
 please']
```

From the scikit-learn documentation (http://scikit-learn.org/stable/modules/feature_extraction.html#text-feature-extraction):

> Text Analysis is a major application field for machine learning algorithms. However the raw data, a sequence of symbols cannot be fed directly to the algorithms themselves as most of them expect **numerical feature vectors with a fixed size** rather than the **raw text documents with variable length**.
>
> - You'll remember from the iris data that every row has 4 features
>   - scikit-learn expects all values to have meaning
>   - scikit-learn does not work with missing values
>   - it assumes all values have meaning
>   - it expects numerical feature vectors with a fixed size
> - Hence we're turn text into numbers

We will use CountVectorizer (http://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html) to "convert text into a matrix of token counts":

4 Steps for Vectorization

1. Import
2. Instantiate
3. Fit
4. Transform

The difference from modelling is that a vectorizer does not predict

In [9]:
```
# 1. import and instantiate CountVectorizer (with the default parameter
s)
from sklearn.feature_extraction.text import CountVectorizer

# 2. instantiate CountVectorizer (vectorizer)
vect = CountVectorizer()
```

In [10]:
```
# 3. fit
# learn the 'vocabulary' of the training data (occurs in-place)
vect.fit(simple_train)
```

Out[10]:
```
CountVectorizer(analyzer='word', binary=False, decode_error='strict',
        dtype=<class 'numpy.int64'>, encoding='utf-8', input='content',
        lowercase=True, max_df=1.0, max_features=None, min_df=1,
        ngram_range=(1, 1), preprocessor=None, stop_words=None,
        strip_accents=None, token_pattern='(?u)\\b\\w\\w+\\b',
        tokenizer=None, vocabulary=None)
```

vect.fit() notes

- It took out "a" due to the token_pattern (regex shown above)
- lower_case=True made all lowercase
- Alphabetical order
- No duplicate words

In [11]:
```
# examine the fitted vocabulary
vect.get_feature_names()
```

Out[11]:
```
['cab', 'call', 'me', 'please', 'tonight', 'you']
```

In [12]:
```
# 4. transform training data into a 'document-term matrix'
simple_train_dtm = vect.transform(simple_train)
simple_train_dtm
```

Out[12]:
```
<3x6 sparse matrix of type '<class 'numpy.int64'>'
        with 9 stored elements in Compressed Sparse Row format>
```

Why is it 3x6

- 3 rows x 6 columns
- document = rows
- term = columns
- That is why it's called a document-term matrix (row-column matrix)
  - 3 rows
    - Because there were 3 documents
  - 6 columns
    - 6 terms that were learned during the fitting steps
    - The terms are shown above when we ran vect.get_feature_names()

In [13]:
```
# convert sparse matrix to a dense matrix
simple_train_dtm.toarray()
```

Out[13]:
```
array([[0, 1, 0, 0, 1, 1],
       [1, 1, 1, 0, 0, 0],
       [0, 1, 1, 2, 0, 0]])
```

sparse matrix

- only store non-zero values
- if you have 0's, it'll only store the coordinates of the 0's

dense matrix

- seeing zero's and storing them
- if you have 1000 x 1000 of 0's, you'll store all

In [14]:
```
# examine the vocabulary and document-term matrix together
# pd.DataFrame(matrix, columns=columns)
pd.DataFrame(simple_train_dtm.toarray(), columns=vect.get_feature_names
())
```

Out[14]:

	cab	call	me	please	tonight	you
0	0	1	0	0	1	1
1	1	1	1	0	0	0
2	0	1	1	2	0	0

We will be training our model on this (X), that's why we need this

From the scikit-learn documentation (http://scikit-learn.org/stable/modules/feature_extraction.html#text-feature-extraction):

> In this scheme, features and samples are defined as follows:
>
> - Each individual token occurrence frequency (normalized or not) is treated as a **feature**.
> - The vector of all the token frequencies for a given document is considered a multivariate **sample**.
>
> A **corpus of documents** can thus be represented by a matrix with **one row per document** and **one column per token** (e.g. word) occurring in the corpus.
>
> We call **vectorization** the general process of turning a collection of text documents into numerical feature vectors. This specific strategy (tokenization, counting and normalization) is called the **Bag of Words** or "Bag of n-grams" representation. Documents are described by word occurrences while completely ignoring the relative position information of the words in the document.

In [15]:
```
# check the type of the document-term matrix
type(simple_train_dtm)
```

Out[15]:
```
scipy.sparse.csr.csr_matrix
```

In [16]:
```
# examine the sparse matrix contents
# left: coordinates of non-zero values
# right: values at that point
# CountVectorizer() will output a sparse matrix
print('sparse matrix')
print(simple_train_dtm)

print('dense matrix')
print(simple_train_dtm.toarray())
```

```
sparse matrix
  (0, 1)        1
  (0, 4)        1
  (0, 5)        1
  (1, 0)        1
  (1, 1)        1
  (1, 2)        1
  (2, 1)        1
  (2, 2)        1
  (2, 3)        2
dense matrix
[[0 1 0 0 1 1]
 [1 1 1 0 0 0]
 [0 1 1 2 0 0]]
```

From the scikit-learn documentation (http://scikit-learn.org/stable/modules/feature_extraction.html#text-feature-extraction):

> As most documents will typically use a very small subset of the words used in the corpus, the resulting matrix will have **many feature values that are zeros** (typically more than 99% of them).
>
> For instance, a collection of 10,000 short text documents (such as emails) will use a vocabulary with a size in the order of 100,000 unique words in total while each document will use 100 to 1000 unique words individually.
>
> In order to be able to **store such a matrix in memory** but also to **speed up operations**, implementations will typically use a **sparse representation** such as the implementations available in the `scipy.sparse` package.

In [17]:
```
# example text for model testing
simple_test = ['Please don\'t call me']
```

In order to **make a prediction**, the new observation must have the **same features as the training observations**, both in number and meaning.

In [18]:
```
# 4. transform testing data into a document-term matrix (using existing vocabulary)
simple_test_dtm = vect.transform(simple_test)
simple_test_dtm.toarray()
```

Out[18]:
```
array([[0, 1, 1, 1, 0, 0]])
```

In [19]:
```
# examine the vocabulary and document-term matrix together
pd.DataFrame(simple_test_dtm.toarray(), columns=vect.get_feature_names())
```

Out[19]:

	cab	call	me	please	tonight	you
0	0	1	1	1	0	0

It dropped the word "don't", why are we ok with the fact that the word "don't" drops?

- We don't know anything about the relationship between the word "don't" and the response (mean or not mean for example)
  - If we give a new word to predict the response, our model would not know what to do anyway
  - In essence, we did not train on the feature "don't" so our model would not be able to predict based on that new feature
- Iris dataset
  - During the predict step, say we collected a new feature
  - Our model would not know because our model was not trained on the feature

Summary:

- `vect.fit(train)` **learns the vocabulary** of the training data
- `vect.transform(train)` uses the **fitted vocabulary** to build a document-term matrix from the training data
- `vect.transform(test)` uses the **fitted vocabulary** to build a document-term matrix from the testing data (and **ignores tokens** it hasn't seen before)

3. Reading a text-based dataset into pandas

In [20]:
```
# read file into pandas using a relative path
path = 'data/sms.tsv'
features = ['label', 'message']
sms = pd.read_table(path, header=None, names=features)
```

In [21]:
```
# alternative: read file into pandas from a URL
# url = 'https://raw.githubusercontent.com/justmarkham/pycon-2016-tutorial/master/data/sms.tsv
 (https://raw.githubusercontent.com/justmarkham/pycon-2016-tutorial/master/data/sms.tsv)'
# sms = pd.read_table(url, header=None, names=['label', 'message'])
```

In [22]:
```
# examine the shape
sms.shape
```

Out[22]:
```
(5572, 2)
```

In [23]:
```
# examine the first 10 rows
sms.head()
```

Out[23]:

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

In [24]:
```
# examine the class distribution
sms.label.value_counts()
```

Out[24]:
```
ham     4825
spam     747
Name: label, dtype: int64
```

In [25]:
```
# convert label to a numerical variable
sms['label_num'] = sms.label.map({'ham':0, 'spam':1})
```

In [26]:
```
# check that the conversion worked
sms.head()
```

Out[26]:

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

In [27]:
```
# how to define X and y (from the iris data) for use with a MODEL
X = iris.data
y = iris.target
print(X.shape)
print(y.shape)
```

```
(150, 4)
(150,)
```

- X: 2 dimension (matrix)
- y: 1 dimension (vector)

In [28]:
```
# how to define X and y (from the SMS data) for use with COUNTVECTORIZE
R
X = sms.message
y = sms.label_num
print(X.shape)
print(y.shape)
```

```
(5572,)
(5572,)
```

- X is 1D currently because it will be passed to Vectorizer to become a 2D matrix
- You must always have a 1D object so CountVectorizer can turn into a 2D object for the model to be built on

In [29]:
```
# split X and y into training and testing sets
# by default, it splits 75% training and 25% test
# random_state=1 for reproducibility
from sklearn.cross_validation import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=
1)
print(X_train.shape)
print(X_test.shape)
print(y_train.shape)
print(y_test.shape)
```

```
(4179,)
(1393,)
(4179,)
(1393,)
```

Why are we splitting into training and testing sets before vectorizing?

Background of train/test split

- Train/test split is for model evaluation
  - Model evaluation is to simulate the future
  - Past data is exchangeable for future data
  - We pretend some of our past data is coming into our future data
  - By training, predicting and evaluating the data, we can check the performance of our model

Vectorize then split

- If we vectorize then we train/test split, our document-term matrix would contain every single feature (word) in the test and training sets
  - What we want is to simulate the real world
  - We would always see words we have not seen before so this method is not realistic and we cannot properly evaluate our models

Split then vectorize (correct way)

- We do the train/test split before the CountVectorizer to properly simulate the real world where our future data contains words we have not seen before

After you train your data and chose the best model, you would then train on all of your data before predicting actual future data to maximize learning.

4. Vectorizing our dataset

In [30]:
```
# 2. instantiate the vectorizer
vect = CountVectorizer()
```

In [31]:
```
# learn training data vocabulary, then use it to create a document-term matrix

# 3. fit
vect.fit(X_train)

# 4. transform training data
X_train_dtm = vect.transform(X_train)
```

In [32]:
```
# equivalently: combine fit and transform into a single step
# this is faster and what most people would do
X_train_dtm = vect.fit_transform(X_train)
```

In [33]:
```
# examine the document-term matrix
X_train_dtm
```

Out[33]:
```
<4179x7456 sparse matrix of type '<class 'numpy.int64'>'
        with 55209 stored elements in Compressed Sparse Row format>
```

In [34]:
```
# 4. transform testing data (using fitted vocabulary) into a document-term matrix
X_test_dtm = vect.transform(X_test)
X_test_dtm

# you can see that the number of columns, 7456, is the same as what we have learned above in X_train_dtm
```

Out[34]:
```
<1393x7456 sparse matrix of type '<class 'numpy.int64'>'
        with 17604 stored elements in Compressed Sparse Row format>
```

5. Building and evaluating a model

We will use multinomial Naive Bayes (http://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.MultinomialNB.html):

> The multinomial Naive Bayes classifier is suitable for classification with **discrete features** (e.g., word counts for text classification). The multinomial distribution normally requires integer feature counts. However, in practice, fractional counts such as tf-idf may also work.

In [35]:
```
# 1. import
from sklearn.naive_bayes import MultinomialNB

# 2. instantiate a Multinomial Naive Bayes model
nb = MultinomialNB()
```

In [36]:
```
# 3. train the model
# using X_train_dtm (timing it with an IPython "magic command")

%time nb.fit(X_train_dtm, y_train)
```

```
CPU times: user 3.15 ms, sys: 1.17 ms, total: 4.32 ms
Wall time: 3.54 ms
```

Out[36]:
```
MultinomialNB(alpha=1.0, class_prior=None, fit_prior=True)
```

Naive bayes is fast as seen above

- This matters when we're using 10-fold cross-validation with a large dataset

In [37]:
```
# 4. make class predictions for X_test_dtm
y_pred_class = nb.predict(X_test_dtm)
```

In [38]:
```
# calculate accuracy of class predictions
from sklearn import metrics
metrics.accuracy_score(y_test, y_pred_class)
```

Out[38]:
```
0.98851399856424982
```

In [39]:
```
# examine class distribution
print(y_test.value_counts())
# there is a majority class of 0 here, hence the classes are skewed

# calculate null accuracy (for multi-class classification problems)
# .head(1) assesses the value 1208
null_accuracy = y_test.value_counts().head(1) / len(y_test)
print('Null accuracy:', null_accuracy)

# Manual calculation of null accuracy by always predicting the majority class
print('Manual null accuracy:',(1208 / (1208 + 185)))
```

```
0    1208
1     185
Name: label_num, dtype: int64
Null accuracy: 0    0.867193
Name: label_num, dtype: float64
Manual null accuracy: 0.8671931083991385
```

In this case, we can see that our accuracy (0.9885) is higher than the null accuracy (0.8672)

In [40]:
```
# print the confusion matrix
metrics.confusion_matrix(y_test, y_pred_class)
```

Out[40]:
```
array([[1203,    5],
       [  11,  174]])
```

Confusion matrix
[TN FP
FN TP]

In [41]:
```
# print message text for the false positives (ham incorrectly classifie
d as spam)

X_test[y_pred_class > y_test]

# alternative less elegant but easier to understand
# X_test[(y_pred_class==1) & (y_test==0)]
```

Out[41]:
```
574              Waiting for your call.
3375            Also andros ice etc etc
45      No calls..messages..missed calls
3415            No pic. Please re-send.
1988    No calls..messages..missed calls
Name: message, dtype: object
```

In [42]:
```
# print message text for the false negatives (spam incorrectly classifi
ed as ham)

X_test[y_pred_class < y_test]
# alternative less elegant but easier to understand
# X_test[(y_pred_class=0) & (y_test=1)]
```

Out[42]:
```
3132    LookAtMe!: Thanks for your purchase of a video...
5       FreeMsg Hey there darling it's been 3 week's n...
3530    Xmas & New Years Eve tickets are now on sale f...
684     Hi I'm sue. I am 20 years old and work as a la...
1875    Would you like to see my XXX pics they are so ...
1893    CALL 09090900040 & LISTEN TO EXTREME DIRTY LIV...
4298    thesmszone.com (http://thesmszone.com) lets you send free anonymous a
n...
4949    Hi this is Amy, we will be sending you a free ...
2821    INTERFLORA - It's not too late to order Inter...
2247    Hi ya babe x u 4goten bout me?' scammers getti...
4514    Money i have won wining number 946 wot do i do...
Name: message, dtype: object
```

In [43]:
```
# example false negative
X_test[3132]
```

Out[43]:
```
"LookAtMe!: Thanks for your purchase of a video clip from LookAtMe!, yo
u've been charged 35p. Think you can do better? Why not send a video in
a MMSto 32323."
```

In [44]:
```
# calculate predicted probabilities for X_test_dtm (poorly calibrated)

# Numpy Array with 2C
# left Column: probability class 0
# right C: probability class 1
# we only need the right column
y_pred_prob = nb.predict_proba(X_test_dtm)[:, 1]
y_pred_prob

# Naive Bayes predicts very extreme probabilites, you should not take t
hem at face value
```

Out[44]:
```
array([  2.87744864e-03,   1.83488846e-05,   2.07301295e-03, ...,
         1.09026171e-06,   1.00000000e+00,   3.98279868e-09])
```

In [45]:
```
# calculate AUC
metrics.roc_auc_score(y_test, y_pred_prob)
```

Out[45]:
```
0.98664310005369604
```

- AUC is useful as a single number summary of classifier performance
- Higher value = better classifier
- If you randomly chose one positive and one negative observation, AUC represents the likelihood that your classifier will assign a higher predicted probability to the positive observation
- AUC is useful even when there is high class imbalance (unlike classification accuracy)
    - Fraud case
        - Null accuracy almost 99%
        - AUC is useful here

6. Comparing models

We will compare multinomial Naive Bayes with logistic regression (http://scikit-learn.org/stable/modules/linear_model.html#logistic-regression):

> Logistic regression, despite its name, is a **linear model for classification** rather than regression. Logistic regression is also known in the literature as logit regression, maximum-entropy classification (MaxEnt) or the log-linear classifier. In this model, the probabilities describing the possible outcomes of a single trial are modeled using a logistic function.

In [46]:
```
# 1. import
from sklearn.linear_model import LogisticRegression

# 2. instantiate a logistic regression model
logreg = LogisticRegression()
```

In [47]:
```
# 3. train the model using X_train_dtm
%time logreg.fit(X_train_dtm, y_train)
```

```
CPU times: user 109 ms, sys: 5.07 ms, total: 114 ms
Wall time: 66.2 ms
```

Out[47]:
```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=
True,
          intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=
1,
          penalty='l2', random_state=None, solver='liblinear', tol=0.00
01,
          verbose=0, warm_start=False)
```

This is a lot slower than Naive Bayes

- Naive Bayes cannot take negative numbers while Logistic Regression can

In [48]:
```
# 4. make class predictions for X_test_dtm
y_pred_class = logreg.predict(X_test_dtm)
```

In [49]:
```
# calculate predicted probabilities for X_test_dtm (well calibrated)
y_pred_prob = logreg.predict_proba(X_test_dtm)[:, 1]
y_pred_prob
```

Out[49]:
```
array([ 0.01269556,  0.00347183,  0.00616517, ...,  0.03354907,
        0.99725053,  0.00157706])
```

This is a good model if you care about the probabilities.

In [50]:
```
# calculate accuracy
metrics.accuracy_score(y_test, y_pred_class)
```

Out[50]:
```
0.9877961234745154
```

In [51]:
```
# calculate AUC
metrics.roc_auc_score(y_test, y_pred_prob)
```

Out[51]:
```
0.99368176123143015
```

7. Examining a model for further insight

We will examine the our **trained Naive Bayes model** to calculate the approximate **"spamminess" of each token**.

In [52]:
```
# store the vocabulary of X_train
X_train_tokens = vect.get_feature_names()
len(X_train_tokens)
```

Out[52]: 7456

In [53]:
```
# examine the first 50 tokens
print(X_train_tokens[0:50])
```

```
['00', '000', '008704050406', '0121', '01223585236', '01223585334', '01
25698789', '02', '0207', '02072069400', '02073162414', '02085076972',
'021', '03', '04', '0430', '05', '050703', '0578', '06', '07', '0700800
9200', '07090201529', '07090298926', '07123456789', '07732584351', '077
34396839', '07742676969', '0776xxxxxxx', '07781482378', '07786200117',
'078', '07801543489', '07808', '07808247860', '07808726822', '078152964
84', '07821230901', '07880867867', '0789xxxxxxx', '07946746291', '0796x
xxxxx', '07973788240', '07xxxxxxxxx', '08', '0800', '08000407165', '080
00776320', '08000839402', '08000930705']
```

In [54]:
```
# examine the last 50 tokens
print(X_train_tokens[-50:])
```

```
['yer', 'yes', 'yest', 'yesterday', 'yet', 'yetunde', 'yijue', 'ym', 'y
mca', 'yo', 'yoga', 'yogasana', 'yor', 'yorge', 'you', 'youdoing', 'you
i', 'youphone', 'your', 'youre', 'yourjob', 'yours', 'yourself', 'youwa
nna', 'yowifes', 'yoyyooo', 'yr', 'yrs', 'ything', 'yummmm', 'yummy',
'yun', 'yunny', 'yuo', 'yuou', 'yup', 'zac', 'zaher', 'zealand', 'zebr
a', 'zed', 'zeros', 'zhong', 'zindgi', 'zoe', 'zoom', 'zouk', 'zyada',
'èn', '⩱ud']
```

In [55]:
```
# Naive Bayes counts the number of times each token appears in each cla
ss
# trailing underscore - learned during fitting
nb.feature_count_
```

Out[55]:
```
array([[  0.,   0.,   0., ...,   1.,   1.,   1.],
       [  5.,  23.,   2., ...,   0.,   0.,   0.]])
```

In [56]:
```
# rows represent classes, columns represent tokens
nb.feature_count_.shape
```

Out[56]: (2, 7456)

Naive Bayes Summary

- For each token, it calculates the conditional probability of that token given each class
  - Does this for every token and both classes
- To make a prediction
  - Calculates conditional probability of a class given the token in that message
- Bottomline to how it thinks
  - Learns spamminess of each token
    - If have a lot of ham then class = ham
    - If have a lot of spam then class = spam

In [57]:
```
# number of times each token appears across all HAM messages
ham_token_count = nb.feature_count_[0, :]
ham_token_count
```

Out[57]:
```
array([ 0.,  0.,  0., ...,  1.,  1.,  1.])
```

In [58]:
```
# number of times each token appears across all SPAM messages
spam_token_count = nb.feature_count_[1, :]
spam_token_count
```

Out[58]:
```
array([  5.,  23.,   2., ...,   0.,   0.,   0.])
```

In [59]:
```
# create a DataFrame of tokens with their separate ham and spam counts
tokens = pd.DataFrame({'token':X_train_tokens, 'ham':ham_token_count,
'spam':spam_token_count}).set_index('token')
tokens.head()
```

Out[59]:

	ham	spam
token		
00	0.0	5.0
000	0.0	23.0
008704050406	0.0	2.0
0121	0.0	1.0
01223585236	0.0	1.0

In [60]:
```
# examine 5 random DataFrame rows
# random_state=6 is a seed for reproducibility
tokens.sample(5, random_state=6)
```

Out[60]:

	ham	spam
token		
very	64.0	2.0
nasty	1.0	1.0
villa	0.0	1.0
beloved	1.0	0.0
textoperator	0.0	2.0

In [61]:
```
# Naive Bayes counts the number of observations in each class
nb.class_count_
```

Out[61]:
```
array([ 3617.,   562.])
```

- 3617 Ham
- 562 Spam

Before we can calculate the "spamminess" of each token, we need to avoid **dividing by zero** and account for the **class imbalance**.

In [62]:
```
# add 1 to ham and spam counts to avoid dividing by 0
tokens['ham'] = tokens.ham + 1
tokens['spam'] = tokens.spam + 1
tokens.sample(5, random_state=6)
```

Out[62]:

	ham	spam
token		
very	65.0	3.0
nasty	2.0	2.0
villa	1.0	2.0
beloved	2.0	1.0
textoperator	1.0	3.0

In [63]:
```
# convert the ham and spam counts into frequencies
tokens['ham'] = tokens.ham / nb.class_count_[0]
tokens['spam'] = tokens.spam / nb.class_count_[1]
tokens.sample(5, random_state=6)
```

Out[63]:

	ham	spam
token		
very	0.017971	0.005338
nasty	0.000553	0.003559
villa	0.000276	0.003559
beloved	0.000553	0.001779
textoperator	0.000276	0.005338

In [72]:
```
# calculate the ratio of spam-to-ham for each token
tokens['spam_ratio'] = tokens.spam / tokens.ham
tokens.sample(5, random_state=6)
```

Out[72]:

	ham	spam	spam_ratio
token			
very	0.017971	0.005338	0.297044
nasty	0.000553	0.003559	6.435943
villa	0.000276	0.003559	12.871886
beloved	0.000553	0.001779	3.217972
textoperator	0.000276	0.005338	19.307829

You should not look at spam ratio and directly interpret

- textoperator is the most spammy word
- very is the least spammy word

In [73]:
```
# examine the DataFrame sorted by spam_ratio
# note: use sort() instead of sort_values() for pandas 0.16.2 and earli
er
tokens.sort_values('spam_ratio', ascending=False)
```

Out[73]:

	ham	spam	spam_ratio
token			
claim	0.000276	0.158363	572.798932
prize	0.000276	0.135231	489.131673
150p	0.000276	0.087189	315.361210
tone	0.000276	0.085409	308.925267
guaranteed	0.000276	0.076512	276.745552
18	0.000276	0.069395	251.001779
cs	0.000276	0.065836	238.129893
www	0.000553	0.129893	234.911922
1000	0.000276	0.056940	205.950178
awarded	0.000276	0.053381	193.078292
150ppm	0.000276	0.051601	186.642349
uk	0.000553	0.099644	180.206406
500	0.000276	0.048043	173.770463
ringtone	0.000276	0.044484	160.898577
000	0.000276	0.042705	154.462633
mob	0.000276	0.042705	154.462633
co	0.000553	0.078292	141.590747
collection	0.000276	0.039146	141.590747
valid	0.000276	0.037367	135.154804
2000	0.000276	0.037367	135.154804
800	0.000276	0.037367	135.154804
10p	0.000276	0.037367	135.154804
8007	0.000276	0.035587	128.718861
16	0.000553	0.067616	122.282918
weekly	0.000276	0.033808	122.282918
tones	0.000276	0.032028	115.846975
land	0.000276	0.032028	115.846975
http	0.000276	0.032028	115.846975
national	0.000276	0.030249	109.411032
5000	0.000276	0.030249	109.411032
...	...	...	...

	ham	spam	spam_ratio
token			
went	0.012718	0.001779	0.139912
ll	0.052530	0.007117	0.135494
told	0.013824	0.001779	0.128719
feel	0.013824	0.001779	0.128719
gud	0.014100	0.001779	0.126195
cos	0.014929	0.001779	0.119184
but	0.090683	0.010676	0.117731
amp	0.015206	0.001779	0.117017
something	0.015206	0.001779	0.117017
sure	0.015206	0.001779	0.117017
ok	0.061100	0.007117	0.116488
said	0.016312	0.001779	0.109084
morning	0.016865	0.001779	0.105507
yeah	0.017694	0.001779	0.100562
lol	0.017694	0.001779	0.100562
anything	0.017971	0.001779	0.099015
my	0.150401	0.014235	0.094646
doing	0.019077	0.001779	0.093275
way	0.019630	0.001779	0.090647
ask	0.019630	0.001779	0.090647
already	0.019630	0.001779	0.090647
too	0.021841	0.001779	0.081468
come	0.048936	0.003559	0.072723
later	0.030688	0.001779	0.057981
lor	0.032900	0.001779	0.054084
da	0.032900	0.001779	0.054084
she	0.035665	0.001779	0.049891
he	0.047000	0.001779	0.037858
lt	0.064142	0.001779	0.027741
gt	0.064971	0.001779	0.027387

7456 rows × 3 columns

In [74]:
```
# look up the spam_ratio for a given token
tokens.loc['dating', 'spam_ratio']
```

Out[74]:
```
83.667259786476862
```

8. Practicing this workflow on another dataset

Please open the **`exercise.ipynb`** notebook (or the **`exercise.py`** script).

9. Tuning the vectorizer (discussion)

Thus far, we have been using the default parameters of CountVectorizer (http://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html):

In [75]:
```
# show default parameters for CountVectorizer
vect
```

Out[75]:
```
CountVectorizer(analyzer='word', binary=False, decode_error='strict',
        dtype=<class 'numpy.int64'>, encoding='utf-8', input='content',
        lowercase=True, max_df=1.0, max_features=None, min_df=1,
        ngram_range=(1, 1), preprocessor=None, stop_words=None,
        strip_accents=None, token_pattern='(?u)\\b\\w\\w+\\b',
        tokenizer=None, vocabulary=None)
```

However, the vectorizer is worth tuning, just like a model is worth tuning! Here are a few parameters that you might want to

- **stop_words:** string {'english'}, list, or None (default)
  - If 'english', a built-in stop word list for English is used
    - A couple of hundred words (a lot of prepositions and indefinite articles)
  - If a list, that list is assumed to contain stop words, all of which will be removed from the resulting tokens
    - You can create your own stop words list
  - **If None, no stop words will be used**

Remember, you want signal not noise. Stop words help you reduce the number of features.

In [76]:
```
# remove English stop words
vect = CountVectorizer(stop_words='english')
```

- **ngram_range:** tuple (min_n, max_n), default=(1, 1)
  - The lower and upper boundary of the range of n-values for different n-grams to be extracted
  - All values of n such that min_n <= n <= max_n will be used
    - one-grams
      - 'welcome', 'to', 'python'
    - two-grams
      - 'welcome to', 'python'
    - n-grams
  - Intuition behind n-grams
    - To capture word phrases' meaning
      - 'Happy' vs 'Not Happy' and 'Very Happy'
      - If you choose one-gram, it would not include the later two features that are critical in learning
  - Danger of using two-grams (bigrams)
    - Number of features will grow really quickly
    - You need to know if you are throwing in noise or signal into your model
    - You need to check if there's potential value for improving the model's performance

In [78]:
```
# include 1-grams and 2-grams
vect = CountVectorizer(ngram_range=(1, 2))
```

- **max_df:** float in range [0.0, 1.0] or int, default=1.0
  - When building the vocabulary, ignore terms that have a document frequency strictly higher than the given threshold (corpus-specific stop words).
  - If float, the parameter represents a proportion of documents.
  - If integer, the parameter represents an absolute count.

In [80]:
```
# ignore terms that appear in more than 50% of the documents
vect = CountVectorizer(max_df=0.5)
```

- **min_df:** float in range [0.0, 1.0] or int, default=1
  - When building the vocabulary, ignore terms that have a document frequency strictly lower than the given threshold. (This value is also called "cut-off" in the literature.)
  - If float, the parameter represents a proportion of documents.
  - If integer, the parameter represents an absolute count.

You can use **min_df** and **ngram_range** to choose terms that appear frequently instead of rare ones

In [81]:
```
# only keep terms that appear in at least 2 documents
vect = CountVectorizer(min_df=2)
```

Guidelines for tuning CountVectorizer:

- Use your knowledge of the **problem** and the **text**, and your understanding of the **tuning parameters**, to help you decide what parameters to tune and how to tune them.
- **Experiment**, and let the data tell you the best approach!

10. Resources

Text classification:

- Read Paul Graham's classic post, A Plan for Spam (http://www.paulgraham.com/spam.html), for an overview of a basic text classification system using a Bayesian approach. (He also wrote a follow-up post (http://www.paulgraham.com/better.html) about how he improved his spam filter.)
- Coursera's Natural Language Processing (NLP) course has video lectures (https://class.coursera.org/nlp/lecture) on text classification, tokenization, Naive Bayes, and many other fundamental NLP topics. (Here are the slides (http://web.stanford.edu/~jurafsky/NLPCourseraSlides.html) used in all of the videos.)
- Automatically Categorizing Yelp Businesses (http://engineeringblog.yelp.com/2015/09/automatically-categorizing-yelp-businesses.html) discusses how Yelp uses NLP and scikit-learn to solve the problem of uncategorized businesses.
- How to Read the Mind of a Supreme Court Justice (http://fivethirtyeight.com/features/how-to-read-the-mind-of-a-supreme-court-justice/) discusses CourtCast, a machine learning model that predicts the outcome of Supreme Court cases using text-based features only. (The CourtCast creator wrote a post explaining how it works (https://sciencecowboy.wordpress.com/2015/03/05/predicting-the-supreme-court-from-oral-arguments/), and the Python code (https://github.com/nasrallah/CourtCast) is available on GitHub.)
- Identifying Humorous Cartoon Captions (http://www.cs.huji.ac.il/~dshahaf/pHumor.pdf) is a readable paper about identifying funny captions submitted to the New Yorker Caption Contest.
- In this PyData video (https://www.youtube.com/watch?v=y3ZTKFZ-1QQ) (50 minutes), Facebook explains how they use scikit-learn for sentiment classification by training a Naive Bayes model on emoji-labeled data.

Naive Bayes and logistic regression:

- Read this brief Quora post on airport security (http://www.quora.com/In-laymans-terms-how-does-Naive-Bayes-work/answer/Konstantin-Tt) for an intuitive explanation of how Naive Bayes classification works.
- For a longer introduction to Naive Bayes, read Sebastian Raschka's article on Naive Bayes and Text Classification (http://sebastianraschka.com/Articles/2014_naive_bayes_1.html). As well, Wikipedia has two excellent articles (Naive Bayes classifier (http://en.wikipedia.org/wiki/Naive_Bayes_classifier) and Naive Bayes spam filtering (http://en.wikipedia.org/wiki/Naive_Bayes_spam_filtering)), and Cross Validated has a good Q&A (http://stats.stackexchange.com/questions/21822/understanding-naive-bayes).
- My guide to an in-depth understanding of logistic regression (http://www.dataschool.io/guide-to-logistic-regression/) includes a lesson notebook and a curated list of resources for going deeper into this topic.

- Comparison of Machine Learning Models (https://github.com/justmarkham/DAT8/blob/master/other/model_comparison.md) lists the advantages and disadvantages of Naive Bayes, logistic regression, and other classification and regression models.

scikit-learn:

- The scikit-learn user guide includes an excellent section on text feature extraction (http://scikit-learn.org/stable/modules/feature_extraction.html#text-feature-extraction) that includes many details not covered in today's tutorial.
- The user guide also describes the performance trade-offs (http://scikit-learn.org/stable/modules/computational_performance.html#influence-of-the-input-data-representation) involved when choosing between sparse and dense input data representations.
- To learn more about evaluating classification models, watch video #9 from my scikit-learn video series (https://github.com/justmarkham/scikit-learn-videos) (or just read the associated notebook (https://github.com/justmarkham/scikit-learn-videos/blob/master/09_classification_metrics.ipynb)).

pandas:

- Here are my top 8 resources for learning data analysis with pandas (http://www.dataschool.io/best-python-pandas-resources/).
- As well, I have a new pandas Q&A video series (http://www.dataschool.io/easier-data-analysis-with-pandas/) targeted at beginners that includes two new videos every week.

Tags: machine_learning (/tag_machine_learning)