Naïve Bayes Classifier

1 Objective

Construct a naïve Bayes classifier to classify email messages as spam or not spam ("ham"). A Bayesian decision rule chooses the hypothesis that maximizes $P(\text{Spam}|x)$ vs $P(\sim\text{Spam}|x)$ for email x.

Use any computer language you like. I recommend using Python as it includes ready access to a number of powerful packages and libraries for natural language processing (NLP). I include a few Python 3.6 excerpts to get you started. I also describe a several tools from the NLTK (natural language toolkit) to *pre-process* data to improve classification.

2 Naïve (conditionally independent) classification

Suppose that you have a dataset $\{x_N\}$. Each $x_k \in \{x_N\}$ is a separate email for this assignment. Each of the N data points $x_k = (f_1, f_2, \ldots, f_n) \in$ Pattern space $= X$ where $f_1, f_2, \ldots$ are called *features*. You extract features from each data point. Features in an email may include the list of "words" (*tokens*) in the message body. The goal in Bayesian classification is to compute two probabilities $P(\text{Spam}|x)$ and $P(\sim\text{Spam}|x)$ for each email. It classifies each email as "spam" or "not spam" by choosing the hypothesis with higher probability. Naïve Bayes assumes that features for x are independent given its class.

$P(\text{Spam}|x)$ is difficult to compute in general. Expand with the definition of conditional probability

$$P(\text{Spam}|x) = \frac{P(\text{Spam} \cap x)}{P(x)}.$$

Look at the denominator $P(x)$. $P(x)$ equals the probability of a particular email given the universe of all possible emails. This is *very* difficult to calculate. But it is just a number between 0 and 1 since it is a probability. It just "normalizes" $P(\text{Spam} \cap x)$. Now look at the numerator $P(\text{Spam} \cap x)$. First expand x into its features $\{f_n\}$. Each feature is an event that can occur or not (*i.e.* the word is in an email or not). So

$$\begin{aligned}\frac{P(\text{Spam} \cap x)}{P(x)} \propto P(\text{Spam} \cap x) &= P(\text{Spam} \cap f_1 \cap \cdots \cap f_n) \\ &= P(\text{Spam}) \cdot P(f_1 \cap \cdots \cap f_n|\text{Spam})\end{aligned}$$

Apply the multiplication theorem (HW2, 1.c) to the second term to give

$$P(f_1 \cap \cdots \cap f_n|\text{Spam}) = P(\text{Spam}) \cdot P(f_1|\text{Spam}) \cdot P(f_2|\text{Spam} \cap f_1) \cdots P(f_n|\text{Spam} \cap f_{n-1} \cap \cdots f_2 \cap f_1).$$

But now you are still stuck computing a big product of complicated conditional probabilities. Naïve Bayes classification makes an **assumption** that features are conditionally independent. This means that

$$P(f_j|f_k \cap \text{Spam}) = P(f_j|\text{Spam})$$

if $j \neq k$. This means that the probability you observe one feature (*i.e.* word) is independent of observing another word given the email is spam. This is a *naïve* assumption and weakens your model. But you can now simplify the above to

$$P(f_1 \cap \cdots \cap f_n|\text{Spam}) = P(\text{Spam}) \cdot P(f_1|\text{Spam}) \cdot P(f_2|\text{Spam}) \cdots P(f_n|\text{Spam}) = P(\text{Spam}) \cdot \prod_{k=1}^{n} P(f_k|\text{Spam}).$$

Therefore

$$P(\text{Spam}|x) = \frac{P(\text{Spam}) \cdot \prod_{k=1}^{n} P(f_k|\text{Spam})}{P(x)}$$

and similarly

$$P(\sim\text{Spam}|x) = \frac{P(\sim\text{Spam}) \cdot \prod_{k=1}^{n} P(f_k|\sim\text{Spam})}{P(x)}$$

You can ignore the $P(x)$ normalizing term since you only care which probability is larger and it is the same in both cases. This leads to the naïve Bayesian rule (called the *maximum a posteriori (MAP) estimator*) between the two hypotheses $\{H_k\}$ (*e.g.* $\{\text{Spam}, \sim\text{Spam}\}$):

$$H^{\text{MAP}} = \arg\max_k P(H_k) \cdot \prod_{k=1}^{n} P(f_k|H_k).$$

This says to calculate the conditional probabilities for each word in your email assuming the email is spam and then again assuming it is not spam. Then choose the hypothesis with higher conditional probability.

3 Implementation

There is a upside to corporate scandal. During prosecution for the Enron accounting scandal (`https://en.wikipedia.org/wiki/Enron_scandal`) plaintiffs were compelled to disclose a massive historical trove of all emails they sent and received. Researchers read through and categorized a subset of these messages as spam and ham. Download one of the curated "pre-processed" files from: `http://nlp.cs.aueb.gr/software_and_datasets/Enron-Spam/`. Each archive contains two folders: "spam" and "ham". Each of folder contains thousands emails each stored in its own file. Open several files in the directories and familiarize yourself with the data.

The next sections step you through one possible way to implement your naïve Bayes classifier. You may use the code blocks as provided or supplement them with your own improvements. You may also implement the classifier from scratch but you must write your own code (do not copy from other sources).

3.1 Read the Emails

The following code block read the raw emails from the extracted Enron archive. It prepares a single list with all the emails and a "tag" to track your classifiers accuracy. It requires the path to the root Enron data folder (`BASE_DATA_DIR`).

```
def load_data(dir):
  list = []
  for file in os.listdir(dir):
    with open(dir + '/' + file, 'rb') as f:
      body = f.read().decode('utf-8', errors='ignore').splitlines()
      list.append(' '.join(body))
  return list

BASE_DATA_DIR='' # fill me in

# load and tag data
ham = [(text, 'ham') for text in load_data(BASE_DATA_DIR + '/ham')]
spam = [(text, 'spam') for text in load_data(BASE_DATA_DIR +
  '/spam')]
```

```
all = ham + spam

'''
when complete:

all = [
  (ham_email_1_str, 'ham'),
  (ham_email_2_str, 'ham'),
  ...
  (spam_email_1_str, 'spam'),
  (spam_email_2_str, 'spam')
  ...
]
'''
```

3.2 Pre-Processing

The raw emails are not ideal for classification. Consider applying each of the following operations in a "pre-processing" step to prepare the data. You can wrap all the functions in a single function as shown below.

```
def preprocess(text):
  # add all preprocessing here,
  # ...
  # ...

  # then return a list of tokens (words)
  return tokens

all = [(preprocess(text), label) for (text,label) in all]
```

Operations

- **normalize** Python is case-sensitive. This means that your classifier will treat differing cases (*e.g.* `Tomorrow` and `tomorrow`) as different words. Try pre-processing your entire message to lower case (`text = text.lower()`) and check if that improves accuracy.

- **tokenize** The python natural language toolkit (NLTK) provides useful string processing utilities. Tokenizing splits a long string into tokens (often called "words"). You can also tell the tokenizer to remove tokens that don't look like words (such as numbers and punctuation). To get a list of tokens from a text string you can use the NLTK tokenizer like

```
from nltk.tokenize import RegexpTokenizer
# only keep alphabet words
tokenizer = RegexpTokenizer(r'[a-z]+')
# get list of list of tokens
tokens = tokenizer.tokenize(text)
```

- **lemmatize** Lemmatizing is the process of removing common word suffixes such as "-ing" and "-ed", "-s" (plural). Otherwise your classifier will treat different forms of the same word (*e.g.* "vacation" vs

"vacations") as different words. To clean this list of tokens you can use the NLTK lemmatizer like

```
from nltk.stem import WordNetLemmatizer
lemmatizer = WordNetLemmatizer()
# map list of tokens
tokens = [lemmatizer.lemmatize(t) for t in tokens]
```

•**stopwords** Stopwords are common words that do not convey much of information (*e.g.* "the", "and", and "as"). And these words can wash out signals from more important words. To filter the stopwords from your list of tokens you can apply the NLTK list of standard English stop words like

```
from nltk.corpus import stopwords
stoplist = stopwords.words('english')
# filter list of list of tokens
tokens = [t for t in tokens if not t in stoplist]
```

•**other** Other filtering? Get creative. You can also wait until you complete your classifier and then go back and examine messages your classifier had trouble classifying. You may be able to identify a common pattern that leads to a number of misclassification. For instance each of the message files includes a "Subject" line as the first line with the email body following. Also some messages are in html instead of plain text format and you may want to augment the list of stopwords with common html tag names.

3.3 Prepare *train* and *test* Sets

It is bad statistical practice to build and test your model with the same data. Instead you should holdout some fraction of your data (often 20%) and use the remaining data (80%) to build (*train*) your model. Then *test* your model with the holdout data. The following shows how to achieve an 80/20 split into `train` and `test` sets.

```
# shuffle (may disable for debug)
random.shuffle(all)

# split train/test
splitp = 0.80 # 80/20 split
train = all[:int(splitp*len(all))]
test  = all[int(splitp*len(all)):]
```

3.4 Estimate $P(f_k|H_k)$ – *i.e.* $P(\textbf{word}|\textbf{Spam})$

You should construct two dictionaries (one spam dictionary and one ham dictionary) to estimate $P(f_k|A)$ and $P(f_k|A^C)$. The following code shows how you can iterate through the **training** set to build the dictionaries. Each time you encounter a word set `Dict[word] = Dict[word] + 1`. When you finish you can estimate of the probability of words in spam messages by looking up the count in your dictionary and dividing by the total number of words across all spam messages.

```
SpamDict = {}
HamDict{}

def featurizeTokens(tokens, is_spam):
  '''
```

```
check each word in the email
count into the correct dictionary (use an if statement)
'''

for (tokens,label) in train:
  featurizeTokens(tokens, label == 'spam')
```

3.5 Compare $P(\textbf{Spam}|x)$ and $P(\sim\textbf{Spam}|x)$

You are ready to calculate $P(\text{Spam}|x)$ for a given email! Iterate over your **test** emails. Compute the following product: $P(\text{Spam}) \cdot \prod_{k=1}^{n} P(f_k|\text{Spam})$. $P(\text{Spam})$ is the probability of a message being spam and depends on the number of spam messages in your training set (how do you compute it?). Use your `SpamDict` to estimate $P(f_k|\text{Spam})$ for each word in the message. Repeat the calculation to compute $P(\sim\text{Spam}) \cdot \prod_{k=1}^{n} P(f_k|\sim\text{Spam})$. Then assign the email to the category with the highest probability.

Two issues

1. $P(f_k|\textbf{Spam}) << 1$**.**

   $P(f_k|\text{Spam})$ will be very small since the probability that the word f_k occurs in any email is small. Therefore $\prod P(f_k|\text{Spam})$ will be *extremely* small. This causes computational precision problems. Avoid this issue by using the fact that you can turn a product into a sum of logs, *i.e.*

$$P(\text{Spam}) \cdot \prod_{k=1}^{N} P(f_k|\text{Spam}) = P(\text{Spam}) \cdot \exp\left\{\sum_{k=1}^{N} \ln P(f_k|\text{Spam})\right\}$$

2. **Test emails contain words that are not in your dictionary.**

   If a word never appears in any training emails then $P(f_k|\text{Spam}) = 0$ since `SpamDict[word] = 0`. Therefore $\prod P(f_k|\text{Spam}) = 0$. You can solve this in a few ways. One method is to "skip" these words by putting $P(f_k|\text{Spam})$ to some value that will not alter the probability when multiplied (what value would you choose)? The other is to use Laplace smoothing:

$$P(\text{word}|\text{Spam}) = \frac{\texttt{SpamDict[word]} + 1}{\texttt{sum(SpamDict.values())} + \texttt{len(SpamDict)} + 1}$$

   The idea here is that you assume that every word (even words not in your training set) has at least some very small probability of occurring (called a "uniform prior"). This resolves the issue by giving $P(\text{word}|\text{Spam}) > 0$ even if `SpamDict[word] = 0`.

4 Analysis

Apply your Naïve Bayes classifier to the test emails. Run your code several times. Use an operation similar to `shuffle` to ensure each trial uses a different train vs. test sets. Check for consistent results between runs. Report the average "correct" and "incorrect" classification rates as percentages. Summarize your two-way findings using a confusion matrix:

		Truth	
	I	spam	∼spam
Pred	spam	178	1
	∼spam	108	748

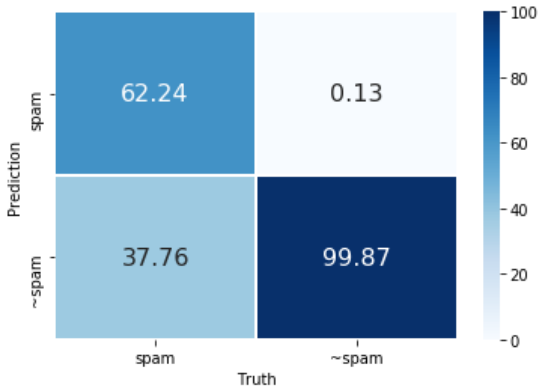
accuracy: 0.895

The seaborn python package can construct this directly from you classification rates:

```
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sn

df = pd.DataFrame([[TruePosRate, FalsePosRate], [FalseNegRate,
  TrueNegRate]])
fig = plt.figure()

ax = sn.heatmap(100*df, vmin=0, vmax=100, cmap='Blues',
  annot=True, fmt='.2f', annot_kws={"size":16}, linewidths=0.5)
ax.set_xlabel('Truth')
ax.set_ylabel('Prediction')
ax.set_xticklabels(['spam', '~spam'])
ax.set_yticklabels(['spam', '~spam'])
plt.show()
```



5 Other hints

If you want to use the Python NLTK you need to first download a few data files and auxiliary packages. It may take a few minutes to download all the files but you only need to do this once on your computer. Go the a python shell and type:

```
In [1]: import nltk

# if the corresponding "Out" line does say True you should run
  this command again
In [2]: nltk.download(['stopwords', 'punkt', 'wordnet'])
...
Out[2]: True
```

6 References

http://www2.aueb.gr/users/ion/docs/ceas2006_paper.pdf