Naive Bayes

Lecture 13

Centre for Data Science
Institute of Technical Education and Research
Siksha 'O' Anusandhan (Deemed to be University)
Bhubaneswar, Odisha, 751030

Overview

Dumb Spam filter

- Imagine a universe that consists of receiving a message chosen randomly from all possible messages.
- Let S be the event *the message is spam* and B be the event *the message contains the word bitcoin*.
- Bayes's theorem tells us that the probability that the message is spam conditional on containing the word bitcoin is:

$$P(S|B) = [P(B|S)P(S)]/[P(B|S)P(S) + P(B|\neg S)P(\neg S)]$$

Dumb Spam filter

- From below given equation,

$$P(S|B) = [P(B|S)P(S)]/[P(B|S)P(S) + P(B|\neg S)P(\neg S)]$$

- The numerator is the probability that a message is spam and contains bitcoin.
- The denominator is just the probability that a message contains bitcoin.
- We can think of this calculation as simply representing the proportion of bitcoin messages that are spam.

Dumb Spam filter

- If we have a large collection of messages we know are spam, and a large collection of messages we know are not spam, then we can easily estimate $P(B|S) and P(B|\neg S)$. If we further assume that any message is equally likely to be spam or not spam (so that $P(S) = P(\neg S) = 0.5$), then:

$$P(S|B) = [P(B|S)]/[P(B|S) + P(B|\neg S)]$$

- Suppose, if 50% of spam messages have the word bitcoin, but only 1% of nonspam messages do, then the probability that any given bitcoin-containing email is spam is:

$$0.5/(0.5 + 0.01) = 98\%$$

Sophisticated Spam filter

- Imagine now that we have a vocabulary of many words, $w_1 ..., w_n$.
- To move this into the realm of probability theory, we'll write X_i for the event *a message contains the word* w_i.
- Imagine that we've come up with an estimate $\mathrm{P}(X_i|S)$ for the probability that a spam message contains the ith word
- A similar estimate $\mathrm{P}(X_i|\neg S)$ for the probability that a nonspam message contains the ith word.
- The key to Naive Bayes is making the assumption that the presences (or absences) of each word are independent of one another, conditional on a message being spam or not.

Sophisticated Spam filter

- This assumption means that knowing whether a certain spam message contains the word *bitcoin* gives you no information about whether that same message contains the word *rolex*.
- In maths terms, this is equivalent to

$$P(X_1 = x_1, \ldots, X_n = x_n|S) = P(X_1 = x_1|S) \times \ldots \times P(X_n = x_n|S)$$

Sophisticated Spam filter

- Imagine that our vocabulary consists only of the words *bitcoin* and *rolex*, and that half of all spam messages are for "earn bitcoin" and that the other half are for "authentic rolex".
- In this case, the Naive Bayes estimate that a spam message contains both *bitcoin* and *rolex* is:

$$P(x_1 = 1, X_2 = 1|S) = P(x_1 = 1|S)P(x_2 = 1|S) = 0.5 \times 0.5 = 0.25$$

- Since we've assumed away the knowledge that *bitcoin* and *rolex* actually never occur together.
- Despite the unrealisticness of this assumption, this model often performs well and has historically been used in actual spam filters.

Sophisticated Spam filter

- The Naive Bayes assumption allows us to compute each of the probabilities on the right simply by multiplying together the individual probability estimates for each vocabulary word.
- In practice, you usually want to avoid multiplying lots of probabilities together, to prevent a problem called *underflow*, in which computers don't deal well with floating-point numbers that are too close to 0.

Sophisticated Spam filter

- Imagine that in our training set the vocabulary word *data* only occurs in nonspam messages.
- Then we'd estimate P(data—S) = 0. The result is that our Naive Bayes classifier would always assign spam probability 0 to any message containing the word data, even a message like *"data on free bitcoin and authentic rolex watches"*. To avoid this problem, we usually use some kind of smoothing.
- In particular, we'll choose a *pseudocount-k-* and estimate the probability of seeing the ith word in a spam message as: $P(X_i|S)$ = P(k + number of spams containing w_i)/(2k + number of spams)

Implementation

- We have all the pieces we need to build our classifier.
- First, let's create a simple function to tokenize messages into distinct words.
- We'll first convert each message to lowercase, then use *re.findall* to extract "words" consisting of letters, numbers, and apostrophes.
- we'll use set to get just the distinct words.

Implementation

```
from typing import Set
import re
def tokenize(text: str) -> Set[str]:
    text = text.lower()
    all_words = re.findall("[a-z0-9']+", text)
    return set(all_words)
assert tokenize("Data Science is science") == {"data", "
    science", "is"}
```

- We'll also define a type for our training data:

```
from typing import NamedTuple
class Message(NamedTuple):
    text: str
    is_spam: bool
```

Implementation

- As our classifier needs to keep track of tokens, counts, and labels from the training data, we'll make it a class.
- we refer to nonspam emails as *ham* emails.

```
from typing import List, Tuple, Dict, Iterable
import math
from collections import defaultdict

class NaiveBayesClassifier:
    def __init__(self, k: float = 0.5) -> None:
        self.k = k  # smoothing factor

        self.tokens: Set[str] = set()
        self.token_spam_counts: Dict[str, int] = defaultdict(int)
        self.token_ham_counts: Dict[str, int] = defaultdict(int)
        self.spam_messages = self.ham_messages = 0
```

Implementation

- we'll give it a method to train it on a bunch of messages. First, we increment the *spam_messages* and *ham_messages* counts.
- Then we tokenize each message text, and for each token we increment the *token_spam_counts* or *token_ham_counts* based on the message type.

Implementation

```
#This is method of class NaiveBayesClassifier
    def train(self, messages: Iterable[Message]) -> None:
        for message in messages:
            # Increment message counts
            if message.is_spam:
                self.spam_messages += 1
            else:
                self.ham_messages += 1

            # Increment word counts
            for token in tokenize(message.text):
                self.tokens.add(token)
                if message.is_spam:
                    self.token_spam_counts[token] += 1
                else:
                    self.token_ham_counts[token] += 1
```

Implementation

- We'll want to predict P(spam — token).
- As we saw earlier, to apply Bayes's theorem we need to know P(token — spam) and P(token — ham) for each token in the vocabulary.
- So we'll create a "private" helper function to compute it.

Implementation

```python
#This is also a method of class NaiveBayesClassifier
    def _probabilities(self, token: str) -> Tuple[float, float]:
        """returns P(token | spam) and P(token | not spam)"""
        spam = self.token_spam_counts[token]
        ham = self.token_ham_counts[token]

        p_token_spam = (spam + self.k) / (self.spam_messages + 2 * self.k)
        p_token_ham = (ham + self.k) / (self.ham_messages + 2 * self.k)

        return p_token_spam, p_token_ham
```

Implementation

- we're ready to write our predict method.

```
#This is also a method of class NaiveBayesClassifier
    def predict(self, text: str) -> float:
        text_tokens = tokenize(text)
        log_prob_if_spam = log_prob_if_ham = 0.0
        for token in self.tokens:
            prob_if_spam, prob_if_ham = self._probabilities(token)
            if token in text_tokens:
                log_prob_if_spam +=math.log(prob_if_spam)
                log_prob_if_ham +=math.log(prob_if_ham)
            else:
                log_prob_if_spam += math.log(1.0 - prob_if_spam)
                log_prob_if_ham += math.log(1.0 - prob_if_ham)
        prob_if_spam = math.exp(log_prob_if_spam)
        prob_if_ham = math.exp(log_prob_if_ham)
        return prob_if_spam/(prob_if_spam + prob_if_ham)
```

Testing our model

- Now we will check our model is working or not properly.

```
messages = [Message("spam rules", is_spam=True),
            Message("ham rules", is_spam=False),
            Message("hello ham", is_spam=False)]

model = NaiveBayesClassifier(k=0.5)
model.train(messages)

assert model.tokens == {"spam", "ham", "rules", "hello"}
assert model.spam_messages == 1
assert model.ham_messages == 2
assert model.token_spam_counts == {"spam": 1, "rules": 1}
assert model.token_ham_counts == {"ham": 2, "rules": 1, "
    hello": 1}
```

Testing our model

- We'll also (laboriously) go through our Naive Bayes logic by hand, and make sure that we get the same result.

```
text = "hello spam"
probs_if_spam = [
    (1 + 0.5) / (1 + 2 * 0.5),    # "spam"  (present)
    1 - (0 + 0.5) / (1 + 2 * 0.5),# "ham"   (not present)
    1 - (1 + 0.5) / (1 + 2 * 0.5),# "rules" (not present)
    (0 + 0.5) / (1 + 2 * 0.5)]    # "hello" (present)
probs_if_ham = [
    (0 + 0.5) / (2 + 2 * 0.5),    # "spam"  (present)
    1 - (2 + 0.5) / (2 + 2 * 0.5),# "ham"   (not present)
    1 - (1 + 0.5) / (2 + 2 * 0.5),# "rules" (not present)
    (1 + 0.5) / (2 + 2 * 0.5)]    # "hello" (present)
p_if_spam=math.exp(sum(math.log(p)for p in probs_if_spam)
    )
p_if_ham=math.exp(sum(math.log(p)for p in probs_if_ham))
# Should be about 0.83
assert model.predict(text)==p_if_spam/(p_if_spam+p_if_ham
    )
```

Using our model

- A popular dataset is the **SpamAssassin public corpus**. We'll look at the files prefixed with *20021010*.
- After downloading the data you should have three folders: *spam*, *easy_ham*, and *hard_ham*.
- Each folder contains many emails, each contained in a single file.
- To keep things really simple, we'll just look at the subject lines of each email.

Using our model

- When we look through the files, they all seem to start with "Subject:". So we'll look for that

```
import glob, re
# modify the path to wherever you've put the files
path = 'spam_data/*/*'
data: List[Message] = []
# glob.glob returns every filename that matches the wildcarded path
for filename in glob.glob(path):
    is_spam = "ham" not in filename
    # There are some garbage characters in the emails; the errors='ignore'
    # skips them instead of raising an exception.
    with open(filename, errors='ignore') as email_file:
        for line in email_file:
            if line.startswith("Subject:"):
                subject = line.lstrip("Subject: ")
                data.append(Message(subject, is_spam))
                break # done with this file
```

Using our model

- Now we can split the data into training data and test data, and then we're ready to build a classifier.

```
import random
from scratch.machine_learning import split_data
random.seed(0) # just so you get the same answers as me
train_messages, test_messages = split_data(data, 0.75)
model = NaiveBayesClassifier()
model.train(train_messages)
```

Using our model

- Let's generate some predictions and check how our model does

```
from collections import Counter
predictions = [(message, model.predict(message.text))for
    message in test_messages]
# Assume that spam_probability > 0.5 corresponds to spam
    prediction
# and count the combinations of (actual is_spam,
    predicted is_spam)
confusion_matrix = Counter((message.is_spam,
    spam_probability > 0.5)for message, spam_probability
    in predictions)
print(confusion_matrix)
```

Using our model

- This gives 84 true positives (spam classified as "spam")
- 25 false positives (ham classified as "spam")
- 703 true negatives (ham classified as "ham")
- 44 false negatives (spam classified as 'ham").
- This means our precision is 84 / (84 + 25) = 77%, and our recall is 84 / (84 + 44) = 65%, which are not bad numbers for such a simple model.

How could we get better performance

- Look at the message content, not just the subject line. You'll have to be careful how you deal with the message headers.
- Our classifier takes into account every word that appears in the training set, even words that appear only once. Modify the classifier to accept an optional *min_count* threshold and ignore tokens that don't appear at least that many times.
- The tokenizer has no notion of similar words (e.g., cheap and cheapest). Modify the classifier to take an optional stemmer function that converts words to equivalence classes of words.

References

[1] Joel Grus. Data Science from Scratch. First Principles with Python. Second Edition. O'REILLY, May 2019.

Thank You