lesson5_naivebayes

April 29, 2018

```
In [1]: # Run this every time you open the spreadsheet
        %load_ext autoreload
        %autoreload 2
        from collections import Counter
        import lib
```

1 Load and inspect the data

```
In [2]: # Load the data.
        # This function returns tweets and test_tweets, both lists of tweets
        tweets, test_tweets = lib.read_data()
```

2 Learn a Naive Bayes classifier

To construct our Naive Bayes classifier, we first need to calculate two things:

2.0.1 Prior probabilities of categories

We need to calculate $P(C_i)$ for each category $C_i \in \{\text{Energy}, \text{Food}, \text{Medical}, \text{Water}, \text{None}\}$.

We estimate $P(C_i)$ by $\frac{\text{\# tweets about } C_i}{\text{\# tweets}}$

2.0.2 Conditional probabilities of tokens

For each token (i.e. word) x_j and each category C_i, we need to calculate $P(x_j|C_i)$.

We estimate $P(x_j|C_i) = \frac{P(x_j \text{ and } C_i)}{P(C_i)}$ by $\frac{\text{\# tweets about } C_i \text{ containing } x_j}{\text{\# tweets about } C_i}$

```
In [3]: # Exercise 1, step-by-step version (challenge version is below).

        # The function below has two arguments: a list of tweets, and a category c
        # which is a string equal to one of "Energy", "Food", "Medical", "Water", "None".
        # The function should calculate the two things described above.
        # Fill in the blanks.


        def calc_probs(tweets, c):
            """
            Input:
                tweets: a list of tweets
                c: a string representing a category; one of "Energy", "Food", "Medical",
        "Water", "None".
            Returns:
                prob_c: the prior probability of category c
                token_probs: a Counter mapping each token to P(token|category c)
            """
```

```
    # Step 1: Calculate the total number of tweets
    num_tweets = len(tweets)

    # Step 2: Calculate the number of tweets that are about category c.
    # Save the answer to a variable called num_tweets_about_c.
    # Remember c is a string, and you can get the category of a tweet via tweet.category
    num_tweets_about_c = sum(map(lambda tweet: tweet.category == c, tweets))


    # Step 3: Calculate the probability of category c using the answers from Steps 1 and
2.
    # Hint: be careful when you divide two integers!
    prob_c = float(num_tweets_about_c)/num_tweets

    # Step 4: Create an empty Counter called token_counts.
    # (We will use it to map each token to the number of category-c tweets containing
that token.)
    token_counts = Counter()

    # Step 5 (tricky): Use a for-loop to iterate over the list of tweets.
    # Use an if-statement to check whether the tweet is in category c.
    # If it is, iterate over the tokens of the tweet (which you can access via
tweet.tokenSet) using a for-loop.
    # For each token, increment its count in token_counts.

    for tweeter in tweets:
        if tweeter.category == c:
            for token in tweeter.tokenSet:
                token_counts[token] += 1

    # Step 6: Create an empty Counter called token_probs.
    # (We will use it to map each token to P(token | category c),
    # i.e. the fraction of all category-c tweets that contain the token)
    token_probs = Counter()

    # Step 7: Now fill token_probs.
    # For each token->count in token_counts, you want to add token->fraction to
token_probs.
    # Use a for-loop over token_counts.
    # Remember that when you iterate over a dictionary/Counter, you access the keys.
    # You'll need to use the variable num_tweets_about_c.
    # Be careful when you divide integers!

    for token in token_counts:
        token_probs[token] = token_counts[token] / num_tweets_about_c

    print("Class %s has prior probability %.2f" % (c, prob_c))
    return prob_c, token_probs


prob_food, token_probs_food = calc_probs(tweets, "Food")
prob_water, token_probs_water = calc_probs(tweets, "Water")
prob_energy, token_probs_energy = calc_probs(tweets, "Energy")
prob_medical, token_probs_medical = calc_probs(tweets, "Medical")
prob_none, token_probs_none = calc_probs(tweets, "None")
```

```
Class Food has prior probability 0.47
Class Water has prior probability 0.09
Class Energy has prior probability 0.12
Class Medical has prior probability 0.04
Class None has prior probability 0.28
```

2.0.3 See what your model has learnt

In [4]:
```
# For each category c, print out the tokens that maximize P(c|token)

token_probs = {'Food': token_probs_food, 'Water': token_probs_water, 'Energy':
token_probs_energy,
               'Medical': token_probs_medical, 'None': token_probs_none}
prior_probs = {'Food': prob_food, 'Water': prob_water, 'Energy': prob_energy, 'Medical':
prob_medical,
               'None': prob_none}
lib.most_discriminative(tweets, token_probs, prior_probs)
```

```
MOST DISCRIMINATIVE TOKENS:

TOKEN               P(Energy|token)
powers              0.8029
dark                0.8029
generator           0.7654
batteries           0.7559
class               0.7534
sandysucks          0.7534
flashlights         0.7345
masks               0.7334
11/3                0.6736
cleaner             0.6707

TOKEN               P(Food|token)
canned              0.9784
non-perishable      0.9767
serve               0.9663
perishable          0.9562
cook                0.9511
soup                0.9489
sandwiches          0.9489
thanksgiving        0.9441
rice                0.9441
meal                0.9383

TOKEN               P(Medical|token)
meds                0.8229
aid                 0.8008
ointment            0.7360
prescription        0.7360
ups                 0.7360
medicine            0.7360
medications         0.7360
4t-5t               0.7360
kits                0.6596
pull                0.6596

TOKEN               P(None|token)
..                  0.9531
everyone            0.8955
last                0.8809
feel                0.8809
im                  0.8618
irene               0.8604
...                 0.8601
thing               0.8314
wow                 0.8314
tropical            0.8314
```

```
TOKEN               P(Water|token)
bottled             0.9059
gallon              0.8307
jugs                0.7970
water               0.7873
gallons             0.7266
pallets             0.6625
spring              0.6625
flood               0.6625
liter               0.6625
parks               0.6625
```

3 Build a Naive Bayes classifier

Now we've calculated $P(C_i)$ and $P(x_j|C_i)$, we can classify any tweet!

Given a tweet which is a set of tokens $\{x_1, ..., x_n\}$, the posterior probability of each category C_i is

$P(C_i|x_1, ..., x_n) \propto P(C_i) \times P(x_1|C_i) \times P(x_2|C_i)... \times P(x_n|C_i)$

We just need to calculate this for each category then determine which is largest.

```
In [5]: # Exercise 2.

        # Complete this function that calculates the posterior probability of P(c|tweet).

        def get_posterior_prob(tweet, prob_c, token_probs):
            """Calculate the posterior P(c|tweet).
            (Actually, calculate something proportional to it).

            Inputs:
                tweet: a tweet
                prob_c: the prior probability of category c
                token_probs: a Counter mapping each token P(token|c)
            Return:
                The posterior P(c|tweet).
            """

            ##### YOUR CODE STARTS HERE #####

            # Hint: first set posterior to prob_c, then use a for-loop over tweet.tokenSet
            # to repeatedly multiply posterior by P(token|c)

            posterior = prob_c
            for token in tweet.tokenSet:
                if token_probs[token] == 0:
                    posterior *= 0.001
                else:
                    posterior *= token_probs[token]

            ##### YOUR CODE ENDS HERE #####

            return posterior


        # Now you've written the function, look at the output for P(Energy|"No power in
        Riverdale").
        # What's gone wrong?
        # Try editing your function above to print out each token and token_probs[token].
        # Can you see what went wrong? How might you fix it?
```

```
riverdale_tweet = lib.Tweet("No power in Riverdale", "Energy", "need")
print("P(Energy|'No power in Riverdale') = ", get_posterior_prob(riverdale_tweet,
prob_energy, token_probs_energy))
```

```
P(Energy|'No power in Riverdale') =  2.806001890359169e-06
```

In [6]:
```
# This cell defines the classification function, that takes a tweet
# and decides which category is most likely using the posteriors you just calculated.


# OPTIONAL EXERCISE (come back to it once you've reached the end of the notebook).
# Rewrite this function to be less repetitive i.e. don't repeat things 5 times.
# There are several possible solutions; you might want to use lists or dictionaries.
# You might also want to rewrite the earlier code that computed prob_food,
token_probs_food etc.


def classify_nb(tweet):
    """Classifies a tweet. Calculates the posterior P(c|tweet) for each category c,
    and returns the category with largest posterior.
    Input:
        tweet
    Output:
        string equal to most-likely category for this tweet
    """
    posterior_food_prob = get_posterior_prob(tweet, prob_food, token_probs_food)
    posterior_water_prob = get_posterior_prob(tweet, prob_water, token_probs_water)
    posterior_energy_prob = get_posterior_prob(tweet, prob_energy, token_probs_energy)
    posterior_medical_prob = get_posterior_prob(tweet, prob_medical,
token_probs_medical)
    posterior_none_prob = get_posterior_prob(tweet, prob_none, token_probs_none)

    max_posterior = max([posterior_food_prob, posterior_water_prob,
                         posterior_energy_prob, posterior_medical_prob,
                         posterior_none_prob])
    if posterior_food_prob == max_posterior:
        return 'Food'
    elif posterior_water_prob == max_posterior:
        return 'Water'
    elif posterior_energy_prob == max_posterior:
        return 'Energy'
    elif posterior_medical_prob == max_posterior:
        return 'Medical'
    else:
        return 'None'
```

3.1 Evaluate the Naive Bayes classifier

In [7]:
```
# Compare true labels and predicted labels in a table

predictions = [(tweet, classify_nb(tweet)) for tweet in test_tweets]  # a list of
(tweet, prediction) pairs
lib.show_predictions(predictions)
```

```
<IPython.core.display.HTML object>
```

In [8]:
```
# Get average F1 score for the test set

predictions = [(tweet, classify_nb(tweet)) for tweet in test_tweets]  # maps each test
tweet to its predicted label
lib.evaluate(predictions)
```

```
Energy
Precision:  50.0
Recall:  60.0
F1:  54.54545454545455

Food
Precision:  83.56164383561644
Recall:  94.57364341085271
F1:  88.72727272727272

Medical
Precision:  85.71428571428571
Recall:  46.15384615384615
F1:  60.0

None
Precision:  82.85714285714286
Recall:  73.41772151898734
F1:  77.85234899328859

Water
Precision:  80.0
Recall:  40.0
F1:  53.333333333333336

Average F1:  66.89168191986984
```

In [9]:
```
# Get average F1 score for the TRAINING set.
# Compare with average F1 for test set above. What's the reason for the difference?

trainset_predictions = [(tweet, classify_nb(tweet))
                        for tweet in tweets]  # maps each training tweet to its
predicted label
lib.evaluate(trainset_predictions)
```

```
Energy
Precision:  91.33333333333333
Recall:  99.27536231884058
F1:  95.13888888888887

Food
Precision:  96.6355140186916
Recall:  97.91666666666667
F1:  97.27187206020695

Medical
Precision:  97.77777777777777
Recall:  100.0
F1:  98.87640449438202

None
Precision:  97.98657718120805
Recall:  94.49838187702265
F1:  96.21087314662273

Water
Precision:  100.0
Recall:  91.08910891089108
F1:  95.33678756476684
```

```
Average F1:  96.56696523097348
```

```
In [10]: lib.show_confusion_matrix(predictions)
```

```
<IPython.core.display.HTML object>
```