Naive Bayes

ref. from book **"Data Science from Scratch"**, Chap 13

```
begin
    using Test
    using Random
    using PlutoUI
    using Printf
end
```

TOC

back to TOC

The model

First we will create a function to tokenize messages into tokens.

```
tokenize (generic function with 1 method)
```

```
function tokenize(text::String)::Set{String}
    lowercase(text) |>
        split |>
        a -> filter(s -> occursin(r"\A[\w']+.?\z", s), a) |>
        a -> map(s -> replace(s, r"\A([\w']+)[^w]?\z" => s"\1"), a) |>
        Set
end
```

```
Test Passed
```

```
begin
    test_msg₀ = "Machine Leaning isn't that fun, if all one does is Data Munging!"
    test_set₀ = tokenize(test_msg₀)

    @test typeof(test_set₀) == Set{String}
    @test length(test_set₀) == 12
    @test test_set₀ == Set{String}(["is", "data", "munging", "one",
        "that", "leaning", "if", "machine", "all", "fun", "isn't", "does"
    ])
end
```

Second, let us define a structure for our messages:

```
struct Message
    text::String
```

```
    is_spam::Bool

    Message(text::String; is_spam::Bool=false) = new(text, is_spam)
```

As our classifier needs to keep track of tokens, counts, and labels from the training data, we will create a (mutable) struct for this and a collection of related functions.

```
Float64
```

```
mutable struct NaiveBayes
    k::TF
    tokens::Set{String}
    token_spam_cnt::Dict{String, Integer}
    token_ham_cnt::Dict{String, Integer}
    n_spam_msg::Integer
    n_ham_msg::Integer

    function NaiveBayes(;k::TF=0.5)
        @assert k > zero(TF)
        new(k, Set{String}(), Dict{String, Integer}(), Dict{String, Integer}(),
            0, 0)
    end
```

Next, let us define a function to train it on a collection of messages.

```
train (generic function with 1 method)
```

```
function train(nb::NaiveBayes, messages::Vector{Message})
    for msg ∈ messages
        if msg.is_spam
            nb.n_spam_msg += 1
        else
            nb.n_ham_msg += 1
        end
        for token ∈ tokenize(msg.text)
            push!(nb.tokens, token)
            if msg.is_spam
                nb.token_spam_cnt[token] = get(nb.token_spam_cnt, token, 0) + 1
            else
                nb.token_ham_cnt[token] = get(nb.token_ham_cnt, token, 0) + 1
            end
        end
    end
```

We will want to predict $P(spam|token)$. In order to apply Bayes's theorem we need to know $P(token|spam)$ and $P(token|ham)$ for each token in the vocabulary.

```
probabilities (generic function with 1 method)
```

```
function probabilities(nb::NaiveBayes, token::String)::Tuple{TF, TF}
    spam = get(nb.token_spam_cnt, token, 0)
    ham = get(nb.token_ham_cnt, token, 0)
    #
    p_token_spam = (spam + nb.k) / (nb.n_spam_msg + 2. * nb.k)
    p_token_ham = (ham + nb.k) / (nb.n_ham_msg + 2. * nb.k)
    #
    (p_token_spam, p_token_ham)
```

Finally our predict function, where to avoid numerical underflow, we will sum up the log of

probabilites instead of multiplying probabilities, before converting them back using exponentiaL.

```
predict (generic function with 1 method)
```

```
function predict(nb::NaiveBayes, text::String)::TF
    tokens = tokenize(text)
    log_prob_spam = log_prob_ham = zero(TF)
    #
    for token ∈ nb.tokens
        prob_spam, prob_ham = probabilities(nb, token)
        if token ∈ tokens
            log_prob_spam += log(prob_spam)
            log_prob_ham += log(prob_ham)
        else
            # otherwise add the log probability of _not_ seeing token in message
            log_prob_spam += log(one(TF) - prob_spam)
            log_prob_ham += log(one(TF) - prob_ham)
        end
    end
    prob_spam, prob_ham = exp(log_prob_spam), exp(log_prob_ham)
    prob_spam / (prob_spam + prob_ham)
end
```

back to TOC

Testing Our Model

```
Test Passed
```

```
begin
    test_msgs = [Message("spam rules", is_spam=true),
        Message("ham rules", is_spam=false),
        Message("hello ham", is_spam=false)
                ]
    test_model = NaiveBayes(;k=0.5)
    train(test_model, test_msgs)
    #
    @test test_model.tokens == Set{String}(["spam", "ham", "rules", "hello"])
    @test test_model.n_spam_msg == 1
    @test test_model.n_ham_msg == 2
    @test test_model.token_spam_cnt == Dict("spam" => 1, "rules" => 1)
    @test test_model.token_ham_cnt == Dict("ham" => 2, "rules" => 1, "hello" => 1)
end
```

Ok, now let's make a prediction.

```
Test Passed
```

```
begin
    test_text = "hello spam"
    d_s = (1. + 2. * .5)
    probs_spam = [
        (1. + .5) / d_s,        # 'spam' present
        1. - (0. + .5) / d_s,   # 'ham' not present
        1. - (1. + .5) / d_s,   # 'rules' not present
        (0. + .5) / d_s,        # 'hello' present
    ]
    d_h = (2. + 2. * .5)
    probs_ham = [
        (0. + .5) / d_h,        # 'spam' present
        1. - (2. + .5) / d_h,   # 'ham' not present
        1. - (1. + .5) / d_h,   # 'rules' not present
        (1. + .5) / d_h,        # 'hello' present
    ]
    p_spam = exp(sum(log.(probs_spam)))
```

```
    p_ham = exp(sum(log.(probs_ham)))
    #
    @test predict(test_model, test_text) ≈ p_spam / (p_spam + p_ham)
end
```

And next let us try on some real data.

back to TOC

Using our Model

A popular dataset is the SpamAssassin public corpus. We will look at the files prefixed with 20021010.

First let us download and untar the files

```
"./spam_data/"
```

```
begin
    const BASE_URL = "https://spamassassin.apache.org/old/publiccorpus"
    const FILE_SDIRS = [
        ("20021010_easy_ham.tar.bz2", "easy_ham"),
        ("20021010_hard_ham.tar.bz2", "hard_ham"),
        ("20021010_spam.tar.bz2", "spam")
    ]
    const OUTPUT_DIR = "./spam_data/"
end
```

```
doit (generic function with 1 method)
```

```
begin
    prepdir() = !isdir(OUTPUT_DIR) && mkdir(OUTPUT_DIR)

    function download_files()
        for (file, _sdir) ∈ FILE_SDIRS
            fp = string(OUTPUT_DIR, "/", file)
            isfile(fp) && continue
            ## Note that this function relies on the availability of external tools
            ## such as curl, wget or fetch to download the file and is provided
            ## for convenience.
            res = download(string(BASE_URL, "/", file), fp)
        end
    end

    function untar_files()
        """
        Create 3 subdirs: spam, easy_ham, hard_ham unless already created...
        """
        for (file, sdir) ∈ FILE_SDIRS
            isdir(sdir) && continue
            cmd = Cmd(`tar xjf $(file)`, ignorestatus=false, detach=false,
                dir=OUTPUT_DIR)
            run(cmd, wait=true)
        end
    end

    function doit()
        prepdir()
        download_files()
        untar_files()
    end
end
```

Next we need to load the files and to keep things simple (for now) we will just load the subject of the email...

```
cleanup_entry (generic function with 1 method)
```

```
function cleanup_entry(line::String)::String
    try
    line |>
        s -> replace(s, r"[^\w^\s^']+" => "") |>
        strip |>
        s -> filter(c -> isvalid(c), collect(s)) |>   ## pass invalid char
        join |>                                       ## re-create string
        split |>                                      ## drop word with less...
        a -> filter(s -> 2 < length(s) < 11, a) |>    ## than 2 letters and nore
        #                                             ## than 11
        a -> filter(s -> !occursin(r"\d+", s), a) |>  ## drop word with digit
        a -> join(a, " ") |>
        strip

    catch
        @warn "problem with line: <$(line)>"
        return ""
    end
end
```

```
load_data (generic function with 1 method)
```

```
function load_data()
    """
    Select only the Subject line from the message
    """
    data = Vector{Message}()
    for (root, dirs, files) ∈ walkdir(OUTPUT_DIR;
        topdown=true, follow_symlinks=false)

        for file in files
            fp = joinpath(root, file)
            is_spam = occursin(r"ham", fp)

            open(fp) do fh
                for line ∈ readlines(fh)
                    !occursin(r"\ASubject:", line) && continue
                    subject = replace(line, "Subject: " => "") |>
                        cleanup_entry
                    push!(data, Message(string(subject); is_spam))
                    break  # we are done, next
                end
            end
        end
    end
    data
end
```

```
load_full_data (generic function with 1 method)
```

```
function load_full_data()
    """
    Select Subject and Body from the message
    Body appears after Subject + 1 empty line
    """
    data = Vector{Message}()
    for (root, dirs, files) ∈ walkdir(OUTPUT_DIR;
        topdown=true, follow_symlinks=false)

```

```
        for file in files
            fp = joinpath(root, file)
            is_spam = occursin(r"ham", fp)
            found_subject = false
            body, subject = "", ""

            open(fp) do fh
                for line ∈ readlines(fh)
                    # 78line = strip(line)
                    if occursin(r"\ASubject:", line)
                        subject = replace(line, "Subject: " => "") |>
                            cleanup_entry
                        found_subject = true
                    elseif length(line) == 0
                        continue
                    elseif found_subject
                        # cumulate cleaned up line found in body
                        body = string(body, " ", cleanup_entry(line))
                    end
                end
            end
            # finally make up full message as subject + body
            msg = string(subject, " ", body)
            push!(data, Message(msg; is_spam))
        end
    end
    data
```

(3305, Main.workspace762.Message[Message(" ", true), Message(" ", true), Message(

```
begin
    data = load_full_data()
    length(data), data
```

train_test_split (generic function with 1 method)

```
function train_test_split(ds::Vector{Message};
                split=0.8, seed=42, shuffled=true)
    Random.seed!(seed)
    nr = length(ds)
    row_ixes = shuffled ? shuffle(1:nr) : collect(1:nr)
    nrp = round(Int, length(row_ixes) * split)
    (ds[row_ixes[1:nrp]], ds[row_ixes[nrp+1:nr]])
```

(2479, 826)

```
begin
    train_messages, test_messages = train_test_split(data; split=0.75)
    model = NaiveBayes()
    train(model, train_messages)
    #
    length(train_messages), length(test_messages)
```

ŷ =

Tuple{Main.workspace762.Message,Float64}[(Message("QOTD Cigarettes fast food beer Da

f₁_score (generic function with 1 method)

```
begin
    function confusion_matrix(ŷ)
        # Assume that spam_probability > 0.5 corresponds to spam prediction
        # and count the combinations of (actual is_spam, predicted is_spam)
```

```
        conf_matrix = Dict{Tuple{Bool, Bool}, Integer}()
        for (msg, spam_prob) ∈ ŷ
            #          real label   pred
            keypair = (msg.is_spam, spam_prob > 0.5)
            conf_matrix[keypair] = get(conf_matrix, keypair, 0) + 1
        end
        conf_matrix
    end

    tp(cm) = cm[(true, true)]     # True Positive
    tn(cm) = cm[(false, false)]   # True Negative
    fp(cm) = cm[(false, true)]    # False Positive
    fn(cm) = cm[(true, false)]    # False Negative

    function precision(cm)
        tp_, fp_ = tp(cm), fp(cm)
        tp_ / (tp_ + fp_)
    end

    function recall(cm)
        tp_, fn_ = tp(cm), fn(cm)
        tp_ / (tp_ + fn_)
    end

    function accuracy(cm)
        """correct predictions / total predictions"""
        tp_, tn_ = tp(cm), tn(cm)
        fp_, fn_ = fp(cm), fn(cm)
        (tp_ + tn_) / (tp_ + fp_ + tn_ + fn_)
    end

    function error_rate(cm)
        1. - accuracy(cm)
    end

    function f₁_score(cm)
        tp_ = tp(cm)
        fp_, fn_ = fp(cm), fn(cm)
        2. * tp_ / (2. * tp_ + fp_ + fn_)
    end
```

Test Passed

```
begin
    cm = confusion_matrix(ŷ)
    @test sum(values(cm)) == length(test_messages)
```

```
tp:  566 / fp:   20
tn:  124 / fn:  116
```

```
with_terminal() do
    @printf("tp: %4d / fp: %4d\n", tp(cm), fp(cm))
    @printf("tn: %4d / fn: %4d\n", tn(cm), fn(cm))
```

(precision = 0.96587, recall = 0.829912, accuracy = 0.835351, f₁_score = 0.892744)

```
(precision=precision(cm), recall=recall(cm),
```

p_spam_given_token (generic function with 1 method)

```
function p_spam_given_token(model::NaiveBayes, token::String)::Float64
    prob_spam, prob_ham = probabilities(model, token)
    prob_spam / (prob_spam + prob_ham)
```

```
spammiest_words:
                adv
          requesting
            attained
               replyb
                  lbs
           prizemama
             lenders
            overlook
                altt
               madam


hammiest_words:
        supplied
         rpmlist
       freshrpms
             rpm
        testsawl
          xpyzor
```

```
begin
    words = sort(collect(model.tokens),
        by=t -> p_spam_given_token(model, t), rev=false)

    with_terminal() do
        println("spammiest_words:")
        for w ∈ words[1:10]
            @printf("\t%15s\n", w)
        end

        println("\n\nhammiest_words:")
        for w ∈ words[end-10:end]
            @printf("%15s\n", w)
        end
    end
```

```
Set{String} with 33422 elements:
  "confined"
  "baleful"
  "archetypes"
  "piecemeal"
  "null"
  "'settle'"
  "icecreams"
  "sdk"
  "frowning"
  "henry"
  "kenbbrbr"
  "bidder"
  "midcall"
  ⋮
```