Naive Bayes Classifier

Lina Conti — M1 LI

1.

```
def generate_examples(name_directory):
    #création d'un objet path correspondant au chemin vers mon corpus
    chemin = Path(name_directory)
    #liste de tuples (liste_mots, etiquette) indiquant pour chaque mail si c'est un spam ou ham
    liste_mails = list()
    nb_spam = 0
    nb_ham = 0
    #boucle qui parcourt tous les chemins (récursivement) se finissant par .txt
    for path in chemin.glob('**/*.txt'):
        fichier = open(str(path),"r")
        contenu = fichier.read()
        liste_mots = contenu.split(" ")
        if "spm" in path.name:
            liste_mails.append((liste_mots, "spam"))
            nb_spam = nb_spam + 1
        else:
            liste_mails.append((liste_mots, "ham"))
            nb_ham = nb_ham + 1
    print("The corpus has " + str(len(liste_mails)) + " examples.")
    print("It contains " + str(nb_ham) + " hams et " + str(nb_spam) + " spams.")
    return liste_mails
```

2. The corpus has 2893 examples. It contains 2412 hams and 481 spams.

3.

```
#Separer aleatoirement la liste en trainning set et test set, la fonction retourne un tuple avec les deux sets
#Pourcentage_train doit être un float compris entre 0 et 1
def split_corpus(liste_mails, pourcentage_train):
    random.shuffle(liste_mails)
    train_set = liste_mails[:int(len(liste_mails)*pourcentage_train)]
    test_set = liste_mails[int(len(liste_mails)*pourcentage_train):]
    return (train_set, test_set)
```

4. $\log(p(y|x)) \propto \log(p(y) \cdot \prod_{i=1}^{k} p(x_i|y))$

$\Leftrightarrow \log(p(y|x)) \propto \log(p(y)) + \log(\prod_{i=1}^{k} p(x_i|y))$

$\Leftrightarrow \log(p(y|x)) \propto \log(p(y)) + \sum_{i=1}^{k} \log(p(x_i|y))$

We can use $\log p(y|x)$ in the MAP decision rule instead of $p(y|x)$ because $\log p(y|x)$ is proportional to $p(y|x)$.

5.

```
def map(list_examples):
    dictionnaire = collections.defaultdict(lambda:collections.defaultdict(float))
    for (liste_mots, etiquette) in list_examples:
        for mot in liste_mots:
            dictionnaire[mot][etiquette] = dictionnaire[mot][etiquette] + 1
    return dictionnaire
```

6.

```
class Classifier:
    # cette fonction estime les parametres du naive bayes classifier (correspond à fit)
    # elle calcule p(ham), p(spam), p(mot|etiquette) pour tout mot et etiquette
    def __init__(self, train_set):
        self.train_set = train_set
        nb_mails = len(train_set)
        # calcul nombre de hams et spams et nombre de mots dans les spams et les hams
        nb_hams = 0
        nb_spams = 0
        mots_dans_hams = 0
        mots_dans_spams = 0
        for (mail, etiquette) in train_set:
            if etiquette == "ham":
                nb_hams = nb_hams+1
                mots_dans_hams = mots_dans_hams + len(mail)
            if etiquette == "spam":
                nb_spams = nb_spams+1
                mots_dans_spams = mots_dans_spams + len(mail)
        self.log_p_ham = log(nb_hams/nb_mails)
        self.log_p_spam = log(nb_spams/nb_mails)
        # log_prob de chaque mot sachant etiquette (permet de calculer log_prob[mot][etiquette])
        self.log_prob = collections.defaultdict(lambda:collections.defaultdict(float))
        occurences = map(train_set)
        for entree in occurences.items():
            # entree[0] est le mot et entree[1]['ham'] est le nombre d'occurences du mot dans des hams
            self.log_prob[entree[0]]["ham"] = log((entree[1]["ham"]+1)/mots_dans_hams)
            self.log_prob[entree[0]]["spam"] = log((entree[1]["spam"]+1)/mots_dans_spams)

    # cette fonction predit pour une liste de mots s'il s'agit d'un spam ou d'un ham
    def predict(self, liste_mots):
        # log(p(ham|x)) ∝ log(p(ham)) + somme_pour_i_allant_de_1_a_k(log(p(xi|ham)))
        # log(p(spam|x)) ∝ log(p(spam)) + somme_pour_i_allant_de_1_a_k(log(p(xi|spam)))
        somme_p_mots_ham = 0
        somme_p_mots_spam = 0
        for mot in liste_mots:
            somme_p_mots_ham = somme_p_mots_ham + self.log_prob[mot]["ham"]
            somme_p_mots_spam = somme_p_mots_spam + self.log_prob[mot]["spam"]
        log_prob_ham = self.log_p_ham + somme_p_mots_ham
        log_prob_spam = self.log_p_spam + somme_p_mots_spam
        if log_prob_ham > log_prob_spam:
            return "ham"
        else:
            return "spam"
    # cette fonction prend en parametre une liste d'exemples annotes
    # et retourne la proportion de mails correctement predits (accuracy)
    def score(self, liste_exemples):
        # liste_exemple est une liste de tuples (mail, etiquette)
        nb_exemples = len(liste_exemples)
        res_corrects = 0
        for (mail, etiquette) in liste_exemples:
            if (self.predict(mail) == etiquette):
                res_corrects = res_corrects + 1
        accuracy = res_corrects/nb_exemples
        # le resultat est un nombre compris entre 0 et 1
        return accuracy
```

7. When running the program with the train set set to 80% of all examples I got an accuracy of 97.58%.

If we take the indicator as loss function, the test error will be $e_{test} = \frac{number\ of\ wrongly\ predicted\ emails}{number\ of\ emails}$.

Since the accuracy is given by accuracy = $\frac{number\ of\ correctly\ predicted\ emails}{number\ of\ emails}$,

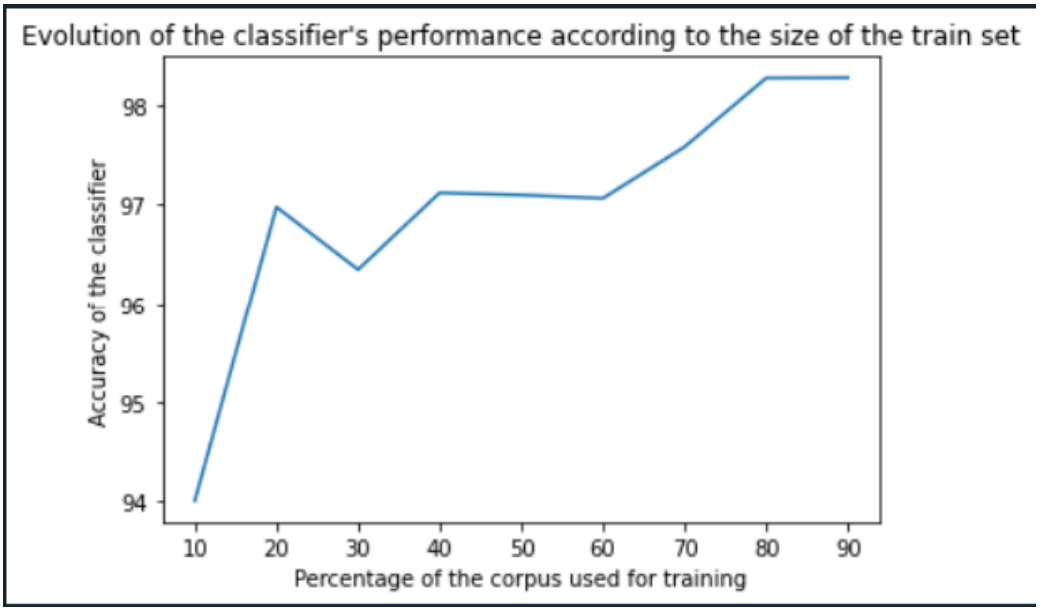
we have e_{test} = 1 - accuracy.

8.

	ham	spam
ham	477	12
spam	2	88

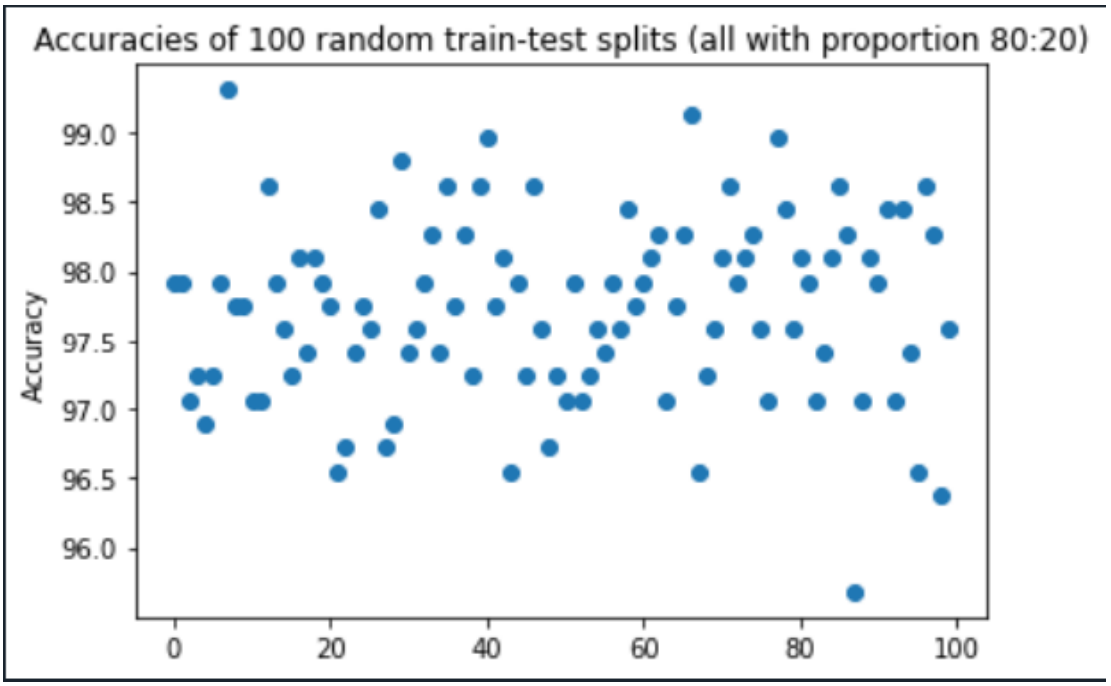
The confusion matrix gives us a better idea of where the classifier is making mistakes. This is useful to know because some mistakes are worse than others, for example, classifying ham as spam is worse than letting a spam pass through. In this case only two ham were classified as spam (which might result in the ham not being read by the user), which seems like quite a good result.

9.



The accuracy of the classifier tends to increase along with the size of the training set. This is to be expected: the more data it has to learn from, the more precise the parameters it calculates. However, when the train set is too big (90%), we sometimes get strange results because the test set is too small to be representative.

10.



The highest accuracy found was 99.13% and the lowest, 96.71%. From the plot, we can see that the accuracy of all of the classifiers is quite high and does not vary too much. The program seems rather satisfying.

11.

Ten words with the highest probability of being in ham:

1: .
2: ,
3: -
4: _
5: :
6: *
7:)
8: (

9: =
10: "

Ten words with the highest probability of being in spam:

1: _
2: .
3: *
4: -
5: ,
6: =
7: !
8: /
9: $
10: '

The words with the highest probability of being in spam correspond in many cases to the words with the highest probability of being in ham. Using the corpus lemm_stop the words found are punctuation signs rather than proper words. Maybe they should not be taken into account, since they are present in both cases and do not seem to be relevant for distinguishing spam from ham. Therefore, there is still room for improvement with this classifier.