

Stefan-Lucian Bucur
Grupa 341C4

Salut,

In acest ReadMe voi explica modul in care am implementat tema 3 , varianta C la invatare automata.

Introducere

Tema este realizata in Python, si poate rula si sub PyPy fara adaugare de module suplimentare pentru un plus de viteza.

Am implementat toate cerintele temei, care vor fi descrise mai jos.

Pentru rulare se foloseste run.sh

bash run.sh <procent antrenare> <procent de date folosit> <valoare k

Intrucat fisierele de test/antrenare sunt prea mari, si modul lor de aranjare e mai special, am hostat tema si pe github, la adresa:


<https://github.com/WindRaven/Tema3IA>

Fisiere trebuie puse in input/train.tsv pentru rotten-tomatoes si in input/mails/good_mails pentru spam/ham. **Tema nu ruleaza direct, fara copierea fisierelor necesare in directoarele care trebuie.**

Naive-Bayes

Pentru acest algoritim, am folosit o versiune specializata in analiza de text.

Ideea principala este descrisa in prezentarea de Stanford, pagina 35:



Stanford University
Natural Language Processing

Smoothing the Naïve Bayes estimates

- Laplace (add-1):
$$\hat{P}(x_i | c_j) = \frac{N(X_i = x_i, C = c_j) + 1}{N(C = c_j) + k}$$

of values of X_i
- Applied to text:
$$\hat{P}(w | c) = \frac{\text{count}(w, c) + 1}{\text{count}(c) + |V|}$$

De asemenea, am gasit un pseudocod mai explicit pe site-ul de la Trinity College

<https://www.scss.tcd.ie/~luzs/t/cs4ll4/>

Learning in multinomial models

```
1 NB-Learn( $Tr, \mathcal{C}$ )
2   /* collect all tokens that occur in  $Tr$  */
3    $\mathcal{T} \leftarrow$  all distinct words and other tokens in  $Tr$ 
4   /* calculate  $P(c_j)$  and  $P(t_k|c_j)$  */
5   for each target value  $c_j$  in  $\mathcal{C}$  do
6      $Tr^j \leftarrow$  subset of  $Tr$  for which target value is  $c_j$ 
7      $P(c_j) \leftarrow \frac{|Tr^j|}{|Tr|}$ 
8      $Text_j \leftarrow$  concatenation of all texts in  $Tr^j$ 
9      $n \leftarrow$  total number of tokens in  $Text_j$ 
10    for each word  $t_k$  in  $\mathcal{T}$  do
11       $n_k \leftarrow$  number of times word  $t_k$  occurs in  $Text_j$ 
12       $P(t_k|c_j) \leftarrow \frac{n_k+1}{n+|\mathcal{T}|}$ 
13    done
14  done
```

Note an additional assumption: position is irrelevant, i.e.:

$$P(a_i = t_k | c_j) = P(a_m = t_k | c_j) \quad \forall i, m$$

Astfel, pentru a antrena un clasificator de text Naive-Bayes, trebuie sa ii dam un dictionar cheie-valoare : (**clasa**, [**"t1, t2, ..., tn"**]), unde t1, t2, ..., tn sunt toate cuvintele textelor care au acea clasa asociata, nu neaparat distincte.

Exemplu:

P1. Obtinem listele de cuvinte cu clase asociate

"ana", "are", "mere" -> c1

"mihai", "are", "pere" -> c2

"cosmin", "are", "mere"->c1

P2. Concatenam listele de cuvinte in fiecare clasa =>

c1: "ana", "are", "mere", "cosmin", "are", "mere"

c2: "mihai", "are", "pere"

P3. Obtinem setul de cuvinte distincte:

["ana", "mihai", "cosmin", "are", "mere", "pere"]

P4. Pentru fiecare clasa obtinem probabilitatile

$$p(c1) = |c1|/nr_tot_cuvinte = 6 / 9 = 0.66$$

$$p(c2) = \dots = 0.33$$

P5. Pentru fiecare cuvant obtinem probabilitatea de a aparea intr-o clasa

C1

$$p(\text{"ana"}, c1) = (1 + 1) / (6 + 6) = 2/12 = 0.16$$

$$p(\text{"mihai"}, c1) = (1 + 1) / (6 + 6) = 0.08$$

$$p(\text{"cosmin"}, c1) = 0.16$$

$$p(\text{"are"}, c1) = 3/12 = 0.25$$

$$p(\text{"mere"}, c1) = 3/12 = 0.25$$

$$p(\text{"pere"}, c1) = 3/12 = 0.08$$

C2

$$p(\text{"ana"}, c2) = 1 / (3 + 6) = 0.11$$

$$p(\text{"mihai"}, c2) = 0.22$$

$$p(\text{"cosmin"}, c2) = 0.11$$

$$p(\text{"are"}, c2) = 0.22$$

$$p(\text{"mere"}, c2) = 0.11$$

$$p(\text{"pere"}, c2) = 0.22$$

P6. Pentru a clasifica un text, vom afla probabilitatea lui de a se afla in fiecare clasa, apoi o alegem pe cea cu probabilitate mai mare

$$t1 = (\text{"maria"}, \text{"are"}, \text{"mere"})$$

Pentru c1

$$\begin{aligned} p(t1, c1) &= p(c1) * p(\text{"maria"}, c1) * p(\text{"are"}, c1) * p(\text{"mere"}, c1) = \\ &= 0.66 * 1 * 0.25 * 0.25 = 0.04125 \end{aligned}$$

Se observa ca ignoram cuvintele care nu apar in textele de antrenare, asignand probabilitatea 1 (adica nu schimba probabilitatile finale)

Pentru c2

$$\begin{aligned} p(t1, c2) &= p(c2) * p(\text{"maria"}, c2) * p(\text{"are"}, c2) * p(\text{"mere"}, c2) = \\ &= 0.33 * 1 * 0.22 * 0.11 = 0.007986 \end{aligned}$$

Se observa ca clasa cu probabilitate mai mare este c1, ceea ce era si rezultatul asteptat.

Trebuie sa mentionez ca in codul efectiv am folosit adunare de logaritmi pentru calcularea probabilitatilor, intrucat probabilitatile mici faceau floating point overflow.

Totodata, toate probabilitatile cuvintelor au fost calculate inainte de avea nevoie de ele, astfel avand un timp de rulare foarte bun.

K-Nearest-Neighbours

Pentru algoritmul K-Nearest-Neighbours am folosit versiunea clasica, adica functioneaza cu orice clasa care are definita metoda dist, adica se poate afla distanta intre ea si un membru al aceleiasi clase.

Astfel, am folosit o clasa WordBag, care defineste un text ca un set de cuvinte:

["ana", "are", "mere", "ana"] ->(WordBag)-> ["ana", "are", "mere"]

Distanta intre doua WordBag-uri este definita ca numarul de cuvinte pe care il au in comun:

WordBag(["ana", "are", "mere"]).dist("mihai", "are", "mere") = 2

Se observa ca cu cat distanta e mai mare, cu atat seturile sunt mai apropiate (intr-un mod cam neintuitiv)

Pentru a antrena algoritmul, trebuie sa i se dea un dictionar cu cheie-valoare :(**WordBag(["t1", "t2", ..., "tn"], "clasa")**), unde t1, t2, ..., tn sunt cuvintele din text, nu neaparat distincte.

Pentru a clasifica un text, se formeaza un nou WordBag din acesta wbt = WordBag(text), dupa care se afla primele k WordBag-uri din text cele mai apropiate din acesta, si se alege clasa cea mai comuna dintre acestea

Exemplu:

P1. Obtinem listele de cuvinte cu clase asociate

t1="ana", "are", "mere" -> c1

t2="mihai", "are", "pere" -> c2

t3="cosmin", "are", "mere"->c1

P2. Formam WordBag-uri din acestea si antrenam clasificatorul

wbt1 -> c1

wbt2 -> c2

wbt3 -> c3

P3. Clasificam textul

tc = ("maria", "are", "mere")

Facem un wordbag din text:

```
wbtc = set("maria", "are", "mere")
```

Aflam distantele pentru wordbag

$d1 = 2$

$d2 = 1$

$d3 = 2$

Pentru $k = 3$, cei mai apropiati 3 vecini sunt

wbt1 -> c1

wbt3 -> c1

wbt2 -> c2

Cea mai comuna clasa (si cea aleasa) este c1, cum era de asteptat.

Bonus: Weighted K-Nearest

Am implementat de asemenea si weighted-k-nearest, adica atunci cand aflu clasa cea mai comuna, in loc sa adaug 1 la voturile acesteia, adaug distanta dintre textul cautat si aceasta

Pentru exemplul de mai sus, avem distantele

$d1 = 2, d2 = 1, d3 = 2$

Deci vecinii cei mai apropiati sunt, in ordine : wbt1, wbt3, wbt2

Astfel, pentru clasa c1 avem valoarea :

$$val(c1) = x.dist(wbt1) + x.dist(wbt3) \quad d1 + d2 = 2 + 2 = 4$$

Analog, pentru c1 avem valoare

$$val(c2) = x.dist(wb3) = 1$$

Astfel, vom alege clasa c1, dar cu o probabilitate mai mare.

Rezultate

Am calculat rezultatele impartind setul de date in 70% antrenare si 30% test, in ambele cazuri

Spam assassin

Pentru setul de date de la spam assassin, rezultatele au fost foarte bune, cu o acuratete de peste 90% in ambele variante.

Matrici de confuzile

a. Naïve-Bayes

	spam	ham
spam	1252	8
ham	26	543

Acuratete : $(1252 + 543) / (1260 + 569) * 100 \% = 98 \%$

b. K-Nearest-Neighbours

k = 11

	spam	ham
spam	1228	32
ham	26	430

Acuratete: $(1228 + 430) / (1260 + 569) * 100 \% = 90\%$

k = 2

	spam	ham
spam	1161	99
ham	76	493

Acuratete: $(1161 + 493) / (1260 + 569) * 100 \% = 90\%$

k = 7

	spam	ham
spam	1208	52
ham	79	490

Acuratete: $(1208 + 490) / (1260 + 569) * 100 \% = 92\%$

c. Weighted K-Nearest-Neighbours

k = 7

	spam	ham
spam	1213	49
ham	80	489

Acuratete = $(1213 + 489) / (1260 + 569) * 100 \% = 93 \%$

k = 20

	spam	ham
spam	1256	3
ham	183	386

Acuratete = $(1256 + 386) / (1260 + 569) * 100 \% = 89 \%$

Se observa ca pentru valori mari ale k este foarte biased spre spam, si greseste mai multe ham-uri

Rotten Tomatoes

Pe acest set, rezultatele nu au fost prea bune, acuratetea osciland intre 50% si 60%

a. Naive-Bayes

Algoritmul a scalat destul de bine, ruland in 2-3 minute pe tot setul de date

- i. 70% antrenare, 30% test - acuratete 56%
- i. 90% antrenare, 10% test - acuratete 59%
- i. 1% antrenare, 99% test - acuratete 40% (setul de date este destul de biased, cam 70 % din date sunt in clasa 2)

b. K-Nearest-Neighbours

Intrucat algoritmul nu scaleaza bine, am paralelizat codul sa ruleze pe 8 threaduri, dar chiar si asa nu poate rula pe acest set de date in mai putin de 30 de minute. Astfel, am folosit 90% antrenare, 10% pentru niste timpi de executie rezonabili.

- i. 90% antrenare, 10% test, k = 11 - acuratete 52%, timp 10 minute
- ii. 90% antrenare, 10% test, k = 200 - acuratete 53%, timp 14 minute

iii. 90% antrenare, 10% test, $k = 2000$ - acuratete 54%, timp 28 minute

Se observa ca pentru k mai mare, acuratetea creste (avand in vedere numarul mare de date in setul de antrenare), dar timpul de rulare creste destul de mult.

c. Weighted K-Nearest-Neighbours

Nu am mai avut timp sa il testez prea mult pe acest set de date, sunt o idee mai bune decat la k-nearest-neighbours.

Concluzii

Prin aceasta tema am invatat doi algoritmi simpli, dar eficienti. Avand in vedere ca knn a avut rezultate mai proaste decat naive-bayes, consider ca ar fi trebuit sa folosesc o functie de distanta mai complexa (principiul meu a fost ca daca e mai simplu va avea performante mai bune, ceea ce este semivalid).

Sper ca ReadMe-ul a fost suficient de explicit.

O zi/seara/dimineata buna,
Stefan