Alien Language Classification

- Documentatie -

Zavoiu Andreea, Grupa 252

Cuprins:

1. Cerinta. Descrierea modelelor folosite
2. Retele Neurale – Keras Sequential Model
3. Masini cu Vectori Suport - SVC
4. **Cerinta**

Antrenarea unui model pentru clasificarea textelor in 3 clase de limbaje. Setul de date de antrenare contine 10,000 de texte, cel de validare 5,000 de texte, iar cel de testare 5,000 texte.

**Descrierea modelelor folosite**

Pentru rezolvarea cerintei, am folosit multiple modele de Machine Learning, obtinand performanta cea mai mare cu Modelul Sequential din biblioteca Keras (interfata pentru Tensorflow), o performanta apropiata cu Metoda SVC a Modelului SVM, o performanta mai putin buna cu Reteaua Multi-Layer de Perceptroni (MLP) si o performanta mica cu Clasificatorul Naive Bayes Multinomial.

1. **Descrierea Retelelor Neurale**

Retelele neurale reprezinta o retea de perceptroni conectati ca nodurile unui graf, grupati in multiple straturi (layer), fiecare reprezentand, de la stanga la dreapta, input-ul pentru urmatorul strat ce calculeaza un output, folosind o functie de activare.

**Implementare**

Modelul foloseste 2 staturi adaugate prin metoda .add, primul cu functia de activare *relu*, dimensiunea output-ului de 100 de perceptroni si dimensiunea input-ului de 10,000 de perceptroni (cate texte sunt in setul de antrenare), iar al doilea cu functia de activare *softmax*, dimensiunea output-ului de 3 de perceptroni (iar dimensiunea input-ului este cea a output-ului stratului anterior, deci 100 de perceptroni). Antrenarea modelului a fost realizata pe datele din setul de training, evaluarea pe datele de validare, iar prezicerea label-urilor pe datele de testare.

from keras.models import Sequential

from keras import layers

dimensiune\_input = X\_train\_TF.shape[1]

modelSeq = Sequential()

modelSeq.add(layers.Dense(10, input\_dim= dimensiune\_input, activation='relu'))

modelSeq.add(layers.Dense(3, activation='softmax'))

Metoda compile a fost folosita pentru configurarea modelului pentru antrenare, ce primeste ca parametri: loss, optimizer si metrics. Ca functie de loss am ales *binary-crossentropy* (masoara performanta clasificarii unui model ce are ca output o valoare de probabilitate intre 0 si 1). Optimizer-ul este folosit pentru algoritmul de Gradient Descent de minimizare a loss-ului prin modificarea ponderilor din starturile de retea (weights, bias) si am utilizat Algoritmul Adam deoarece are performante mai bune. Iar metrics reprezinta modalitatea prin care masor performanta modelului, anume cu acuratetea.

modelSeq.compile(loss='binary\_crossentropy',

optimizer='adam',

metrics=['accuracy'])

Antrenarea este realizata de metoda fit ce primeste ca parametri: array-ul cu textele si array-ul cu label-urile din setul de antrenare ca input, nr de iteratii (epoci) facute pe input pentru antrenare, setarea fara verbozitate (detalii), setul pe care sa se faca evaluarea scorului adica cel de validare si, in final, numarul de texte dupa care sa actualizeze gradientul daca acesta nu s-a imbunatatitit.

modelSeq.fit(X\_train\_TF, train\_labels\_encoded, epochs=100,

verbose=False,

validation\_data=(X\_val\_TF, validation\_labels\_encoded),

batch\_size=10)

Evaluarea acuratetii se efectueaza prin metoda evaluate cu parametrii: ca input textele de validare si labelurile de validare, setare verbose fara detalii:

loss, accuracy = model.evaluate(X\_val\_TF, validation\_labels\_encoded, verbose=False)

Iar prezicerea label-urilor pentru setul de testare cu metoda predict:

predicted\_test\_labels = model.predict(X\_test\_TF)

# este nevoie de rotunjirea label-urilor din cauza ca TfidVectorizer a returnat probabilitati pentru label-urile prezise

predicted\_test\_labels\_rounded = np.around(predicted\_test\_labels)

Modelul a fost testat pe date preprocesate cu TfidVectorizer sau cu CountVectorizer, pe unele teste cu normalizare cu metrica ‘l2’ (functia din laboratorul 4) si label-uri preprocesate cu one hot encoding.

train\_labels\_encoded = np.zeros((10000, 3), dtype=int)

for i in range(0,5000):

tl = train\_labels[i]

if tl==1:

train\_labels\_encoded[i, 0] = 1

if tl==2:

train\_labels\_encoded[i, 1] = 1

if tl==3:

train\_labels\_encoded[i, 2] = 1

validation\_labels\_encoded = np.zeros((5000, 3), dtype=int)

for i in range(0,5000):

tl = validation\_labels[i]

if tl==1:

validation\_labels\_encoded[i, 0] = 1

if tl==2:

validation\_labels\_encoded[i, 1] = 1

if tl==3:

validation\_labels\_encoded[i, 2] = 1

TfidVectorizer calculeaza frecventa de aparitie a unui cuvant in texte sub forma de probabilitati, ignorandu-le pe cele foarte apropiate de 0, ceea ce ii ofera performanta la utilizare. La definirea sa, am setat pattern-ul de impartire a textului in cuvinte dupa spatii, pentru a-si putea forma vocabularul, apoi am facut antrenarea cu metoda *fit\_transform* pentru a invata vocabularul si a construi matricea cu probabilitatile de aparitie a cuvintelor, pe setul de date de antrenare. Pe seturile de validare si testare doar am actualizat matricea de probabilitati, fara a mai invata cuvinte noi, folosind metoda *transform*.

from sklearn.feature\_extraction.text import TfidfVectorizer

tfidf\_vec = TfidfVectorizer(token\_pattern = r'[^\s]+')

X\_train\_TF = tfidf\_vec.fit\_transform(train\_samples)

X\_val\_TF = tfidf\_vec.transform(validation\_samples)

X\_test\_TF = tfidf\_vec.transform(test\_samples)

Din observatiile mele, trag concluzia ca am facut overfitting.

|  |  |  |
| --- | --- | --- |
| Parametri | Acuratete Validare | Acuratete Testare |
| TF, fara normalizare  dense = 10, epochs=20, batch\_size=10 | 80.78 | 72.23 |
| TF, fara normalizare  dense = 10, epochs=30, batch\_size=10 | 80.79 | - |
| TF, fara normalizare  dense = 10, epochs=100, batch\_size=128 | 76.89 | - |
| **TF, fara normalizare**  **dense=100, , epochs=20, batch\_size=128** | **81.86** | **73.69** |
| CV, cu normalizare  dense = 10, epochs=20, batch\_size=10 | 72.10 | **73.50** |
| TF, cu normalizare  dense = 10, epochs=20, batch\_size=10 | 70.80 | 72.92 |

1. **Descrierea Masinilor cu Vectori Suport**

Este un model care are ca rol gasirea unui hiperplan de margine maxima pentru a separa elementele unor clase. Implementarea din biblioteca sklearn foloseste abordarea ONE vs. ONE, care imparte clasele in perechi si face clasificare binara pentru fiecare dintre ele.

**Implementare**

Modelul a fost testat pe date preprocesate cu clasa *Bag\_of\_words* implementata la laboratul 4, normalizate cu functia *normalize\_data* de la acelasi laborator.

# preluat din lab4

class Bag\_of\_words:

def \_\_init\_\_(self):

self.vocabulary = {}

self.words = []

self.vocabulary\_length = 0

def build\_vocabulary(self, data):

for document in data:

for word in document:

if word not in self.vocabulary.keys():

self.vocabulary[word] = len(self.vocabulary)

self.words.append(word)

self.vocabulary\_length = len(self.vocabulary)

self.words = np.array(self.words)

def get\_features(self, data):

features = np.zeros((len(data), self.vocabulary\_length))

for document\_idx, document in enumerate(data):

for word in document:

if word in self.vocabulary.keys():

features[document\_idx, self.vocabulary[word]] += 1

return features

Clasa *Bag\_of\_words* construieste vocabularul pe baza frecventei de aparitie a cuvintelor in textele din setul de antrenare, creand practic un dictionar in care cheile sunt cuvintele (pentru a nu permite duplicate). Metoda *get\_features* returneaza vectorul in care cuvintelor le-a fost asignat un index pentru care s-a pus valoarea frecventei acestora in seturile de antrenare, validare si testare, pe baza vocabularului din setul de antrenare.

bow\_model = Bag\_of\_words()

bow\_model.build\_vocabulary(train\_list)

train\_features = bow\_model.get\_features(train\_list)

validation\_features = bow\_model.get\_features(validation\_list)

test\_features = bow\_model.get\_features(test\_list)

Functia *normalize\_data* aplica normalizare cu metricile *l1* sau *l2*, pentru a avea aceeasi medie si aceeasi deviatie pe seturile de date.

# preluat din lab4

def normalize\_data(train\_data, test\_data, type=None):

scaler = None

if type == 'l1':

scaler = preprocessing.Normalizer(norm='l1')

elif type == 'l2':

scaler = preprocessing.Normalizer(norm='l2')

if scaler is not None:

scaler.fit(train\_data)

scaled\_train\_data = scaler.transform(train\_data)

scaled\_test\_data = scaler.transform(test\_data)

return (scaled\_train\_data, scaled\_test\_data)

else:

print("No scaling was performed. Raw data is returned.")

return (train\_data, test\_data)

De asemenea, si functia de calculare a acuratetii este preluata din laboratorul 4, aceasta aplicand media pe vectorul de valori boolene ce indica daca label-urile reale si cele prezise corespund (pe setul de validare).

# preluat din lab4

def accuracy\_score(true\_labels, predicted\_labels):

return (true\_labels==predicted\_labels).mean()

Am folosit Support-Vector-Classification (SVC) din biblioteca svm din sklearn, cu parametrii: C = parametru de regularizare si kernel = functia kernel utilizata de algoritm pentru a calcula performanta.

Cand C este mare, se va alege un hiperplan cu o margine mai mica, daca acesta are rezultate mai bune pe setul de antrenare, iar cand C este mic se va alege un hiperplan cu o margine mai mare, chiar daca acesta duce la clasificarea gresita a unor puncte din setul de antrenare.

Functia kernel *poly* este cunoscuta ca fiind mai putin performanta, *rbf* cea mai performanta, fiind si default pentru SVC, iar *linear* este functia kernel clasica pentru date liniare.

|  |  |  |
| --- | --- | --- |
| Parametri | Acuratete Validare | Acuratete Testare |
| **C = 12, kernel = ‘rbf’** | **71.92** | **73.03** |
| C = 12.5, kernel = ‘rbf’ | 71.88 | 72.93 |
| C = 13, kernel = ‘rbf’ | 72.7 | 14.66 |
| C = 10, kernel = ‘rbf’, normalizare cu ‘l2’ | 73.08 | 14.66 |
| C = 10, kernel = ‘rbf’, normalizare cu ‘l1’ | 72.74 | 14.66 |
| C = 25, kernel = ‘poly’ | 59.64 | - |
| C = 10, kernel = ‘linear’ | 68.12 | - |
| C = 10, kernel = ‘linear’, normalizare cu ‘l2’ | 70.52 | - |

**Matricea de confuzie** (pentru modelul cel mai performant)

confusion\_matrix(predicted\_labels\_svm, validation\_labels)

array([[1510, 337, 339],

[ 272, 1056, 131],

[ 218, 107, 1030]], dtype=int64)

