Clasificarea între dialect românesc și moldovenesc pe texte codificate

Marchidanu Adrian-Vlad, Grupa 311

05.04.2020

1 Analiza datelor

Datele se împart în date de antrenare (train), date de validare (validation) și date de test, primele două dintre acestea venind și cu etichete (0 sau 1 corespunzând câte unui dialect). Datele se prezintă sub formă de texte criptate, delimitate prin spatii și având la început un id. După procesarea inițială, inclusiv eliminarea spațiilor, obținem următoarele:

```
numarul de texte de train: 7757
numarul de texte de validare: 2656
numarul de texte de test: 2623
exemplu de text: ['|eF!', 'a:Z@', 'pg', '@<wtq', 'c=.HUz', 'fhk#', 'XBvC', 'rl#:', '|@UD', '&e', 'RElTp', 'em@#', 'op!', 'AxZF
```

Se poate observa din mai multe exemple că semnele de punctuație nu mai sunt folosite în modul uzual, ci sunt parte din schema de criptare, la fel ca literele uppercase. Deci nu are sens o conversie la lowercase sau o eliminare a semnelor de punctuație.

1.1 Statistici

Observăm distribuția claselor pe datele de train și validare (primul procent este al exemplelor din clasa 1, al doilea al celor din clasa 0).

```
distributia claselor pe train: tensor([0.5068, 0.4932])
distributia claselor pe validare: tensor([0.5102, 0.4898])
```

Nu avem probleme de balansare a claselor.

Ne uităm la lungimile textelor din fiecare dataset, apoi la media și deviația standard a acestora per fiecare dataset.

```
lungimile textelor de train: tensor([349.,  90., 169.,  ..., 785.,  88., 207.])
lungimile textelor de validare: tensor([165., 214., 205.,  ..., 165., 305., 180.])
lungimile textelor de test: tensor([124., 273., 233.,  ...,  63., 148., 211.])
media lungimilor textelor de train: 182.95204162597656
media lungimilor textelor de validare: 186.8911895751953
media lungimilor textelor de test: 176.7014923095703
deviatia standard a textelor de train: 192.9701690673828
deviatia standard a textelor de validare: 192.45263671875
deviatia standard a lungimilor textelor de test: 149.76451110839844
```

Lungimile par similar distribuite (luând în calcul Teorema Limită Centrală, care oferă predicția rezonabilă că distribuția pe cele trei dataseturi se aproprie de o normală, având în vedere numărul mare de exemple), deși deviația standard a lungimilor textelor din datasetul de test este mai mică.

Ne vom uita acum, pentru fiecare exemplu din datele de train, validare și test, la media lungimilor cuvintelor din acel exemplu, apoi la media și deviația standard a lungimilor per tot setul (pentru fiecare dintre cele trei seturi).

```
mediile lungimiilor cuvintelor per fiecare text de train: tensor([5.0172, 4.0000, 4.6333,  ..., 4.9470, 4.5000, 4.4474])
mediile lungimiilor cuvintelor per fiecare text de validare: tensor([4.3226, 4.7838, 4.1250,  ..., 4.5000, 4.6481, 4.8065])
mediile lungimiilor cuvintelor per fiecare text de test: tensor([4.6364, 3.9636, 4.6829,  ..., 4.7273, 4.2857, 4.1463])
media mediilor lungimiilor cuvintelor per text de train: 4.512233734130859
media mediilor lungimiilor cuvintelor per text de validare: 4.498715400695801
media mediilor lungimiilor cuvintelor per text de test: 4.51282262802124
deviatia standard a mediilor lungimiilor cuvintelor per text de train: 0.3247920274734497
deviatia standard a mediilor lungimiilor cuvintelor per text de validare: 0.32956647872924805
deviatia standard a mediilor lungimiilor cuvintelor per text de test: 0.3234666883945465
```

Luând în calcul Teorema Limită Centrală, lungimile cuvintelor par similar distribuite pe cele trei dataseturi.

1.2 Analiza vocabularelor

Plec de la premisa că principala caracteristică a unui dialect este prezența în el a anumitor cuvinte care nu se regăsesc în celălalt. De aceea ne uităm la vocabularele dataseturilor. Am folosit modelul din CountVectorizer pentru a obține vocabularele corespunzătoare celor trei dataseturi, precum și structura set pentru operații mai ușoare cu aceste vocabulare.

```
numar total de cuvinte din train: 32874
numar total de cuvinte din validation: 18912
numar total de cuvinte din test: 18104
numar de cuvinte care sunt in validation si nu sunt in train: 5507; procentual: 0.2911907783417936
numar de cuvinte care sunt in test si nu sunt in train: 5158; procentual: 0.284909412284578
numar de cuvinte care sunt in test si nu sunt nici in validation si nici in train: 4457; procentual: 0.24618868758285462
```

Observăm o primă problemă majoră: aproape o treime din cuvintele din dataseturile de validare și test nu apar deloc în setul de train. Asta indică faptul că predicțiile doar pe baza cuvintelor vor fi insuficiente. Pe de altă parte, dacă schema de criptare a cuvintelor nu folosește o bijecție între alfabetul normal și alfabetul de criptare, atunci prezența cuvintelor e singurul feature la care ne putem uita.

Conform presupunerii că anumite cuvinte dau apartenență la un dialect, singurele cuvinte care ar putea indica această apartenețâ sunt cele care se regăsec exclusiv în vocabularul textelor dintr-un dialect. Mai facem și următoarele observații:

```
numar de cuvinte care apar in datele de train in texte cu eticheta 0
 si nu apar in niciun text (de train)   cu eticheta 1: 11740;
 procentual din toate cuvintele care apar in train cu eticheta 0: 0.5129325410695561

numar de cuvinte care apar in datele de train in texte cu eticheta 1
 si nu apar in niciun text (de train) cu eticheta 0: 9986;
 procentual din toate cuvintele care apar in train cu eticheta 0: 0.47250875366707673

numar de cuvinte care apar in datele de validare in texte cu eticheta 0
 si nu apar in niciun text (de validare)cu eticheta 1: 6603;
 procentual din toate cuvintele care apar in validare cu eticheta 0: 0.4932397101665795

numar de cuvinte care apar in datele de validare in texte cu eticheta 1
 si nu apar in niciun text (de validare) cu eticheta 0: 5525;
 procentual din toate cuvintele care apar in validare cu eticheta 0: 0.4488585587781298

 numar de cuvinte care apar in datele de train exclusiv in texte cu eticheta 0
 si apar si in datele de validare exclusiv in texte cu eticheta 0: 1665;
  procentual din toate cuvintele de train care apar in texte cu eticheta 1: 0.07274554351625306;
 procentual din toate cuvintele de train care apar exclusiv
 in texte de train cu eticheta 0: 0.1418228279386712;
 procentual din toate cuvintele de validare care apar
 in texte de validare cu eticheta 0: 0.1243743930679017;
 procentual din toate cuvintele de validare care apar exclusiv
 in texte de validare cu eticheta 0: 0.2521581099500227


 numar de cuvinte care apar in datele de train exclusiv in texte cu eticheta 1
 si apar si in datele de validare exclusiv in texte cu eticheta 1: 1209;
  procentual din toate cuvintele de train care apar in texte cu eticheta 1: 0.05720639727453393;
 procentual din toate cuvintele de train care apar exclusiv
 in texte de train cu eticheta 0: 0.1210694972962147;
 procentual din toate cuvintele de validare care apar
 in texte de validare cu eticheta 0: 0.09822081403850841;
 procentual din toate cuvintele de validare care apar exclusiv
 in texte de validare cu eticheta 0: 0.2188235294117647
```

Observăm că sunt foarte puține cuvinte în vocabularul de train care se regăsesc numai la textele din train

dintr-un dialect, și care se regăsesc și printre cuvintele din textele de validare sau test care aparțin acelui dialect. Cu alte cuvinte, majoritatea cuvintelor din textele de validare sau test care au șanse să definească apartenența la dialect nu se regăsesc printre cele din vocabularul de train.

Mai mult, chiar dacă vom considera ca vocabular pentru encodarea cuvintelor vocabularul reunit al datelor de train, validare și test, ne așteptăm ca acele cuvinte care apar doar în test să fie învățate ca feature-uri irelevante, pentru că vor fi mereu 0 pentru textele din train și validare. Dar, după cum arată intersecțiile dintre cuvintele exclusive fiecărei clase pe train și validare, ne așteptăm ca și intersecțiile dintre cuvintele exclusive fiecărei clase pe test și cuvintele exclusive acelei clase pe train sau validare să fie mică.

Pe scurt, vor fi ignorate multe din feature-urile (cuvintele) care definesc dialectul pentru datele de test, deoarece chiar dacă le includem în vocabularul inițial, ele nu pot fi învățate ca feature-uri relevante.

Deci ne așteptăm să nu avem acuratețe excelentă, doarece datasetul de train e prea mic pentru a conține suficiente cuvinte relevante din fiecare dialect.

2 Modele încercate

2.1 Feature extraction

Feature-urile au fost extrase folosind TfIdf antrenat pe vocabularul din textele de train. TfIdf-ul generează pentru fiecare text un vector de dimensiunea vocabularului, cu conținut 0 pentru cuvintele care nu apar în text și frecvența cuvântului în tot corpusul de antrenare pentru cuvintele care apar. Am încercat de asemenea encodarea binară în loc de cea pe bază de frecvență (1 dacă apare cuvântul în text, 0 dacă nu) și nu am observat diferențe de acuratețe pe datele de validare, deci am păstrat-o pe aceasta pentru eficiență.

Am încercat și antrenarea unui model de tip word2vec antrenat pe textele de antrenare. Acesta produce un vector de o dimensiune specificată (am ales 100) pentru fiecare cuvânt, doi astfel de vectori având produs scalar cu atât mai mare cu cât tind să apară mai aproape unul de celălalt în textele de antrenare.

Problema cu această reprezentare este dimensiunea variabilă a textelor. O primă opțiune încercată a fost să păstrăm ca feature al unui text media aritmetică a tuturor vectorilor de cuvinte ai textului, însă această reprezentare nu are cum să captureze dialectul, deoarece dialectele nu sunt atât de diferite încât propozițiile în ele să creeze clustere diferite (multe cuvinte le sunt comune), deci media va ignora aberațiile date de un dialect sau altul. Indiferent de parametrii optimizatorului sau ai modelelor folosite, antrenarea cu feature-urile de text obținute astfel nu a produs nici mărcar loss convergent către 0 pe datele de antrenare, ceea ce confirmă faptul că dialectul nu poate fi reprezentat de această medie.

2.2 Modele pentru rezolvarea task-ului

Primele modele încercate sunt două rețele neurale fully-connected construite folosind pytorch, denumite ClassifierNN și DescendingClassifier. Pytorch pune la dispoziție clasa torch.nn.Module, pe care o moștenește rețeaua construită, precum și atributul requires grad pentru tensori, care, setat la True pe cel puțin un tensor implicat într-o operație, construiește automat un graf computațional al operației, ceea ce permite calculul automat al gradientului unui obiect (un loss calculat) atunci când se apelează metoda backward() a acestuia în raport cu parametrii rețelei. Pentru straturile rețelei am folosit modulul torch.nn.Linear.

Classifier NN are un strat de intrare de dimensiunea vocabularului, un număr arbitrar (2 sau 3 la instanțiere) de straturi intermediare de dimensiune specificată (am încercat cu valori $\in [10, 50, 100, 1000, 10000]$) și un strat final de dimensiune 2, cei doi perceptroni ai acestuia reprezentând probabilitățile de apartenență la fiecare clasă. Pentru a obține probabilități am folosit softmax ca activare pe ultimul strat și pentru eficiență am folosit log softmax. Ca activări intermediare am folosit ReLU pentru eficiență.

DescendingClassifier are aceleași funcții de activare, dar are o arhitectură piramidală, având un strat intermediar de dimensiune 1000 de perceptroni și încă unul de dimensiune 100.

```python
class ClassifierNN(nn.Module):
  def __init__(self, in_size: int, out_size: int, interm_size: int, no_interm_layers: int):
    super().__init__()
    #apeleaza constructorul clasei mostenite
    self._no_interm_layers=no_interm_layers
    self._layers = [nn.Linear(in_size, interm_size)]
    for i in range(no_interm_layers - 1) :
      self._layers.append(nn.Linear(interm_size, interm_size))
    self._layers.append(nn.Linear(interm_size, out_size))
    self._layers=nn.ModuleList(self._layers)
  def __call__(self, x: torch.Tensor) -> torch.Tensor:
    out = torch.relu(self._layers[0](x))
    for i in range(1, self._no_interm_layers - 1) :
      out = torch.relu(self._layers[i](out))
    out =self._layers[-1](out)
    out=F.log_softmax(out, dim=1)
    return out
```

```python
class DescendingClassifier(nn.Module):
  def __init__(self, in_size):
    super().__init__()
    self.f1=nn.Linear(in_size, 1000)
    self.f2=nn.Linear(1000, 100)
    self.f3=nn.Linear(100, 2)
  def __call__(self, x: torch.Tensor) -> torch.Tensor:
    out=torch.relu(self.f1(x))
    out=torch.relu(self.f2(out))
    out=self.f3(out)
    out=torch.log_softmax(out, dim=1)
    return out
```

2.3 Rețea recurentă

O opțiune încercată pentru a exploata totuși ideea din spatele word2vec a fost construirea unei rețele neurale recurente (care include și un strat de embedding in partea de input, strat antrenat concomitent cu partea principală a rețelei și care va ajunge să aibă aceeași funcție cu un model de tip word2vec) și procesarea textului cu aceasta, cu speranța că ultima stare ascunsă a rețelei va fi o sinteză suficient de bogată a propoziției încât să captureze și dialectul; dacă ar fi astfel, ar fi suficientă adăugarea unuia sau două straturi fully-connected peste ultimul hidden state al rețelei recurente pentru a rezolva problema de clasificare. Problema cu această abordare a fost imposibilitatea de a antrena rețeaua în timp real, dat fiind că nu se pot forma batch-uri de texte, deoarece textele au lungime variabilă și nu se pot pur și simplu trunchia (atunci se poate pierde un cuvânt important pentru recunoașterea dialectului), deci textele trebuie date rețelei unul câte unul.

Arhitectura de rețea recurentă primește la propagare un input (aici va fi un cuvânt encodat printr-un număr de la 0 la dimensiunea vocabularului) și un hidden state (0 dacă nu există hidden state anterior sau outputul rețelei pe cuvântul anterior. Mai întâi cuvântul este trecut printr-un strat de embedding (similar word2vec, dar care se va antrena concomitent cu rețeaua, și nu preantrenat), obținându-se un vector de feature-uri. Apoi acesta este trecut împreună cu hidden-state-ul primit printr-un torch.nn.GRUcell, obținându-se hidden state-ul actual. Pentru a obține o predicție a următorului cuvânt din propoziție (pe baza cuvântului primit și a hidden-state-ului primit) rezultatul acestei celule este trecut printr-un strat linear fully-connected (logits). Antrenarea se realizează pe baza comparării cuvântului care urmează în mod real în propoziție (ca vector de dimensiunea vocabularului și având 0 peste tot în afară de indicele său în vocabular) vectorul de probabilități produs de stratul linear. Arhitectura:

```python
class RNN(nn.Module):
    def __init__(self, vocab_size: int, embedding_size: int,
                 rnn_size: int):
        super().__init__()
        self.vocab_size = vocab_size
        self.embedding_size = embedding_size
        self.rnn_size = rnn_size

        self.embedding=nn.Embedding(num_embeddings=vocab_size, embedding_dim=embedding_size)
        self.rnn_cell = nn.GRUCell(input_size = embedding_size,
                                   hidden_size = rnn_size)
        self.logits = nn.Linear(in_features=rnn_size, out_features=vocab_size)
        self.softmax = nn.Softmax(dim = 2)
        self.loss = nn.CrossEntropyLoss()
```

Propagarea:

```python
def forward(self, text_vectors_raw: torch.LongTensor,
            hidden_start: torch.FloatTensor = None) -> torch.FloatTensor:
    #batch_size x trim_len x embedding_dim
    text_vectors=self.embedding(text_vectors_raw)
    # print(f'embedded shape: {text_vectors.shape}')
    #compute hidden states and logits for each time step
    hidden_states_list = []
    prev_hidden = hidden_start
    # print(f'previous hidden shape {prev_hidden.shape}')
    for t in range(text_vectors.shape[0]):
        hidden_state = self.rnn_cell(text_vectors[t,:], prev_hidden) #the t-th word of processed text i.e. t-th word vector
        hidden_states_list.append(hidden_state)
        prev_hidden = hidden_state

    #one element of hidden_states_list has shape batch_size x max_len x embedding_dim
    hidden_states = torch.stack(hidden_states_list, dim=1)
    # print(f'final hidden states shape: {hidden_states.shape} \n\n')
    return hidden_states
```

Problema principală a fost imposibilitatea de antrenare (din punct de vedere atât al timpului cât și al convergenței) având în vedere faptul că loss-ul nu se poate calcula decât pe un singur text o dată.

```
epoch 1, text 3000 has batch loss = 10.204875946044922
epoch 1, text 3100 has batch loss = 8.265103340148926
epoch 1, text 3200 has batch loss = 13.765363693237305
epoch 1, text 3300 has batch loss = 5.414170742034912
epoch 1, text 3400 has batch loss = 6.925825595855713
epoch 1, text 3500 has batch loss = 7.736629962921143
epoch 1, text 3600 has batch loss = 9.405031204223633
epoch 1, text 3700 has batch loss = 7.414373874664307
epoch 1, text 3800 has batch loss = 8.623140335083008
epoch 1, text 3900 has batch loss = 8.72674560546875
epoch 1, text 4000 has batch loss = 5.3289313316345215
epoch 1, text 4100 has batch loss = 5.360785484313965
epoch 1, text 4200 has batch loss = 3.431330442428589
epoch 1, text 4300 has batch loss = 5.838303565979004
epoch 1, text 4400 has batch loss = 2.5990829467773438
epoch 1, text 4500 has batch loss = 5.645176887512207
epoch 1, text 4600 has batch loss = 5.350670337677002
epoch 1, text 4700 has batch loss = 4.12013053894043
epoch 1, text 4800 has batch loss = 5.736607551574707
epoch 1, text 4900 has batch loss = 11.466652870178223
```

După aproximativ o oră de antrenare (5 epoci) ale rețelei recurente, am încercat antrenarea rețelei ”simple” fully-connected care ia ca input ultimul hidden-state corespunzător fiecărui text. Lossul nu converge pe datele de antrenare.

```
loss on batch 80 in epoch 52 for the simple nn is: 0.5099362134933472
loss on batch 81 in epoch 52 for the simple nn is: 0.450280100107193
loss on batch 82 in epoch 52 for the simple nn is: 0.7012249231338501
loss on batch 83 in epoch 52 for the simple nn is: 0.5384724736213684
loss on batch 84 in epoch 52 for the simple nn is: 0.5603920817375183
loss on batch 85 in epoch 52 for the simple nn is: 0.5163697004318237
loss on batch 86 in epoch 52 for the simple nn is: 0.5169010758399963
loss on batch 87 in epoch 52 for the simple nn is: 0.5013272762298584
loss on batch 88 in epoch 52 for the simple nn is: 0.3928861916065216
loss on batch 89 in epoch 52 for the simple nn is: 0.5197718739509583
loss on batch 90 in epoch 52 for the simple nn is: 0.4773578643798828
loss on batch 91 in epoch 52 for the simple nn is: 0.5114725232124329
loss on batch 92 in epoch 52 for the simple nn is: 0.5644943118095398
loss on batch 93 in epoch 52 for the simple nn is: 0.5044478178024292
loss on batch 94 in epoch 52 for the simple nn is: 0.42780178785324097
loss on batch 95 in epoch 52 for the simple nn is: 0.5225924253463745
loss on batch 96 in epoch 52 for the simple nn is: 0.4252766966819763
loss on batch 97 in epoch 52 for the simple nn is: 0.5053868889808655
loss on batch 98 in epoch 52 for the simple nn is: 0.4645036458969116
loss on batch 99 in epoch 52 for the simple nn is: 0.4712454676628113
```

Am abandonat această variantă, parțial din lipsă de timp.

3 Antrenare și rezultate

Ca metodă de optimizare a modelelor construite manual am folosit Adam, cu learning rate $\in [0.1, 0.01, 0.001]$.

Am realizat antrenarea prin împărțirea datasetului de train în batch-uri de dimensiune batch-size $\in [100, 200, 500, 800, 1000]$. Rutina de antrenare este una standard: pentru fiecare epocă de antrenare și pentru fiecare batch din acea epocă, se resetează gradienții la 0 folosind funcția predefinită din optimizator (care primește la inițializare parametrii modelului), se calculează predicțile și pierderea (folosind negative log likelihood loss, corespunzătoare folosirii log softmax ca activare pe ultimul strat), apoi se propagă înapoi gradienții pierderii, aceștia calculându-se automat în fiecare parametru al rețelei. În final se apelează funcția de step a optimizatorului, care folosește gradienții calculați pentru a minimiza funcția de cost în conformitate cu algoritmul Adam (care ține cont și pașii precedenți de optimizare etc.)

Pentru toate combinațiile de hiperparametri menționați mai sus și pentru reprezentările textelor obținute cu TfIdf binar, modelul converge pe training către loss 0. Pentru a împiedica overfitting-ul, am setat un treshold (ultimul hiperparametru) al pierderii dincolo de care modeul nu se mai antrenează. Pentru toate combinațiile de hiperparametri menționați mai sus, modelul obține acurateție în intervalul $[0.5 - 0.6]$.

Un exemplu de antrenare și evaluare a acurateții pentru un model din clasa ClassifierNN pentru parametrii:

- dimensiunea straturilor intermediare: 10000
- learning rate: 0.01
- treshold: 0.2
- dimensiunea batch-ului: 800
- 1 epocă de antrenare

```
epoch 0 has train loss: 0.6930774450302124 on batch 0 (batch_size is 800
At epoch 0 on batch 0 confusion matrix is [[392  25]
 [351  32]]
epoch 0 has train loss: 0.5369364023208618 on batch 1 (batch_size is 800
At epoch 0 on batch 1 confusion matrix is [[417   0]
 [384   0]]
epoch 0 has train loss: 0.06213144585490227 on batch 2 (batch_size is 800
At epoch 0 on batch 2 confusion matrix is [[417   1]
 [  0 384]]
```

Rezultatele pe datele de validare:

```
# converteste matricea la una densa
word_count_validation_dense=torch.Tensor(word_count_validation.todense())
output_validation=model(torch.Tensor(word_count_validation_dense))
with torch.no_grad():
  loss_validation=torch.abs(torch.argmax(output_validation, dim=1)-validation_labels)
  loss_validation=loss_validation.type(torch.FloatTensor)
print(f'Validation loss calculated with argmax is {torch.mean(loss_validation)}')
from sklearn.metrics import f1_score
print(f'Validation F1 score is {f1_score(validation_labels, torch.argmax(output_validation, dim=1))}')
from sklearn.metrics import confusion_matrix
print(f'Validation confusion matrix is {confusion_matrix(validation_labels, torch.argmax(output_validation, dim=1))}')
```

```
Validation loss calculated with argmax is 0.4100150465965271
Validation F1 score is 0.6289608177172061
Validation confusion matrix is [[644 657]
 [432 923]]
```

Ultimul model încercat este un Bayes Naiv Multinomial din biblioteca sklearn, care a produs cea mai bună acuratețe pe datele de validare.

```
from sklearn.naive_bayes import MultinomialNB
#fit prior is false because classes are uniformly distributed both on train and on validation
premodel_multinomial_NB=MultinomialNB(alpha=0.1, fit_prior=False)
premodel_multinomial_NB.fit(word_count_train, train_labels)
print(premodel_multinomial_NB.score(word_count_train, train_labels))
```

```
0.968544540415109

scorul pe datele de validare: 0.71875
scorul F1 pe datele de validare: 0.7323539949838768
matricea de confuzie pe datele de validare: [[ 887  414]
 [ 333 1022]]
```

Pentru a nu mă lovi de problema dimensionalității trecând la n-grame, am folosit ca feature-uri doar n-grame de dimensiune de la 10 la 12; MultinomialNB a produs acuratețe similară și un pic scăzută pe validare.

```
from sklearn.naive_bayes import MultinomialNB
#fit prior is false because classes are uniformly distributed both on train and on validation
premodel_multinomial_NB=MultinomialNB(alpha=0.01, fit_prior=False)
premodel_multinomial_NB.fit(word_count_train_ngrams, train_labels)
print(premodel_multinomial_NB.score(word_count_train_ngrams, train_labels))
```

```
0.9739590047698853
scorul F1 pe datele de validare: 0.7242500903505601
matricea de confuzie pe datele de validare: [[ 891  410]
 [ 353 1002]]
```

4 Concluzii

Cea mai bună acuratețe pe datele de validare a fost din păcate produsă de modelul prefabricat Naive Bayes Multinomial, cu coeficientul $\alpha = 0.01$ (am încercat ca valori ale sale cele din $[1, 0.1, 0.001]$, obținând rezultate similare). În lipsă de alte perspective, se menține ipoteza că acuratețea proastă pe datele de validare a modelului fully-connected adânc construit în pytorch este într-o măsură substanțială cauzată de lipsa prezenței suficientor cuvinte relevante pentru diferențierea dialectului în datele de train.