

Documentatie Alien Language Classification

- **Preprocesare & augmentare de date & approaches with hyperparameter proofs**

Deoarece dataset-ul nostru e format dintr-o coloana de stringuri si una de label-uri, vom fi nevoiti sa facem encoding pentru coloana de stringuri. Astfel, procesul descris aici, este utilizat pentru toti algoritmi folositi ulterior. Am aplicat constant aceste modificari de aici, pentru fiecare algoritm. Evident, ca obtiuni de encoding avem CountVectorizer, TfidfVectorizer, Tokenizer si altele. Modalitatea de formare a vocabularului difera, dar am ajuns destul de repede la concluzia ca Tokenizer si TfidfVectorizer, nu sunt utile pentru niciun algoritm in contextul asta. Initial am facut o comparatie rapida intre niste clasificari banali, de genul Naive Bayes, Random Forest, Logarithmic Regression, ANN, XGBClassifier etc, iar fara niciun parametru optimizat, am observat ca un simplu Multinomial Naive Bayes - (MNB) cu un CountVectorizer - (CV), ofera o acuratete de 68%, fara prea mult efort. Asadar, am continuat prin a ajusta parametrul de la CV, iar dupa ce am inteles documentatia, am asumat ca datorita structurii stringurilor pe care le avem in dataset, care este complet random si fara niciun pattern, ar fi mai util sa utilizam analyzer='char'. Am continuat prin a lasa calculatorul sa gaseasca un ngram_range optim si a ajuns la cateva optiuni care erau similare : (2,8), (3,7), (4,6), (5,7) etc. Mai apoi, am adaugat si binary=True pentru a avea un vocabular mai bogat un pic. Totusi, am observat numarul de caractere speciale non-ascii existente, astfel am utilizat libraria unidecode, pentru a aplica pe fiecare rand si pentru a scoate acele caractere intr-o maniera care nu va deteriora cantitatea de data pe care o avem.

Una din schemele importante este sa mutam evident, datele din validare in train, lucru pe care l-am facut manual cu copy paste in alte fisiere. Motivul e clar pentru a avea un dataset mai mare si a evita sa dau overfit, mai ales ca datasetul pe care il avem e mic. De asemenea, datele citite au fost citite cu encoding="mbcs", pentru a recunoaste caracterele speciale

Momentan, am ajuns la acest prototip:

- `from unidecode import unidecode`
- Aplicat `unidecode` pe dataset, implicit transformare din mbcs in ascii
- `CountVectorizer(analyzer='char', ngram_range=(2,8), binary=True)`
- `MultinomialNB(alpha=0.1)`
- Validation in train

Aceasta combinatie ofera in jur de 77.2% acc pe test, in sa pe train in jur de 77.6-78%.

Evident, deja am ajuns sa dau overfit si orice modificare o faceam, desi crestea trainul, scadea scorul pe kaggle. Astfel, am ajuns la concluzia ca numarul de ngrams pe caractere, produce un size al CV-ului imens, pe care l am verificat sa fie undeva intre 1.5m si 2.5m features. Asa ca am decis sa fac practic un gridsearch, pentru ngrams, `max_df` - (scoatem cuvintele cu frecvente prea mari) si `max_features` - (reducem marimea vocabularului). Am facut niste submisii de test, pentru a observa relatia dintre train si test si am remarcat ca ceva in ideea de `ngram_range=(2,8)` sau `ngram_range=(3,7)`, cu `max_df` intre 0.2 si 0.4 si `max_features` intre 300k si 800k, pare sa dea cea mai mare acuratete pe train, iar desi era slight underfit, pe kaggle obtinuse un scor in jur de 77.45%. Am verificat sa vad distributia label-urilor in dataset si prin niste experimental submissions, am observat ca intr-adevar, datasetul este balanced si cel mai important lucru este sa avem un numar minim de `max_features` si un `max_df` cat mai mic. Totusi, am remarcat un aspect interesant, cum ca daca pornesc de la `ngram_range=(3,7)` si ma duc mai in (4,4) de exemplu, rezulta un `f1_score` semnificativ diferit fata de (5,5) de exemplu. Prin asta, era evident ca desi `ngram_range`-ul de (3,7) este optim, vocabularul este poluat cu alte substringuri inutile. Am ajuns la ideea extraordinara ca cel mai bine este sa unesc mai multe `ngram_range`uri diferite, fapt care nu se poate realiza direct din CV. Astfel, am apelat la Feature Union. Prototipul anterior a devenit obsolete si am facut mai intai un gridsearch pe un singur CV si l-am ales pe cel cu `ngram_range` minim. Am ajuns la `ngram_range=(3,4)`. Totusi, era mult mai slab ca prototipul anterior, asa ca am facut gridsearch in combinatie cu cel cu (3,4) si cu un alt CV. Noul prototip functional si final a devenit:

- `CountVectorizer(analyzer='char', ngram_range=(3,4), binary=True)`
- `CountVectorizer(analyzer='char', ngram_range=(7,7), binary=True)`
- `FeatureUnion([("vec1",vec1),("vec2",vec2)])`

Pe langa sutele de incercari esuate, una dintre cele mai semnificative a fost ca am incercat sa adaug un CV, pe `CountVectorizer(binary=True,token_pattern = r"[\w\.\]{1,2}")`, unde folosesc cuvintele de lungimi 1 si 2. Desi in train si in practica, era benefic, cu cat am progresat in proiect, devenea unreliable si imposibil de spus daca este mai util sau nu, sa adaug al 3-lea CV. Asa ca am renuntat complet la acea optiune, pentru extra siguranta emotionala si redus riscul ca ma trezesc cu un overfit in final leaderboard.

In acest punct, f1_score pe train era in jur de 78.4% si pe test - kaggle 77.6%. Desi am reusit sa cresc acuratetea, evident a crescut si overfitul. Asa ca am fost nevoit intr-un final, sa fac gridsearch pe cei 2 CV finali si sa adaug max_df si max_features. Intr-o incercare de modificat min_df, s-a dovedit sa fie inutil de modificat. Dupa 1363261 submisii si incercari, am obtinut valorile finale:

- CountVectorizer(analyzer='char', ngram_range=(3,4), binary=True, max_df=0.3, max_features=79000)
- CountVectorizer(analyzer='char', ngram_range=(7,7), binary=True, max_df=0.3, max_features=500000)
- FeatureUnion([("vec1",vec1),("vec2",vec2)])

In general, max_df a fost cam mereu in jur de 0.3, deci acel aspect nu a fost o problema, doar max_features nu eram sigur daca sa pun 400k sau 80k etc. In punctul asta, f1_score pe train era in jur de 77.2% si pe test 77.89% cu MNB. Evident, slight underfit, dar nu era o alternativa mai buna, asa ca m-am decis ca acest prototip este bun. Am incercat si sa elimin niste substringuri de lungimi mici din text, doar ca sa mai cresc putin scorul si am eliminat substringuri, pana cand dadea overfit. Am testat toate acestea prin citirea de validation in train, deci cu toate datele si dat

- Train_test_split pe 0.06, ca sa pot simula relativ cele 1000 de date pe care sunt testate public leaderboardul. Am facut asta ca sa fiu sigur ca varianta dintre ce e pe public si pe private, nu o sa fie foarte mare.

Deci, concluzia este ca pentru data preprocessing, am folosit urmatoarea configuratie:

- eliminare de substringuri
- unicode
- Validation + train
- CountVectorizer(analyzer='char', ngram_range=(3,4), binary=True, max_df=0.3, max_features=79000)
- CountVectorizer(analyzer='char', ngram_range=(7,7), binary=True, max_df=0.3, max_features=500000)
- FeatureUnion([("vec1",vec1),("vec2",vec2)])

Tin sa precizez ca fiecare prototip prezentat mai sus, a fost testat pe majoritatea cazurilor relevante cu diferiti algoritmi, pana am ajuns la un consens ca in general, daca un algorithm da mai bine pe un anumit prototip, va da mai bine pe clasificarii.

Totodata, extrem de important, cele mentionate deasupra sunt un oversimplification imens la fiecare cifra testata, dar ar lua mai mult de 5 pagini, doar sa descriu fiecare cifre care au putut fi considerate pentru submitii.

Din acest moment, orice este precizat aici, classifier, teste, incercari etc, utilizeaza exact prototipul anterior cu FeatureUnion si nicio alta combinatie. Scopul in a cauta un numar minim de features a fost exact pentru a reduce la minim overfit si de a mentine o consistenta peste toti algoritmi incercati, pentru a garanta perfect fit si de a da match algoritmi intre ei, pentru pasul urmator, fara a risca perturbarea interactiunilor intre ei.

Pana acum, relevant a fost doar MNB, toate submitiile au fost doar pe asta, ca sa pot da tuning ca lumea. Reluand pe scurt de ce sunt buni acesti parametri, max_df si max_features sunt extrem de importanti pentru a avea overfit minim, daca am lasa default analyzer='word', nu retine cum trebuie, doar pentru ca datasetul e prea random, asa ca pe 'char', e mai accurate. Ngram_range n-are nicio logica, e doar trial and error. De fapt tot proiectul e trial and error, un sample e o combinatie random de caractere ascii si non-ascii si aparent ar trebui sa gasesc o explicatie pentru parametri la CV, intr-un dataset cu o coloana si 3 label-uri.

Singura logica pe care pot sa o zic legat de ngram_range, e ca am pornit de la (3,7) si cu ce am precizat mai sus, evident ca in rangeul asta, o sa fie intervalele potrivite, deci (3,4) si (7,7) erau cele optime.

In continuare, urmatorul pas de a ajunge la 79% pe test, a fost unirea la mai multe csv-uri de la algoritmi diferiti. Pattern-ul evident e ca trebuie sa fie minim 3 algoritmi, toti trebuie sa dea o acuratete pe train de macar 76-77% si sa aibe mecanisme de functionare complet diferite. In acest sens, utilizarea unui ANN si un MLP simultan, nu ar fi productiv, deoarece sunt foarte similare, la fel putem spune si de comparatii de genul MNB si SVC etc. Din fericire, nu prea au fost asa multi algoritmi care sa si indeplineasca toate conditiile, asa ca lista relevanta a fost:

- ANN, MNB, MLP, RFC, LR, XGBClassifier
- Evident, o combinatie de 3+ algoritmi dintre acestia, care sunt si diferiti
- Nu am mai adaugat algoritmi pe langa de gen CategoricalNB, LinearSVC, doar pentru ca erau identici cu celelalte optiuni si dadeau scor mai mic

Fast forward, combinatia potrivita era ANN, MNB, XGBClassifier. MLP era un contender decent, dar nu destul de bun comparativ cu un ANN, setat de mine cum vreau.

Ca idee, am ales ca al doilea submission pentru private sa fie o combinatie de 5 algoritmi, cei de sus, dar fara MLP, insa a fost extrem de prost, deci doar voi mentiona ca am utilizat si LR si RFC, dar realistic, e doar un unsuccessful attempt.

Desi acum avem combinatia de ANN, MNB, XGBClassifier. A aparut o problema finala - cum vom alege label-ul in caz incert. Evident, metoda logica de a uni cele 3 csv-uri de algoritmi, era sa preluam cel mai comun label intalnit pe fiecare rand. De aceea a fost si mult mai util sa folosesc 3 algoritmi, fata de 4 sau 5, unde apar mai multe cazuri de incertitudine, rezultand astfel intr-o nevoie de algoritmi cu scor mai mare si cat mai diferiti - imposibil de realizat in 3 zile ramase. Revenind, in cazul cu 3 algoritmi, avem mai multe combinatii posibile de label-uri, dar cel problematic e atunci cand fiecare algoritm alege un label diferit, adica ANN - predict 1, MNB - predict 2, XGB - predict 3. Deci [1,2,3] case. Am avut doua metode de a face asta, una cu median si una cu mean. Desi median nu ar trebui sa ne ajute in caz de frecventa a unei cifre, fiindca aveam doar 3 algoritmi, mediana ar avea comportament identic cu mean. Asa ca am testat mediana, iar in caz de exceptie - [1,2,3], ar returna pe rand mereu 1, 2, 3, random sau de la un algoritm anume. S-a dovedit ca atunci cand returnezi mereu 2, este exceptional mai performant. Cu mean, am folosit `weighted_mean`, am incercat posibilitatile, iar `weights = [2,3,2]`, in exact ordinea ANN, MNB, XGB, a fost cel mai performant. Aceasta combinatie a fost dovedita ca fiind mai buna chiar si decat `median return 2`, deci prototipul final este:

- `FeatureUnion + Ann, Mnb, Xgb (weights=[2,3,2])`

Motivul pentru care `weights`-urile sunt asa, e probabil pentru ca MNB avea cea mai mare acuratete pe train si test si e mai relevant.

● Multinomial Naive Bayes

Sunt mai multe optiuni de ales si chiar daca nu aveau sens altele, am ales sa incerc si Bernoulli, Categorical, Complement si Gaussian NB. Totusi, toate au dat semnificativ mai rau. MultinomialNB are design specific pentru dataseturi de genul nostru, cu stringuri si se descurca bine cu CV. Aici am utilizat doar `alpha=0.1`, doar pentru ca nu e prea customizable. Cifra a fost obtinuta prin gridsearch si evident, in general `alpha` e bine sa fie cat mai mare, pentru a compensa probabilitatile de 0, in caz ca nu intalnim cuvantul cautat in train. Totusi, datorita formatului nostru, cu numeroase cuvinte intamplatoare si fiindca iau multe ngrams, este mai benefic sa utilizam un `alpha` mic, gasit random. Acest algoritm a fost utilizat in forma finala a prototipului. Ca majoritatea algoritmilor simplii, MNB merge deosebit de bine pe dataset-uri mici, fiind destul de evident ca e o optiune buna.

- `MultinomialNB(alpha=0.1)`

Multinomial Naive Bayes (alpha=0.1) + unidecode + string removal			
CV(binary=True)	Train	Public	Private
None	0.685	0.684	0.69
'Char', (3,7)	0.76	0.75	0.744
'Char', (3,7) +val	0.77	0.7737	0.7666
'Char', (3,4), (7,7), 0.3, 590k total + val	0.772	0.778	0.769

Putem observa ca legat de limba 1, nu se descurca asa bine ca si ceilalati algoritmi.

● XGBClassifier

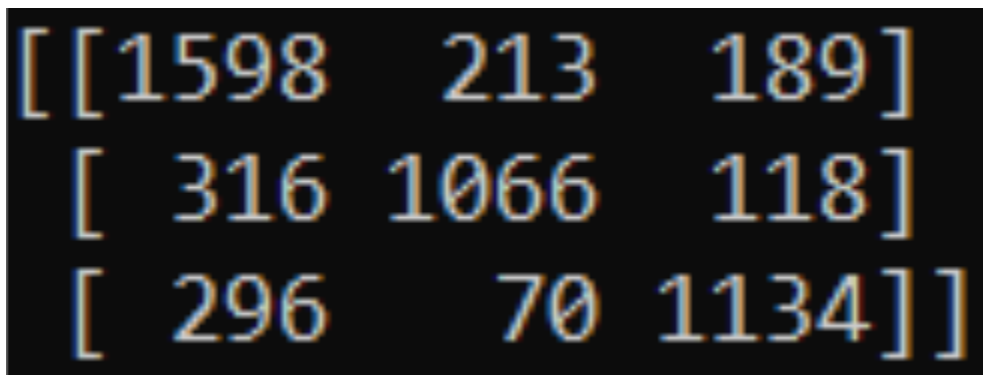
Face parte din XGBoost, iar acesta functioneaza foarte bine cand numarul de observatii e mai mare decat numarul de features. In cazul nostru, evident, avem 15000 de observatii si `max_features=579k`. Ar parea la prima impresie ca nu are rost nici sa consideram acest algoritm. Totusi, utilizam stringuri - object type. Acestui algoritm nu i-am dat submitie individuala, deoarece nu ne intereseaza scorul pe care il scoate neaparat. Desi pe train a dat in jur de 76%, ceea ce e decent, tot e subpar cu MNB si ANN cu 78% aproape pe test. Cel mai important criteriu in a alege cei 3 algoritmi de combinat, este sa fie cat mai diferiti. Utilizand un model naiv si banal, ca naive bayes, un neural network si adaugand dupa un classifier cu un design bazat pe gradient-boosting si decision tree, ar fi ca si cum unim un algoritm de genul RFC, DTC si unul pe gradient, toate in unul singur. Astfel, acceptam un scor mic, dar completitudinea pe care ne-o ofera XGBClassifier, o face sa devina element principal de fapt, in combinatia noastra. Functioneaza pe acelasi principiu cu un ETF - index fund, unde in loc de adaugat 500 de companii, adaugam o diversitate de algoritmi, pentru a forma o combinatie optima. Este o piesa extraordinara in mixul pe care il avem, mai ales datorita laturii de gradient boosting, care optimizeaza loss-ul, implicit scade numarul de erori. Noi totusi avem nevoie de algoritmi care sunt simplii in idee, datorita datasetului cu informatii complexe. Astfel, XGB actioneaza prin

utilizarea unei multitudini de algoritmi banali. Prin folosirea combinatie de ANN, MNB, XGB, am avea un debate daca e posibil macar, sa folosim mai mult de 3 algoritmi, cand deja avem acoperite atat de multe categorii de classifieri.

Ca si hiperparametrii relevanti, avem `objective = 'multi:softprob'`, care adapteaza algoritmul nostru pentru multiclass classification, iar `n_estimators = 500`, un numar care evident, reprezinta numarul de boosting rounds, cu scopul de a reduce bias-ul si loss - erori care pot aparea. Fiind un dataset atat de randomizat si cu atatea optiuni de repetitie a ngram-urilor, un numar mare de estimators, ajuta in a ajunge la un scor mai bun. Totusi, daca crestem peste, vom avea overfit, deci e nevoie un perfect balance. Am utilizat XGB cu varianta de histograme, doar pentru ca profita atunci cand avem numar mic de features, iar de aceea ne-a ajutat semnificativ sa reduc `max_df` si `max_features` la CV. In general XGB merge bine pe dataset mic, iar de aceea a ajutat mult sa folosim asta.

- `XGBClassifier(objective='multi:softprob', n_estimators=500)`

XGBClassifier (n_estimators=500, objective='multi:softprob') + rest			
CV(binary=True)	Train	Public	Private
'Char', (3,4), (7,7), 0.3, 590k total	0.759	-	-



Ca si la ANN, predictul pt limba 1 e foarte bun si completeaza frumos rezultatele de la MNB.

● Artificial Neural Network

Aici am avut optiunea de a opta pt un ANN sau pentru un MLP. Totodata, nu am inteles de ce cineva ar opta macar pentru un MLP si nu ar face direct manual un ANN. Totusi, e destul

de clar ca un MLP nu ar avea nicio utilitate la noi. Tot scopul la problema asta e sa folosesti clasificari cu design cat mai rudimentar, mai ales din cauza numarului minim de date. MLP foloseste mai multi perceptroni si realizeaza un neural network foarte complex. Nu avem nevoie de asa ceva pentru cazul nostru. In cazul nostru, deja avem un classifier liniar, gradient boosting, decision tree, practic acoperim cam toate tipurile de clasificatori. Singurul lucru ramas e sa vedem daca CNN, RNN, LSTM etc e cel mai eficient. Desi era evident ca niciunul n-ar functiona, deoarece ar fi mult prea complexa orice varianta de genul, am facut totusi un test ca sa vad ca da 50% val_f1_score, deci da. Totusi, in combo-ul de 3 algoritmi, pana acum avem combinatia perfecta si lipseste doar un ANN, asa ca am zis totusi ca o idee ar fi doar sa fac cel mai banal neural network posibil. Asa ca, am ajuns la aceasta forma:

```
train_data=train_data.todense()
validation_data=validation_data.todense()
test_data=test_data.todense()
for i in range(len(validation_labels)):
    validation_labels[i]-=1
for i in range(len(train_labels)):
    train_labels[i]-=1
inp=579000
model=Sequential()
model.add(Dense(50, input_dim=inp, activation='relu'))
model.add(Dense(25, input_dim=inp, activation='relu'))
model.add(Dense(3, input_dim=inp, activation='softmax'))
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
def lr_scheduler(epoch, lr):
    if epoch < 50:
        lr = 0.05
    return lr
return lr
model_checkpoint = ModelCheckpoint('best.hdf5', monitor='val_accuracy', mode='max',
save_best_only=True, save_weights_only=True)
callback=[model_checkpoint,tf.keras.callbacks.LearningRateScheduler(lr_scheduler)]
model.fit(train_data, to_categorical(train_labels), epochs=1,
batch_size=32, validation_data=(validation_data, to_categorical(validation_labels)), callbacks=callback)
model.load_weights('best.hdf5')
print(model.evaluate(validation_data, to_categorical(validation_labels)))
preds=model.predict(test_data)
```


Este cat se poate de banal, avem doar 2 layere de dense si unul de activare, adica nu prea e nimic de explicat, tot scopul era sa fie cel mai basic model sequential posibil. Am facut hot-encoding la labels. Adica sincer sa fiu, poate ar fi mers si mai bine daca lasam doar layerul de activare, dar n-am mai avut timp sa testez. Am crescut cat mai mult learning_rate-ul si am pus pe o epoca si am luat un numar mediu de batch_size, doar ca un shotgun method. Daca as lasa asa modelul, dupa a 2a epoca, chiar si cu un learning_rate mic, ar da overfit instant, asa ca am concentrat tot learningul in prima epoca. Astfel, cu varianta asta (cu algoritmi combinati), a iesit 79.4 pe public si 78.9 pe private. Insa, in punctul asta, eram pe locul 3 si nu eram sigur daca o sa cresc sau scad, de la overfit. Asa ca am zis ca poate trebuie sa cresc scorul la ann, sa-l fac un pic mai complex. Cel anterior dadea 75% pe train, ceea ce era foarte putin. Asa ca am zis sa adaug 20 de epoci si sa pun:

- lr=0.0005
- model.add(Dropout(0.8))
- model.add(Dropout(0.3))

Cu toata scamatoria asta, am scos 79.63 pe public si eram primul, si am zis ca ok, nu are cum ANN-ul anterior sa dea mai bine, din moment ce ar trebui sa fie direct improvement. Dar aparent versiunea cu dropout a dat doar 79.39 pe private si am castigat la limita. Era deja ultima zi si n-am mai gandit ca lumea, dar era evident ca daca cresc numarul de epoci si pun dropout, doar fac prea complex ANN-ul si ar trebui sa dea overfit. Ar fi trebuit sa aleg ca a 2-a optiune, ANN-ul anterior, dar am zis ca poate ala cu 5 algoritmi, o sa dea ok, dar nu a fost asa, bine ca am avut noroc, oricine care alegea combinatia potrivita, si daca avea 78.5 pe public, putea sa iasa peste submisia mea de 1.

ANN + unicode + string removal			
CV(binary=True)	Train	Public	Private
'Char', (3,4), (7,7), 0.3, 590k total + val	0.75	0.778	0.768
'Char', (3,4), (7,7), 0.3, 590k total + val + dropout + lr	0.76	-	-

[[1590 200 210]
[391 987 122]
[244 81 1175]]

- **Random Forest Classifier & Logistic Regression**

Nu am sa dau mai multe informatii, decat ca le-am pus intamplator cu realistic niciun hiperparametru, doar ca sa am o submitie secundara pentru kaggle.

Combined classifiers table			
‘Char’, (3,4), (7,7), 0.3, 590k total + val	Train	Public	Private
ANN, MNB, XG, LR, RFC	0.771	0.79	0.784
ANN(dropout+lr), MNB, XG, LR, RFC	0.771	0.79	0.78
ANN, MNB, XG	0.773	0.794	0.789
ANN(dropout+lr), MNB, XG	0.773	0.796	0.7839
MNB, MLP, XG	-	0.785	0.784

Este greu de crezut ca ar fi dat un scor mai bun cu alti algoritmi pentru >3, pentru ca deja s-ar fi suprapus arhetipurile de classifier intre ei. Putem remarca oricum greseala semnificativa pe care am facut-o, cand am ales a 2-a si a 4-a submitie ca cele finale, fiind motivat de avansarea de pe locul 3, pe locul 1. Daca ne imaginam cazul in care sunt pe locul 3, in stresul din ultima zi, e cam greu sa alegi prima si a 3-a submitie, in ideea ca da overfit de la prea multe epoci, pana si pe public. Totusi, un compromis decent, era sa aleg a-3a si a 4-a submitie, care ar fi fost mult mai logic, avand in vedere ce am zis pana acum, dar stresul de la final, nu a lasat neaparat sa gandesc logic ca ar da semnificativ mai bine optiunea precedenta de ANN. Era bias-ul ca poate totusi combinatia de 5 algoritmi, ar da mai bine, dar s-a dovedit ca ar fi trebuit sa ma iau dupa ce e logic, anume ca se suprapuneau arhetipurile. In final, e bine ca am avut noroc, probabil si restul colegilor au fost stresati in ultima zi, dar am adaugat ca un ultim rand, o optiune cu MLP, care

dadea foarte putin pe public, dar destul de luat locul 1 pe private. Avand in vedere ca primele 6 persoane au avut peste 0.785 pe public, intr-un caz anume, era foarte posibil sa adauge, cel putin ca optiune secundara, o submisie de locul 1. Singurul lucru pe care ar putea cineva sa il invete de aici, ar fi poate ca nu ai cum sa castigi, fara un pic de noroc si ca toata lumea e supusa stresului si in final, desi minim 5-10 persoane ar fi avut un submission destul de bun de locul 1, ca si mine, din cauza circumstantelor, nu au fost capabili sa aleaga cel mai bun submission.