

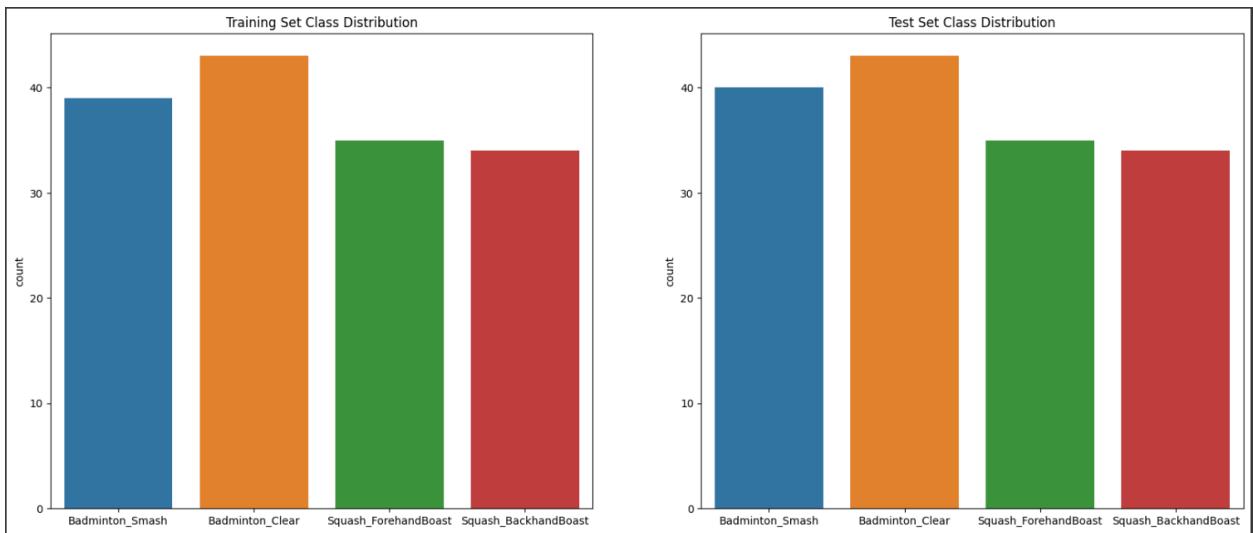
Raport

Matei Vlad Cristian 342C4

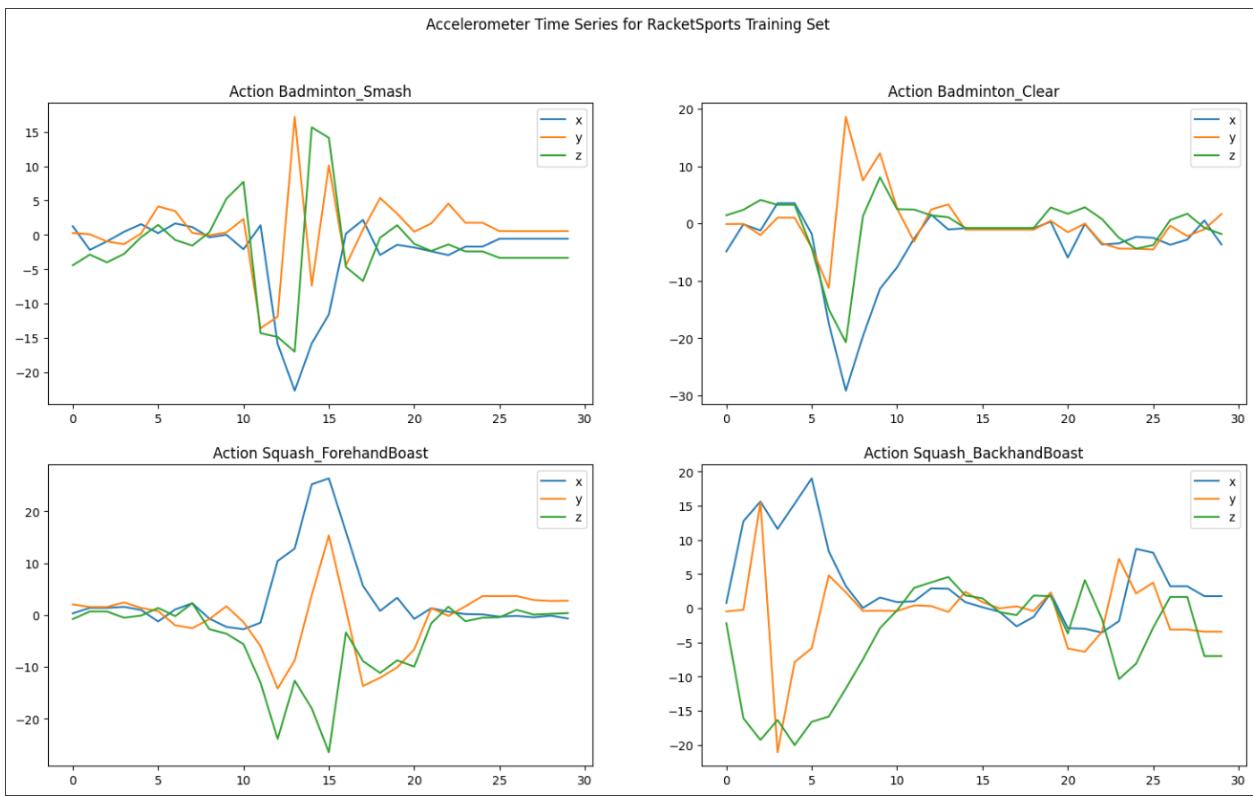
!! Raportul este insotit de fisierul Proiect_ML.xlsx care contine rezultatele obtinute !!

RacketSports

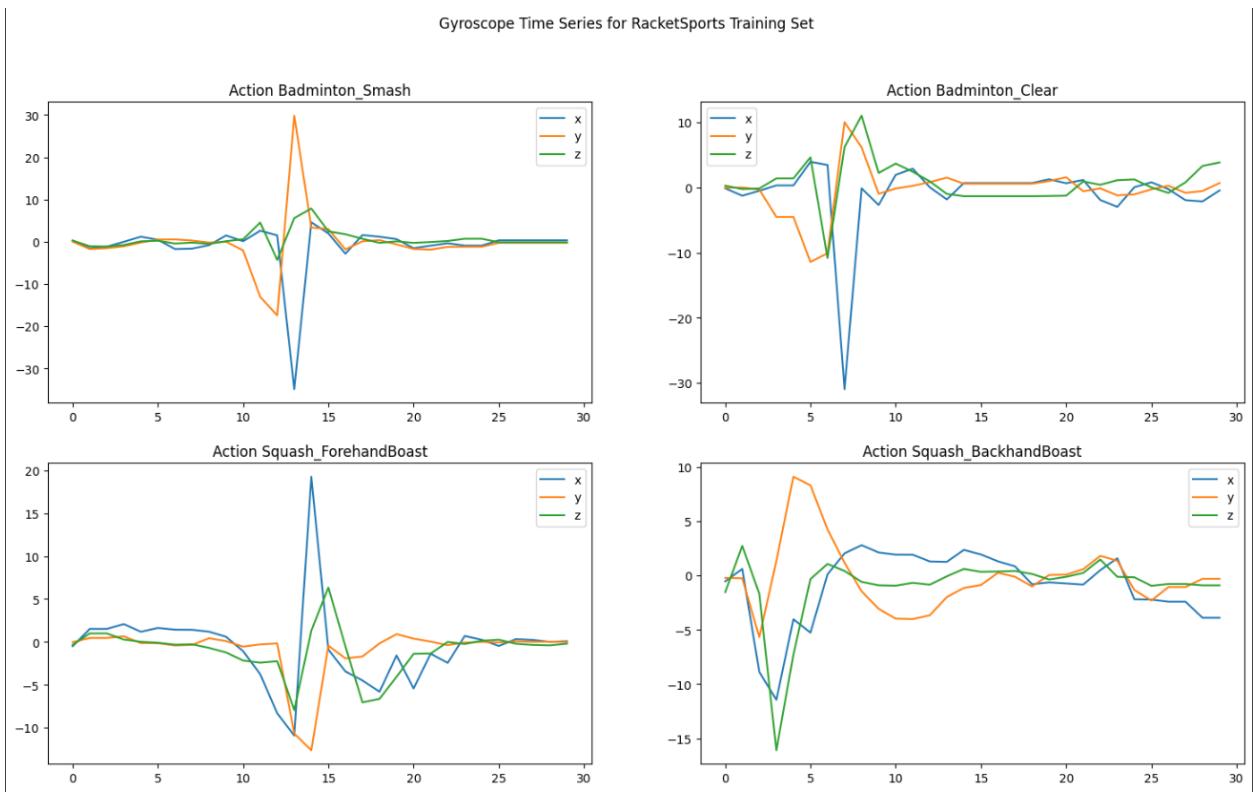
1. Explorarea datelor



Așa cum se poate vedea, setul de date este echilibrat. Cele 4 categorii de acțiuni care pot fi clasificate contin număr aproape egal de exemple. De asemenea, setul de date prezintă aceeași pondere din fiecare categorie între seturile de date Training/Test. Acest lucru este necesar pentru a avea o antrenare eficientă.



Acest grafic prezinta cate o actiune, aleasa random din dataset, pentru fiecare categorie si arata valorile pe fiecare axa. In acest grafic sunt prezente doar valorile acceleratiei. Se pot vedea diferente destul de mari intre diferitele actiuni. Analizand aceste diferente se poate enunta faptul ca si modelul le va evidenta, astfel concluzia ar fi ca problema poate fi rezolvata printr-un algoritm / model de ML. Nu doar ca ar putea fi rezolvata, dar cred ca ar fi si o problema usor de rezolvat, tinand cont atat de distributia valorilor / axa / actiuni si faptul ca setul de date este echilibrat.

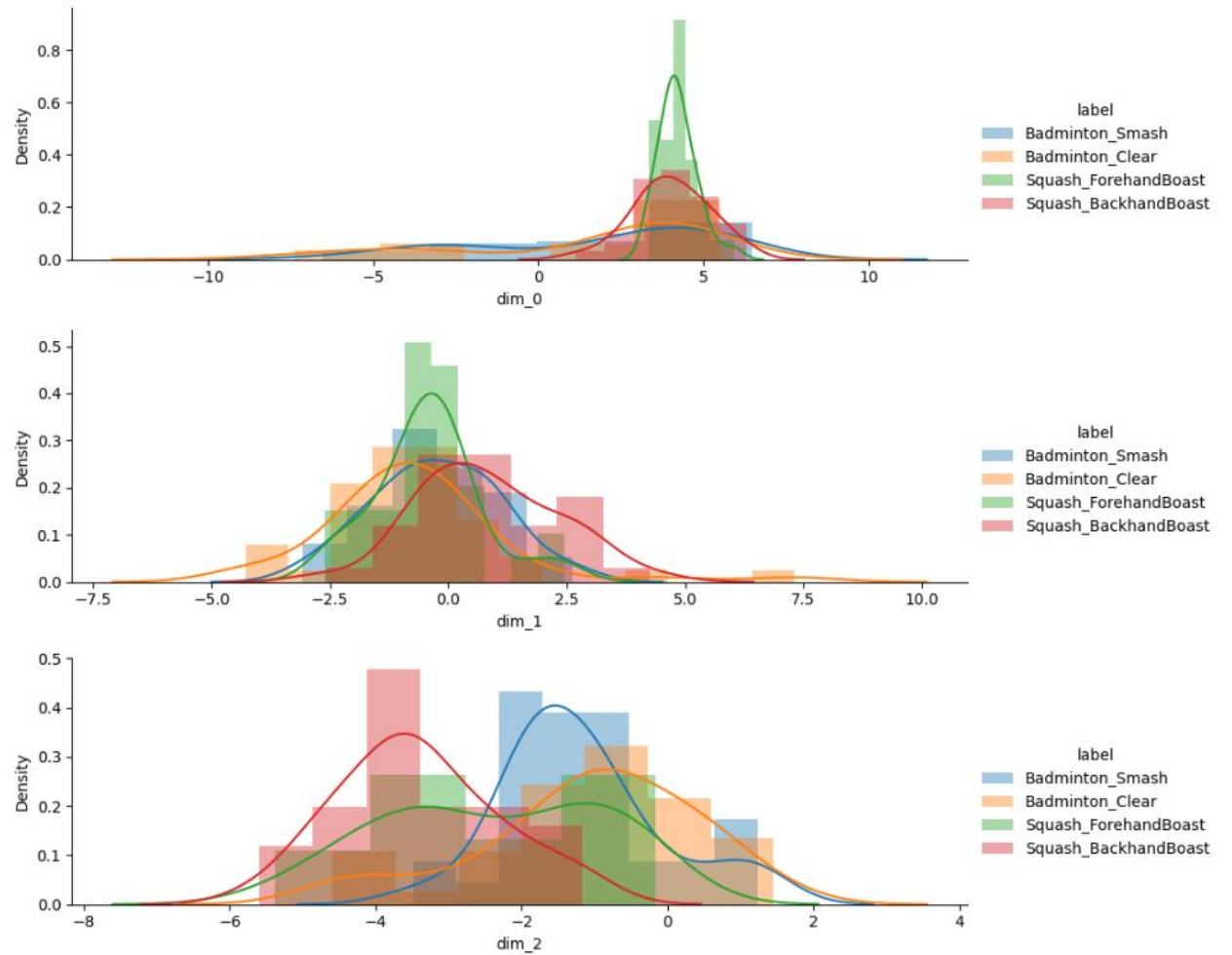


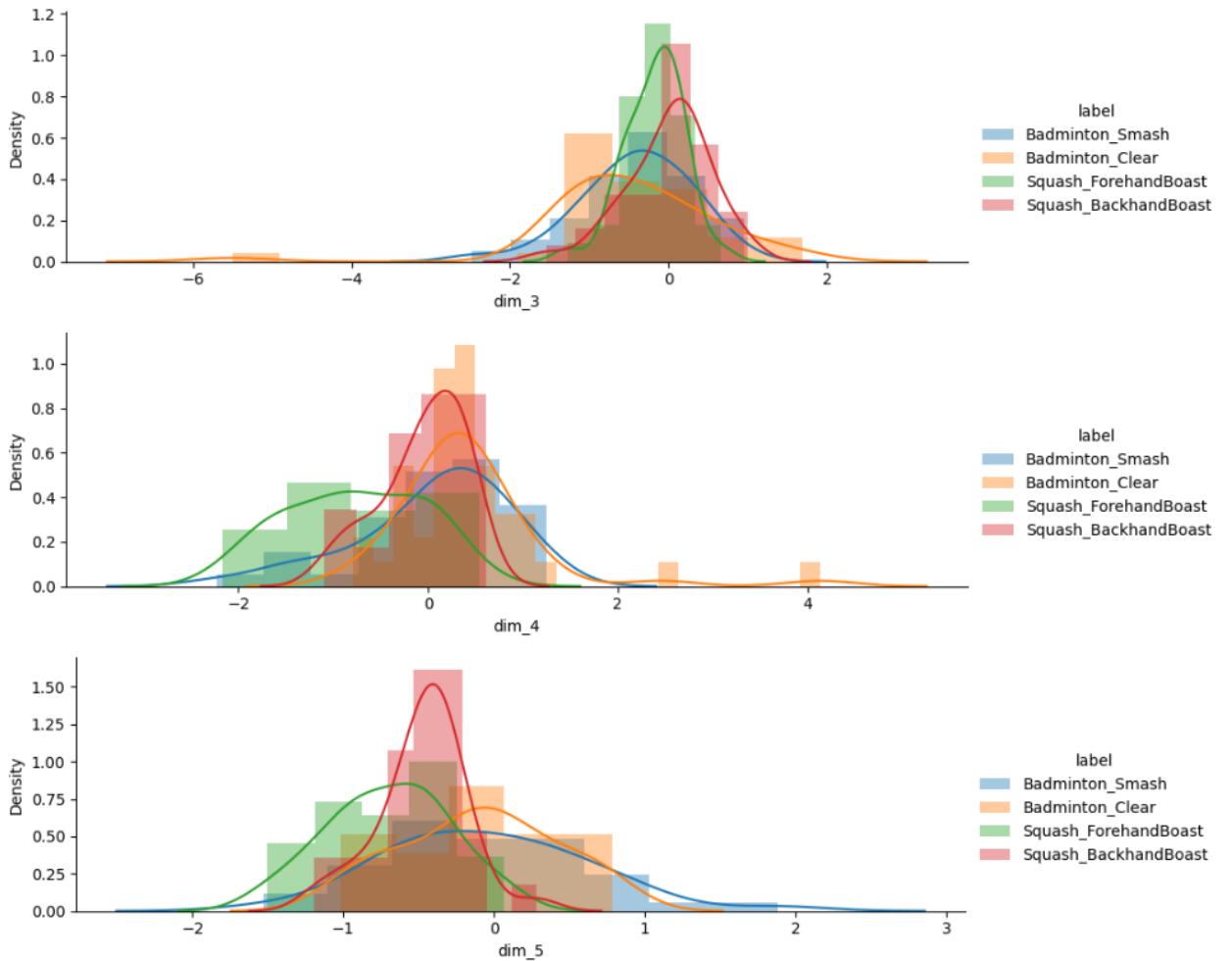
Să în privința giroscopului există diferențe accentuate. Acțiunea ‘Squash_BackhandBoast’ pare să fie cea mai diferită față de toate celelalte, asadar așteptarea este că ea să fie cel mai ușor de clasificat, chiar dacă are un număr mai mic de exemple în dataset. (nu foarte mic)

Prezentarea valorilor medii / unicat pe axe în funcție de acțiune

! Notația $\dim_{\{i\}}$ reprezintă o axă. Valorile lui $i = \{0, 1, 2\} =$ acceleratie, $\{3, 4, 5\} =$ giroscop.

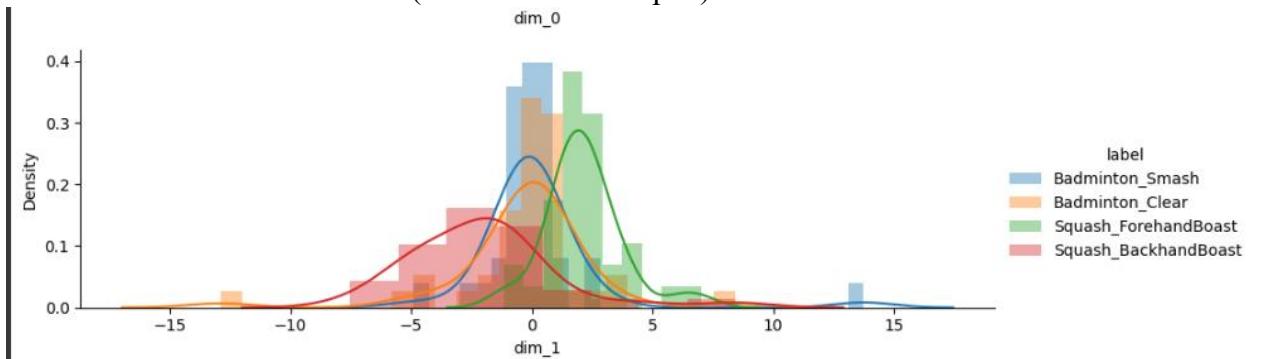
- Valorile medi



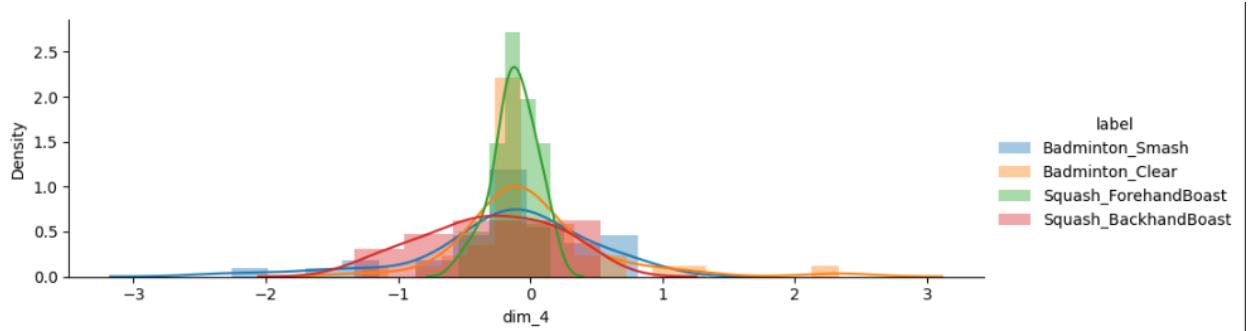


Pe dim_0 si dim_5 se observa cat de bine se evidențiază acțiunea ‘Squash_BackhandBoast’. De asemenea, se poate spune că acțiunea ‘Squash_ForehandBoast’ se evidențiază foarte bine pe dimensiunile 0 și 4. Mai greu de distins între ele sunt acțiunile ‘Badminton_Smash’ și ‘Badminton_Clear’, totuși se deosebesc pe dimensiunea 2.

- Valorile la momentul $t = 0$. (doar câteva exemple)

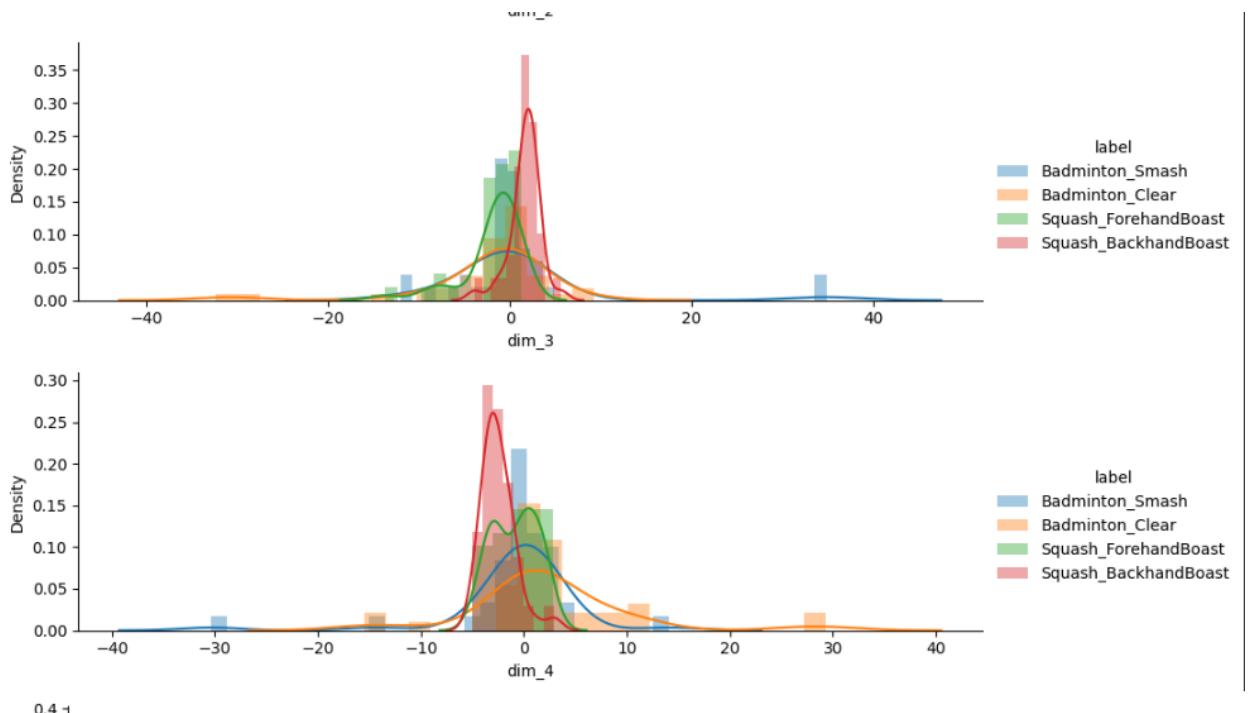


! Rosu și verde se evidențiază foarte bine.



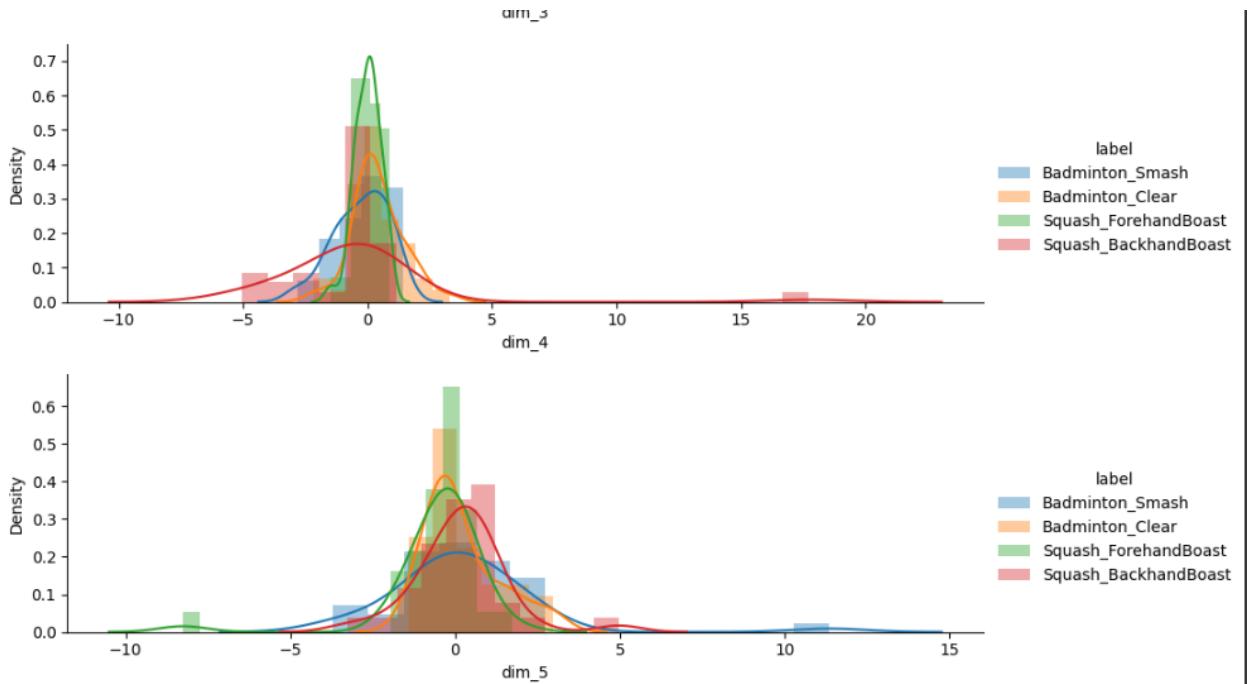
Verde este foarte diferit.

- Valorile la momentul $t = 10$. (doar cateva exemple)



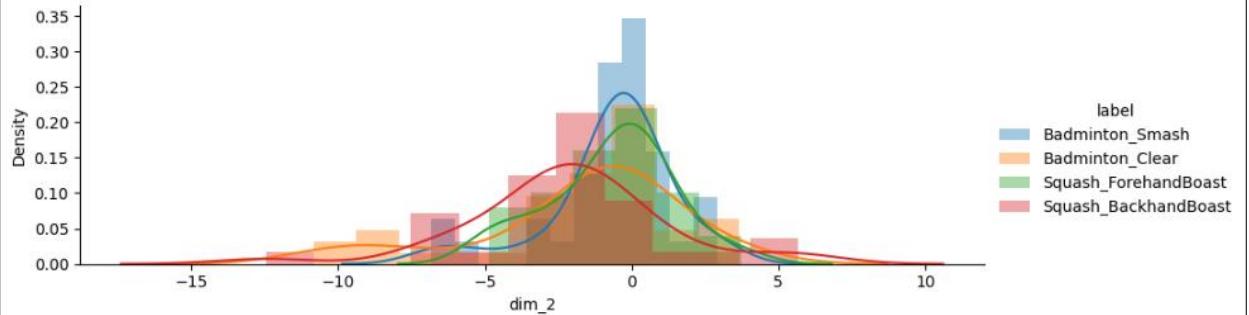
Rosu se evidențiază foarte bine la $t = 10$.

- Valorile la momentul $t = 20$. (doar cateva exemple)



Dupa cum se poate vedea, verdele din nou este foarte diferit. Ce este mai interesant, există diferențe mai mari între acțiunile ‘Badminton_Smash’ și ‘Badminton_Clear’ pentru dim_5 și chiar dim_4. Deci, desigur mai greu de învățat, algoritmul ar trebui să reusească chiar și din datele brute să gasească niște diferențe.

- Valorile la momentul $t = 29$. (doar câteva exemple)



Se observă diferențe între albastru și portocaliu. De asemenea, roșu este foarte diferit. Este un exemplu bun.

2. Extragerea manuală de atrăzite

Extragerea de atrăzite a concis în selectarea atrăzitelor statistice și de tip Fourier.

! Important !

Extragerea de atrăzite a fost exersată în cadrul temei. Initial, am extras atrăzitelor fără a face o standardizare a datelor. Am ajuns până la antrenarea modelului și am observat rezultatele. De asemenea, am încercat o standardizare după extragerea atrăzitelor. Am încercat folosirea funcției `MinMaxScaler()`, dar și a funcției `StandardScaler()`, pe rând. O alta metodă pe care am încercat-o a fost standardizarea celor 30 de valori de pe o axă înainte de orice prelucrare și pastrarea atrăzitelor astăzi cum reies din calcule fără modificări ulterioare. Am observat că astfel, algoritmul reușește să obțină rezultate mai bune. Dacă standardizam datele după

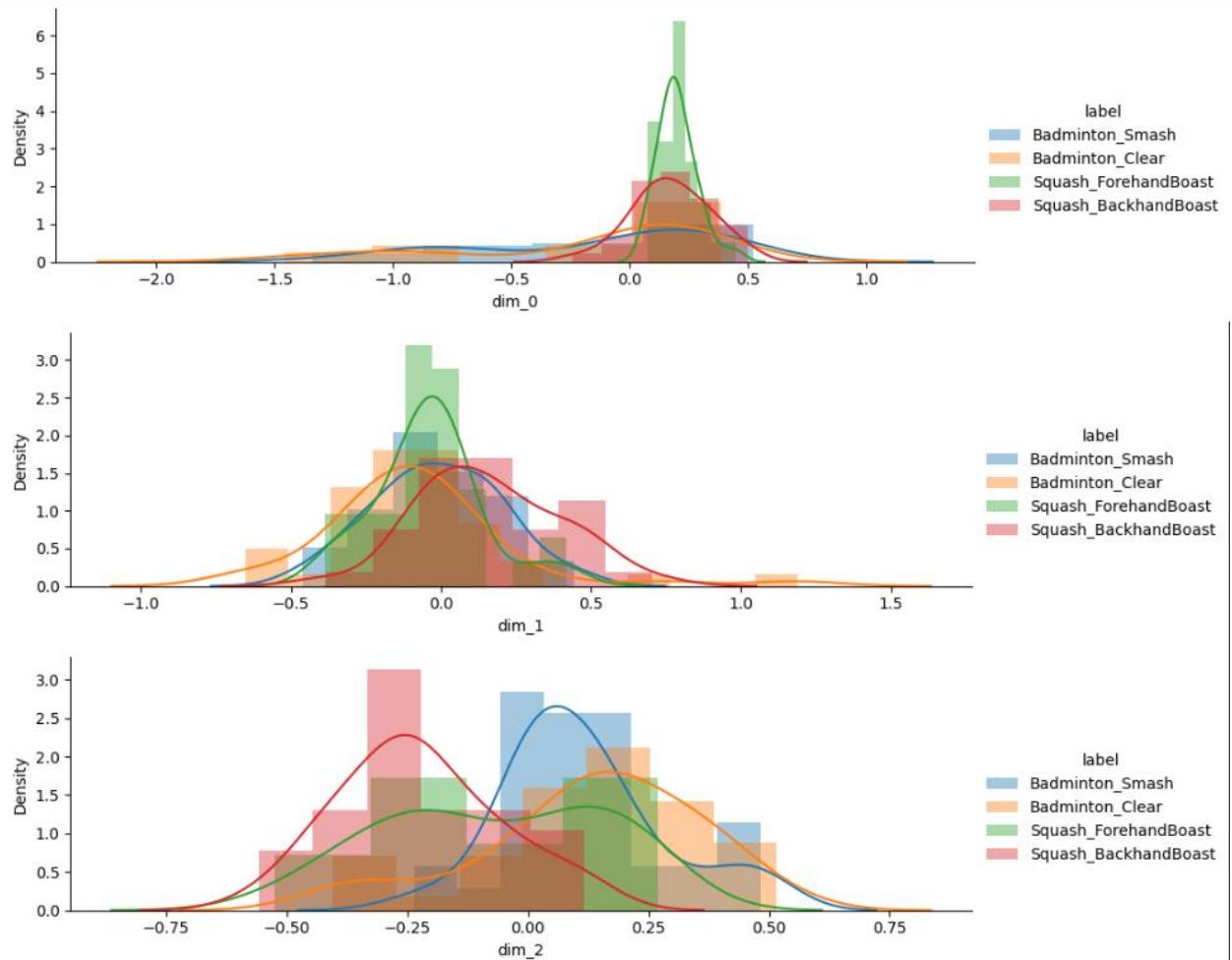
extragerea atributelor, cred ca variatia datelor devinea foarte mica si era si mai dificil sa observi diferente intre actiunile “albastre” si “portocalii”. Daca nu standardizam deloc datele, existau totusi diferente mai mari, unele serii de date aveau valori in intervalul [0,1] , iar altele in intervalul [15, 50] si cred ca exista un dezechilibru evident.

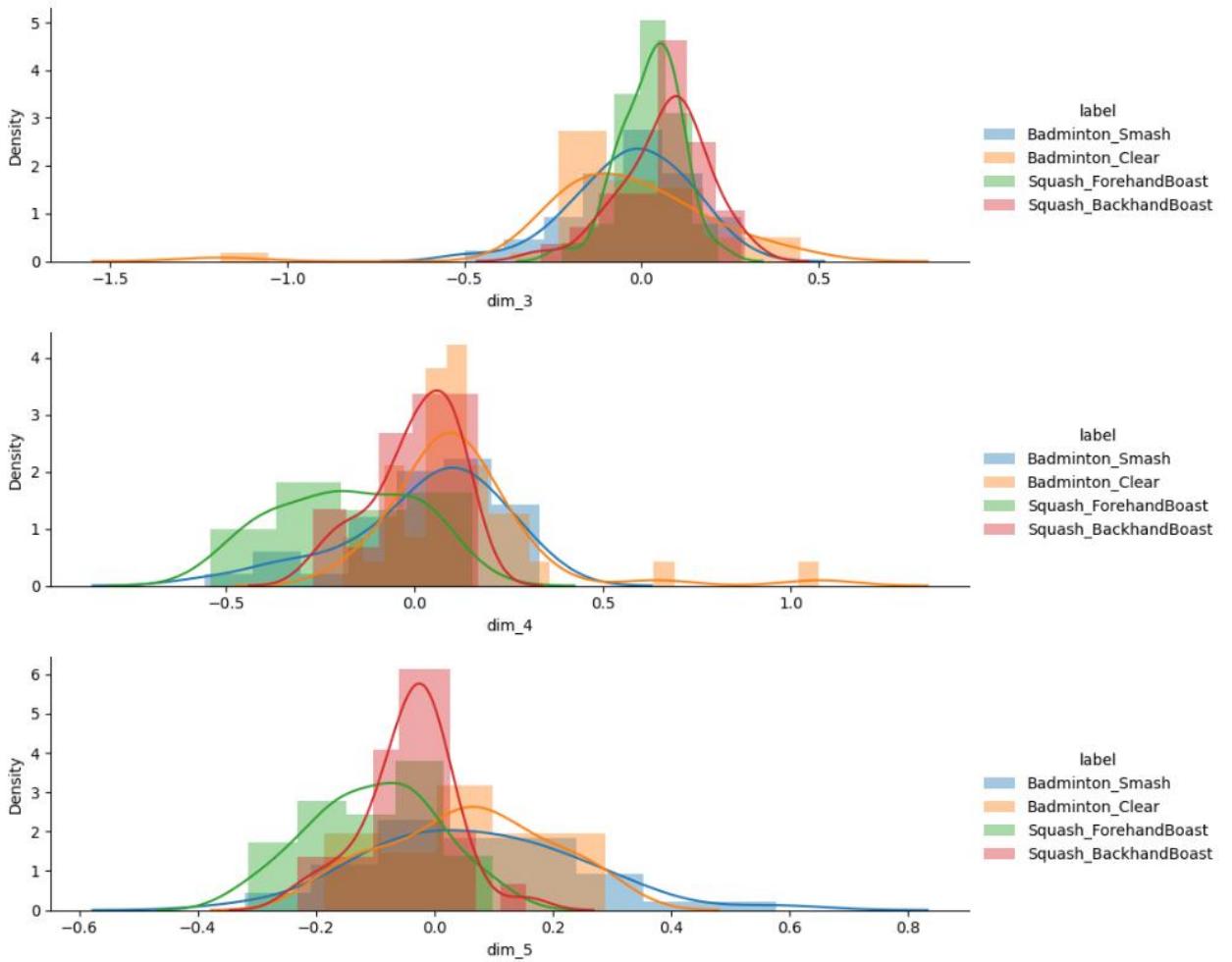
Concluzie

Prin standardizarea datelor inainte de extragerea atributelor, se obtine o varietate a datelor acceptabila si buna pentru antrenare.

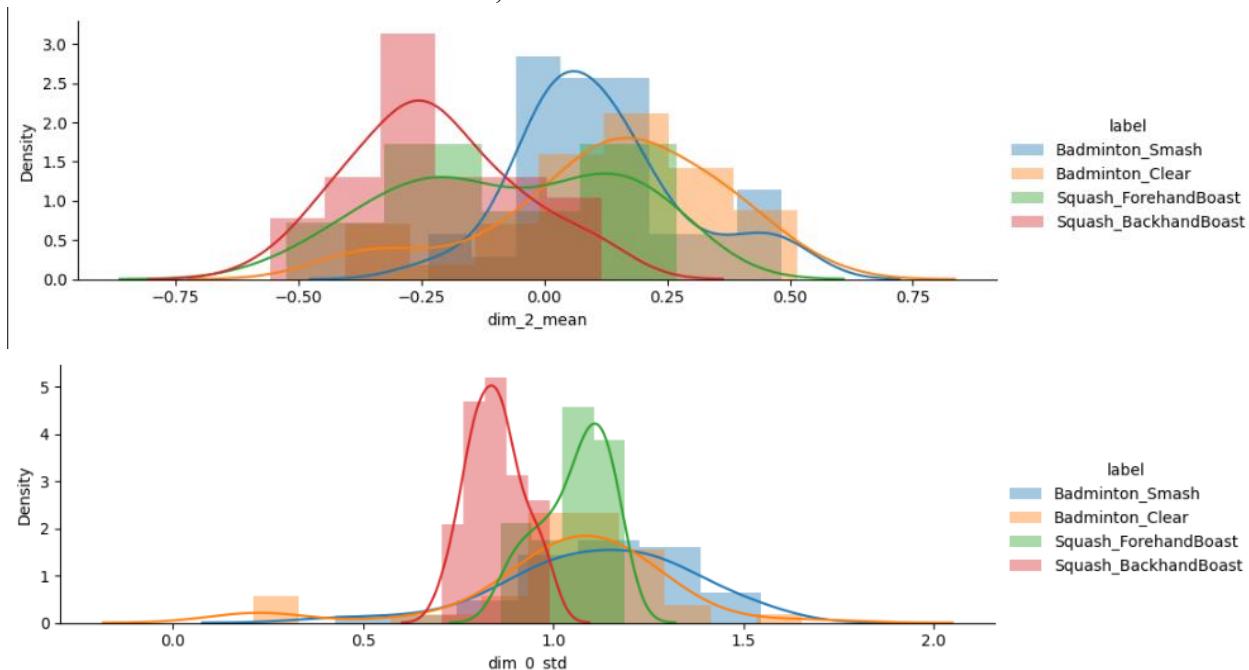
Cateva grafice care ilustreaza datele:

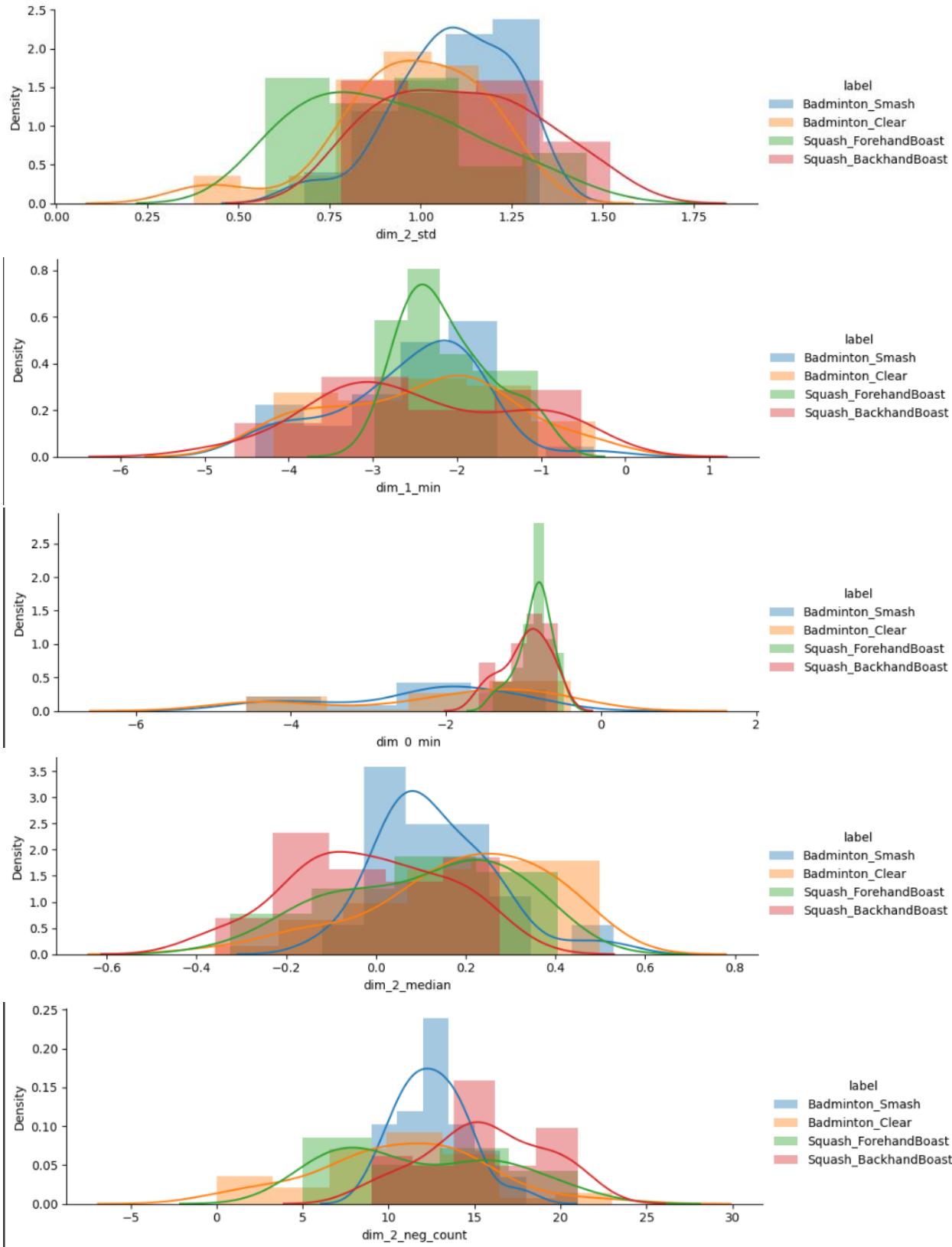
- Cum arata datele de pe axe dupa standardizare:

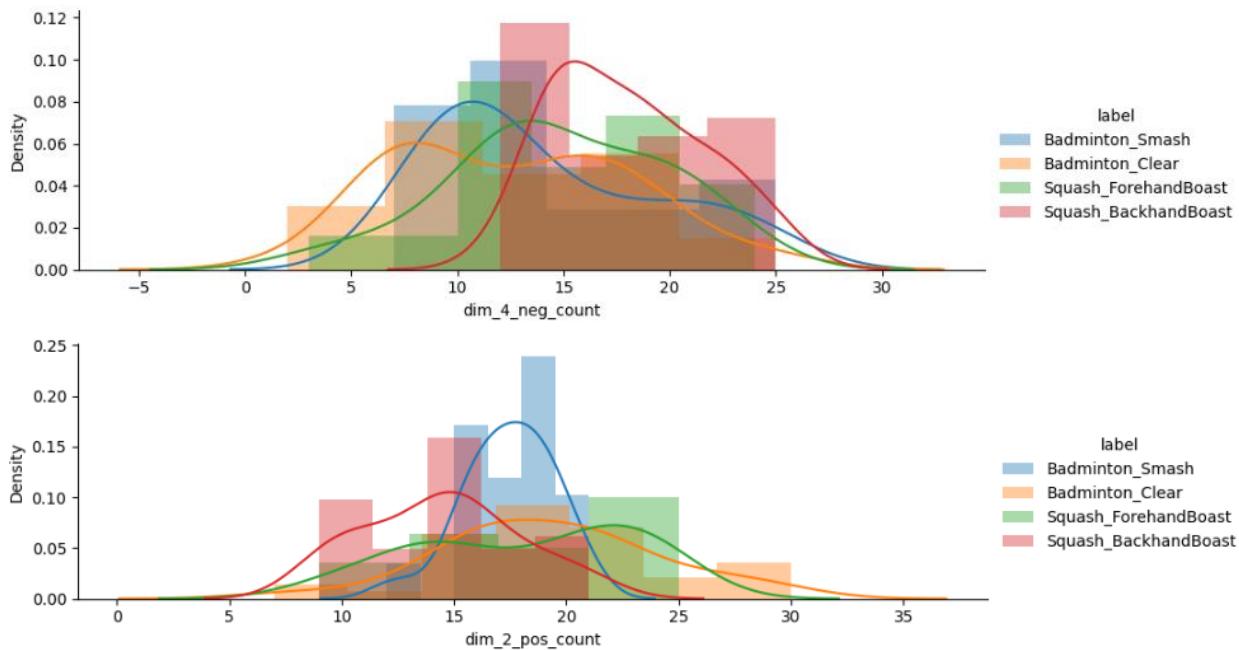




- Cum arata cateva dintre atrbutele extrase, statistice:

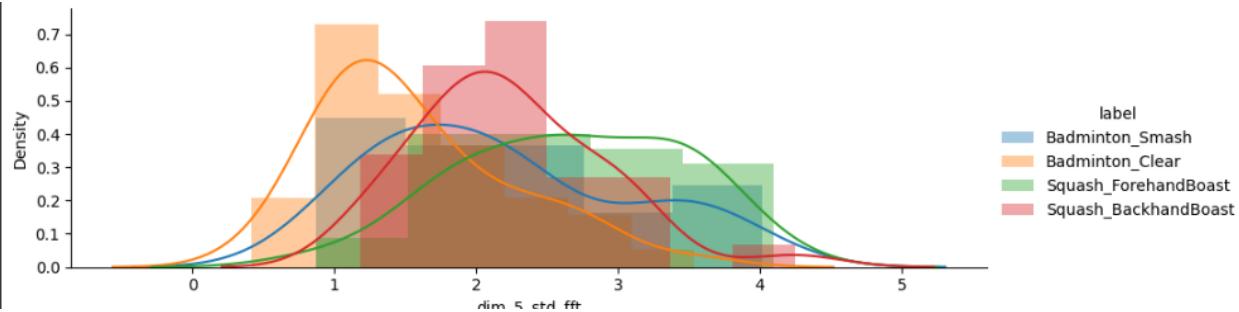


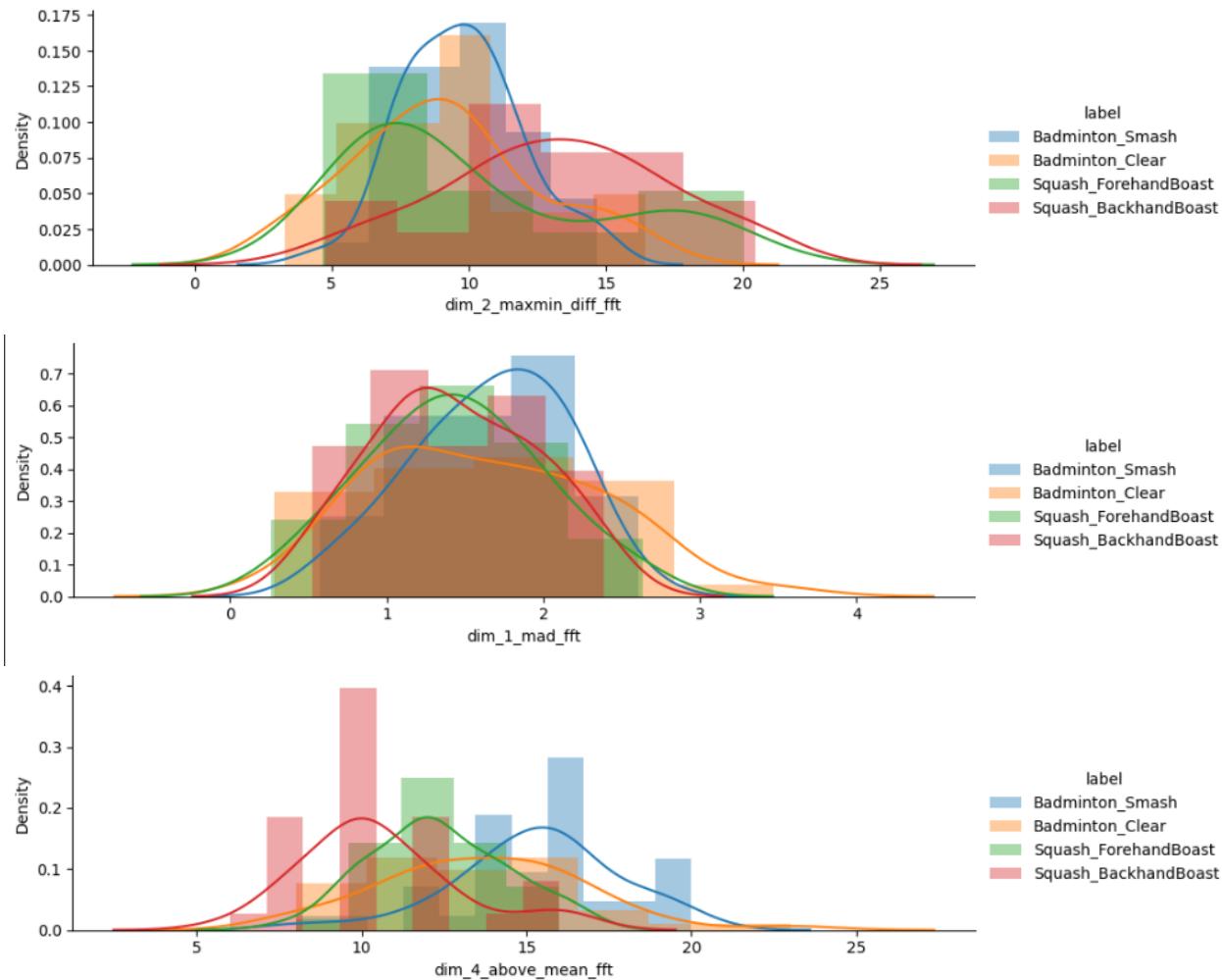




Aceasta reprezinta cateva grafice care cred ca au reusit sa evidenteze diferente importante.

- Cum arata cateva dintre atrbutele extrase, de tip Fourier.



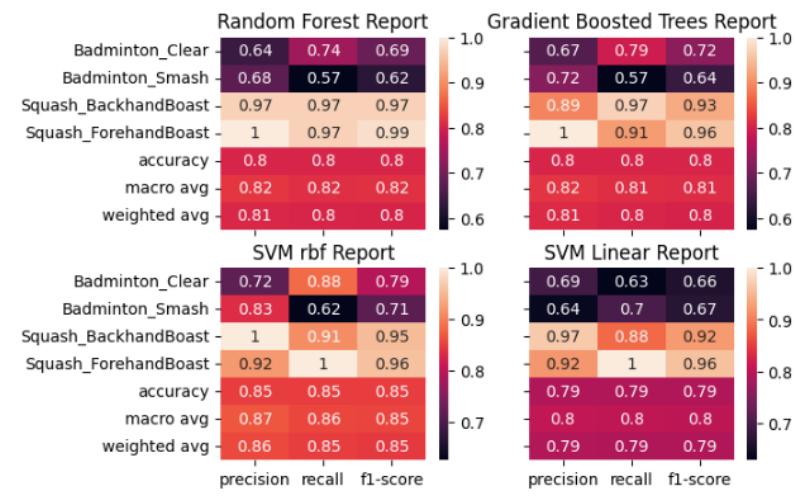


! Am filtrat attributele folosind functia *VarianceThreshold()* cu o valoare a treshold-ului de 0.4.
Astfel, in loc de 168 de attribute obtinute, am ramas cu 128.

3. Algoritmi de Invatare Automata

A se consulta fisierul Project_ML.xlsx pentru rezultate !

Initial, am creat 4 modele : RandomForest, Gradient Boosted Trees, SVC rbf si SVC Linear.



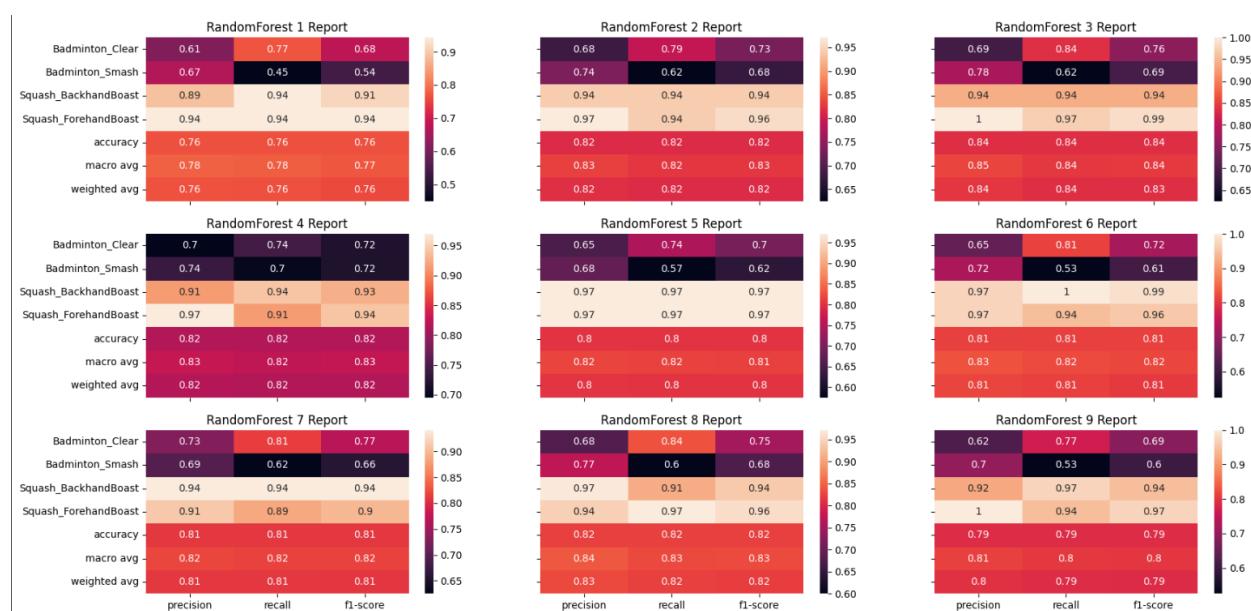
Modelele au fost create cu parametri stabiliți aleator. Scopul meu în continuare a fost de a găsi combinația de parametri pentru a obține cea mai bună variantă a fiecărui model.

Am creat 6 modele de RandomForest:

```
random_forest_1 = RandomForestClassifier(n_estimators = 60, max_depth = 5, max_samples = 0.3)
random_forest_2 = RandomForestClassifier(n_estimators = 60, max_depth = 5, max_samples = 0.6)
random_forest_3 = RandomForestClassifier(n_estimators = 60, max_depth = 5, max_samples = 0.9)

random_forest_4 = RandomForestClassifier(n_estimators = 120, max_depth = 10, max_samples = 0.3)
random_forest_5 = RandomForestClassifier(n_estimators = 120, max_depth = 10, max_samples = 0.6)
random_forest_6 = RandomForestClassifier(n_estimators = 120, max_depth = 10, max_samples = 0.9)

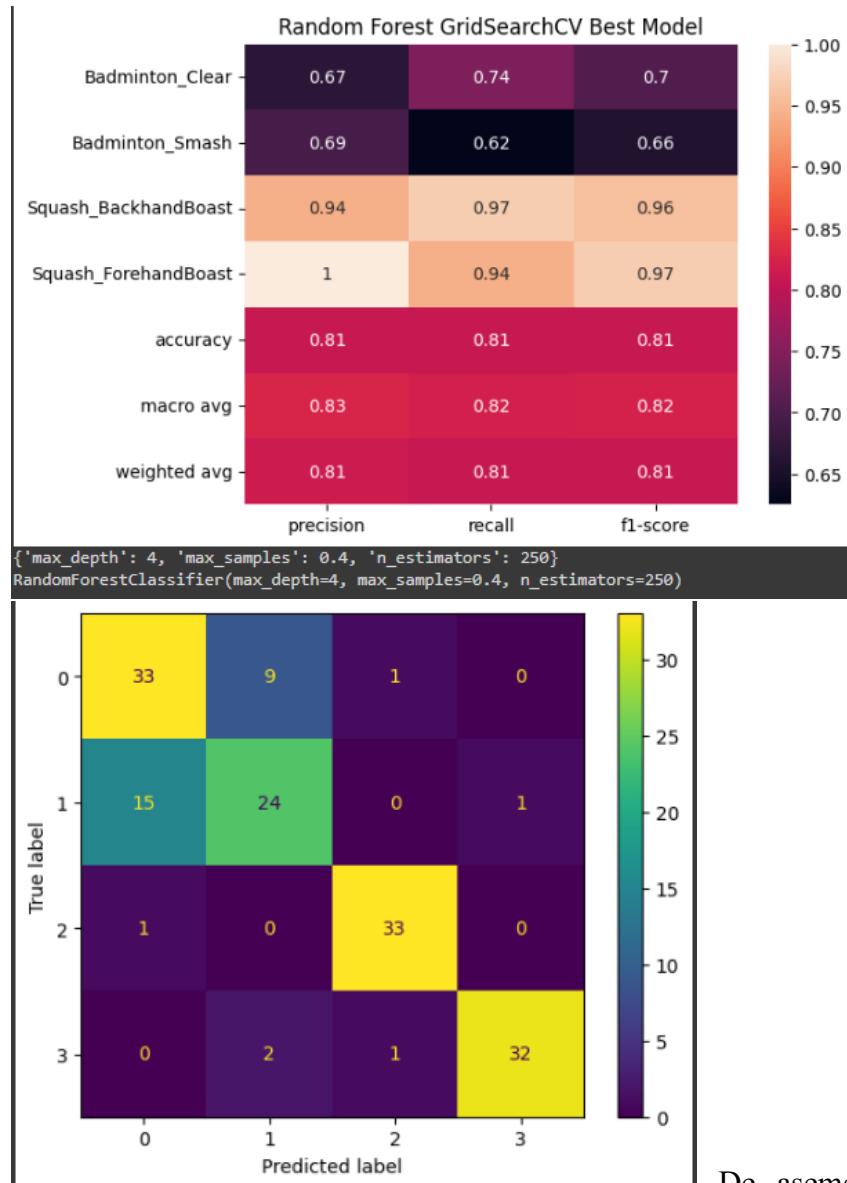
random_forest_7 = RandomForestClassifier(n_estimators = 80, max_depth = 15, max_samples = 0.3)
random_forest_8 = RandomForestClassifier(n_estimators = 80, max_depth = 15, max_samples = 0.6)
random_forest_9 = RandomForestClassifier(n_estimators = 80, max_depth = 15, max_samples = 0.9)
```



Am realizat acest lucru pentru a selecta cele mai bune combinatii pentru metoda de cautare a parametrilor GridSearchCV. De observat ca modelele 3, 4, 8 obtin cam cele mai bune rezultate. Odata ce creste numarul de copaci si adancimea, rezultate mai bune se obtin cu un procent de samples mai mic. Deci, cred ca modelul se descurca destul de bine, datele fiind usor de invatat, nu are nevoie de foarte multi copaci si adancimi mari (cumva, ambele in acelasi timp).

Rezultatele obtinute in urma rularii algoritmului GridSearchCV cu cv = 5.

```
# Define a dictionary of hyperparameters for the Random Forest Classifier model
parameters = {
    'max_depth': [2, 4, 6, 5, 7, 8, 9, 10, 11, 12, 15, 17, 20, 25],
    'n_estimators': [50, 70, 100, 125, 150, 200, 250, 300],
    'max_samples': [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]
}
```



De asemenea, se observa ca greselile algoritmului au loc intre ‘albastru’ si ‘portocaliu’, adica feature-urile cele mai grele de invatat.

Se observa combinatia de parametri ales: max_depth = 4, max_samples = 0.4, n_estimators = 250. Acet lucru arata ca feature-urile sunt usor de invatat, dar probabil sunt multe si atunci este preferat un numar mari de copaci intr-o combinatie cu max_samples=0.4 pentru a se feri de overfitting.

RacketSports																							
Numar de feature-uri considerate			168	Numar de feature-uri la antrenare			128																
Random Forest Classifier																							
n_estimators	max_depth	max_samples	Badminton_Clear			Badminton_Smash			Squash_BackhandBoast			Squash_ForehandBoast			Media			Varianta			Acuratetea		
Precision	Recall	F1-score	Precision	Recall	F1-score	Precision	Recall	F1-score	Precision	Recall	F1-score	Precision	Recall	F1-score	Precision	Recall	F1-score	Precision	Recall	F1-score			
60	5	0.4	0.3	0.61	0.77	0.68	0.67	0.45	0.54	0.89	0.94	0.91	0.94	0.94	0.94	0.78	0.78	0.77	0.026	0.053	0.036	0.76	
		0.6	0.68	0.79	0.73	0.74	0.62	0.68	0.94	0.94	0.94	0.97	0.94	0.96	0.83	0.82	0.83	0.021	0.023	0.020	0.82		
		0.9	0.69	0.84	0.76	0.78	0.62	0.69	0.94	0.94	0.94	1	0.97	0.99	0.85	0.84	0.85	0.020	0.025	0.020	0.84		
80	15	0.4	0.3	0.73	0.81	0.77	0.69	0.62	0.66	0.94	0.94	0.94	0.91	0.89	0.9	0.82	0.82	0.82	0.016	0.020	0.016	0.81	
		0.6	0.68	0.84	0.75	0.77	0.6	0.68	0.97	0.91	0.94	0.94	0.97	0.96	0.84	0.83	0.83	0.019	0.026	0.019	0.82		
		0.9	0.62	0.77	0.69	0.7	0.53	0.6	0.92	0.97	0.94	1	0.94	0.97	0.81	0.80	0.80	0.032	0.041	0.034	0.79		
120	10	0.4	0.3	0.7	0.74	0.72	0.74	0.7	0.72	0.91	0.94	0.93	0.97	0.91	0.94	0.83	0.82	0.83	0.017	0.014	0.015	0.82	
		0.6	0.65	0.74	0.7	0.68	0.57	0.62	0.97	0.97	0.97	0.97	0.97	0.97	0.82	0.81	0.82	0.031	0.038	0.033	0.8		
		0.9	0.65	0.81	0.72	0.72	0.53	0.61	0.97	1	0.99	0.97	0.94	0.96	0.83	0.82	0.82	0.028	0.044	0.034	0.81		
100	10	-	0.64	0.74	0.69	0.68	0.57	0.62	0.97	0.97	0.97	1	0.97	0.99	0.82	0.81	0.82	0.036	0.038	0.036	0.8		
250	4	0.4	0.67	0.74	0.7	0.69	0.62	0.66	0.94	0.97	0.96	1	0.94	0.97	0.83	0.82	0.82	0.029	0.028	0.027	0.81		

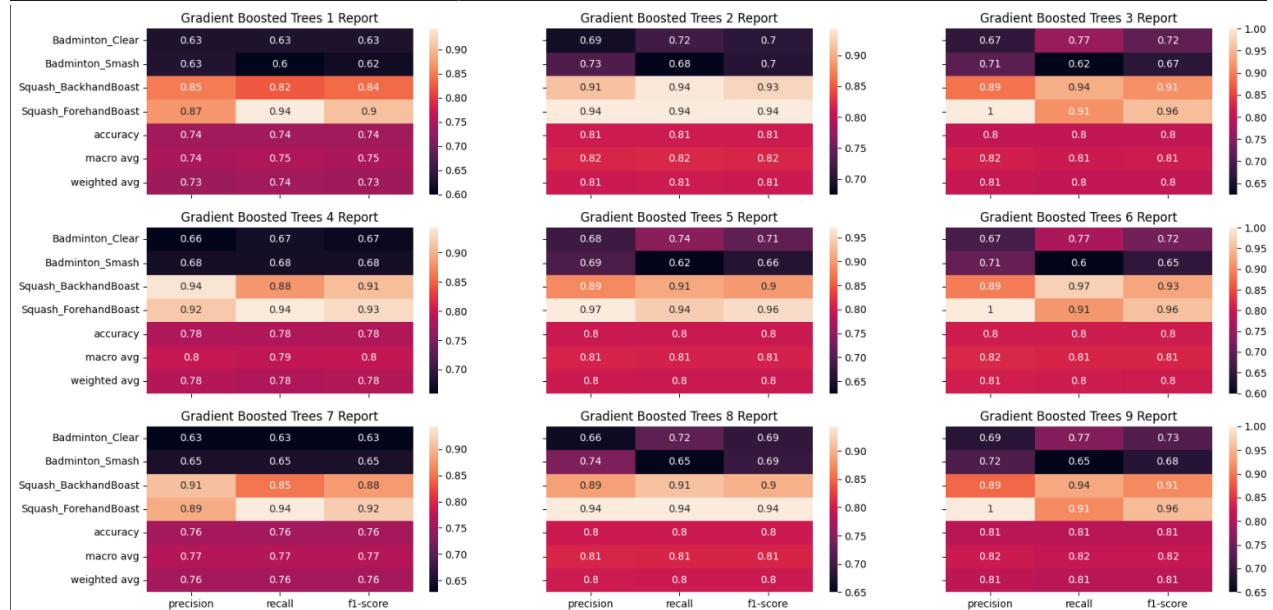
De asemenea, am creat 6 modele XGB.

```
# Define XGBClassifiers with different hyperparameters

gb_trees_1 = XGBClassifier(n_estimators = 60, max_depth = 5, learning_rate = 0.01)
gb_trees_2 = XGBClassifier(n_estimators = 60, max_depth = 5, learning_rate = 0.1)
gb_trees_3 = XGBClassifier(n_estimators = 60, max_depth = 5, learning_rate = 0.2)

gb_trees_4 = XGBClassifier(n_estimators = 120, max_depth = 10, learning_rate = 0.01)
gb_trees_5 = XGBClassifier(n_estimators = 120, max_depth = 10, learning_rate = 0.1)
gb_trees_6 = XGBClassifier(n_estimators = 120, max_depth = 10, learning_rate = 0.2)

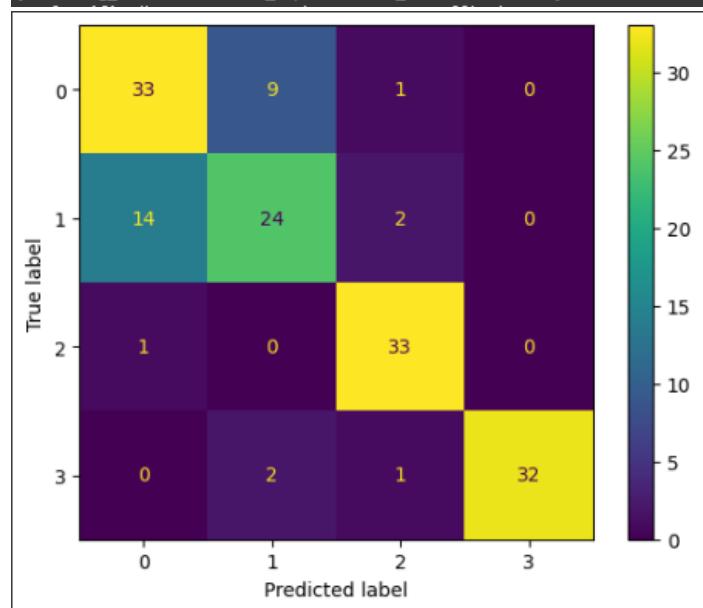
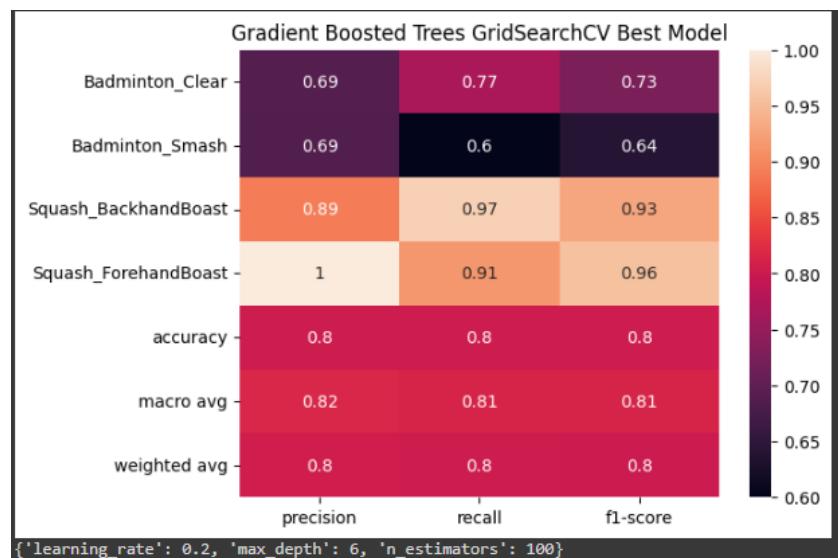
gb_trees_7 = XGBClassifier(n_estimators = 80, max_depth = 15, learning_rate = 0.01)
gb_trees_8 = XGBClassifier(n_estimators = 80, max_depth = 15, learning_rate = 0.1)
gb_trees_9 = XGBClassifier(n_estimators = 80, max_depth = 15, learning_rate = 0.2)
```



Se observa foarte clar ca un learning_rate de 0.01 este prea mic. De asemenea, cele mai bune modele ar fi modelul 2 si modelul 9. Reusind sa obtina rezultate bune pe ambele configuratii, imi dau seama ca exista flexibilitate, cumva este loc de invatat mai multe feature-uri printr-o configuratie buna.

Am rulat GridSearchCV cu combinatiile:

```
# Parameter grid for Grid Search Cross Validation
parameters = {
    'max_depth': [2, 4, 6, 8, 10, 12, 15, 17, 20, 25],
    'n_estimators': [50, 70, 100, 125, 150, 200, 250, 300],
    'learning_rate': [0.05, 0.1, 0.15, 0.2, 0.25]
}
```



Nu exista diferență față de RandomForest. Parametrii alesi au fost: learning_rate = 0.2, max_depth = 6, n_estimators = 100. Deci, din nou este preferată opțiunea în care sunt ales mai mulți copaci de

adancime mai mica. Astfel, nu se ajunge la overfit, dar fiind mai multi copaci, sunt invatate mai multe feature-uri. Probabil, pentru a rezolva mai bine problema primelor 2 clase, este nevoie de structuri mai complexe, dar structuri mai complexe, ar suprainvata clasele 2,3.

De asemenea, am creat 6 modele SVC.

```

svm_1 = SVC(kernel = 'rbf', C = 0.1)
svm_2 = SVC(kernel = 'rbf', C = 1)
svm_3 = SVC(kernel = 'rbf', C = 10)

svm_4 = SVC(kernel = 'linear', C = 0.1)
svm_5 = SVC(kernel = 'linear', C = 1)
svm_6 = SVC(kernel = 'linear', C = 10)

svm_7 = SVC(kernel = 'poly', C = 0.1)
svm_8 = SVC(kernel = 'poly', C = 1)
svm_9 = SVC(kernel = 'poly', C = 10)

```

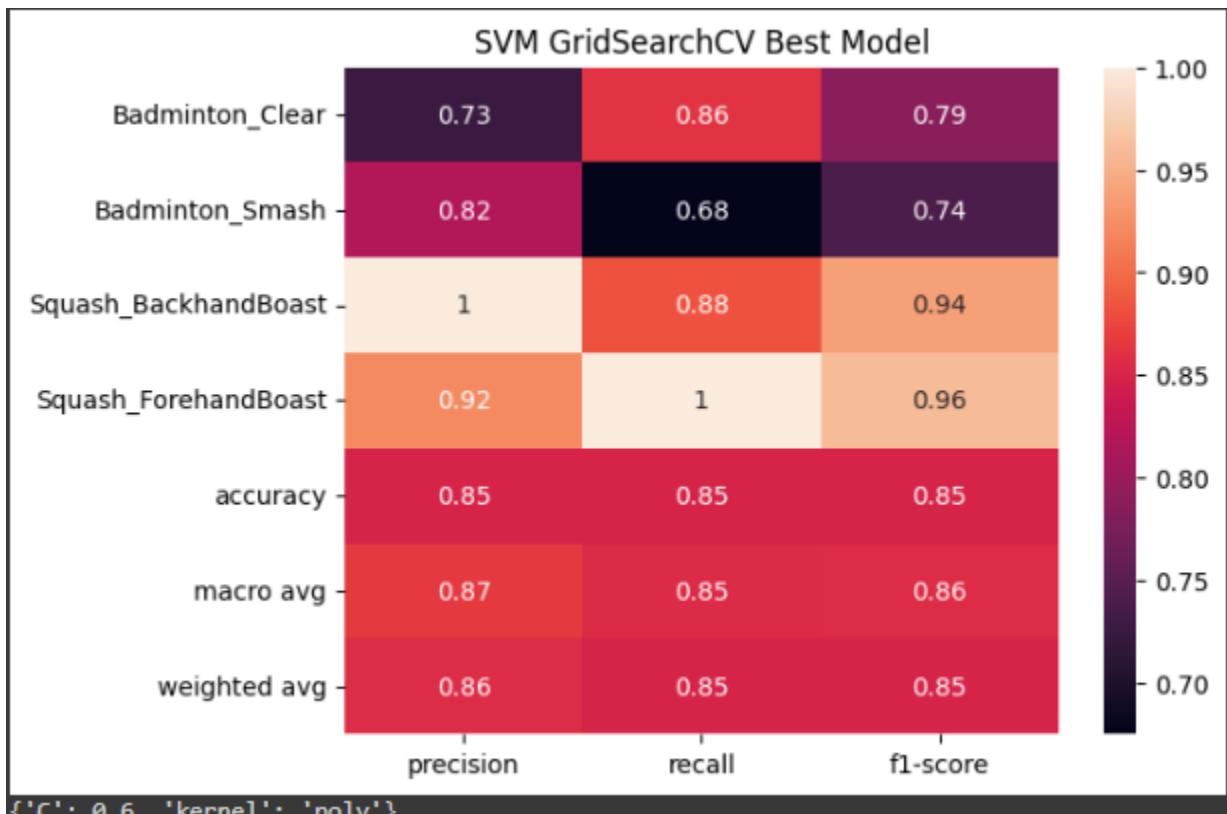


Se observă ca cele mai slabe modele sunt 4,5,6 ca o generalizare a faptului că un kernel linear nu este potrivit pentru aceasta problema. Multe mai bune sunt rezultatele pentru kernelul rbf sau poly.

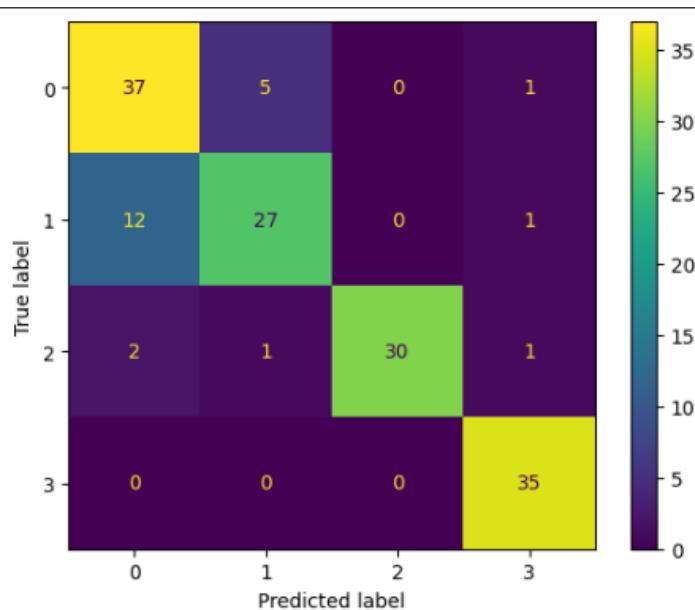
De asemenea, un factor C mic nu este bun. Algoritmul își subțiaza marginile când C crește și rezultate mai bune sunt obținute cu $C > 1$, dar și când $C = 1$.

Am rulat GridSearchCV cu parametrii:

```
# Define a dictionary of parameter ranges to search over
parameters = {'kernel': ['poly', 'linear', 'rbf'], 'C': [0.08, 0.09, 0.1, 0.2, 0.3, 0.4, 0.55, 0.75, 0.8, 0.6, 0.7, 1, 2]}
```



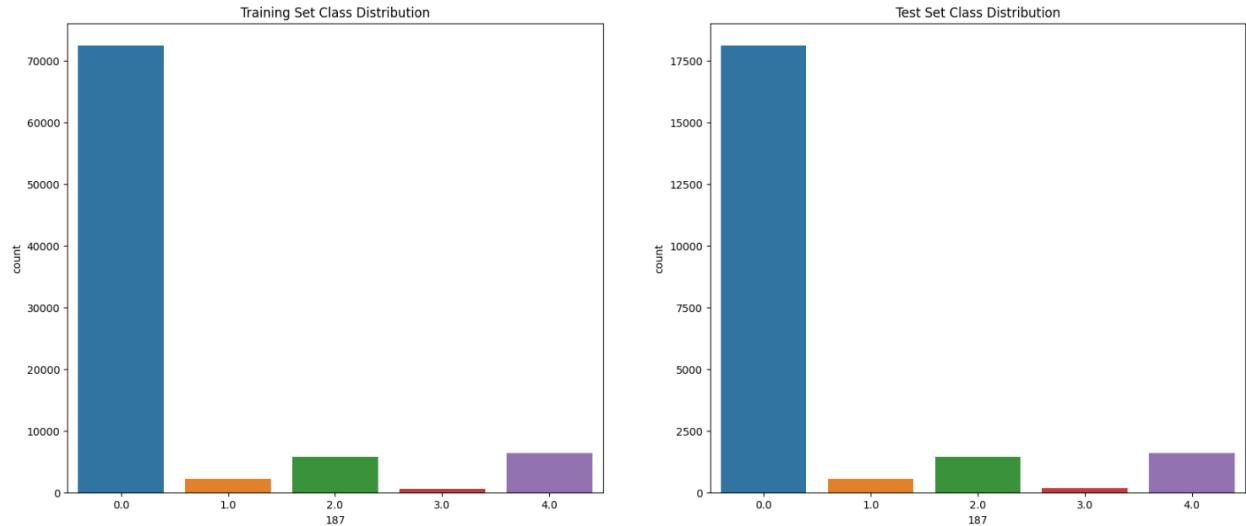
Parametrii obtinuti au fost C = 0.6, kernel = ploy.



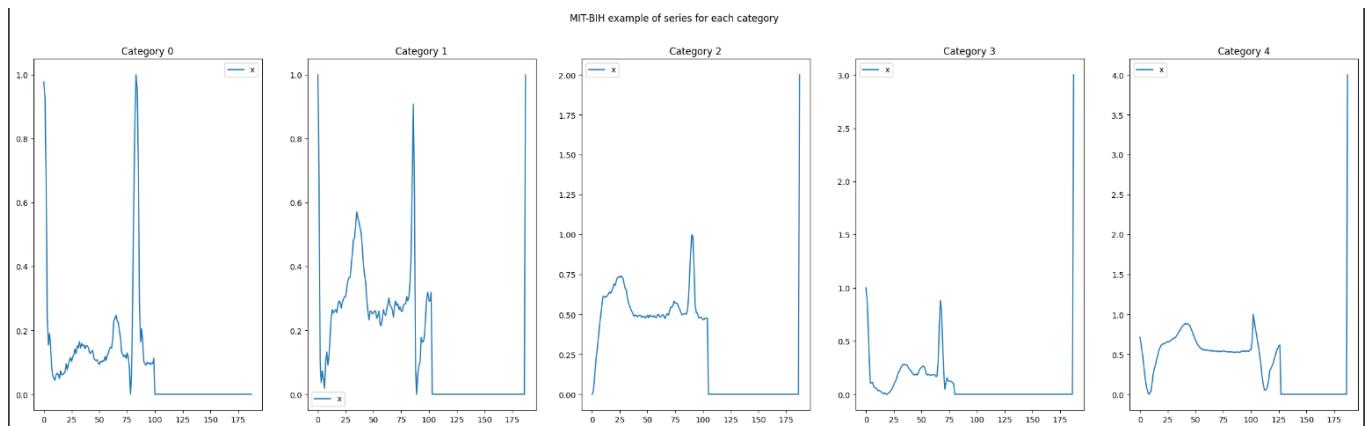
Exista o imbunatatire foarte mare fata de ceilalti algoritmi. Clasa de actiune 2, nu mai este incurcata cu alta clasa. Si intre clasele 0,1 exista mai putine greseli.

MIT-BIH

1. Explorarea datelor



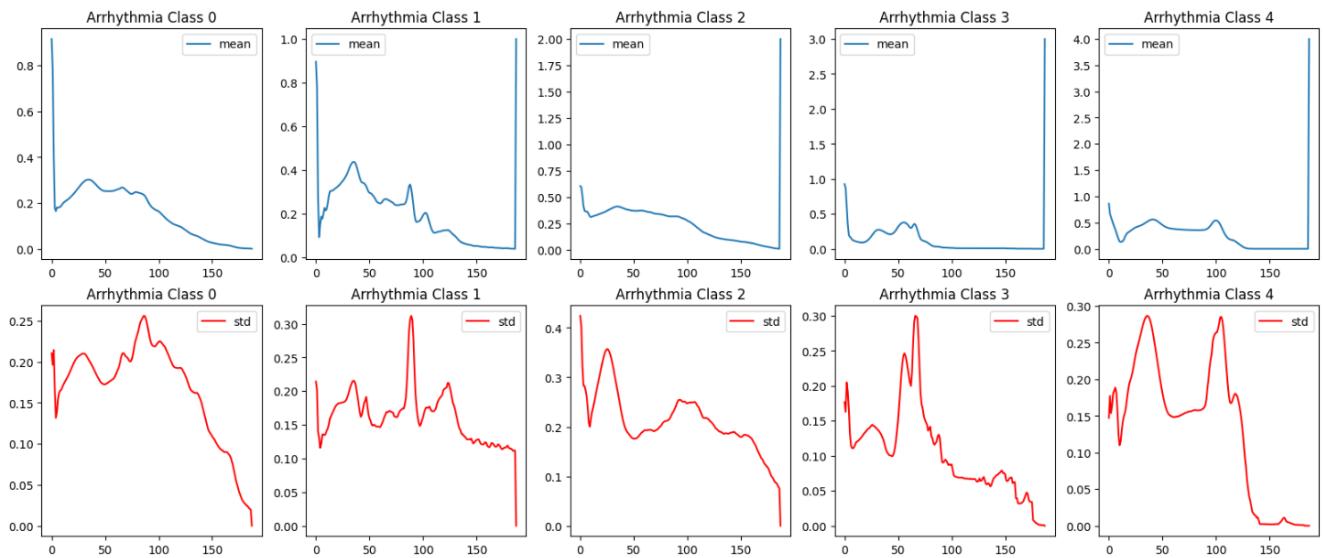
Exista un dezechilibru in privinta setului de date. Acest lucru nu este bun. Tinand cont ca fiecare exemplu din dataset contribuie egal, un learning_rate egal cu al celorlalte exemple la calcularea loss-ului, inseamna ca clasa 0 va eclipsa celelalte clase. Algoritmul va tinde sa perfectioneze clasa 0 in timp ce interesul lui fata de clasa 3 va fi foarte mic. Acest lucru este adevarat daca algoritmul trateaza in mod clasic, fara schimbari, setul de date. Totusi, un lucru bun este faptul ca nu exista diferente de proportii intre train-test.



Se pot observa diferente destul de mari intre toate exemplele. La prima vedere, acestea par usor de invatat.

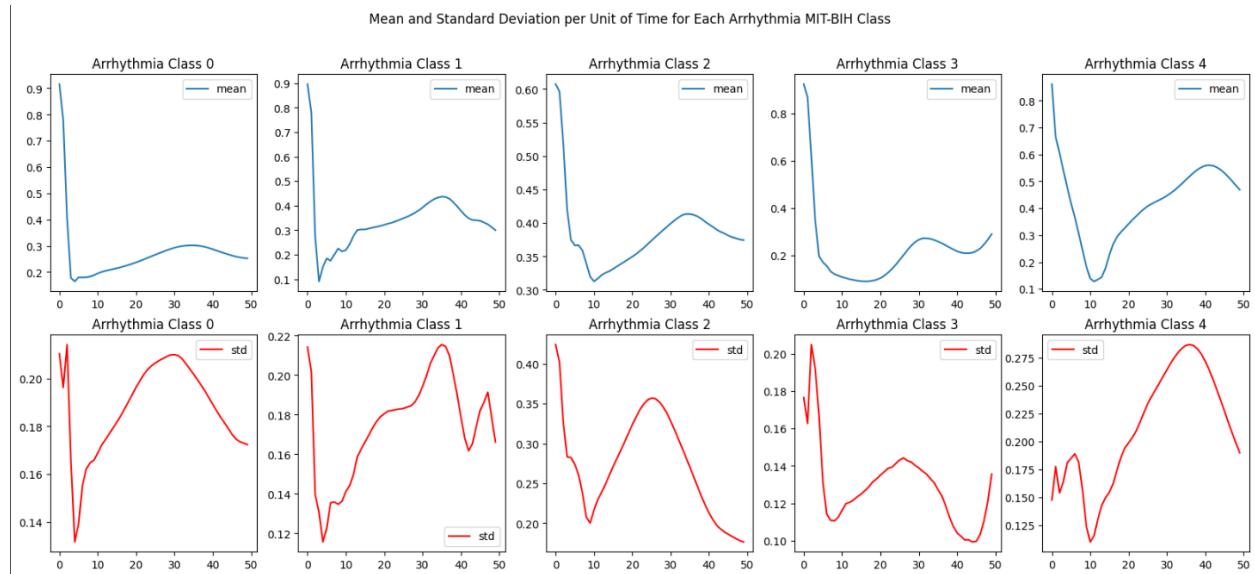
Grafic cu media si varianta pentru fiecare clasa:

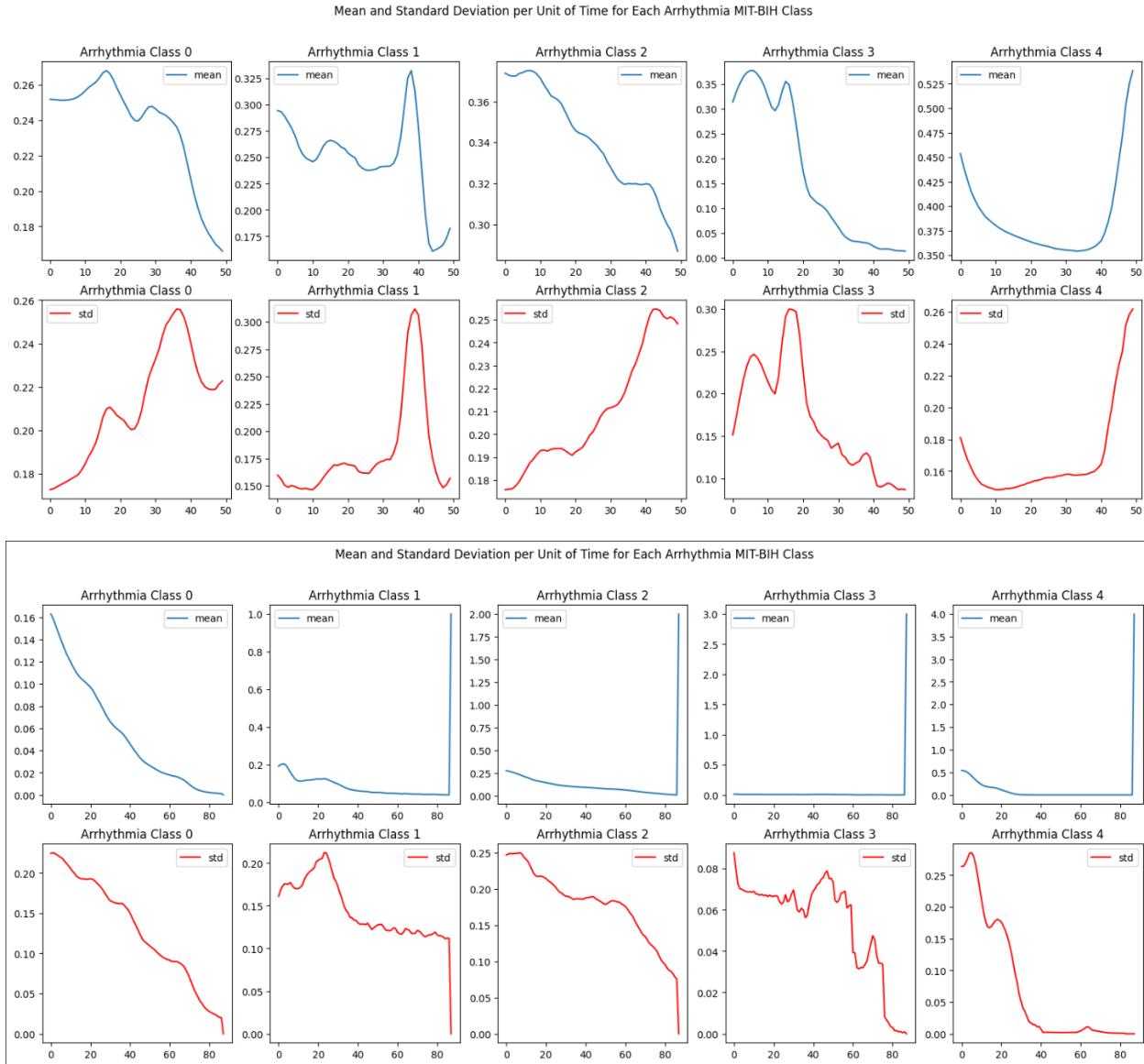
Mean and Standard Deviation per Unit of Time for Each Arrhythmia MIT-BIH Class



Se pot vedea mici asemanari intre clasa 0 si 1. Deci tinand cont ca exista foarte multe exemple din clasa 0, clasa 1 ar putea fi confundata cu aceasta. De asemenea, clasa 3 cred ca ar putea fi confundata mai usor cu clasa 4. Par sa se asemenea un pic.

Urmatoarele grafice prezinta datele prezentate mai sus pe chunk-uri de 50 de exemplu (timpi):





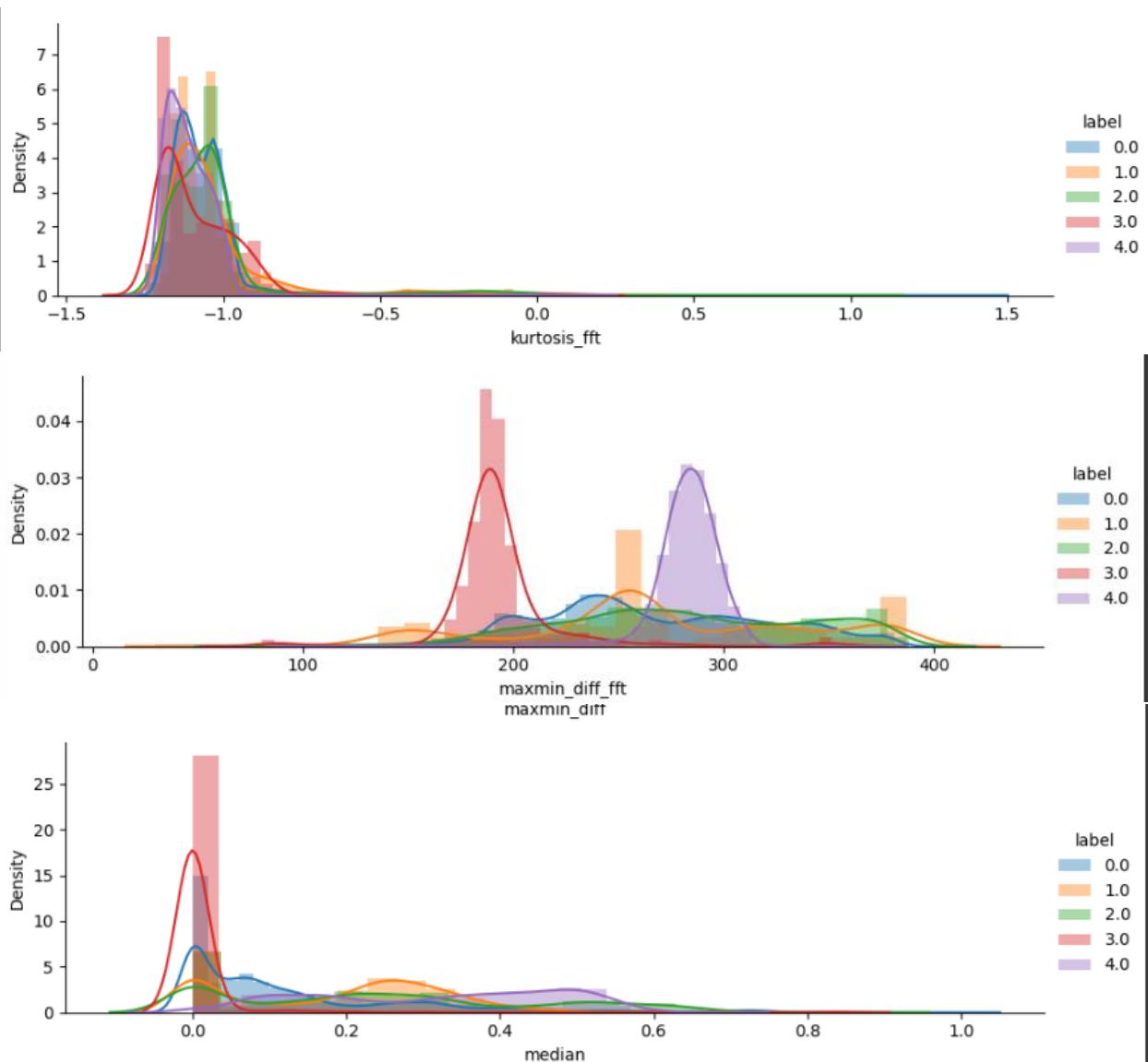
Observ ca există date redundante în ce privește primii 10 timpi și ultimii 10 timpi din fiecare serie. De aceea elimin aceste coloane.

2. Extragerea manuală de atrbute

In privinta atributelor, am extras aceleasi atrbute ca si in cazul dateset-ului RacketSports. In ce privește standardizarea, nu am realizat-o ca si in cazul celuilalt dataset intrucat aici bataile inimii se află intre [0,1] si este deja ok.

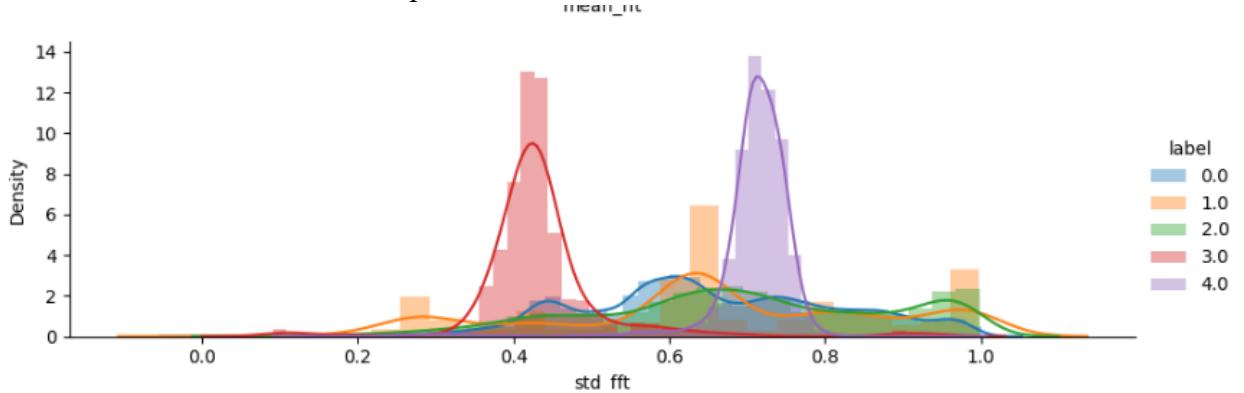
Deci, standardizarea s-a realizat pentru feature-urile extrase, folosind functia *MinMaxScaler()*. Alegerea functiei s-a realizat tot pe cale experimentală.

Cateva vizualizari ale datelor înainte de Standardizare:

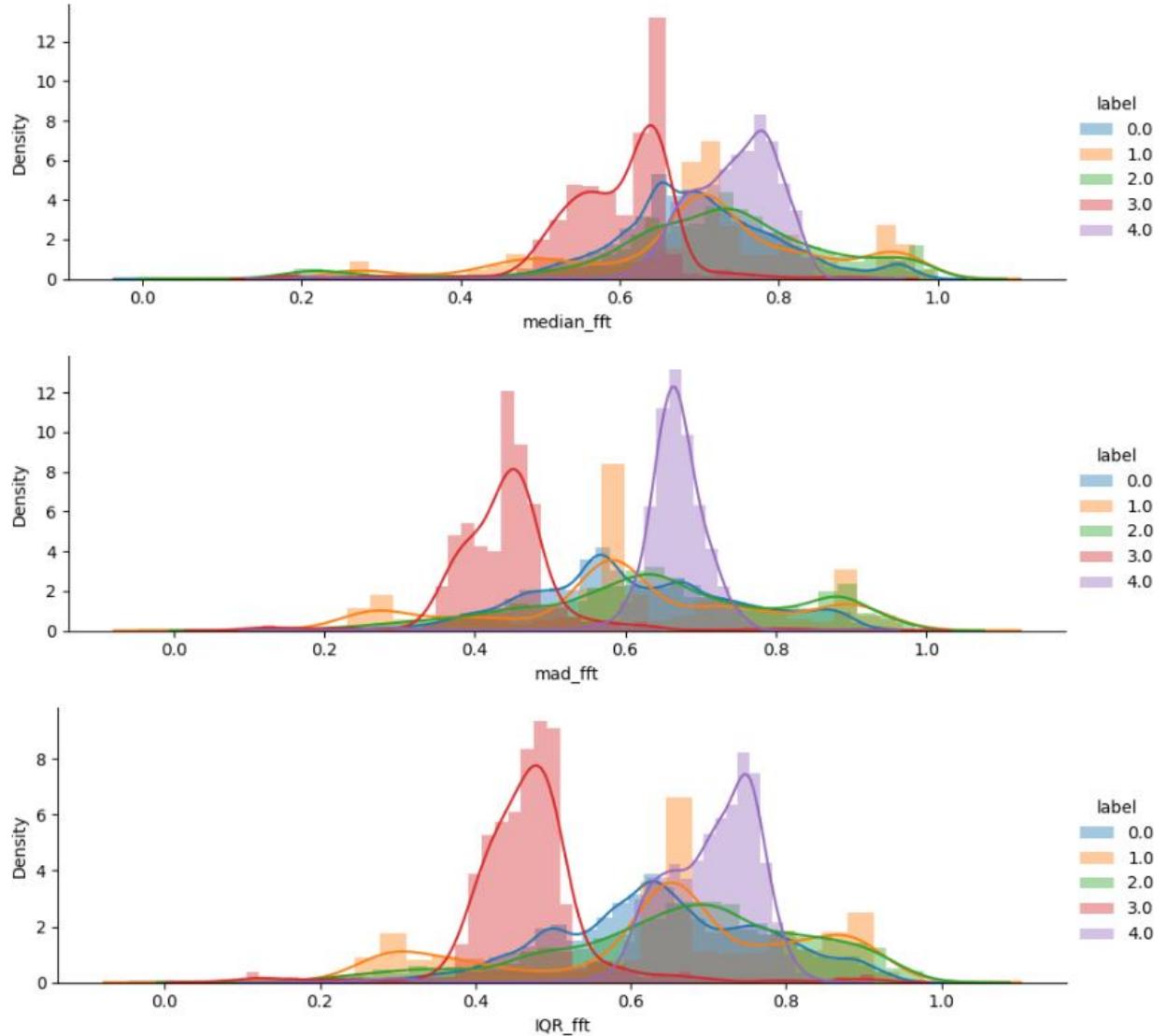


Cred ca se poate vedea foarte clar nevoie de standardizare. Numerele oscileaza de la [0,1] la [0, 400].

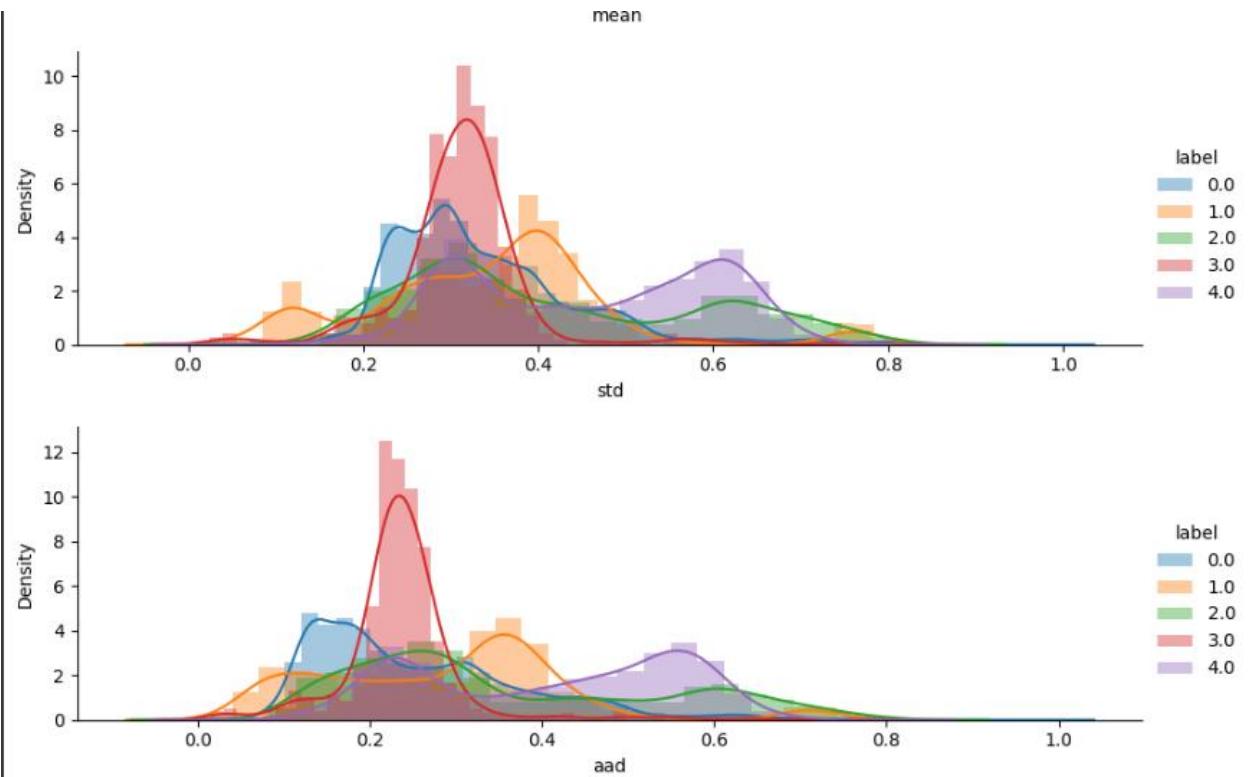
Cateva vizualizari ale datelor dupa Standardizare:



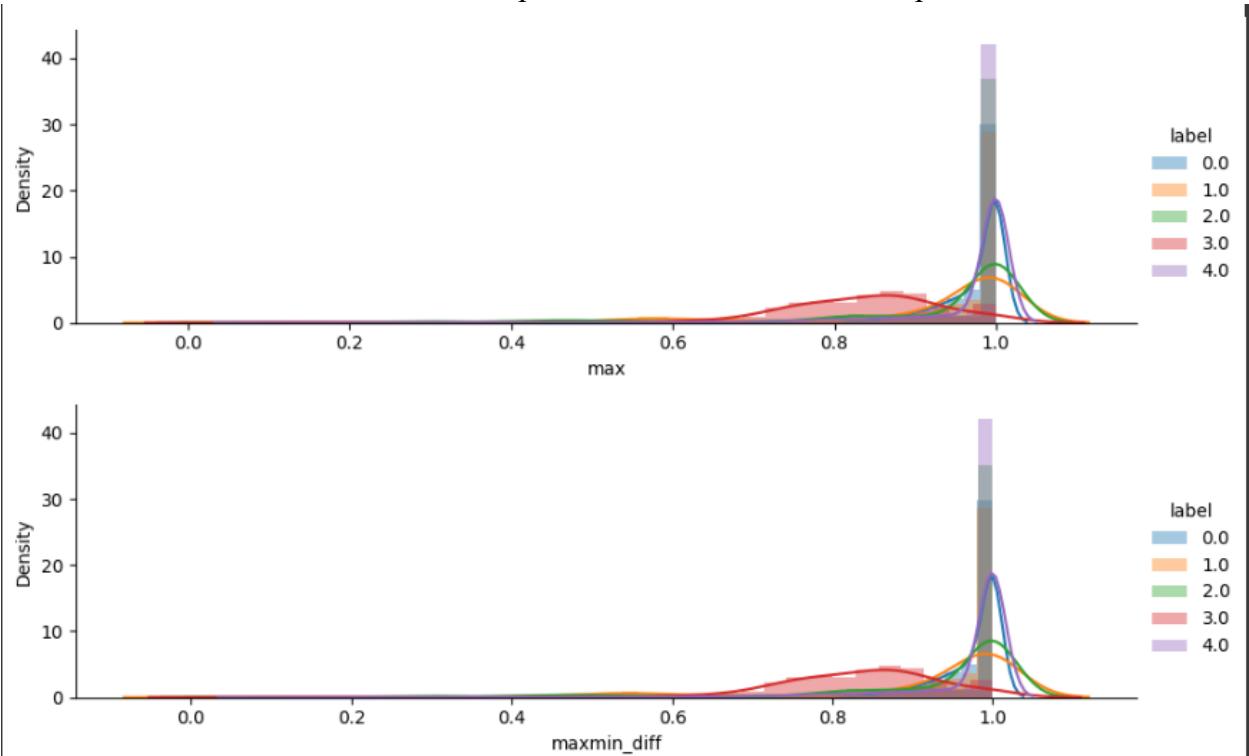
Usor de invatat rosu si mov. Asa cum am spus clasa portocalie este usor de confundat cu clasa albastra. (clasele 0 si 1)



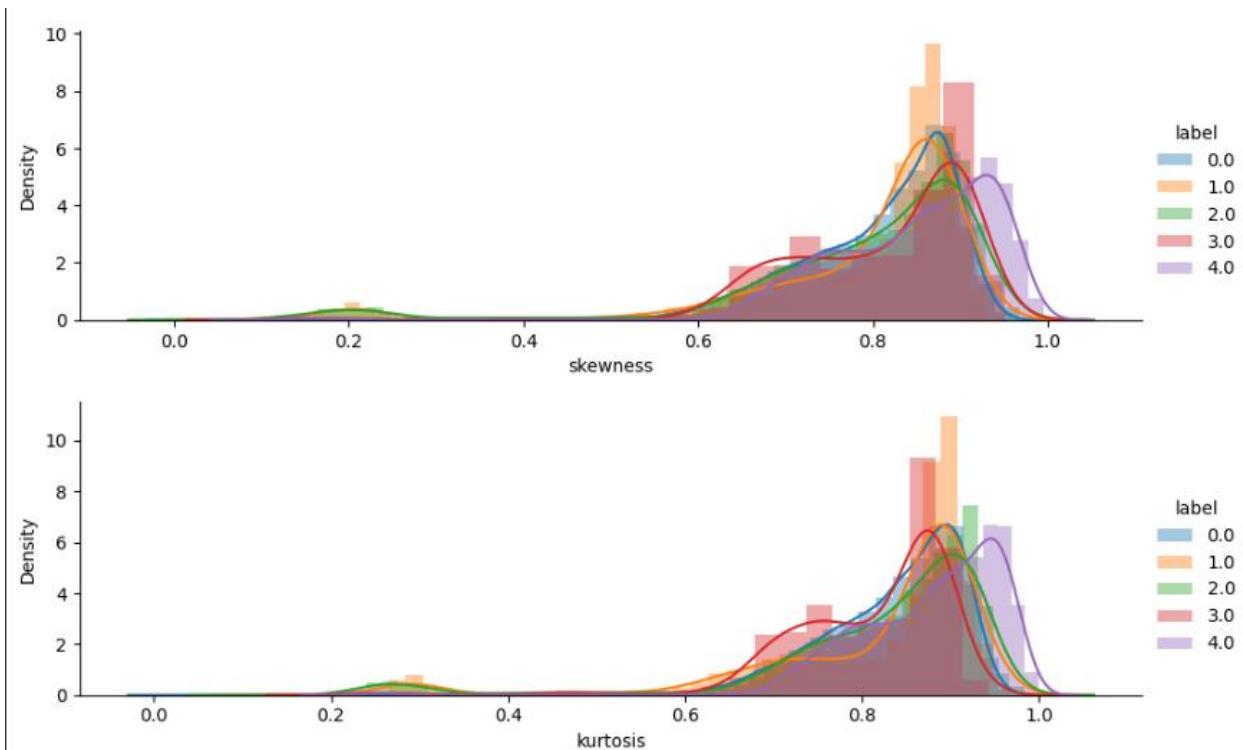
Din nou, mov si rosu, clasele 4 si 3 sunt mult diferite de celelalte clase, dar cumva ele fac parte dintr-un grup al lor, se aseamana intr-un fel, dar sunt pe phase diferit. In continuare, clasele 0 si 1 sunt usor de confundat, ca si clasa 2.



Pare ca aici existe niste diferente mai importante intre clasele 0 si 1, respectiv 4 si 3.



Greu de invatat ceva de aici.



Greu de invatat ceva de aici de asemenea.

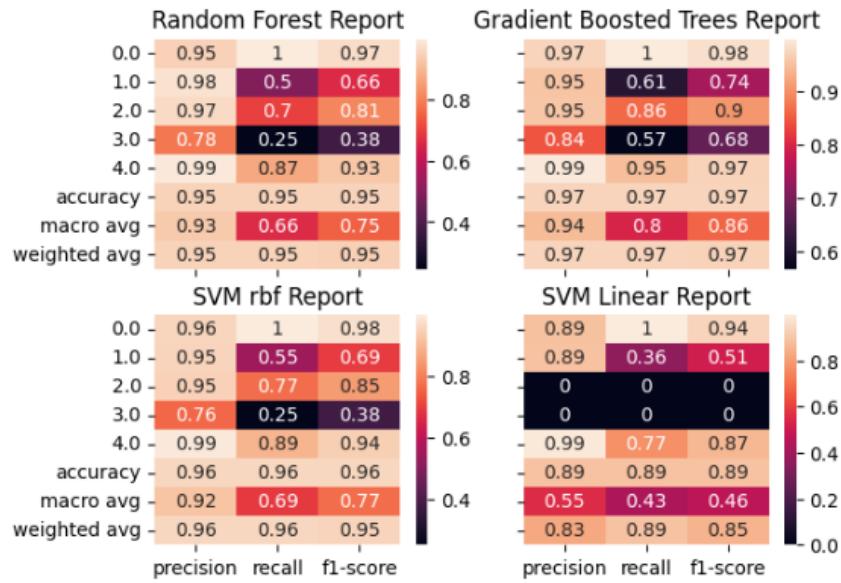
In concluzie, as spune ca setul de date este unul greu. In primul rand este un set de date dezechilibrat, apoi feature-urile nu sunt asa de folositoare ca in cazul RacketSports. De asemenea, am precizat clasele care pot fi usor de confundat: perechea 0-1 si perechea 3-4 si poate si 2.

Deoarece exista o varianta mai mica a datelor, folosind *MinMaxScaler()* peste feature-uri, filtrarea atributelor prin *VarianceThreshold()* s-a realizat cu un threshold = 0.015 obtinand din 28 de atribute, doar 16 in final. Orientarea pentru numarul 0.015 a fost realizata in functie de numarul de atribute ramase la final. Am considerat 16 un numar bun. De asemenea, am observat ca celelalte 12 atribute eliminate se pierdeau destul de repede. (si la un threshold mai mare)

3. Algoritmi de Invatare Automata

A se consulta fisierul [Proiect_ML.xlsx](#) pentru rezultate !

Initial, am creat 4 modele : RandomForest, Gradient Boosted Trees, SVC rbf si SVC Linear.



Se remarcă rezultate foarte bune pentru XGB. Este singurul algoritm care reușește să obțină un scor bun de recall și f1 pentru clasa 1 și clasa 3. (clasele dezavantajate de balansare)

In continuare, am încercat să imbunatătesc algoritmii încercând cale 6 exemple din fiecare model pentru a contura, înțelegând mai bine combinațiile pe care le-aș putea oferi în cadrul apelului de GridSearchCV.

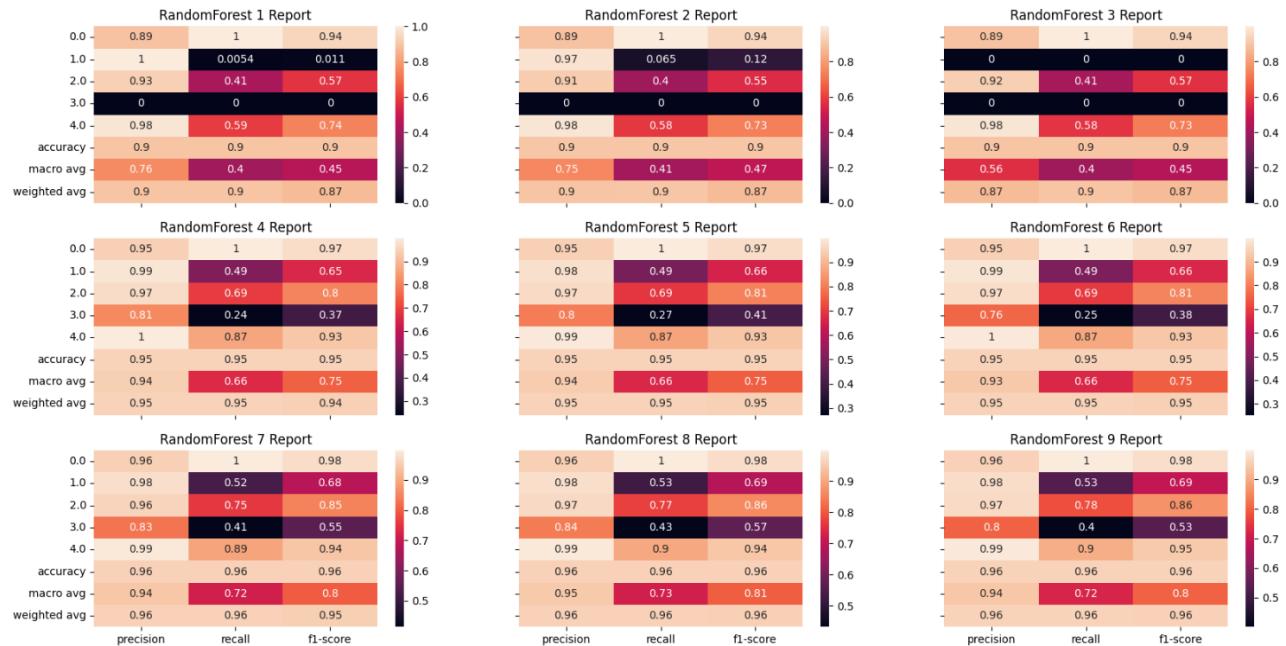
Am creat 6 modele de RandomForest:

```
# Define several instances of Random Forest classifier with different hyperparameters

random_forest_1 = RandomForestClassifier(n_estimators = 60, max_depth = 5, max_samples = 0.3)
random_forest_2 = RandomForestClassifier(n_estimators = 60, max_depth = 5, max_samples = 0.6)
random_forest_3 = RandomForestClassifier(n_estimators = 60, max_depth = 5, max_samples = 0.9)

random_forest_4 = RandomForestClassifier(n_estimators = 120, max_depth = 10, max_samples = 0.3)
random_forest_5 = RandomForestClassifier(n_estimators = 120, max_depth = 10, max_samples = 0.6)
random_forest_6 = RandomForestClassifier(n_estimators = 120, max_depth = 10, max_samples = 0.9)

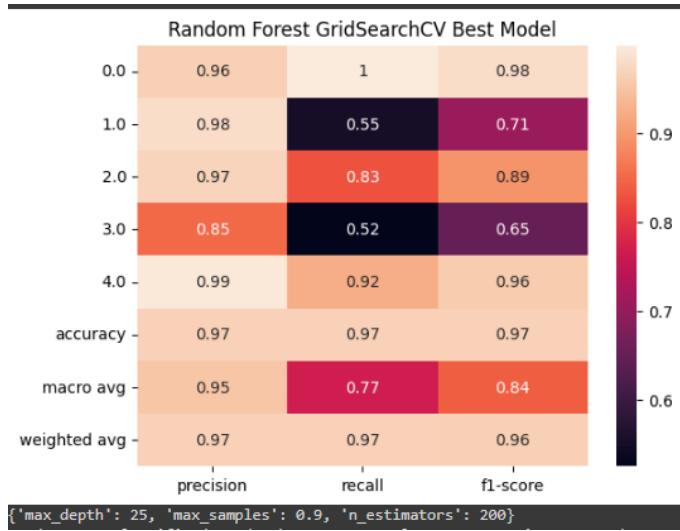
random_forest_7 = RandomForestClassifier(n_estimators = 80, max_depth = 15, max_samples = 0.3)
random_forest_8 = RandomForestClassifier(n_estimators = 80, max_depth = 15, max_samples = 0.6)
random_forest_9 = RandomForestClassifier(n_estimators = 80, max_depth = 15, max_samples = 0.9)
```



Dupa cum se poate vedea, modelele cu 60 de copaci si `max_depth = 5` sunt prea mici pentru a reusi sa clasifice cele 5 categorii de aritmii. Task-ul este mai greu. Se poate observa o imbunatatire pentru cazul cu 120 de copaci si `max_depth = 10`, dar chiar mai bine este cu 80 de copaci si `max_depth = 15`. Se pare ca `max_depth` influenteaza mai puternic spatiul de invatare, cat de mult poate invata modelul. Rezultate mai bune se obtin in cazul modelelor 7,8,9.

Deci, am rulat GridSearchCV cu parametrii:

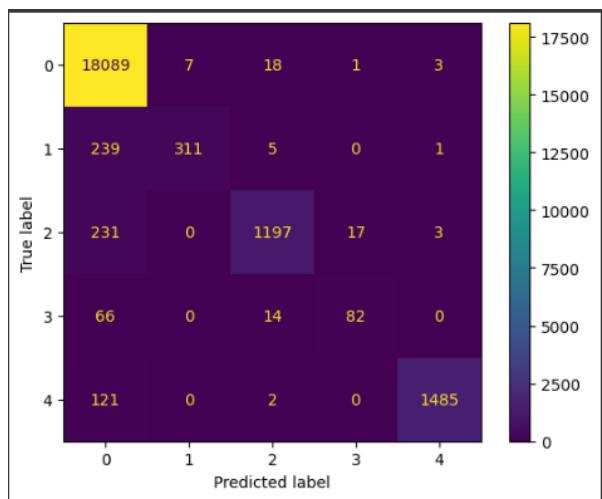
```
# Define a dictionary of hyperparameters for the Random Forest Classifier model
parameters = {
    'max_depth': [8, 13, 18, 25],
    'n_estimators': [80, 100, 150, 200],
    'max_samples': [0.5, 0.9]
}
```



De precizat : Numarul de parametrii este mai mic intrucat rularea a durat foarte mult.

Se poate remarcă un scor destul de bun la Recall pentru clasa 1 și clasa 3. Modelul a reusit să mai reducă din efectul de dezechilibru oferit de setul de date.

Parametrii alesi au fost: max_depth = 25, max_samples = 0.9, n_estimators = 200. Astfel, în cazul acestui task mai dificil se poate vedea că este preferat, spre deosebire de RacketSports atât un număr mare de copaci 200, cât și max_depth = 25, destul de mare.



Matricea de confuzie arată că aritmii din clasa 1 sunt ușor de confundat cu aritmii din clasa 0. De asemenea, aritmii din clasa 3 sunt ușor de confundat cu aritmii din clasa 4. Pentru că există aceste asemănări, task-ul este considerat greu.

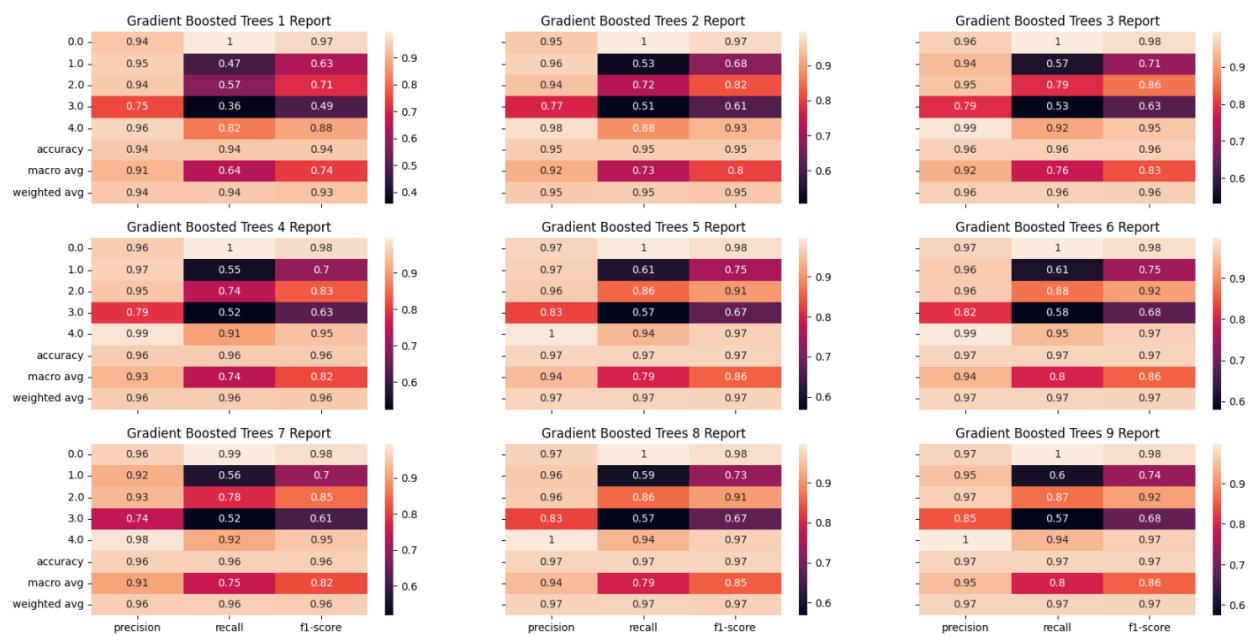
Am creat 6 modele de XGBoost:

```
# Define XGBClassifiers with different hyperparameters

gb_trees_1 = XGBClassifier(n_estimators = 60, max_depth = 5, learning_rate = 0.01)
gb_trees_2 = XGBClassifier(n_estimators = 60, max_depth = 5, learning_rate = 0.1)
gb_trees_3 = XGBClassifier(n_estimators = 60, max_depth = 5, learning_rate = 0.2)

gb_trees_4 = XGBClassifier(n_estimators = 120, max_depth = 10, learning_rate = 0.01)
gb_trees_5 = XGBClassifier(n_estimators = 120, max_depth = 10, learning_rate = 0.1)
gb_trees_6 = XGBClassifier(n_estimators = 120, max_depth = 10, learning_rate = 0.2)

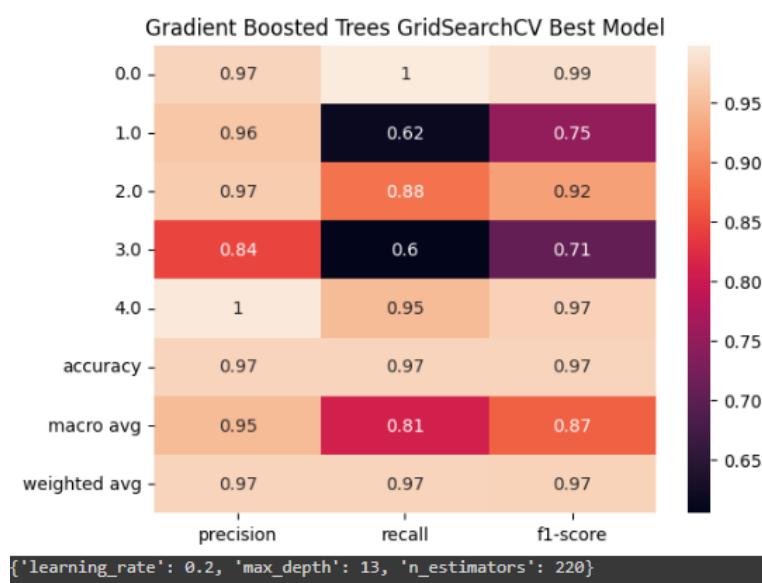
gb_trees_7 = XGBClassifier(n_estimators = 80, max_depth = 15, learning_rate = 0.01)
gb_trees_8 = XGBClassifier(n_estimators = 80, max_depth = 15, learning_rate = 0.1)
gb_trees_9 = XGBClassifier(n_estimators = 80, max_depth = 15, learning_rate = 0.2)
```



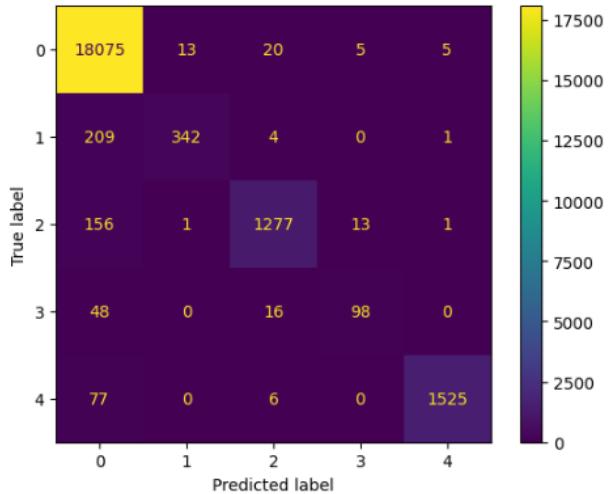
Se poate vedea ca un learning_rate = 0.01 este mult prea mic. De asemenea, rezultate un pic mai bune par a fi obtinute cu un learning_rate = 0.02 si cu un numar de copaci/adancime mai mare, precum modelele 5,6,8,9.

Am rulat GridSearchCV cu parametrii:

```
# Parameter grid for Grid Search Cross Validation
parameters = {
    'max_depth': [5, 13, 20, 25],
    'n_estimators': [70, 100, 170, 220],
    'learning_rate': [0.1, 0.2]
}
```



Dupa cum se poate vedea, a fost ales un numar mai mare de copaci = 200, si o adancime mai mare = 13, cu un learning_rate = 0.2. Motivele sunt aceleasi ca si un cazul modelului RandomForest, task-ul este greu si are nevoie de o capacitatea mai mare de invatare. Un numar mai mare de copaci se remarcă, parca aici fiind de preferat mai mult sa crească numărul de copaci decât adancime. Intuitiv, se spune că fenomenul de boosting ar prefera să aibă mai multă flexibilitate privind numărul de copaci, decât doar adancime mare.



Matricea de confuzie prezintă aceleasi caracteristici ca si la RandomForest, dar parca aici pare mai accentuat fenomenul de confuzie dintre clasele 0 si 1, precum 3 si 4.

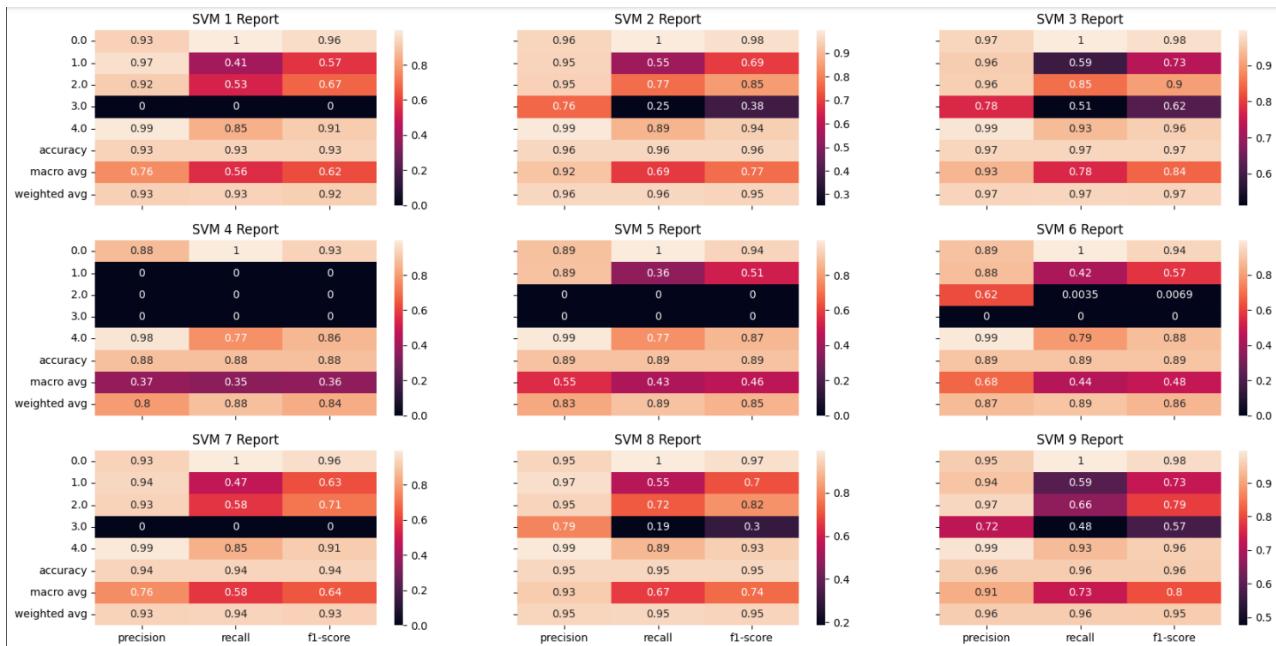
Am creat 6 modele de SVC:

```
# Define SVM Classifiers with different hyperparameters

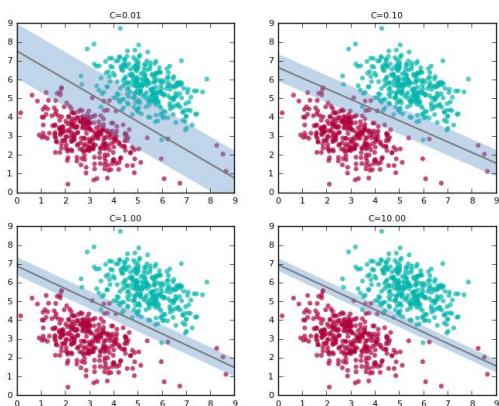
svm_1 = SVC(kernel = 'rbf', C = 0.1)
svm_2 = SVC(kernel = 'rbf', C = 1)
svm_3 = SVC(kernel = 'rbf', C = 10)

svm_4 = SVC(kernel = 'linear', C = 0.1)
svm_5 = SVC(kernel = 'linear', C = 1)
svm_6 = SVC(kernel = 'linear', C = 10)

svm_7 = SVC(kernel = 'poly', C = 0.1)
svm_8 = SVC(kernel = 'poly', C = 1)
svm_9 = SVC(kernel = 'poly', C = 10)
```



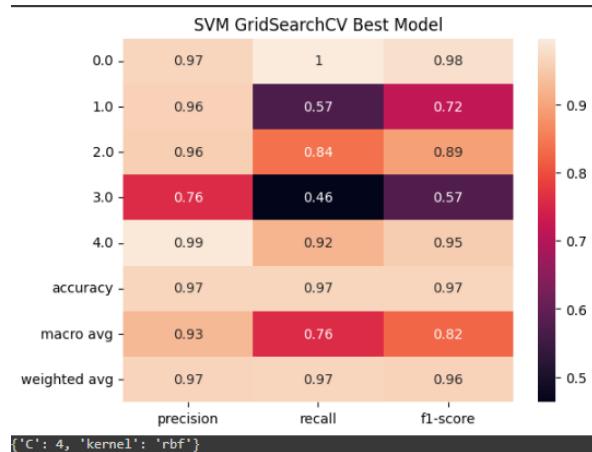
Asa cum se poate vedea, datele nu pot fi separate cu un kernel linear. Spatiul de caracteristici fiind complex, este nevoie de o supradimensionare, de un kernel poly sau rbf. Pare ca cel mai bun rezultat se obtine de aceasta data cu un $C > 1$, chiar 10. Algoritmul se perfectioneaza astfel si separa mai bine datele, marginea de separare fiind minimala, mai mica, tinand cont de mai putine puncte:



Am rulat GridSearchCV cu parametrii:

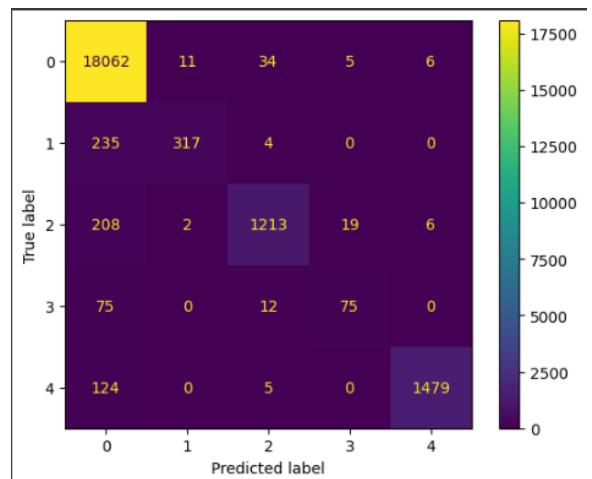
```
1 # Define a dictionary of parameter ranges to search over
2 parameters = {'kernel':['poly', 'rbf'], 'C':[0.8, 1, 2, 3, 4, 0.9, 0.7]}
3
```

Nu am pus un C mare. Am vrut sa explorez si zona de mijloc. Nu se obtin rezultate mai bune decat modelele 9,4 de mai sus.



S-a ales cel mai mare C care i-a fost dat : C = 4 si kernel = 'rbf'. Recall-ul nu obtine cel mai bun scor aici, desi acuratetea este buna.

Matricea de confuzie:



Dupa cum se poate vedea, din nou, clasa 1 este incurcata cu clasa 0 si clasa 3 este pe de o parte incurcata cu clasa 4, iar pe de alta parte, din cauza dezechilibrului tinde spre clasa 1. De altfel, toate clasele tind sa fie incurcate cu clasa 0. Prima coloana contine numere foarte mari.

MIT-BIH		
Numar de feature-uri considerate	28	
Numar de feature-uri la antrenare	16	
Random Forest Classifier		
n_estimators	max_depth	
60	5	
60	6	
60	9	
80	15	
80	6	
80	9	
120	10	
120	6	
120	9	
100	10	
200	25	
200	9	
XGB Classifier		
n_estimators	max_depth	learning_rate
60	5	0.01
60	5	0.1
60	5	0.2
80	15	0.01
80	15	0.1
80	15	0.2
120	10	0.01
120	10	0.1
120	10	0.2
100	6	-
220	13	0.2
SVC Classifier		
-	kernel	C
-	rbf	0.1
-	rbf	1
-	rbf	10
-	linear	0.1
-	linear	1
-	linear	10
-	poly	0.1
-	poly	1
-	poly	10
-	rbf	-
-	linear	1
-	poly	0.6

4. Retele neurale

In ce priveste rezolvarea task-ului prin retele neurale, nu mai sunt extrase manual feature-uri si tinand cont de acest lucru, am reanalizat graficele cu seriile de timp si am considerat ca ar fi mai bine sa elimin doar primele 5 si ultimele 5 coloane, pe considerente explicate mai sus.

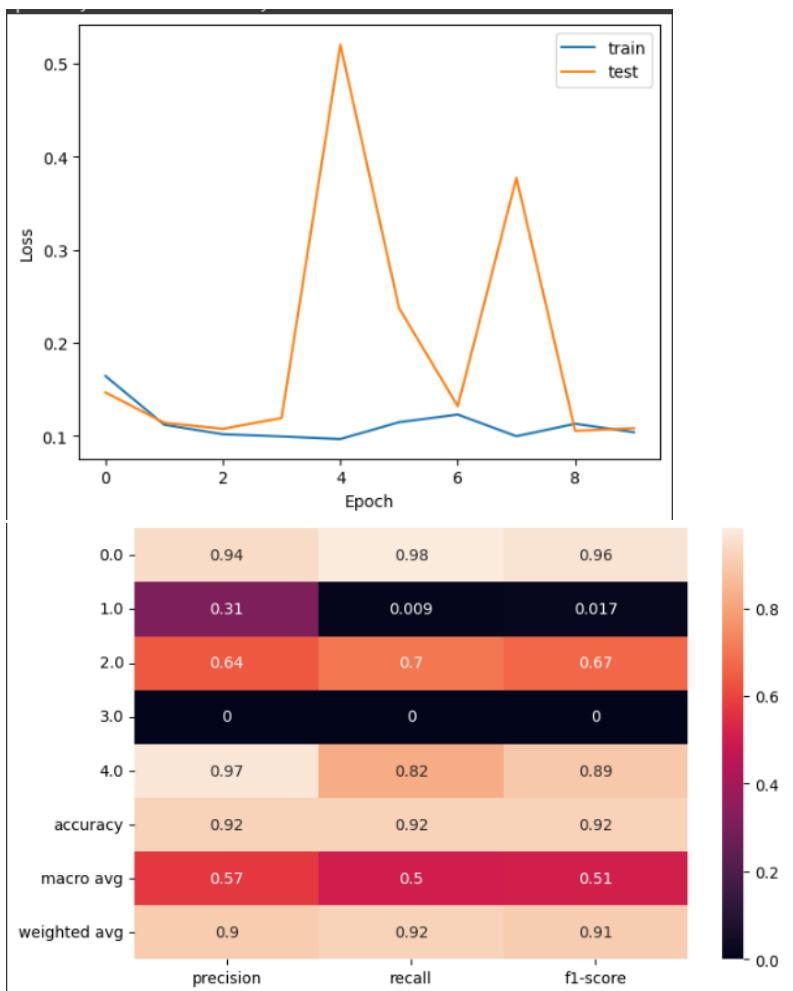
Dupa eliminarea coloanelor, am standardizat datele. Pe cale experimentală am observat ca se obtin rezultate mai bune astfel.

Prima arhitectura pe care am rulat este cea de MLP.

```

1  class MLP(nn.Module):
2      def __init__(self):
3          super(MLP, self).__init__()
4          self.fc1 = nn.Linear(178, 64)
5          self.fc2 = nn.Linear(64, 32)
6          self.fc3 = nn.Linear(32, 16)
7          self.fc4 = nn.Linear(16, 5)
8

```



Dupa cum se poate vedea, exista un learning_rate un pic cam mare. Totusi, un astfel de model reuseste sa evidenteze existenta clasei 1, ceea ce cred ca este remarcabil totusi.

Am incercat sa maresc arhitectura pentru a testa daca este nevoie de capacitate mai mare, asa cum intuiam. Deci am incercat antrenarea pentru aceasta arhitectura:

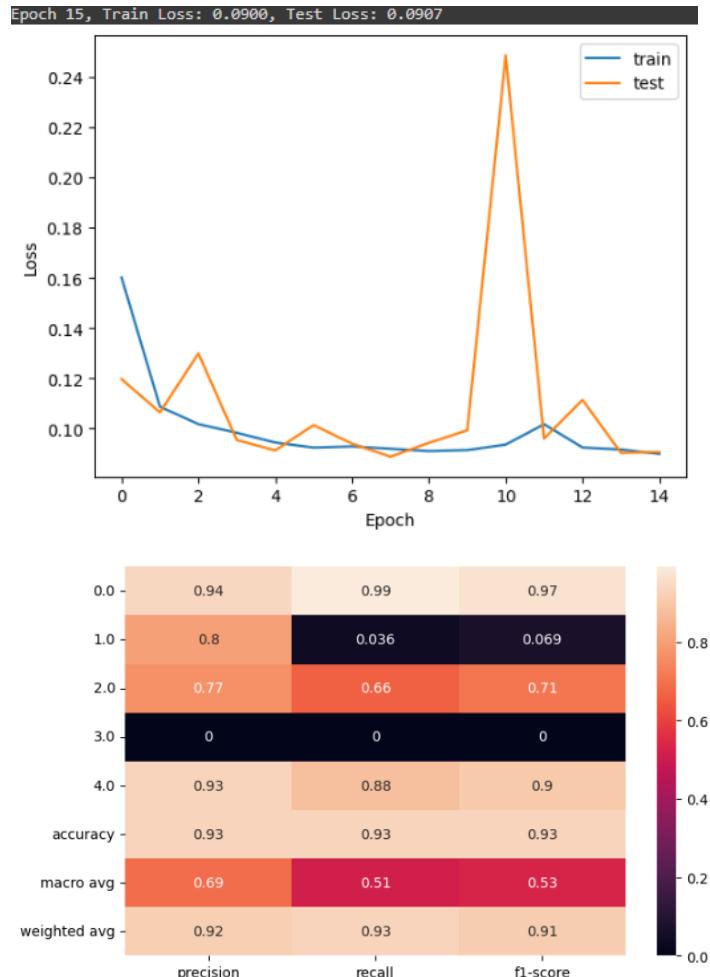
Arhitectură de tip MLP 2.0

```

1  class MLP2(nn.Module):
2      def __init__(self):
3          super(MLP2, self).__init__()
4          self.fc1 = nn.Linear(178, 128)
5          self.fc2 = nn.Linear(128, 64)
6          self.fc3 = nn.Linear(64, 48)
7          self.fc4 = nn.Linear(48, 32)
8          self.fc5 = nn.Linear(32, 16)
9          self.fc6 = nn.Linear(16, 5)
10
11     def forward(self, x):
12         x = x.view(-1, 178)
13         x = F.relu(self.fc1(x))
14         x = F.relu(self.fc2(x))
15         x = F.relu(self.fc3(x))
16         x = F.relu(self.fc4(x))
17         x = F.relu(self.fc5(x))
18         x = self.fc6(x)
19         #x = F.softmax(self.fc6(x), dim=1)
20         return x
21

```

Test-loss-ul reuseste sa scada mai mult:



Imbunatatirea este minimala, totusi se reuseste o convergenta, chiar daca learning_rate este un pic mai mare.

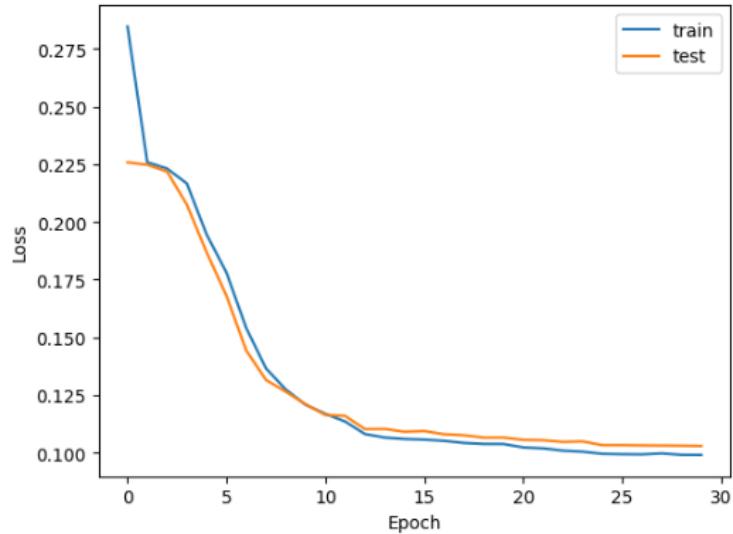
Urmatoarea varianta este o incercare de reglare a learning-rate-ului prin folosirea unui scheduler

```
model = MLP2()

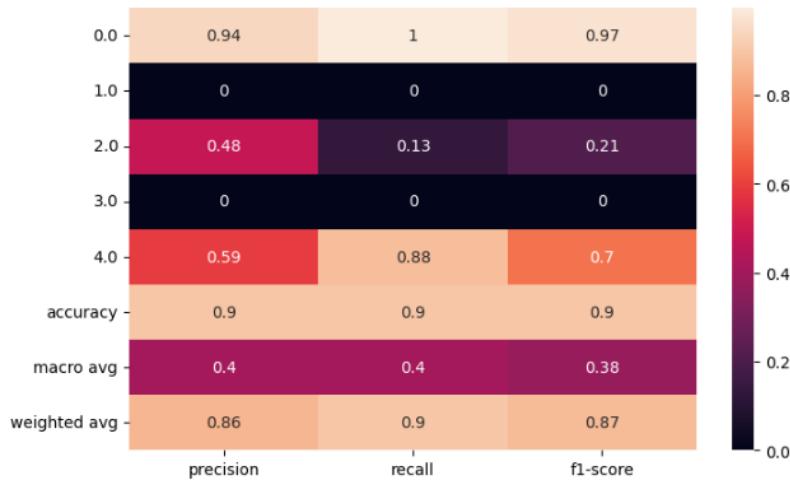
# define the loss function and optimizer
criterion = nn.BCEWithLogitsLoss()

# define the optimizer with SGD and momentum
optimizer = torch.optim.SGD(model.parameters(), lr=0.02, momentum=0.9, weight_decay=0.001)

# define the learning rate scheduler
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=12, gamma=0.1)
```



Deci se obtine o curba mai frumoasa de loss, totusi nu se obtin si rezultate mai bune:



Clasa 0 acapareaza din nou celelalte clase mai mici. Modelul nu reuseste sa mai vada diferentele.

Concluzie: MLP este un model prea mic pentru a reusi sa acapareze toata informatia necesara pentru a rezolva un task asa de greu. Chiar si cu o arhitectura mai complex, nu este indeajuns.

A doua arhitectura pe care am rulat este InceptionTime.

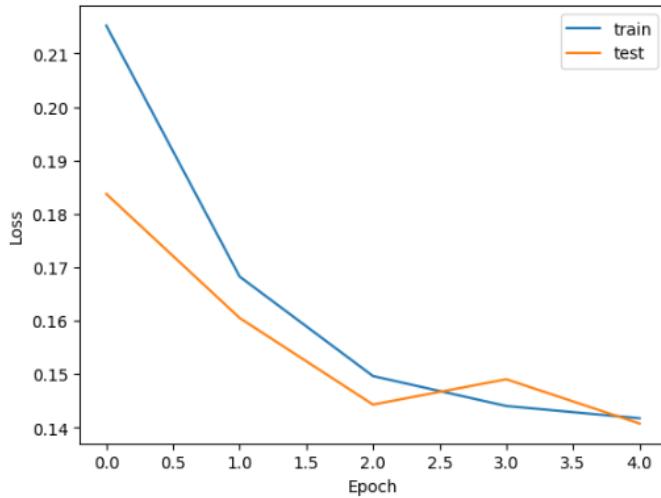
Parametrii cu care am rulat la inceput au fost acestia:

```
model = InceptionModel(num_blocks=3, in_channels=1, out_channels=2,
                      bottleneck_channels=2, kernel_sizes=41, use_residuals=True,
                      num_pred_classes=5)

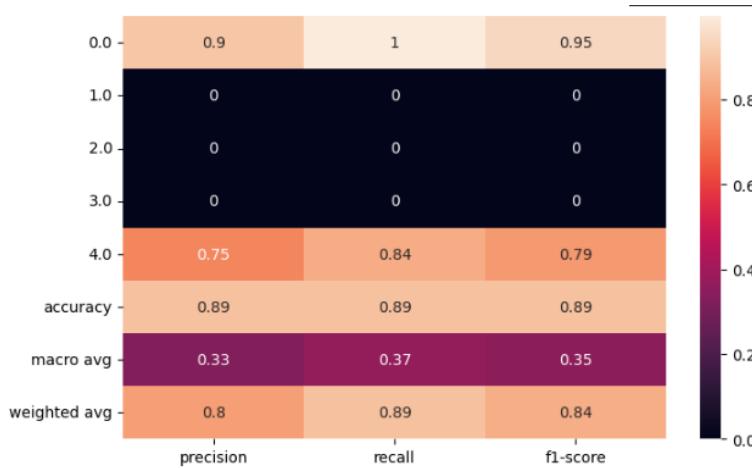
# define the loss function and optimizer
criterion = nn.BCEWithLogitsLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001, weight_decay=0.001)

# lists to store the loss values for plotting
train_losses = []
test_losses = []

# train the model
for epoch in range(5):
```



Am observat ca exista potential in ce priveste curba de loss, probabil numarul de epoci este prea mic de asemenea. Rezultatele nu sunt bune:



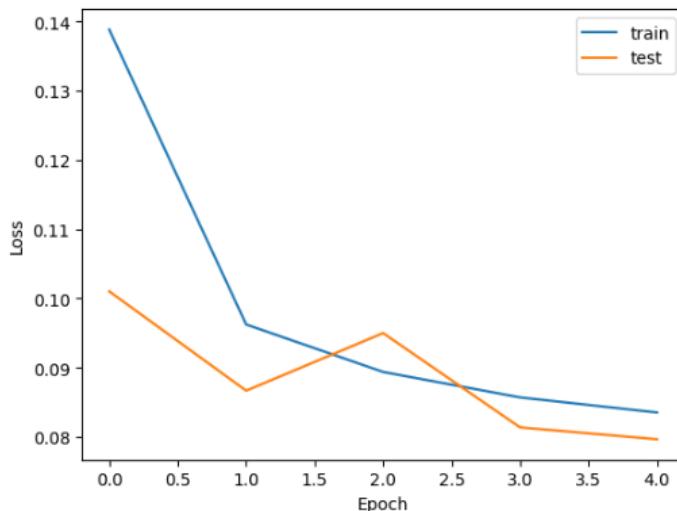
Am incercat, pentru inceput, cresterea kernel_size-ului si a numarului de bottleneck_channels:

```
model2 = InceptionModel(num_blocks=3, in_channels=1, out_channels=8,
                        bottleneck_channels=8, kernel_sizes=61, use_residuals=True,
                        num_pred_classes=5)

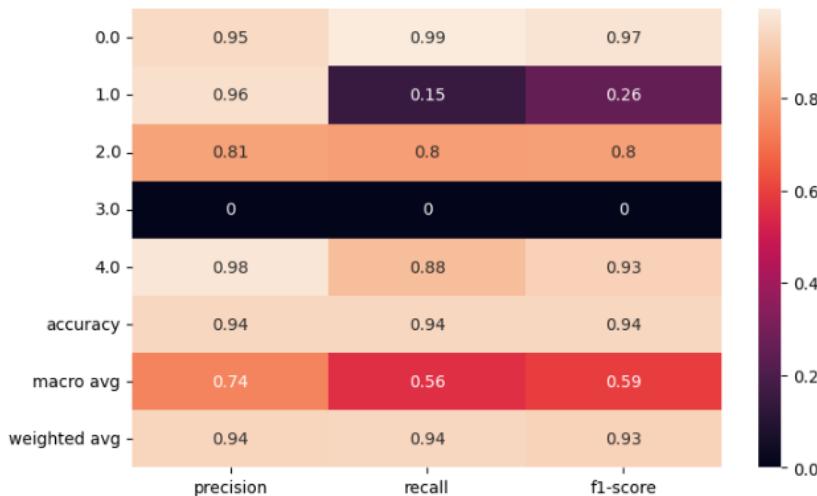
# define the loss function and optimizer
criterion = nn.BCEWithLogitsLoss()
optimizer = torch.optim.Adam(model2.parameters(), lr=0.001, weight_decay=0.001)

# lists to store the loss values for plotting
train_losses = []
test_losses = []

# train the model
for epoch in range(5):
    epoch_train_loss = 0
```

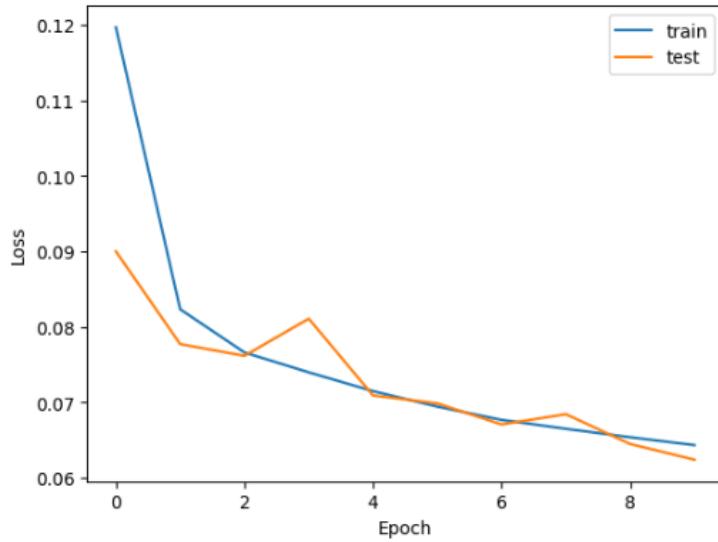


Se poate observa din nou potential in ce priveste curba de loss. Numarul de epoci cred ca in continuare este cam mic. Modelul reuseste totusi sa conveargă mai bine. Deci nu se face overfit, modelul anterior fiind cu siguranta intr-un regim de underfit. Exista mai mult potential in cazul acestei arhitecturi. Rezultatele obtinute sunt :

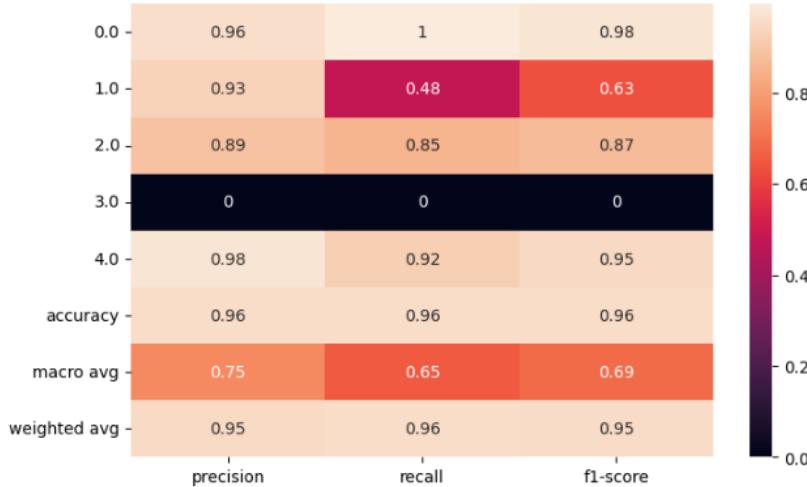


Deci, există îmbunătățiri în ce privește clasa 1.

Pe cale experimentală am luat în considerare și rularea acesta: creșterea numărului de block-uri de la 3 la 4 și creșterea numărului de epoci până la 10.



Se observă o convergență foarte bună. De asemenea, există potențial de convergență. Curba de loss a dataset-ului de test converge în același ritm cu curba de loss de la train. Deci nu există overfit, și traiectoria pare să fie spre un loss tot mai mic. Rezultatele obținute sunt :



Clasa 1 este observată. Clasa 3 nu este observată, dar este un task foarte greu acesta. Numărul de exemple din dataset din clasa 3 este cel mai mic. Fără strategie de feature-extraction manuală, pentru o rețea neurală să descopere singura acestea feature-uri specifice clasei 3 este destul de greu. În plus, aceasta este usor de incurcat cu clasa 4. Totuși modelul, obține acum rezultate mult mai bune, observându-se un recall de 0.48 pentru clasa 1.

Am realizat o schimbare în arhitectura modelului InceptionTime. Am adăugat un layer de dropout de 0.1 înainte de layer-ul linear pentru clasificarea aritmilor.

De asemenea, observand potentialul arhitecturii, dar si faptul ca un layer de dropout pierde 10% din informatie la fiecare inferenta, am crescut numarul de block-uri la 6 si numarul de epoci la 15.

```

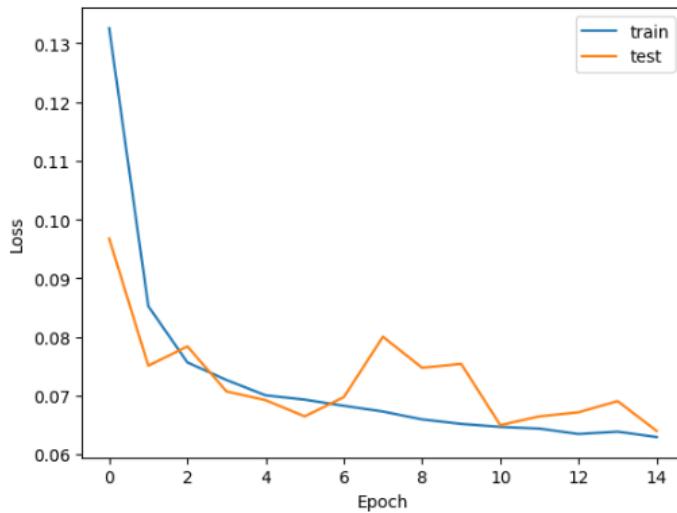
model3 = InceptionModel(num_blocks=6, in_channels=1, out_channels=8,
                        bottleneck_channels=8, kernel_sizes=[1, 3, 3, 3, 3, 1],
                        use_residuals=True,
                        num_pred_classes=5)

# define the loss function and optimizer
criterion = nn.BCEWithLogitsLoss()
optimizer = torch.optim.Adam(model3.parameters(), lr=0.001, weight_decay=0.001)

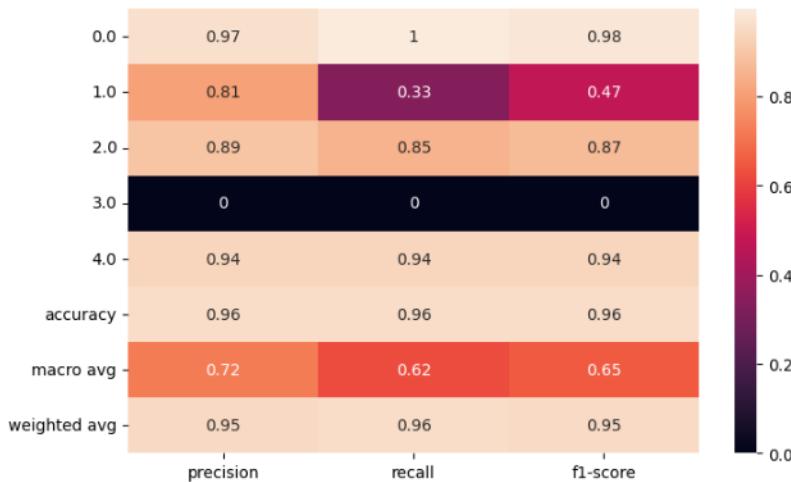
# lists to store the loss values for plotting
train_losses = []
test_losses = []

# train the model
for epoch in range(15):
    ...

```



Curba de loss pare sa fie mai putin optimista acum, ruland 15 epoci, s-a trecut de partea de “cot”. Observand curba intre epocile 5-15, se poate constata un learning_rate prea mare pentru aceasta perioada de train. Rezultatele, de asemenea, nu sunt mai bune, asa cum ma asteptam:



La urmatoarea rulare, am introdus un scheduler pentru controlul learning_rate-ului. Pe cale experimentală am stabilit urmatorii parametri:

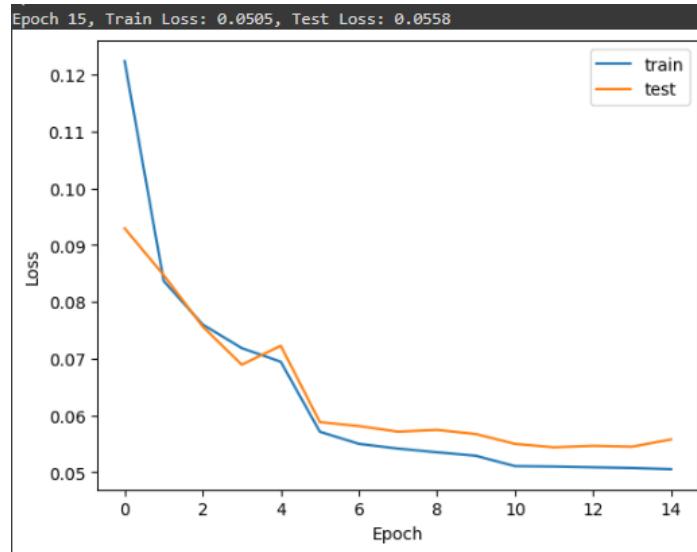
```
'model2 = InceptionModel(num_blocks=6, in_channels=1, out_channels=8,
                           bottleneck_channels=8, kernel_sizes=[1, 3, 5, 3, 1],
                           use_residuals=True,
                           num_pred_classes=5).to(device)

# define the loss function and optimizer
criterion = nn.BCEWithLogitsLoss()
optimizer = torch.optim.Adam(model2.parameters(), lr=0.001, weight_decay=0.001)

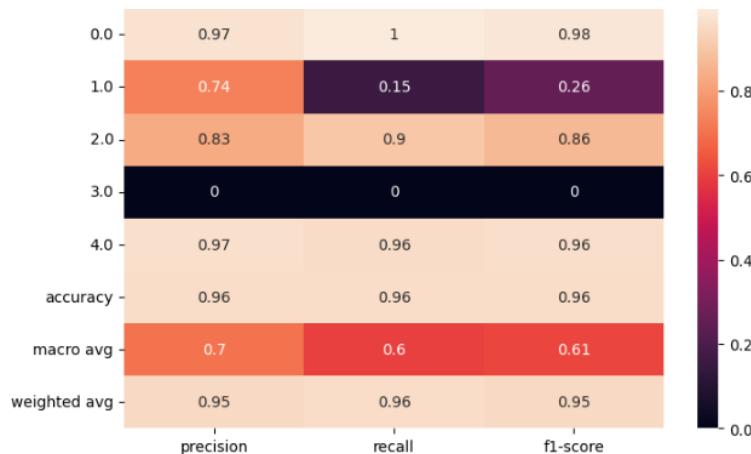
# define the learning rate scheduler
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=5, gamma=0.1)

# lists to store the loss values for plotting
train_losses = []
test_losses = []

# train the model
for epoch in range(15):
    # ... training code ...
    # ... validation code ...
    train_losses.append(train_loss)
    test_losses.append(test_loss)
```



Curba de loss arată mult mai bine și converge mai bine, ajungând la un loss de 0.055. Rezultatele nu sunt totuși mai bune:



Clasa 1 isi pierde din scorul de recall, iar clasa 3 ramane in continuare cu scoruri de 0. Desi se obtine o acuratate de 0.96, algoritmul nu reuseste sa generalizeze mai bine.

Am incercat reducere numarului de block-uri la 4. Am schimbat si kernel-size-ul la 41 de la 61. Folosind tot un scheduler, dar un batch de 16 elemente (in loc de 32) am obtinut rezultatele:

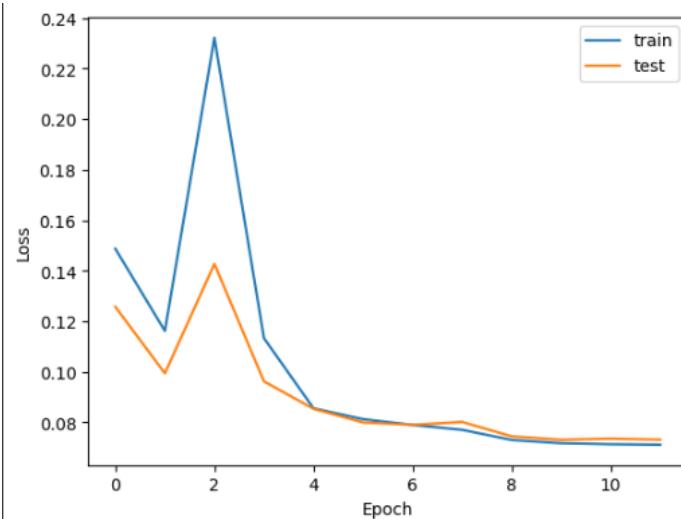
```
model2 = InceptionModel(num_blocks=4, in_channels=1, out_channels=8,
                        bottleneck_channels=8, kernel_sizes=41, use_residuals=True,
                        num_pred_classes=5).to(device)

# define the loss function and optimizer
criterion = nn.BCEWithLogitsLoss()
optimizer = torch.optim.Adam(model2.parameters(), lr=0.002, weight_decay=0.001)

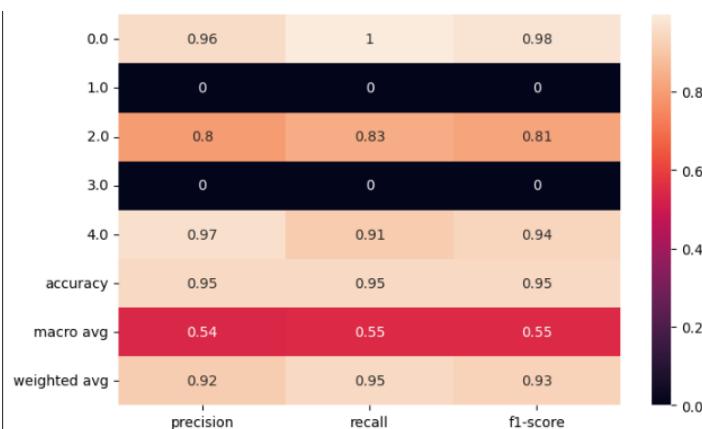
# define the learning rate scheduler
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=4, gamma=0.1)

# lists to store the loss values for plotting
train_losses = []
test_losses = []

# train the model
for epoch in range(12):
    train_loss, test_loss = train_and_val(model2, criterion, optimizer, scheduler, device)
```

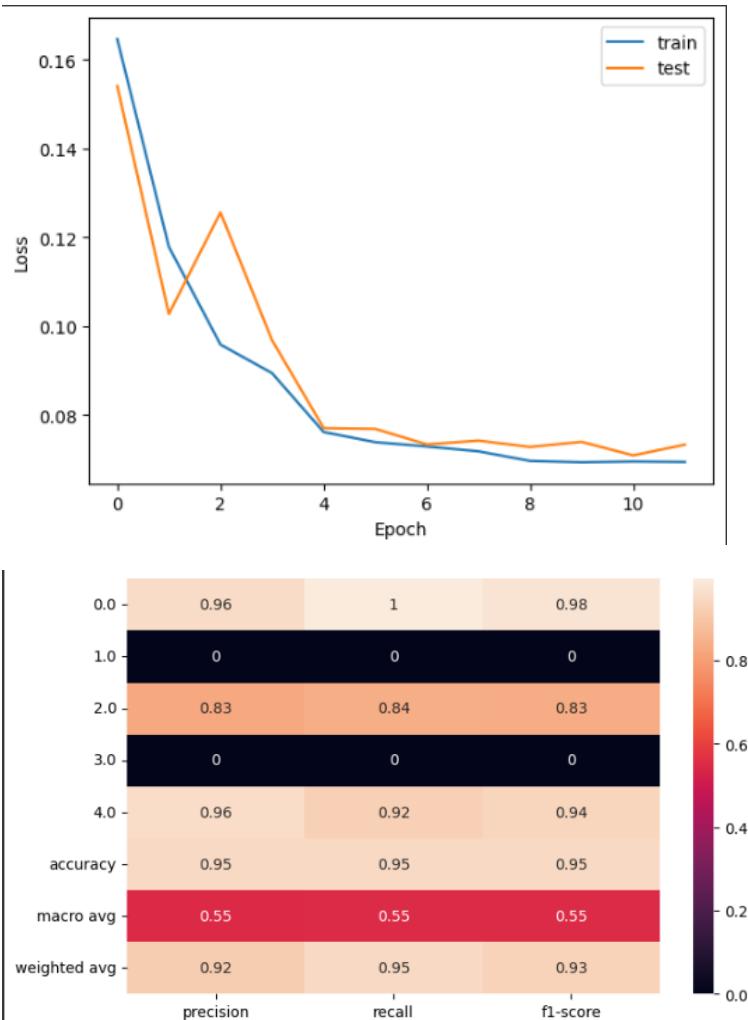


Curba de loss arata bine, desi nu converge la un loss foarte mic.



Rezultatele sunt si mai slabe.

In continuare, pentru a intelege cată influență a avut un batch_size = 16, am rulat din nou, dar de data aceasta schimbând batch_size-ul = 128.



Nici de data aceasta nu se obțin rezultate mai bune. Batch-size-ul se dovedește a nu schimba foarte mult rezultatele.

Am încercat și o altă arhitectură de InceptionTime. De data aceasta, am schimbat layer-ul linear și un modul de MLP 3.0 atasat la sfârșit. Există în continuare un dropout = 0.1. De asemenea, am folosit în continuare un scheduler.

```

model = InceptionModel(num_blocks=4, in_channels=1, out_channels=8,
                      bottleneck_channels=8, kernel_sizes=41, use_residuals=True,
                      num_pred_classes=5)

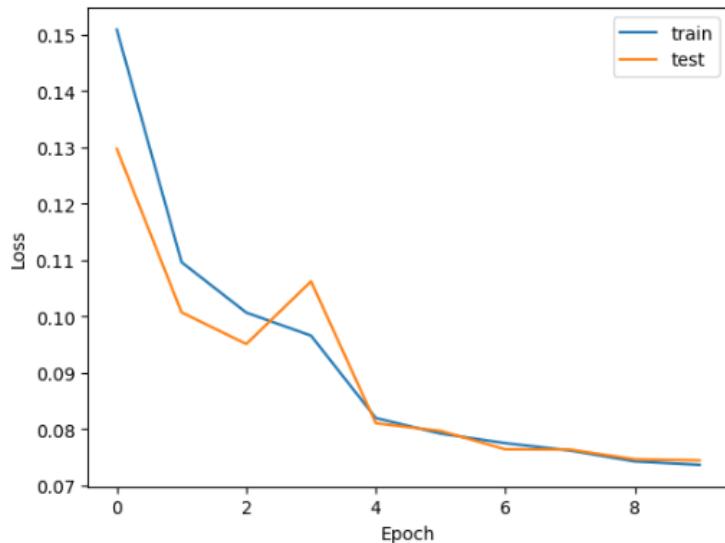
# define the loss function and optimizer
criterion = nn.BCEWithLogitsLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.002, weight_decay=0.001)

# define the learning rate scheduler
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=4, gamma=0.1)

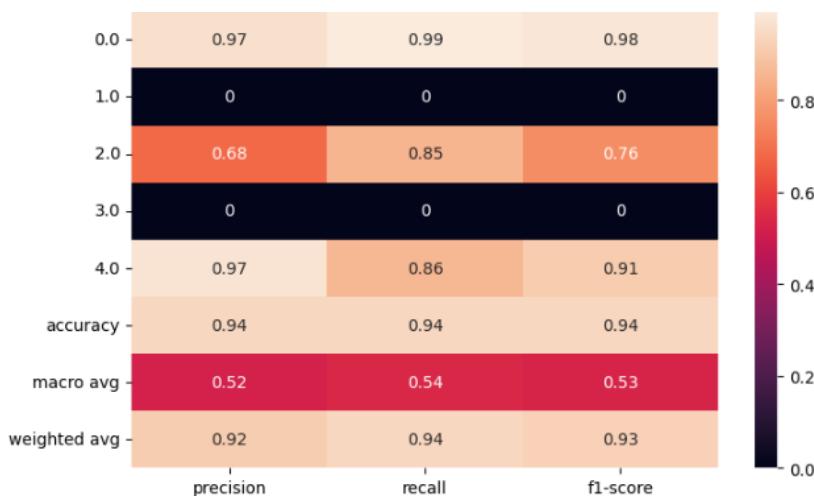
# lists to store the loss values for plotting
train_losses = []
test_losses = []

# train the model
for epoch in range(10):
    train_loss = 0.0
    test_loss = 0.0
    ...
    train_losses.append(train_loss)
    test_losses.append(test_loss)

```



Curba de loss arata bine, dar nu converge foarte mult.



Nu genereaza atat de bine. Rezultatele nu sunt mai bune ca inainte.

In concluzie, dropout-ul nu a ajutat deloc. Batch-size-ul nu prea a influentat rezultatele. Singurele lucruri care au ajutat la o generalizare mai bune au fost numarul de block-uri din modul Inception, numarul de canale bottleneck si kernel-size-ul.

InceptionTime reuseste totusi sa gaseasca mai multe feature-uri decat un modul MLP.

		MIT-BIH																									
		MLP																									
batch_size	Architectura	epochs / lr	dropout	step_size / gamma	0			1			2			3			4			Media			Varianta	Acuratetea			
					Precision	Recall	F1-score																				
32	178 / 64 / 32 / 16 / 5	10 / 0.01	-	-	0.94	0.98	0.96	0.31	0.009	0.02	0.64	0.7	0.67	0.00	0	0.00	0.97	0.82	0.89	0.57	0.50	0.51	0.174	0.216	0.219	0.92	
MLP 2.0																				Media			Varianta				
32	178 / 128 / 64 / 48 / 32 / 16 / 5	15 / 0.01	-	-	0.94	0.99	0.97	0.80	0.036	0.07	0.77	0.66	0.71	0.00	0	0.00	0.93	0.88	0.90	0.69	0.51	0.53	0.154	0.219	0.214	0.93	
MLP 3.0																				Media			Varianta				
32	178 / 128 / 64 / 48 / 32 / 16 / 5	30 / 0.02	-	12 / 0.1	0.94	1	0.97	0.00	0	0.00	0.48	0.13	0.21	0.00	0	0.00	0.59	0.88	0.7	0.31	0.33	0.32	0.295	0.333	0.314	0.9	
Inception Time																				Media			Varianta				
32	num_blocks / out_ch / bottleneck_ch / kernel_size	epochs / lr	dropout	step_size / gamma	0			1			2			3			4			Media			Varianta			Acuratetea	
32	3 / 2 / 2 / 41	5 / 0.001	-	-	0.90	1	0.95	0.00	0	0.00	0	0.00	0	0.00	0	0.00	0.75	0.84	0.79	0.33	0.37	0.35	0.207	0.257	0.230	0.89	
32	3 / 8 / 8 / 61	5 / 0.001	-	-	0.95	0.99	0.97	0.96	0.15	0.26	0.81	0.8	0.80	0.00	0	0.00	0.98	0.88	0.93	0.74	0.56	0.59	0.176	0.207	0.190	0.94	
32	4 / 8 / 8 / 61	10 / 0.001	-	-	0.96	1	0.98	0.93	0.48	0.63	0.89	0.85	0.87	0.00	0	0.00	0.98	0.92	0.95	0.75	0.65	0.69	0.178	0.172	0.166	0.96	
InceptionTime 2.0																				Media			Varianta				
32	num_blocks / out_ch / bottleneck_ch / kernel_size	epochs / lr	dropout	step_size / gamma	0			1			2			3			4			Media			Varianta			Acuratetea	
32	6 / 8 / 8 / 61	15 / 0.001	0.1	-	0.97	1	0.98	0.81	0.33	0.47	0.89	0.85	0.87	0.00	0	0.00	0.94	0.94	0.94	0.72	0.62	0.65	0.167	0.192	0.174	0.96	
32	6 / 8 / 8 / 61	15 / 0.001	0.1	5 / 0.1	0.97	1	0.98	0.74	0.15	0.26	0.83	0.9	0.86	0.00	0	0.00	0.97	0.96	0.96	0.57	0.50	0.49	0.257	0.500	0.480	0.96	
16	4 / 8 / 8 / 41	12 / 0.002	0.1	4 / 0.1	0.95	1	0.98	0.00	0	0.00	0.8	0.83	0.81	0.00	0	0.00	0.97	0.91	0.94	0.32	0.33	0.33	0.307	0.333	0.320	0.95	
128	4 / 8 / 8 / 41	12 / 0.002	0.1	4 / 0.1	0.96	1	0.98	0.00	0	0.00	0.83	0.84	0.83	0.00	0	0.00	0.96	0.92	0.94	0.55	0.55	0.55	0.255	0.257	0.255	0.95	
InceptionTime 3.0 (InceptionTime 2.0 + MLP 3.0)																				Media			Varianta				
32	num_blocks / out_ch / bottleneck_ch / kernel_size	epochs / lr	dropout	step_size / gamma	0			1			2			3			4			Media			Varianta			Acuratetea	
32	4 / 8 / 8 / 41	10 / 0.002	0.1	4 / 0.1	0.97	0.99	0.98	0.00	0	0.00	0.68	0.85	0.76	0.00	0	0.00	0.97	0.86	0.91	0.52	0.54	0.53	0.243	0.246	0.240	0.94	

In continuare, am incercat rezolvarea task-ului cu ajutorul unei arhitecturi LSTM.

```
class ComplexLSTM(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, output_size):
        super(ComplexLSTM, self).__init__()

        self.input_size = input_size
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.output_size = output_size

        self.lstm = nn.LSTM(input_size, hidden_size, num_layers, batch_first=True)
        self.fc1 = nn.Linear(hidden_size, 64)
        self.fc2 = nn.Linear(64, output_size)

    def forward(self, x):
        # Initialize the hidden state and cell state
        h0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(x.device)
        c0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(x.device)

        # Forward pass through the LSTM layer
        out, (hn, cn) = self.lstm(x, (h0, c0))

        # Pass the final hidden state through Linear layers
        out = self.fc1(hn[-1])
        out = self.fc2(out)

        return out
```

Am inceput experimentele:

Am setat un `hidden_size = 50`, si numar de layers `num_layers = 1`. Pentru un `batch_size = 256` si parametrii de mai jos, rezultatele au fost:

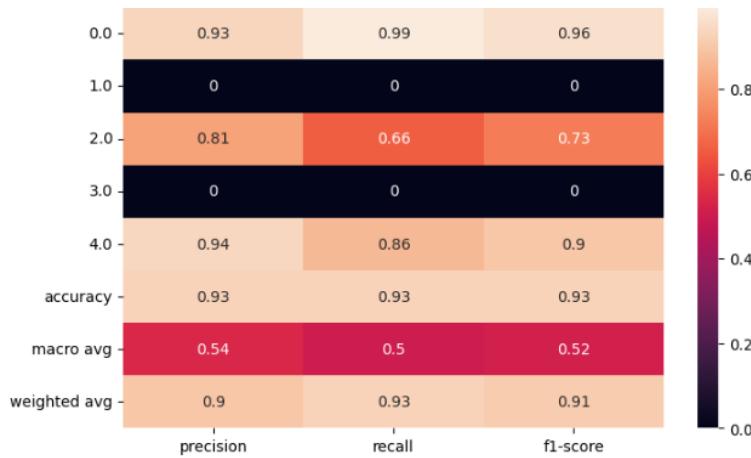
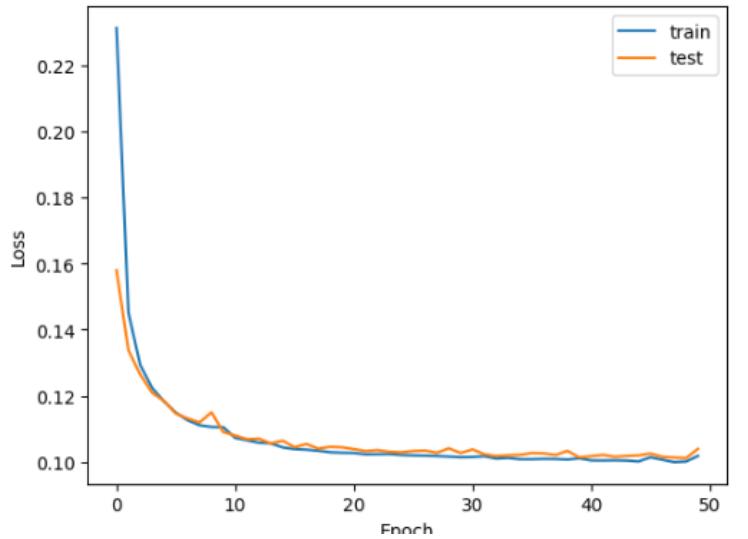
```
model = ComplexLSTM(input_size = 178, hidden_size = 50, num_layers = 1, output_size = 5)

# define the loss function and optimizer
criterion = nn.BCEWithLogitsLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001, weight_decay=0.001)

# lists to store the loss values for plotting
train_losses = []
test_losses = []

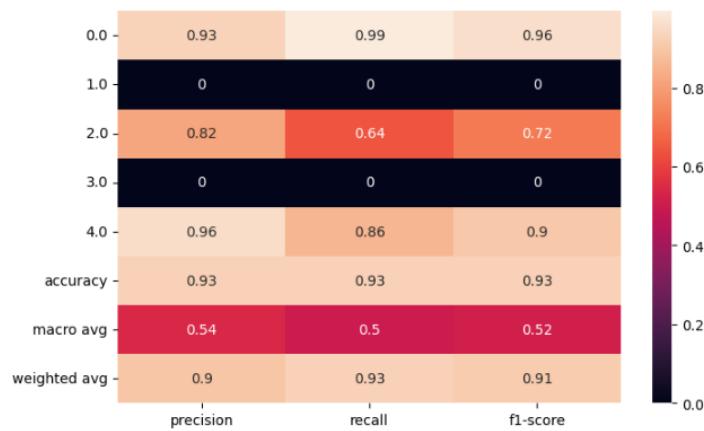
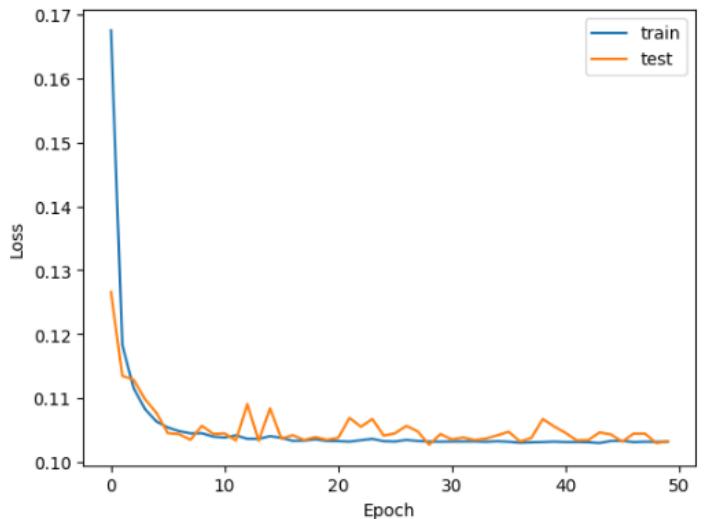
# train the model
for epoch in range(50):
    # training loop
    # ...
    # testing loop
    # ...
    train_losses.append(train_loss)
    test_losses.append(test_loss)

    if epoch % 5 == 0:
        print(f'Epoch {epoch}: Train Loss: {train_loss}, Test Loss: {test_loss}')
```



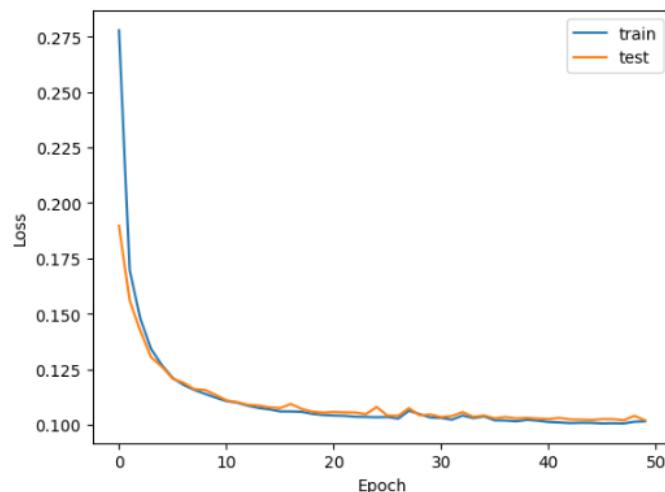
Curba de loss pare saturata, dar rezultatele nu sunt foarte bune, comparandu-le cu InceptionTime. Nu se generalizeaza suficient pentru a observa clasele 1 si 3. De asemenea, recall-ul este mic si pentru clasa 2.

Am incercat sa intieleg influenta batch size-ului. Asa ca am incercat sa observ rezultatele pentru `batch_size = 64`



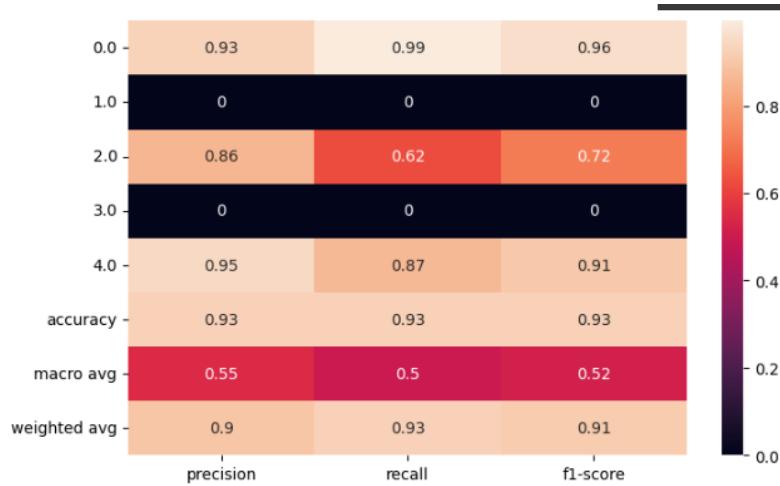
Se converge asemanator, dar exista mai mult noise. Comparand-o cu rularea anterioara, consider ca putin noise este sanatos pentru a nu se opri intr-un minim local, dar nereusind totusi sa converaga mai mult, consider rularea anterioara mai buna. In continuare, rezultatele nu sunt bune.

Am incercat si cu un batch_size = 512.



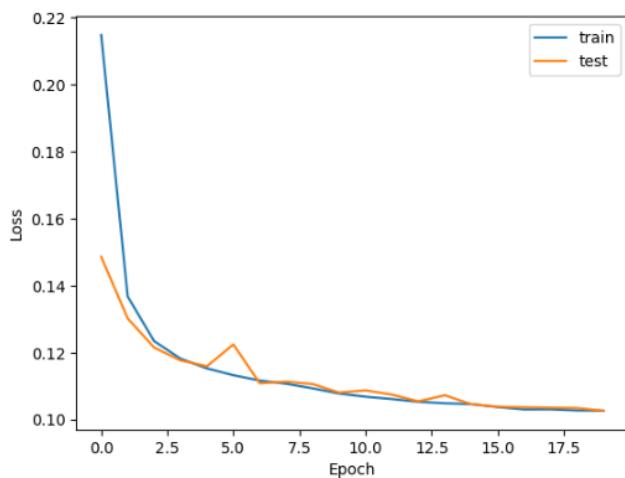
Se observa lipsa noise-ului, dar totusi convergenta ramane la fel.

Rezultatele par asemanatoare:

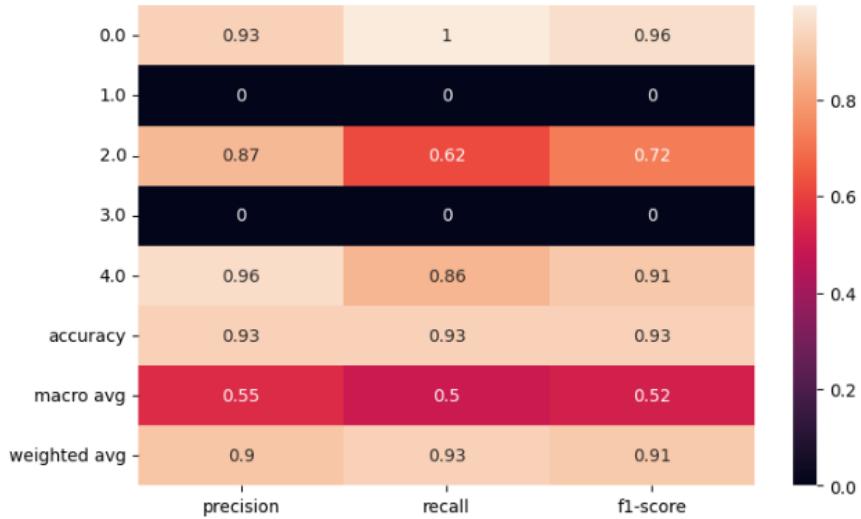


Deci, daca batch_size-ul nu influenteaza foarte mult, m-am intors la un batch_size = 256, dar am schimbat hidden_size-ul la 100.

```
model = ComplexLSTM(input_size = 178, hidden_size = 100, num_layers = 1, output_size = 5)
```



Curba de loss pare ceva mai buna, dar rezultatele nu par a fi schimbate.



Am incercat o arhitectura putin schimbata, prin introducere unui layer de dropout:

```
class ComplexLSTM(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, output_size):
        super(ComplexLSTM, self).__init__()

        self.input_size = input_size
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.output_size = output_size

        self.lstm = nn.LSTM(input_size, hidden_size, num_layers, batch_first=True)
        self.fc1 = nn.Linear(hidden_size, 64)
        self.fc2 = nn.Linear(64, output_size)
        # dropout
        self.dropout = nn.Dropout(0.1)

        # ReLU activation
        self.relu = nn.ReLU()

    def forward(self, x):
        # Initialize the hidden state and cell state
        h0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(x.device)
        c0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(x.device)

        # Forward pass through the LSTM layer
        out, (hn, cn) = self.lstm(x, (h0, c0))
        out = self.dropout(out)
        out = self.relu(out)
        # Pass the final hidden state through Linear layers
        out = self.fc1(hn[-1])
        out = self.fc2(out)

        return out
```

Pentru a contracara efectul de dropout am crescut la 0.002 learning_rate-ul:

```

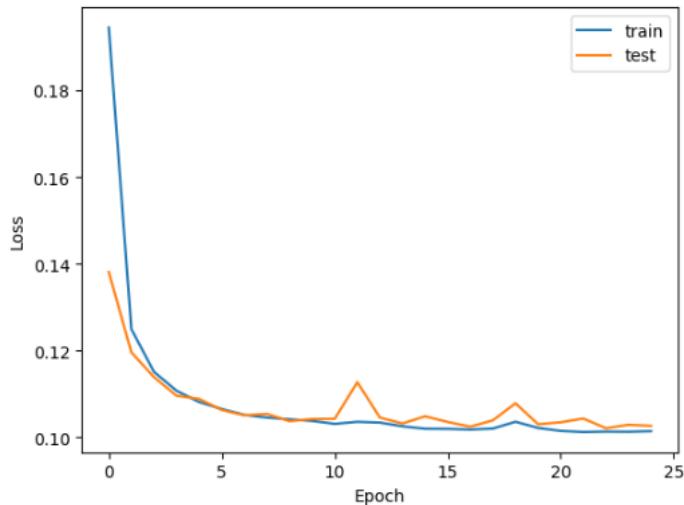
model = ComplexLSTM(input_size = 178, hidden_size = 50, num_layers = 1, output_size = 5)

# define the loss function and optimizer
criterion = nn.BCEWithLogitsLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.002, weight_decay=0.001)

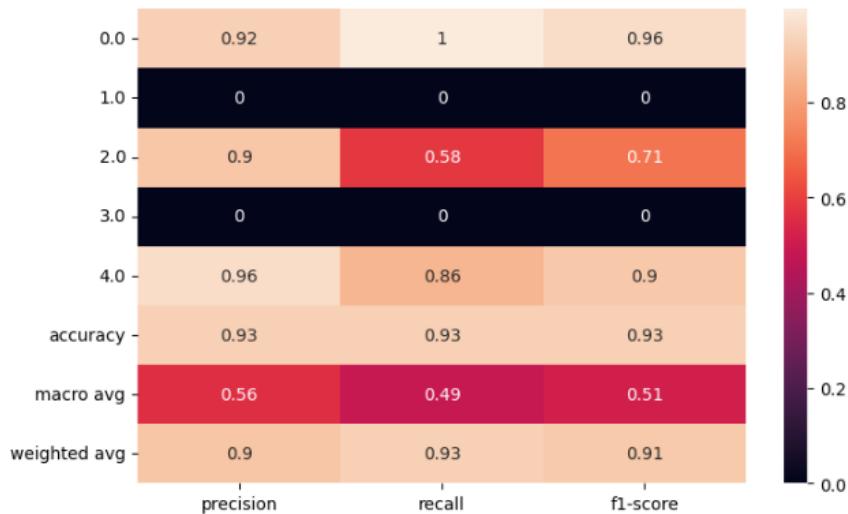
# lists to store the loss values for plotting
train_losses = []
test_losses = []

# train the model
for epoch in range(25):

```



Nu se converge mai mult. Nu se obtin rezultate mai bune:

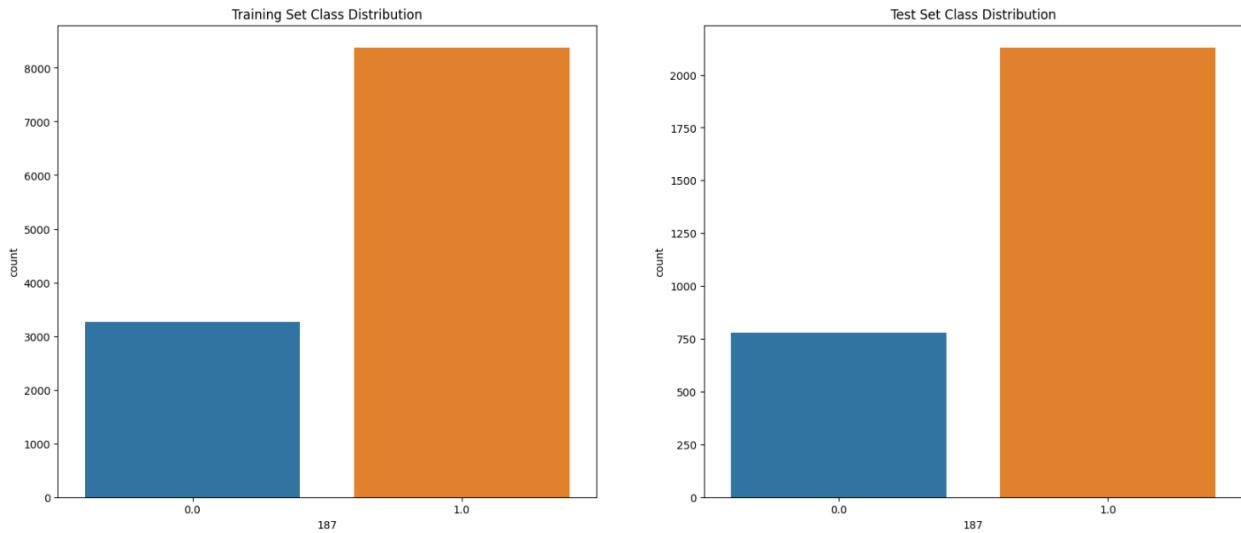


In concluzie: cel mai bun model de retele neurale a fost InceptionTime. Acesta a realizat ceva mai buna extragere de atribute si a reusit sa generalizeze suficient cat sa observe existenta clasei 1. Totusi, in ce priveste acest task, modele de ML au reusit scoruri mai bune, fiind ajutate de

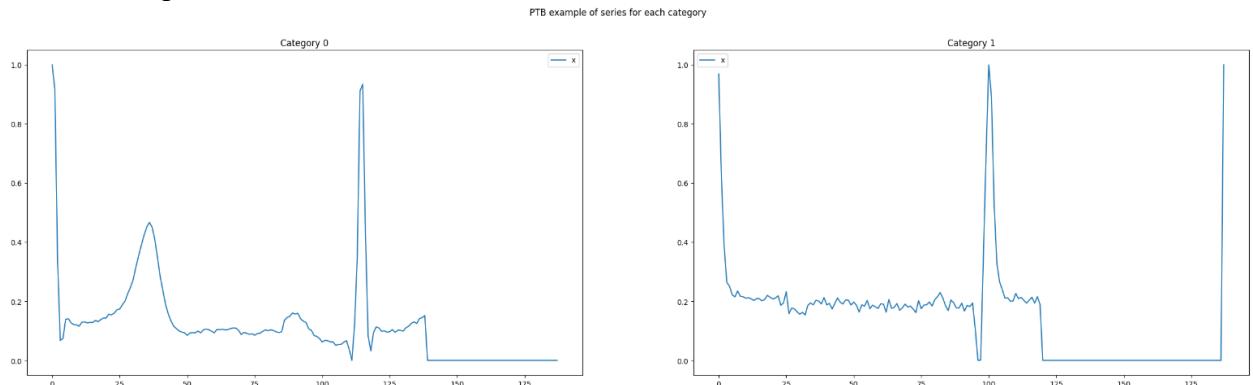
extragerea manuala de atribute. Modulul LSTM nu a reusit sa inteleaga foarte bine diferentele dintre clase, MLP-ul reusind performanta de a identifica clasa 1. Cred ca as fi putut insista mai mult pe arhitectura de InceptionTime, care obtinea un loss bun si nu dadea semne de overfitting.

PTB

1. Explorarea datelor

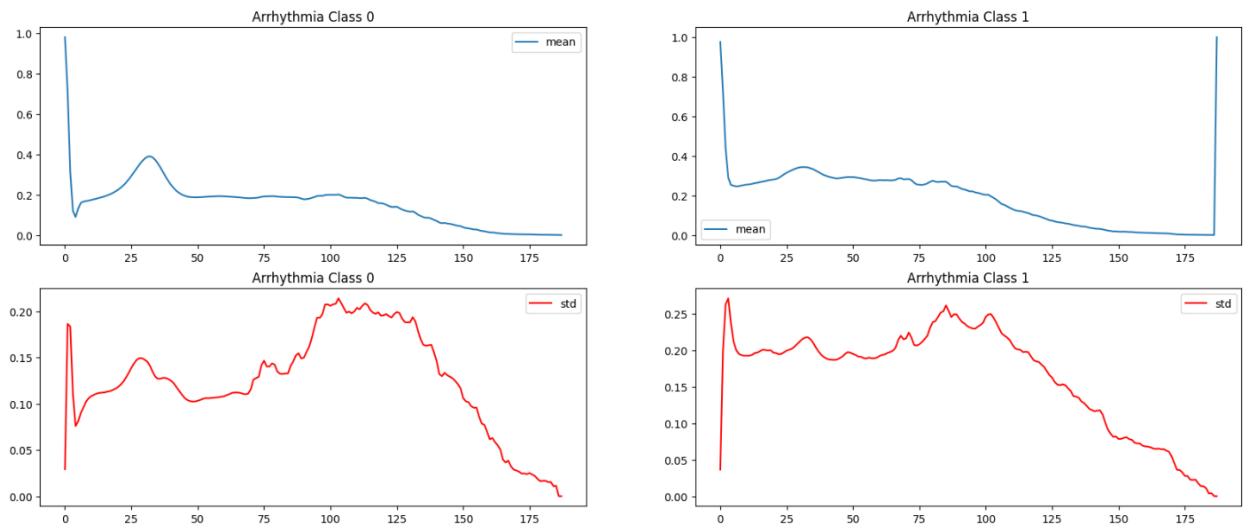


Dupa cum se poate vedea, exista un dezechilibru intre clase. Totusi, raportul este de (2-2.5):1. Nu este foarte accentuat. Un lucru bun este ca raportul din dataset-ul de training este acelasi cu raportul din dataset-ul de test.



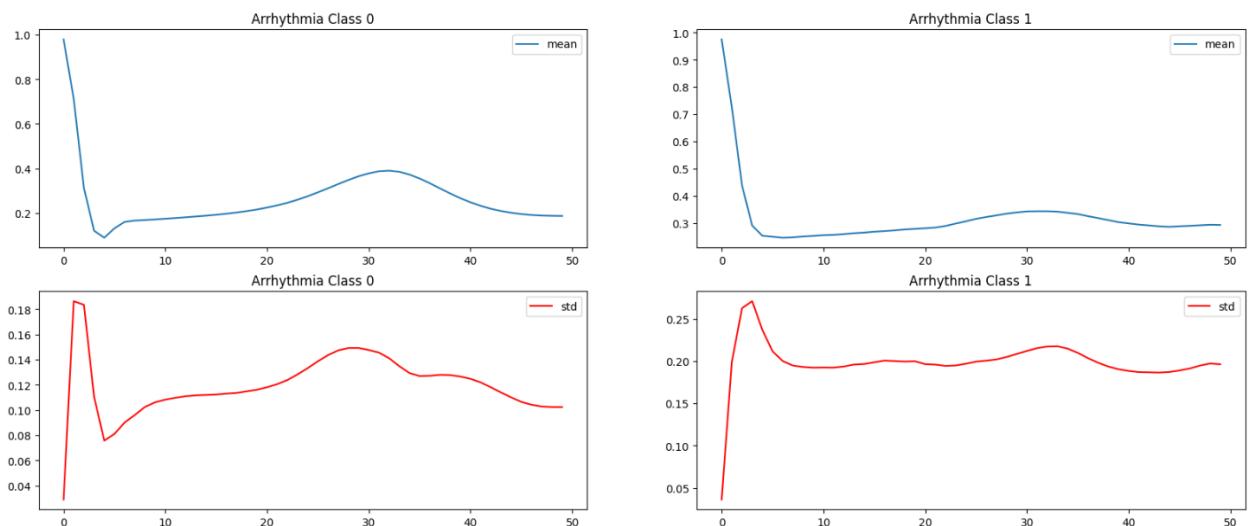
Acest este un exemplu de aritmie. Pentru aceste exemple, par sa fie evidențiate cîteva diferențe. Inima bolnava, clasa 1, prezintă o zonă constantă și cu mult zgomot, zona dintre timpii 25 și 100. Clasa 0 prezintă treceți mai liniști și există 1 moment de high care lipsește la clasa 1, cel dintre timpii 25-50.

Mean and Standard Deviation per Unit of Time for Each Arrhythmia PTB Class



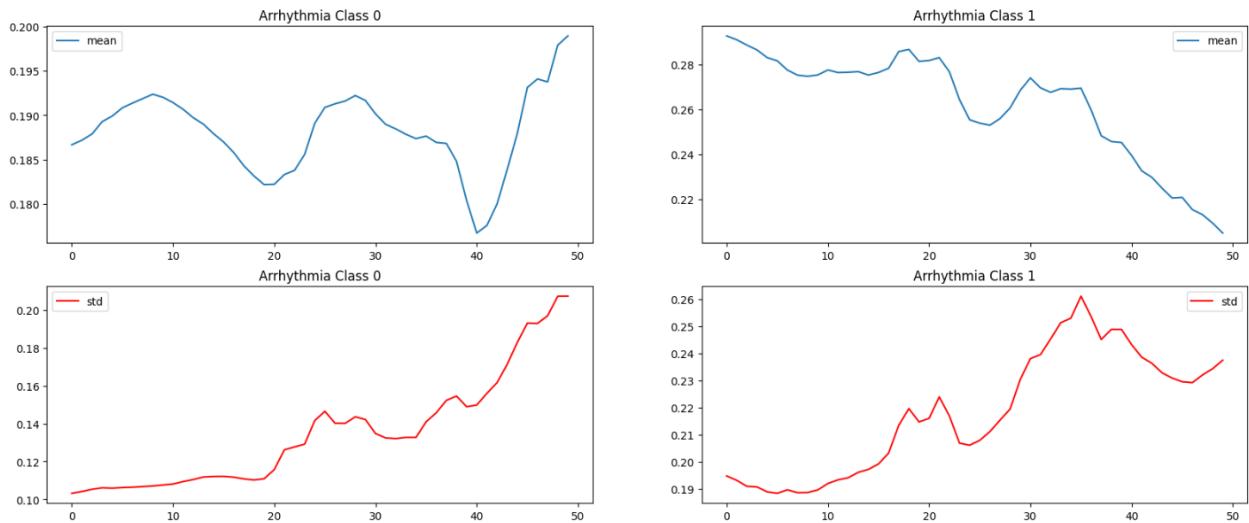
Aşa cum spune si titlul, vazând media valorilor și deviația standard se pot observa diferențe. Există acel moment “high” la inima sănătoasă care la clasa 1 nu există. Totul pare mai plat pentru clasa 1.

Mean and Standard Deviation per Unit of Time for Each Arrhythmia PTB Class



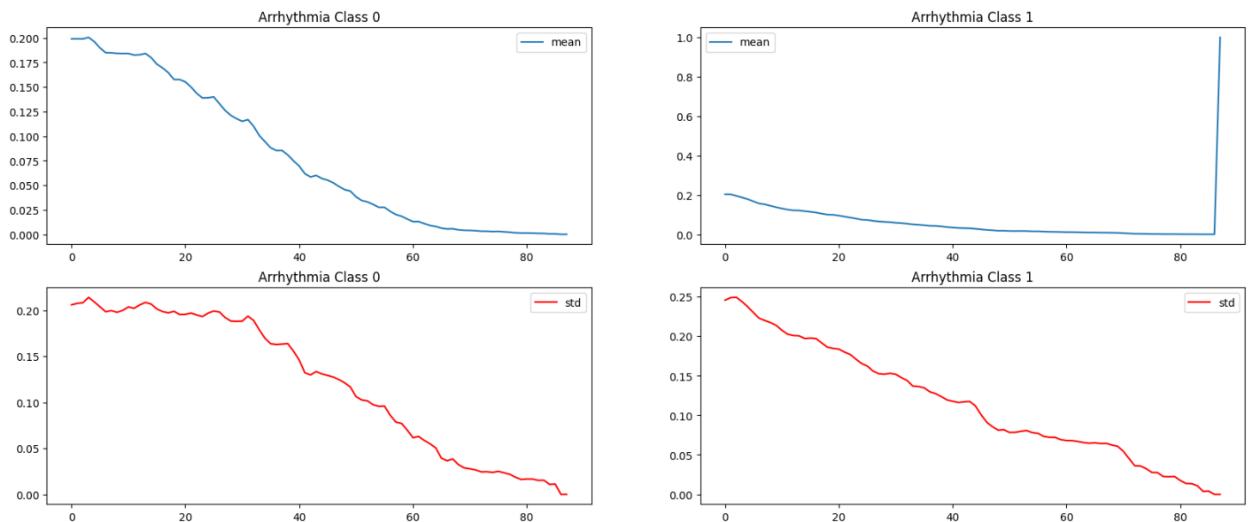
Acesta reprezintă un zoom pentru primii 50 de timpi. Se poate evidenția mai bine diferența dintre clasa 0 și clasa 1.

Urmatorii 50 de timpi sunt reprezentati in urmatoarea imagine:



A se evidenta nu doar decalarea “ deal-vale”, dar si valorile de pe axa.
Din nou, efectul de “plat” in loc de “panta” este evidențiat în următorii timpi:

Mean and Standard Deviation per Unit of Time for Each Arrhythmia PTB Class



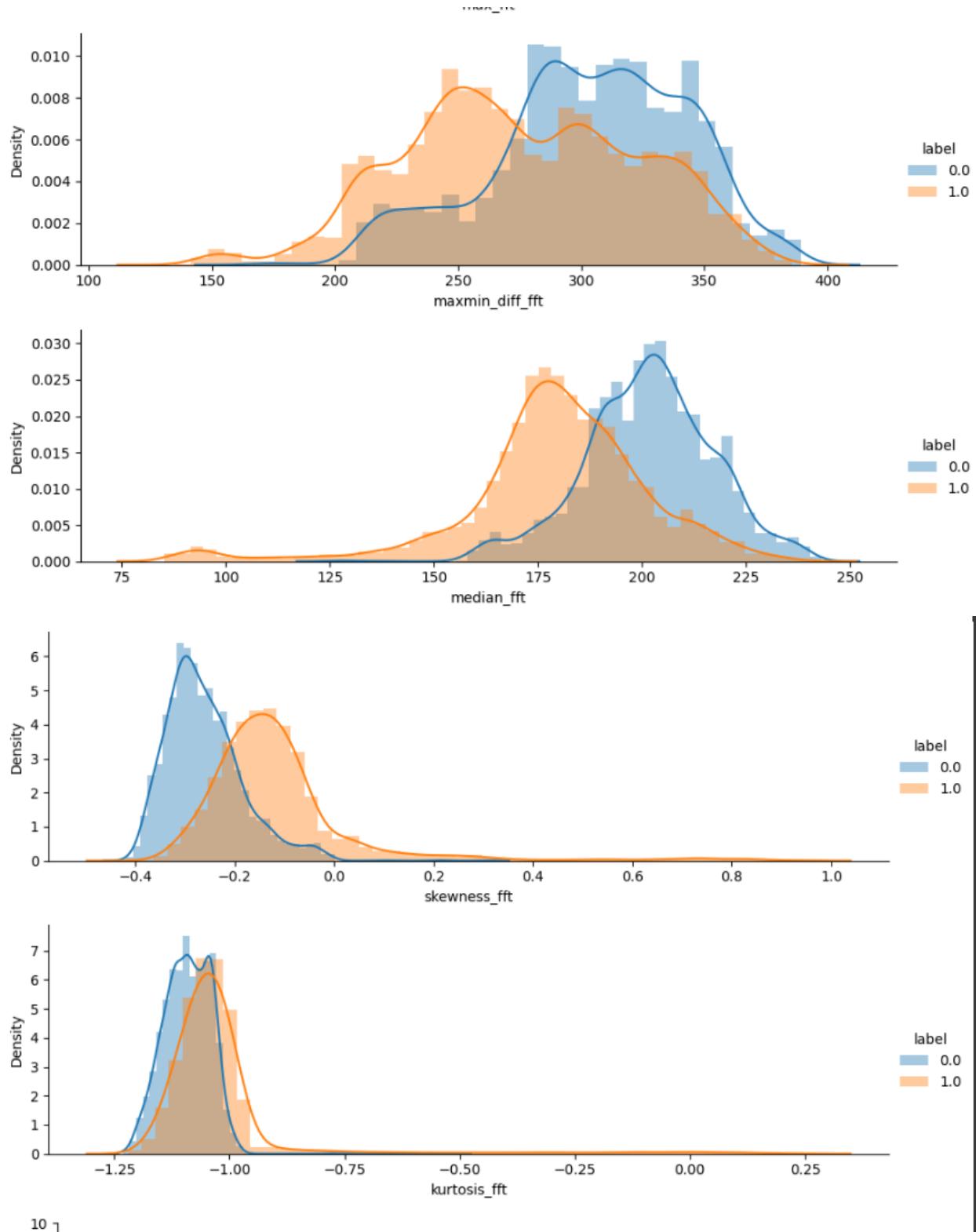
2. Extragerea manuala de atribute

Observ ca există date redundante în ce privește primii 10 timpi și ultimii 10 timpi din fiecare serie. De aceea elimin aceste coloane.

In privinta atributelor, am extras aceleasi atribute ca si in cazul dataset-ului RacketSports. In ce privește standardizarea, nu am realizat-o ca si in cazul celuilalt dataset intrucat aici bataile inimii se afla intre [0,1] si este deja ok.

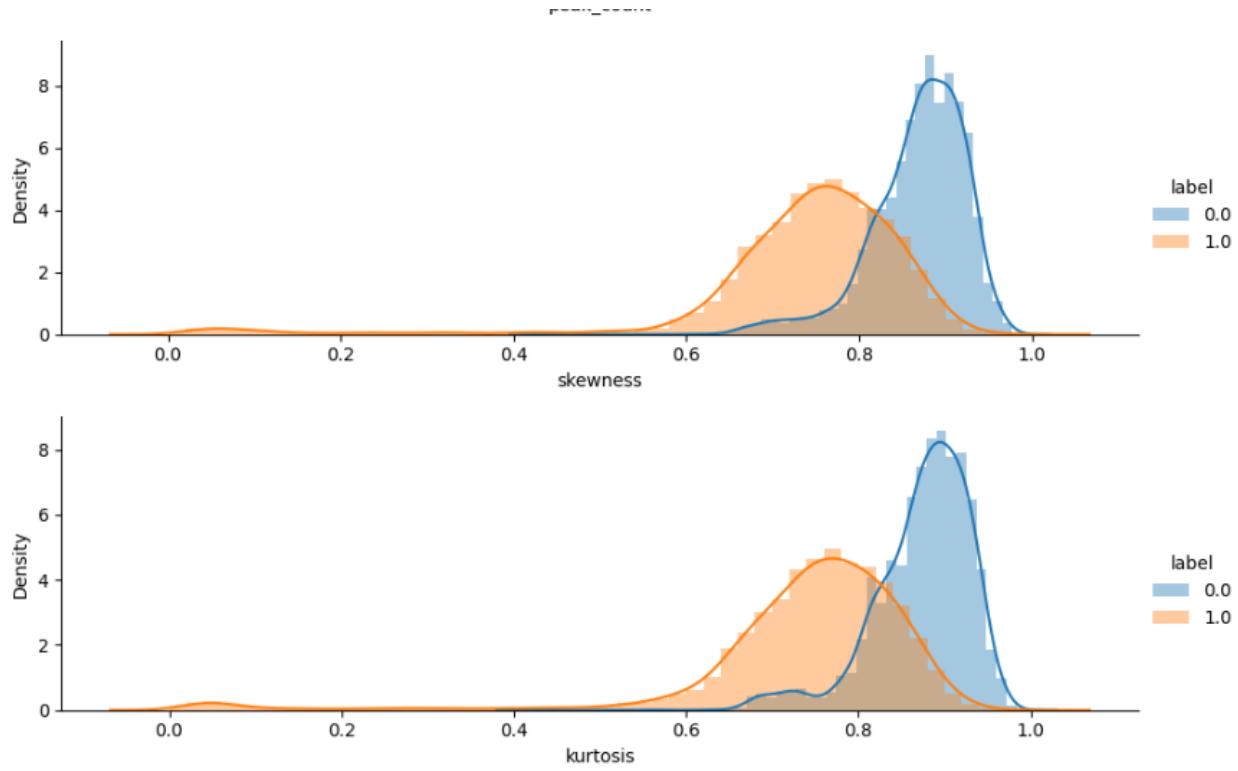
Deci, standardizarea s-a realizat pentru feature-urile extrase, folosind functia *MinMaxScaler()*. Alegerea functiei s-a realizat tot pe cale experimentală.

Cateva vizualizari ale datelor inainte de Standardizare:

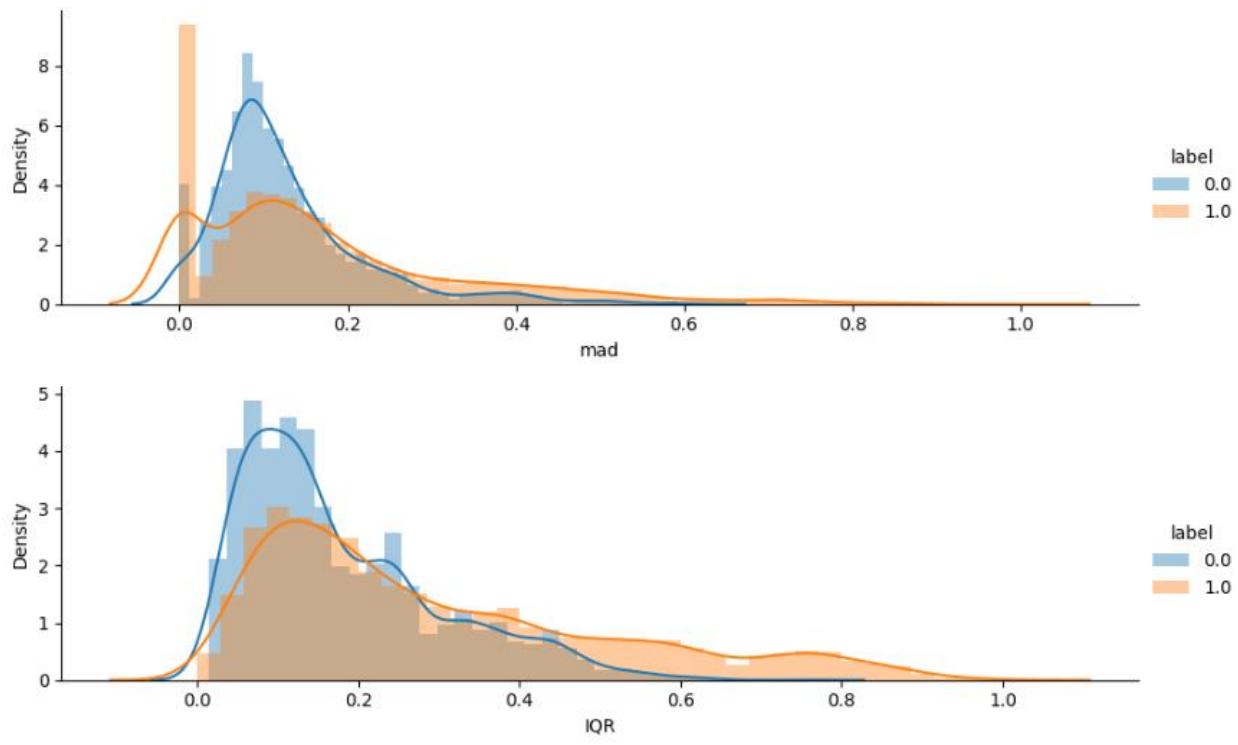


Fiind valori asa de diferite, se remarcă nevoia de standardizare.

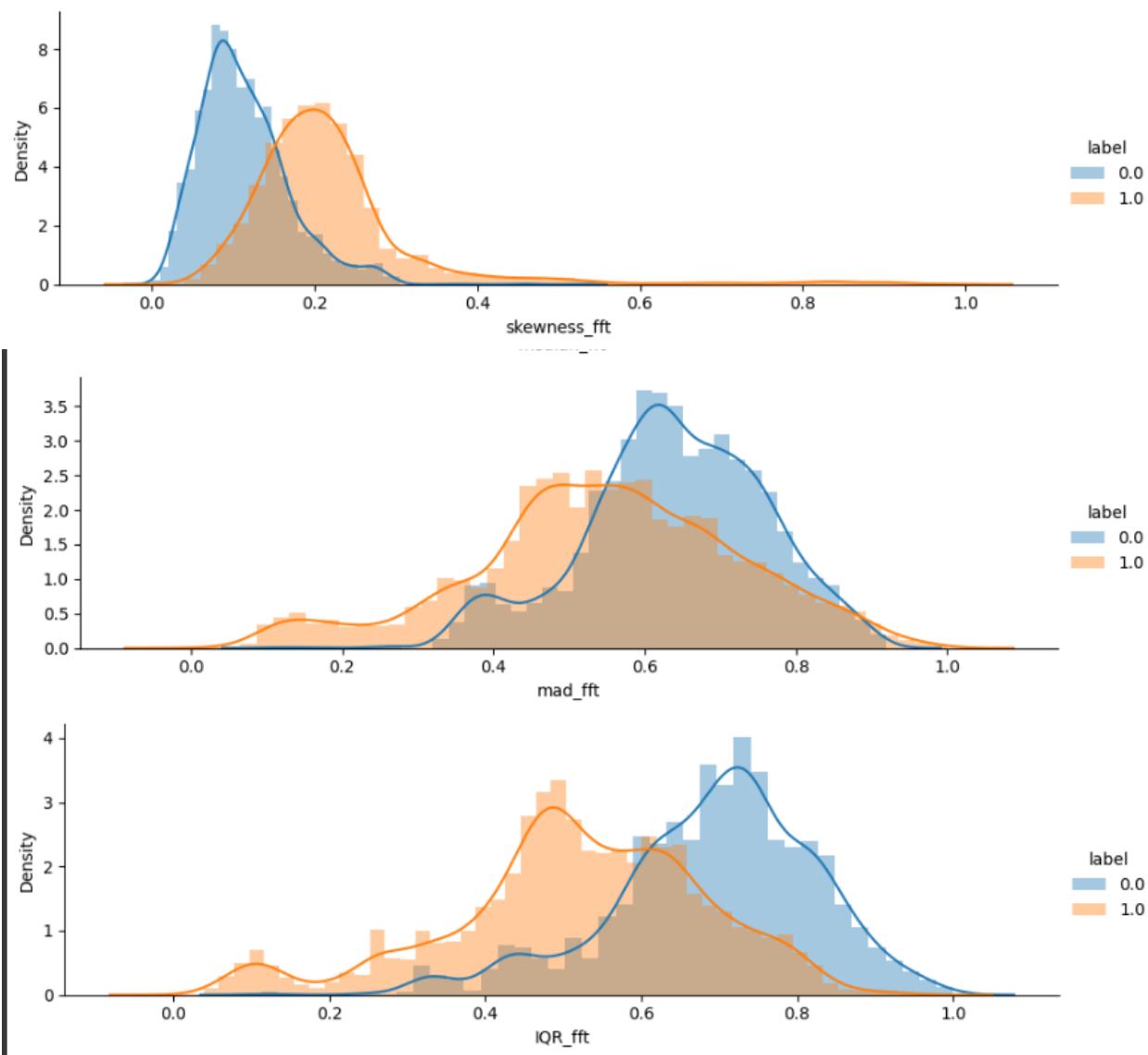
Cateva vizualizari ale datelor dupa Standardizare:

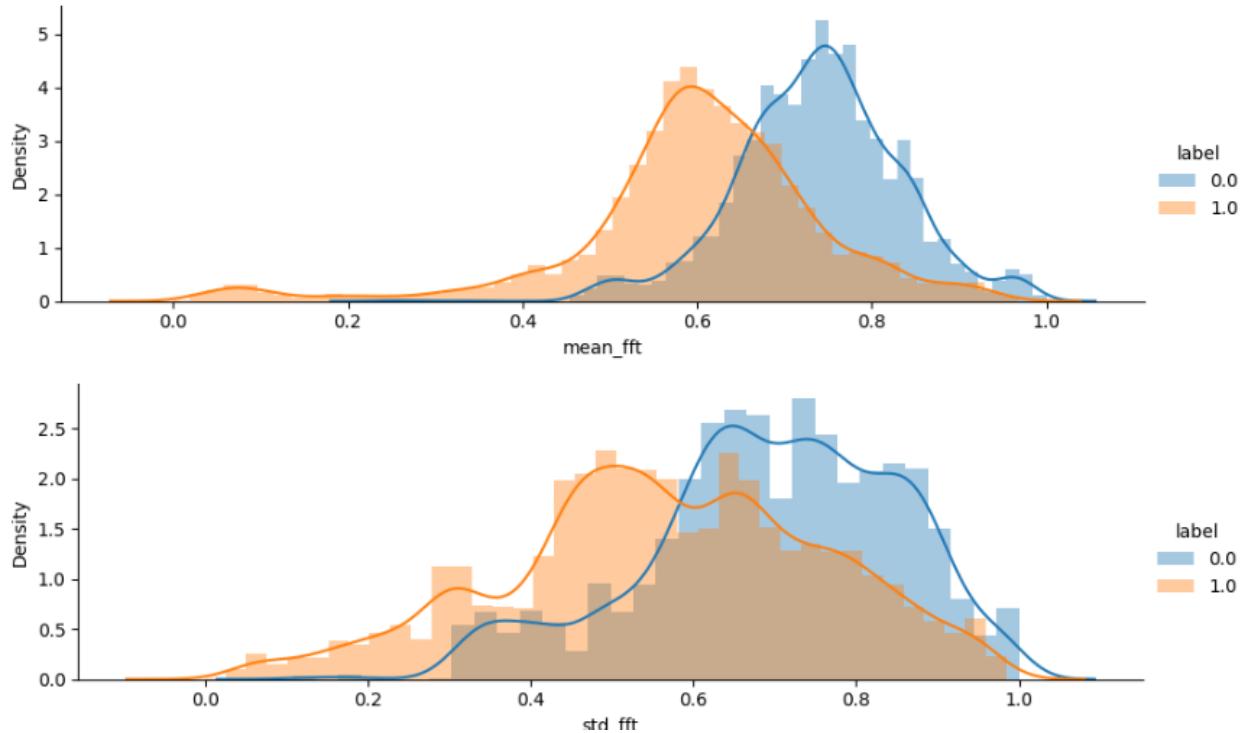


Se remarcă diferențe mari între cele două clase.



Alte exemple care arată diferențe între cele două clase.





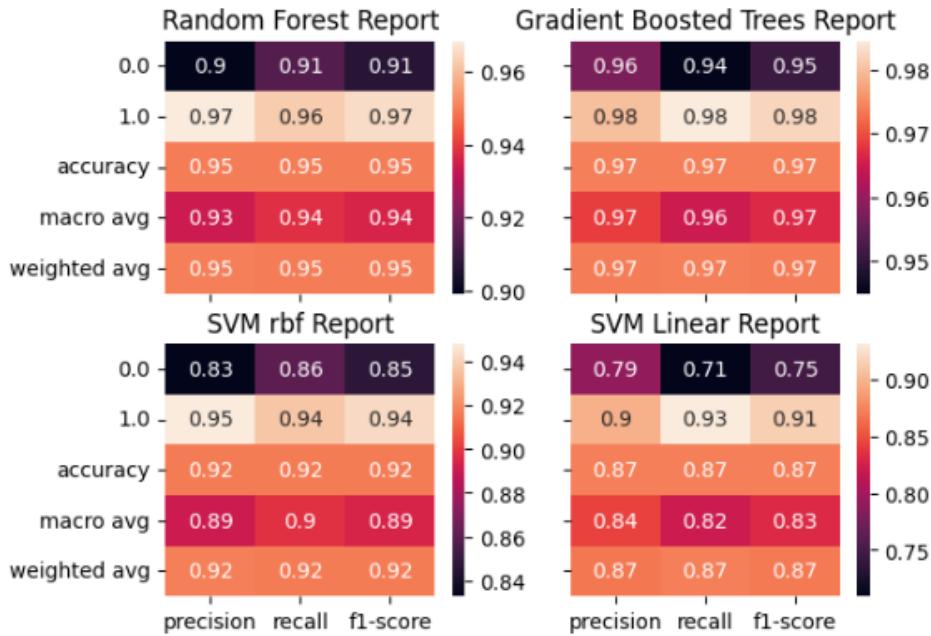
Pare ca valorile mai mari apartin clasei 0 😊. Important este faptul ca statistic, privind feature-urile extrase se pot vedea multe diferente intre cele doua clase, ceea ce face task-ul usor de invatat de catre modele.

Deoarece exista o varianta mai mica a datelor, folosind *MinMaxScaler()* peste feature-uri, filtrarea atributelor prin *VarianceThreshold()* s-a realizat cu un threshold = 0.015 obtinand din 28 de atribute, doar 19 in final. Orientarea pentru numarul 0.015 a fost realizata in functie de numarul de atribute ramase la final. Am considerat 19 un numar bun. De asemenea, am observat ca celelalte 9 atribute eliminate se pierdeau destul de repede. (si la un threshold mai mare)

3. Algoritmi de Invatare Automata

A se consulta fisierul Project_ML.xlsx pentru rezultate !

Initial, am creat 4 modele : RandomForest, Gradient Boosted Trees, SVC rbf si SVC Linear.



Se remarcă rezultate mult mai bune pentru Gradient Boosted Trees. Se observă de asemenea că se obțin scoruri mai bune în ce privește clasa 1. O explicație ar putea fi dezechilibrul din dataset-ură. Există mai multe exemple pentru clasa 1, deci modelul este mai pregătit să prezice clasa 1.

In continuare, am încercat să imbunătățesc algoritmii încercând cete 6 exemple din fiecare model pentru a contura, înțelegi mai bine combinațiile pe care le-ai putea oferi în cadrul apelului de GridSearchCV.

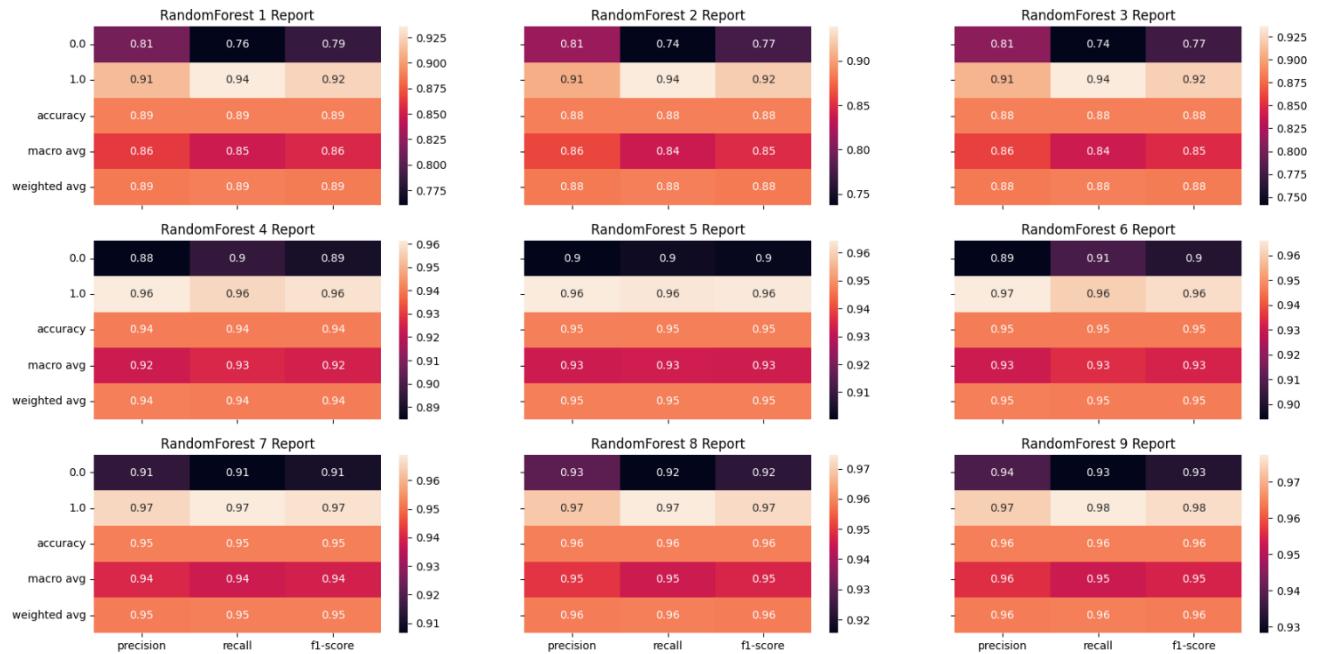
Am creat 6 modele de RandomForest:

```
# Define several instances of Random Forest classifier with different hyperparameters

random_forest_1 = RandomForestClassifier(n_estimators = 60, max_depth = 5, max_samples = 0.3)
random_forest_2 = RandomForestClassifier(n_estimators = 60, max_depth = 5, max_samples = 0.6)
random_forest_3 = RandomForestClassifier(n_estimators = 60, max_depth = 5, max_samples = 0.9)

random_forest_4 = RandomForestClassifier(n_estimators = 120, max_depth = 10, max_samples = 0.3)
random_forest_5 = RandomForestClassifier(n_estimators = 120, max_depth = 10, max_samples = 0.6)
random_forest_6 = RandomForestClassifier(n_estimators = 120, max_depth = 10, max_samples = 0.9)

random_forest_7 = RandomForestClassifier(n_estimators = 80, max_depth = 15, max_samples = 0.3)
random_forest_8 = RandomForestClassifier(n_estimators = 80, max_depth = 15, max_samples = 0.6)
random_forest_9 = RandomForestClassifier(n_estimators = 80, max_depth = 15, max_samples = 0.9)
```



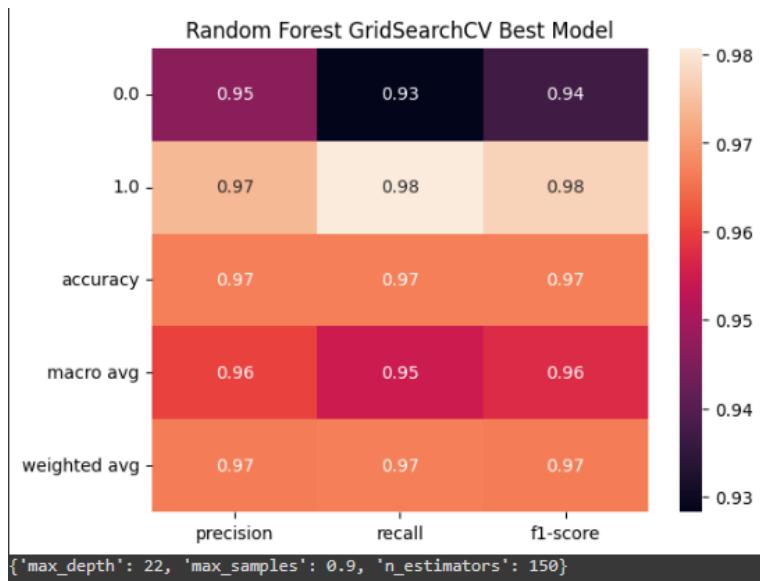
Un lucru foarte interesant, nu exista diferente pe linii, diferente mari. Asta inseamna ca parametrul max_samples nu influenteaza foarte mult, desi este setat cu valorile : 0.3, 0.6, 0.9. Cred ca se intampla acest lucru, deoarece modelul invata un task usor si atunci nu are nevoie de asa multe exemple. Cu alte cuvinte, modelul s-ar afla mai degraba in risc de overfitting decat in risc de underfitting.

De asemenea, se remarcă faptul că o adâncime mai mare este preferată față de un număr mai mare de copaci, modelele 7-8-9 având scoruri mai bune decât modelele 4-5-6.

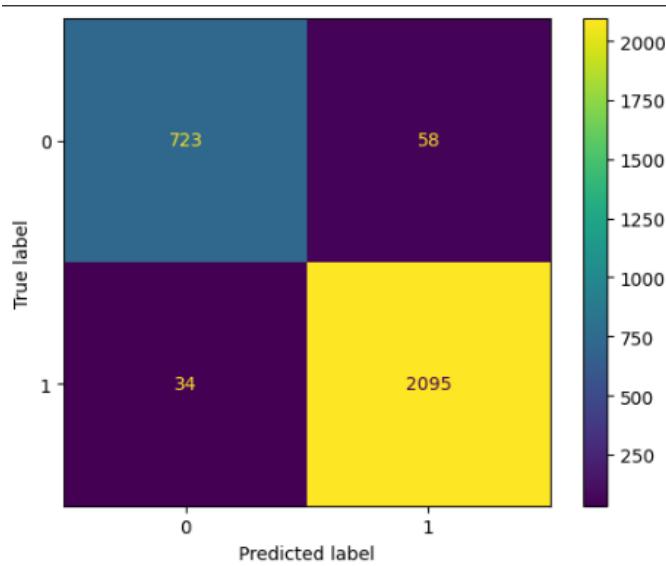
Deci, am rulat GridSearchCV cu parametrii:

```
# Define a dictionary of hyperparameters for the Random Forest Classifier model
parameters = {
    'max_depth': [8, 13, 18, 22, 25, 30],
    'n_estimators': [80, 100, 150, 200],
    'max_samples': [0.5, 0.9]
}
```

A fost aleasă urmatoarea configurație : {max_depth = 22, max_samples = 0.9, n_estimators = 150}, cu rezultatele:



Se obtin rezultate foarte bune. Se remarcă totuși nevoie de model complex, modelul având 150 de copaci cu o adâncime egală cu 22.



Matricea de confuzie prezintă faptul că într-adevar, mai multe exemple din clasa 0 sunt prezise greșit în clasa 1, adică clasa cu mai multe exemple în dataset decât invers.

PTB - ECG																		
Numar de feature-uri considerate			28															
Numar de feature-uri la antrenare			19															
Random Forest Classifier																		
n_estimators	max_depth	max_samples	Precision	Recall	F1-score	Acurateza												
60	5	0.3	0.81	0.76	0.79	0.91	0.94	0.92	0.86	0.85	0.86	0.005	0.016	0.008	0.89			
		0.6	0.81	0.74	0.77	0.91	0.94	0.92	0.86	0.84	0.85	0.005	0.020	0.011	0.88			
		0.9	0.81	0.74	0.77	0.91	0.94	0.92	0.86	0.84	0.85	0.005	0.020	0.011	0.88			
80	15	0.3	0.91	0.91	0.91	0.97	0.97	0.97	0.94	0.94	0.94	0.002	0.002	0.002	0.95			
		0.6	0.93	0.92	0.92	0.97	0.97	0.97	0.95	0.95	0.95	0.001	0.001	0.001	0.96			
		0.9	0.94	0.93	0.93	0.97	0.98	0.98	0.96	0.96	0.96	0.000	0.001	0.001	0.96			
120	10	0.3	0.88	0.9	0.89	0.96	0.96	0.96	0.92	0.93	0.93	0.003	0.002	0.002	0.94			
		0.6	0.9	0.9	0.9	0.96	0.96	0.96	0.93	0.93	0.93	0.002	0.002	0.002	0.95			
		0.9	0.89	0.91	0.9	0.97	0.96	0.96	0.93	0.94	0.93	0.003	0.001	0.002	0.95			
100	10	-	0.9	0.91	0.91	0.97	0.96	0.97	0.94	0.94	0.94	0.002	0.001	0.002	0.95			
150	22	0.9	0.95	0.93	0.94	0.97	0.98	0.98	0.96	0.96	0.96	0.000	0.001	0.001	0.97			

Se remarcă o variație foarte bună a datelor. O variație mică este de dorit. Reprezintă un echilibru al predicțiilor.

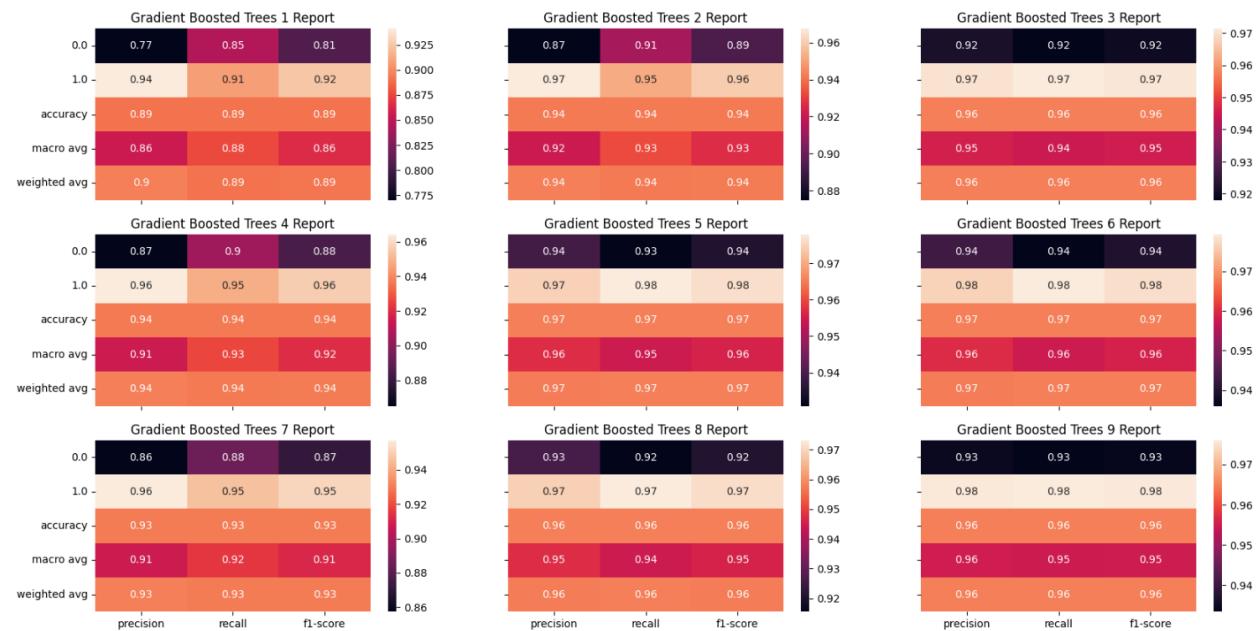
Am creat 6 modele de XGBoost :

```
# Define XGBClassifiers with different hyperparameters

gb_trees_1 = XGBClassifier(n_estimators = 60, max_depth = 5, learning_rate = 0.01, nthread=-1, tree_method='gpu_hist')
gb_trees_2 = XGBClassifier(n_estimators = 60, max_depth = 5, learning_rate = 0.1, nthread=-1, tree_method='gpu_hist')
gb_trees_3 = XGBClassifier(n_estimators = 60, max_depth = 5, learning_rate = 0.2, nthread=-1, tree_method='gpu_hist')

gb_trees_4 = XGBClassifier(n_estimators = 120, max_depth = 10, learning_rate = 0.01, nthread=-1, tree_method='gpu_hist')
gb_trees_5 = XGBClassifier(n_estimators = 120, max_depth = 10, learning_rate = 0.1, nthread=-1, tree_method='gpu_hist')
gb_trees_6 = XGBClassifier(n_estimators = 120, max_depth = 10, learning_rate = 0.2, nthread=-1, tree_method='gpu_hist')

gb_trees_7 = XGBClassifier(n_estimators = 80, max_depth = 15, learning_rate = 0.01, nthread=-1, tree_method='gpu_hist')
gb_trees_8 = XGBClassifier(n_estimators = 80, max_depth = 15, learning_rate = 0.1, nthread=-1, tree_method='gpu_hist')
gb_trees_9 = XGBClassifier(n_estimators = 80, max_depth = 15, learning_rate = 0.2, nthread=-1, tree_method='gpu_hist')
```

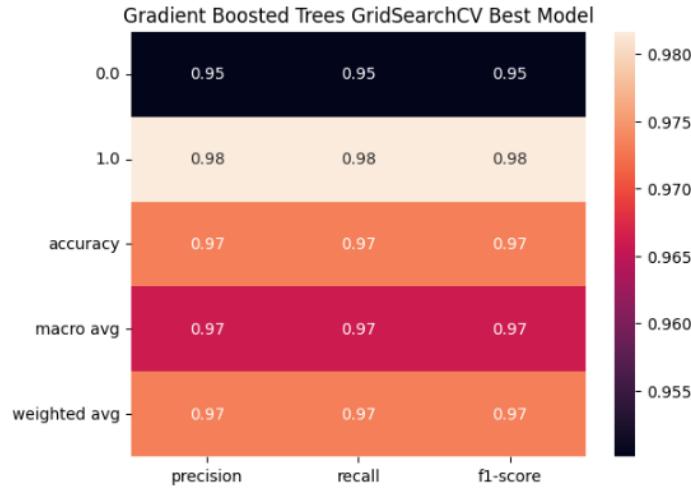


Se remarcă rezultatele bune obținute cu o valoare mai mare a parametrului `learning_rate`. De asemenea, modelul are nevoie să fie complex, de aceea, atunci când am ales parametrii ce vor fi rulati prin metoda GridSearchCV() am ales valori mai mari.

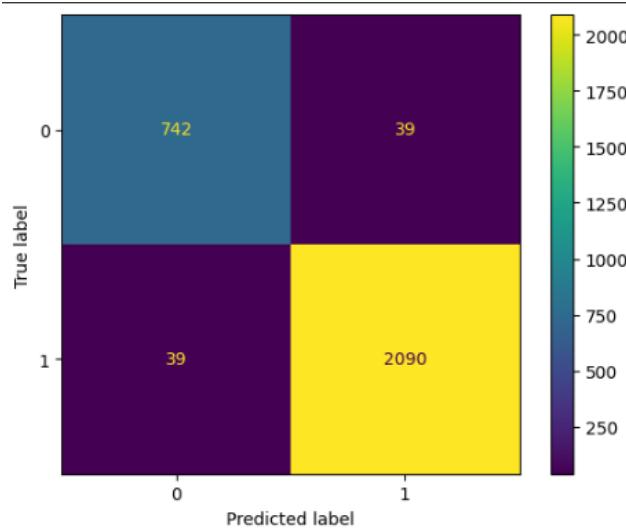
```
# XGBoost Classifier Initialization
estimator = XGBClassifier(nthread=-1, tree_method='gpu_hist', seed=42)

# Parameter grid for Grid Search Cross Validation
parameters = {
    'max_depth': [5, 13, 20, 25, 7, 10, 22, 30, 40],
    'n_estimators': [70, 100, 170, 220, 140, 90, 250, 300, 50],
    'learning_rate': [0.1, 0.2]
}
```

Parametrii alesi au fost: {`learning_rate` = 0.2, `max_depth` = 5, `n_estimators` = 300 } Rezultatele fiind:



Rezultatele sunt foarte bune. Variatia este 0. La astfel de numere, nu poti astepta ca modelul sa aiba acuratete 100%. Deci 98% este un scor foarte bun. Este nevoie de generalizare, nu de overfitting.



Se observa un lucru interesant, se obtine acelasi numar de clasificari gresite, pentru ambele clase. Cu aceasta rulare, cifrele chiar sunt echilibrate.

XGB Classifier															
n_estimators	max_depth	learning_rate	0			1			Media			Varianta			Acuratetea
			Precision	Recall	F1-score										
60	5	0.01	0.77	0.85	0.81	0.94	0.91	0.92	0.86	0.88	0.87	0.014	0.002	0.006	0.89
		0.1	0.87	0.91	0.89	0.97	0.95	0.96	0.92	0.93	0.93	0.005	0.001	0.002	0.94
		0.2	0.92	0.92	0.92	0.97	0.97	0.97	0.95	0.95	0.95	0.001	0.001	0.001	0.96
80	15	0.01	0.86	0.88	0.87	0.96	0.95	0.95	0.91	0.92	0.91	0.005	0.002	0.003	0.93
		0.1	0.93	0.92	0.92	0.97	0.97	0.97	0.95	0.95	0.95	0.001	0.001	0.001	0.96
		0.2	0.93	0.93	0.93	0.98	0.98	0.98	0.96	0.96	0.96	0.001	0.001	0.001	0.96
120	10	0.01	0.87	0.9	0.88	0.96	0.95	0.96	0.92	0.93	0.92	0.004	0.001	0.003	0.94
		0.1	0.94	0.93	0.94	0.97	0.98	0.98	0.96	0.96	0.96	0.000	0.001	0.001	0.97
		0.2	0.94	0.94	0.94	0.98	0.98	0.98	0.96	0.96	0.96	0.001	0.001	0.001	0.97
100	6	-	0.96	0.94	0.95	0.98	0.98	0.98	0.97	0.96	0.97	0.000	0.001	0.000	0.97
300	5	0.2	0.95	0.95	0.95	0.98	0.98	0.98	0.97	0.97	0.97	0.000	0.000	0.000	0.97

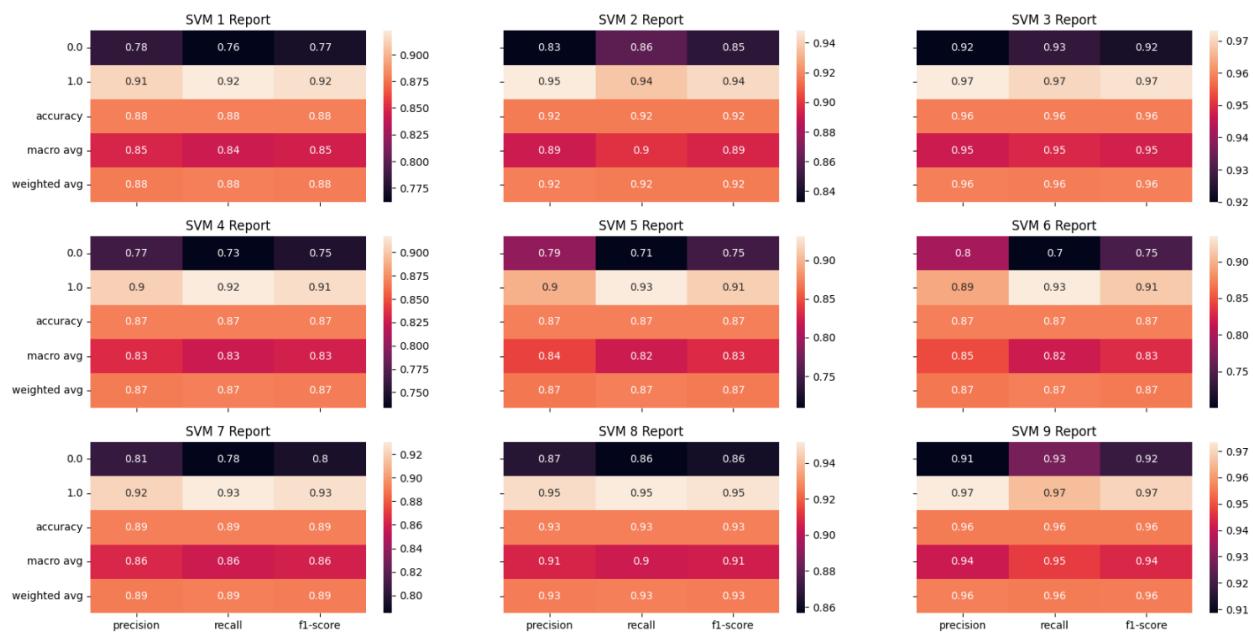
Am creat 6 modele de SVC :

```
# Define SVM Classifiers with different hyperparameters

svm_1 = SVC(kernel = 'rbf', C = 0.1)
svm_2 = SVC(kernel = 'rbf', C = 1)
svm_3 = SVC(kernel = 'rbf', C = 10)

svm_4 = SVC(kernel = 'linear', C = 0.1)
svm_5 = SVC(kernel = 'linear', C = 1)
svm_6 = SVC(kernel = 'linear', C = 10)

svm_7 = SVC(kernel = 'poly', C = 0.1)
svm_8 = SVC(kernel = 'poly', C = 1)
svm_9 = SVC(kernel = 'poly', C = 10)
```

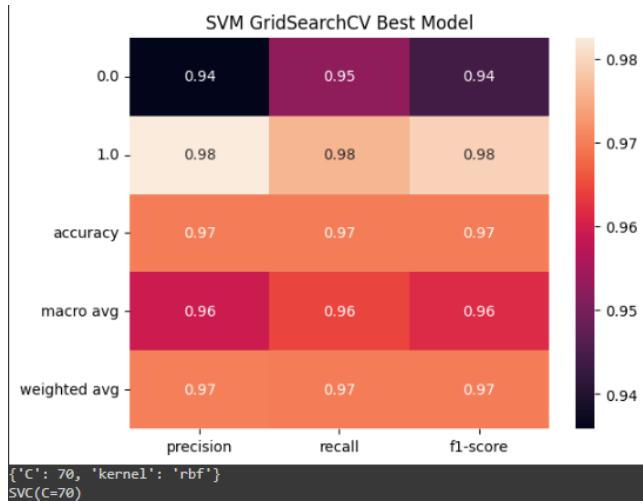


Se remarcă rezultate mai bune pentru un $C > 1$. De asemenea, chiar și pentru acest dataset, un kernel ‘linear’ nu este destul de bun. Este nevoie de o supradimensionare și de o interpretare în alt spațiu al datelor. Un kernel ‘rbf’ sau ‘poly’ pare mai potrivit.

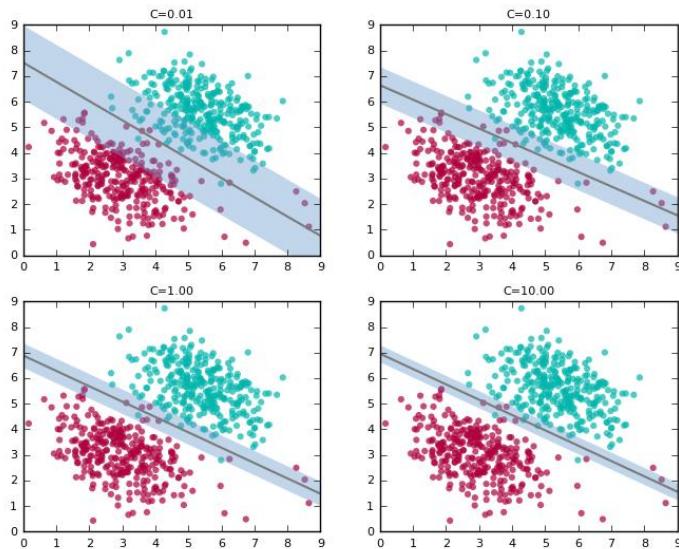
Am rulat GridSearchCV() cu parametrii:

```
# Define a dictionary of parameter ranges to search over
parameters = {'kernel':['poly', 'rbf'], 'C':[1, 2, 3, 4, 8, 12, 20, 30, 40, 50, 70]}
```

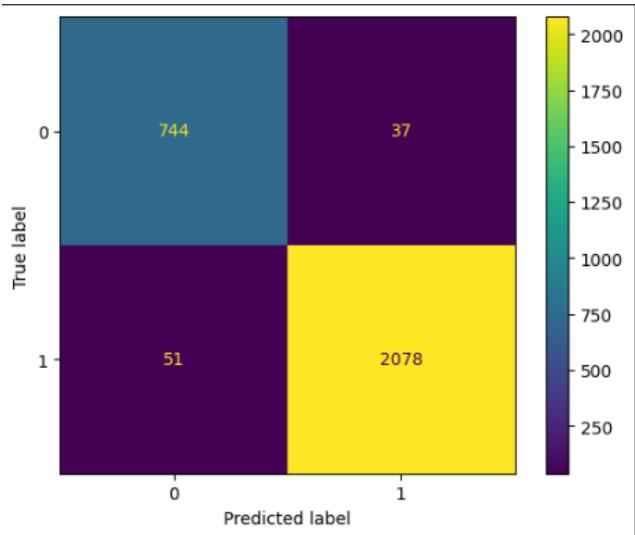
S-au obținut rezultatele:



Parametrii alesi au fost : kernel = ‘rbf’ si C = 70. Folosesc urmatoare poza pentru o intuitie mai buna:



Un C mai mare accepta mai putin puncte in zona dintre margini.



De asemenea, se remarcă ca mai multe exemple din clasa 1 sunt prezise în clasa 0. O explicație în cazul SVM-ului ar fi faptul că având un C atât de mare, vectorul despartitor este ales tinând cont de un număr mai mic de puncte de pe margine, se obține o pantă care în total generalizează mai bine, dar sacrifică câteva puncte din clasa 1.

4. Retele neurale

În ce privește rezolvarea task-ului prin retele neurale, nu mai sunt extrase manual feature-uri și tinând cont de acest lucru, am reanalizat graficele cu seriile de timp și am considerat că ar fi mai bine să elimin doar primele 5 și ultimele 5 coloane, pe considerente explicate mai sus.

După eliminarea coloanelor, am standardizat datele. Pe cale experimentală am observat că se obțin rezultate mai bune astfel.

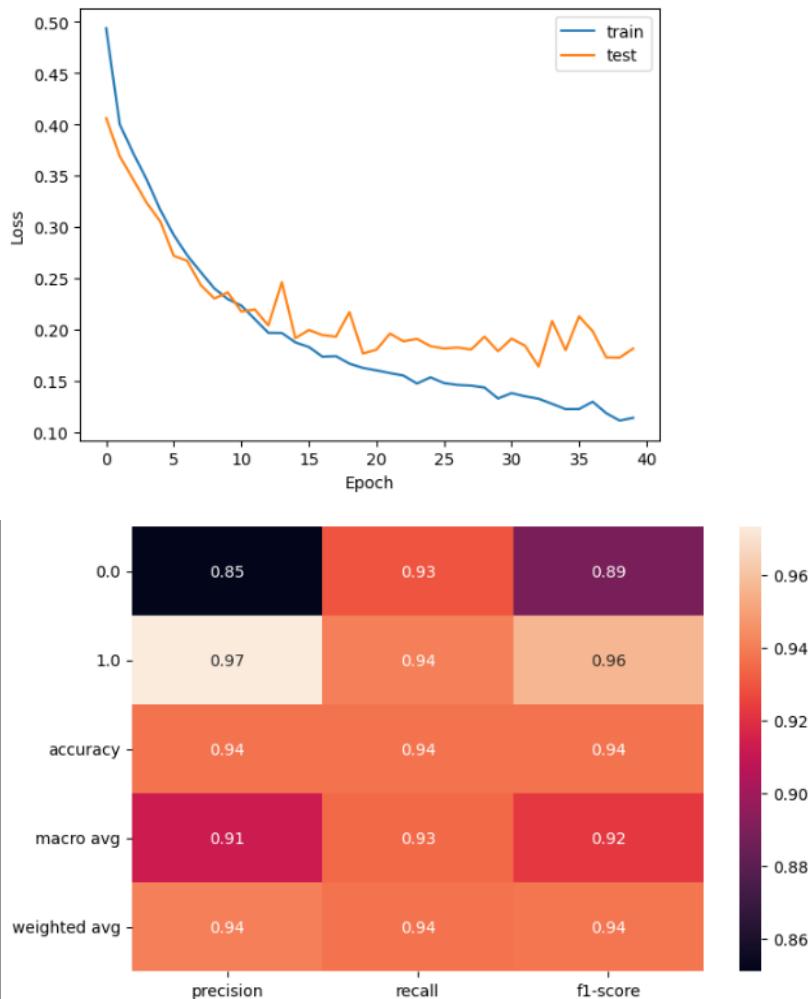
Prima arhitectură pe care am rulat este cea de MLP.

Arhitectură de tip **MLP**

```

1  class MLP(nn.Module):
2      def __init__(self):
3          super(MLP, self).__init__()
4          self.fc1 = nn.Linear(178, 64)
5          self.fc2 = nn.Linear(64, 32)
6          self.fc3 = nn.Linear(32, 16)
7          self.fc4 = nn.Linear(16, 2)
8

```



Pentru o arhitectura atat de simpla se obtin rezultate foarte bune. Aceasta este o dovada a faptului ca exista foarte multe diferente in exemple intre cele doua clase, task-ul fiind usor. In ce priveste curba de loss, se observa o convergenta buna pentru curba de train, in timp ce curba de test ramane aproape constanta de la epoca 15. Cred ca un motiv ar fi un learning_rate prea mare si o intrare in regimul de overfitting. Arhitectura modelului este prea simpla.

Am incercat si urmatoarea arhitectura de MLP modificata:

```
Arhitectură de tip MLP 2.0

[ ] 1  class MLP2(nn.Module):
2      def __init__(self):
3          super(MLP2, self).__init__()
4          self.fc1 = nn.Linear(178, 128)
5          self.fc2 = nn.Linear(128, 64)
6          self.fc3 = nn.Linear(64, 48)
7          self.fc4 = nn.Linear(48, 32)
8          self.fc5 = nn.Linear(32, 16)
9          self.fc6 = nn.Linear(16, 2)
10
```

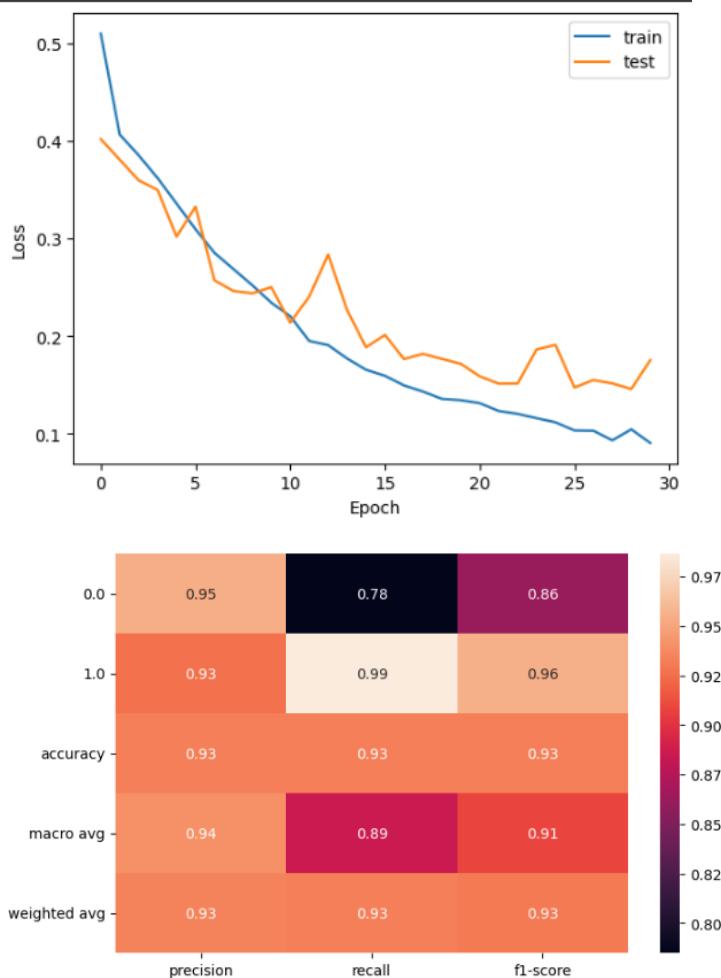
Am folosit un learning_rate mai mic:

```

model = MLP2()

# define the loss function and optimizer
criterion = nn.BCEWithLogitsLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.0005, weight_decay=0.001)

```



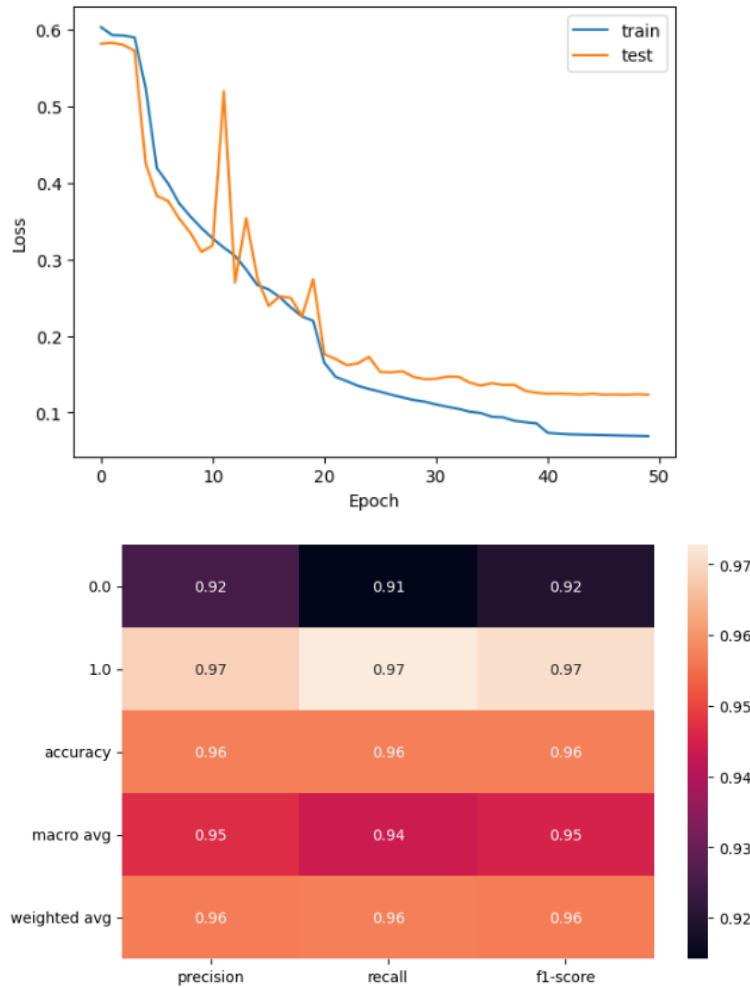
Nu se obtin rezultate foarte diferite. Curba de loss pentru test pare sa fie mai apropiata de curba de loss pentru train. Totusi, numarul de epoci este mai mic. Faptul ca arhitectura MLP-ului are cateva layer-e in plus nu influenteaza foarte mult.

Am incercat sa rulez aceast model si cu un scheduler care sa controleze learning_rate-ul:

```

1 model = MLP2()
2
3 # define the loss function and optimizer
4 criterion = nn.BCEWithLogitsLoss()
5
6 # define the optimizer with SGD and momentum
7 optimizer = torch.optim.SGD(model.parameters(), lr=0.01, momentum=0.9, weight_decay=0.001)
8
9 # define the learning rate scheduler
10 scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=20)
11 # lists to store the loss values for plotting

```



Se obtin rezultate mult mai bune. Acuratetea este imbunatatita si scorurile pentru clasa 0. Se observa intre epocile 10 si 20 cum learning_rate-ul pare prea mare, dar apoi este ajustat si mai scade un pic pana la epoca 50.

PTB - ECG																	
MLP																	
batch_size	Arhitectura	epochs / lr	dropout	step_size / gamma	0			1			Media			Varianta			Acuratetea
32	178 / 64 / 32 / 16 / 2	40 / 0.001	-	-	0.94	0.93	0.89	0.97	0.94	0.96	0.96	0.94	0.93	0.000	0.000	0.002	0.94
MLP 2.0																	
batch_size	Arhitectura	epochs / lr	dropout	step_size / gamma	0			1			Media			Varianta			Acuratetea
32	178 / 128 / 64 / 48 / 32 / 16 / 2	30 / 0.0005	-	-	0.95	0.78	0.86	0.93	0.99	0.96	0.94	0.89	0.91	0.000	0.022	0.005	0.93
MLP 3.0																	
batch_size	Arhitectura	epochs / lr	dropout	step_size / gamma	0			1			Media			Varianta			Acuratetea
32	178 / 128 / 64 / 48 / 32 / 16 / 2	50 / 0.01	-	20 / -	0.92	0.91	0.92	0.97	0.97	0.97	0.95	0.94	0.95	0.001	0.002	0.001	0.96

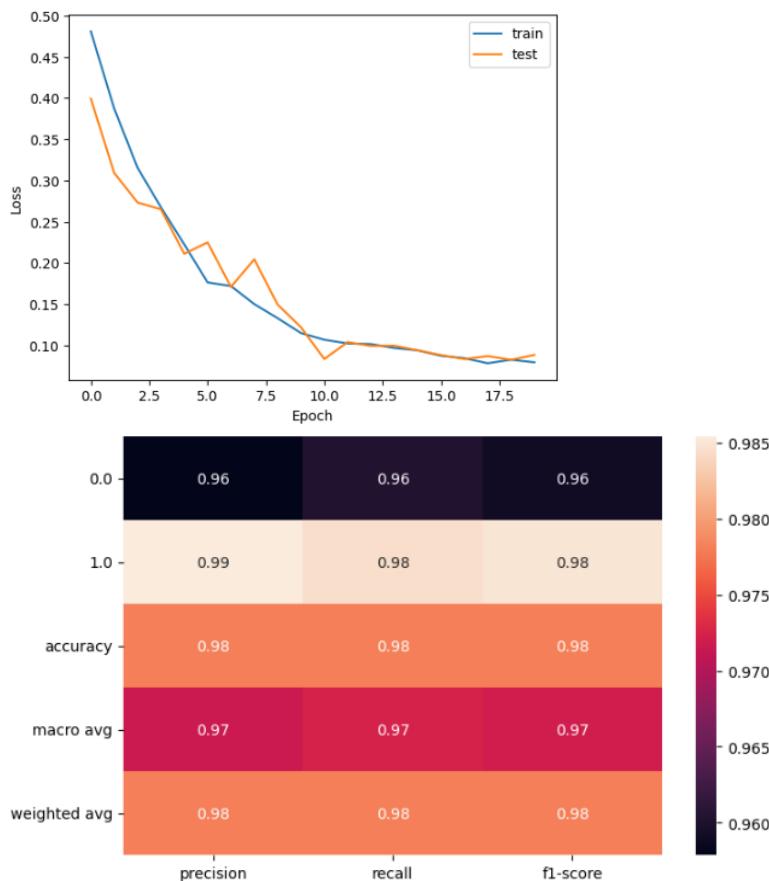
In continuare am folosit arhitectura InceptionTime:

```

model = InceptionModel(num_blocks=5, in_channels=1, out_channels=12,
                      bottleneck_channels=12, kernel_sizes=61, use_residuals=True,
                      num_pred_classes=2)

# define the loss function and optimizer
criterion = nn.BCEWithLogitsLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001, weight_decay=0.001)

```



Se obtin rezultate mult mai bune decat cele obtinute cu arhitectura MLP. Acest lucru se datoreaza datorita modului in care sunt extrase feature-urile. Curba de loss-test converge foarte bine, existand chiar potential pentru rezultate mai bune.

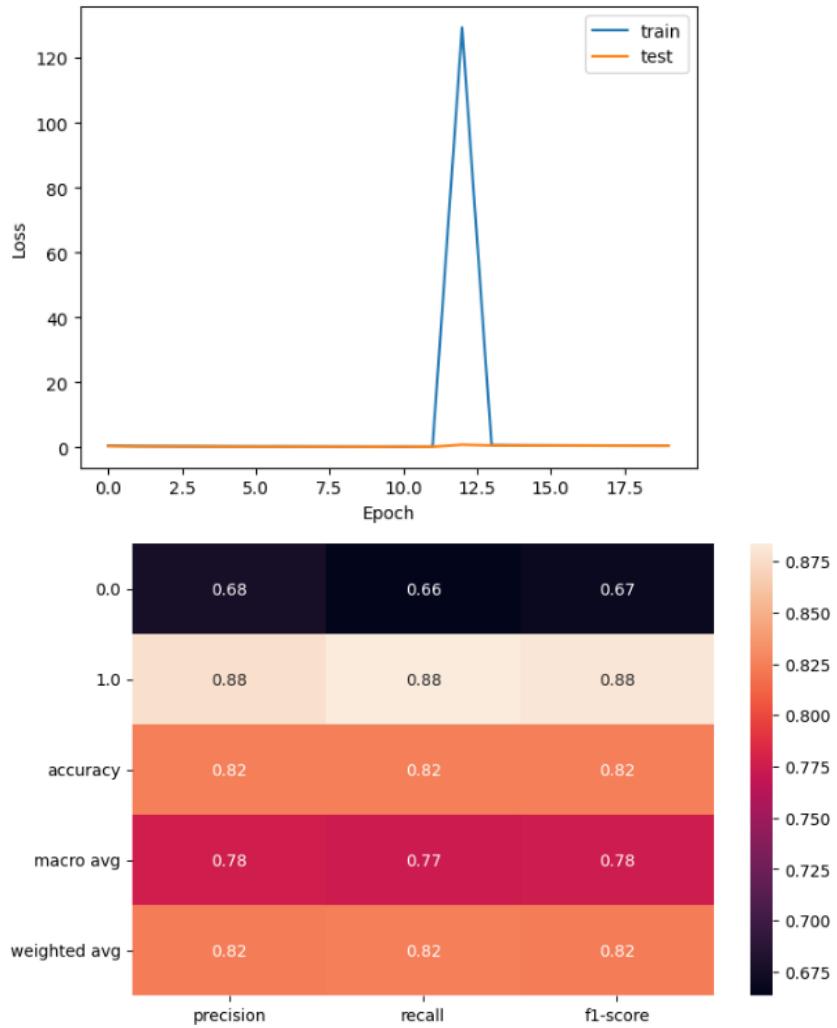
Am incercat si acesta rulare:

```

model = InceptionModel(num_blocks=5, in_channels=1, out_channels=14,
                      bottleneck_channels=14, kernel_sizes=61, use_residuals=True,
                      num_pred_classes=2)

# define the loss function and optimizer
criterion = nn.BCEWithLogitsLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001, weight_decay=0.001)

```



Probabil din cauza unui learning_rate prea mare, dar cred ca mult mai mult a influentat negativ cresterea de la 12 la 14 a numarului de canale. S-a fortat extragerea unor feature-uri, ceva ce a influentat negativ convergenta modelului. Pur si simplu prea multe canale in combinatia aceasta de parametrii.

Am incercat o alta rulare a modelului:

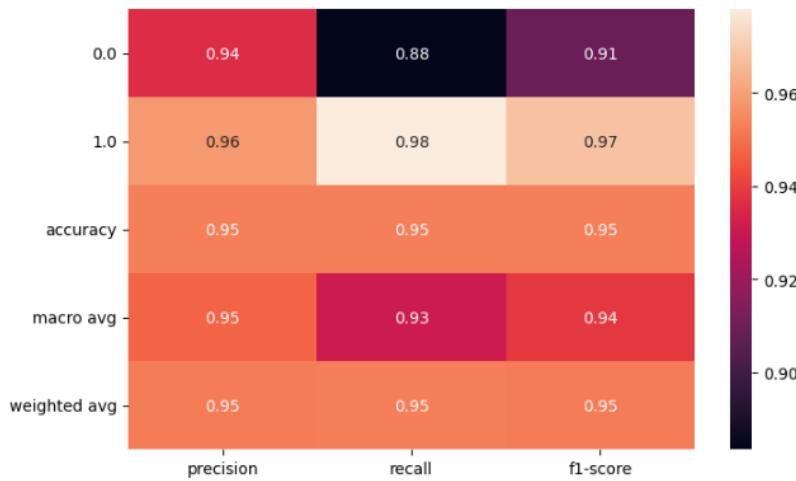
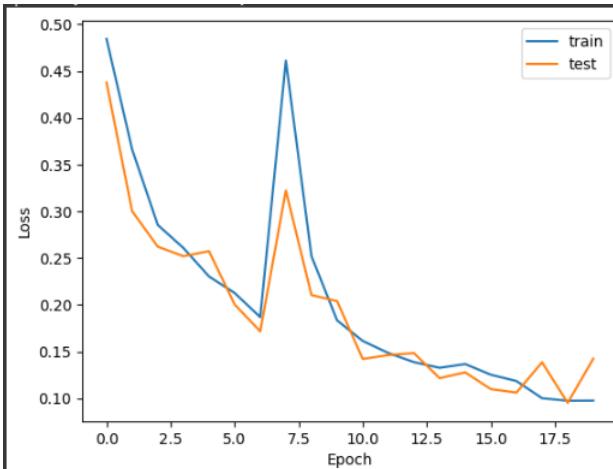
```
model = InceptionModel(num_blocks=5, in_channels=1, out_channels=12,
                      bottleneck_channels=12, kernel_sizes=41, use_residuals=True,
                      num_pred_classes=2)

# define the loss function and optimizer
criterion = nn.BCEWithLogitsLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001, weight_decay=0.001)

# lists to store the loss values for plotting
train_losses = []
test_losses = []

# train the model
for epoch in range(20):
```

Am ales parametrii din prima rulare, dar am schimbat kernel_size-ul din 61 in 41.



Exista destul de mult noise in traiectoria curbelor de loss. De asemenea, acea divergenta, este un pic cam accentuata. Totusi, cred ca as fi putut lasa modelul sa se antreneze pe un numar mai mare de epoci, parea ca inca converge. De asemenea, poate un scheduler as fi fost potrivit. Rezultatele nu au fost mai bune decat la prima rulare.

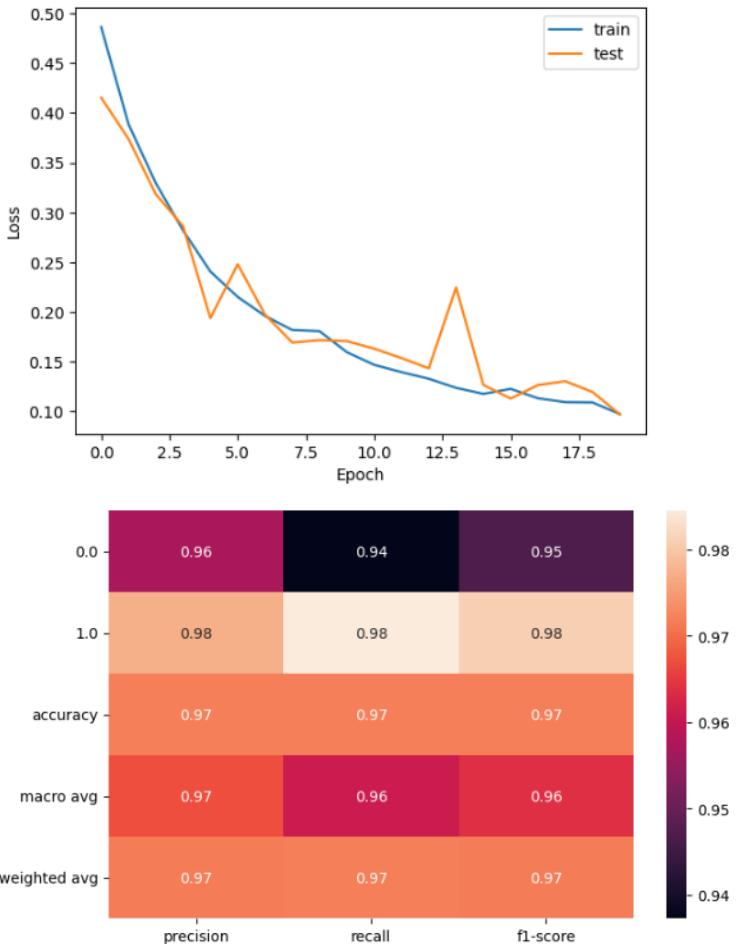
Am creat un alt model de tip InceptionTime. Este aceeasi arhitectura, doar are un layer de dropout = 0.1 inainte de layer-ul Linear.

```
model = InceptionModel(num_blocks=5, in_channels=1, out_channels=12,
                       bottleneck_channels=12, kernel_sizes=61, use_residuals=True,
                       num_pred_classes=2)

# define the loss function and optimizer
criterion = nn.BCEWithLogitsLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001, weight_decay=0.001)

# lists to store the loss values for plotting
```

Am selectat aceeasi parametri ca la prima rulare din modelul precedent pentru a vedea diferentele:

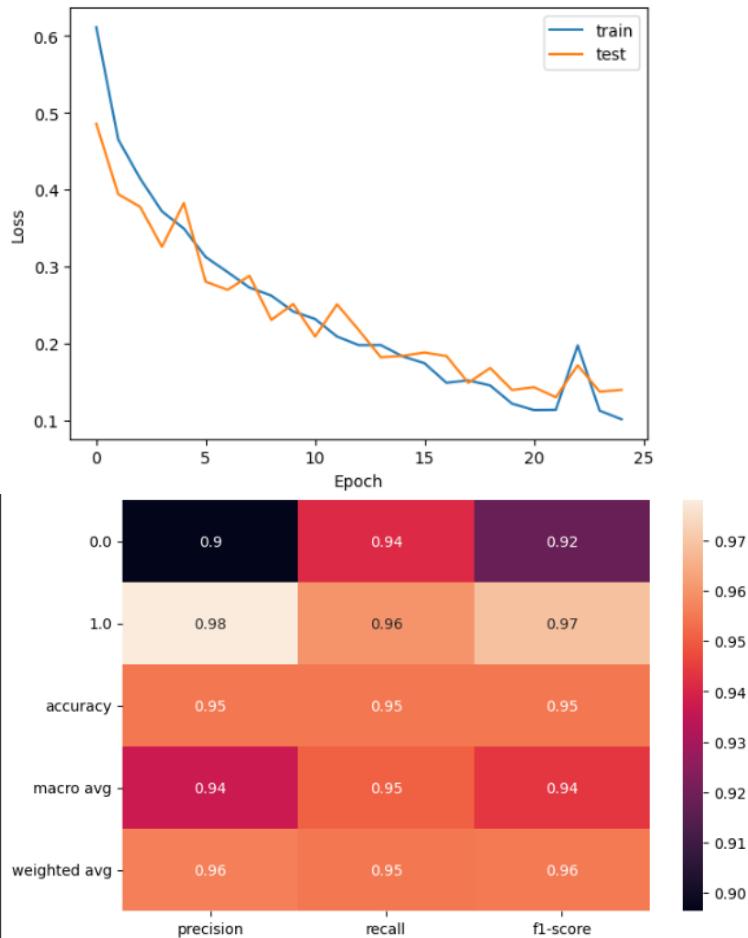


Se obtin rezultate aproape la fel de bune ca ale celuilalt model. Totusi, cred ca exista mai mult potential pentru acest model. Curba de loss indica faptul ca un numar mai mare de epoci mai mare de epoci ar fi bun. Un lucru important de precizat ar fi faptul ca se obtin aceleasi rezultate, dar acest model are un layer de dropout care renunta la 10% dintre neuroni la antrenare. Deci un numar mai mare de epoci ar putea echilibrata pierderea de informatie generata de dropout.

Am incercat sa vad pe acest model, cum se va comporta antrenarea daca cresc numarul de canale la 14 de la 12.

```
model = InceptionModel(num_blocks=5, in_channels=1, out_channels=14,
                       bottleneck_channels=14, kernel_sizes=61, use_residuals=True,
                       num_pred_classes=2)

# define the loss function and optimizer
criterion = nn.BCEWithLogitsLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001, weight_decay=0.001)
```



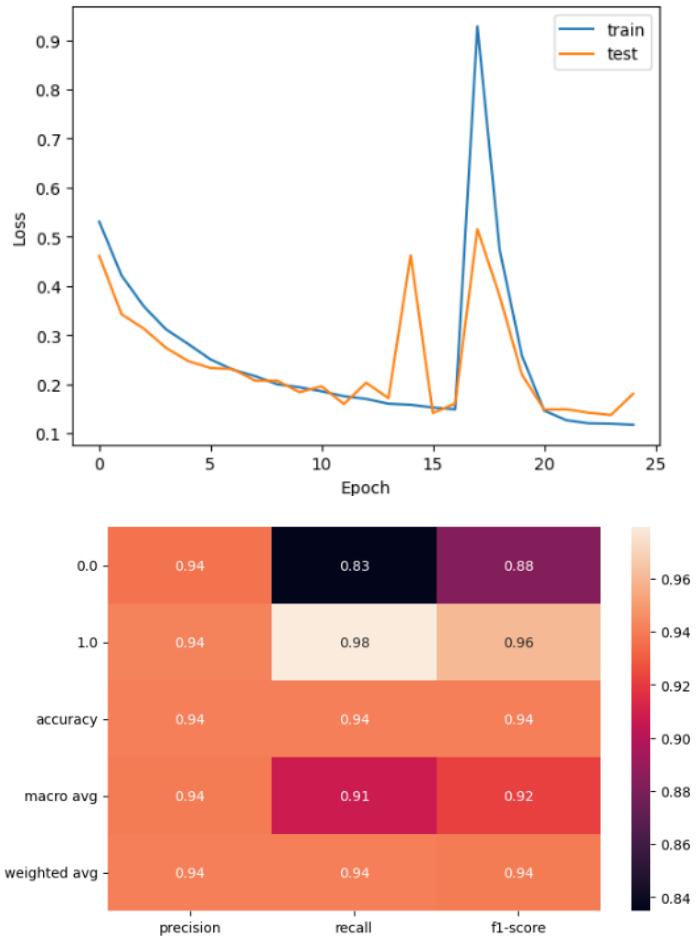
Se observa ca layer-ul de dropout a ajutat, modelul nu mai diverge. Se obtin rezultate bune, dar nu mai bune decat cu alta configuratie de parametrii.

Am incercat sa reiau modelul rulat cu 12 canale in loc de 14 (cea mai buna rulare de pana acum) si sa o rulez cu batch_size = 16, in loc de 32 pentru a vedea diferențele.

```
model = InceptionModel(num_blocks=5, in_channels=1, out_channels=12,
                       bottleneck_channels=12, kernel_sizes=[6,1], use_residuals=True,
                       num_pred_classes=2)

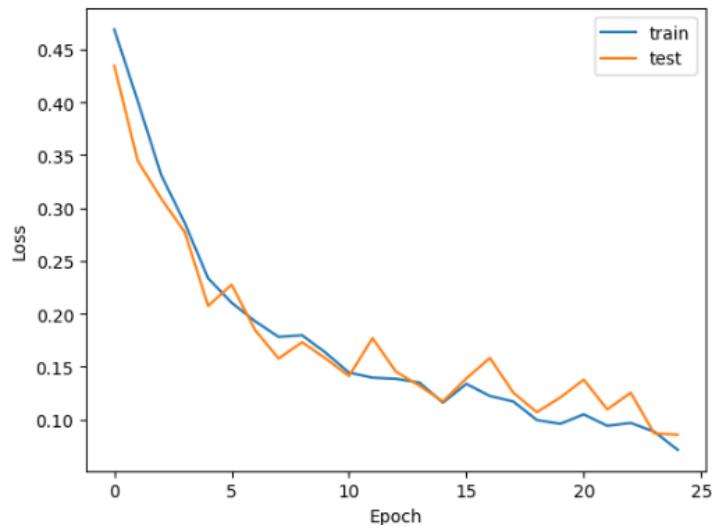
# define the loss function and optimizer
criterion = nn.BCEWithLogitsLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001, weight_decay=0.001)

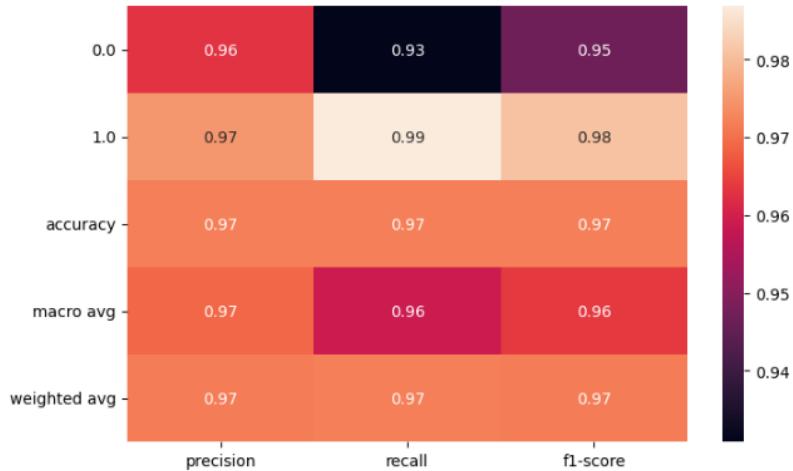
# lists to store the loss values for plotting
```



Desi la inceput curba de loss coboara frumos, probabil din cauza unui learning_rate neajustat cu un scheduler, incepe sa aiba mult noise. Este normal ca atunci cand batch_size-ul scade sa apară mai mult noise. Apare efectul care se evidențiază cel mai mult la Stochastic Run.

De asemenea, am testat ce se întâmplă cu aceasta rulare dacă cresc batch_size-ul la 128.





Se obtin rezultate la fel de bune ca si in rularea cu batch_size = 32. Curba de loss converge, totusi exista mai mult noise in traectoria sa. Poate ar fi fost bine sa cresc numarul de epoci.

Am incercat sa implementez un model nou de InceptionTime. Am folosit al doilea model si in loc de layer-ul clasic Linear, am adaugat dupa layer-ul de dropout, un set de layer-e care se regasesc si la MLP 3.0.

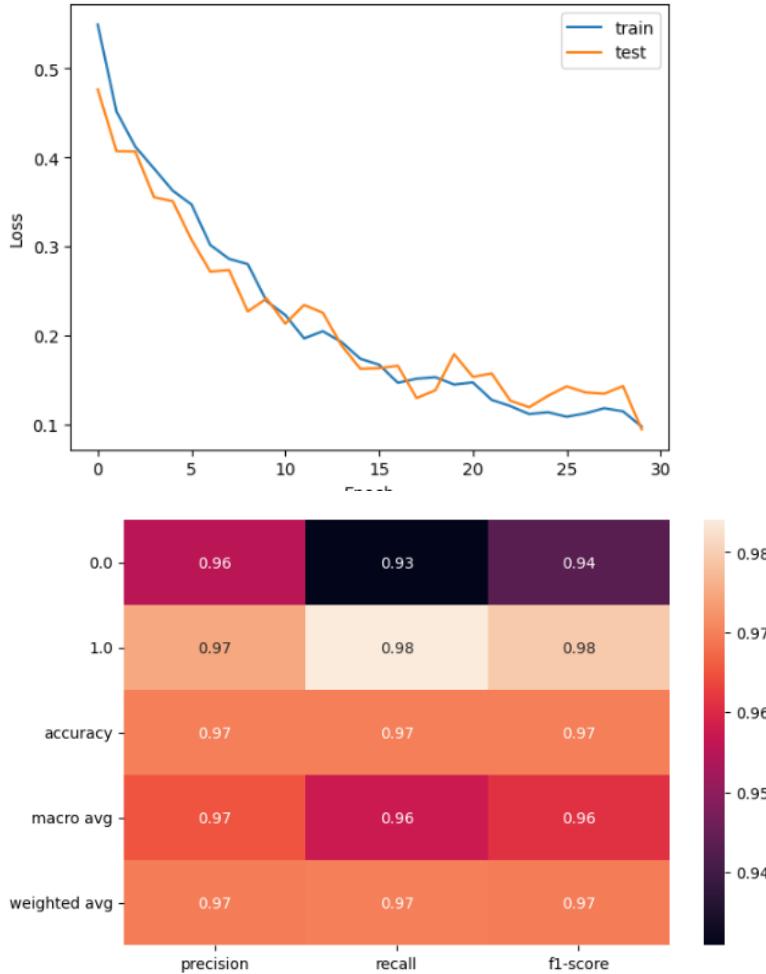
```
def forward(self, x: torch.Tensor) -> torch.Tensor: # type: ignore
    x = self.blocks(x).mean(dim=-1) # the mean is the global average pooling

    dropout = nn.Dropout(0.1);
    x = dropout(x)
    x = F.relu(self.fc1(x))
    x = F.relu(self.fc2(x))
    x = F.relu(self.fc3(x))
    x = F.relu(self.fc4(x))
    x = F.relu(self.fc5(x))
    x = self.fc6(x)
    return x
```

Am rulat modelul cu aceasta configuratie:

```
1 model = InceptionModel(num_blocks=5, in_channels=1, out_channels=12,
2                         bottleneck_channels=12, kernel_sizes=61, use_residuals=True,
3                         num_pred_classes=2)
4
5 # define the loss function and optimizer
6 criterion = nn.BCEWithLogitsLoss()
7 optimizer = torch.optim.Adam(model.parameters(), lr=0.001, weight_decay=0.001)
8
9 # lists to store the loss values for plotting
10 train_losses = []
11 test_losses = []
12
13 # train the model
14 for epoch in range(30):
15     epoch_train_loss = 0
```

Am crescut numarul de epoci si am ales in rest cea mai buna varianta de parametrii de pana acum.



Se obtin rezultate la fel de bune. Nu exista diferente prea mari. Am crescut un pic numarul de epoci, dar layer-urile lineare de tip MLP 3.0 trebuie sa fie si ele ajustate si atunci compenseaza cu cele 5 epoci adaugate in plus. Poate ar fi trebuit totusi sa las mai multe epoci, dar rezultatele sunt bune si algoritmul converge bine.

InceptionTime																	
batch_size	num_blocks / out_ch / bottleneck_ch / kernel_size	epochs / lr	dropout	step_size / gamma	0			1			Media			Varianta			Acuratetea
					Precision	Recall	F1-score										
32	5 / 12 / 12 / 61	20 / 0.001	-	-	0.96	0.96	0.96	0.99	0.98	0.98	0.98	0.97	0.97	0.000	0.000	0.000	0.98
32	5 / 14 / 14 / 61	20 / 0.001	-	-	0.68	0.66	0.67	0.88	0.88	0.88	0.78	0.77	0.78	0.020	0.024	0.022	0.82
32	5 / 12 / 12 / 41	20 / 0.001	-	-	0.94	0.88	0.91	0.96	0.98	0.97	0.95	0.93	0.94	0.000	0.005	0.002	0.95

InceptionTime 2.0																	
batch_size	num_blocks / out_ch / bottleneck_ch / kernel_size	epochs / lr	dropout	step_size / gamma	0			1			Media			Varianta			Acuratetea
					Precision	Recall	F1-score										
32	5 / 12 / 12 / 61	20 / 0.001	0.1	-	0.96	0.94	0.95	0.98	0.98	0.98	0.97	0.96	0.97	0.000	0.001	0.000	0.97
32	5 / 14 / 14 / 61	25 / 0.001	0.1	-	0.90	0.94	0.92	0.98	0.96	0.97	0.94	0.95	0.95	0.003	0.000	0.001	0.95
16	5 / 12 / 12 / 61	25 / 0.001	0.1	-	0.94	0.83	0.88	0.94	0.98	0.96	0.94	0.91	0.92	0.000	0.011	0.003	0.94
128	5 / 12 / 12 / 61	25 / 0.001	0.1	-	0.96	0.93	0.95	0.97	0.99	0.98	0.97	0.96	0.97	0.000	0.002	0.000	0.97

InceptionTime 3.0 (InceptionTime 2.0 + MLP 3.0)																	
batch_size	num_blocks / out_ch / bottleneck_ch / kernel_size	epochs / lr	dropout	step_size / gamma	0			1			Media			Varianta			Acuratetea
					Precision	Recall	F1-score										
128	5 / 12 / 12 / 61	30 / 0.001	0.1	-	0.96	0.93	0.94	0.97	0.98	0.98	0.97	0.96	0.96	0.000	0.001	0.001	0.97

Mai departe, am folosit o arhitectura de tip LSTM.

```

1 class ComplexLSTM(nn.Module):
2     def __init__(self, input_size, hidden_size, num_layers, output_size):
3         super(ComplexLSTM, self).__init__()
4
5         self.input_size = input_size
6         self.hidden_size = hidden_size
7         self.num_layers = num_layers
8         self.output_size = output_size
9
10        self.lstm = nn.LSTM(input_size, hidden_size, num_layers, batch_first=True)
11        self.fc1 = nn.Linear(hidden_size, 64)
12        self.fc2 = nn.Linear(64, output_size)
13
14    def forward(self, x):
15        # Initialize the hidden state and cell state
16        h0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(x.device)
17        c0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(x.device)
18
19        # Forward pass through the LSTM layer
20        out, (hn, cn) = self.lstm(x, (h0, c0))
21
22        # Pass the final hidden state through Linear layers
23        out = self.fc1(hn[-1])
24        out = self.fc2(out)
25
26    return out

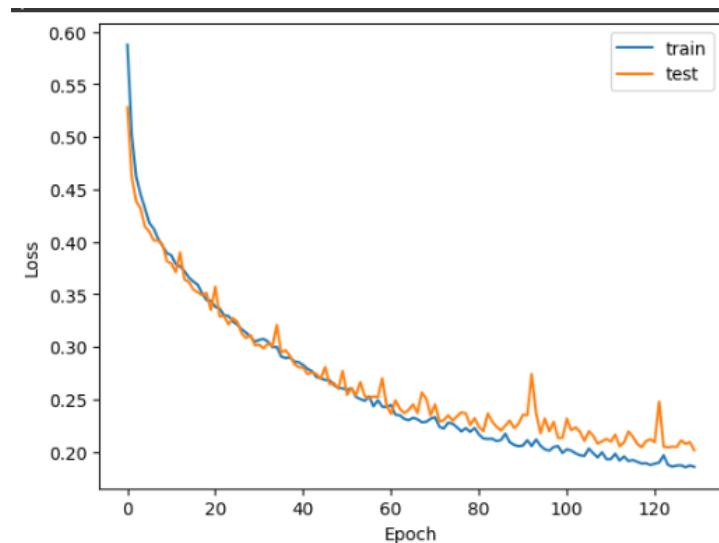
```

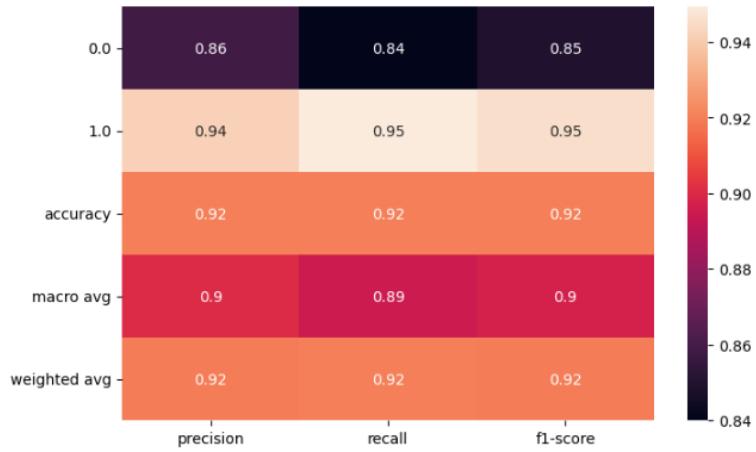
Folosind aceasta configuratie am rulat avand un batch_size = 256.

```

1 model = ComplexLSTM(input_size = 178, hidden_size = 50, num_layers = 1, output_size = 2)
2
3 # define the loss function and optimizer
4 criterion = nn.BCEWithLogitsLoss()
5 optimizer = torch.optim.Adam(model.parameters(), lr=0.001, weight_decay=0.001)
6
7 # lists to store the loss values for plotting
8 train_losses = []
9 test_losses = []
10
11 # train the model
12 for epoch in range(130):

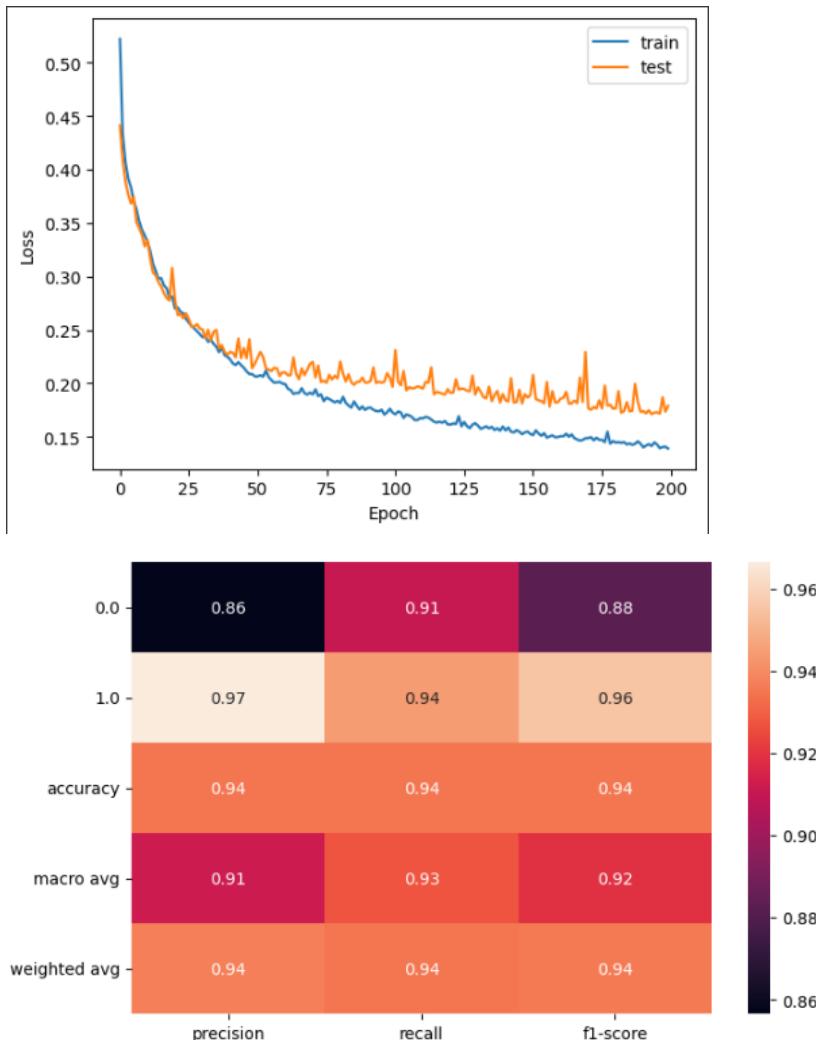
```





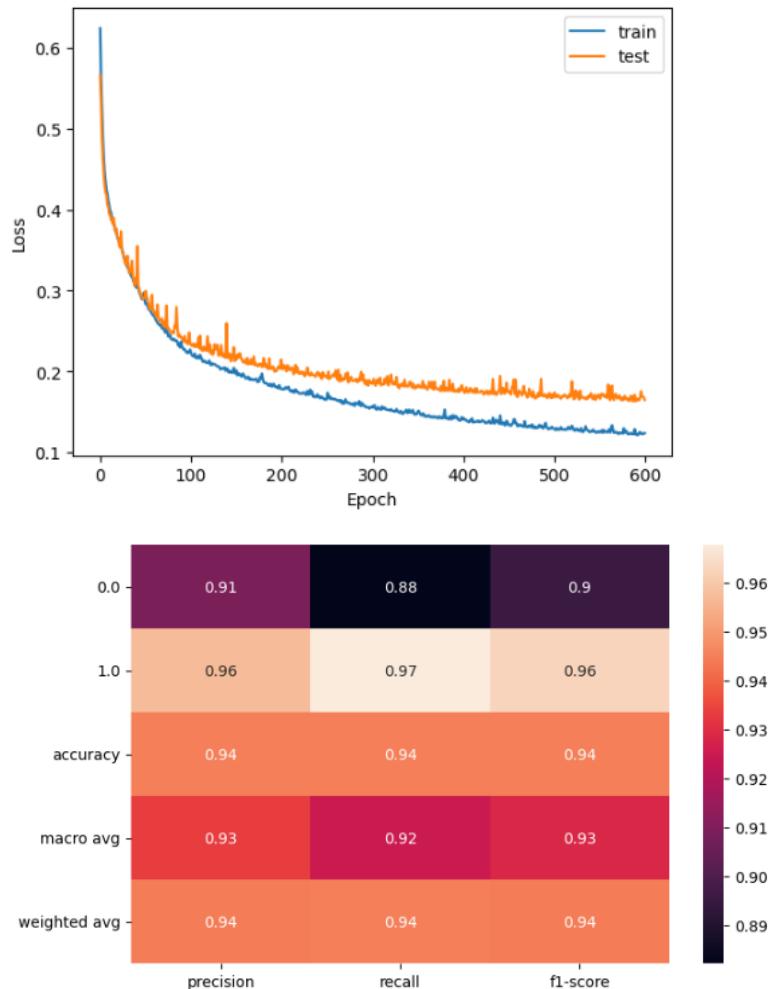
Rezultatele nu sunt mai bune decat cele obtinute prin InceptionTime. Curba de loss arata bine, dar algoritmul converge cam greu si desi pare sa mai aiba potential de convergenta, totusi loss-ul test-ului pare sa se distanteze un pic de loss-ul de tip train.

Am incercat si o rulare cu batch_size = 64.



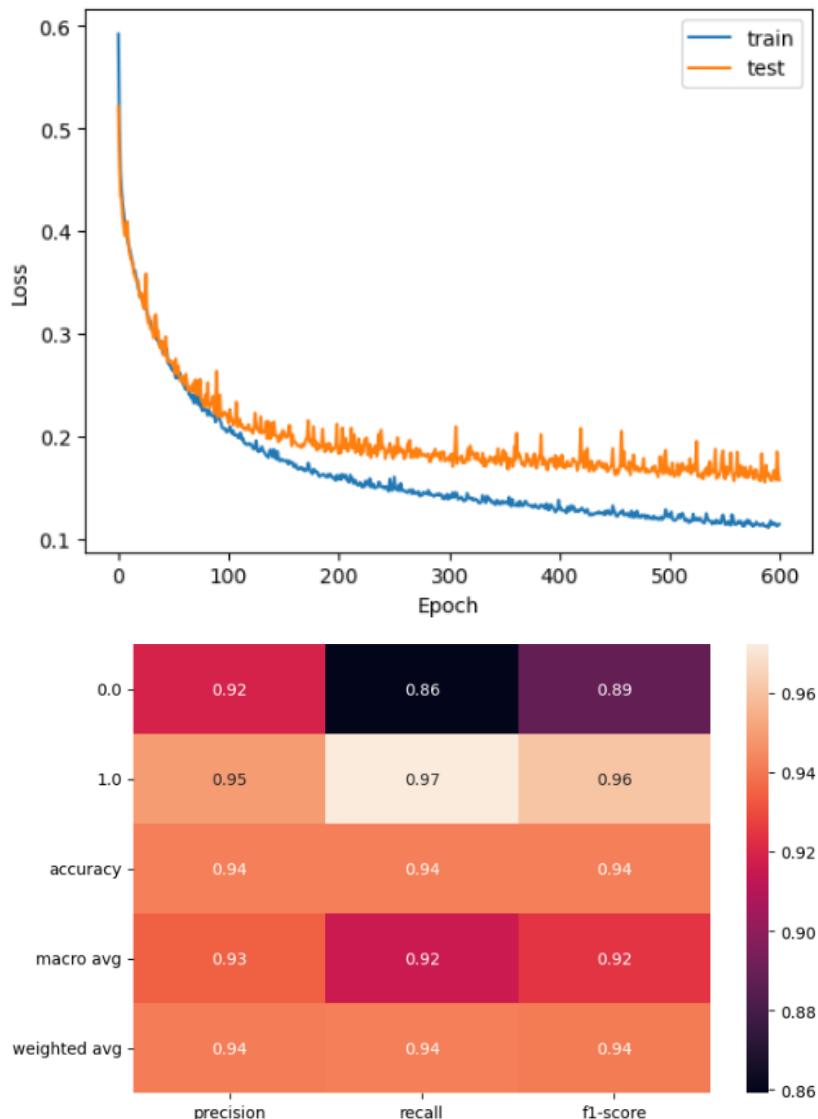
Am adaugat 70 de epoci in plus. Se remarcă faptul că distanța dintre curba de loss de tip test și cea de tip train se realizează mai devreme, încă de la epoca 50 și distanța pare mai mare. Nu este de preferat.

Am incercat și o rulare cu batch_size = 512. Am setat 600 de epoci.



Se observă că algoritmul nu reușește să conveară mai mult. Rezultatele nu sunt cu mult diferite. Noise-ul este destul de mic.

Am incercat o alta combinatie de parametrii. Am setat hidden_size-ul = 100.



Nu se obtin rezultate diferite. Loss-ul pare sa contina mai mult zgomot, dar nu converge mai bine.

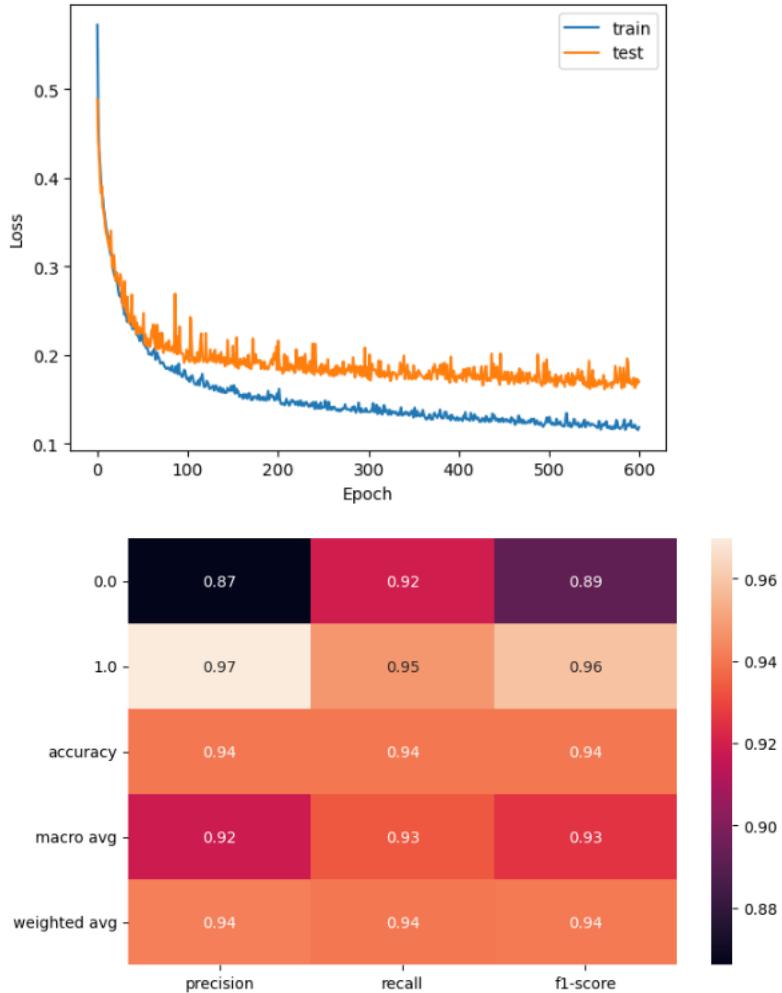
Am incercat crearea unui modul de LSTM cu dropout = 0.1.

```
model = ComplexLSTM(input_size = 178, hidden_size = 50, num_layers = 1, output_size = 2)

# define the loss function and optimizer
criterion = nn.BCEWithLogitsLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.002, weight_decay=0.001)

# lists to store the loss values for plotting
```

Cu aceasta configuratie de parametrii nu am obtinut rezultate deosebite.



Se remarcă doar noise-ul destul de puternic pe care îl are curba de loss.

Am încercat să creasc numărul de layer-e de tip LSTM din block.

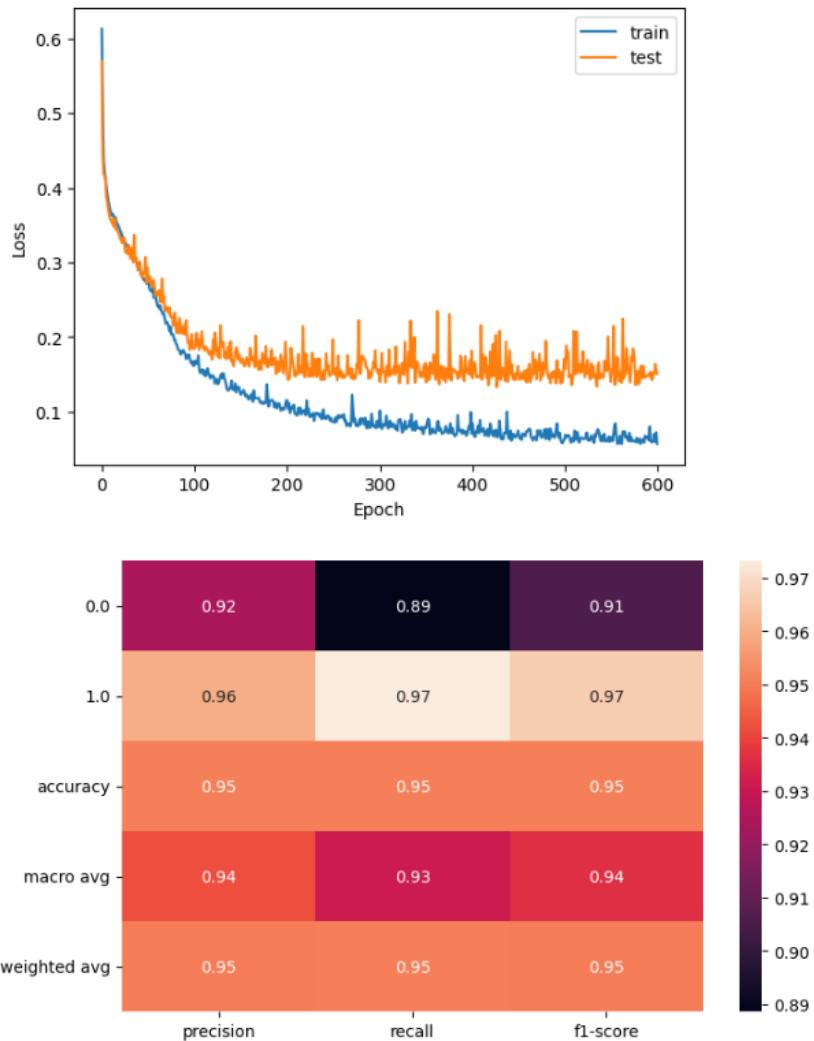
```
model = ComplexLSTM(input_size = 178, hidden_size = 50, num_layers = 3, output_size = 2)

# define the loss function and optimizer
criterion = nn.BCEWithLogitsLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.002, weight_decay=0.001)

# lists to store the loss values for plotting
train_losses = []
test_losses = []

# train the model
for epoch in range(600):
    epoch_train_loss = 0
```

Rezultatele au fost mai bune, dar există mult noise, care probabil trebuie ajustat cu un scheduler:



Am revenit la primul model de LSTM si am introdus un scheduler:

```
model = ComplexLSTM(input_size = 178, hidden_size = 50, num_layers = 1, output_size = 2)

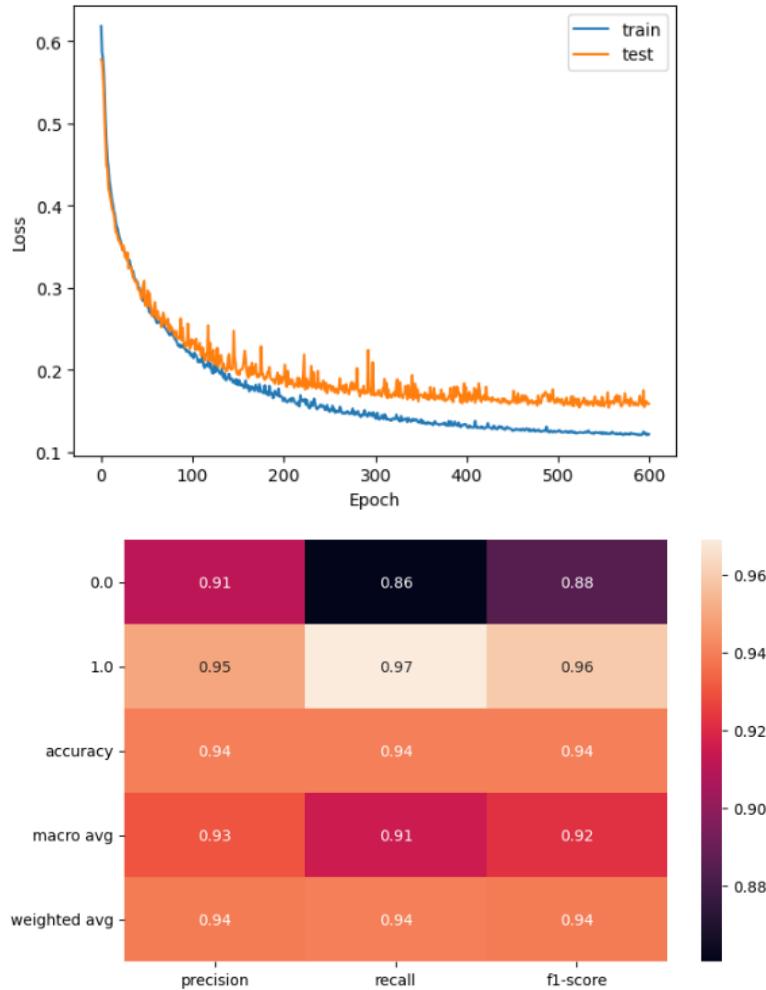
# define the loss function and optimizer
criterion = nn.BCEWithLogitsLoss()

# define the optimizer with SGD and momentum
optimizer = torch.optim.SGD(model.parameters(), lr=0.03, momentum=0.9, weight_decay=0.001)

# define the learning rate scheduler
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=50, gamma = 0.8)

# lists to store the loss values for plotting
train_losses = []
test_losses = []

# train the model
for epoch in range(600):
    epoch_train_loss = 0
```



Rezultate sunt bune, dar nu deosebite de celelalte. Curba de loss pare sa coboare un pic mai bine.

Am incercat alt scheduler:

```
model = ComplexLSTM(input_size = 178, hidden_size = 50, num_layers = 1, output_size = 2)

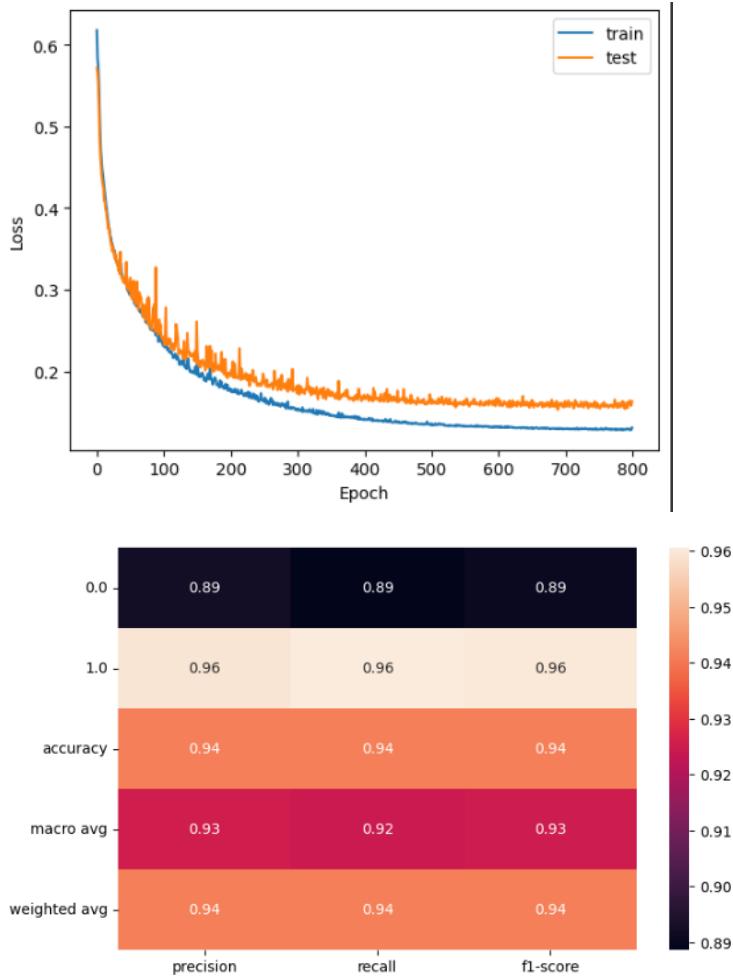
# define the loss function and optimizer
criterion = nn.BCEWithLogitsLoss()

# define the optimizer with SGD and momentum
optimizer = torch.optim.SGD(model.parameters(), lr=0.03, momentum=0.9, weight_decay=0.001)

# define the learning rate scheduler
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=40, gamma = 0.8)

# lists to store the loss values for plotting
train_losses = []
test_losses = []

# train the model
for epoch in range(800):
```



Scheduler-ul nu pare sa influenteze foarte mult rezultatele.

Am incercat sa fac modelul mai complex prin cresterea hidden_size-ului:

```
model = ComplexLSTM(input_size = 178, hidden_size = 70, num_layers = 1, output_size = 2)

# define the loss function and optimizer
criterion = nn.BCEWithLogitsLoss()

# define the optimizer with SGD and momentum
optimizer = torch.optim.SGD(model.parameters(), lr=0.03, momentum=0.9, weight_decay=0.001)

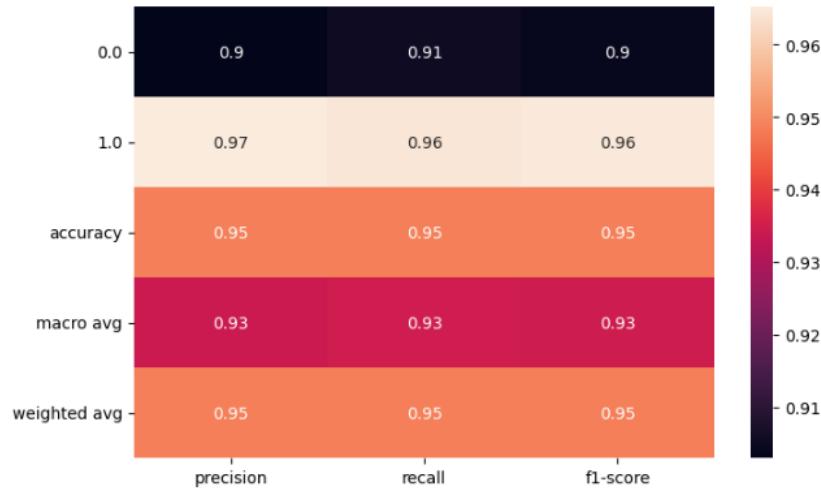
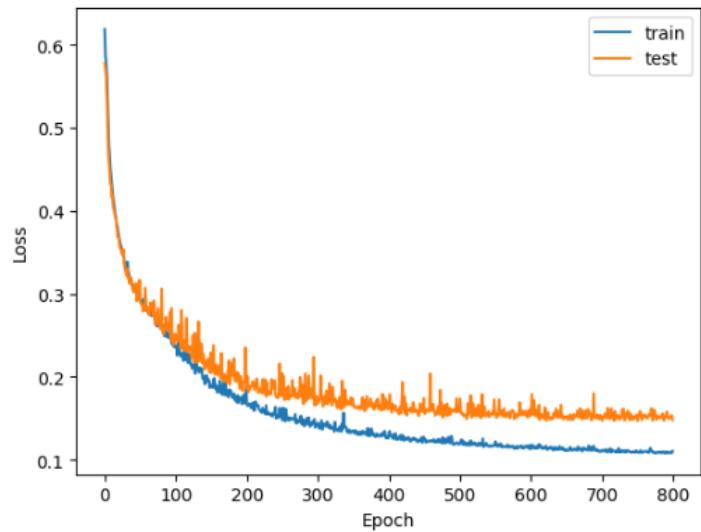
# define the learning rate scheduler
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=70, gamma = 0.8)

# lists to store the loss values for plotting
train_losses = []
test_losses = []

# train the model
for epoch in range(800):
    epoch_train_loss = 0

    for xb, yb in ttrain_dl:
```

Pare sa obtine rezultate mai bune:



Concluzie: O combinatie dintre un scheduler bine ales, un hidden_size bun si un numar de layer-e de tip LSTM mai mare ≥ 1 poate fi solutia pentru o solutie mai performanta.

LSTM																	
batch_size	Arhitectura - Dropout	epochs / lr	layers / hidden_size	step_size / gamma	0			1			Media			Varianta			Acuratetea
					Precision	Recall	F1-score										
256	178 / 64 / 2 - no dropout	130 / 0.001	1 / 50	-	0.86	0.84	0.85	0.94	0.95	0.95	0.90	0.90	0.90	0.003	0.006	0.005	0.92
64	178 / 64 / 2 - no dropout	200 / 0.001	1 / 50	-	0.84	0.91	0.88	0.97	0.94	0.96	0.91	0.93	0.92	0.008	0.000	0.003	0.94
512	178 / 64 / 2 - no dropout	600 / 0.001	1 / 50	-	0.91	0.88	0.90	0.96	0.97	0.96	0.94	0.93	0.93	0.001	0.004	0.002	0.94
256	178 / 64 / 2 - no dropout	600 / 0.001	1 / 100	-	0.92	0.86	0.89	0.95	0.97	0.96	0.94	0.92	0.93	0.000	0.006	0.002	0.94
LSTM 2.0																	
batch_size	Arhitectura - Dropout	epochs / lr	layers / hidden_size	step_size / gamma	0			1			Media			Varianta			Acuratetea
					Precision	Recall	F1-score										
256	178 / 64 / 2 - dropout = 0.1	600 / 0.002	1 / 50	-	0.87	0.92	0.89	0.97	0.95	0.96	0.92	0.94	0.93	0.005	0.000	0.002	0.94
256	178 / 64 / 2 - dropout = 0.1	600 / 0.002	3 / 50	-	0.92	0.89	0.91	0.96	0.97	0.97	0.94	0.93	0.94	0.001	0.003	0.002	0.95
LSTM 3.0																	
batch_size	Arhitectura - Dropout	epochs / lr	layers / hidden_size	step_size / gamma	0			1			Media			Varianta			Acuratetea
					Precision	Recall	F1-score										
256	178 / 64 / 2 - no dropout	600 / 0.03	1 / 50	50 / 0.8	0.91	0.86	0.88	0.95	0.97	0.96	0.93	0.92	0.92	0.001	0.006	0.003	0.94
256	178 / 64 / 2 - no dropout	800 / 0.03	1 / 50	40 / 0.8	0.89	0.89	0.89	0.96	0.96	0.96	0.93	0.93	0.93	0.002	0.002	0.002	0.94
256	178 / 64 / 2 - no dropout	800 / 0.03	1 / 70	70 / 0.8	0.90	0.91	0.90	0.97	0.96	0.96	0.94	0.94	0.93	0.002	0.001	0.002	0.95

