Georgescu Elena-Alexandra, Grupa 405

Aplicație pentru recunoasterea genurilor muzicale

1. Introducere - prezentarea setului de date și enuntarea obiectivelor

In cadrul acestui proiect am utilizat setul de date disponibil pe platforma Kaggle: https://www.kaggle.com/datasets/andradaolteanu/gtzan-dataset-music-genre-classification

Folosind acest dataset am ales sa construiesc o aplicație de recunoaștere a genurilor muzicale pe baza input-urilor audio furnizate.

În cadrul dataset-ului regasim atat fișiere .wav cu 10 categorii muzicale si 100 de exemple în fiecare dintre acestea, cat și două fișiere .csv cu datele importante deja extrase din fișierele audio. Totodata, regasim si distribuția liniara a sunetelor pentru fiecare sample melodic, in format .png.

Totusi, pentru a rezolva cerințele proiectului am utilizat fisierele .wav pentru a extrage exact datele de care aveam nevoie.

Preprocesarea datelor

Asa cum enuntam mai sus, am utilizat intreaga colectie de fișiere .wav puse la dispozitie de catre datasetul ales. Scopul este acela de a extrage feature-urile care ne vor fi de folos pentru antrenarea modelelor. Am realizat acest lucru folosind librăria Python *librosa*, care ajută la analiza și procesarea semnalelor audio.

```python
import librosa
import os
import numpy as np
import pandas as pd
from librosa.feature.rhythm import tempo as rhythm_tempo

base_path = "data/Data/genres_original"
genres = os.listdir(base_path)
features = []

for genre in genres:
    genre_path = os.path.join(base_path, genre)
    for file in os.listdir(genre_path):
        if file.endswith(".wav"):
            path = os.path.join(genre_path, file)
            try:
                y, sr = librosa.load(path)
```

```
                tempo_val = rhythm_tempo(y=y, sr=sr)[0] # variatia de ritm a
piesei
                spectral_centroid = librosa.feature.spectral_centroid(y=y,
sr=sr).mean() # inaltimea sunetului
                zero_crossing_rate =
librosa.feature.zero_crossing_rate(y).mean() # frecventa valorii 0 in
evolutia semnalului - ajuta la identificarea zgomotelor percutante
                rmse = librosa.feature.rms(y=y).mean() # Root Mean Square
Energy - energia generala a semnalului
                bandwidth = librosa.feature.spectral_bandwidth(y=y,
sr=sr).mean() # latimea benzii de frecventa - urmareste cat de mult variaza
semnalul
                chroma = librosa.feature.chroma_stft(y=y, sr=sr).mean() #
distributia semnalului pe notele muzicale - reflecta tonalitatea
                mfccs = librosa.feature.mfcc(y=y, sr=sr, n_mfcc=13) #
Mel-Frequency Cepstral Coefficients - reprezentare compacta a sunetului,
apropiata de perceptia umana
                mfccs_mean = mfccs.mean(axis=1) # calculam media valorilor
mfcc extrase
                beat_strength = np.mean(librosa.onset.onset_strength(y=y,
sr=sr)) # puterea batailor
                tempogram = librosa.feature.tempogram(y=y, sr=sr)
                tempo_variation = np.std(tempogram) # deviatia tempo-ului
                onsets = librosa.onset.onset_detect(y=y, sr=sr)
                onset_density = len(onsets) / librosa.get_duration(y=y,
sr=sr) # densitatea evenimentelor sonore

                feature = {
                    'filename': file,
                    'genre': genre,
                    'tempo': tempo_val,
                    'spectral_centroid': spectral_centroid,
                    'zero_crossing_rate': zero_crossing_rate,
                    'rmse': rmse,
                    'bandwidth': bandwidth,
                    'chroma': chroma,
                    'beat_strength': beat_strength,
                    'tempo_variation': tempo_variation,
                    'onset_density': onset_density
                }

                # adaugam individual densitatile pentru fiecare valoare din
mfcc

                for i in range(13):
                    feature[f"mfcc{i+1}"] = mfccs_mean[i]

                features.append(feature)

            except Exception as e:
```

```
                print(f"[Eroare la {file}]: {e}")

# cream csv-ul care va fi utilizat de catre programul PySpark
df = pd.DataFrame(features)
df.to_csv('features/genre_features.csv', index=False)
```

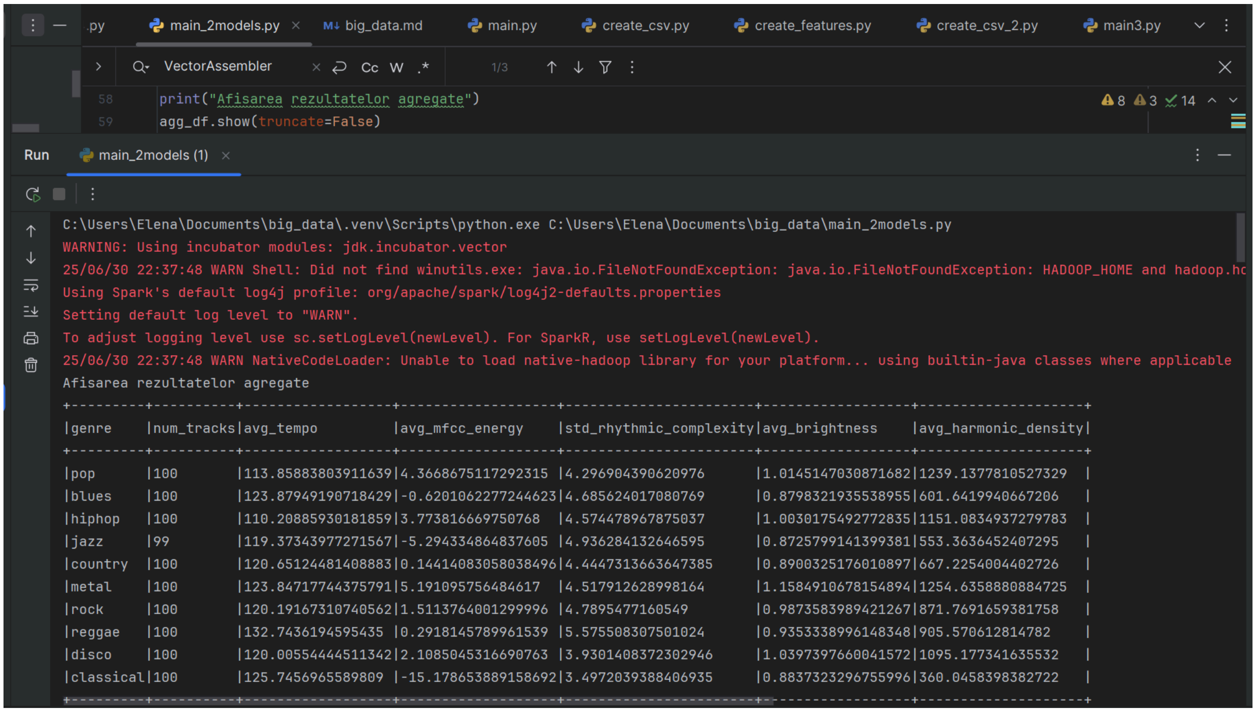
2. Procesarea datelor

În cadrul acestui pas voi initializa sesiunea de Spark, voi încarca datele extrase din fișierul .csv într-un Dataframe si voi crea coloane noi pe baza datelor deja încărcate folosind agregari din SparkSQL.

```
from pyspark.sql import SparkSession
from pyspark.ml.feature import StringIndexer, VectorAssembler, StandardScaler
from pyspark.sql.functions import expr, col, when, lit
from pyspark.sql.functions import avg, stddev, count


# Inițializare Spark
spark = SparkSession.builder.appName("MusicClassification_CV").getOrCreate()

# Citirea si curatarea datelor
df = spark.read.csv("features/genre_features.csv", header=True,
inferSchema=True).dropna()

# Recreearea view-ului SQL care ne ajuta la crearea unor coloane suplimentare
df.createOrReplaceTempView("audio_raw")

# Script SparkSQL pentru transformarea datelor
sql_df = spark.sql("""
    SELECT *,
           tempo * zero_crossing_rate AS rhythmic_complexity, --
complexitatea ritmica ajuta la recunoasterea melodiilor mai energice, cu
variatii dese de ritm
           chroma * spectral_centroid AS harmonic_density, -- densitatea
armonica ajuta la masurarea inaltimii sunetelor facand diferenta dintre jazz
si muzica clasica
           CASE WHEN bandwidth != 0 THEN spectral_centroid / bandwidth ELSE
0.0 END AS brightness_score -- luminozitatea timbrala ajuta la masurarea
concentrarii sunetului pe suprafata sonora
    FROM audio_raw
""")

mfcc_cols = [f"mfcc{i}" for i in range(1, 14)]
sql_df = sql_df.withColumn("mfccs_array", expr(f"array({',
'.join(mfcc_cols)})"))
```

```
sql_df = sql_df.withColumn("mfcc_energy", expr("aggregate(mfccs_array, 0D,
(acc, x) -> acc + x) / size(mfccs_array)")) # masoara consistenta timbrala
sql_df = sql_df.withColumn("percussive_ratio", when(
    col("mfcc_energy") != 0, col("rhythmic_complexity") / col("mfcc_energy")
).otherwise(lit(0.0))) # indica dominanta ritmica in raport cu structura
generala a piesei

# renuntam la aceasta coloana intrucat nu este necesara in vederea antrenarii
df_proc = sql_df.drop("mfccs_array")

# Calculam numarul, media, varianta si agregarea pe anumite coloane
utilizate, grupate dupa genul muzical
agg_df = df_proc.groupBy("genre").agg(
    count("*").alias("num_tracks"),
    avg("tempo").alias("avg_tempo"),
    avg("mfcc_energy").alias("avg_mfcc_energy"),
    stddev("rhythmic_complexity").alias("std_rhythmic_complexity"),
    avg("brightness_score").alias("avg_brightness"),
    avg("harmonic_density").alias("avg_harmonic_density")
)

print("Afisarea rezultatelor agregate")
agg_df.show(truncate=False)

# Transformam echitele genurilor in valori numerice
indexer = StringIndexer(inputCol="genre", outputCol="label")
df_indexed = indexer.fit(df_proc).transform(df_proc)

label_to_genre = {i: genre for i, genre in
enumerate(indexer.fit(df_proc).labels)}


# Definim coloanele ce vor fi utilizate in procesul de antrenare
feature_cols = [
    "tempo", "spectral_centroid", "zero_crossing_rate", "rmse", "bandwidth",
"chroma",
    "mfcc_energy", "rhythmic_complexity", "harmonic_density",
"brightness_score", "percussive_ratio",
    "mfcc1", "mfcc2", "mfcc3", "mfcc4", "mfcc5", "mfcc6", "mfcc7", "mfcc8",
"mfcc9", "mfcc10", "mfcc11", "mfcc12", "mfcc13",
    "beat_strength", "tempo_variation", "onset_density"
]
```

Output agregare:



```
58    print("Afisarea rezultatelor agregate")
59    agg_df.show(truncate=False)

C:\Users\Elena\Documents\big_data\.venv\Scripts\python.exe C:\Users\Elena\Documents\big_data\main_2models.py
WARNING: Using incubator modules: jdk.incubator.vector
25/06/30 22:37:48 WARN Shell: Did not find winutils.exe: java.io.FileNotFoundException: java.io.FileNotFoundException: HADOOP_HOME and hadoop.hc
Using Spark's default log4j profile: org/apache/spark/log4j2-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
25/06/30 22:37:48 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Afisarea rezultatelor agregate
+---------+----------+------------------+------------------+----------------------+------------------+-------------------+
|genre    |num_tracks|avg_tempo         |avg_mfcc_energy   |std_rhythmic_complexity|avg_brightness    |avg_harmonic_density|
+---------+----------+------------------+------------------+----------------------+------------------+-------------------+
|pop      |100       |113.85883803911639|4.3668675117292315 |4.296904390620976      |1.0145147030871682|1239.1377810527329  |
|blues    |100       |123.87949190718429|-0.6201062277244623|4.685624017080769      |0.8798321935538955|601.6419940667206   |
|hiphop   |100       |110.20885930181859|3.773816669750768  |4.574478967875037      |1.0030175492772835|1151.0834937279783  |
|jazz     |99        |119.37343977271567|-5.294334864837605 |4.936284132646595      |0.8725799141399381|553.3636452407295   |
|country  |100       |120.65124481408883|0.14414083058038496|4.4447313663647385     |0.8900325176010897|667.2254004402726   |
|metal    |100       |123.84717744375791|5.191095756484617  |4.517912628998164      |1.1584910678154894|1254.6358880884725  |
|rock     |100       |120.19167310740562|1.5113764001299996 |4.7895477160549        |0.9873583989421267|871.7691659381758   |
|reggae   |100       |132.7436194595435 |0.2918145789961539 |5.575508307501024      |0.9353338996148348|905.570612814782    |
|disco    |100       |120.00554444511342|2.1085045316690763 |3.9301408372302946     |1.0397397660041572|1095.177341635532   |
|classical|100       |125.7456965589809 |-15.178653889158692|3.4972039388406935     |0.8837323296755996|360.0458398382722   |
+---------+----------+------------------+------------------+----------------------+------------------+-------------------+
```

3. Aplicarea metodelor ML

a. Random Forest Classifier

Am ales utilizarea acestui model de Machine Learning intrucat este eficient în problemele de clasificare. Acesta folosește un număr variat de arbori de decizie care favorizează un rezultat apropiat de realitate în momentul antrenarii pe mai multe date ce reprezinta caracteristici interdepedente, așa cum sunt cele legate de sunetele dintr-o creatie muzicala. In plus, gestionează eficient cantități mari de date. Folosind acest model am evaluat acuratețea predicției unei melodii la oricare dintre genurile muzicale prezente în setul de antrenament. O trasatura importanta a acestui model este aceea de “feature importance”, care modelează antrenarea în funcție de cele mai importante caracteristici.

```python
from pyspark.ml.feature import  VectorAssembler, StandardScaler
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
from pyspark.ml.classification import RandomForestClassifier
from pyspark.ml import Pipeline
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder

def print_confusion_matrix(predictions, model_name, label_to_genre):
```

```
    pred_labels = predictions.select("label", "prediction")
    confusion_matrix = pred_labels.groupBy("label",
"prediction").count().orderBy("label", "prediction")

    labels = sorted(label_to_genre.keys())
    confusion = confusion_matrix.collect()
    conf_dict = {(row['label'], row['prediction']): row['count'] for row in
confusion}

    print(f"\nMatrice de confuzie pentru {model_name}")
    print(" " * 18 + "\t".join([label_to_genre[l] for l in labels]))
    for actual in labels:
        row_counts = [str(conf_dict.get((actual, pred), 0)) for pred in
labels]
        print(f"{label_to_genre[actual]:18}:\t" + "\t".join(row_counts))


# Definim coloanele ce vor fi utilizate in procesul de antrenare
feature_cols = [
    "tempo", "spectral_centroid", "zero_crossing_rate", "rmse", "bandwidth",
"chroma",
    "mfcc_energy", "rhythmic_complexity", "harmonic_density",
"brightness_score", "percussive_ratio",
    "mfcc1", "mfcc2", "mfcc3", "mfcc4", "mfcc5", "mfcc6", "mfcc7", "mfcc8",
"mfcc9", "mfcc10", "mfcc11", "mfcc12", "mfcc13",
    "beat_strength", "tempo_variation", "onset_density"
]

# Impachetam datele pentru a putea fi folosite mai departe in procesul de
antrenare
assembler = VectorAssembler(inputCols=feature_cols, outputCol="features_vec")
# Normalizam datele
scaler = StandardScaler(inputCol="features_vec", outputCol="features",
withMean=True, withStd=True)

# Impartim datele intre cele de antrenare si de testare
train_data, test_data = df_indexed.randomSplit([0.8, 0.2], seed=42)

# Evaluator
evaluator = MulticlassClassificationEvaluator(labelCol="label",
predictionCol="prediction", metricName="accuracy")

# Initializarea modelului
rf = RandomForestClassifier(featuresCol="features", labelCol="label")
# Cream un pipeline pentru a automatiza tot procesul
rf_pipeline = Pipeline(stages=[assembler, scaler, rf])

# Construim grila de combinatii de hiperparametrii
rf_paramGrid = ParamGridBuilder() \
```

```
    .addGrid(rf_pipeline.getStages()[2].numTrees, [100, 150]) \
    .addGrid(rf_pipeline.getStages()[2].maxDepth, [5, 10]) \
    .addGrid(rf_pipeline.getStages()[2].maxBins, [32]) \
    .addGrid(rf_pipeline.getStages()[2].featureSubsetStrategy, ['auto',
'sqrt']) \
    .build()

# Configuram validarea incrucisata
rf_cv = CrossValidator(
    estimator=rf_pipeline,
    estimatorParamMaps=rf_paramGrid,
    evaluator=evaluator,
    numFolds=5
)

# Antrenam si evaluam modelul
rf_cv_model = rf_cv.fit(train_data)
rf_preds = rf_cv_model.transform(test_data)
rf_acc = evaluator.evaluate(rf_preds)

print(f" Random Forest Accuracy:       {rf_acc:.3f}")
print_confusion_matrix(rf_preds, "Random Forest", label_to_genre)
```

```
Matrice de confuzie pentru Random Forest
                  blues classical   country disco   hiphop  metal   pop reggae  rock    jazz
blues             : 16  0   0   0   0   0   0   0   1   0
classical         : 0   12  2   0   0   0   0   0   0   0
country           : 3   0   11  1   0   0   0   0   1   0
disco             : 0   0   1   6   2   0   1   0   1   0
hiphop            : 0   0   0   0   15  0   0   1   0   0
metal             : 0   0   0   0   0   17  0   0   0   0
pop               : 0   1   0   0   0   0   12  1   2   1
reggae            : 1   0   1   2   1   0   0   10  0   2
rock              : 1   0   3   1   1   2   0   0   7   0
jazz              : 0   3   2   0   0   0   0   0   1   15
```

b. Gradient Boosted Tree Classifier

De asemenea, o opțiune buna în problemele de clasificare, GBT își construiește arborii de decizie secvențial, invatand continuu de la predecesori, fiind o soluție eficienta în ceea ce privește clasificarea genurilor muzicale care prezinta particularitati asemenatoare si care pot fi cu usurinta confundate de către modelul RF. GBT accepta doar clase binare, prin urmare, folosind acest model, am ales sa evaluez capacitatea modelului de a distinge o melodie încadrată într-o anumită categorie.

Conform matricei de confuzie furnizate de modelul RF se poate observa cum categoria "disco" e cel mai slab clasificata, prin urmare, prin modelul GBT urmarim sa vedem performanța modelului de a clasifica o melodie apartinand acestui gen, fata de oricare alta.

```python
from pyspark.ml import Pipeline
from pyspark.ml.feature import VectorAssembler, StandardScaler
from pyspark.ml.classification import GBTClassifier
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder
from pyspark.sql.functions import col, when

# Cream array-ul de etichete
df_binary = df_indexed.withColumn("label_disco", when(col("genre") ==
"disco", 1).otherwise(0))

gtb_label_to_genre = {0: "not_disco", 1: "disco"}

assembler = VectorAssembler(inputCols=feature_cols, outputCol="features_vec")
scaler = StandardScaler(inputCol="features_vec", outputCol="features",
withMean=True, withStd=True)

gbt = GBTClassifier(featuresCol="features", labelCol="label_disco",
maxIter=100)
gbt_pipeline = Pipeline(stages=[assembler, scaler, gbt])

gbt_train_data, gbt_test_data = df_binary.randomSplit([0.8, 0.2], seed=42)

gbt_paramGrid = ParamGridBuilder() \
                .addGrid(gbt.maxDepth, [2]) \
                .addGrid(gbt.maxIter, [50]) \
                .build()

gb_evaluator = MulticlassClassificationEvaluator(labelCol="label_disco")

gbt_cv = CrossValidator(
    estimator=gbt_pipeline,
    estimatorParamMaps=gbt_paramGrid,
    evaluator=gb_evaluator,
    numFolds=5
)

gbt_cv_model = gbt_cv.fit(gbt_train_data)
gbt_predictions = gbt_cv_model.transform(gbt_test_data)
gbt_acc = gb_evaluator.evaluate(gbt_predictions)
```

```
print(f" GBTClassifier  Accuracy: {gbt_acc:.3f}")
```

4. Data Pipeline

Am utilizat cate un pipeline pentru fiecare dintre cele doua modele de ML, astfel:

```
# Pipeline RF

assembler = VectorAssembler(inputCols=feature_cols, outputCol="features_vec")
scaler = StandardScaler(inputCol="features_vec", outputCol="features",
withMean=True, withStd=True)

rf = RandomForestClassifier(featuresCol="features", labelCol="label")
rf_pipeline = Pipeline(stages=[assembler, scaler, rf])

# Pipeline GBT

assembler = VectorAssembler(inputCols=feature_cols, outputCol="features_vec")
scaler = StandardScaler(inputCol="features_vec", outputCol="features",
withMean=True, withStd=True)

gbt = GBTClassifier(featuresCol="features", labelCol="label_disco",
maxIter=100)
gbt_pipeline = Pipeline(stages=[assembler, scaler, gbt])
```

5. Optimizarea hiperparametrilor

Am folosit aceasta tehnica pentru fiecare dintre cele doua modele, astfel:

```
# RF

rf = RandomForestClassifier(featuresCol="features", labelCol="label")
rf_pipeline = Pipeline(stages=[assembler, scaler, rf])
evaluator = MulticlassClassificationEvaluator(labelCol="label",
predictionCol="prediction", metricName="accuracy")

rf_paramGrid = ParamGridBuilder() \
    .addGrid(rf_pipeline.getStages()[2].numTrees, [100, 150]) \
    .addGrid(rf_pipeline.getStages()[2].maxDepth, [5, 10]) \
    .addGrid(rf_pipeline.getStages()[2].maxBins, [32]) \
    .addGrid(rf_pipeline.getStages()[2].featureSubsetStrategy, ['auto',
'sqrt']) \
```

```
    .build()

rf_cv = CrossValidator(
    estimator=rf_pipeline,
    estimatorParamMaps=rf_paramGrid,
    evaluator=evaluator,
    numFolds=5
)

# GBT

assembler = VectorAssembler(inputCols=feature_cols, outputCol="features_vec")
scaler = StandardScaler(inputCol="features_vec", outputCol="features",
withMean=True, withStd=True)

gbt = GBTClassifier(featuresCol="features", labelCol="label_disco",
maxIter=100)
gbt_pipeline = Pipeline(stages=[assembler, scaler, gbt])

gbt_train_data, gbt_test_data = df_binary.randomSplit([0.8, 0.2], seed=42)

gbt_paramGrid = ParamGridBuilder() \
                .addGrid(gbt.maxDepth, [2]) \
                .addGrid(gbt.maxIter, [50]) \
                .build()

gb_evaluator = MulticlassClassificationEvaluator(labelCol="label_disco")

gbt_cv = CrossValidator(
    estimator=gbt_pipeline,
    estimatorParamMaps=gbt_paramGrid,
    evaluator=gb_evaluator,
    numFolds=5
)
```

6. Aplicarea unei metode DL

Pentru a rezolva aceasta cerinta am ales utilizarea metodei Keras Sequential, un model de retea neuronala, performanta pe problemele de clasificare și care are o capacitate buna de gestiune a datelor non-liniare, cu o relatie complexa intre caracteristici.

In primul rand, am exportat datele importante deja procesate de către modelele anterioare, creand noi .csv-uri folosite ca și input-uri pentru noul model KS.

```
import numpy as np

# Aleg modelul RF optim
final_df = rf_cv_model.bestModel.transform(df_indexed)
# Preiau datele necesare antrenarii
pandas_df = final_df.select("features", "label").toPandas()

X = np.array(pandas_df["features"].tolist())
y = pandas_df["label"].values

# Salvez in CSV
pd.DataFrame(X).to_csv("features.csv", index=False)
pd.DataFrame(y, columns=["label"]).to_csv("labels.csv", index=False)
```

Urmand ca aceste date sa fie folosite mai departe în noul model KS.

```
import pandas as pd
import tensorflow as tf
from tensorflow.keras.utils import to_categorical
from sklearn.model_selection import train_test_split
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau

X = pd.read_csv('/kaggle/input/dataset2/features.csv').values
y = pd.read_csv('/kaggle/input/dataset2/labels.csv').values.ravel()

# Pre-procesarea array-ului de etichete
y_cat = to_categorical(y)

# Impartirea datelor in date de test si de antrenament
X_train, X_test, y_train, y_test = train_test_split(X, y_cat, test_size=0.2,
random_state=42)

# Definirea modelului
model = tf.keras.Sequential([
    tf.keras.layers.BatchNormalization(input_shape=(X.shape[1],)),
    tf.keras.layers.Dense(256, activation='relu'),
    tf.keras.layers.Dropout(0.3),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dropout(0.3),
    tf.keras.layers.Dense(y_cat.shape[1], activation='softmax')
])

# Pentru compilarea modelului am adaugat optimizatorul clasic "adam" si
functia de pierdere "categorical_crossentropy", standard in problemele de
clasificare cu mai multe clase
model.compile(optimizer='adam',
```

```
              loss='categorical_crossentropy',
              metrics=['accuracy'])

model.fit(X_train, y_train, epochs=100, batch_size=32, validation_split=0.2)

loss, acc = model.evaluate(X_test, y_test)
print(f"Test Accuracy: {acc:.3f}")
```

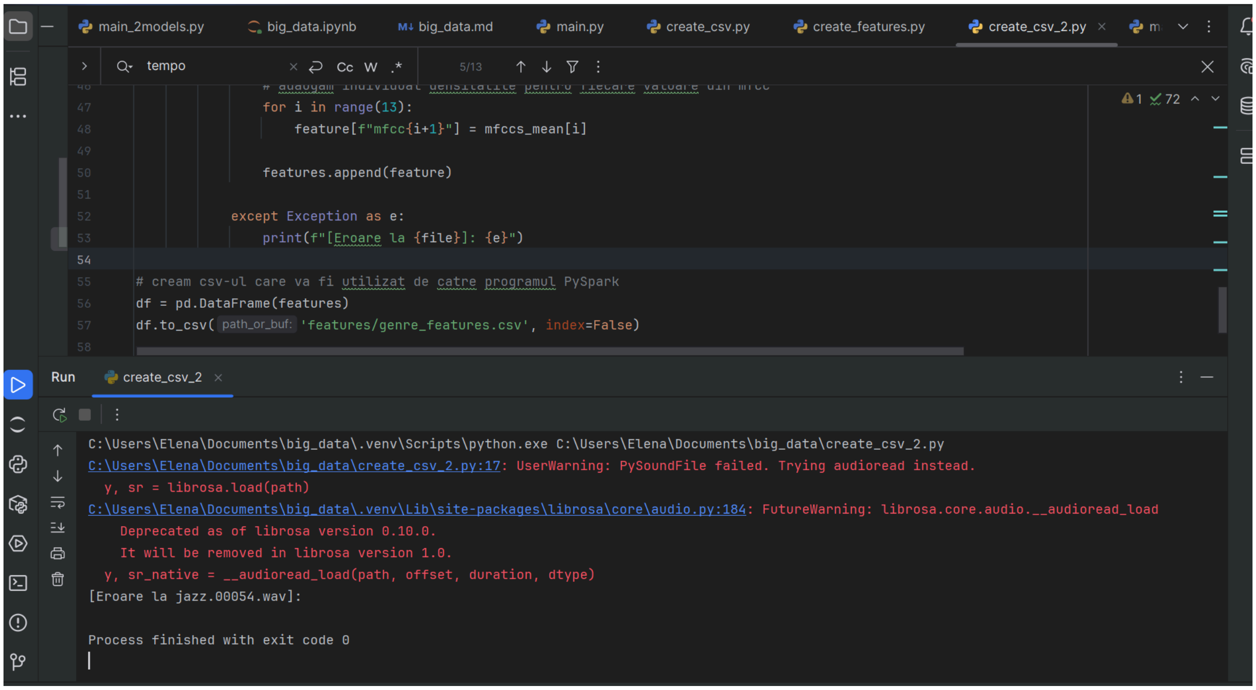
Sursa preprocesare

```
import librosa
import os
import numpy as np
import pandas as pd
from librosa.feature.rhythm import tempo as rhythm_tempo

base_path = "data/Data/genres_original"
genres = os.listdir(base_path)
features = []

for genre in genres:
    genre_path = os.path.join(base_path, genre)
    for file in os.listdir(genre_path):
        if file.endswith(".wav"):
            path = os.path.join(genre_path, file)
            try:
                y, sr = librosa.load(path)
                tempo_val = rhythm_tempo(y=y, sr=sr)[0] # variatia de ritm a
piesei
                spectral_centroid = librosa.feature.spectral_centroid(y=y,
sr=sr).mean() # inaltimea sunetului
                zero_crossing_rate =
librosa.feature.zero_crossing_rate(y).mean() # frecventa valorii 0 in
evolutia semnalului - ajuta la identificarea zgomotelor percutante
                rmse = librosa.feature.rms(y=y).mean() # Root Mean Square
Energy - energia generala a semnalului
                bandwidth = librosa.feature.spectral_bandwidth(y=y,
sr=sr).mean() # latimea benzii de frecventa - urmareste cat de mult variaza
semnalul
                chroma = librosa.feature.chroma_stft(y=y, sr=sr).mean() #
distributia semnalului pe notele muzicale - reflecta tonalitatea
                mfccs = librosa.feature.mfcc(y=y, sr=sr, n_mfcc=13) #
Mel-Frequency Cepstral Coefficients - reprezentare compacta a sunetului,
apropiata de perceptia umana
                mfccs_mean = mfccs.mean(axis=1) # calculam media valorilor
mfcc extrase
                beat_strength = np.mean(librosa.onset.onset_strength(y=y,
sr=sr)) # puterea batailor
```

```
                tempogram = librosa.feature.tempogram(y=y, sr=sr)
                tempo_variation = np.std(tempogram) # deviatia tempo-ului
                onsets = librosa.onset.onset_detect(y=y, sr=sr)
                onset_density = len(onsets) / librosa.get_duration(y=y,
sr=sr) # densitatea evenimentelor sonore

                feature = {
                    'filename': file,
                    'genre': genre,
                    'tempo': tempo_val,
                    'spectral_centroid': spectral_centroid,
                    'zero_crossing_rate': zero_crossing_rate,
                    'rmse': rmse,
                    'bandwidth': bandwidth,
                    'chroma': chroma,
                    'beat_strength': beat_strength,
                    'tempo_variation': tempo_variation,
                    'onset_density': onset_density
                }

                # adaugam individual densitatile pentru fiecare valoare din
mfcc
                for i in range(13):
                    feature[f"mfcc{i+1}"] = mfccs_mean[i]

                features.append(feature)

            except Exception as e:
                print(f"[Eroare la {file}]: {e}")

# cream csv-ul care va fi utilizat de catre programul PySpark
df = pd.DataFrame(features)
df.to_csv('features/genre_features.csv', index=False)
```

Georgescu Elena-Alexandra, Grupa 405

Output pre-procesare:



Sursa cerinte 2-5

```
import pandas as pd
from pyspark.sql import SparkSession
from pyspark.ml import Pipeline
from pyspark.ml.feature import StringIndexer, VectorAssembler, StandardScaler
from pyspark.ml.classification import RandomForestClassifier,
LogisticRegression, GBTClassifier
from pyspark.ml.evaluation import MulticlassClassificationEvaluator,
BinaryClassificationEvaluator
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder
from pyspark.sql.functions import expr, size, col, when, lit
from pyspark.sql.functions import avg, stddev, count


def print_confusion_matrix(predictions, model_name, label_to_genre):
    pred_labels = predictions.select("label", "prediction")
    confusion_matrix = pred_labels.groupBy("label",
"prediction").count().orderBy("label", "prediction")

    labels = sorted(label_to_genre.keys())
    confusion = confusion_matrix.collect()
```

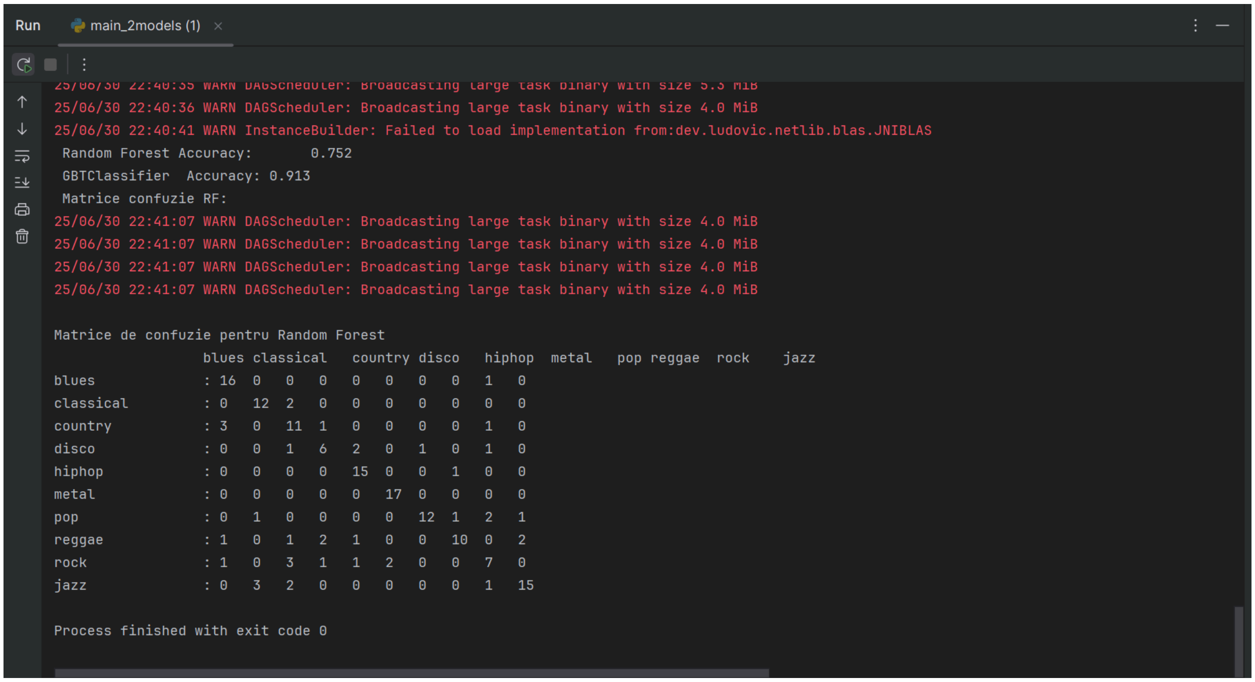
```
    conf_dict = {(row['label'], row['prediction']): row['count'] for row in
confusion}

    print(f"\nMatrice de confuzie pentru {model_name}")
    print(" " * 18 + "\t".join([label_to_genre[l] for l in labels]))
    for actual in labels:
        row_counts = [str(conf_dict.get((actual, pred), 0)) for pred in
labels]
        print(f"{label_to_genre[actual]:18}:\t" + "\t".join(row_counts))


spark = SparkSession.builder.appName("MusicClassification_CV").getOrCreate()

df = spark.read.csv("features/genre_features.csv", header=True,
inferSchema=True).dropna()
df.createOrReplaceTempView("audio_raw")

sql_df = spark.sql("""
    SELECT *,
           tempo * zero_crossing_rate AS rhythmic_complexity,
           chroma * spectral_centroid AS harmonic_density,
           CASE WHEN bandwidth != 0 THEN spectral_centroid / bandwidth ELSE
0.0 END AS brightness_score
    FROM audio_raw
""")

mfcc_cols = [f"mfcc{i}" for i in range(1, 14)]
sql_df = sql_df.withColumn("mfccs_array", expr(f"array({',
'.join(mfcc_cols)})"))
sql_df = sql_df.withColumn("mfcc_energy", expr("aggregate(mfccs_array, 0D,
(acc, x) -> acc + x) / size(mfccs_array)"))
sql_df = sql_df.withColumn("percussive_ratio", when(
    col("mfcc_energy") != 0, col("rhythmic_complexity") / col("mfcc_energy")
).otherwise(lit(0.0)))

df_proc = sql_df.drop("mfccs_array")

agg_df = df_proc.groupBy("genre").agg(
    count("*").alias("num_tracks"),
    avg("tempo").alias("avg_tempo"),
    avg("mfcc_energy").alias("avg_mfcc_energy"),
    stddev("rhythmic_complexity").alias("std_rhythmic_complexity"),
    avg("brightness_score").alias("avg_brightness"),
    avg("harmonic_density").alias("avg_harmonic_density")
)

print("Afisarea rezultatelor agregate")
agg_df.show(truncate=False)
```

```
indexer = StringIndexer(inputCol="genre", outputCol="label")
df_indexed = indexer.fit(df_proc).transform(df_proc)

label_to_genre = {i: genre for i, genre in
enumerate(indexer.fit(df_proc).labels)}

feature_cols = [
    "tempo", "spectral_centroid", "zero_crossing_rate", "rmse", "bandwidth",
"chroma",
    "mfcc_energy", "rhythmic_complexity", "harmonic_density",
"brightness_score", "percussive_ratio",
    "mfcc1", "mfcc2", "mfcc3", "mfcc4", "mfcc5", "mfcc6", "mfcc7", "mfcc8",
"mfcc9", "mfcc10", "mfcc11", "mfcc12", "mfcc13",
    "beat_strength", "tempo_variation", "onset_density"
]

assembler = VectorAssembler(inputCols=feature_cols, outputCol="features_vec")
scaler = StandardScaler(inputCol="features_vec", outputCol="features",
withMean=True, withStd=True)

train_data, test_data = df_indexed.randomSplit([0.8, 0.2], seed=42)

evaluator = MulticlassClassificationEvaluator(labelCol="label",
predictionCol="prediction", metricName="accuracy")

rf = RandomForestClassifier(featuresCol="features", labelCol="label")
rf_pipeline = Pipeline(stages=[assembler, scaler, rf])

rf_paramGrid = ParamGridBuilder() \
    .addGrid(rf_pipeline.getStages()[2].numTrees, [100, 150]) \
    .addGrid(rf_pipeline.getStages()[2].maxDepth, [5, 10]) \
    .addGrid(rf_pipeline.getStages()[2].maxBins, [32]) \
    .addGrid(rf_pipeline.getStages()[2].featureSubsetStrategy, ['auto',
'sqrt']) \
    .build()

rf_cv = CrossValidator(
    estimator=rf_pipeline,
    estimatorParamMaps=rf_paramGrid,
    evaluator=evaluator,
    numFolds=5
)

rf_cv_model = rf_cv.fit(train_data)
rf_preds = rf_cv_model.transform(test_data)
rf_acc = evaluator.evaluate(rf_preds)

df_binary = df_indexed.withColumn("label_disco", when(col("genre") ==
"disco", 1).otherwise(0))
```

```
gtb_label_to_genre = {0: "not_disco", 1: "disco"}

assembler = VectorAssembler(inputCols=feature_cols, outputCol="features_vec")
scaler = StandardScaler(inputCol="features_vec", outputCol="features",
withMean=True, withStd=True)

gbt = GBTClassifier(featuresCol="features", labelCol="label_disco",
maxIter=100)
gbt_pipeline = Pipeline(stages=[assembler, scaler, gbt])

gbt_train_data, gbt_test_data = df_binary.randomSplit([0.8, 0.2], seed=42)

gbt_paramGrid = ParamGridBuilder() \
                .addGrid(gbt.maxDepth, [2]) \
                .addGrid(gbt.maxIter, [50]) \
                .build()

gb_evaluator = MulticlassClassificationEvaluator(labelCol="label_disco")

gbt_cv = CrossValidator(
    estimator=gbt_pipeline,
    estimatorParamMaps=gbt_paramGrid,
    evaluator=gb_evaluator,
    numFolds=5
)

gbt_cv_model = gbt_cv.fit(gbt_train_data)
gbt_predictions = gbt_cv_model.transform(gbt_test_data)
gbt_acc = gb_evaluator.evaluate(gbt_predictions)

print(f" Random Forest Accuracy (CV):       {rf_acc:.3f}")
print(f" GBTClassifier  Accuracy (CV): {gbt_acc:.3f}")

print(" Matrice confuzie RF:")
print_confusion_matrix(rf_preds, "Random Forest", label_to_genre)

# print("Matrice confuzie GBT:")
# print_confusion_matrix(gbt_predictions, "GBTClassifier",
gtb_label_to_genre)

final_df = rf_cv_model.bestModel.transform(df_indexed)
pandas_df = final_df.select("features", "label").toPandas()

import numpy as np

X = np.array(pandas_df["features"].tolist())  # features = list de vectori
y = pandas_df["label"].values
```

```
pd.DataFrame(X).to_csv("features.csv", index=False)
pd.DataFrame(y, columns=["label"]).to_csv("labels.csv", index=False)
```

Output-ul execuției celor doua modele de ML:



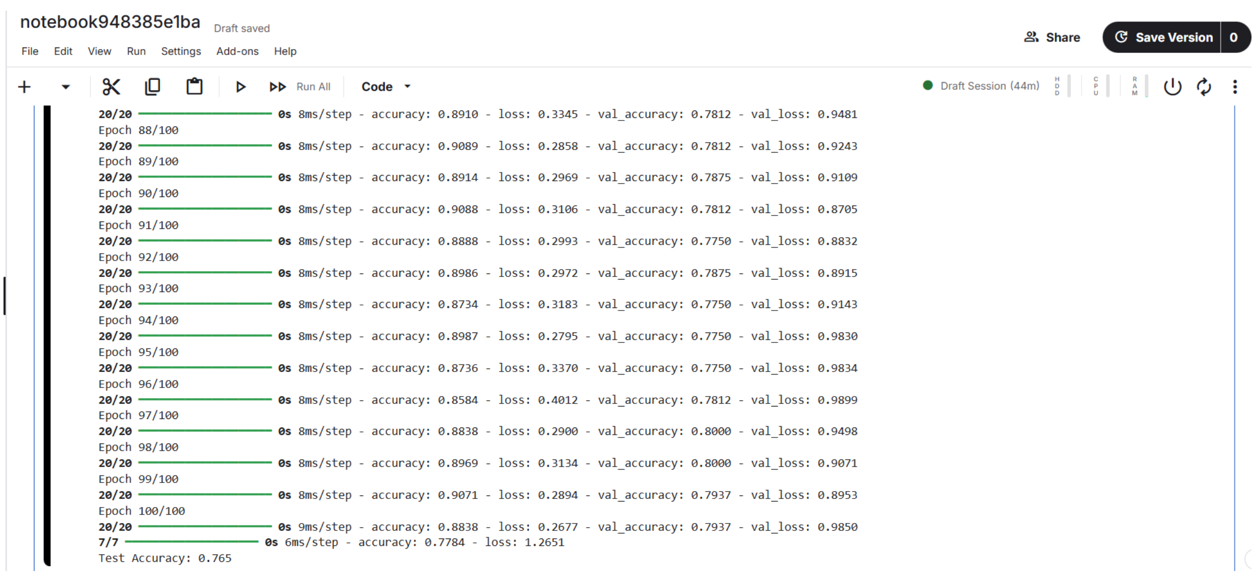
```
Run    main_2models (1)
25/06/30 22:40:35 WARN DAGScheduler: Broadcasting large task binary with size 5.3 MiB
25/06/30 22:40:36 WARN DAGScheduler: Broadcasting large task binary with size 4.0 MiB
25/06/30 22:40:41 WARN InstanceBuilder: Failed to load implementation from:dev.ludovic.netlib.blas.JNIBLAS
 Random Forest Accuracy:       0.752
 GBTClassifier  Accuracy: 0.913
 Matrice confuzie RF:
25/06/30 22:41:07 WARN DAGScheduler: Broadcasting large task binary with size 4.0 MiB
25/06/30 22:41:07 WARN DAGScheduler: Broadcasting large task binary with size 4.0 MiB
25/06/30 22:41:07 WARN DAGScheduler: Broadcasting large task binary with size 4.0 MiB
25/06/30 22:41:07 WARN DAGScheduler: Broadcasting large task binary with size 4.0 MiB

Matrice de confuzie pentru Random Forest
                  blues classical   country disco   hiphop  metal   pop reggae  rock    jazz
blues             : 16  0   0   0   0   0   0   0   1   0
classical         : 0   12  2   0   0   0   0   0   0   0
country           : 3   0   11  1   0   0   0   0   1   0
disco             : 0   0   1   6   2   0   1   0   1   0
hiphop            : 0   0   0   0   15  0   0   1   0   0
metal             : 0   0   0   0   0   17  0   0   0   0
pop               : 0   1   0   0   0   0   12  1   2   1
reggae            : 1   0   1   2   1   0   0   10  0   2
rock              : 1   0   3   1   1   2   0   0   7   0
jazz              : 0   3   2   0   0   0   0   0   1   15

Process finished with exit code 0
```

Sursa cerinta 6

```
import pandas as pd
import tensorflow as tf
from tensorflow.keras.utils import to_categorical
from sklearn.model_selection import train_test_split
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau

X = pd.read_csv('/kaggle/input/dataset2/features.csv').values
y = pd.read_csv('/kaggle/input/dataset2/labels.csv').values.ravel()

y_cat = to_categorical(y)

X_train, X_test, y_train, y_test = train_test_split(X, y_cat, test_size=0.2,
random_state=42)

model = tf.keras.Sequential([
```

```
    tf.keras.layers.BatchNormalization(input_shape=(X.shape[1],)),
    tf.keras.layers.Dense(256, activation='relu'),
    tf.keras.layers.Dropout(0.3),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dropout(0.3),
    tf.keras.layers.Dense(y_cat.shape[1], activation='softmax')
])

model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

model.fit(X_train, y_train, epochs=100, batch_size=32, validation_split=0.2)

loss, acc = model.evaluate(X_test, y_test)
print(f"Test Accuracy: {acc:.3f}")
```

Output-ul cerintei 6:



Concluzii

În urma acestor rezultate observam ca RF si KS au performante similare, în timp ce GBT capturează eficient diferențele subtile ale categoriei slab clasificata de către RF, reușind sa returneze o acuratețe notabilă.

Georgescu Elena-Alexandra, Grupa 405

Cuprins