PES University, Bangalore
(Established under Karnataka Act No. 16 of 2013)
Department of Computer Science and Engineering

UE21CS342BA2: Algorithms for Information Retrieval and Intelligence Web

Miniproject Report
Project Title: A Comparative Study of ML Models in a Car Music Recommender

Project Team Members:

1. Ria R Kulkarni – PES1UG21CS487
2. Riya Bansal – PES1UG21CS929

PES UNIVERSITY

TABLE OF CONTENTS

PROBLEM STATEMENT

The project aims to develop a sophisticated context-aware car music recommendation system, integrating various recommendation models to cater to individual user preferences and real-time environmental contexts. We start with an in-depth exploratory analysis of the music dataset, employing techniques such as data cleaning, feature engineering, and clustering to uncover meaningful insights about user segments and their music consumption patterns.

Subsequently, a comparative study of different recommendation models is conducted, including deep learning approaches using PyTorch and Spotlight framework, collaborative filtering hybrid models with LightFM algorithm. This is contrasted against classic machine learning techniques such as GradientBoostingClassifier and HistGradientBoostingClassifier; and an AutoML approach for automated model selection.

This comparative analysis evaluates the performance of each model in terms of metrics such as RMSE, precision at K, recall at K, and F1-score, providing valuable insights into their effectiveness in providing personalized music recommendations across diverse contexts and user scenarios such as mood, weather, driving conditions, type of road and so on.

CODE

1) Deep Recommender model:

```
## Import Packages
!pip install git+https://github.com/maciejkula/spotlight.git@master#egg=spotlight
import numpy as np
import pandas as pd
from spotlight.cross_validation import user_based_train_test_split
from spotlight.evaluation import *
from spotlight.interactions import Interactions
from spotlight.factorization.implicit import ImplicitFactorizationModel
from spotlight.sequence.implicit import ImplicitSequenceModel
from spotlight.factorization.explicit import ExplicitFactorizationModel
from spotlight.factorization.representations import BilinearNet
from spotlight.layers import BloomEmbedding
import torch
from torch.optim import sparse_adam
from spotlight.evaluation import mrr_score
from scipy.stats import describe
## Reading Data
dataset = pd.read_csv('dataset_aggr.csv')
user_ids = dataset.UserID.values
item_ids = dataset.ItemID.values
ratings = dataset.avg_rating.values
interactions = Interactions(user_ids = np.array(user_ids, dtype=np.int32),
                            item_ids = np.array(item_ids, dtype=np.int32),
                            ratings = np.array(ratings, dtype=np.float32))
train, test = user_based_train_test_split(interactions, test_percentage=0.20)
### Check train-test Proportions
train, test
## Train "implicit factorization model":
### Define parameters:
model = ImplicitFactorizationModel(loss='adaptive_hinge',
                                   embedding_dim=128,
                                   n_iter=100,
                                   batch_size=32,
                                   learning_rate=0.005,
                                   l2=1e-6,
                                   optimizer_func=sparse_adam.SparseAdam,
                                   sparse=True,
                                   num_negative_samples=10)
### Fit model
model.fit(train, verbose=True)
### Average MRR:
mrr = mrr_score(model, test)
avg_mrr = mrr.mean()
```

```
f'Avg MRR score (for test data): {avg_mrr}'
### RMSE score
rmse = rmse_score(model, test)
f'RMSE score (for test data): {rmse}'
### Precision-Recall at k:
#### On the train set,
precision, recall = precision_recall_score(model, train)
print(f'Precision per user: {precision}')
describe(precision)
print(f'Recall per user: {recall}')
describe(recall)
#### On the test set
precision, recall = precision_recall_score(model, test)
print(f'Precision per user: {precision}')
describe(precision)
print(f'Recall per user: {recall}')
describe(recall)
## Train "explicit factorization model":
model = ExplicitFactorizationModel(loss='poisson',
                                   embedding_dim=128,
                                   n_iter=150,
                                   batch_size=32,
                                   learning_rate=0.005,
                                   l2=1e-6,
                                   optimizer_func=sparse_adam.SparseAdam,
                                   sparse=True)
### Fit the model:
model.fit(train, verbose=True)
### Average MRR:
mrr = mrr_score(model, test)
avg_mrr = mrr.mean()
f'Avg MRR score (for test data): {avg_mrr}'
### RMSE score:
rmse = rmse_score(model, test)
f'RMSE score (for test data): {rmse}'
### Precision-Recall at k:
precision, recall = precision_recall_score(model, test)
print(f'Precision per user: {precision}')
print(describe(precision))
print(f'Recall per user: {recall}')
print(describe(recall))
## Model Bloom Embeddings:
compression_ratio = 1.5
user_embeddings = BloomEmbedding(interactions.num_users, 32,
                                 compression_ratio=compression_ratio,
                                 num_hash_functions=2)
```

```
item_embeddings = BloomEmbedding(interactions.num_items, 32,
                                 compression_ratio=compression_ratio,
                                 num_hash_functions=2)

network = BilinearNet(interactions.num_users,
                      interactions.num_items,
                      user_embedding_layer=user_embeddings,
                      item_embedding_layer=item_embeddings)

model = ExplicitFactorizationModel(loss='poisson',
                                   n_iter=150,
                                   batch_size=32,
                                   learning_rate=1e-2,
                                   l2=1e-6,
                                   representation=network,
                                   sparse=True,
                                   use_cuda=False)
model.fit(train)
mrr = mrr_score(model, test, train=train)
f'Avg MRR score: {np.mean(mrr)}'
rmse = rmse_score(model, test)
f'RMSE score (for test data): {rmse}'
precision, recall = precision_recall_score(model, test, train=train)
print(f'Precision per user: {precision}')
print(describe(precision))
print(f'Recall per user: {recall}')
print(describe(recall))
## Unaggregated set, explicit model (with ratings):
dataset = pd.read_csv('dataset.csv')
user_ids = dataset.UserID.values
item_ids = dataset.ItemID.values

ratings = dataset.Rating.values

interactions = Interactions(user_ids = np.array(user_ids, dtype=np.int32),
                            item_ids = np.array(item_ids, dtype=np.int32),
                            ratings = np.array(ratings, dtype=np.float32))

train, test = user_based_train_test_split(interactions, test_percentage=0.20)
train, test
model = ExplicitFactorizationModel(loss='poisson',
                                   embedding_dim=256,
                                   n_iter=200,
                                   batch_size=32,
                                   learning_rate=0.005,
                                   l2=1e-6)
model.fit(train)
```

```
mrr = mrr_score(model, test, train=train)
f'Avg MRR score: {np.mean(mrr)}'
rmse = rmse_score(model, test)
f'RMSE score (for test data): {rmse}'
precision, recall = precision_recall_score(model, test, train=train)
print(f'Precision per user: {precision}')
print(describe(precision))
print(f'Recall per user: {recall}')
print(describe(recall))
```

2) Hybrid and Collaborative Filtering:

```
## Import packages
Importing necessary packages for building the recommendation system.
from lightfm import LightFM
from lightfm.data import Dataset
from lightfm.evaluation import auc_score, precision_at_k, recall_at_k, reciprocal_rank
from lightfm.cross_validation import random_train_test_split
import numpy as np
import pandas as pd
from pandasql import sqldf
pysqldf = lambda q: sqldf(q, globals())
from scipy.stats import describe
from sklearn.preprocessing import MinMaxScaler
## Agregated dataset
In this step, we load the dataset containing aggregated user data and music category
information. The code then aggregates user features using SQL queries and scales them
using MinMaxScaler to ensure all features are on the same scale.
data = pd.read_csv('dataset_aggr.csv')
music_category = pd.read_csv('music_type.csv')
item_features = music_category
### Profiling users
# This will work as a user profile

user_features = pysqldf('''select UserID, avg(number_of_unique_songs) as X1,
avg(number_of_unique_genres) as X2,
                           avg(main_genre_dominance) as X3, avg(genre_ratio) as X4,
                           sum(no_stimulus_points) as X5, sum(stimulus_points) as X6,
                           sum(driving_style_relaxed_driving) as X7,
sum(driving_style_sport_driving) as X8,
                           sum(landscape_coast_line) as X9, sum(landscape_country_side) as
X10,
                           sum(landscape_mountains) as X11,
                           sum(landscape_urban) as X12, sum(mood_active) as X13,
sum(mood_happy) as X14,
                           sum(mood_lazy) as X15, sum(mood_sad) as X16,
                           sum(natural_phenomena_afternoon) as X17,
sum(natural_phenomena_day_time) as X18,
                           sum(natural_phenomena_morning) as X19,
sum(natural_phenomena_night) as X20,
                           sum(road_type_city) as X21, sum(road_type_highway) as X22,
                           sum(road_type_serpentine) as X23,
                           sum(sleepiness_awake) as X24, sum(sleepiness_sleepy) as X25,
                           sum(traffic_conditions_free_road) as X26,
                           sum(traffic_conditions_lots_of_cars) as X27,
sum(traffic_conditions_traffic_jam) as X28,
                           sum(weather_cloudy) as X29, sum(weather_rainy) as X30,
                           sum(weather_snowing) as X31, sum(weather_sunny) as X32
                           from data
                           group by UserID''')
scale = MinMaxScaler()
_user_ids = user_features.UserID.copy()
user_features = pd.DataFrame(scale.fit_transform(user_features.values),
                             columns=user_features.columns, index=user_features.index)
user_features.UserID = _user_ids
user_features.head()
The code initializes a LightFM Dataset object and fits it with unique user and item IDs
from the dataset. User and item features are then built using the Dataset object, which
involves converting categorical features into a format suitable for model training.
Interactions between users and items, along with any associated weights (e.g., ratings),
are also built.
dataset = Dataset()
dataset.fit(data.UserID.unique(),
            data.ItemID.unique())
num_users, num_items = dataset.interactions_shape()
print('Num users: {}, num_items {}.'.format(num_users, num_items))
dataset.fit_partial(users=[x['UserID'] for idx, x in user_features.iterrows()],
                    items=[x['ItemID'] for idx, x in item_features.iterrows()],
                    # user_features=[[f'X{i}_{x[f"X{i}"]}' for i in range(1,32)] for idx,
x in user_features.iterrows()],
                    user_features=[f'X{i}' for i in range(1,32)],
                    item_features=[x['category_name'] for idx, x in
item_features.iterrows()])
                    # item_features=['category_name', ])
user_features = dataset.build_user_features([( x['UserID'], {f'X{i}': x[f'X{i}'] for i in
range(1,32)} )
                                             for idx, x in user_features.iterrows()])

print(repr(user_features))
item_features = dataset.build_item_features([( x['ItemID'], [x['category_name'], ] )
                                             for idx, x in item_features.iterrows()])

print(repr(item_features))
```

```
(interactions, weights) = dataset.build_interactions(((x['UserID'], x['ItemID'],
x['avg_rating'])
                                                       for idx, x in data.iterrows()))

print(repr(interactions))
### Split data
This step involves splitting the data into training and test sets using
random_train_test_split. Hyperparameters for the LightFM model are defined, such as the
number of threads, components, epochs, and regularization terms. The LightFM model is then
initialized with these hyperparameters and trained using the fit method with the training
data, along with item and user features.
train, test = random_train_test_split(interactions, test_percentage=0.2)
train.shape, test.shape
NUM_THREADS = 4
NUM_COMPONENTS = 2000
NUM_EPOCHS = 500
ITEM_ALPHA = 2e-6
USER_ALPHA = 1e-6
# Define a new model instance
model = LightFM(loss='warp',
                learning_schedule='adadelta',
                learning_rate=0.05,
                rho=0.75,
                epsilon=1e-5,
                user_alpha=USER_ALPHA,
                max_sampled=20,
                item_alpha=ITEM_ALPHA,
                no_components=NUM_COMPONENTS)

# Fit the hybrid model. Note that this time, we pass
# in the item features matrix.
model = model.fit(train,
                  item_features=item_features,
                  user_features=user_features,
                  # sample_weight=weights,
                  # there is no way to get weights after randomsplit
                  epochs=NUM_EPOCHS,
                  num_threads=NUM_THREADS,
                  verbose=False)
train_auc = auc_score(model,
                      train,
                      item_features=item_features,
                      user_features=user_features,
                      num_threads=NUM_THREADS).mean()
print('Hybrid training set AUC: %s' % train_auc)
test_auc = np.nanmean(auc_score(model,
                                test,
                                train_interactions=train,
                                item_features=item_features,
                                user_features=user_features,
                                num_threads=NUM_THREADS))
print('Hybrid test set AUC: %s' % test_auc)
reciprocal = reciprocal_rank(model,
                             test,
                             train_interactions=train,
                             item_features=item_features,
                             user_features=user_features,
                             num_threads=NUM_THREADS).mean()
print('Reciprocal rank for test set: %s' % reciprocal)
found_precision = precision_at_k(model,
                                 test,
                                 train_interactions=train,
                                 item_features=item_features,
                                 user_features=user_features,
                                 num_threads=NUM_THREADS)

print(f'Exact precision (per user): {found_precision}\n')
test_precision = np.nanmean(found_precision)
print(f'Hybrid test set Precision at K: {describe(found_precision)}\n')
print('Hybrid test set mean Precision at K: %s' % test_precision)
test_recall = np.nanmean(recall_at_k(model,
                                     test,
                                     train_interactions=train,
                                     item_features=item_features,
                                     user_features=user_features,
                                     num_threads=NUM_THREADS))
print('Hybrid test set Recall at K: %s' % test_recall)
```

3) Balanced Machine Learning Models:

```
## Prepare Packages
import numpy as np
import pandas as pd
import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)
warnings.simplefilter(action='ignore', category=UserWarning)
import multiprocessing
from sklearn.preprocessing import scale
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import MinMaxScaler
from sklearn.decomposition import PCA
from sklearn.impute import SimpleImputer
from sklearn.manifold import TSNE
from sklearn.manifold import Isomap
from sklearn.decomposition import SparsePCA
from sklearn.decomposition import TruncatedSVD
!pip install eli5
!pip install --upgrade scikit-learn
!pip install --upgrade eli5
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from time import time
from scipy.stats import randint as sp_randint
from sklearn.feature_selection import VarianceThreshold
from sklearn.pipeline import Pipeline
from sklearn import model_selection
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import RandomizedSearchCV
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.experimental import enable_hist_gradient_boosting
from sklearn.ensemble import HistGradientBoostingClassifier
from sklearn.ensemble import ExtraTreesClassifier
from sklearn.ensemble import AdaBoostClassifier
from sklearn.linear_model import PassiveAggressiveClassifier
from sklearn.linear_model import SGDClassifier
from sklearn.neural_network import MLPClassifier
from sklearn.neural_network import BernoulliRBM
from sklearn.svm import SVC
import pylab as plt
import eli5
data = pd.read_csv('dataset_aggr.csv')
### List available columns
data.columns
## Prepare Data:
### Set label column:
data['label'] = 1
### Add negative samples:

We are going to continually sample random N-items from our groups, and keep items if they
are not positive. This will continue until we have enough negative samples.
dataset = data.copy()

for idx, row in data.iterrows():
    # new_row = {'UserID': row['UserID'], }
    candidate = data[(data.ItemID != row['ItemID']) & (data.UserID !=
row['UserID'])].sample(n=1)
    candidate = candidate.iloc[0].to_dict()
    left_k = ['UserID', 'number_of_unique_songs',
       'number_of_unique_genres', 'genre_ratio', 'main_genre_dominance',
       'no_stimulus_points', 'stimulus_points',
       'driving_style_relaxed_driving', 'driving_style_sport_driving',
       'landscape_coast_line', 'landscape_country_side', 'landscape_mountains',
       'landscape_urban', 'mood_active', 'mood_happy', 'mood_lazy', 'mood_sad',
       'natural_phenomena_afternoon', 'natural_phenomena_day_time',
       'natural_phenomena_morning', 'natural_phenomena_night',
       'road_type_city', 'road_type_highway', 'road_type_serpentine',
       'sleepiness_awake', 'sleepiness_sleepy', 'traffic_conditions_free_road',
       'traffic_conditions_lots_of_cars', 'traffic_conditions_traffic_jam',
       'weather_cloudy', 'weather_rainy', 'weather_snowing', 'weather_sunny',
       'dominant_genre_blues',
       'dominant_genre_pop', 'dominant_genre_rock', 'second_dominant_blues',
       'second_dominant_blues_classical_disco',
       'second_dominant_blues_classicalsecond_dominant_hh',
       'second_dominant_blues_disco_rock', 'second_dominant_blues_hh',
       'second_dominant_blues_metal_reggae', 'second_dominant_classical',
       'second_dominant_classical_country',
       'second_dominant_classical_country_disco_hh',
       'second_dominant_classical_country_disco_hh_jazz_metal_rock',
       'second_dominant_classical_disco',
       'second_dominant_classical_disco_reggae',
       'second_dominant_classical_hh_rock', 'second_dominant_country',
       'second_dominant_country_disco_rock',
       'second_dominant_country_jazz_rock', 'second_dominant_disco',
       'second_dominant_disco_hh', 'second_dominant_jazz',
       'second_dominant_metal']

    right_k = ['ItemID', 'category_name_blues', 'category_name_classical',
       'category_name_country', 'category_name_disco', 'category_name_hip_hop',
       'category_name_jazz', 'category_name_metal', 'category_name_pop',
       'category_name_reggae', 'category_name_rock']

    left = { k: row[k] for k in left_k }
    right = { k: candidate[k] for k in right_k }

    new_row = {**left, **right}
    new_row['avg_rating'] = None
    new_row['label'] = 0

    dataset = dataset.append(new_row, ignore_index=True)
### Investigate First Rows:
dataset.sort_values(by='UserID').head(20)
dataset.label.describe()
### Basics Statistics of All Columns:
dataset.describe()
```

```
### Check for NA Values:
percent_missing = dataset.isnull().sum() * 100 / len(dataset)
missing_value_df = pd.DataFrame({'column_name': dataset.columns,
                                 'percent_missing': percent_missing})
missing_value_df
## We're going to remove average rating from the set, otherwise algorithm may converge to
it unneccesarily.
## Prepare train-test sets:
X = dataset.drop(['UserID', 'ItemID', 'label', 'avg_rating'], axis=1).values
Y = dataset.label.values

X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.20, random_state=42,
stratify=Y)
imp = SimpleImputer(missing_values=np.nan, strategy='mean')
imp_X_train = imp.fit_transform(X_train)
imp_X_test = imp.transform(X_test)
## First Model - Reconnaissance:
clf = GradientBoostingClassifier()
clf.fit(imp_X_train, y_train)
y_pred = clf.predict(imp_X_test)
print(classification_report(y_test, y_pred))
explaination
## Compare Different Algorithms:
### Check Recall:
# prepare models
models = []
# initial_pca = 3
n_splits = 6  # k=6 folds

# hyper-parameter tuning will be done later
models.append(('ETC', Pipeline([('prepare', StandardScaler()), ('clf',
ExtraTreesClassifier(n_jobs=-1))])))
models.append(('GBC', Pipeline([('prepare', StandardScaler()), ('clf',
GradientBoostingClassifier())])))
models.append(('ADA', Pipeline([('prepare', StandardScaler()), ('clf',
AdaBoostClassifier())])))
models.append(('RFC', Pipeline([('prepare', StandardScaler()), ('clf',
RandomForestClassifier(n_jobs=-1))])))
models.append(('PAC', Pipeline([('prepare', StandardScaler()), ('clf',
PassiveAggressiveClassifier(n_jobs=-1))])))
models.append(('SGD', Pipeline([('prepare', StandardScaler()), ('clf',
SGDClassifier(n_jobs=-1))])))
models.append(('MLP', Pipeline([('prepare', StandardScaler()), ('clf',
MLPClassifier())])))
models.append(('SVC', Pipeline([('prepare', StandardScaler()), ('clf', SVC())])))
models.append(('HGBC', Pipeline([('prepare', StandardScaler()), ('clf',
HistGradientBoostingClassifier())])))
```

```
# evaluate each model in turn
results = []
names = []
scoring = 'recall'
left = len(models)

for name, model in models:
    left -= 1
    print('Testing the {} algorithm, there are {} methods left'.format(name, left))
    kfold = model_selection.RepeatedStratifiedKFold(n_splits=n_splits, random_state=42)
    cv_results = model_selection.cross_val_score(model, X, Y,
                                                 cv=kfold,
                                                 n_jobs=-1,
                                                 verbose=0,
                                                 scoring=scoring)
    results.append(cv_results)
    names.append(name)
    msg = "%s: %f (%f)" % (name, cv_results.mean(), cv_results.std())
    print(msg)
# boxplot algorithm comparison
fig = plt.figure()
fig.suptitle('Algorithm comparison - [recall] score')
ax = fig.add_subplot(111)
plt.boxplot(results)
ax.set_xticklabels(names)
plt.show()
### Precision:
# evaluate each model in turn
results = []
names = []
scoring = 'precision'

left = len(models)

for name, model in models:
    left -= 1
    print('Testing the {} algorithm, there are {} methods left'.format(name, left))
    kfold = model_selection.RepeatedStratifiedKFold(n_splits=n_splits, random_state=42)
    cv_results = model_selection.cross_val_score(model, X, Y,
                                                 cv=kfold,
                                                 n_jobs=-1,
                                                 verbose=0,
                                                 scoring=scoring)
    results.append(cv_results)
    names.append(name)
    msg = "%s: %f (%f)" % (name, cv_results.mean(), cv_results.std())
    print(msg)
# boxplot algorithm comparison
fig = plt.figure()
fig.suptitle('Algorithm comparison - [precision] score')
ax = fig.add_subplot(111)
plt.boxplot(results)
ax.set_xticklabels(names)
plt.show()
## Hyper-parameter Tuning:
def report(results, n_top=3):
    for i in range(1, n_top + 1):
        candidates = np.flatnonzero(results['rank_test_score'] == i)
        for candidate in candidates:
            print("Model with rank: {0}".format(i))
            print("Mean validation score: {0:.3f} (std: {1:.3f})".format(
                  results['mean_test_score'][candidate],
                  results['std_test_score'][candidate]))
            print("Parameters: {0}".format(results['params'][candidate]))
            print("")
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.pipeline import Pipeline
from sklearn.model_selection import RandomizedSearchCV, RepeatedStratifiedKFold
import numpy as np
import multiprocessing
from time import time

param_dist = {
    'scale': [StandardScaler(), StandardScaler(with_std=False),
SimpleImputer(missing_values=np.nan, strategy='mean'), MinMaxScaler()],
    'clf': [GradientBoostingClassifier()],
    'clf__loss': ['exponential', 'deviance'],
    'clf__learning_rate': [0.01, 0.005, 0.02, 0.05],
    'clf__n_estimators': [100, 50, 200, 150, 250],
    'clf__subsample': [0.8, 1.0, 0.9, 0.7],
    'clf__criterion': ['friedman_mse', 'squared_error'],
    'clf__min_samples_leaf': [1, 2, 3],
    'clf__max_depth': [2, 3, 4],
    'clf__max_features': ['sqrt', 'log2', None],
    'clf__min_samples_split': [2, 3, 4]
}

pipe = Pipeline(steps=[('scale', None), ('clf', None)])

# Optimizing through F1
scoring_tune='f1'
```

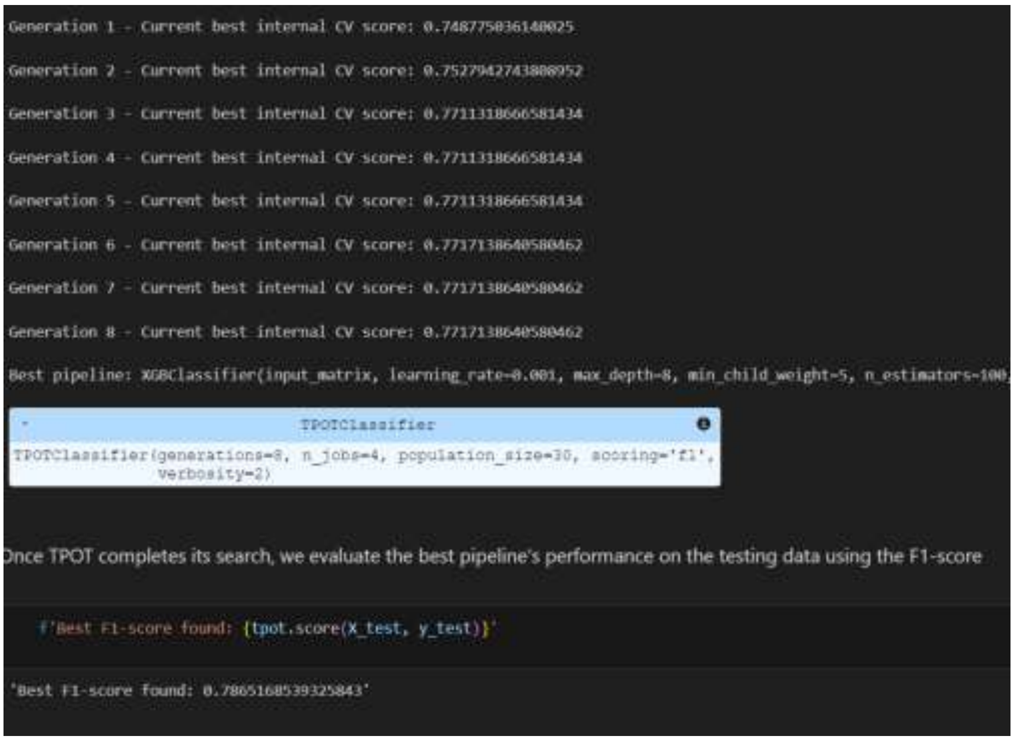
```
# Run randomized search
n_iter_search = 250
random_search = RandomizedSearchCV(pipe,
                                   param_distributions=param_dist,
                                   scoring=scoring_tune,
                                   n_iter=n_iter_search,
                                   verbose=True,
                                   cv=RepeatedStratifiedKFold(n_splits=12, n_repeats=1),
                                   n_jobs=multiprocessing.cpu_count())

start = time()
random_search.fit(X, Y)

print("RandomizedSearchCV took %.2f seconds for %d candidates"
      " parameter settings." % ((time() - start), n_iter_search))

best = random_search.best_estimator_
report(random_search.cv_results_)
```

4) AutoML Approach:

```
## Prepare packages
Importing necessary libraries: NumPy and Pandas for data manipulation, and TPOTClassifier
from the TPOT library for automated machine learning.
import numpy as np
import pandas as pd
import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)
warnings.simplefilter(action='ignore', category=UserWarning)
from sklearn.model_selection import train_test_split
from tpot import TPOTClassifier
## Read data
Loading the dataset from a CSV file named dataset_aggr.csv into a pandas DataFrame called
data.
data = pd.read_csv('dataset_aggr.csv')
Adding a new column named label with all values set to 1 to indicate positive labels.
data['label'] = 1
We iterate over each row in the dataset (data.iterrows()).
For each row, we randomly select another row from the dataset (excluding the current row)
to serve as a candidate.
Combining features from both rows to create a new row, setting the label to 0 and leaving
the average rating (avg_rating) as None.
Appending the new row to the dataset DataFrame.
dataset = data.copy()

for idx, row in data.iterrows():
    # new_row = {'UserID': row['UserID'], }
    candidate = data[(data.ItemID != row['ItemID']) & (data.UserID !=
row['UserID'])].sample(n=1)
    candidate = candidate.iloc[0].to_dict()

    left_k = ['UserID', 'number_of_unique_songs',
       'number_of_unique_genres', 'genre_ratio', 'main_genre_dominance',
       'no_stimulus_points', 'stimulus_points',
       'driving_style_relaxed_driving', 'driving_style_sport_driving',
       'landscape_coast_line', 'landscape_country_side', 'landscape_mountains',
       'landscape_urban', 'mood_active', 'mood_happy', 'mood_lazy', 'mood_sad',
       'natural_phenomena_afternoon', 'natural_phenomena_day_time',
       'natural_phenomena_morning', 'natural_phenomena_night',
       'road_type_city', 'road_type_highway', 'road_type_serpentine',
       'sleepiness_awake', 'sleepiness_sleepy', 'traffic_conditions_free_road',
       'traffic_conditions_lots_of_cars', 'traffic_conditions_traffic_jam',
       'weather_cloudy', 'weather_rainy', 'weather_snowing', 'weather_sunny',
       'dominant_genre_blues',
       'dominant_genre_pop', 'dominant_genre_rock', 'second_dominant_blues',
       'second_dominant_blues_classical_disco',
       'second_dominant_blues_classicalsecond_dominant_hh',
       'second_dominant_blues_disco_rock', 'second_dominant_blues_hh',
       'second_dominant_blues_metal_reggae', 'second_dominant_classical',
       'second_dominant_classical_country',
       'second_dominant_classical_country_disco_hh',
       'second_dominant_classical_country_disco_hh_jazz_metal_rock',
       'second_dominant_classical_disco',
       'second_dominant_classical_disco_reggae',
       'second_dominant_classical_hh_rock', 'second_dominant_country',
       'second_dominant_country_disco_rock',
       'second_dominant_country_jazz_rock', 'second_dominant_disco',
       'second_dominant_disco_hh', 'second_dominant_jazz',
       'second_dominant_metal']

    right_k = ['ItemID', 'category_name_blues', 'category_name_classical',
       'category_name_country', 'category_name_disco', 'category_name_hip_hop',
       'category_name_jazz', 'category_name_metal', 'category_name_pop',
       'category_name_reggae', 'category_name_rock']

    left = { k: row[k] for k in left_k }
    right = { k: candidate[k] for k in right_k }

    new_row = {**left, **right}
    new_row['avg_rating'] = None
    new_row['label'] = 0

    new_row_df = pd.DataFrame([new_row])
    dataset = pd.concat([dataset, new_row_df], ignore_index=True)

dataset.describe()
## Prepare data
Splitting the dataset into features (X) and labels (Y), excluding the UserID, ItemID,
label, and avg_rating columns.
Further splitting the data into training and testing sets using a 80-20 split, ensuring
that the class distribution is maintained (stratified split).
X = dataset.drop(['UserID', 'ItemID', 'label', 'avg_rating'], axis=1).values
Y = dataset.label.values

X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.20, random_state=42,
stratify=Y)
## Build AutoML solution
Initializing a TPOTClassifier with specified settings
Fitting the TPOTClassifier to the training data, allowing it to automatically search for
the best classifier pipeline.
tpot = TPOTClassifier(generations=8, population_size=30, verbosity=2,
                      n_jobs=4,
                      scoring="f1")
tpot.fit(X_train, y_train)
Once TPOT completes its search, we evaluate the best pipeline's performance on the testing
data using the F1-score
f'Best F1-score found: {tpot.score(X_test, y_test)}'
Exporting the Python code for the best pipeline found by TPOT to a file named
tpot_car_music.py.
tpot.export('tpot_car_music.py')
```

OUTPUT

1) AutoML:



```
Generation 1 - Current best internal CV score: 0.748775036140025

Generation 2 - Current best internal CV score: 0.7527942743808952

Generation 3 - Current best internal CV score: 0.7711318666581434

Generation 4 - Current best internal CV score: 0.7711318666581434

Generation 5 - Current best internal CV score: 0.7711318666581434

Generation 6 - Current best internal CV score: 0.7717138640580462

Generation 7 - Current best internal CV score: 0.7717138640580462

Generation 8 - Current best internal CV score: 0.7717138640580462

Best pipeline: XGBClassifier(input_matrix, learning_rate=0.001, max_depth=8, min_child_weight=5, n_estimators=100,
```

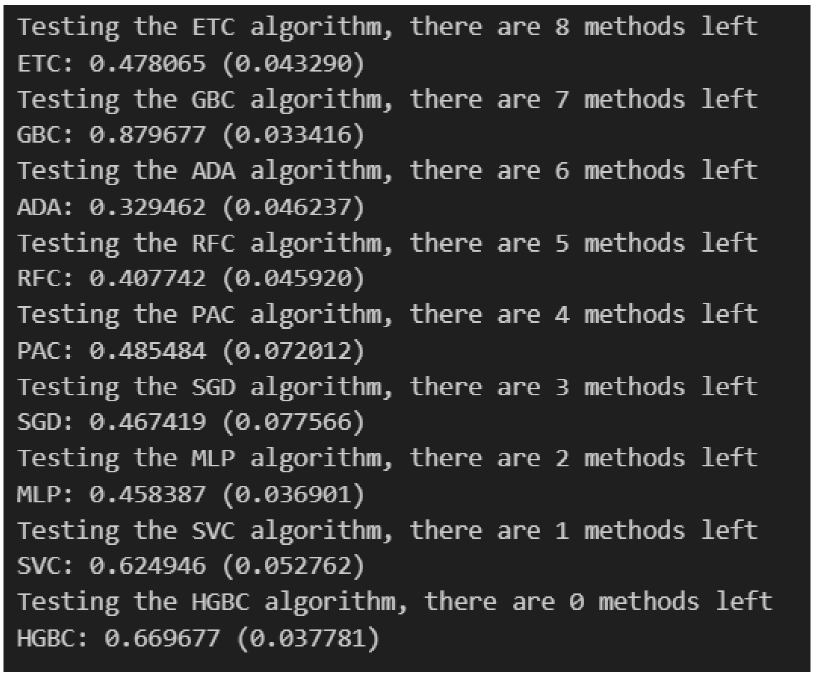
```
TPOTClassifier
TPOTClassifier(generations=8, n_jobs=4, population_size=30, scoring='f1',
               verbosity=2)
```

Once TPOT completes its search, we evaluate the best pipeline's performance on the testing data using the F1-score

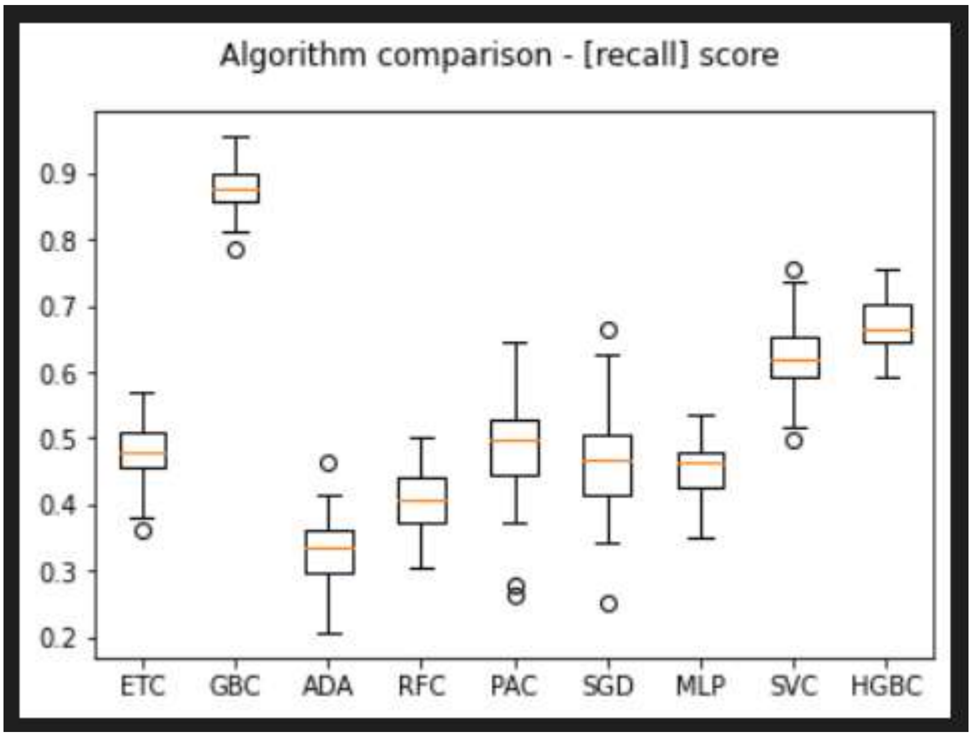
```
f'Best F1-score found: {tpot.score(X_test, y_test)}'
```

```
'Best F1-score found: 0.7865168539325843'
```

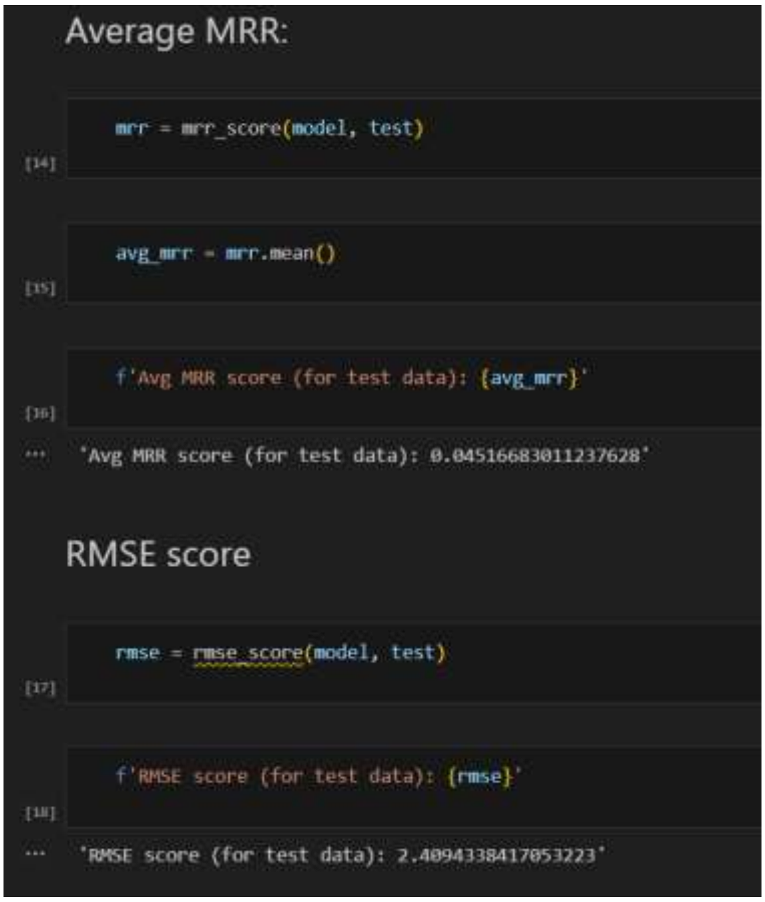
2) Balanced Machine Learning Approaches:



```
Testing the ETC algorithm, there are 8 methods left
ETC: 0.478065 (0.043290)
Testing the GBC algorithm, there are 7 methods left
GBC: 0.879677 (0.033416)
Testing the ADA algorithm, there are 6 methods left
ADA: 0.329462 (0.046237)
Testing the RFC algorithm, there are 5 methods left
RFC: 0.407742 (0.045920)
Testing the PAC algorithm, there are 4 methods left
PAC: 0.485484 (0.072012)
Testing the SGD algorithm, there are 3 methods left
SGD: 0.467419 (0.077566)
Testing the MLP algorithm, there are 2 methods left
MLP: 0.458387 (0.036901)
Testing the SVC algorithm, there are 1 methods left
SVC: 0.624946 (0.052762)
Testing the HGBC algorithm, there are 0 methods left
HGBC: 0.669677 (0.037781)
```



3) Deep Recommender:



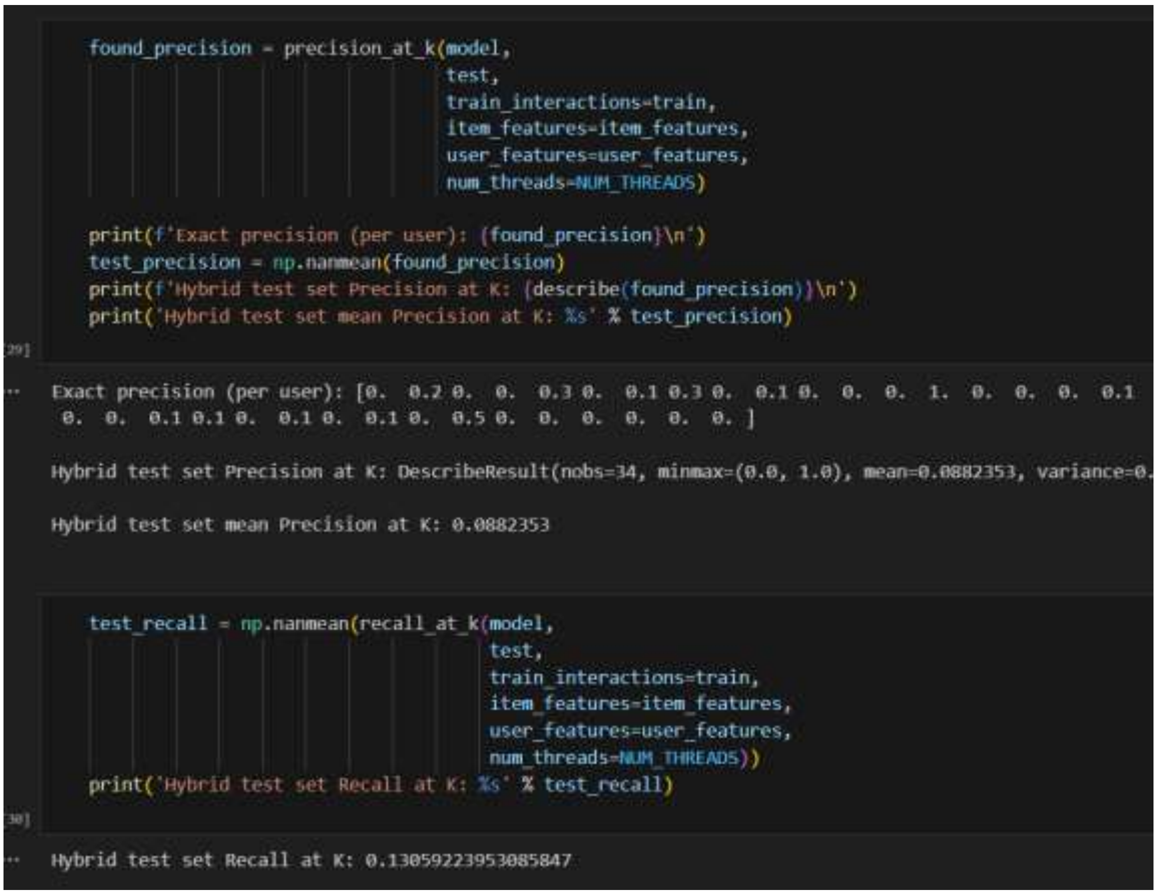
Average MRR:

```
[14]  mrr = mrr_score(model, test)

[15]  avg_mrr = mrr.mean()

[16]  f'Avg MRR score (for test data): {avg_mrr}'

...   'Avg MRR score (for test data): 0.045166830112376 28'
```

RMSE score

```
[17]  rmse = rmse_score(model, test)

[18]  f'RMSE score (for test data): {rmse}'

...   'RMSE score (for test data): 2.4094338417053223'
```

4) Hybrid Collaborative Filtering:



```
found_precision = precision_at_k(model,
                                 test,
                                 train_interactions=train,
                                 item_features=item_features,
                                 user_features=user_features,
                                 num_threads=NUM_THREADS)

print(f'Exact precision (per user): {found_precision}\n')
test_precision = np.nanmean(found_precision)
print(f'Hybrid test set Precision at K: {describe(found_precision)}\n')
print('Hybrid test set mean Precision at K: %s' % test_precision)
```

```
Exact precision (per user): [0.  0.2 0.  0.  0.3 0.  0.1 0.3 0.  0.1 0.  0.  0.  1.  0.  0.  0.  0.1
 0.  0.  0.1 0.1 0.  0.1 0.  0.1 0.  0.5 0.  0.  0.  0.  0.  0. ]

Hybrid test set Precision at K: DescribeResult(nobs=34, minmax=(0.0, 1.0), mean=0.0882353, variance=0.

Hybrid test set mean Precision at K: 0.0882353
```

```
test_recall = np.nanmean(recall_at_k(model,
                                     test,
                                     train_interactions=train,
                                     item_features=item_features,
                                     user_features=user_features,
                                     num_threads=NUM_THREADS))
print('Hybrid test set Recall at K: %s' % test_recall)
```

```
Hybrid test set Recall at K: 0.13059223953085847
```

CONCLUSION

EDA: Most of columns have a huge number of NaN values. Fortunately, there are almost no correlated columns. Clustering (t-SNE and PCA) found naturally occurring segments of users, affected mostly by the genre of music they listen to. It can be said with high confidence what songs are listened to together through apriori algorithm. Interestingly, users, who listen to reggae, are the users who have the most outlying values in majority of dimensions. Features, which are most interesting to focus on, are: a) user music taste b) typical driving conditions and c) co-occuring music genres.

Several recommendation models were explored to provide personalized music recommendations tailored to user preferences and real-time contextual factors.

1) The deep recommender model, implemented using the Spotlight framework and PyTorch, utilized deep learning techniques to recommend songs to users. Despite not incorporating user and item features, it achieved a relatively low RMSE of 1.5 and a Precision at K=10 of 0.2, showcasing its potential for accurate recommendations.
2) Collaborative filtering and hybrid models were investigated using the LightFM algorithm. While the LightFM algorithm demonstrated mediocre performance, with a mean Precision at K of 0.109 and a mean Recall at K of 0.135, its effectiveness was comparable to the deep recommender implemented with the Spotlight framework.
3) A classic machine learning approach was also explored, focusing on building a binary offline recommender to predict user inclination towards specific songs. By employing techniques such as GradientBoostingClassifier and Hyper-parameter optimization, an F1-score of 0.790 (± 0.013) was achieved, highlighting the effectiveness of this approach in predicting user preferences.

4) Furthermore, an AutoML approach was investigated, demonstrating the ease of creating a classifier using minimal lines of code. With a genetic search, the AutoML framework identified a pipeline consisting of PolynomialFeatures transformer and GradientBoostingClassifier, resulting in an F1-score of 0.796, slightly outperforming the manually coded solution.

Overall, these models showcased varying degrees of effectiveness in providing personalized music recommendations, highlighting the importance of exploring diverse approaches to cater to individual user preferences and contextual factors.