# DATA420-20S2

## Assignment 2

## The Million Song Dataset (MSD)

Jiangwei Wang (19364744)

October 6, 2020

# Content

DATA PROCESSING Q1
(a) Data structure..... 2
(b) Repartition method..... 4
(c) Row counting..... 4
DATA PROCESSING Q2
(a) Filter Taste Profile dataset..... 5
(b) Audio feature attribute..... 5
AUDIO SIMILARITY Q1
(a) Descriptive statistical analysis ..... 6
(b) Genre distribution..... 7
(c) Join genre and audio feature..... 9
AUDIO SIMILARITY Q2
(a) Research on classification algorithms..... 9
(b) Convert is “Rap” and check class balance..... 10
(c) Split the dataset by applying different sampling strategies..... 11
(d) Train each classification algorithm..... 15
(e) Model confusion matrixes and performance metrics..... 15
(f) Discussion on model performance..... 16
AUDIO SIMILARITY Q3
(a) Classification algorithms hyperparameters..... 17
(b) Cross-validation and hyperparameter tuning..... 21
AUDIO SIMILARITY Q4
(a) Choose multiclass classification algorithm..... 22
(b) Indexing genre..... 22
(c) Split, train and evaluate multiclass classification models..... 23
SONG RECOMMENDATIONS Q1
(a) Unique count on Taste Profile dataset..... 26
(b) Different songs most active user played..... 26
(c) Distribution visualization of song popularity and user activity..... 27
(d) Clean dataset..... 28
(e) Split the dataset..... 29
SONG RECOMMENDATIONS Q2
(a) Train collaborative filtering model..... 29
(b) Compare the recommendation result..... 29
(c) Collaborative filtering model evaluation metrics..... 30
WORK CITED..... 32

## DATA PROCESSING Q1 (a)
### Data structure

Following is the data structured and the data formats of how "The Million Song Dataset" is stored in HDFS:

```
/data/msd
|---audio
|   |-----attributes
|   |       |-------msd-jmir-area-of-moments-all-v1.0.attributes.csv
|   |       |-------...
|   |       |-------msd-tssd-v1.0.attributes.csv
|   |-----features
|   |       |-------msd-jmir-area-of-moments-all-v1.0.csv
|   |       |           |---------part-00000.csv.gz
|   |       |           |---------...
|   |       |           |---------part-00007.csv.gz
|   |       |-------...
|   |       |-------msd-tssd-v1.0.csv
|   |       |           |---------part-00000.csv.gz
|   |       |           |---------...
|   |       |           |---------part-00007.csv.gz
|   |-----statistics
|   |       |-------sample_properties.csv.gz
|---genre
|   |-----msd-MAGD-genreAssignment.tsv
|   |-----msd-MASD-styleAssignment.tsv
|   |-----msd-topMAGD-genreAssignment.tsv
|---main
|   |-----summary
|   |       |-------analysis.csv.gz
|   |       |-------metadata.csv.gz
|---tasteprofile
    |-----mismatches
    |       |-------sid_matches_manually_accepted.txt
    |       |-------sid_mismatches.txt
    |-----triplets.tsv
            |-------part-00000.tsv.gz
            |-------...
            |-------part-00007.tsv.gz
```

There are 4 main directories under hdfs:///data/msd, which are audio, genre, main and tasteprofile.

The total size of the msd directory is 12.9 GB. According to table 1.1, 12.3 GB is audio, 30.1 MB is genre, 174.4 MB is main and 490.4 is tasteprofile.

| size | directory |
|---|---|
| 12.3 G | /data/msd/audio |
| 30.1 M | /data/msd/genre |
| 174.4 M | /data/msd/main |
| 490.4 M | /data/msd/tasteprofile |

Table 1.1 File Sizes Under "The Million Song Dataset"

Under audio, there are another 3 directories, which are attributes, features and statistics. 12.2 GB are from features, 40.3 MB is from statistics and attributes is only occupying 103.0 KB of the space.
There are 13 csv formatted files under attributes, and there are 2 columns in each csv file, and they are both string data type.
There are 13 csv formatted directories under features, and each directory is consisted by 8 gzipped csv files. Each file has 21 columns, most of them are double data type, only the last column is string.
Under statistics directory, there are only 1 gzipped csv file, and it has 11 columns, and it contains headers for each column, including track id, title, artist name, duration etc. The first 3 columns are string data type and the rest are double data type.

There are 3 tsv files under genre directory. They have very similar sizes, which are 11.1 MB, 8.4 MB and 10.6 MB. Each tsv file has 2 columns, the first seems like track id, which starts with "TRA", and second should be the different genre, like pop_rock, rap and electronic etc.

There is another directory under main, which is summary, and there are 2 gzipped csv files under that directory. Metadata's size is 118.5 MB, which is twice large as analysis, which is 55.9 MB. They both have column headers. Analysis has many columns and it has track id, rather than song id or artist id, which metadata has, but metadata doesn't have track id.

Under tasteprofile directory, there are 2 directories, mismatches and triplets.tsv. Sizes are 2.0 MB and 488.4 MB, triplets.tsv is much larger than mismatches. Mismatches has 2 text files, one is matches manually accepted, and the other one is mismatches. They both lines of texts, both have ERROR at the beginning of each line, matches manually accepted has a "<" in front of it as well. Then followed by song id and track id inside of "<>". Then, "artist name - song name - != artist name - song name".
Triplets.tsv has 8 gzipped tsv files, each gzipped tsv consisted by 3 columns, 2 string columns, user id and song id, and followed by an integer column, which is how many times the particular song is played by each user.

## DATA PROCESSING Q1 (b)
### Repartition method

Spark split dataset into partitions that in order to execute computations in parallel on each partition. It boosts computation efficiency. Memory partitioning is extremely useful in Spark to increase the computation efficiency utility here. In general, when Spark load hdfs blocks into RDD memories with zipped formatted, 1 block 1 partition. However, each block sizes are different all the time, as long as they are less than 128 M each (this is already been split under HDFS with blocks are smaller than 128 MB each). Like under hdfs:///data/msd/audio/features, there are 13 csv directories, each csv directory has 8 csv.gz parts. The block sizes are different, depends on the size of each csv file, some of them are only 7 M each and some of them are 70 M each. When we load all of the csv.gz files, or even some of them from multiple csv directories, the partitions size in our Spark RDD memory will be different too. In this scenario, repartition is handy to formalize the size of each partition, shuffle the partitions to reach that size each, to increase the parallel computation more evenly and more efficiently, which shuffle would be the most computational expensive part [1]. It is trying to avoid small size partitions went through computations faster than large size partitions, and some cores are just resting and doing nothing while other cores are still trying to compute large partitions. What we should to is utilize all the cores we have and get data move along the computation process as fast as they can.

## DATA PROCESSING Q1 (c)
### Row counting

| Datasets Directories | Row Count |
|---|---|
| hdfs:///data/msd/audio/attributes | 3929 |
| hdfs:///data/msd/audio/features | 12927867 |
| hdfs:///data/msd/audio/statistics | 992866 |
| hdfs:///data/msd/genre | 1103077 |
| hdfs:///data/msd/main/analysis.csv.gz | 1000002 |
| hdfs:///data/msd/main/metadata.csv.gz | 1000000 |
| hdfs:///data/msd/tasteprofile/mismatches | 20032 |
| hdfs:///data/msd/tasteprofile/triplets.tsv | 48373586 |

Table 1.2 Row counts for each directory containing datasets

From table 1.2 above, we can see “tastepropile” directory has the most observations stored, most of them are come from “triplets.tsv”. Second most is “audio” directory, most of it are come from features. Metadata under “main” should be where the 1 million songs stored.

After we load the datasets into spark and counted, there are 998963 unique song ids in metadata under main directory. It was 1000000 songs, so there are 1000000 – 998963 = 1037 duplicated songs.

There are 384546 unique song ids in triplets under “tasteprofile” directory. Original row count is 48373586, and there are 1019318 unique user ids. In terms of these figures, 48373586 / 1019318 ≈ 47.46, each user is recorded in average about 47.46 times in the dataset. 1019318 / 384546 ≈ 2.65. Each song is played in average about 2.65 times by each user.

## DATA PROCESSING Q2 (a)
### Filter Taste Profile dataset

Firstly, we load all datasets under “tasteprofile” directory in schema to Spark one by one. In matches manually accepted and mismatches dataset, we remove the error message when we load into schema, and we assign each useful information into correct labeled columns. Then, we use left anti join to remove all the manual correctly matched songs from mismatches dataset, which they are manually checked afterwards but they are actually matching. The rest of the mismatch dataset are the legitimate song ids are mismatched with corresponding track ids. Even the row count is only showing there is only 1 row removed from mismatches dataset, this is due to the manually corrected matches are already taken places by them as per the introduction of the dataset. This step is only to make sure this process is necessary in general idea. Finally, we use left anti join again to remove all these mismatched song ids from “triplets” dataset. There are 48373586 rows in triplets, after we removed, there are 45795111 rows left, we removed 2578475 rows, which is more than what mismatched dataset has. Because one song appears more than one time in triplets, so this is reasonable.

## DATA PROCESSING Q2 (b)
### Audio feature attribute

Under audio directory, the csv files under attributes and features are sharing the same name prefixes. The attributes csv files just have extra “attributes” behind those prefixes. For the csv files under features directory, they are actually directories, the actual csv files are contained by each csv directory which are split into 8 parts and gzipped. The csv files under attributes directory are just names and data types for each columns of corresponding csv files under features by name prefixes. They can be used as for defining schemas for each dataset under features.

Therefore, we created a dictionary with 13 key values pairs, to map and generate schemas for each csv datasets under features directory automatically. The keys are just those shared name prefixes, values are schemas for the csv datasets under features directory. In addition, the original column names are very long, so we renamed those to reasonable short but meaningful names as well.

## AUDIO SIMILARITY Q1

The smallest dataset is called “msd-jmir-methods-of-moments-all-v1.0.csv”, its size is 35.8 MB, and 994623 rows. It has 11 columns, from fearture_0000 to freature_0009, and plus track_id. The 10 features columns are continuous numeric data type, obtained using methods such as

digital signal processing and psycho-acoustic modeling. Track_id is the id column, has 18 digits. I will use this dataset for the following.

## AUDIO SIMILARITY Q1 (a)

### Descriptive statistical analysis

| | count | mean | stddev | min | max |
|---|---|---|---|---|---|
| 0 | 994623 | 0.15498176 | 0.06646213 | 0 | 0.959 |
| 1 | 994623 | 10.3845506 | 3.86800139 | 0 | 55.42 |
| 2 | 994623 | 526.813972 | 180.437755 | 0 | 2919 |
| 3 | 994623 | 35071.9754 | 12806.8163 | 0 | 407100 |
| 4 | 994623 | 5297870.37 | 2089356.44 | 0 | 4.66E+07 |
| 5 | 994623 | 0.35084444 | 0.18557957 | 0 | 2.647 |
| 6 | 994623 | 27.463868 | 8.3526486 | 0 | 117 |
| 7 | 994623 | 1495.80918 | 505.893764 | 0 | 5834 |
| 8 | 994623 | 143165.462 | 50494.2762 | -146300 | 452500 |
| 9 | 994623 | 2.40E+07 | 9307340.3 | 0 | 9.48E+07 |

Table 1.3 Audio features numeric descriptive statistics analysis

From the numeric feature descriptive statistics analysis in table 1.3, which apart from track_id column, it analyses count, mean, standard deviation, minimum and maximum for each column from feature_0000 to feature_0009. The counts are all the same as 994623, feature_0000 has the lowest mean, which is approximate 0.15, and feature_0004 has the highest mean, which is approximate 5297870.37. feature also has the minimum standard deviation, approximate 0.07, feature_0009 has the highest standard deviation, approximate 9307340.30. Only feature_0008 column's min is -146300, all other columns are 0. Feature_0000 has the lowest maximum value, 0.959, feature_0009 has the highest maximum value, 94,777,000. We can see the scales are quite different, so center and scaling are might going to help if we would like to build a prediction model later on.

### Correlation

From the correlation analysis for audio features dataset, feature_0008 and feature_0009 are highly correlated with a correlation coefficient larger than 0.95.

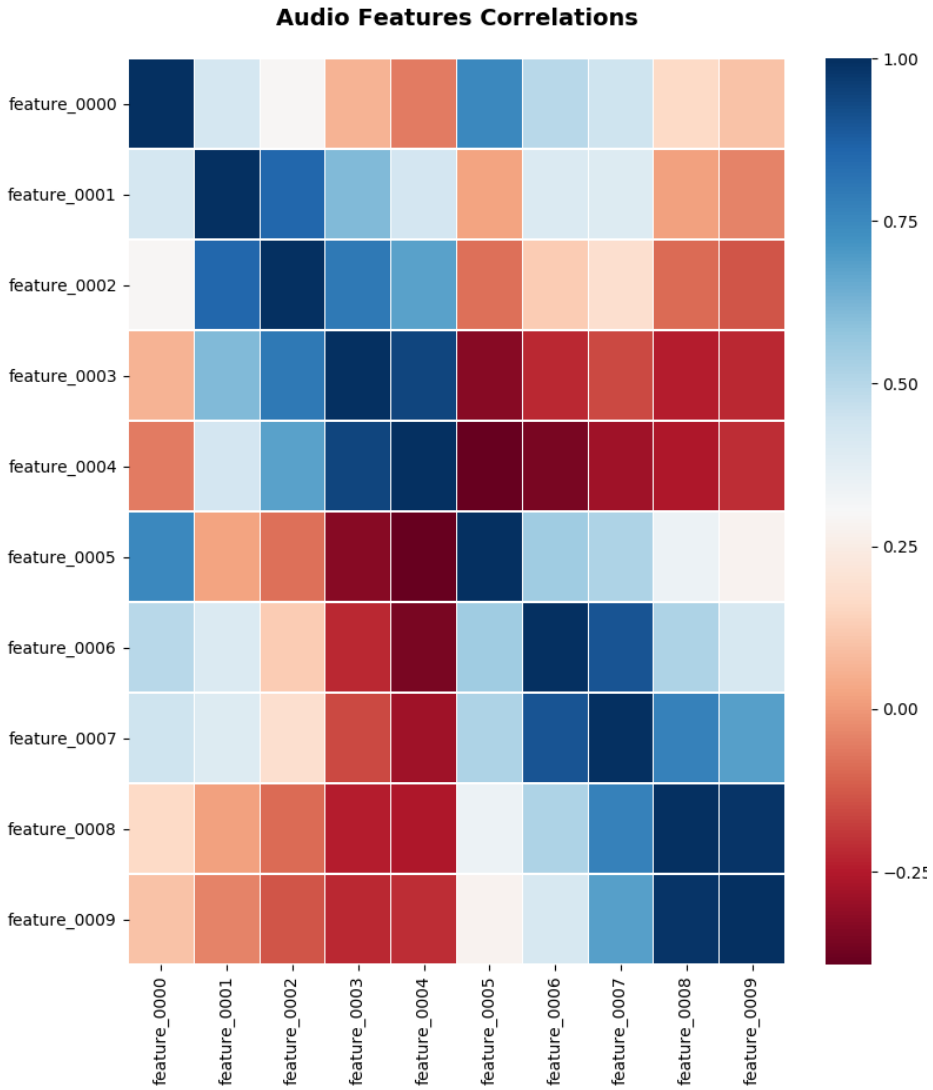


Figure 1.1 Audio features correlation heatmap

From figure 1.1, we also can see the highly correlated features by searching for darker blue colour, which means if we remove one of them, we won't loss much information for our prediction. This plot clearly shows feature 8 and feature 9 has the darkest colour respectively, thus, we managed removed one of it automatically.

## AUDIO SIMILARITY Q1 (b)

### Genre distribution

There are 2 columns in All Music Genre Dataset, one is an 18 digits track_id column, the other one is a genre for each corresponding track_id. There are 422714 records in the dataset. If we would like to have the matched songs' track ID only, we left anti join it with "mismatches not accepted"
data frame, which we already removed manually matched songs away from it, to remove any mismatched songs' track ID.

| | genre | count |
|---|---|---|
| 0 | Pop_Rock | 234107 |
| 1 | Electronic | 40430 |
| 2 | Rap | 20606 |
| 3 | Jazz | 17673 |
| 4 | Latin | 17475 |
| 5 | International | 14094 |
| 6 | RnB | 13874 |
| 7 | Country | 11492 |
| 8 | Religious | 8754 |
| 9 | Reggae | 6885 |
| 10 | Blues | 6776 |
| 11 | Vocal | 6076 |
| 12 | Folk | 5777 |
| 13 | New Age | 3935 |
| 14 | Comedy_Spoken | 2051 |
| 15 | Stage | 1604 |
| 16 | Easy_Listening | 1533 |
| 17 | Avant_Garde | 1000 |
| 18 | Classical | 542 |
| 19 | Children | 468 |
| 20 | Holiday | 198 |

Table 1.4 Distribution of matched songs' genre styles

Figure 1.2 shows there are 21 types of genres in the dataset, Pop_Rock is dominating with more than 200,000 track IDs, second place is Electronic, but with not even 50000 track IDs. Children, Classical and Holiday have the least popularity. From Table1.4, Children and Classical data frames with only around 500 track IDs, Holiday still has the least track IDs overall, which is only 198.

## Visualization

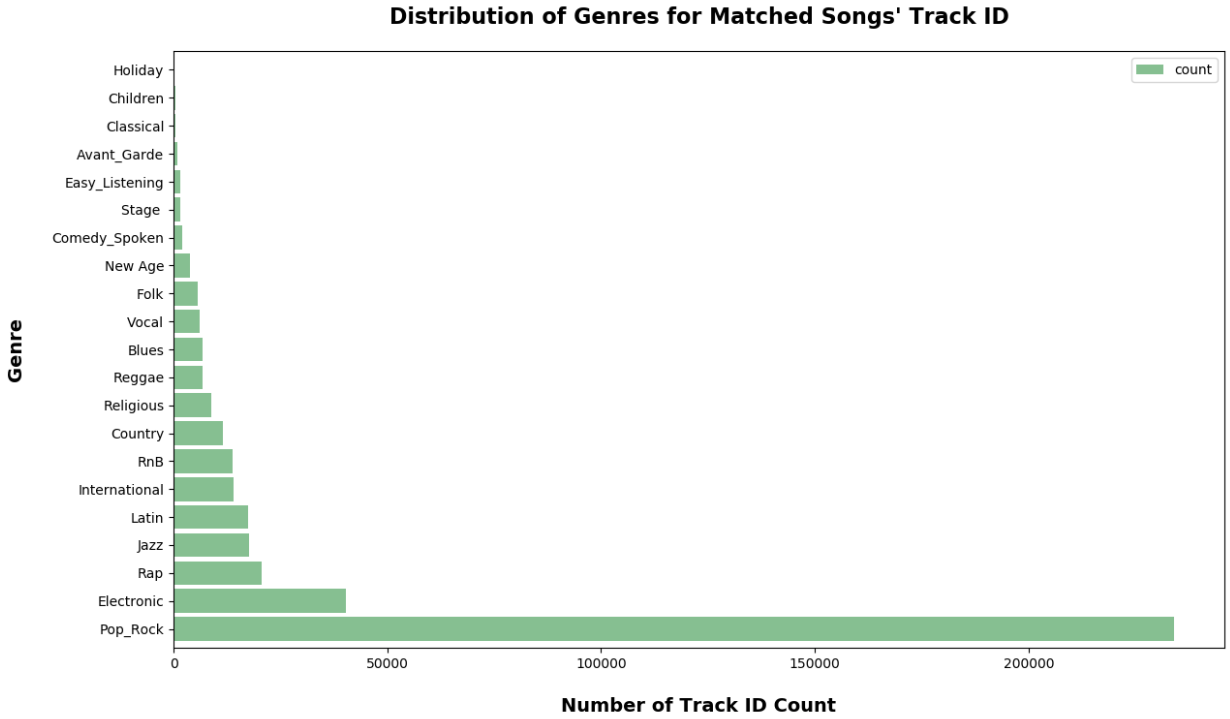


Figure 1.2 Distribution of Genres for Matched Songs' Track ID

## AUDIO SIMILARITY Q1 (c)
## Join genre and audio feature

To join audio features and genre data frames, they both have an 18 digits track_id column, with a same name, so we can join them on this column. If we want every song has a label, we can use inner join, to avoid null values. After the join, there are 420620 rows. There were 994623 rows in the audio features data frame and 422714 rows in the genre data frame. Genre data frame has less track IDs than audio feature data frame, but the joined data frame still has less rows than the genre data frame, which indicates there are more track IDs in the audio feature data frame, and there are also some track IDs in the genre data frame that audio feature doesn't have.

## AUDIO SIMILARITY Q2 (a)
## Research on classification algorithms

### Logistic Regression

I am going to start with Logistic Regression, since it is simple as a liner method, can train in a short time, and great on explainability and interpretability, which easy to tell whether a variable is dependent or independent. However, its accuracy is subject to whether the decision boundary is linear or not. Overfitting should be concerned relatively more than some other methods, like Support Vector Machine, in terms of this, hyperparameter should be tuned. It is sensible to high dimensional dataset and outliers, so center, scaling and correlation reduction should be taken in place.

### Liner Support Vector Machine

As we mentioned above, I would like to follow up with Liner Support Vector Machine method, it avoids overfitting issue better than Logistic Regression. In addition, it usually deals with outliers better than Logistic Regression. Its "soft margin constant" hyperparameter should be tuned to boost its performance by different penalties will be applied on outliers. Even is has less explainability and interpretability and slower to train than Logistic Regression as it is more complicated, but it generally performs better if there is a linear boundary. Certain preprocessing methods should be carried out too to increase the performance.

### Random Forest

Random Forest would be my next method, it has great explainability and interpretability since it can generate a variable importance chart. It can deal with categorical variables, the training speed is acceptable, but most importantly, it can deal generate a nonlinear boundary. If there is a nonlinear relationship there, this method will perform better. Its maximum number of trees, number of features considered at each split, and maximum depth of each tree hyperparameters should be tuned to make it perform as its best. Center and scaling preprocessing should be encountered, and dimensional reduction would reduce the training time.

### Gradient-Boosted Trees

Gradient-Boosted Trees would be my final approach. It is very similar as Random Forest, which is an ensemble method too. Its advantage than Random Forest is when it builds trees, each tree is learnt from the previous one, to minimize error. It has very similar characters as Random Forest regarding as hyperparameter tuning, preprocessing methods, explainability and interpretability. I would expect its better performance if there is a nonlinear boundary, without taking much longer on its training process

### Preprocessing

From the descriptive statistics analysis and correlation analysis on "msd-jmir-methods-of-moments-all-v1.0.csv" dataset earlier on, the numeric predictors' scales are different, to standardize them, such as center and scale preprocess are necessary. For the correlation part, we already dropped one of the highly correlated features, so we won't worry about it anymore.

## AUDIO SIMILARITY Q2 (b)

### Convert is "Rap" and check class balance

| is_rap | count |
|---|---|
| 1 | 20899 |
| 0 | 399721 |

Table 2.1 Class balance for the binary classification problem

There is a user define function created to add a column, whether 1 represents is "Rap", or 0 represents other genres. According to table 2.1, The class balance is 20899 observations are in

class 1, whose genre are rap, and 399721 observations are class 0, whose genre are not rap. The class imbalance ration is 20899 / 399721 ≈ 0.05. It is highly imbalanced. The rap class is only 5% of no rap class, in this binary classification problem, which means if we only predict everything as class 0 will has an accuracy as high as 95%.

## AUDIO SIMILARITY Q2 (c)

## Split the dataset by applying different sampling strategies

### Splitting

To split this dataset into training and test, I tried to split them to 80% training, 20% test, by applying different sampling strategies, including random split, stratified random split, subsampling, hybrid sampling and observation reweighting. There are some tables below to demonstrate how dataset is split, associate with their imbalanced problematic. After that, since logistic regression classification algorithm is fast to train, they are all measured by trained with it, and measured by the split test dataset with a set of measurement, including confusion matrix, precision, recall, accuracy and Auroc metrics, to have a clear comparison between them.

### Evaluate each splitting on Logistic Regression algorithm

| Random Split (not stratified) | | | |
|---|---|---|---|
| Dataset: | features | Total count: | 420620 |
| | is_rap | count | ratio |
| 0 | 0 | 399721 | 0.950314 |
| 1 | 1 | 20899 | 0.049686 |
| Dataset: | Training | Total count: | 336776 |
| | is_rap | count | ratio |
| 0 | 0 | 320046 | 0.950323 |
| 1 | 1 | 16730 | 0.049677 |
| Dataset: | Test | Total count: | 83844 |
| | is_rap | count | ratio |
| 0 | 0 | 79675 | 0.950277 |
| 1 | 1 | 4169 | 0.049723 |

Table 2.2 Class balance after training and test split by applying random split strategy

| Stratified Random Sampling | | | |
|---|---|---|---|
| Dataset: | features | Total count: | 420620 |
| | is_rap | count | ratio |
| 0 | 0 | 399721 | 0.950314 |
| 1 | 1 | 20899 | 0.049686 |
| Dataset: | Training | Total count: | 336495 |
| | is_rap | count | ratio |
| 0 | 0 | 319776 | 0.950314 |
| 1 | 1 | 16719 | 0.049686 |
| Dataset: | Test | Total count: | 84125 |
| | is_rap | count | ratio |
| 0 | 0 | 79945 | 0.950312 |
| 1 | 1 | 4180 | 0.049688 |

Table 2.3 Class balance after training and test split by applying stratified random sampling strategy

| Subsampling after Stratification | | | |
|---|---|---|---|
| Dataset: | Training | Total count: | 50272 |
| | is_rap | count | ratio |
| 0 | 0 | 33553 | 0.667429 |
| 1 | 1 | 16719 | 0.332571 |

Table 2.4 Training dataset class balance by applying subsampling strategy

| Hybrid Sampling after Stratification | | | |
|---|---|---|---|
| Dataset: | Training | Total count: | 50272 |
| | is_rap | count | ratio |
| 0 | 0 | 33553 | 0.500993 |
| 1 | 1 | 33420 | 0.499007 |

Table 2.5 Training dataset class balance by applying hybrid sampling strategy

| Observation Reweighting after Stratification | | | |
|---|---|---|---|
| Dataset: | Training | Total count: | 50272 |
| | is_rap | weights | |
| 0 | 0 | 20 | |
| 1 | 1 | 1 | |

Table 2.6 Training dataset class weights by applying observation reweighting strategy

From table 2.2, we can see after the training and test are split, the imbalance ratio roughly remains the same in each of them, but not exactly. Table 2.3 demonstrated the imbalance ratio

remained exactly the same as before splitting, by applying the stratified random sampling strategy. Table 2.4 shows after we applied subsampling strategy on the training dataset only, which is split by applying stratified random sampling strategy, randomly subsamples the class with greater amount of observation, to twice as much as the fewer class. The class balance is about 2/3 and 1/3 now. In table 2.5, we continuously apply up sampling strategy to the fewer observation class, to randomly duplicate its observation size to as twice much as before. This subsample greater class then up sample the fewer class would the hybrid sampling strategy usually be. Now, both classes are pretty much balanced, with almost 50% and 50% balance ratio. The next table, table 2.6, we applied observation reweighting method to balance out the class imbalance issue. After the dataset is split by applying stratified random sampling strategy, instead of subsample and up sample the observations from the training dataset, we assign the weights to each observation. By the consideration of class 0 is about 20 times more than class 1 on observation amount, we assign weight 1 to each observation in class 0, and weight 20 to each observation in class 1, to balance them out in the reweighting method. However, we need to be careful, according to not all of the machine learning classification algorithms accept case weights. Logistic regression algorithm accepts case weights, so we can apply it to measure how well does the observation reweighting strategy will affects our model performance.

| Random Split Confusion Matrix with LR | | |
|---|---|---|
| | Actual 1 | Actual 0 |
| Prediction 1 | 63 | 273 |
| Prediction 0 | 4106 | 79402 |
| **Stratified Confusion Matrix with LR** | | |
| | Actual 1 | Actual 0 |
| Prediction 1 | 78 | 278 |
| Prediction 0 | 4102 | 79667 |
| **Subsampling Confusion Matrix with LR** | | |
| | Actual 1 | Actual 0 |
| Prediction 1 | 2544 | 9459 |
| Prediction 0 | 1636 | 70486 |
| **Hybrid sampling Confusion Matrix with LR** | | |
| | Actual 1 | Actual 0 |
| Prediction 1 | 3367 | 16848 |
| Prediction 0 | 813 | 63097 |
| **Obs Reweighting Confusion Matrix with LR** | | |
| | Actual 1 | Actual 0 |
| Prediction 1 | 3342 | 19295 |
| Prediction 0 | 838 | 60650 |

Table 2.7 Confusion matrixes of different sampling strategies applied with logistic regression method

| Metric | Sampling Methods with Logistic Regression | | | | |
|---|---|---|---|---|---|
| | Random Split | Stratification | Subsampling | Hybrid Sampling | Observation Reweighting |
| Precision | 0.1875 | 0.219101124 | 0.211947013 | 0.166559486 | 0.147634404 |
| Recall | 0.015111538 | 0.018660287 | 0.60861244 | 0.805502392 | 0.799521531 |
| Accuracy | 0.947772053 | 0.947934621 | 0.868112927 | 0.790062407 | 0.760677563 |
| Auroc | 0.84330187 | 0.841247357 | 0.844177072 | 0.862468014 | 0.844210447 |

Table 2.8 Model performance metric on each sampling methods with logistic regression method

After we trained each training dataset, which are generated by applying different sampling methods, with logistic regression algorithm, we can see from table 2.8, their performance metrics are different. Random split has a 18.75% precision, the recall is as low as about 1.51%. Even it has an approximate 94.78% accuracy, which is good, and 84.33% auroc, which is not bad, but by looking at table 2.7, its confusion matrix tells us the precision is higher than hybrid sampling and observation reweighing strategies, only because there are only 336 out of actual 4169 positive cases are predicted, this is also why its recall is low. Even the accuracy is comparatively high, but this is mainly due to the imbalance issue, it predicts most cases as 0. By applying stratified random sample strategy, the result improved. There are more positive cases are predicted, and more true positives and true negatives than before, but still not much. With subsampling strategy kicked in, the predicted positive cases increased incredibly. There are 12033 positive cases are predicted, and in terms of this, a lot more true positives than before, which these are demonstrated in table 2.7. As a result, recall score is increased hugely, but the precision and accuracy are both dropped, because of more positive cases are predicted and a lot of them are false positives, and due to more positive cases are predicted, fewer negative cases are predicted, so the overall accuracy dropped. Auroc is improved a little bit, which is a good sign. These improvements are mainly due to the imbalance problem is partially solved by subsample the class 0, the imbalance problem is improved.
With the hybrid sampling strategy, class imbalance pretty much disappeared, precision and accuracy scores are dropped from table 2.8, is mainly due to there are further amount of total positive cases are predicted, false positives increased more than true positives, thus, fewer negative cases are predicted, overall accuracy dropped. However, the recall score improved by almost 20% and auroc improved about 2%. Observation reweighting strategy generated very similar model performance metrics as hybrid sampling strategy. Overall, hybrid sampling strategy dominating in this comparison respect to their performance metric.

In conclusion, I would use the hybrid sampling strategy, since it solved the class imbalance issue, it's model performance metric is better overall. The observation reweighing method is good as well, it solves the class imbalance in a different way, but its performance metric still not as good as hybrid resampling method, plus not all machine learning algorithms are accepting case weights in current.

## AUDIO SIMILARITY Q2 (d)
## Train each classification algorithm

## AUDIO SIMILARITY Q2 (e)
## Model confusion matrixes and performance metrics

| | Logistic Regression | | Random Forests | |
|---|---|---|---|---|
| | Actual 1 | Actual 0 | Actual 1 | Actual 0 |
| Prediction 1 | 3367 | 16848 | 3403 | 19681 |
| Prediction 0 | 813 | 63097 | 777 | 60264 |
| | Gradient-Boosted Trees | | Linear SVM | |
| Prediction 1 | 3394 | 18257 | 3322 | 17859 |
| Prediction 0 | 786 | 61688 | 858 | 62086 |

Table 2.9 Model confusion matrixes

| Logistic Regression | | | |
|---|---|---|---|
| | without CV | CV with 5 Folds | CV with 10 Folds |
| Precision | 0.166559486 | 0.163068238 | 0.163076313 |
| Recall | 0.805502392 | 0.787799043 | 0.787799043 |
| Accuracy | 0.790062407 | 0.788552748 | 0.788564486 |
| Auroc | 0.862468014 | 0.855647849 | 0.855644860 |
| Random Forests | | | |
| | without CV | CV with 5 Folds | CV with 10 Folds |
| Precision | 0.147418125 | 0.145949394 | 0.145949394 |
| Recall | 0.814114833 | 0.821052632 | 0.821052632 |
| Accuracy | 0.756814264 | 0.752380386 | 0.752380386 |
| Auroc | 0.855572524 | 0.857328593 | 0.857328593 |
| Gradient-Boosted Trees | | | |
| | without CV | CV with 5 Folds | CV with 10 Folds |
| Precision | 0.156759503 | 0.168065507 | 0.169531652 |
| Recall | 0.811961722 | 0.785645933 | 0.788038278 |
| Accuracy | 0.773634473 | 0.796112927 | 0.797658247 |
| Auroc | 0.869900733 | 0.868510129 | 0.87091473 |
| Linear Support Vector Machine | | | |
| | without CV | CV with 5 Folds | CV with 10 Folds |
| Precision | 0.156838676 | 0.160598528 | 0.158395967 |
| Recall | 0.794736842 | 0.788277512 | 0.811722488 |
| Accuracy | 0.777509658 | 0.784760773 | 0.776344725 |
| Auroc | 0.854189378 | 0.854506669 | 0.857804952 |

Table 2.10 Model performance evaluation metrics

## AUDIO SIMILARITY Q2 (f)
## Discussion on model performance

### Model performance

There are four models I have trained; their performance metrics and confusion matrixes are shown in table 2.9 and 2.10. From these two tables, the model with the best performance overall is Gradient-Boosted Trees (GBT). It has a highest Auroc score, rather a highest accuracy, compare to other three models. Logistic Regression's performing is outstanding too. However, compare Logistic Regression (LR) with GBT, GBT predicted more true positive (TP) cases than LR, which is shown is table 2.9, without losing much true negative (TN) cases than LR, TP is what we usually more care about or targeted. Thus, GBT wins with higher recall, without losing much precision, but with a slightly better Auroc score than LR model. Random Forests (RF) predicted the most TP overall, but it tends to predict a large amount of cases as positive, that's why its false positive number is the most overall too. By considering both recall and precision, it wins more recall than all the others, but it lost more precision than it wins on recall, so the overall Auroc is slightly lower than GBT and LR models. Linear Support Vector Machine (LSVM) model is losing by gain not enough TP, but still with a relatively larger amount of false positive cases.

RF and GBT are both tree-based methods, who can handle non-linear relationship better than the other two. By looking at their performance, they didn't do much better than the LR and LSVM, from statistical speaking, we can't say the features has non-linear relationship with the outcome variable. GBT perform better than RF might due to its weak learner feature, random forest builds each tree independently by random sample the data, it helps the model with robust feature and avoid overfitting than Decision Trees. GBT builds trees sequentially, each tree is built by learning from the previous one to correct errors. In general, GBT is taking longer to train, it is computational more expensive [2]. In addition, since RF and LSVM are both robust methods, who can handle outliers better. In this case, from statistical speaking, there is no significant evidence shows outliers will be an issue here.

### Threshold

| | Logistic Regression Threshold | | | | | |
|---|---|---|---|---|---|---|
| | 0.5 | | 0.2 | | 0.7 | |
| | Actual 1 | Actual 0 | Actual 1 | Actual 0 | Actual 1 | Actual 0 |
| **Prediction 1** | 3367 | 16848 | 4005 | 44596 | 2384 | 7904 |
| **Prediction 0** | 813 | 63097 | 175 | 35349 | 1796 | 72041 |

Table 2.11 Confusion matrixes of Logistic Regression with different thresholds

| | Logistic Regresion Threshold | | |
|---|---|---|---|
| | 0.5 | 0.2 | 0.7 |
| Precision | 0.16655949 | 0.08240571 | 0.23172628 |
| Recall | 0.80550239 | 0.95813397 | 0.57033493 |
| Accuracy | 0.79006241 | 0.46780386 | 0.88469539 |
| Auroc | 0.86246801 | 0.85905907 | 0.85925668 |

Table 2.12 Performance metrics of Logistic Regression with different thresholds

From table 2.12, we noticed that threshold won't change the model overall performance by looking at the Auroc score, it only changes the trade off between precision and recall, based on a same performance level. From table 2.11, the matrixes show if we move threshold lower to 0.2, the prediction probability that is going to predict as positive is lower, there are a lot of more positive cases are predicted, thus the TP increased, but the true negative cases dropped significantly accordingly. In comparison, if we move the threshold to 0.7, the prediction probability that is going to be predicted as 1 are more restricted, the amount of positive cases is predicted affected instantly. TP decreased in a certain percentage and true negative cases increased accordingly.

### Class balance

Again, as what we discussed before, class imbalance affects our model performance significantly. It doesn't like threshold only change the tradeoff between precision and recall, it affects the overall performance and can be judged by Auroc score. In this case, the binary classes are significantly imbalanced, there are only 5% positive cases. If we predict everything as negative, we can achieve a 95% accuracy. Class imbalance can be solved in different strategies, which I demonstrated earlier in the assignment and justified my choice on the strategy I'm using as well.

## AUDIO SIMILARITY Q3 (a)

## Classification algorithms hyperparameters

### Logistic Regression

The default hyperparameter settings for LR is LogisticRegression(penalty='l2', *, dual=False, tol=0.0001, C=1.0, fit_intercept=True, intercept_scaling=1, class_weight=None, random_state=None, solver='lbfgs', max_iter=100, multi_class='auto', verbose=0, warm_start=False, n_jobs=None, l1_ratio=None) [3].

- The 'newton-cg', 'sag' and 'lbfgs' solvers support only l2 penalties. 'elasticnet' is only supported by the 'saga' solver.
- Prefer dual=False when n_samples > n_features.
- Tol is Tolerance for stopping criteria.
- C is Inverse of regularization strength; must be a positive float. Like in support vector machines, smaller values specify stronger regularization
- Fit_intercept specifies if a constant (a.k.a. bias or intercept) should be added to the decision function.
- intercept_scaling is useful only when the solver 'liblinear' is used and self.fit_intercept is set to True. In this case, x becomes [x, self.intercept_scaling], i.e. a "synthetic" feature

with constant value equal to intercept_scaling is appended to the instance vector. The intercept becomes intercept_scaling * synthetic_feature_weight.

- Class weight is set to None as default by not accepting class weights
- Random_state is used when solver == ‘sag’, ‘saga’ or ‘liblinear’ to shuffle the data.
- Solver='lbfgs'can handle L2, no penalty and multiclassification proplem.
- Max_iter=100 is the maximum number of iterations taken for the solvers to converge.
- Multi-class is set to auto as default.
- Verbose is for the liblinear and lbfgs solvers set verbose to any positive number for verbosity.
- When set warm_start to True, reuse the solution of the previous call to fit as initialization, otherwise, just erase the previous solution. Useless for liblinear solver.
- N-jobs is number of CPU cores used when parallelizing over classes if multi_class=’ovr’”.
- The Elastic-Net mixing parameter l1_ratio, with 0 <= l1_ratio <= 1. Only used if penalty='elasticnet'. Setting l1_ratio=0 is equivalent to using penalty='l2', while setting l1_ratio=1 is equivalent to using penalty='l1'. For 0 < l1_ratio <1, the penalty is a combination of L1 and L2.

I have trained the LR model with all above default hyperparameter settings, which most of them seem reasonable and sensible for this dataset, such as prefer dual, class weight and max iteration etc. But some of the them can still be specified for hyperparameter tuning, like C1 and Elastic-Net mixing parameter l1_ratio.

### Random Forests

The default hyperparameter settings for RF is RandomForestClassifier(n_estimators=100, *, criterion='gini', max_depth=None, min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_features='auto', max_leaf_nodes=None, min_impurity_decrease=0.0, min_impurity_split=None, bootstrap=True, oob_score=False, n_jobs=None, random_state=None, verbose=0, warm_start=False, class_weight=None, ccp_alpha=0.0, max_samples=None) [4].

- N_estimators defines the number of trees in the forest.
- Criterion is the function to measure the quality of a split. Supported criteria are “gini” for the Gini impurity and “entropy” for the information gain.
- max_depth defines the maximum depth of the tree.
- Min_samples_split defines the minimum number of samples required to split an internal node.
- Min_samples_leaf is the minimum number of samples required to be at a leaf node.
- Min_weight_fraction_leaf is the minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Samples have equal weight when sample_weight is not provided.
- Max_features is the number of features to consider when looking for the best split
- Grow trees with max_leaf_nodes in best-first fashion. Best nodes are defined as relative reduction in impurity. If None then unlimited number of leaf nodes
- Min_impurity_decrease defines a node will be split if this split induces a decrease of the impurity greater than or equal to this value.

- Min_impurity_split defines a threshold for early stopping in tree growth. A node will split if its impurity is above the threshold, otherwise it is a leaf.
- Bootstrap is whether bootstrap samples are used when building trees. If False, the whole dataset is used to build each tree.
- Oob_score states whether to use out-of-bag samples to estimate the generalization accuracy.
- N_jobs defines the number of jobs to run in parallel.
- Random_state controls both the randomness of the bootstrapping of the samples used when building trees.
- Verbose Controls the verbosity when fitting and predicting.
- When set warm_start to True, reuse the solution of the previous call to fit and add more estimators to the ensemble, otherwise, just fit a whole new forest.
- Class_weight is looking for a feature specifies class weights, if set as none, all classes are supposed to have weight 1.
- Ccp_alpha is the complexity parameter used for Minimal Cost-Complexity Pruning. The subtree with the largest cost complexity that is smaller than ccp_alpha will be chosen. By default, no pruning is performed.
- Max_samples is when bootstrap is True, the number of samples to draw from X to train each base estimator.

I trained the RF model by using its default hyperparameter settings as above, most of them are reasonable and sensible. However, RF is an ensemble method, it has a lot of options for tuning hyperparameters, such as number of trees, depth of trees and bootstrapping, which we can tune them later.

### Gradient-Boosted Trees

The default hyperparameter settings for GBT is GradientBoostingClassifier(*, loss='deviance', learning_rate=0.1, n_estimators=100, subsample=1.0, criterion='friedman_mse', min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_depth=3, min_impurity_decrease=0.0, min_impurity_split=None, init=None, random_state=None, max_features=None, verbose=0, max_leaf_nodes=None, warm_start=False, presort='deprecated', validation_fraction=0.1, n_iter_no_change=None, tol=0.0001, ccp_alpha=0.0) [5].

GBT is also a tree-based ensemble method, it has a lot of similar characters and principles as RF, apart RF is resampling from the dataset to grow each tree, and GBT is utlising its weak learner character to grow each tree by learning from the previous one to correct errors. They both have a range of same hyperparameters, following is the hyperparameters distinctive for GBT only:

- Loss is a loss function to be optimized. ‘deviance’ refers to deviance (= logistic regression) for classification with probabilistic outputs.
- Learning_rate is earning rate that shrinks the contribution of each tree by learning_rate. There is a trade-off between learning_rate and n_estimators.
- N_estimators is the number of boosting stages to perform. Gradient boosting is fairly robust to over-fitting, so a large number usually results in better performance.

- Subsample is the fraction of samples to be used for fitting the individual base learners. If smaller than 1.0 this results in Stochastic Gradient Boosting.
- Init is an estimator object that is used to compute the initial predictions. init has to provide fit and predict_proba. If 'zero', the initial raw predictions are set to zero. By default, a DummyEstimator predicting the classes priors is used.
- Presort is deprecated and will be removed in v0.24.
- Validation_fraction is the proportion of training data to set aside as validation set for early stopping. Must be between 0 and 1. Only used if n_iter_no_change is set to an integer.
- N_iter_no_change is used to decide if early stopping will be used to terminate training when validation score is not improving. By default, it is set to None to disable early stopping. If set to a number, it will set aside validation_fraction size of the training data as validation and terminate training when validation score is not improving in all of the previous n_iter_no_change numbers of iterations. The split is stratified.
- Tol is tolerance for the early stopping. When the loss is not improving by at least tol for n_iter_no_changeiterations (if set to a number), the training stops.

I trained the GBT model by using all default hyperparameter settings, they are all very detailed too and has a lot of options for tuning, such as tree depth, boosting and learning rate.

### Linear Support Vector Machine

The default hyperparameter settings for LSVM is
LinearSVC(penalty='l2', loss='squared_hinge', *, dual=True, tol=0.0001, C=1.0, multi_class='ovr', fit_intercept=True, intercept_scaling=1, class_weight=None, verbose=0, random_state=None, max_iter=1000) [6].

- Penalty specifies the norm used in the penalization. The 'l2' penalty is the standard used in SVC. The 'l1' leads to coef_ vectors that are sparse.
- Loss specifies the loss function. 'hinge' is the standard SVM loss (used e.g. by the SVC class) while 'squared_hinge' is the square of the hinge loss.
- Dual selects the algorithm to either solve the dual or primal optimization problem. Prefer dual=False when n_samples > n_features.
- Tol is tolerance for stopping criteria.
- C is the regularization parameter. The strength of the regularization is inversely proportional to C. Must be strictly positive.
- Multi_class determines the multi-class strategy if y contains more than two classes. "ovr" trains n_classes one-vs-rest classifiers, while "crammer_singer" optimizes a joint objective over all classes.
- Fit_intercept is whether to calculate the intercept for this model. If set to false, no intercept will be used in calculations (i.e. data is expected to be already centered).
- Intercept_scaling is when self.fit_intercept is True, instance vector x becomes [x, self.intercept_scaling], i.e. a "synthetic" feature with constant value equals to intercept_scaling is appended to the instance vector.

- Class_weight is looking for a feature specifies class weights, if set as none, all classes are supposed to have weight 1.
- Verbose enables verbose output. Note that this setting takes advantage of a per-process runtime setting in liblinear that, if enabled, may not work properly in a multithreaded context.
- Random_state controls the pseudo random number generation for shuffling the data for the dual coordinate descent (if dual=True). When dual=False the underlying implementation of LinearSVC is not random andrandom_state has no effect on the results.
- Max_iter is the maximum number of iterations to be run.

The LSVM model is trained by all the default hyperparameter settings, the settings are reasonable and sensible already, but we still can tune the regularization parameter, which is the C value.

### In summary

Different classification algorithms have different hyperparameters. However, some of them are similar, especially the RF and GBT, they both based on growing trees, resample and boosting to grow a forest. LR and LSVM both have L1 and L2 penalties. LR, RF and GBT all have warm start option to reuse the previous solution when fit a model. RF, GBT and LSVM all have random states option, but are used in different ways between tree-bases methods and LSVM model. LR and RF both can set number of jobs to execute parallel processing. All the four methods have case weights acceptation options.

## AUDIO SIMILARITY Q3 (b)

## Cross-validation and hyperparameter tuning

### Logistic Regression

For the cross-validation to tune LR models hyperparameters, I choose to tune the C value, which is regularization parameter, and Elastic-Net mixing parameter l1_ratio. I trained the model with cross-validation twice, one is with 5 folds and the other is with 10 folds cross-validation training. From table 2.10, we can see the cross-validation either with 5 or 10 folds seems didn't improve the result. They are actually slightly lower than without cross-validation training might because I trained them in different time in pyspark, hence, they are different training and testing dataset are used, might affect the result a little bit. These two hyperparameters are the shadow hyperparameters in this case from statistical speaking, because they are insensitive with different settings.

### Random Forests

Number of trees, maximum tree depth and maximum bins are tuned with cross-validation in RF model, either with 5 folds or 10 folds cross-validation training, they both increased recall by 0.01 and Auroc by 0.002 approximately, which are shown in table 2,10. There is no difference between 5 folds and 10 folds, the hyperparameter is affecting the result, but not much.

### Gradient-Boosted Trees

Maximum bins and maximum depth hyperparameters are also tuned in GBT, with 5 folds and 10 folds cross-validation training. The cross-validation training didn't affect the result much, it increased 0.01 on precision but sacrificed 0.03 on recall, which means it predicted more true negative cases and less true positive cases than before. The accuracy increased illustrate the increase amount on true negative cases is more than the decrease amount on TP cases. The Auroc are very similar again in all three models. The hyperparameters are insensitive again here.

### Linear Support Vector Machine

Only regularization parameter, the C value is tuned in LSVM model, with 5 folds and 10 folds cross-validation training. From table 2.10 again, 5 folds cross-validation training didn't change the result, it only predicted more true negative cases and less TP, by look at the precision increased and recall decreased, but the accuracy and Auroc are both very similar. The 10 folds cross-validation training boosted the model performance a little bit by the increased recall and Auroc scores. The hyperparameter is sensitive here, if we set the cross-validation folds to a greater integer number, maybe 15 or 20, it might will lift the model performance even further.

## AUDIO SIMILARITY Q4 (a)

### Choose multiclass classification algorithm

There are only two algorithms are supporting multiclass classification, which are LR and RF. I would like to choose the Logistic Regression, the reasons are:

- LR has a better performance metric.
- LR is simple as fast, more computational efficient than RF.
- LR has better explainability and interpretability.
- There is no evidence is showing a non-linear relationship between the predictors and outcome, so LR is appropriate.
- There is no strong evidence showing that outlier is an issue here, so LR is appropriate.
- LR has a model summary function to display model performance metric automatically for multiclass classification problems, that RF doesn't capable.

## AUDIO SIMILARITY Q4 (b)

### Indexing genre

| genre | label |
|---|---|
| Pop_Rock | [0.0] |
| Electronic | [1.0] |
| Rap | [2.0] |
| Jazz | [3.0] |
| Latin | [4.0] |
| RnB | [5.0] |
| International | [6.0] |
| Country | [7.0] |

| | |
|---|---|
| Religious | [8.0] |
| Reggae | [9.0] |
| Blues | [10.0] |
| Vocal | [11.0] |
| Folk | [12.0] |
| New Age | [13.0] |
| Comedy_Spoken | [14.0] |
| Stage | [15.0] |
| Easy_Listening | [16.0] |
| Avant_Garde | [17.0] |
| Classical | [18.0] |
| Children | [19.0] |
| Holiday | [20.0] |

Table 2.13 Genre column with corresponding label

I used string indexer, which imported from "pysprk.ml.feature" library, created a new column with label on the 21 genre styles. Table 2.13 shows the result by applying the string indexer, and I found the order from 0 to 20 labels are exactly same as their counting order, from the greatest group to the least.

## AUDIO SIMILARITY Q4 (c)

## Split, train and evaluate multiclass classification models

### Splitting

| genres | | | | training | | | | test | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 420620 | | | | 336199 | | | | 84421 | | | |
| | label | count | ratio | | label | count | ratio | | label | count | ratio |
| 0 | 12.0 | 5789 | 0.013763 | 0 | 12.0 | 4623 | 0.013751 | 0 | 6.0 | 2861 | 0.033890 |
| 1 | 6.0 | 14194 | 0.033745 | 1 | 6.0 | 11333 | 0.033709 | 1 | 13.0 | 818 | 0.009690 |
| 2 | 13.0 | 4000 | 0.009510 | 2 | 13.0 | 3182 | 0.009465 | 2 | 12.0 | 1166 | 0.013812 |
| 3 | 1.0 | 40666 | 0.096681 | 3 | 1.0 | 32494 | 0.096651 | 3 | 1.0 | 8172 | 0.096801 |
| 4 | 10.0 | 6801 | 0.016169 | 4 | 10.0 | 5473 | 0.016279 | 4 | 10.0 | 1328 | 0.015731 |
| 5 | 16.0 | 1535 | 0.003649 | 5 | 16.0 | 1223 | 0.003638 | 5 | 16.0 | 312 | 0.003696 |
| 6 | 19.0 | 463 | 0.001101 | 6 | 19.0 | 375 | 0.001115 | 6 | 19.0 | 88 | 0.001042 |
| 7 | 5.0 | 14314 | 0.034031 | 7 | 5.0 | 11373 | 0.033828 | 7 | 5.0 | 2941 | 0.034837 |
| 8 | 7.0 | 11691 | 0.027795 | 8 | 7.0 | 9340 | 0.027781 | 8 | 7.0 | 2351 | 0.027849 |
| 9 | 4.0 | 17504 | 0.041615 | 9 | 4.0 | 14069 | 0.041847 | 9 | 4.0 | 3435 | 0.040689 |
| 10 | 17.0 | 1012 | 0.002406 | 10 | 17.0 | 816 | 0.002427 | 10 | 17.0 | 196 | 0.002322 |
| 11 | 20.0 | 200 | 0.000475 | 11 | 20.0 | 160 | 0.000476 | 11 | 20.0 | 40 | 0.000474 |
| 12 | 8.0 | 8780 | 0.020874 | 12 | 8.0 | 6964 | 0.020714 | 12 | 9.0 | 1389 | 0.016453 |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 13 | 9.0 | 6931 | 0.016478 | 13 | 9.0 | 5542 | 0.016484 | 13 | 8.0 | 1816 | 0.021511 |
| 14 | 14.0 | 2067 | 0.004914 | 14 | 14.0 | 1643 | 0.004887 | 14 | 14.0 | 424 | 0.005022 |
| 15 | 0.0 | 237649 | 0.564997 | 15 | 0.0 | 189983 | 0.565091 | 15 | 0.0 | 47666 | 0.564623 |
| 16 | 18.0 | 555 | 0.001319 | 16 | 18.0 | 436 | 0.001297 | 16 | 18.0 | 119 | 0.001410 |
| 17 | 3.0 | 17775 | 0.042259 | 17 | 3.0 | 14225 | 0.042311 | 17 | 3.0 | 3550 | 0.042051 |
| 18 | 2.0 | 20899 | 0.049686 | 18 | 2.0 | 16730 | 0.049762 | 18 | 2.0 | 4169 | 0.049383 |
| 19 | 15.0 | 1613 | 0.003835 | 19 | 11.0 | 4929 | 0.014661 | 19 | 15.0 | 327 | 0.003873 |
| 20 | 11.0 | 6182 | 0.014697 | 20 | 15.0 | 1286 | 0.003825 | 20 | 11.0 | 1253 | 0.014842 |

Table 2.14 Training and test split count and ratio in each class

After we randomly split the training and test dataset, from table 2.14, each class ratio is similar as before. This is good to remain the same ratio in both test and training that can reflex the original truth.

### Preprocessing

Following that, I preprocessed the numeric predictors by centering and scaling.

### Training

| LR | | | LR with CV(5f) | | | RF | | |
|---|---|---|---|---|---|---|---|---|
| label | count | ratio | label | count | ratio | label | count | ratio |
| 0 | 78503 | 0.929899 | 0 | 78503 | 0.929899 | 0 | 82861 | 0.981521 |
| 1 | 3191 | 0.037799 | 1 | 3191 | 0.037799 | 1 | 1560 | 0.018479 |
| 2 | 1994 | 0.02362 | 2 | 1994 | 0.02362 | | | |
| 3 | 542 | 0.00642 | 3 | 542 | 0.00642 | | | |
| 5 | 171 | 0.002026 | 5 | 171 | 0.002026 | | | |
| 9 | 5 | 0.000059 | 9 | 5 | 0.000059 | | | |
| 14 | 6 | 0.000071 | 14 | 6 | 0.000071 | | | |
| 15 | 9 | 0.000107 | 15 | 9 | 0.000107 | | | |

Table 2.15 Model predicted results by class

The multiclass training dataset is trained by LR, RF and LR with 5 folds cross-validation hyperparameter tuning training. Table 2.15 demonstrates how the predicted result distributed in all classes. LR predicted most of the cases as 0 as almost 93%, which is “Pop Rock”, which can be justified in table 2.13, 1, “Electric”, and 2, “Rap”, are roughly 3.8% and 2.4%. The least predicted class shown in the table is 9, which is “Reggae”, only predicted with 5 observations. “Comedy Spoken” and “Stage” are both only predicted with single digit too. All the class are missing from the table are resulting in 0 predicted. The ratios are very different with their original, “Pop Rock” ratio is only 0.565 in the actual dataset, 1 is 0.097, 2 is almost 0.05. As a result, this class imbalance issue affected our prediction performance in a certain level. The larger class is significantly over predicted in our model, the rest are under predicted, and not all classes are presented in the prediction.

The RF model is worse as expected and it affected by the class imbalance issue even further, with only 2 classes predicted, which are 0 and 1, the two largest classes in the dataset. More than 98% are class 0.

The cross-validation I trained with the LR model, has the same setting for hyperparameter tuning, has exactly the same performance without it. This result is the same as before, while we train the binary classification problem. Those two hyperparameters are shadow hyperparameters with statistical speaking, they are insensitive in this case.

### Evaluation metrics

| label | precision | recall |
|---|---|---|
| 0 | 0.58797626 | 0.96987625 |
| 1 | 0.400407 | 0.15138179 |
| 2 | 0.35839665 | 0.16353855 |
| 3 | 0.31720676 | 0.04885765 |
| 4 | 0 | 0 |
| 5 | 0.23604466 | 0.01301328 |
| 6 | 0 | 0 |
| 7 | 0 | 0 |
| 8 | 0 | 0 |
| 9 | 0.08 | 0.00036088 |

Table 2.16 LR multiclass classification evaluation metrics by class

Table 2.16 demonstrates the precision and recall scores for each class in this multiclass classification problem that trained with LR algorithm. This algorithm over predicted most cases as class 0, thus, its precision score is not high, too many false positives, but recall is high, most class 0 are predicted correctly. The rest of the classes' precision and recall are reduced sequentially according to their size. Since class 0 is over predicted, all the other classes are under predicted, the ones are with precision and recall scores, their precision are higher than recall, which means each classes that are predicted, are still with a certain level of accuracy, so higher precision, but not enough cases are predicted in each class, so lower recall.

| | LR | LR with CV(5f) | RF |
|---|---|---|---|
| actual total: | 84421 | 84421 | 84421 |
| f1: | 0.45061284 | 0.45061284 | 0.42500618 |
| accuracy: | 0.57290248 | 0.57290248 | 0.56990559 |
| weightedPrecision: | 0.40868765 | 0.40868765 | 0.37092492 |
| weightedRecall: | 0.57290248 | 0.57290248 | 0.56990559 |

Table 2.17 Model performance evaluation metrics

Table 2.17 are the overall evaluation scores that considering the average in all classes. F1 score is the most obvious score, which indicates the overall performance, by balancing precision and

recall scores. F1 is a similar evaluation metric as Auroc in binary classifications. We can generate around 0.86 Auroc scores with binary classification, but here, f1 is only 0.45. In the binary classification problem before, due to the highly imbalance character, if we predict everything to the larger class, we can still get a high accuracy and reasonably high Auroc score. In this multiclassification problem here, there are 56.5% of the dataset are class 0, the model is impacted by the imbalance issue and still try to predict most of the cases as the largest class, hence, the performance dropped significantly. In this case, if we would like to achieve a better score, we can use coalescing strategy and combine the rest of the classes to form another class, to simplify it to a binary classification problem, and the imbalance issue is less significant as we have a 56.5% and 43.5% class balance ratio.

The LR model with 5 folds cross-validation training has exactly performance metrics. RF model has lower performance metrics score as expected, the f1 score dropped by about 0.025.

## SONG RECOMMENDATIONS Q1 (a)
### Unique count on Taste Profile dataset

Firstly, we need to make sure the Taste Profile dataset not including any mismatched songs apart from those manually accepted mismatches. There were originally 48,373,586 observations in the triplets dataset, remove 19,093 mismatches not accepted songs, there are 45,795,111. We actually removed 48,373,586 - 45,795,111 = 2,578,475 songs from the dataset, which means the mismatched songs appeared in the dataset more than once, or 2,578,475 / 19,093 ≈ 135 times each song in average. After we removed those songs, we found out:

- There are 378,310 different songs left in the dataset.
- There are 1,019,318 users in this dataset.

There are more users than song count.

## SONG RECOMMENDATIONS Q1 (b)
### Different songs most active user played

| User ID | Play Counts | Unique Song Counts |
|---|---|---|
| 093cb74eb3c517c5179ae24caf0ebec51b24d2a2 | 13074 | 195 |
| 119b7c88d58d0c6eb051365c103da5caf817bea6 | 9104 | 1362 |
| 3fa44653315697f42410a30cb766a4eb102080bb | 8025 | 146 |
| a2679496cd0af9779a92a13ff7c6af5c81ea8c7b | 6506 | 518 |
| d7d2d888ae04d16e994d6964214a1de81392ee04 | 6190 | 1257 |
| 4ae01afa8f2430ea0704d502bc7b57fb52164882 | 6153 | 453 |
| b7c24f770be6b802805ac0e2106624a517643c17 | 5827 | 1364 |
| 113255a012b2affeab62607563d03fbdf31b08e7 | 5471 | 1096 |
| 99ac3d883681e21ea68071019dba828ce76fe94d | 5385 | 939 |
| 6d625c6557df84b60d90426c0116138b617b9449 | 5362 | 1307 |

Table 3.1 Top 10 active players

According to table 3.1, the most active player played 13,074 times, but only 195 different songs. That's 195 / 378310 * 100 ≈ 0.05%.
We also can see, the some less active users played more than 1,000 different songs. Thus, different users, different activities, e.g. a café could play fewer songs, but it will play many times on each song.

## SONG RECOMMENDATIONS Q1 (c)

## Distribution visualization of song popularity and user activity

### Statistics

From our statistics analysis on the user activity, it shows each user played at least 3 different songs, or played at least 3 times with the play count. The most different songs played by one user is 4,316, and the most active user, played 13,074 times, which we also can see this from table 3.1. In average, about 45 different songs played by each user. Also, each user played about 129 times in average. The variation of play times count is larger than different song count.

For song popularity analysis, the most popular song is played by 90,444 users, and the most play times is 726,885 on a single song. Each song is play at least by 1 user or 1 time. In average, about 121 users played each song and each song is played 347 times. The variation on play count is way larger than user count.

### Visualization

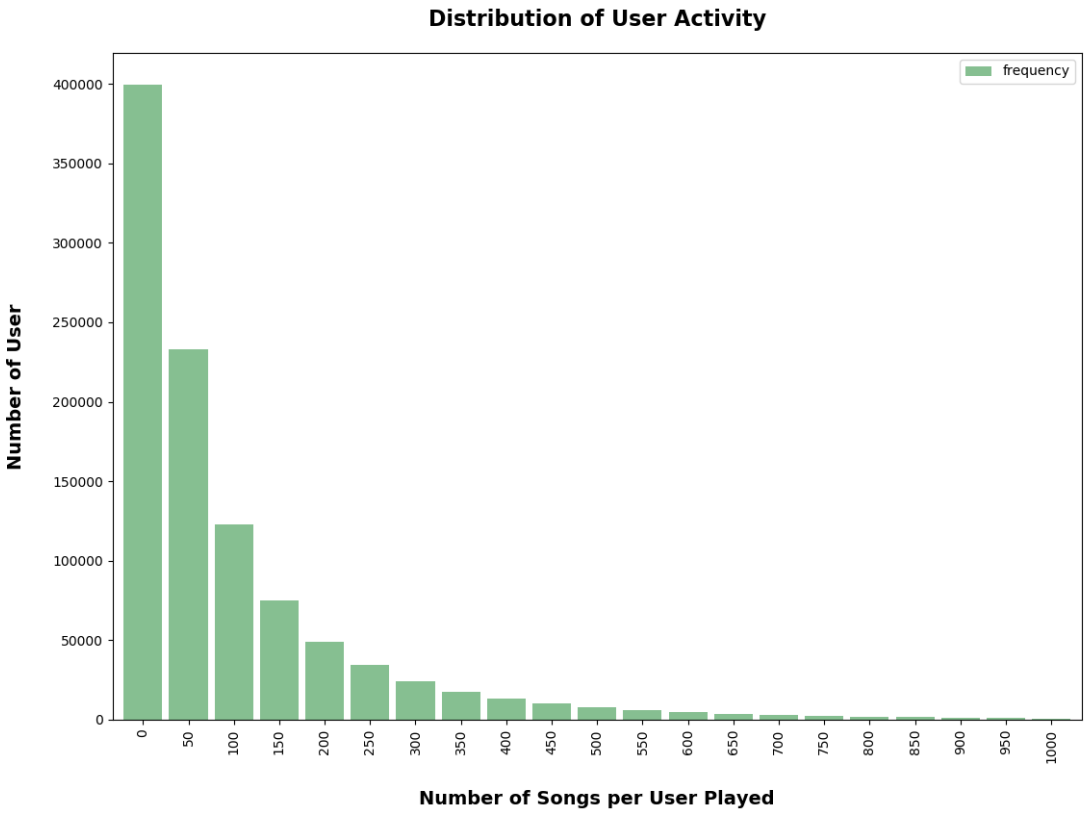


Figure 3.1 Distribution of user activity

Figure 3.1 gives us a broad idea of most users played less than 50 songs, which is almost 400,000 users, and the number of users amount who played between 50 and 99 songs, is only about half of it. If halves again from 100 to 149 songs. This is left skewed, which we can tell from our quantile analysis as well, 75% users played less than 73 songs, or played any songs less than 168 times.

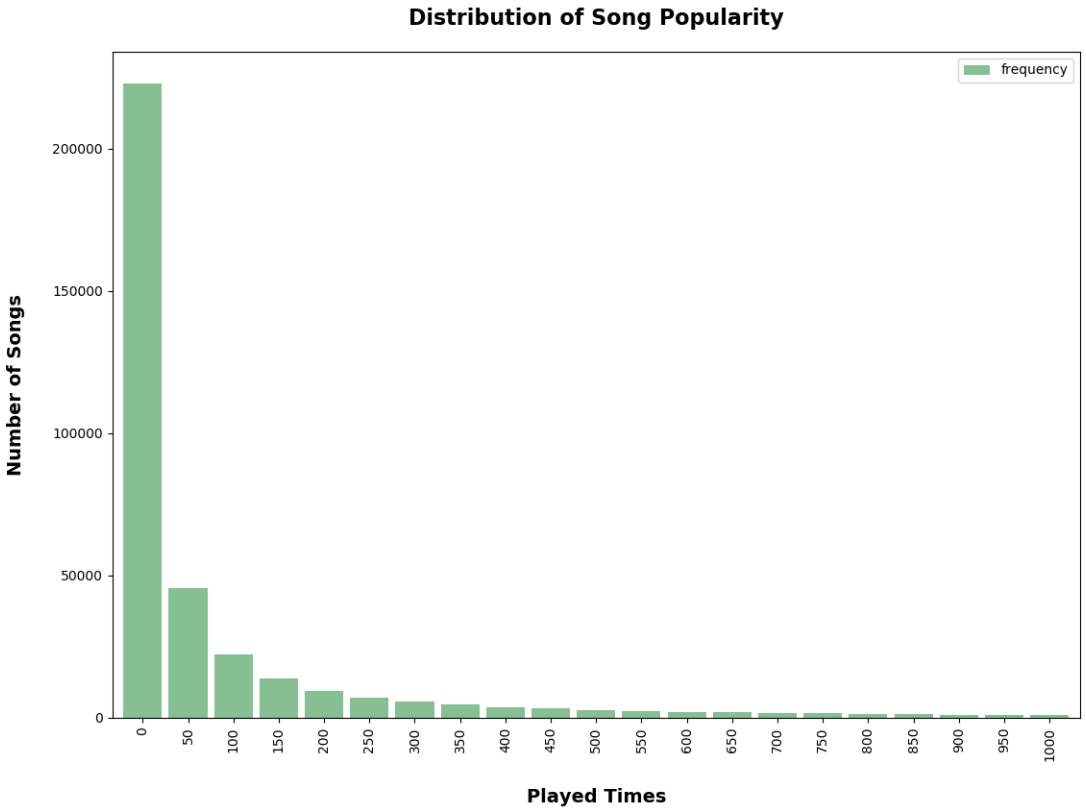


Figure 3.2 Distribution of song popularity

Figure 3.2 also shows the left skew shape on song popularity distribution. Most of them, which is more than 200,000 songs, are played less than 50 time, there are not even quarter of the songs are played by between 50 and 99 times. The quantile analysis also brought this point out, which 75% songs are played less than 107 times or by 58 users.

## SONG RECOMMENDATIONS Q1 (d)

### Clean dataset

To clean the dataset, by remove less active users and less popular song, to give us a more informative dataset, I removed users who listened less than 50 songs and songs are played less than 25 users. This choice is made upon the consideration of:

- There are more users in the dataset, if we throw out users who played less than 50 songs, we still get a reasonable amount of data remaining.

- Throw out users based on songs they play rather than played times, to filter out some users like a café, who will play a song a lot of times, but only with play fewer songs than normal users.

Finally, we get a dataset with 27,563,234 observations, 45,795,111 - 27,563,234 = 18,231,877 observation removed. Now, 75% of the users played less than 120 songs, not 73 songs, and 75% songs are played by less than 249 users, not 58 anymore. This dataset is a lot more informative now.

### SONG RECOMMENDATIONS Q1 (e)
### Split the dataset

When we split the dataset into training and test, if users in test but don't have records in training, after the model generate a matrix of user-latent, will have no corresponding vector for this user, and can't make a recommendation for this user. Collaborative filtering makes recommendations based on songs they played history by similar users.

The following step are in order to ensure the splitting is totally random and the above goal is achieved:

- Split the data randomly to 75% as training and 25% as test.
- Find out if there any observation in the test but not in training (1 observation).
- Randomly split the test not in training into 75% training and 25% test again by using window partition function again by user ID.
- There is no observation in the training partition, thus I just removed the 1 observation from the test dataset.

Now we have 20,675,406 observations in training, 68,87,827 observations in testing, and all test are in training.

### SONG RECOMMENDATIONS Q2 (a)
### Train collaborative filtering model

We encode the song ID column and user ID column to numeric for prediction, which the Alternating Least Square method only take numbers. Its rating is based on the play count. The implicit parameter of the method is also been turn on, since the play count is only an implicit feedback of the users, they didn't actually express which songs they like explicitly.

### SONG RECOMMENDATIONS Q2 (b)
### Compare the recommendation result

We set the k value quals 5 first, to give every user 5 recommendations, these are the top 5 recommendations, which sorted by corresponding prediction value, from most to least. Then we generate a list of relevant songs that each user played in the test dataset, and the songs are ordered by the times they played by each corresponding user, from most to least.

After we prepare both dataset for comparing, we combine them together by inner join on encoded user ID. I manually selected 3 users, by comparing the result, either 1 or none in the

relevant raking list, which we should do better than this, by removing less observations from the dataset, or using more comprehensive methods.

## SONG RECOMMENDATIONS Q2 (c)

### Collaborative filtering model evaluation metrics

| Evaluation Metric | Value | Evaluation Metric | Value |
|---|---|---|---|
| Precision @ 5 | 0.05441117 | Precision @ 10 | 0.02720559 |
| NDCG @ 5 | 0.05642663 | NDCG @ 10 | 0.03669889 |
| MAP @ 5 | 0.00661809 | MAP @ 10 | 0.00661809 |

Table 3.2 Collaborative filtering model evaluation metrics

#### Evaluation metrics

For precision evaluation metric, it doesn't take recommendation ranking into account, it just evaluates the fractions of correct recommendations have been made.
Both NDCG and MAP will take ranking position into account, e.g. each evaluation element in MAP is a precision at its k position. Based on same fractions of correct recommendations made, better ranking gets a higher score. While NDCG not just take ranking into account, it also taking if that correct recommendation is in the right ranking position as judgment as well.

By using Ranking Metrics module form pyspark.mllib.evaluation, we get all the metrics showing in table 3.2. Precision and NDCG are very similar, however, Map is much lower, this would tell us only a few of the recommendation we made was correct, maybe they are in the correct relative ranking position, but they always came a lot later, which respect to MAP metric. In addition, we can see, the precision and NDCG dropped by more songs are recommended. This shows there are more irrelevant songs are recommended.

#### Limitations

The metrics all have such limitations as I mentioned before, but what about the users not included in this model? Collaborative filtering has a "cold start" character, either if there is a new user, who has not played songs yet, or some users' play history is very limit, like what we move away from our dataset before, we won't be able to make recommendation on those users at all, or with very limited accuracy, if they haven't interacted enough.

#### Suggestions

We can use online and this offline recommendation system approach combined, we generate an offline recommendation for users, then put these online to be able to have certain interaction with the users, after a period of time, we can evaluate the recommendation system by collecting the data from user interaction, like which recommended songs they played, and how many times have they play. The evaluation metric can be generated based on their reactive play lists.

We also can suggest a A/B testing, which recommendations are introduced by two system. To do this, we randomly split the users as half and half. After online recommendation user interaction data collected, we can evaluate the two system, which one has more recommendation power.

### User-song based recommendation

If user-song based reacted information are gather, we can evaluate them more detailly, like whether they listen the song, yes or no, whether the whole length of the recommended song is played or not, the length can be measured as a portion from 0 to 1, and whether the song is played in more than one time, or how many times it played. Of course, we still can generate a new updated relevant play list of each users, and evaluate the recommendation system based on this list by generating another set of Precision MAP and NDCG metrics.

## Work Cited

[1] https://www.hadoopinrealworld.com/improving-performance-in-spark-using-partitions/

[2] https://www.quora.com/What-are-the-advantages-disadvantages-of-using-Gradient-Boosting-over-Random-Forests

[3] https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html

[4] https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html#sklearn.ensemble.RandomForestClassifier

[5] https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html#sklearn.ensemble.GradientBoostingClassifier

[6] https://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html#sklearn.svm.LinearSVC