CMRM Homework Assignment No. 1 (HWA1)

Veronica Coccoli – 10706276

Gregorio Secchi – 10680774

a.a. 2023/2024

1 Question 1

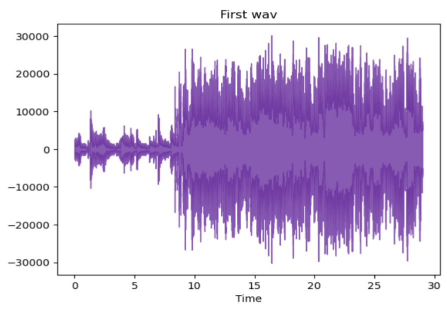
For the current project **GTZAN-genre**[1] dataset has been used. The mentioned dataset is widely used for evaluation in machine learning research for music genre recognition, that is giving SMC[2] researchers the opportunity to understand what differentiates a genre from another. The files were collected in 2000-2001 from a variety of sources including personal CDs, radio, microphone recordings, in order to represent a variety of recording conditions [1]. The dataset comes with the following features:

1. it provides a set of audio files respectively labeled with their corresponding genre, a visual representation of each audio file and 2 CSV files containing features gathered from audio tracks analysis;
2. the number of available tracks is 1000, and their data shape is `(1000, None, 1)`, while the genre tensor provides data with a shape of `(1000, 1)`;
3. reported musical genres are: *pop, metal, classical, rock, blues, jazz, hiphop, reggae, disco, country.* The mapping is therefore 100 audio files to 1 genre;
4. mapping: in accordance to the listing of the previous point, genres are labeled from 0 (pop), 1 (metal) to 9 (country). Such mapping could be easily done in the code below:

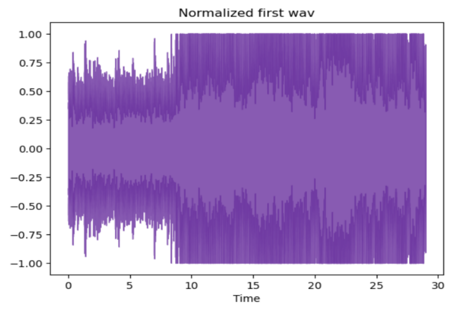
```
genre_names = ['pop', 'metal', 'classical', 'rock', 'blues
    ', 'jazz', 'hiphop', 'reggae', 'disco', 'country']
id2label = {id: label for id, label in enumerate(
    genre_names)}
```

5. file format: audio files are coded in **wav**, each track has data size of 1323632 bytes and a length of 30 seconds (in the current project, only 29 seconds will be taken into account);
6. sampling frequency is 22050 Hz;
7. bitrate: each audio file is a mono track of 16 bit-depth, at sampling frequency of 22050 Hz, meaning bitrate is 16 * 22050, that is 352.8 Kbps.

[1]`hub://activeloop/gtzan-genre`
[2]Sound and Music Computing



(a) Non-normalized wav.



(b) Normalized wav.

In this first part of the project, two arrays for training and two for testing have been computed, with respect to both audio and genre tensor. As a consequence, the resulting *numpy arrays* have the following shapes:

- genre_train: `(100, 1)`, meaning 100 rows containing one genre label (10 per genre);
- genre_test: `(20, 1)`, meaning 20 rows containing one genre label (2 per genre);
- audio_train: `(100, 639450)`, meaning 100 rows containing 639450[3] samples each;
- audio_test: `(20, 639450)`, meaning 20 rows containing 639450 samples each;

At the end of this section, the first element of `audio_train` is plotted (figure 1a)and eventually played in order to verify the correctness of loaded data.

2 Question 2

2.1 Preprocessing

In this second part, first step to carry out is applying preprocessing in order to normalize the dataset: for this purpose, a `sklearn.preprocessing.MinMaxScaler` [2] is used. Such object transforms features by scaling each feature to a given range, in this case `(-1, 1)`, as visible from the code below.

```
scaler = preprocessing.MinMaxScaler(feature_range=(-1, 1))
audio_train = scaler.fit_transform(audio_train)
audio_test = scaler.transform(audio_test)
```

While `fit()` method is used to compute mean and standard deviation later used for scaling, the `transform()` method actually performs scaling using the previously computed mean and standard deviation. Very intuitively, the `fit_transform()` method is a shortcut to execute both steps in one line of code.

One thing to remark is that a `fit_transform()` method is called on the train set, while test set only occurs with `transform()` method. That is because using the `fit` method on the test data too would otherwise compute a new mean and standard deviation, thus leading to a new scale for each feature.

[3]This number comes from the computation of 29 seconds * sampling_frequency

2.2 Define compute_local_average function

Local average is not a single measurement from the signal, yet a quantity which changes in time as the signal progresses. The aim of a function which computes local average is to transform the input signal $x[n]$ into an output $y[n]$. The output signal is obtained using a sliding window of fixed length, over which a mean of underlying samples is calculated.

The local average function indeed takes as input parameter the signal $x[n]$ and the length of the window in samples, returning the averaged signal.

2.3 Define principal_argument function

In a further step, a function `principal_argument` is defined, with the subsequent purpose of mapping phase differences in a range `[-0.5, 0.5]`. Such a mapping is useful because of the periodic nature of phase, leading to a discontinuity between two consecutive frames of a discrete signal when phase is wrapping.

2.4 Define compute_phase_novelty function

The phase of complex spectral coefficients can retrieve information for onset[4] detection. In particular, one can exploit the fact that stationary tones have a stable phase, while transients have an unstable phase [3]. To compute a phase-based novelty function, the following steps have been implemented:

1. compute the STFT[5]: onset detection methods rely on changes that can be observed in the signal's short-time spectrum. In this case, `librosa` package has been used for computing STFT.

2. compute the new Fs for novelty representation: the feature rate is given by sampling rate over hop size.

3. extract and normalize the phase: phase ϕ is extracted using `angle()`, a numpy built-in function. It is subsequently normalized by 2π.

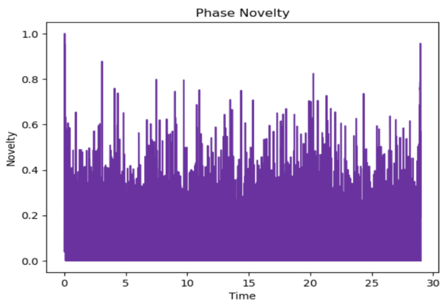
4. compute phase first derivative: the first-order phase difference may be written as $\phi' := \phi(n,k) - \phi(n-1,k)$, with k being a frequency and n the current sample; in the project, this computation is carried out using `diff()`, also a numpy built-in function. Principal argument function (see previous subsection 2.3) is then applied to avoid discontinuities due to phase wrapping. It is to remember that in a region where behaviour is steady, the frame-wise phase difference remains approximately constant.

5. compute phase second derivative: the second-order difference, defined as $\phi'' := \phi'(n,k) - \phi'(n-1,k)$, is also computed using `np.diff()`. It is to note that $\phi''(n,k)$ tends to be zero in steady regions of the signal (as ϕ' would be constant), while it shows an unpredictable behaviour in transient regions. As a result, second-order phase difference represents a good indicator for onset detection.

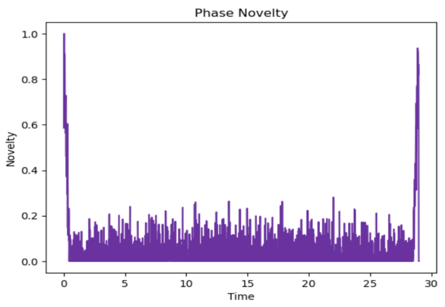
6. perform accumulation: as the phase-based novelty function can be defined as $\Delta(n) = \sum_{k=0}^{K} |\phi''(n,k)|$, the following code is executed:

[4]An onset is the earliest time point at which a transient can be reliably detected.

[5]*Short-Time Fourier Transform*



(a) Novelty with H=64



(b) Novelty with H=256

```
novelty = np.sum(np.abs(phi_double_prime), axis=0)
novelty = np.concatenate((novelty, np.array([0, 0])))
```

The code means a summation is performed along frequency axis, and the result's shape is later corrected by concatenating an array of two zeros.

7. subtract local average and apply half-wave rectification: this is a postprocessing step, intended for obtaining a novelty function with enhanced peak structure (and suppressed negligible fluctuations). Therefore, the local average of the novelty is subtracted from the novelty itself; in addition, only positive differences are taken into account through half-wave rectification.

8. apply normalization: if requested, the novelty function can be normalized by dividing it by its maximum value.

9. plot: if requested, the phase novelty can be displayed over time.

The function is ultimately tested on the first wav of the training set:

```
n, fs_n = compute_phase_novelty(audio_train[0], Fs, N=N, H=H,
    plot=True)
```

Based on various function calls made with different parameters, the length of the hop size H seems to be the parameter which most sensitively influences the novelty representation. In fact, the two figures above show the phase novelty computed on the first `audio_train` element, cast respectively with $H = 64$ (figure 2a) and $H = 256$ (figure 2b). Ideal hop size for computational velocity would be half of window length, but tests' results show that a small hop size increases accuracy level of the model.

The novelty representation results in being quite noisy; novelty function typically consists of impulse-like spikes, each indicating a note onset position: the fact that the plot is noisy means onsets are not clearly detected. Actually, the existence of various pulse levels such as measure, tactus, and tatum often makes the determination of real tempo difficult: it could be that there is a mismatch between actual pulse position and the detected onsets.

3 Question 3

Next step consist in defining and computing a `compute_feature_vector` function, taking as input parameters the signal x, its sampling frequency Fs, window length in samples N and hop size H.

The above mentioned function retrieves a numpy feature vector, containing rhythmic features that are going to be extracted from each audio file of both train and test set. In order to compute the feature vector, the following steps have been implemented:

1. compute the phase-based novelty function: the function discussed in subsection 2.4 is called, thus retrieving phase novelty n and its frequency *fs_n*. The novelty function will be later used to compute tempogram, and its sampling frequency will be used to analyze the periodicity of its behaviour.

```
n, fs_n = compute_phase_novelty(x, Fs, N, H)
```

2. compute the standard deviation and the mean of the novelty function: for this purpose, numpy built-in functions have been used.

```
n_std = np.std(n)
n_mean = np.mean(n)
```

3. compute the tempogram: a tempogram is a representation that encodes local tempo information; it indicates for each time instance t the local relevance of a specific tempo (in BPM) for a given music recording.

```
oenv = librosa.onset.onset_strength(y=n, sr=Fs)
tempogram = librosa.feature.tempogram(y=n, sr=fs_n,
    win_length=N, hop_length=H, onset_envelope=oenv)
```

4. compute the zero-crossing rate: the zero crossing rate (ZCR) measures how many times the waveform crosses the horizontal axis over a certain period. It can be obtained counting how many times the following condition is fulfilled for a signal $x[n]$, that is when $\{x(n) > 0$ and $x(n+1) < 0\}$ or viceversa. ZCR function is widely used in music information retrieval; indeed, this function can capture rhythmic patterns, since rapid changes in the signal (as indicated by frequent zero crossings) are likely to correspond to percussive elements or sharp transients.

```
zcr = librosa.feature.zero_crossing_rate(y=x,
    hop_length=H)
```

5. compute the standard deviation and the mean of the zero-crossing rate: as before, numpy functions have been used:

```
zcr_std = np.std(zcr)
zcr_mean = np.mean(zcr)
```

6. compute the spectral flux: spectral flux is a measure of the variability of the spectrum over time, that is for distinguishing signals whose spectrum changes slowly from signals whose spectrum changes quickly. Such a feature can be computed as the squared difference between the normalized magnitudes (audio frame to the power of two) of the spectra of the two successive short-term windows. The pseudo-code for this would be:

```
let curr_STFT \\current audio frame spectrum
let prev_STFT \\previous audio frame spectrum

curr_STFT = curr_STFT \ sum(curr_STFT) \\normalize
prev_STFT = prev_STFT \ sum(prev_STFT) \\normalize

flux = sum((curr_STFT - prev_STFT) **2) \\squared difference
```

However, in this project the computation was carried out by a `librosa` function:

```
spectral_flux = librosa.onset.onset_strength(y=x, sr=
    Fs)
```

7. compute the standard deviation and the mean of the spectral flux:

```
spectral_flux_mean = np.mean(spectral_flux)
spectral_flux_std = np.std(spectral_flux)
```

8. compute the tempo: for bpm detection, an updated version of `librosa.beat.tempo` function has been used, as that one seemed to be deprecated.

```
bpm = librosa.feature.rhythm.tempo(y=x, sr=Fs,
    onset_envelope=spectral_flux, hop_length=H)
```

The outcome is global tempo estimation. One could have set 'aggregate' parameter equal to `None`, in order to estimate tempo independently for each frame; however, for this dissertation, dynamic tempo is quite useless due to the reduced duration of audio files.

9. define the feature vector: `f_vector` combines all the features previously computed in one monodimensional vector, obtained through `np.concatenate()` function with `axis` parameter equal to `None`.

```
f_vector = np.concatenate([n, n_mean, n_std, tempogram
    , zcr, zcr_mean, zcr_std, spectral_flux,
    spectral_flux_mean, spectral_flux_std, bpm], axis =
     None)
```

Finally, two lists filled with feature vectors of both train and test set samples are computed. This computation is the most expensive of the project, as the feature vector `f_vector` has to be calculated on each audio sample.

To conclude, it is worth noticing that one may think computing as many rhythmic features as possible is good idea: indeed, it is when it comes to features that change a lot from a genre to another, like for what concerns the spectral flux. On the other hand, there might be a problem of overfitting: the model would be learning the training data too well, including the noise or specific patterns that don't generalize to new unknown data.

4 Question 4

In this further step, it is finally time to train a model. For this project, an SVM[6] classifier has been used. Such classifier separates data points using a hyperplane with the largest amount of

[6]Support Vector Machine

margin[7], thus clearly distinguishing one class from another. The core idea of SVM is to find a *maximum marginal hyperplane* [4] that best divides the dataset into classes. One more term it's worth focusing on is *support vectors*: those are the data points, which are closest to the hyperplane. These points will define the separating line better by calculating margins, therefore they happen to be the more relevant in class segregation.

Back to the project code, one has first of all to create an SVM classifier:

```
clf = svm.SVC(C=1, kernel='linear')
```

SVM uses a technique called the *kernel trick*. The kernel takes a low-dimensional input space and transforms it into a higher dimensional space. In other words, you can say that it converts non-separable problem to separable problems by adding more dimension to it. Based on the knowledge of the data being trained, there are several types of kernel one can choose for building the classifier:

- gaussian
- linear
- polynomial
- Laplace radial basis function
- sigmoid
- etc.

In the project, linear and rbf kernel have been used to train the model, as they're respectively conceived for dealing with large and inhomogeneous data vectors.

Always referring to the code reported above, there is one parameter that hasn't been discussed yet: the C parameter, also known as a *hyperparameter*. Most of the machine learning and deep learning algorithms let adjust some parameters which are indeed called hyperparameters, that play a critical role in setting training models. An accurate hyperparameter-tuning could prevent from overfitting and underfitting data, and act deeply on determining the decision boundary [5].

In particular, C parameter adds a penalty for each misclassified data point. If C is small, the penalty for misclassified points is low so a decision boundary with a large margin is chosen at the expense of a greater number of misclassifications. If C is large, SVM tries to minimize the number of misclassified examples due to high penalty which results in a decision boundary with a smaller margin, but that could lead to overfitting.

With non-linear kernels, also a *gamma* hyperparameter can be tuned, in order to change the way in which the decision boundary is built, either smoother or more precise.

Since for parameter tuning many tests are necessary, a method for loading the correct model (if present) or training a new one with new parameters has been developed using `pickle` library [6]:

```
if not os.path.exists('my_model/'):
os.mkdir('my_model/')

pkl_filename = f"my_model/svc_{kernel}_C_{C}_N_{N}_H_{H}"
```

[7]The margin is the distance between the either nearest points.

```
if not os.path.exists(pkl_filename):
    with open(pkl_filename, 'xb') as file:
        clf.fit(train_fvector, genre_train) #Train SVC
        pickle.dump(clf, file)
else:
    with open(pkl_filename, 'rb') as file:
        clf = pickle.load(file)
```

At this point, it's time to predict an output based on the feature vector extracted from test set computation. Once the model is trained, the `pickle` library leaves a `predict()` method at disposal, which requires a 2d input array to predict output values (in our case, genre labels), on the basis of the input (the feature vector).

```
genre_pred = clf.predict(test_fvector)
genre_pred_train = clf.predict(train_fvector)
```

Then, it is time to compare train and test set accuracy: the function which tests accuracy basically checks if the predicted labels (`genre_pred_train`, `genre_pred`) are coherent to actual labels (`genre_train`, `genre_test`).

```
print("Train Accuracy:",metrics.accuracy_score(genre_train,
   genre_pred_train)*100,"%")
print("Test Accuracy:",metrics.accuracy_score(genre_test,
   genre_pred)*100,"%")
```

The accuracy score computed with default parameters ($N = 2048, H = 128, C = 1, kernel =' linear'$) equals to 100% for train set and 25% for test set. Looking at these results, an hypothesis of data overfitting could be advanced, as accuracy on train model is very high, while the predictions on test set have high error rate. That means the model fits too closely to the training set, even though C parameter is already small, allowing some misclassifications.

To avoid overfitting, some countermeasures can be adopded, such as:

- **cross-validation**: the dataset could be devided into k subsets (also called *folds*); the model is then trained on $k - 1$ folds and tested on the remaining one. This process is repeated k times, each time using a different fold as the test set;
- **hyperparameter tuning**: as already said, a smaller C should prevent from overfitting (in cases different from linear kernel, it would make sense also to adjust gamma parameter);
- **feature selection**: it could be that some of the features computed in the feature vector are not relevant and just make the model learn noisy patterns.

5 Question 5

5.1 Tests

In order to assess which are the best parameters to later train the SVC model, an evaluation grid has been made. To estimate which are the best pairings, the grid search also uses cross-validation (in this case, with 5 folds).

```
param_grid = {'C': [0.001, 0.01, 0.1, 1, 10, 100], 'kernel': [
   'linear', 'rbf', 'poly', 'sigmoid']}
```

```
grid_search = GridSearchCV(svm.SVC(), param_grid, cv=5)
grid_search.fit(train_fvector,genre_train)
```

Results show that most suitable kernels and C hyperparameter combinations for the current SVC model would be: rbf[8] with $C = 100$, linear with $C = [0.001, 0.01, 0.1, 1, 10, 100]$ (all of those values would be good). However, evaluation grid is just an indicator, so the next step involves actually testing those combinations.

As observed in previous section (4), the trained SVC classifier performs with 25% accuracy on default parameters. Better results can be achieved, training the model with different parameters; for this reason, the program has been running with different configurations of N (window length), H (hop size), C (regularization parameter) and kernel parameters. A grid of accuracy score is reported below:

N	**H**	**C**	**kernel**	**train acc %**	**test acc %**
4096	32	1	linear	100	45
4096	32	100	linear	100	45
2048	64	1	linear	100	40
2048	128	1	linear	100	25
1024	32	1	linear	100	35
1024	32	0.001	linear	81	25
1024	64	1	linear	100	25
1024	128	1	linear	100	20

For non-linear kernels, also gamma parameter has been taken into account.

N	**H**	**C**	**gamma**	**kernel**	**train acc %**	**test acc %**
4096	32	100	0.001	rbf	100	60
4096	32	100	0.01	rbf	100	35
4096	32	1	0.001	rbf	95	20
2048	64	1	0.1	rbf	100	35
2048	64	1	1	rbf	100	20
2048	64	0.1	1	rbf	100	15
2048	64	0.001	0.01	rbf	100	30
2048	64	100	0.01	rbf	100	35
2048	64	1	0.001	rbf	71	25
1024	32	1	0.1	rbf	100	10

5.2 Evaluation of the model

The model is evaluated through a **classification report** and a **confusion matrix** as well.

```
#confusion matrix
disp = metrics.ConfusionMatrixDisplay.from_predictions(
    genre_test, genre_pred, xticks_rotation = 'vertical')
```

[8]Radial Basis Function

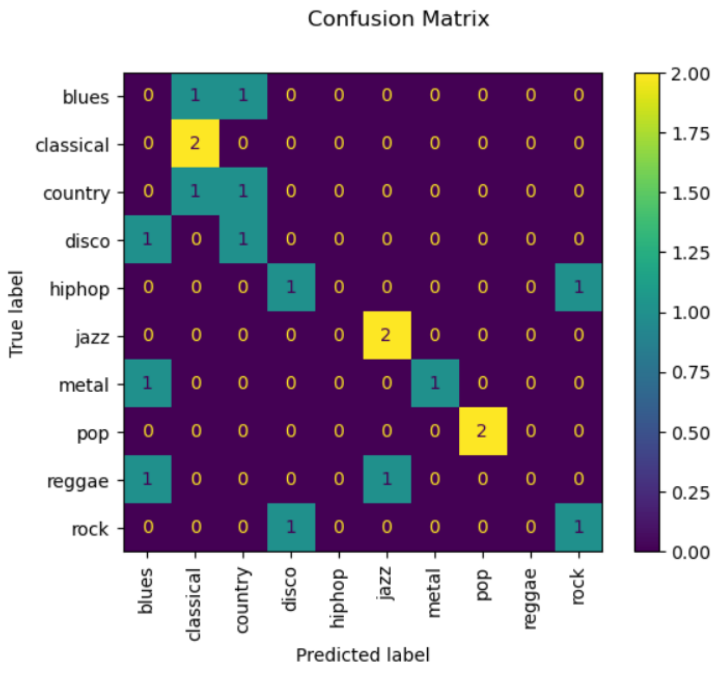
```
    disp.figure_.suptitle("Confusion Matrix")
    plt.show()
```

A confusion matrix allows to visualize the predicted labels against the true labels. For example, reading the confusion matrix of figure 3, it can be seen that one reggae track was mismatched with a blues label, or that one blues track was classified as classical, ans so on.

On the other side, the classification report builds a textual chart showing the main classification metrics, i.e.

- precision: ability of the classifier not to label as positive a sample that is negative;
- recall: ability of the classifier to find all the positive samples;
- F-measure: weighted harmonic mean of the precision and recall. An F-measure reaches its best value at 1 and its worst score at 0.

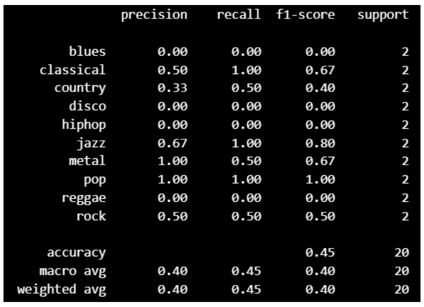
For instance, the report of figure 4 (which is calculated over the same model the confusion matrix of figure 3 was built on), claims the model comes with 0.67 precision and 1.0 recall on jazz audio files. That means that 100% of jazz tracks belonging to test set have been correctly labelled as "jazz" (those are the true positives), while out of the instances predicted as jazz, just 67% are true jazz audio files (the false positive rate is 33%).



True label \ Predicted label	blues	classical	country	disco	hiphop	jazz	metal	pop	reggae	rock
blues	0	1	1	0	0	0	0	0	0	0
classical	0	2	0	0	0	0	0	0	0	0
country	0	1	1	0	0	0	0	0	0	0
disco	1	0	1	0	0	0	0	0	0	0
hiphop	0	0	0	1	0	0	0	0	0	1
jazz	0	0	0	0	0	2	0	0	0	0
metal	1	0	0	0	0	0	1	0	0	0
pop	0	0	0	0	0	0	0	2	0	0
reggae	1	0	0	0	0	1	0	0	0	0
rock	0	0	0	1	0	0	0	0	0	1

Figura 3: Confusion matrix on test set with N=4096, H=32, C=1, kernel=linear.

```
    #classification report
    print(f"Classification report for classifier {clf}:")
    print(f"{metrics.classification_report(genre_test, genre_pred)
        }")
```



```
              precision    recall  f1-score   support

       blues       0.00      0.00      0.00         2
   classical       0.50      1.00      0.67         2
     country       0.33      0.50      0.40         2
       disco       0.00      0.00      0.00         2
      hiphop       0.00      0.00      0.00         2
        jazz       0.67      1.00      0.80         2
       metal       1.00      0.50      0.67         2
         pop       1.00      1.00      1.00         2
      reggae       0.00      0.00      0.00         2
        rock       0.50      0.50      0.50         2

    accuracy                           0.45        20
   macro avg       0.40      0.45      0.40        20
weighted avg       0.40      0.45      0.40        20
```

Figura 4: Classification report on test set with N=4096, H=32, C=1, kernel=linear.

Riferimenti bibliografici

[1] Kaggle website. `https://www.kaggle.com/datasets/andradaolteanu/gtzan-dataset-music-genre-classification`.

[2] Sklearn documentation. `https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html`.

[3] Meinard Müller. *Fundamentals of music processing: Audio, analysis, algorithms, applications*, volume 5. Springer, 2015.

[4] Svm tutorial. `https://www.datacamp.com/tutorial/svm-classification-scikit-learn-python`.

[5] Hyperparameter tuning. `https://towardsdatascience.com/hyperparameter-tuning-for-support-vector-machines-c-and-gamma-parameters-6a5097416167`.

[6] Pickle library. `https://docs.python.org/3/library/pickle.html#module-pickle`.