Piano Recognition in Classical Music Using MLP

Alina Dima (s3919951), Lisa Koopmans (s3933083),
Júlia Vághy (s3994759), Maria Kapusheva (s3946231)

July 4, 2021

Abstract

The present paper explores piano recognition in classical music recordings using a multi-layer perceptron (MLP). Training data was extracted from a dataset of MusicNet, and was preprocessed to contain features consisting of 46ms musical intervals defined by 13 mel-frequency cepstral coefficients (MFCCs), and binary labels indicating whether or not a piano is playing. Our MLP architecture consisted of 13 input nodes, a hidden layer of 9 nodes, and a single output node indicating an estimated probability of a piano playing. The hidden and output nodes were defined with sigmoid activations, and the learning algorithm used binary cross-entropy loss, L2 regularization, and early stopping. We used stratified 10-fold cross-validation to tune hyperparameters and account for imbalance in the data. The results on the optimized architecture shows that it performs relatively accurate, but there is room for improvement.

Keywords: piano recognition; multi-layer perceptron; binary classification

Contents

1 Introduction

Streaming services involving music processing such as Spotify and Deezer kept rising in popularity over the past decade. The high demand for such applications does not only stem from the access to music. Such services also provide recommendations and genre classification tuned to the preferences of their users based on the music classification algorithms. Similarly, dedicated music recognition services like Shazam are gaining traction by offering to recognize the name and the performer of a song while its playing. Moreover, similar functionalities have been implemented in social media applications and operating systems.

Despite that the algorithms in these applications achieve success in their tasks, there are still challenges when it comes to music recognition. One of those challenges is musical instrument recognition. The reason for this is that instruments can be played in various ways, or have different sound quality. Furthermore, if a recording is polyphonic, which is often the case, it creates an additional obstacle involving the separation of the different instruments (Yu, Duan, Fang, & Zeng, 2020). Although much research in this direction focuses on pop music, instrument recognition in classical music could be more complex due to the various types of instruments being used. Thus, we chose to focus on instrument recognition in classical music.

We obtained data from MusicNet, which contains 330 recordings of classical music with about 1.3 million labels. These labels consist of the start and end times for each note in each recording as well as the note's position and the music instrument playing it (Thickstun, Harchaoui, & Kakade, 2017).

Considering the fact that MusicNet's data is heavily imbalanced and foremost consists of piano plays, we aimed to create a model able to recognize a piano in a classical music recording. In line with previous research on instrument recognition, we utilized the mel-frequency cepstral coefficients (MFCC) (Eronen, 2001; Hall, Ezzaidi, & Bahoura, 2012). More on MFCC is explained in section 3. For our model, we used a multi-layer perceptron (MLP).

Due to the data we work with being imbalanced, we will evaluate our model based on the accuracy, precision-recall curves, a confusion matrix and the F_1 score obtained from a stratified k-fold cross-validation. Moreover, these metrics combined are appropriate for binary classification problems where no class is being prioritized.

2 Data

To train and test our model, we used the MusicNet dataset (Thickstun et al., 2017). The complete dataset consists of 330 recordings of classical music. Each recording is labeled for intervals of 11 different instruments playing specific notes with a given measure, beat, and note value. The total duration of the recordings is 2048 minutes, and 1346 minutes of this contain piano. Figure 1 shows the amount of minutes for each instrument. The labels have been verified by trained musicians, and have an error rate of 4%. The labels are encoded using digital MIDI scores, where an instrument is represented as a number, e.g. violin is encoded as 41. As we used Python to process the data and build our model, we used the NumPy representation of the dataset provided by MusicNet. This is directly downloadable from `https://homes.cs.washington.edu/~thickstn/start.html`. Here, the music itself is represented by arrays of pressure magnitudes sampled at a frequency of 44100 Hz, and labels (structured as aforementioned) are collected in interval trees. Generally, multiple instruments play at a given time, and a single instrument is likely to play multiple notes at once, which results in overlapping intervals.

To demonstrate the data we are working with, we will introduce an example. Suppose one wants to obtain information about Beethoven's String Quartet No 11 in F minor (recording ID 2494). We can choose to analyze the interval from 2.27s to 2.49s, for instance. Since the sampling frequency is 44100 Hz, this is indicated by the indices 100000 to 110000 in the data corresponding to the recording. From the feature, one can extract the array of pressure magnitudes on the analyzed interval using the indices. Thus, we obtain [0.02734375, 0.02468872, ..., 0.05358887, 0.05249023]. The recording's label is an interval tree, and its functionality allows us to extract all the musical information about the analyzed interval using the indices. We obtain an array of labeled intervals that overlap with or are contained in the analyzed interval, represented in Table 1.

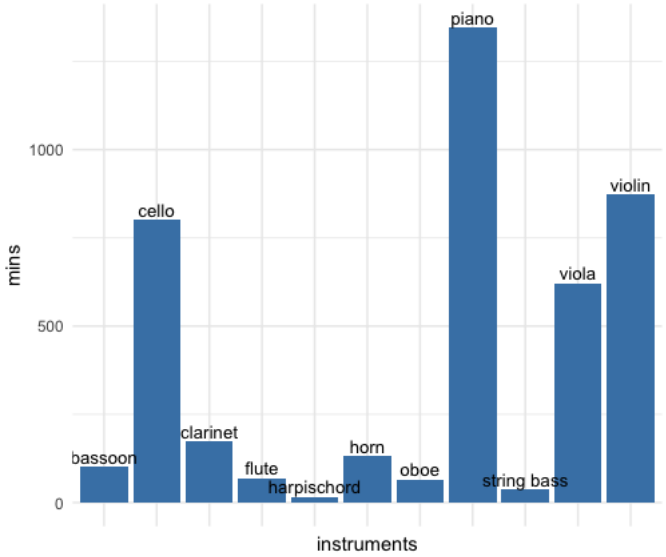

Figure 1: Distribution of minutes of recording for each instrument.

Start	End	Instrument	Note	Measure	Beat	Note value
98782	105438	41	62	1	0.75	Triplet
98782	105438	42	50	1	0.75	Triplet
98782	105438	43	38	1	0.75	Triplet
107998	114142	41	64	1	0.875	Triplet
107998	114142	42	52	1	0.875	Triplet
107998	114142	43	40	1	0.875	Triplet

Table 1: Example of data obtained from MusicNet (Thickstun et al., 2017).

3 Methods

3.1 Preprocessing

3.1.1 Motivation

To determine what features we would use, we had to make several decisions. First of all, we judged that it would be ideal to take small intervals of the recordings as training features, as the signal will be more or less constant (in terms of note) on these which makes them easier to analyze. Secondly, we had to determine what information to extract from the windows. One available format of the MusicNet data is an array of pressure magnitudes sampled at a frequency of 44100 Hz. First, we attempted to use this data extracted on each interval, however, the input size resulted in a complexity that proved to be too ambitious for our computational resources. Another option was to use frequency information for the feature extraction. One method for feature extraction using frequencies would be to obtain the spectrograms of the windows. Another would be to use energies in predefined frequency bands as input vectors. As spectrograms require more processing power and time, we decided to choose the latter option.

The next decision to be made was how to define the frequency bands. Specifically, whether we should use linearly spaced bands or ones spaced by psycho-acoustic criteria, such as the mel scale. Logan (2000) explains that mel-frequency cepstral coefficients (MFCCs) are generally used for speech recognition technologies, which does not necessarily imply that they would be suitable for modeling music. However, the author consequently proves that they are a suitable candidate, as explained in the next paragraph. Moreover, while reviewing the existing literature, we found that MFCCs appeared to be rather successful in musical instrument recognition. Eronen (2001) compared different features for instrument classification tasks and found that MFCCs lead to better performance than linear prediction cepstral coefficients. The authors found the best performance for 12 filterbanks. Similarly, Hall et al. (2012) found that MFCCs outperform linear predictive coding coefficients when using 13 filterbanks. Consequently, we arbitrarily chose to use 13-dimensional feature vectors of MFCCs, as more than satisfactory results were obtained with both

dimensions and we considered that either of those two would be appropriate.

So how does one obtain MFCCs and what makes them appropriate for musical instrument recognition? As Logan (2000) explains, one must first take the discrete Fourier transform of a window, thus obtaining the component frequencies of the sound. Although this method ignores phase, this should not be a problem for mono recordings such as those in our data. In mono recordings, the left and right channels receive identical signals, thus they are phase-independent. The next step is taking the logarithm of the amplitude spectrum and applying the mel filterbank. Using a logarithm is important for psycho-acoustic realism, as perceived loudness was found to be logarithmic. Again, to stay true to human sound perception, the mel frequency filters have unequal spacing, with higher frequency filters being wider. This is because, in human hearing, lower frequencies have more perceptual importance which results from the workings of the human cochlea, a spiral-shaped organ in the ear that sends nerve signals to the brain depending on the frequency of the incoming sound. The cochlea's ability to differentiate between similar frequencies decreases as the frequencies increase. Thus, spacing filters on the mel scale is equivalent to log spacing in the normal frequency scale, as demonstrated in Figure 2 (Logan, 2000). The mel filterbank energies are obtained by taking the weighted sum of the energies in each filter. Since filters overlap, the resulting energies are correlated. In order to decorrelate them, one must take the discrete cosine transform (DCT) of the resulting filterbank energies, which then gives the MFCCs. According to the experiments performed by the author, mel-spaced spectral features lead to at least equally as good or better performance as linear-based spectral features, and DCTs are appropriate for decorrelating musical spectra. Therefore, he concludes that MFCCs are indeed a suitable candidate for modeling music (Logan, 2000).

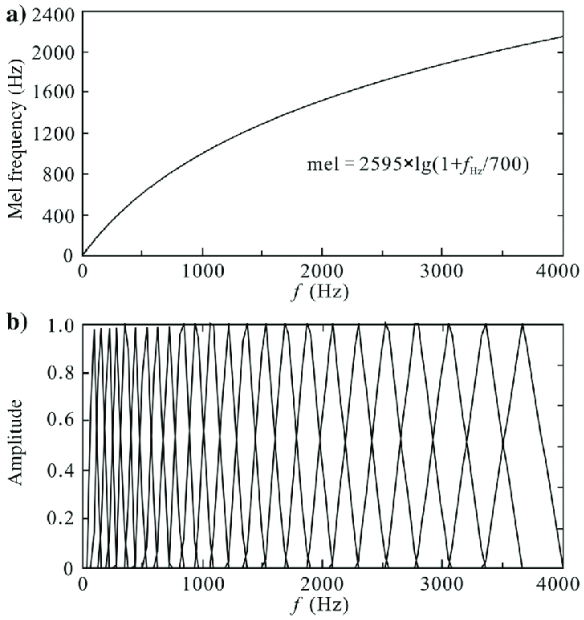

Figure 2: a) The Mel-frequency scale plotted against normal frequency. b) A 24-filter Mel filterbank. Figure obtained from Xie et al. (2017)

Therefore, our goal for preprocessing was to have 13-dimensional arrays as features, representing 13 mel-frequency cepstral coefficients on 46ms windows, and binary indicators as labels, denoting whether or not a piano is playing in the window. We chose this window size arbitrarily as it was also used in Thickstun et al. (2017). Their motivation was that this window size would be sufficiently small to also capture the relevant local information of the signal. As we obtained satisfactory results with this window size, we decided to not further experiment with other values.

3.1.2 Implementation

We can then define our preprocessing task as follows:

Given: For each of the 330 recordings, an array of pressure levels, sampled at a 44100 Hz frequency from the original audio, and an interval tree. The tree contains intervals defined by the specific instruments that play in that time span, the notes, the measures, the beats, and the note values.

Wanted: A training sample $S = (\mathbf{u_i}, y_i)_{i=1,\dots,N}$ where N is the number of samples, the input vectors $\mathbf{u_i} \in \mathbb{R}^{13}$ represent 13 MFCCs taken on 2048 windows of the pressure level arrays, and $y_i \in \{0,1\}$. The

target y_i has a value of 1 if a piano is playing in the analyzed interval and 0 otherwise.

In order to obtain the desired training sample we followed the next steps on each of the 330 recordings contained by MusicNet:

- **Step 1:** Extract the pressure level array and the interval tree of labels for the current recording.
- **Step 2:** Extract the MFCCs of all 2048 sample windows with a 1024 stride/shift between consequent windows.

  For this step we used the `mfcc` function from the `python_speech_features` library. The function was called on the pressure level array extracted in **Step 1** at the MusicNet sampling frequency of 44100 Hz. The parameters that we modified to fit our decisions were the dimension length of the window (in seconds), the step between windows (in seconds) and the size of the fast Fourier transform. As we were not experienced in working with MFCCs we decided to leave all other parameters related to this type of feature extraction to their default values in an attempt to work with "tried and true" values.

  The window dimension is the real duration t in seconds of a samples window of size N, where N is a natural number. We calculated the window dimension by dividing the number of samples in a window by the sampling frequency. Therefore, since MusicNet recordings were sampled at 44100 Hz and we chose to work on 2048 size windows, the window dimension t was 46ms in our case.

  The step between windows was calculated in a similar fashion to the window dimension. We divided the number of sample points between consequent intervals by the sampling frequency. Shifting the consecutive windows by 1024 sample points resulted in a step between windows of 23ms.

  The fast Fourier transform size was set to 2048 accordingly.

  Finally, as the default size of cepstral coefficients to return is 13, the function returned 13-dimensional MFCC arrays for the 2048 windows with 1024 stride.
- **Step 3:** Loop over all windows and find their corresponding labels.

  Using the functionality of the original interval tree labels, we obtained all the labels that are contained in the respective window. The interval tree representation facilitated getting the labels from all overlapping MusicNet intervals that contain the window in question.

  We then iterated through all found MusicNet labels. During the iteration, we extracted the instrument for each of the labels. If a value of 1 was found (MIDI instrument code for piano), we set the output of that window to 1, otherwise to 0.

3.1.3 Remarks

The recordings contain 1346 minutes of piano playing and the total duration of musical recordings is 2048 minutes, thus, there are 702 minutes of music in which no piano is present, as demonstrated in Figure 3. Since this is where our training sample originates, we are facing an imbalance of (1:1.92), meaning that for every example without a piano playing, there are approximately 1.92 examples with piano. Thus, as we extracted samples from all recordings using a constant stride, the imbalanced data will continue to be a challenge after preprocessing. An imbalanced class distribution may lead to predicting the majority class too often - in our case, the positive class. Our decision was to use stratified k-fold cross-validation to deal with the imbalance.

3.2 MLP architecture

Implementation: The MLP classifier was implemented using `TensorFlow` (including the `Keras` module).

To define our MLP architecture, we will first specify the input and output:
Input: The input consists of an array with MFCC features $\mathbf{u_i} \in \mathbb{R}^{13}_{i=1,\dots,N}$.
Output: The corresponding output consists of a label $y_i \in \{0,1\}_{i=1,\dots,N}$.
The entire training dataset can then be defined as $(\mathbf{u_i}, y_i)_{i=1,\dots,N}$.

The process of building the MLP for our binary classification problem can be viewed as being comprised of two main tasks:

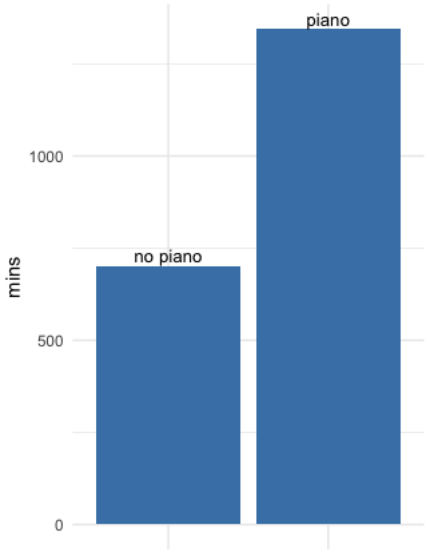

Figure 3: Minutes of recording without piano, and with piano, respectively.

- Settling on a weight optimization architecture, by choosing an optimizer, loss function, regularizer and activation function.
- Tuning the hyperparameters for optimal performance, either by trial and error or by using a hyperparameter optimization technique.

The rest of this subsection will be dedicated to addressing both of these tasks in detail.

3.2.1 Weight optimization architecture

- **Optimizer**

  For this MLP classifier we decided to use a gradient-based approach, as presented in the lecture notes.

  We used stochastic gradient descent to solve this problem. As explained in the lecture notes (2021), stochastic gradient descent is an iterative model optimization method. In each training epoch, the weights of the model are updated using the weights calculated at the previous epoch (Jaeger, 2021). We initialized the model weights with the Glorot normal initializer which samples from a truncated normal distribution, and set the initial bias weights to 1, as suggested in the lecture notes (Jaeger, 2021). During a training epoch, the algorithm finds the negative gradient $-\nabla R^{emp}(W_h)$ - the direction of steepest descent - at a given point on the performance surface of the risk function R^{emp} over the network weights W. At a given iteration, the weights $w_i \in W$ are nudged in this direction. The degree of change in an iteration depends on the learning rate μ (Jaeger, 2021). Thus, the weights W_{h_k} of model h_k in epoch k are given by the equation

$$W_{h_k} = W_{h_{k-1}} - \mu \nabla R^{emp}(h_{k-1}) = W_{h_{k-1}} - \mu \nabla (\frac{1}{N} \sum_{i=1}^{N} L(\hat{y}_{ik}, y_i) + \alpha^2 reg(W_{h_k})), \tag{1}$$

  where N is the number of training examples (x_i, y_i), $L(\hat{y}_{ik}, y_i)$ denotes the loss of the model output $\hat{y}_{ik}$ on the training example (x_i, y_i), and $\alpha^2 reg(W_{h_k})$ is the regularization term on the the set of weights W_{h_k} of the model h_k, scaled by the regularization weight α^2. The terms are explained in more detail in the paragraphs that follow.

  More specifically, the optimization approach that our model uses is minibatch stochastic gradient descent (SGD) with momentum.

  According to Chapter 8.1.3 of Goodfellow, Bengio, and Courville (2016), minibatch SGD implies that at each training step the gradient is calculated for a number of training samples. The weights are then updated based on the mean of these gradients. The alternatives would be using one training sample per step or using all training samples (the entire batch). We preferred using the minibatch

approach as it is less computationally expensive than SGD and keeping a small batch size could serve as a regularizing factor.

Additionally, our model uses momentum in an attempt to accelerate the learning of the gradient-based model. In short, the reasoning behind the concept of momentum in machine learning is that using a decaying average of past gradients to update the parameter space helps in maintaining a smoother direction (not as oscillating) towards local minima. Additionally, the weight of the average of past gradients is characterized by a momentum parameter $\alpha \in [0,1)$. Thus, decaying in time (Chapter 8.3.2 of Goodfellow et al. (2016)).

- **Loss**

  For the loss function choice, we consulted Goodfellow et al. (2016) once more. According to Chapter 6.2 of the aforementioned publication, mean squared error and mean absolute error do not tend to perform well with gradient descent optimizers. As an alternative the authors proposed the cross-entropy approach, which we will be using in our model.

  Our task is a binary classification task. Consequently, we used binary cross-entropy to obtain the loss. The binary cross-entropy loss $L(\hat{y}_i)$ for network output $\hat{y}_i$ with $i = 1,\ldots,N$ is calculated from the difference between the predicted probability $\hat{y}_i \in [0,1]$ and the training label $y_i \in \{0,1\}$, as seen in Equation 2 (Peltarion, n.d.).

$$L(\hat{y}_i, y_i) = \begin{cases} -log(\hat{y}_i), & \text{if } y_i = 1 \\ -log(1-\hat{y}_i), & \text{if } y_i = 0 \end{cases} \tag{2}$$

- **Regularizer**

  Minimizing the loss could result in overfitting. An overfit model does not generalize and thereby performs poorly on real data, since it will capture too much of the noise in the data instead of the underlying patterns (Jaeger, 2021). To prevent overfitting, we use a regularization term, which aims to downregulate the model's flexibility. As Karim (2020) explains, L1 regularization aims to push some weights towards 0, thereby reducing the number of features, while L2 regularization aims only to downregulate the magnitude of weights. Because all of our features are of importance to the output, we chose to use L2 regularization, also known as weight decay. According to Karim (2020), L2 regularization downregulates the squared value of the weights $w_i \in W_h$, W_h being the set of all model weights in model h, as seen in Equation 3. The full L2 regularization term in the optimization problem is scaled by the regularization weight α^2, as seen in Equation 5.

$$reg(W_h) = \sum_{w_i \in W_h} (w_i)^2 \tag{3}$$

  An additional regularization method that we used is early stopping. In the lecture notes (2021), early-stopping was described as a method which would allow us to stop training as soon as the performance of our model would not be increasing anymore. As a metric for checking performance we chose the loss and not the accuracy. This choice was mostly arbitrary, as we have noticed that most of the times the accuracy increased with a decrease in loss. Therefore, we thought the difference between using one or the other would not be considerable. For this type of regularization we will also have to select a patience (the number of epochs after which training will stop when no performance increase is observed). The tuning of this parameter is discussed in the next subsection.

- **Activation**

  As the binary cross-entropy deals with probability distributions, the output neuron in the last layer should have a value in the interval [0, 1]. Therefore, the logistic sigmoid function defined in Equation 4 appeared to be an appropriate choice, as the domain of this function is $\mathbb{R}$ and the co-domain is the interval [0, 1]. Additionally, the function $exp(x)$ from Equation 4 returns the natural exponential of the real number x.

$$\sigma(x) = \frac{1}{1+exp(-x)} \tag{4}$$

  Although, this type of activation was required only for the output layer, we decided to apply it to the hidden layers as well in an attempt to preserve simplicity.

To conclude, the optimization problem to solve is then finding the model h_{opt} that minimizes the empirical loss R^{emp}. In our case, this is defined as minimizing the binary cross entropy loss between outputs $\hat{y_i}$ and training labels y_i and the L2 regularization penalty $\alpha^2 reg(W_h)$.

$$h_{opt} = argmin(R^{emp}(W_h)) = argmin(\frac{1}{N}\sum_{i=1}^{N} L(\hat{y_i}, y_i) + \alpha^2 reg(W_h)) \tag{5}$$

3.2.2 Hyperparameter tuning

- **Number of hidden layers and neurons**

  As the input has a dimensionality of 13, and the output is a label for one of the possible binary outcomes, we defined a MLP with 13 input nodes and a single output node. Additionally, after experimenting for the number of nodes in the hidden layer and the number of hidden layers, we defined one hidden layer containing 9 nodes, as this brought about the best performance.

- **Learning rate**

  The learning rate μ of the stochastic gradient descent also requires careful consideration. A learning rate that is too small leads to slow convergence, which is far from ideal if the available computational resources are limited, like in our case. On the other hand, a too large μ may lead the gradient descent process to take steps that are too large and thereby miss local minima. After experimentation with the learning rate in the range [0.001, 0.5], we set it to $\mu = 0.01$.

- **Regularization weight**

  It is crucial to choose α^2 carefully, as a too high value may lead to an underfit model that does not capture important characteristics of the data, while a too low value may fail to counter overfitting (Jaeger, 2021). Therefore, our strategy was choosing the smallest value that would prevent overfitting. After experimenting with multiple values, we found that a value of 0.2 performed best.

- **Momentum parameter** Chapter 8.3.2 of Goodfellow et al. (2016) mentions that some commonly used momentum parameters α are 0.5, 0.9 and 0.99. We then experimented with $/alpha$ around these values and obtained the best results for $\alpha = 0.95$.

- **Number of epochs**

  The number of epochs and patience were the last hyperparameters that we tuned. During the final trial runs of the model we ran the classifier for 100 epochs. We then noticed that the model converged at around 10 epochs. We decided to reduce the number of epochs, as we did not want the classifier to get trained for too many epochs after convergence to avoid overfitting. The number of epochs was not reduced to 10 though, as we considered that the model should have some flexibility and account for differences between runs. Therefore, we set the number of epochs to 30, but with the addition of early stopping.

- **Patience**

  For determining the patience of early stopping we tried multiple natural numbers in the range [3, 15]. This parameter was eventually set to 5, as this value lead to the best performance.

- **Batch size**

  We experimented with the batch size according to the guidelines found in Chapter 8.1.3 of Goodfellow et al. (2016), using powers of two ranging from 32 to 256. The batch size was set to the hyperparameter resulting in the best performance, namely 32.

3.3 Stratified k-fold cross-validation

To ensure that the performance of our classifier will be generalizable, we implemented a k-fold cross-validation scheme. The reasoning behind k-fold cross-validation is splitting the data into k folds. The next step would be iterating through all of the folds using each of them as testing/validation data, while the other k-1 folds are used as training data.

As we were dealing with imbalanced data and did not want to lose data through undersampling or increase the computational expenses, we decided to deal with the imbalance by using stratified k-fold cross-validation. Stratified k-fold validation ensures that each of the folds has (about) the same majority/minority

ratio as the full dataset. We decided to use this technique using 10-folds, as recommended in the conclusions of He and Ma (2013).

To implement the cross-validation scheme, we used the `StratifiedKFold` function from the `model_selection` module of `sklearn`.

4 Results

After obtaining our final model, we evaluated our model using stratified k-fold cross-validation. For the number of folds k, we chose $k = 10$. so, the processed data was split into 10 folds and 10 training and testing sets were created as well as 10 models. This number of folds would allow us to have a general estimate of our model performance while limiting the necessary processing power and time. We then evaluated our model based on multiple metrics appropriate to use for imbalanced data and for classifiers. These metrics are the accuracy, F_1 score, precision-recall curves (PR-curves) and a confusion matrix. This combination of metrics also does not prioritize one class over another. We obtained the accuracy and F_1 scores by averaging over all folds. The PR-curve over all folds and the confusion matrix are obtained from concatenating all predicted labels from the test sets and the correct labels from the test sets.

4.0.1 Qualitative analysis

First, we can observe what occurred during training regarding the loss and accuracy. In Figure 4 we can see that the accuracy converged to 0.810. In Figure 5 we can see that the loss converged to about 0.452. Moreover, both figures indicate that this convergence occurs relatively quick, as values are reached after about 3 epochs.

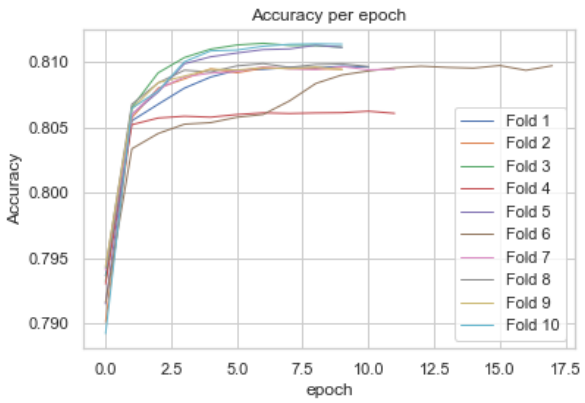

Figure 4: Accuracy on training data per epoch for all folds in stratified 10-fold cross-validation.

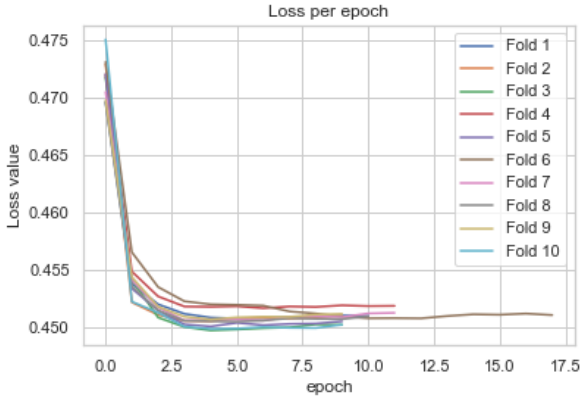

Figure 5: Loss of training data per epoch for all folds in stratified 10-fold cross-validation.

4.0.2 Quantitative analysis

The model's mean accuracy was 0.811 with a standard deviation of ± 0.011, meaning that the model predicted 81.1% of the test data correctly across all folds.

In the confusion matrix shown in Figure 6 we can see the true negative (TN), false negative (FN), false positive (FP) and true positive (TP) counts over all folds. We can see that the dataset is indeed imbalanced, as there are more TP than TN, while the number of FP and FN are relatively close to each other. From the confusion matrix, we can obtain the false negative rate (FNR) (Luque, Carrasco, Martín, & de las Heras, 2019).

$$FNR = \frac{FN}{FN+TP} = \frac{486728}{486728+2725774} = 0.152$$

Likewise, we can obtain the false positive rate (FPR) (Luque et al., 2019).

$$FPR = \frac{FP}{FP+TN} = \frac{511919}{511919+1566920} = 0.246$$

Thus, our classifier has a FNR of 0.152 and a FPR of 0.246, meaning that 15.2% of the examples that should have been classified as 'Piano' were wrongfully classified as 'No Piano' while 24.6% of the examples that should have been classified as 'No Piano' were wrongfully classified as 'Piano'.

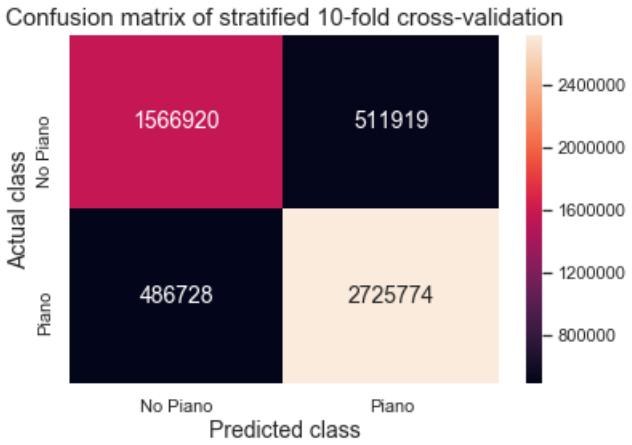

Figure 6: Confusion matrix for stratified 10-fold cross-validation over all folds. Upper left: TN, lower left FN, upper right: FP, lower right: TP.

The PR-curves per fold and over all folds are shown in Figure 7. The blue line indicates the PR-curve over all folds while the gray dashed line indicates PR-curve when the classifier would predict all examples as Piano. Model performance is considered optimal when the PR-curve has a precision of 1.0 when the recall is also 1.0, i.e. when the curve is in the uppermost right. Since the PR-curve over all folds is in the upper right, our model performs relatively well. This is supported with the Area Under the Curve (AUC) score, which is $AUC = 0.81$ over all folds.

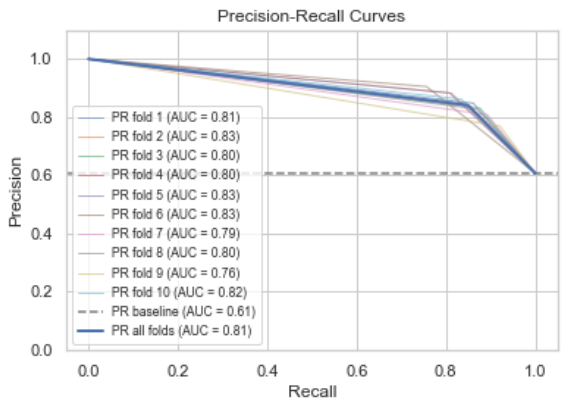

Figure 7: PR-curves per fold and over all folds (blue) for stratified 10-fold cross-validation.

The F_1 score is computed with

$$F_1 = 2 * \frac{PRC * SNS}{PRC + SNS}$$

where

$$precision = PRC = \frac{TP}{TP+FP}$$
$$sensitivity = SNS = \frac{TP}{TP+FN}$$

In these formula, TP is the number of true positives, FP the number of false positives and FN the number of false negatives (Luque et al., 2019). The model's mean F_1 score was 0.845 with a standard deviation of ± 0.008. Considering the fact that the maximum F_1 score is 1.0, our model's score supports the earlier statement that our model performs relatively well.

5 Discussion

5.1 Conclusions

In this project we aimed to create an MLP able to recognize whether a piano was playing in a classical music recording. We obtained our data from MusicNet and preprocessed it using MFCCs. We used stratified 10-fold cross-validation on training data to tune our hyperparameters and tested our model accordingly.

Quantitative analysis showed our model converges to a loss and accuracy quickly. This is possibly due to our learning rate being higher than what would be optimal for our particular problem. Therefore, it would be good to experiment with lower learning rates.

Our model correctly predicted 81.1% of the examples. However, we need to consider the fact that the dataset was imbalanced. Of the examples actually labelled with 'Piano', 15.2% was wrongfully classified. Of the examples actually labelled with 'no piano', 24.6% were wrongfully classified. This difference was to be expected due to the majority of the data being labelled as 'piano'. These FNR and FPR also indicate that our model correctly classified examples as piano when a piano was actually playing more often than examples as no piano when no piano was actually playing. Additionally we obtained the PR-curve across all folds as well as its AUC and F_1 scores. All three of these measures indicated our model performed relatively well, however, there is still room for improvement.

5.2 Reflection

The process of obtaining our model and to appropriately preprocess MusicNet's data included various challenges to overcome, which turned out to be more complex than we initially anticipated. Due to our lack of experience in working with large datasets, we had to spent much time on discovering the data's structure and how to access everything we required. We also had to do additional research as we had to uncover how to work with imbalanced data and which methods to use for preprocesing the data. Creating the neural network also proved to be challenging. Both the architecture and the hyperparameters required much thought and experimenting, however, this allowed us to deepen our understanding of them as well.

5.3 Improvements and Future Directions

One improvement to our model could be how we preprocess our data. For instance, we could apply a different method to handle the data imbalance. Currently, we use stratified k-fold cross-validation, however we still train the model on imbalanced data. Techniques for oversampling or undersampling could be considered, for example. As the data we worked with is relatively large, undersampling might be a more realistic option, however it risks the loss of important data. Oversampling would be ideal if the necessary computational power is available.

Regarding the hyperparameters, we could possibly further decrease the learning rate to prevent the fast convergence of the loss and accuracy we observed. It is possible that with our current learning, steps are taken in the gradient descent process which might be slightly too large and end up in a suboptimal local minimum. One method we could use to find the optimal learning rate is to start with a low learning rate and to slowly increase it. By observing the loss, the optimal learning rate could be determined.

Several options that could lead to better model performance exist to experiment with in future research. The process through which we preprocessed our data contains parameters that could be varied and optimized. For instance, one could experiment with different numbers of mel filters, different ranges of

observed frequencies, different window sizes, or applying filters to the windows, such as the Hamming filter. More generally, one could approach this task in completely different ways. For example, by creating spectrograms on the given windows and using convolutional neural networks for classification. Although spectrograms are more computationally complex than MFCCs, they contain information about how the signal varies over the span of the window which could be valuable for larger window sizes.

References

Eronen, A. (2001). Comparison of Features for Musical Instrument Recognition. In *Proceedings of the 2001 IEEE Workshop on the Applications of Signal Processing to Audio and Acoustics (Cat. No.01TH8575)* (p. 19-22). doi: 10.1109/ASPAA.2001.969532

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press. (`http://www.deeplearningbook.org`)

Hall, G., Ezzaidi, H., & Bahoura, M. (2012, 12). Study of Feature Categories for Musical Instrument Recognition. *Communications in Computer and Information Science*, *322*, 152-161. doi: 10.1007/978-3-642-35326-0_16

He, H., & Ma, Y. (2013). *Imbalanced learning: Foundations, algorithms, and applications* (1st ed.). Wiley-IEEE Press.

Jaeger, H. (2021, March). *Lecture Notes in Neural Networks (AI).* Faculty of Science and Engineering, University of Groningen.

Karim, R. (2020, Oct). *Intuitions on L1 and L2 Regularisation.* Towards Data Science. Retrieved from `https://towardsdatascience.com/intuitions-on-l1-and-l2-regularisation-235f2db4c261`

Logan, B. (2000). Mel Frequency Cepstral Coefficients for Music Modeling. In *In International Symposium on Music Information Retrieval.* doi: 10.1.1.11.9216

Luque, A., Carrasco, A., Martín, A., & de las Heras, A. (2019). The Impact of Class Imbalance in Classification Performance Metrics Based on the Binary Confusion Matrix. *Pattern Recognition*, *91*, 216-231. Retrieved from `https://www.sciencedirect.com/science/article/pii/S0031320319300950` doi: https://doi.org/10.1016/j.patcog.2019.02.023

Peltarion. (n.d.). *Binary Crossentropy* . Retrieved 2021-07-04, from `https://peltarion.com/knowledge-center/documentation/modeling-view/build-an-ai-model/loss-functions/binary-crossentropy`

Thickstun, J., Harchaoui, Z., & Kakade, S. (2017). *Learning Features of Music from Scratch.*

Xie, T., Zheng, X., & Zhang, Y. (2017, 05). Seismic Facies Analysis Based on Speech Recognition Feature Parameters. *GEOPHYSICS*, *82*, O23-O35. doi: 10.1190/geo2016-0121.1

Yu, D., Duan, H., Fang, J., & Zeng, B. (2020). Predominant Instrument Recognition Based on Deep Neural Network With Auxiliary Classification. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, *28*, 852–861.