Surgical mask detection

Discriminate between utterances with and without surgical mask

Kaggle competition link: https://www.kaggle.com/c/ml-fmi-23-2020

Project repository link: https://github.com/Natasa-C/Surgical-mask-detection

Table of Contents

Data Description

Task

Participants have to train a model for surgical mask detection. This is a binary classification task in which an utterance (audio file) must be labeled as without mask (label 0) or with mask (label 1).

The training data is composed of 8000 audio files. The validation set is composed of 1000 audio files. The test is composed of 3000 audio files.

File Descriptions

- train.txt - the training metadata file containing the audio file names and the corresponding labels (one example per row)
- validation.txt - the validation metadata file containing the audio file names and the corresponding labels (one example per row)
- test.txt - the test metadata file containing the audio file names (one sample per row)
- sample_submission.txt - a sample submission file in the correct format

Data Format

Metadata Files The metadata files are provided in the following format based on comma separated values:

- 1.wav,0
- 2.wav,1

Each line represents an example where:

- The first column shows the audio file name of the example.
- The second column is the label associated to the example.

Audio Files

The audio files are provided in .wav format.

Research and code implementation

In the documentation file, I used for exemplification fragments of the code I implemented to solve the task.

[1] Setting the data paths. Reading and storing the data.

I created a `MaskDetection` class in which I defined all the data variables and methods required to solve the mask detection challenge.

[1.1] Setting the paths to data

I started by defining the paths to the data that had to be loaded for processing and for the data that had to be written out after processing.

```
class  MaskDetection():
...
    TRAIN_DATA_PATH = './ml-fmi-23-2020/train/train'
    VALIDATION_DATA_PATH = './ml-fmi-23-2020/validation/validation'
    TEST_DATA_PATH = './ml-fmi-23-2020/test/test'
    ...
```

[1.2] Reading and storing the audio .wav files

I read the audio files and stored them locally.

```
def  readData(self):
    ...
    self.train_data = []
    self.train_names = []

    # read train data
    for filepath in tqdm(glob.glob(self.TRAIN_DATA_PATH + '/*')):
        data, sr = librosa.load(filepath, sr=self.sr)
        self.sr = sr
        self.train_names.append(os.path.basename(filepath))
        self.train_data.append(data)

    self.train_data = np.array(self.train_data)
    ...
```

[1.3] Reading and storing the labels for the .wav files

After that, I read each label for the audio files, matched it with the corresponding audio name and stored it locally.

```
def  readLabels(self):
    ...
    # read train labels
    fd = open(self.TRAIN_LABELS_PATH, 'r')
    self.train_labels = [0] * len(self.train_data)

    for line in tqdm(fd.readlines()):
        name = line.split(',')[0]
        if name in  self.train_names:
            self.train_labels[self.train_names.index(name)] = (int(line.split(',')[1])

    fd.close()
    self.train_labels = np.array(self.train_labels)
    ...
```

[2] Preprocessing: cleaning the data

To remove the redundant parts of the audio files, I built an `envelope` function and I used it to create True/False masks (envelopes) to keep only the relevant parts of the signal from the .wav files.

[2.1] Creating an envelope function

I built an envelope function used to create a mask with True/False values which will be used to reduce the empty/under the threshold portions of data. The function follows the next steps:

- convert the numpy array into a series and transform each value in the series into it's absolute value
- create a rolling window over the signal with pandas which provides rolling window calculations and get the mean of the window (window = window size is going to be a tenth of a second which translates to a tenth of the collection rate samples (we have 44100 samples/second, so in a tenth o a second, we go over a tenth of them), min_periods = the minimum number of values that we need in our window to create a calculation, center = center the window)

```
def  envelope(self, signal, rate, threshold):
    # create a mask with True/False values which will be used
    # to reduce the empty/under the threshold portions of data
    mask = []
    y = copy.deepcopy(signal)
    # convert the numpy array into a series and transform each value
    # in the series into it's absolute value
    y = pd.Series(y).apply(np.abs)
    # create a rolling window over the signal with pandas
    # which provides rolling window calculations
    # and get the mean of the window
    y_mean = y.rolling(window=int(rate/10),min_periods=1, center=True).mean()
    # create the True/False mask based on the threshold
    for mean in y_mean:
        if mean > threshold:
            mask.append(True)
        else:
            mask.append(False)
    return mask
```

[2.2] Cleaning the data

This function is going to create True/False masks (envelopes) to keep only the relevant parts of the signal from the .wav files using a `threshold = 0.0005` and write the created clean files into the specified folder. Once the data has been cleaned, we changed the paths to the data to point to the clean files.

If the folders in which the clean files are stored are empty, then the .wav files are cleaned. Otherwise, the function will specify that the files have been cleaned before and the cleaning process is no longer required. If I want to clean the files again, I would remove the old files from the folder and then run the script. As there will be no files in the specified directory, the cleaning process will start.

```
def  cleanData(self):
    threshold = 0.0005
    if  len(os.listdir(self.TRAIN_DATA_PATH)) == 0:
        # clean train data
        for filepath in tqdm(glob.glob(self.TRAIN_DATA_PATH + '/*')):
            name = os.path.basename(filepath)
            data, sr = librosa.load(filepath, sr=self.sr)
            mask = self.envelope(data, self.sr, threshold)
            wavfile.write(self.TRAIN_DATA_PATH + name,
            rate=sr, data=data[mask])

        # clean validation data
        ......
    else:
        print('\nAttention: Data files have been cleaned before.
        If you want to clean them again, try removing old folder files
        and then clean again.')

    # once the data has been cleaned, we changed the paths to the data to point
    # to the clean ones
    self.TRAIN_DATA_PATH = self.CLEAN_TRAIN_DATA_PATH
    self.VALIDATION_DATA_PATH = self.CLEAN_VALIDATION_DATA_PATH
    self.TEST_DATA_PATH = self.CLEAN_TEST_DATA_PATH
    self.CSV_FILES_FOLDER_PATH = self.CLEAN_CSV_FILES_FOLDER_PATH
```

Resources:

- [envelope function] https://www.youtube.com/watch?v=mUXkj1BKYk0&t=289s
- [series/rolling window] https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.html

[3] Audio spectrogram and feature extraction

> A spectrogram is a visual way of representing the signal strength, or “loudness”, of a signal over time at various frequencies present in a particular waveform. Every audio signal consists of many features from which we must extract the characteristics that are relevant to the problem we are trying to solve. The spectral features (frequency-based features) are obtained by converting the time-based signal into the frequency domain using the Fourier Transform. source

The functions used to extract the features come, mainly, from `librosa.feature` and `numpy` library.

The features are extracted in separate directories for the raw data and for the clean data and stored in .csv files. By making this, I reduced the preprocessing time: instead of extracting the features every time I run the script, I extract them once, at the first running, and store them in .csv files. At the next execution, I will use the data already extracted, so I reduce the feature extraction time expenses as long as I do not want to extract more features, case in which I have to rerun the script and extract the desired data. The general features are stored separately from the mfcc values to make the loading of the extracted data more intuitive and more flexible regarding the number of features extracted.

[3.1] Creating a function to extract frequency and magnitude

```
def  get_fft(self, y, rate):
    n = len(y)
    frequency = np.fft.rfftfreq(n, d=1/rate)
    magnitude = abs(np.fft.rfft(y)/n)
    return (magnitude, frequency)
```

[3.2] Extracting features

```
def  extractFeaturesForDataSet(self, set_of_data, names_for_data, filename):
    # create the header for the csv file
    to_append = f'filename,mean_spectral_centroids,mean_spectral_rolloff,
    mean_spectral_bandwidth_2,sum_zero_crossings,mean_mfccs,'
    to_append += f'mean_magnitude,mean_freq,mean_chroma_stft,mean_rms,mean_bank,
    mean_mel,mean_spectral_contrast,mean_chroma_med,mean_melspectrogram,
    mean_flatness'

    g = open(filename, 'w')
    g.write(to_append)
    g.close()

    g = open(filename, 'a')

    for index in tqdm(range(len(set_of_data))):
        data = set_of_data[index]

        spectral_centroids = librosa.feature.spectral_centroid(y=data, sr=self.sr)
        spectral_rolloff = librosa.feature.spectral_rolloff(y=data, sr=self.sr)
        spectral_bandwidth_2 = librosa.feature.spectral_bandwidth(y=data, sr=self.sr)

        zero_crossings = librosa.zero_crossings(data)
        mfccs = librosa.feature.mfcc(y=data, sr=self.sr)
        chroma_stft = librosa.feature.chroma_stft(y=data, sr=self.sr)

        rms = librosa.feature.rms(y=data)
        bank = logfbank(data[:self.sr], self.sr, nfilt=26, nfft=1103)
        mel = mfcc(data, self.sr, numcep=13, nfilt=26, nfft=1103)
        spectral_contrast = librosa.feature.spectral_contrast(data, sr=self.sr)
        magnitude, freq = self.get_fft(data, self.sr)

        chroma_cqt = librosa.feature.chroma_cqt(y=data, sr=self.sr)
        chroma_med = librosa.decompose.nn_filter(chroma_cqt, aggregate=np.median,
        metric='cosine')

        melspectrogram = librosa.feature.melspectrogram(y=data, sr=self.sr)
        flatness = librosa.feature.spectral_flatness(y=data)

        # calculate the mean or sum for the extracted features
        mean_spectral_centroids = np.mean(spectral_centroids)
        mean_spectral_rolloff = np.mean(spectral_rolloff)
        mean_spectral_bandwidth_2 = np.mean(spectral_bandwidth_2)
        sum_zero_crossings = sum(zero_crossings)
        mean_mfccs = np.mean(mfccs)
        .........

        to_append = f'\n{names_for_data[index]},{mean_spectral_centroids},
{mean_spectral_rolloff},{mean_spectral_bandwidth_2},{sum_zero_crossings},
{mean_mfccs},'
        to_append += f'{mean_magnitude},{mean_freq},{mean_chroma_stft},{mean_rms},
{mean_bank},{mean_mel},{mean_spectral_contrast},{mean_chroma_med},'
        to_append += f'{mean_melspectrogram},{mean_flatness}'

        g.write(to_append)

    g.close()
```

[3.2] Extracting MFCCs

```
def  extractMeanMfccsForDataSet(self, set_of_data, names_for_data, filename):
    set_n_mfcc = 40
    set_n_fft = 2048
    set_hop_length = 512

    to_append = 'filename'
    for i in  range(set_n_mfcc):
        to_append += f',{str(i+1)}'

    g = open(filename, 'w')
    g.write(to_append)
    g.close()

    g = open(filename, 'a')

    for index in tqdm(range(len(set_of_data))):
        data = set_of_data[index]
        mfccs = librosa.feature.mfcc(y=data, sr=self.sr, n_fft=set_n_fft,
hop_length=set_hop_length, n_mfcc=set_n_mfcc)
        mfccs = np.mean(mfccs.T, axis=0)
        to_append = f'\n{names_for_data[index]}'
        for ind in  range(len(mfccs)):
            to_append += f',{np.mean(mfccs[ind])}'

        g.write(to_append)

    g.close()
```

Resources:

- [fft function] https://www.youtube.com/watch?v=mUXkj1BKYk0&t=289s
- [librosa.feature] https://librosa.github.io/librosa/feature.html

- [implementation examples] https://www.kdnuggets.com/2020/02/audio-data-analysis-deep-learning-python-part-1.html

[4] Definitions for some of the extracted features

The definitions below have been extracted from different sources from the internet. They helped me better understand the concepts. I have links to the original posts and pages for each of them for further reading.

[4.1] Spectral Centroid

> The spectral centroid is commonly associated with the measure of the brightness of a sound. This measure is obtained by evaluating the “center of gravity” using the Fourier transform’s frequency and magnitude information. ... In practice, centroid finds this frequency for a given frame, and then finds the nearest spectral bin for that frequency. The centroid is usually a lot higher than one might intuitively expect, because there is so much more energy above (than below) the fundamental which contributes to the average.source

> Intuitive: It is center of mass of the spectrum. Since spectrum gives the indication of how the signal's mass (amplitude) is distributed among the frequencies, its center of mass indicates the average amount of amplitude. From speech perspective, it is the average loudness. From the image perspective, it is the average brightness. The mathematical equation used is, as you must be knowing or have guessed by now, weighted average. 'Weighted' because, the frequency components may be for instance non-uniformly separated (depending upon the transformation used) or due to application of filters sometimes it makes more sense to use the frequency information also into average instead of giving equal importance.source

Resources:

- [Spectral Centroid] https://ccrma.stanford.edu/~unjung/AIR/areaExam.pdf
- [Spectral Centroid] https://www.quora.com/In-an-intuitive-explanation-what-is-spectral-centroid

[4.2] Spectral Rolloff

> It is a measure of the shape of the signal. It represents the frequency at which high frequencies decline to 0. To obtain it, we have to calculate the fraction of bins in the power spectrum where 85% of its power is at lower frequencies. source

> The roll-off frequency is defined as the frequency under which some percentage (cutoff) of the total energy of the spectrum is contained. The roll-off frequency can be used to distinguish between harmonic (below roll-off) and noisy sounds (above roll-off) source

Resources:

- [roll-off frequency] https://essentia.upf.edu/reference/streaming_RollOff.html

- [Spectral Rolloff] https://www.kdnuggets.com/2020/02/audio-data-analysis-deep-learning-python-part-1.html

[4.3] Spectral Bandwidth

> The spectral bandwidth is defined as the width of the band of light at one-half the peak maximum (or full width at half maximum [FWHM]) and is represented by the two vertical red lines and λSB on the wavelength axis. source

Resources:

- https://www.analiticaweb.com.br/newsletter/02/AN51721_UV.pdf

[4.4] Zero-Crossing Rate

> By looking at different speech and audio waveforms, we can see that depending on the content, they vary a lot in their **smoothness**. For example, voiced speech sounds are more smooth than unvoiced ones. Smoothness is thus a informative characteristic of the signal.
>
> A very simple way for measuring the smoothness of a signal is to calculate the number of zero-crossing within a segment of that signal. A voice signal oscillates slowly — for example, a 100 Hz signal will cross zero 100 per second — whereas an unvoiced fricative can have 3000 zero crossings per second. source

Resources:

- [Zero-Crossing Rate] https://wiki.aalto.fi/display/ITSP/Zero-crossing+rate

[4.5] Mel-Frequency Cepstral Coefficients(MFCCs)

> The Mel frequency cepstral coefficients (MFCCs) of a signal are a small set of features (usually about 10–20) which concisely describe the overall shape of a spectral envelope. It models the characteristics of the human voice. source

As I mentioned earlier, I stored the MFCCs in a separate .csv file to make the loading of the extracted data more intuitive and more flexible regarding the number of features extracted.

Resources:

- [explanation] https://www.youtube.com/watch?v=m3XbqfIij_Y
- [implementation example] https://www.youtube.com/watch?v=Oa_d-zaUti8
- [librosa.feature.mfcc] https://librosa.github.io/librosa/generated/librosa.feature.mfcc.html
- [tutorial] http://practicalcryptography.com/miscellaneous/machine-learning/guide-mel-frequency-cepstral-coefficients-mfccs/
- [explanation with graphics] https://medium.com/@LeonFedden/comparative-audio-analysis-with-wavenet-mfccs-umap-t-sne-and-pca-cb8237bfce2f

[4.6] Chroma feature

> A chroma feature or vector is typically a 12-element feature vector indicating how much energy of each pitch class, {C, C#, D, D#, E, …, B}, is present in the signal. In short, It provides a robust way to describe a similarity measure between music pieces. source

Resources:

- https://www.kdnuggets.com/2020/02/audio-data-analysis-deep-learning-python-part-1.html

[4.7] FFT - frequency and magnitude

> A fast Fourier transform (FFT) is an algorithm that computes the discrete Fourier transform (DFT) of a sequence, or its inverse (IDFT). Fourier analysis converts a signal from its original domain (often time or space) to a representation in the frequency domain and vice versa. source

Resources:

- [fft] https://www.karlsims.com/fft.html
- [fft] https://en.wikipedia.org/wiki/Fast_Fourier_transform
- [numpy.fft] https://numpy.org/doc/stable/reference/routines.fft.html
- [explanations and examples] https://www.mathworks.com/help/signal/examples/practical-introduction-to-frequency-domain-analysis.html

[4.8] Spectral Contrast

> Spectral contrast considers the spectral peak, the spectral valley, and their difference in each frequency subband. For more information. source

Resources:

- [Spectral Contrast] https://musicinformationretrieval.com/spectral_features.html

[5] Loading features and MFCC values

[5.1] Loading features

```
def  loadFeaturesForDataSet(self, filename, data_names):
    data_features = [0] * len(data_names)
    fd = open(filename, 'r')

    # we jump over the first line which contains the names of the fields
    for line in tqdm(fd.readlines()[1:]):
        features = line.split(',')
        name = features[0]
        if name in data_names:
            numeric_data_features = [float(elem) for elem in features[1:]]
            data_features[data_names.index(name)] = numeric_data_features

    fd.close()
    return data_features
```

[5.2] Loading MFCC values

Loading MFCCs values is done similar to loading features.

[6] Models

[6.1] Support Vector Machines

I implemented the SVM using `sklearn.svm.SVC` (C-Support Vector Classification).

[6.1.1] Parameters:

- `probability: True` We need to compute probabilities in order to plot the precision-recall curve. To compute probabilities of possible outcomes for samples in X, the model needs to have probability information computed at training time: fit with attribute `probability` set to True.
- `C: 5` Regularization parameter. The strength of the regularization is inversely proportional to C. Intuitively, larger C -> more squiggly (a plane with very few misclassifications will be given precedence.), smaller C -> less squiggly (planes that separate the points well will be found, even if there are some misclassifications)
- `kernel: 'rbf'`
- `gamma: 0.001` Kernel coefficient for ‘rbf’. A small gamma will give you low bias and high variance while a large gamma will give you higher bias and low variance.

[6.1.2] Code:

```
def svcAlgorithm(self):
    start_time = time.time()

    self.readData()
    self.readLabels()
    self.loadMeanAndFeaturesAllData()
    self.standardizationScale()

    model = SVC(probability=True, C=5, gamma=0.001, kernel='rbf')

    print("\nfit train features... ")
    model.fit(self.train_features, self.train_labels)
    print("fit train features... done")

    print("predict validation features... ")
```

```
        predictions = model.predict(self.validation_features)
        print("predict validation features... done")

        # plot precision-recall curve and confusion matrix
        prob = model.predict_proba(self.validation_features)
        skplt.metrics.plot_precision_recall_curve(self.validation_labels, prob)
        skplt.metrics.plot_confusion_matrix(
        self.validation_labels, predictions)
        plt.show()

        # calculate recall, precision and accuracy
        self.recall = round(recall_score(self.validation_labels, predictions), 3)
        self.precision = round(
        average_precision_score(self.validation_labels, predictions), 3)
        self.accuracy = np.mean(predictions == self.validation_labels)

        print("predict test features... ")
        predictions = model.predict(self.test_features)
        print("predict test features... done")

        # create the submission file with the .wav file name and the predicted value
        g = open(self.OUTPUT_FILE_PATH, 'w')
        g.write('name,label')

        for index in  range(len(self.test_names)):
            g.write(f'\n{self.test_names[index]},{predictions[index]}')
        g.close()

        # calculate the amount of time the algorithm has been running for
        stop_time = time.time()
        self.runningTime = round(int(stop_time - start_time)/60, 2)
```

[6.1.3] Output:

Output for validation data:

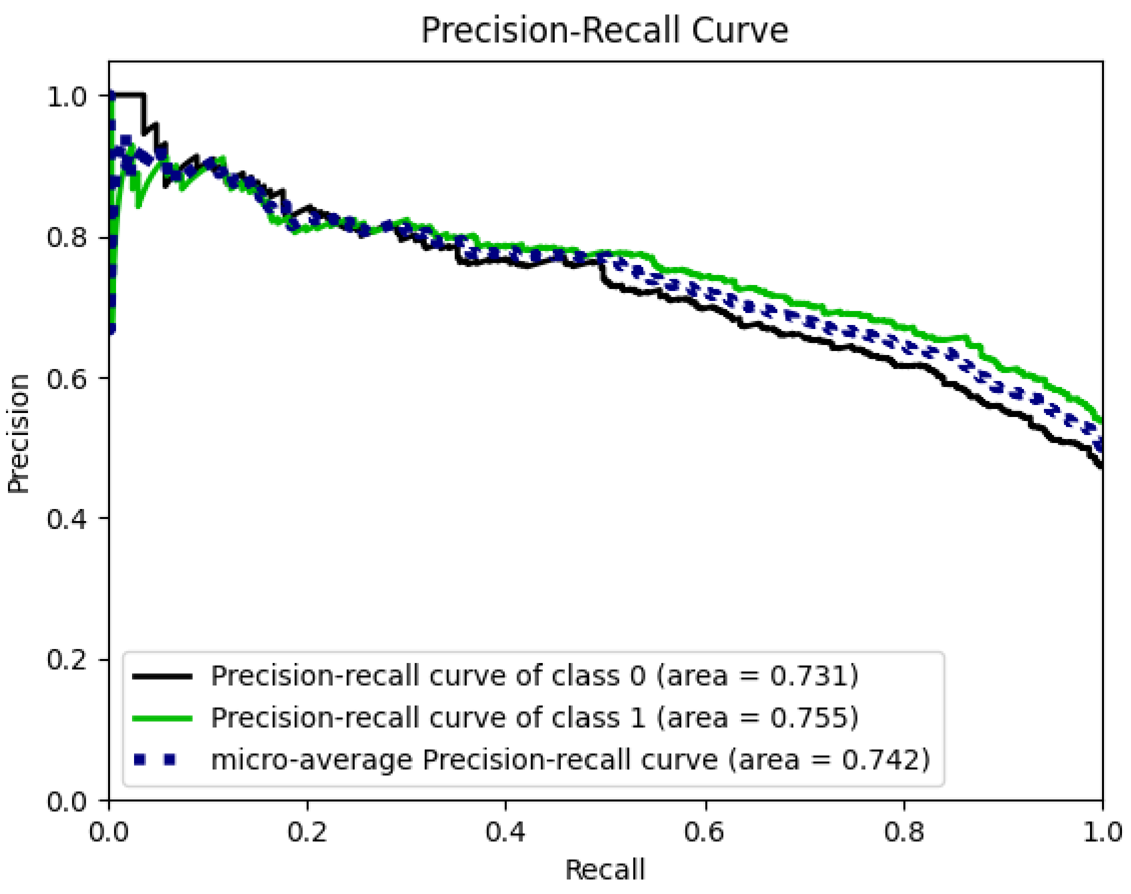
```
              precision    recall  f1-score   support

     class 0       0.66      0.70      0.68       472
     class 1       0.71      0.68      0.69       528

    accuracy                           0.69      1000
   macro avg       0.69      0.69      0.69      1000
weighted avg       0.69      0.69      0.69      1000
```

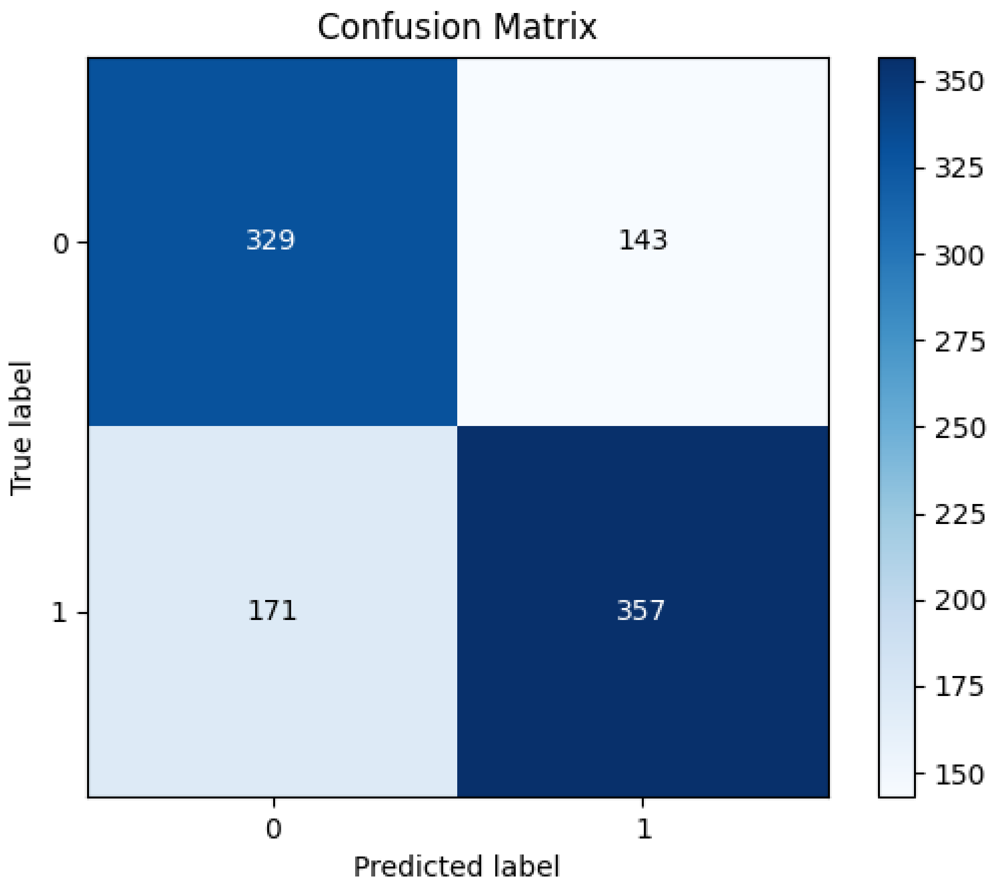
> f1-score is a measure of a test's accuracy. It considers both the precision *p* and the recall *r* of the test to compute the score: *q/p* is the number of correct positive results divided by the number of all positive results returned by the classifier, and *r* is the number of correct positive results divided by the number of all relevant samples (all samples that should have been identified as positive). source

Output for test data from kaggle:

```
Public score: 0.63222
Private score: 0.62761
```

[6.1.4] Precision-recall curve and confusion matrix plots:





[6.1.5] Observation:

If we train the model on both the train and the validation data, we obtain the following score on kaggle:

```
Public score: 0.63000
Private score: 0.63428
```

Resources:

- [SVC] https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html
- [parameters] http://queirozf.com/entries/choosing-c-hyperparameter-for-svm-classifiers-examples-with-scikit-learn

[6.2] Neural Network - sklearn

I implemented the neural network using `sklearn.neural_network.MLPClassifier` (C-Support Vector Classification). The definitions for parameters have been take from MLPClassifier.

[6.2.1] Parameters:

- `hidden_layer_sizes=(100,)` The ith element represents the number of neurons in the ith hidden layer.
- `random_state=1` Determines random number generation for weights and bias initialization, train-test split if early stopping is used, and batch sampling when solver='sgd' or 'adam'. *Pass an int for reproducible results across multiple function calls.*
- `max_iter=200` Maximum number of iterations.
- `early_stopping=False` Whether to use early stopping to terminate training when validation score is not improving.

[6.2.2] Code:

```
def  neuralAlgorithm(self):
    start_time = time.time()

    self.readData()
    self.readLabels()
    self.loadMeanAndFeaturesAllData()
    self.standardizationScale()

    print("\nfit train features... ")
    clf = MLPClassifier(hidden_layer_sizes=(100,), random_state=1, max_iter=200,
    early_stopping=False).fit(self.train_features, self.train_labels)
    print("fit train features... done")

    print("predict validation features... ")
    predictions = clf.predict(self.validation_features)
    print("predict validation features... done")
```

```
        # plot precision-recall curve and confusion matrix
        prob = clf.predict_proba(self.validation_features)
        skplt.metrics.plot_precision_recall_curve(self.validation_labels, prob)
        skplt.metrics.plot_confusion_matrix(
        self.validation_labels, predictions)
        plt.show()

        # calculate recall, precision and accuracy
        self.recall = round(recall_score(
        self.validation_labels, predictions), 3)
        self.precision = round(average_precision_score(
        self.validation_labels, predictions), 3)
        self.accuracy = clf.score(
        self.validation_features, self.validation_labels)

        print("predict test features... ")
        predictions = clf.predict(self.test_features)
        print("predict test features... done")

        # create the submission file with the .wav file name and the predicted value
        g = open(self.OUTPUT_FILE_PATH, 'w')
        g.write('name,label')
        for index in  range(len(self.test_names)):
        g.write(f'\n{self.test_names[index]},{predictions[index]}')
        g.close()

        # calculate the amount of time the algorithm has been running for
        stop_time = time.time()
        self.runningTime = round(int(stop_time - start_time)/60, 2)
```

[6.2.3] Output:

Output for validation data:

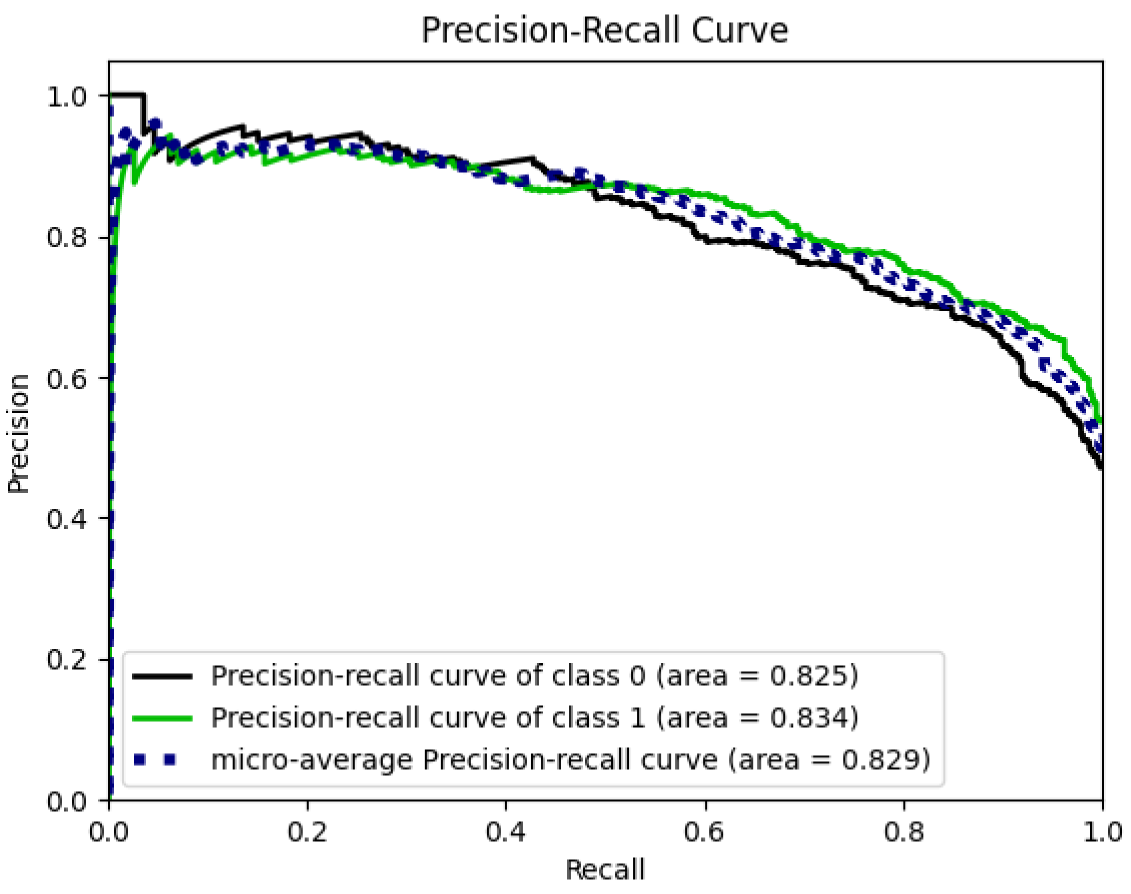
```
              precision    recall  f1-score   support

     class 0       0.75      0.74      0.75       472
     class 1       0.77      0.78      0.78       528

    accuracy                           0.77      1000
   macro avg       0.76      0.76      0.76      1000
weighted avg       0.76      0.77      0.76      1000
```

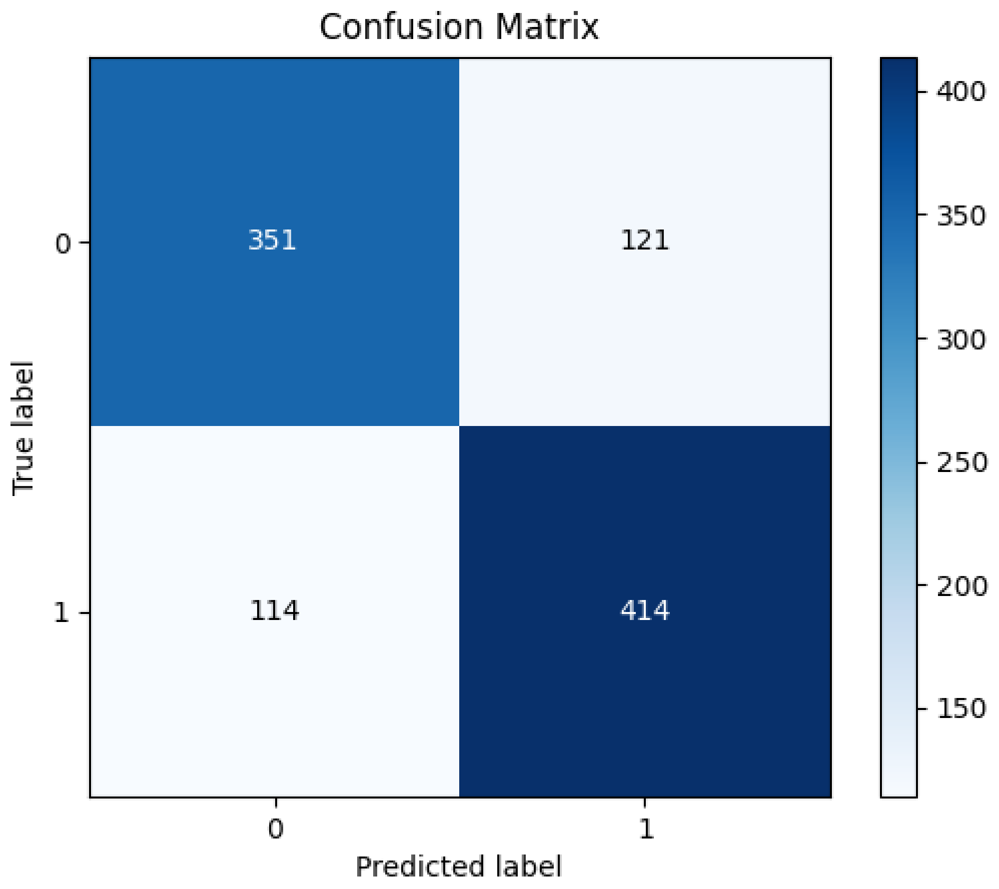
> f1-score is a measure of a test's accuracy. It considers both the precision *p* and the recall *r* of the test to compute the score: *q/p* is the number of correct positive results divided by the number of all positive results returned by the classifier, and *r* is the number of correct positive results divided by the number of all relevant samples (all samples that should have been identified as positive). source

Output for test data from kaggle:

```
Public score: 0.60555
Private score: 0.61142
```

[6.2.4] Precision-recall curve and confusion matrix plots:





[6.1.5] Observation:

If we train the model on both the train and the validation data, we obtain the following score on kaggle:

```
Public score: 0.65444
Private score: 0.63571
```

Resources:

- [MLPClassifier] https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html

[6.3] Neural Network - keras

I implemented the neural network using `tf.keras.Sequential` .

[6.3.1] Parameters:

I created 5 layers with the following structure and I added a dropout after each one of them, except for the last one. Dropout is a technique where randomly selected neurons are ignored during training so their contribution to the activation of downstream neurons is temporally removed on the forward pass and any weight updates are not applied to the neuron on the backward pass. source

```
model.add(Dense(units=110, input_dim=55, activation='relu'))
model.add(Dense(units=70, activation='relu'))
model.add(Dense(units=50, activation='relu'))
model.add(Dense(units=20, activation='relu'))
model.add(Dense(units=1, activation='sigmoid'))
```

For the `model.compile` I used the following parameters:

- `loss="binary_crossentropy"` **binary** classification as I have two target classes (class 0 and class 1) info
- `optimizer="adam"`
- `metrics=['accuracy']`

For the `model.fit` I used the following parameters:

- `epochs=200` the number of epochs is a hyperparameter that defines the number times that the learning algorithm will work through the entire training dataset
- `batch_size=50` the batch size defines the number of samples that will be propagated through the network

[6.3.2] Code:

```
def  kerasNeuralNetwork(self):
    start_time = time.time()

    self.readData()
    self.readLabels()
    self.loadMeanAndFeaturesAllData()
    self.standardizationScale()

    # create model, add dense layers one by one specifying activation function
    # Units are the dimensionality of the output space for the layer,
    # which equals the number of hidden units
    model = Sequential()

    # input layer requires input_dim param
    model.add(Dense(units=110, input_dim=55, activation='relu'))
    model.add(Dropout(.2))
    model.add(Dense(units=70, activation='relu'))
    model.add(Dropout(.2))
    model.add(Dense(units=50, activation='relu'))
    model.add(Dropout(.2))
    model.add(Dense(units=20, activation='relu'))
    model.add(Dropout(.2))
    # sigmoid instead of relu for final probability between 0 and 1
    model.add(Dense(units=1, activation='sigmoid'))

    # The compile method configures the model’s learning process
    model.compile(loss="binary_crossentropy",
    optimizer="adam", metrics=['accuracy'])

    # The fit method does the training in batches
    # validation_features and validation_labels are Numpy arrays
    # --just like in the Scikit-Learn API.
    model.fit(self.train_features, self.train_labels,
    epochs=200, batch_size=50)

    # The evaluate method calculates the losses and metrics for the trained model
    # Evaluating the model on the training set
    score = model.evaluate(self.train_features,
    self.train_labels, verbose=0)
    print("Training Accuracy: {0:.3%}".format(score[1]))

    # Evaluating the model on the validation set
    score = model.evaluate(self.validation_features,
    self.validation_labels, verbose=0)
    print("Validation Accuracy: {0:.3%}".format(score[1]))

    # Predict labels for validation data set
    predictions = model.predict_classes(
    self.validation_features, verbose=0)
    predictions = [x[0] for x in predictions]

    # calculate recall, precision and accuracy
    self.recall = round(recall_score(
    self.validation_labels, predictions), 3)
```

```
        self.precision = round(average_precision_score(
        self.validation_labels, predictions), 3)
        self.accuracy = np.mean(predictions == self.validation_labels)

        # Predict labels for test data set
        predictions = model.predict_classes(self.test_features, verbose=0)
        predictions = [x[0] for x in predictions]

        g = open(self.OUTPUT_FILE_PATH, 'w')
        g.write('name,label')

        for index in  range(len(self.test_names)):
            g.write(f'\n{self.test_names[index]},{predictions[index]}')
        g.close()

        stop_time = time.time()
        self.runningTime = round(int(stop_time - start_time)/60, 2)
```

[6.3.3] Output:

Output for validation data:

```
  Training Accuracy: 99.200%
  Validation Accuracy: 79.200%

                precision    recall  f1-score   support

       class 0       0.78      0.78      0.78       472
       class 1       0.80      0.81      0.80       528

      accuracy                           0.79      1000
     macro avg       0.79      0.79      0.79      1000
  weighted avg       0.79      0.79      0.79      1000
```

Output for test data from kaggle:

```
  Public score: 0.63666
  Private score: 0.59857
```

Resources:

- [implementation] https://www.youtube.com/watch?v=T91fsaG2L0s&fbclid=IwAR195aBLTdjq-6PEvatw6bSu398mne1pXFjrwQU9dSvxaORM_h_6NZdK6Pk
- [implementation] https://github.com/jg-fisher/diabetesNeuralNetwork/blob/master/keras_diabetes_classification.py
- [predictions] https://machinelearningmastery.com/how-to-make-classification-and-regression-predictions-for-deep-learning-models-in-keras/

- [dropout] https://machinelearningmastery.com/dropout-regularization-deep-learning-models-keras/
- [batch size] https://stats.stackexchange.com/questions/153531/what-is-batch-size-in-neural-network
- https://towardsdatascience.com/how-to-apply-machine-learning-and-deep-learning-methods-to-audio-analysis-615e286fcbbc
- [functions and parameters] https://keras.io/api/models/model/#model-class
- [overfitting] https://www.youtube.com/watch?v=Gf5DO6br0ts&list=PL-wATfeyAMNrtbkCNsLcpoAyBBRJZVlnf&index=14
- https://www.infoworld.com/article/3336192/what-is-keras-the-deep-neural-network-api-explained.html
- [binary_crossentropy] https://stackoverflow.com/questions/42081257/why-binary-crossentropy-and-categorical-crossentropy-give-different-performances

[7] Hyperparameter Tuning

I used `sklearn.model_selection.GridSearchCV` to try different parameters for `sklearn.svm.SVC` and `sklearn.neural_network.MLPClassifier` models.

Code for SVC hyperparameter tuning:

```
def  svcHyperparameterTuning(self):
    self.readData()
    self.readLabels()
    self.loadMeanAndFeaturesAllData()

    # try many parameters to find the best ones
    clf = GridSearchCV(SVC(gamma='auto'), {
        'C': [1, 5, 10, 15, 20, 25, 30, 35],
        'gamma': ['scale', 'auto', 0.1, 0.01, 0.001, 0.0001, 1, 5],
        'kernel': ['rbf', 'linear', 'poly', 'sigmoid']
    }, cv=5, return_train_score=False)

    clf.fit(self.train_features, self.train_labels)
    data_frame = pd.DataFrame(clf.cv_results_)

    g = open(self.SVC_HYPERPARAMETER_TUNING, 'a')
    g.write('\n\n\nNew hyperparameter tuning\n')
    g.write(str(data_frame[['param_C', 'param_kernel', 'mean_test_score']]))
    g.write(f'\nBest params: {clf.best_params_}')
    g.write(f'\nBest score: {clf.best_score_}')
    g.close()
```

Resources:

- https://github.com/codebasics/py/blob/master/ML/15_gridsearch/Exercise/15_grid_search_cv_exercise.ipynb
- https://github.com/codebasics/py/blob/master/ML/15_gridsearch/15_grid_search.ipynb
- https://www.youtube.com/watch?v=HdlDYng8g9s
- https://www.youtube.com/watch?v=pooXM9mM7FU
- https://www.kaggle.com/funxexcel/p2-logistic-regression-hyperparameter-tuning

[8] Standardization. Normalization

I used `sklearn.preprocessing.scale` , `sklearn.preprocessing.StandardScaler` and `sklearn.preprocessing.normalize` for scaling and normalizing data.

```
def  standardizationScale(self):
    self.train_features = preprocessing.scale(self.train_features)
    self.validation_features = preprocessing.scale(self.validation_features)
    self.test_features = preprocessing.scale(self.test_features)



def  normalizationNormalize(self):
    self.train_features = preprocessing.normalize(self.train_features)
    self.validation_features = preprocessing.normalize(self.validation_features)
    self.test_features = preprocessing.normalize(self.test_features)



def  normalizationStandardizationLab(self):
    scaler = preprocessing.StandardScaler()
    scaler.fit(self.train_features)
    self.train_features = scaler.transform(self.train_features)
    self.validation_features = scaler.transform(self.validation_features)
    self.test_features = scaler.transform(self.test_features)
```

Resources:

- [standardization/normalization] https://machinelearningmastery.com/rescaling-data-for-machine-learning-in-python-with-scikit-learn/?fbclid=IwAR31clqIFgUfDgvh4GoU4TY-Qgse1qOuDdQp6wVu8qzr2BnxBfkZFOX9hYU

[9] Plotting

I used `skplt.metrics.plot_precision_recall_curve` and `skplt.metrics.plot_confusion_matrix` for plotting in order to analyze parameters.

Code for plotting fragment:

```
# plot precision-recall curve and confusion matrix
prob = model.predict_proba(self.validation_features)
skplt.metrics.plot_precision_recall_curve(self.validation_labels, prob)
skplt.metrics.plot_confusion_matrix(
self.validation_labels, predictions)
plt.show()
```

Resources:

- https://github.com/reiinakano/scikit-plot/issues/87

[10] Precision. Recall. Accuracy

I used `sklearn.metrics` to import `recall_score` and `average_precision_score` in order to calculate recall and precision for the validation data set. I calculated accuracy manually, comparing the obtained predictions and the real labels for the validation data set.

To calculate precision, recall and accuracy for each class, I used `sklearn.metrics.classification_report`.

Resources:

- [classification_report] https://scikit-learn.org/stable/modules/generated/sklearn.metrics.classification_report.html
- [recall_score] https://scikit-learn.org/stable/modules/generated/sklearn.metrics.recall_score.html#sklearn.metrics.recall_score
- [average_precision_score] https://scikit-learn.org/stable/modules/generated/sklearn.metrics.average_precision_score.html#sklearn.metrics.average_precision_score

[11] Other auxiliary functions used

- `tryDifferentModels` I implemented this function to try many models with different parameters in order to quickly find out which model is more likely to work the best
- `tryAllFeatureCombinationsForSvc` this function tests, for a certain model, all combinations of features to find out which one will give the best results

[12] Other related resources

- https://haythamfayek.com/2016/04/21/speech-processing-for-machine-learning.html?fbclid=IwAR1w6-IcQ3yvuH2frW3vEDl7CqeC4yY6KOnrrZKSMz2b_MHO6qadmj-PSKg
- https://dev.to/zenulabidin/python-audio-processing-at-lightspeed-part-1-zignal-5658