Data Science Lab

Lab 7 solution

1 Introduction

In the following solution, we will go through the pipeline that achieved the baseline result you saw on the competition platform. Therefore, careful readers will found several things that can be improved and others left apart for further discussions.

2 Free Spoken Digit classification

[1]:
```
import os
import numpy as np
import matplotlib.pyplot as plt
```

2.1 Load the data

[2]:
```
dev_data_path = "./dataset/dev/"
eval_data_path = "./dataset/eval/"
```

First of all it is better to define a function to load the data from the file system, with the structure specified in the laboratory text.
In this function you can use the `wavfile` package from *scipy.io* to read the *.wav* audio files.

Loading the data exploiting the `wavfile.read()` function gives you two main information: 1. The sampling rate of the signal (in samples/sec) 2. The array with the amplitudes of the signal recorded for each sample

[3]:
```
from scipy.io import wavfile

def load_data(data_path, phase):
    files = os.listdir(data_path)
    wav_files = [file for file in files if file.endswith(".wav")]
    X = [None] * len(wav_files)
    y = [None] * len(wav_files)
    SR = [None] * len(wav_files)

    for file in wav_files:
```

```
        file_name = file.split(".")[0]
        if phase == "dev":
            f_id, label = tuple(file_name.split("_"))
        elif phase == "eval":
            f_id, label = file_name, float("NaN")
        else:
            raise Exception(f"Error - phase '{phase}' not recognised.")

        f_id, label = int(f_id), float(label)

        sr, data = wavfile.read(os.path.join(data_path,file))  # this return sr:
↪ sample rate, data: audio as array

        X[f_id] = data.astype(np.float32)
        SR[f_id] = sr
        y[f_id] = label

    return X, y, SR
```

Then use the function defined above to load the development data into memory

[4]:
```
dev_X, dev_y, dev_SR = load_data(dev_data_path, phase="dev")
dev_X[:5], dev_y[:5]
```

[4]:
```
([array([ 14.,  21.,   0., …,   3., -48., -80.], dtype=float32),
  array([-2., 12.,  1., …,  3., -3.,  3.], dtype=float32),
  array([-9., -7., -5., …, -5., -1., -7.], dtype=float32),
  array([-256., -512.,    0., …,    0.,    0.,    0.], dtype=float32),
  array([-20.,  10.,  26., …,  -3.,   5., -10.], dtype=float32)],
 [4.0, 7.0, 5.0, 2.0, 3.0])
```

Similarly you can load also the evaluation data into memory.
Remember that the evaluation data does not provide the ground truth labels.

[5]:
```
eval_X, _, eval_SR = load_data(eval_data_path, phase="eval")
eval_X[:5]
```

[5]:
```
[array([   0.,    0., -512., …,    0., -256., -256.], dtype=float32),
 array([  6.,   0.,  -3., …, -11.,  -2., -20.], dtype=float32),
 array([-580.,  153., -668., …, -360., -339., -343.], dtype=float32),
 array([-2., -6.,  0., …, 23.,  9., 22.], dtype=float32),
 array([-1.,  1., -5., …,  1., -4.,  2.], dtype=float32)]
```

2.2 Preprocessing

Once the data has been loaded, it has to be preprocessed in order to extract the most valuable features that can help us to train the classification model.

When dealing with digital signals two main analysis approaches can be exploited: - Analysis in time domain - Analysis in frequency domain

Only the analysis in time domain, that allows you to reach the baseline score on the evaluation set, is proposed in the following solution.

In particular, to perform the preprocessing steps that will transform the input signals into feature vectors, a windowed descriptive statistical analysis can be performed. The signal can be split into chunks and they can be characterized by means of statistical measures (e.g. mean, standard deviation). Doing so you can characterize each signal only with their more relevant information, reducing the dimensionality of the data and simplifying the classifier's training process.

2.2.1 Data normalization

Before applying the preprocessing however, some considerations have to be done: - Were the audios recorded in the same conditions? - Do the audios have the same duration?

To answer these questions we can perform some simple checks and adopt visualization techniques to look our data.

[6]:
```
SR = dev_SR[0]
print(f"Sampling rate = {SR} [sample/s]")
print("Do the sampling rate is always the same for all the development␣
↪recordings?",
      all([sr == SR  for sr in dev_SR]))
print("Do the sampling rate for the evaluation set is equal to development set?
↪",
      all([sr == SR  for sr in eval_SR]))
```

```
Sampling rate = 8000 [sample/s]
Do the sampling rate is always the same for all the development recordings? True
Do the sampling rate for the evaluation set is equal to development set? True
```

With the above check we assure that the sampling rate used to record the audios is equal for all the samples. Thus we can plot the audio waves showing the duration of the audios in seconds.

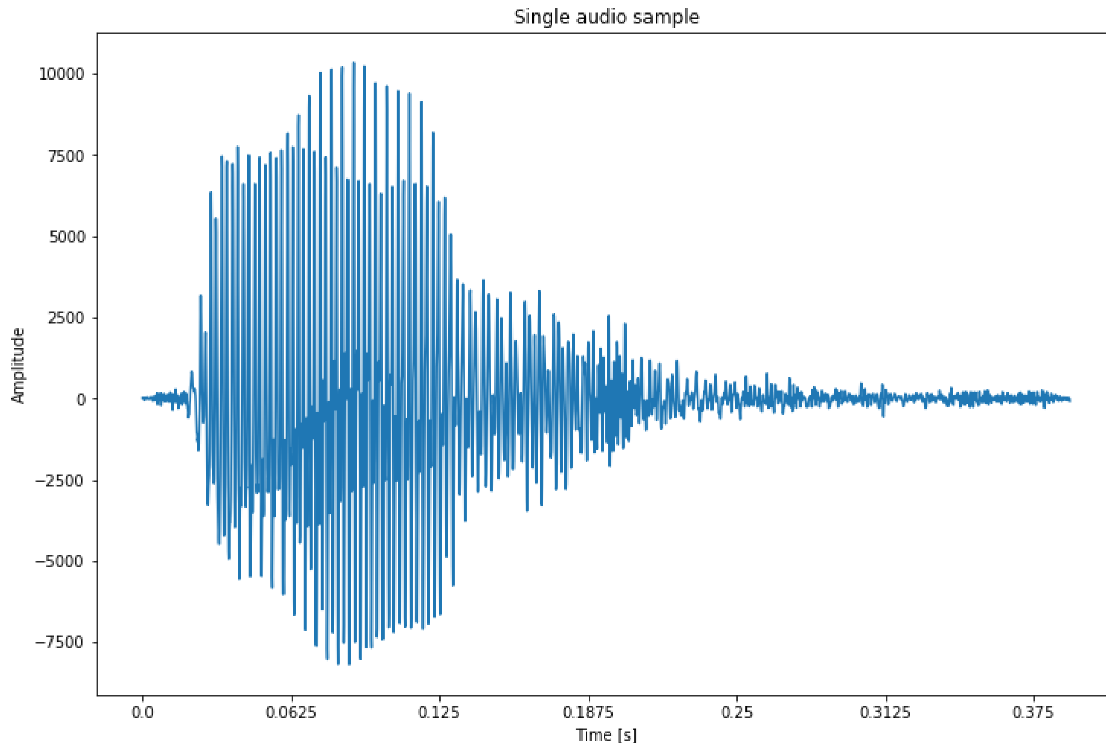
[7]:
```
fig, ax = plt.subplots(1,1,figsize=(12,8))

ax.plot(dev_X[0]) # Plot the first signal

# To show time on the x axes
ax.set_xticks(np.arange(0, len(dev_X[0]), 500))
ax.set_xticklabels([sample/SR for sample in np.arange(0, len(dev_X[0]), 500)])

ax.set_ylabel("Amplitude")
ax.set_xlabel("Time [s]")
ax.set_title("Single audio sample")

plt.show()
```

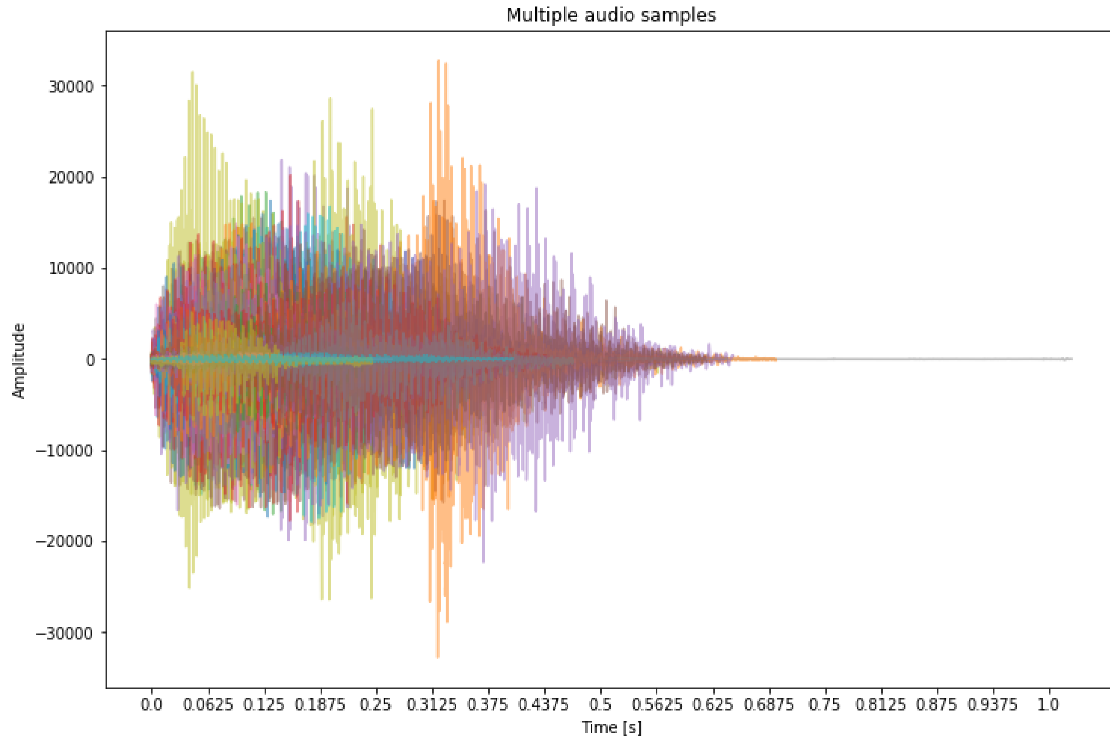


[8]:
```
def plot_audio_samples(X):
    lengths = []
    fig, ax = plt.subplots(1,1,figsize=(12,8))

    for idx in range(0, len(X), 10):
        ax.plot(X[idx], alpha=0.5)
        lengths.append(len(dev_X[idx]))


    # To show time on the x axes
    ax.set_xticks(np.arange(0, max(lengths), 500))
    ax.set_xticklabels([sample/SR for sample in np.arange(0, max(lengths),␣
↪500)])
    return ax

ax = plot_audio_samples(dev_X)
ax.set_ylabel("Amplitude")
ax.set_xlabel("Time [s]")
ax.set_title("Multiple audio samples")
plt.show()
```



2.2.2 Amplitude normalization

From to the visualization of the audio samples it is possible to notice that the vocal samples has very different amplitudes in many cases. This is due to different recording conditions such as the sensibility of the microphone or the recording volume and so on.

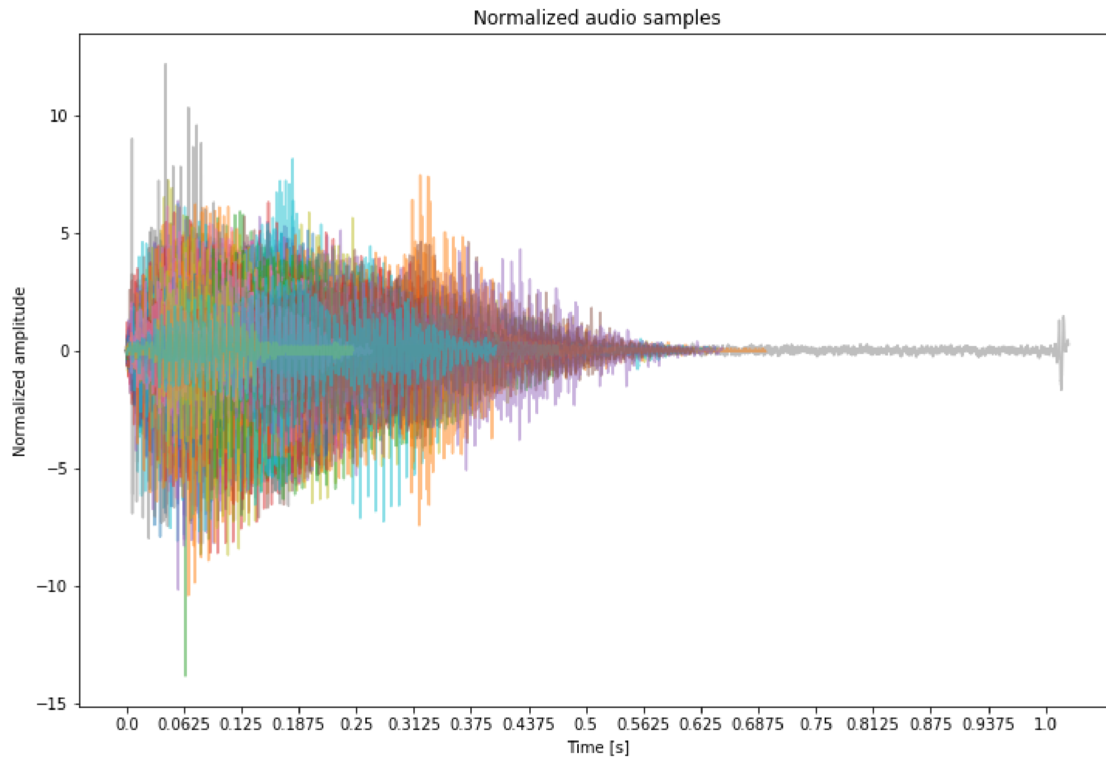
To partially overcome these issues it is possible to normalize the amplitude of each recording exploiting the standardization.

[9]:
```
from scipy.stats import zscore

def normalize_data(X):
    return [zscore(x) for x in X]

dev_X = normalize_data(dev_X)
eval_X = normalize_data(eval_X)
```

[10]:
```
ax = plot_audio_samples(dev_X)
ax.set_ylabel("Normalized amplitude")
ax.set_xlabel("Time [s]")
ax.set_title("Normalized audio samples")
plt.show()
```

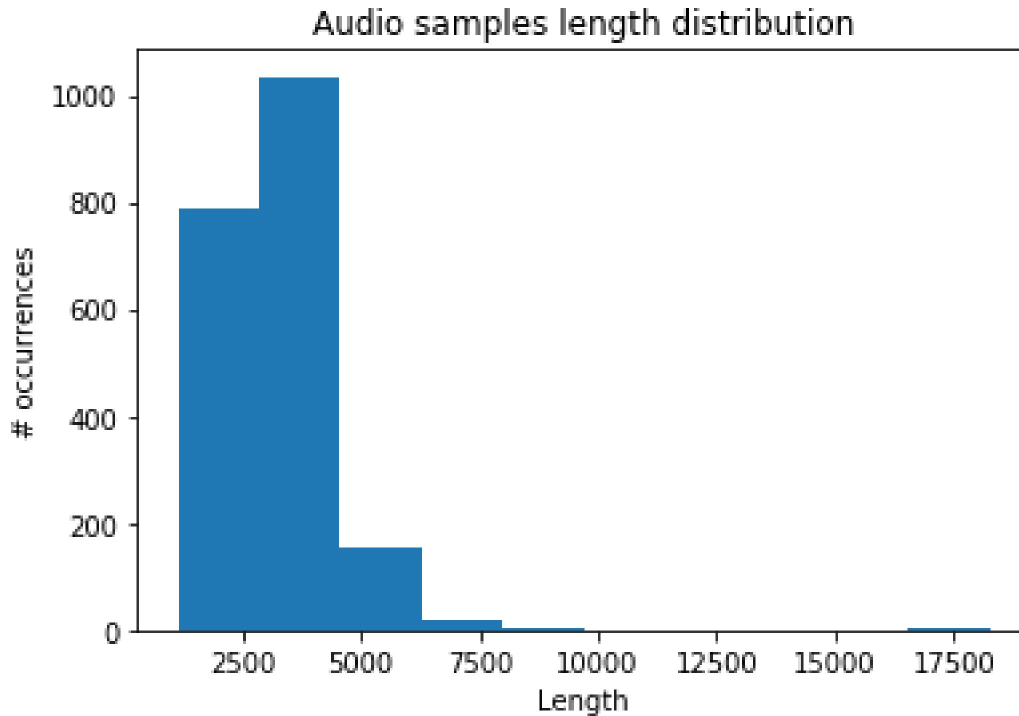


2.2.3 Length normalization

Also the duration of the audios is very different. It is possible to inspect the distributions of the lengths in both dev and eval sets to check if any preprocessing is required.

[11]:
```
dev_lengths = [len(x) for x in dev_X]
eval_lenghts = [len(x) for x in eval_X]

plt.hist(dev_lengths+eval_lenghts)
plt.title("Audio samples length distribution")
plt.xlabel("Length")
plt.ylabel("# occurrences")
plt.show()
plt.close()
```



From the histogram above it is possible to notice that there is a small number of samples that has a length of around 17500. For example this can be caused by microphones left recording. These can be considered outliers, but, instead of removing them you can try to cut their length reaching the lengths of the other samples. Of course this is a problem only when dealing with analysis in time domain.

Cut and Pad
To manage this problem you can consider to define an acceptable threshold length so then you can cut the signals longer than the threshold length and add some pad to the audio signal shorter then the threshold.

Consider that the training and the prediction phase of a classifier in many cases (e.g. Rendom Forest) require that the data has always the same number of features. Thus, the dev and eval data should have the same number of features after the feature computation process.

N.B. Frequency domain can have different kind of problems such as the presence of noisy frequencies.

2.2.4 Cut

To choose the threshold we can inspect the percentiles of the lengths vector. In particular we want to find a threshold that includes most of the data and leave out only the outliers. Let's say we want that at least the 95% of our data should have a length equal or lower than the threshold. Which is the threshold value?

[12]:
```
perc = 95

length_95_perc = int(np.ceil(np.percentile(dev_lengths+eval_lenghts, perc)))
print(f"The {perc}% of the data has length lower or equal to {length_95_perc}.")
```

```
The 95% of the data has length lower or equal to 5000.
```

Thus, only 5% of data has a length higher than 5000. This is a suitable value that can be used as length threshold.

Apply the cut to both dev and eval data with the threshold value previously defined.

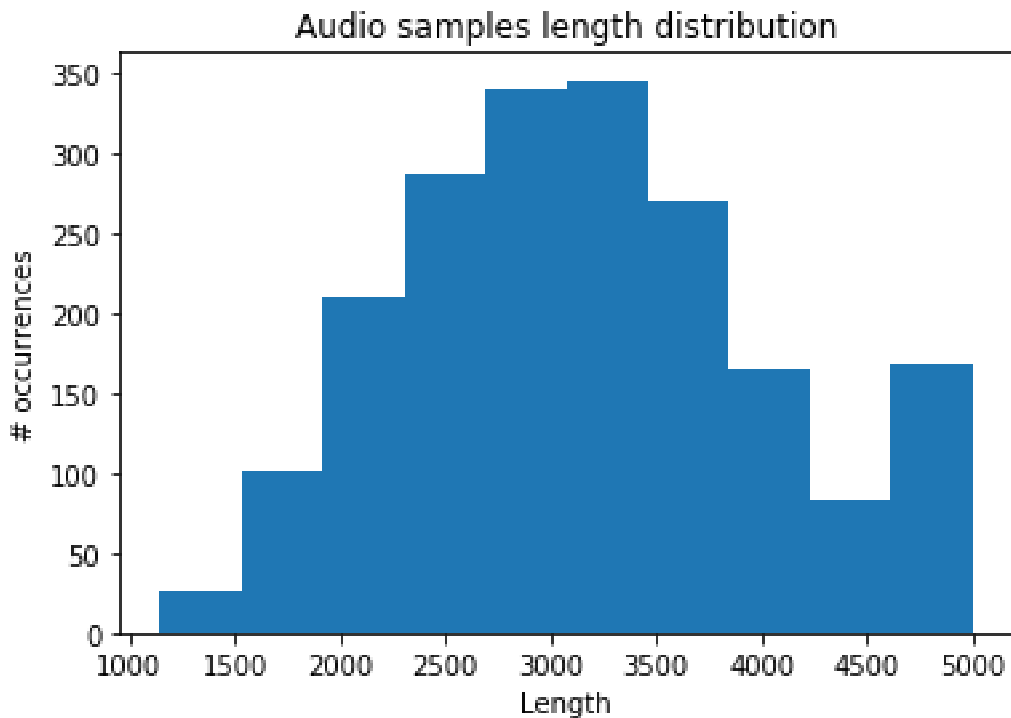
[13]:
```
dev_X = [x[:length_95_perc] for x in dev_X]

eval_X = [x[:length_95_perc] for x in eval_X]
```

Now, looking at the distribution of the signal lengths you can see that they are very well distributed.

[14]:
```
dev_lengths = [len(x) for x in dev_X]
eval_lenghts = [len(x) for x in eval_X]

plt.hist(dev_lengths+eval_lenghts)
plt.title("Audio samples length distribution")
plt.xlabel("Length")
plt.ylabel("# occurrences")
plt.show()
plt.close()
```



2.2.5 Pad

Padding all the audio samples with zeros allows every signal to have the same length. In the case of audio sample, including zeros at the end of the recording has a specific meaning: since the values in the array are representing the amplitudes of the signal at each timestamp, the value zero represent the *silence*.

[15]:
```
dev_X = [np.pad(x, (0,length_95_perc-len(x)), constant_values=0.0) for x in␣
 ↪dev_X]
eval_X = [np.pad(x, (0,length_95_perc-len(x)), constant_values=0.0) for x in␣
 ↪eval_X]
```

2.2.6 Window size

In order to perform a windowed analysis the size of the window should be carefully analyzed. In this specific case, the size of the window can be derived by the domain.
In fact, studies proved that phonetic elements have specific characteristics. The paper Linguistic uses of segmental duration in English: Acoustic and perceptual evidence highlights that in average unstressed vowels take 70 ms while long consonants such as *s* can reach a duration of about 200 ms.

In this work we take in consideration a window size that is 1/5 of the maximum duration of 200 ms to consider also short consonant and vowels. For these reasons the window size has been selected to be equal to **40 ms** and the number of the corresponding samples has been taken into account accordingly to the sampling rate.

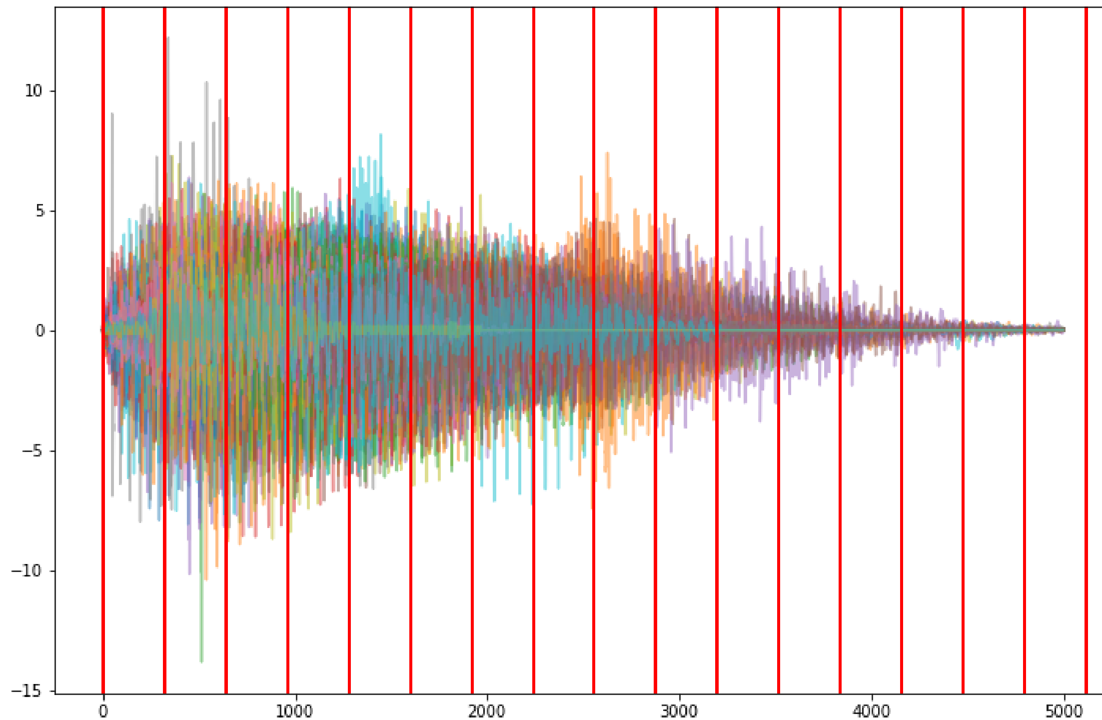
[16]:
```
step = int(SR * 0.04) # steps of 40 ms
step
```

[16]: 320

Then, it is possible to visualize the windows with the specified step.

[17]:
```
fig, ax = plt.subplots(1,1,figsize=(12,8))
for idx in range(0, len(dev_X), 10):
    ax.plot(dev_X[idx], alpha=0.5)

for s in range(0,length_95_perc+step,step):
    ax.axvline(s, color="red", linewidth=2)
plt.show()
plt.close()
```



2.2.7 Compute statistical features

After normalization phase all the audio samples has the same length. The windows size has been specified to inspect each phonetic element. Thus, it is necessary to define the features that need to be extracted from each window. A small set of possible features that can be calculated is: - Mean - Standard deviation - Min value - Max value - Root mean square

Each window will be characterized by the statistics reported above. There are also many other indexes that you can take into account to perform this analysis, the one reported above are the one used to obtain the baseline score.

In the next cell the functions to transform the input signals into statistical features, performing a window analysis with the selected window size, are defined.

[18]:
```
def RMS(a):
    return np.sqrt(np.mean(np.power(a, 2)))

def compute_signal_statistical_features(x, max_len, step):
    feat_x = []
    for idx in range(step, max_len+step, step):
        seg_x = x[idx-step:idx]
        if len(seg_x)>0:
            feat = [np.mean(seg_x),
                    np.std(seg_x),
```

```
                        np.min(seg_x),
                        np.max(seg_x),
                        RMS(seg_x)]
        else:
            feat = [0.0, 0.0, 0.0, 0.0, 0.0]

        feat_x.append(
            np.array(feat)
        )

    return np.nan_to_num(np.hstack(feat_x))

def compute_statistical_features(X, max_len, step):
    features_X = []
    for x in X:
        f_x = compute_signal_statistical_features(x, max_len, step=step)
        features_X.append(f_x)
    return np.vstack(features_X)
```

Apply the transformation to dev dataset

[19]:
```
feat_dev_X = compute_statistical_features(dev_X, length_95_perc, step=step)
feat_dev_X.shape
```

[19]:
```
(1500, 80)
```

It is possible to visualize some of the statistical features computed in the feature engineering process. In particular the following plots show the mean and the standard deviations of sample of signals for each class.

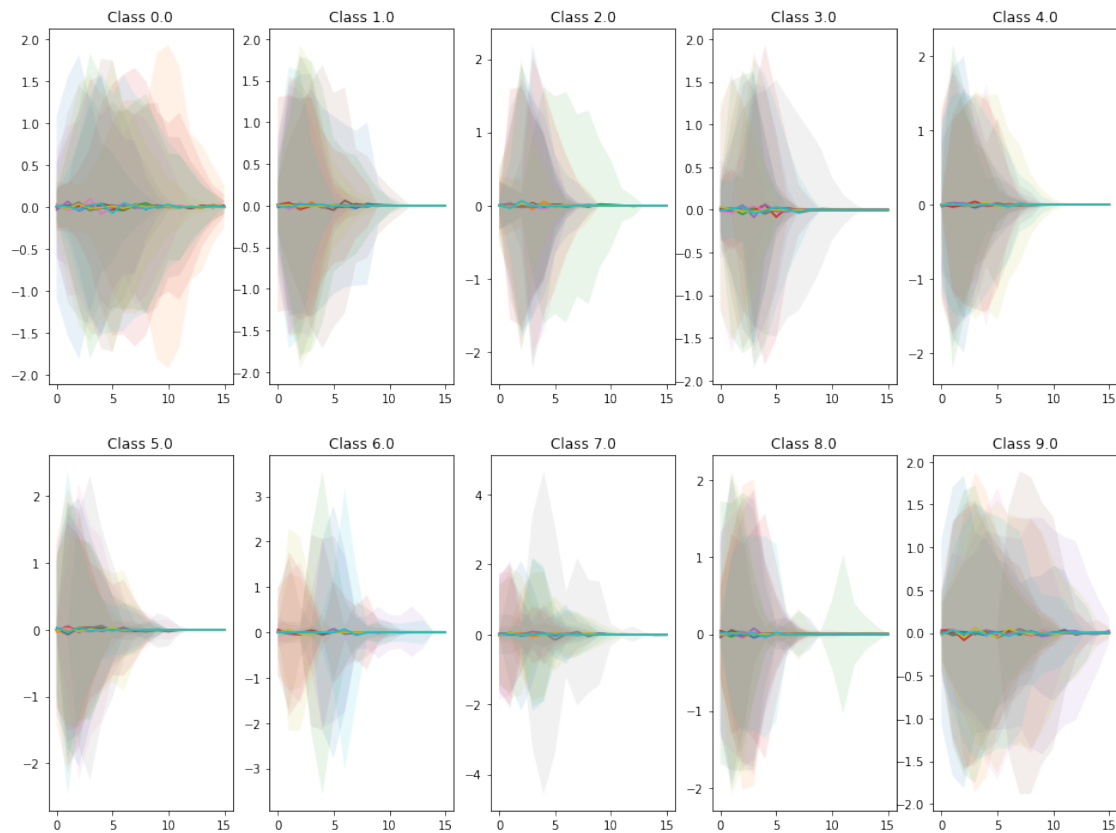
[20]:
```
fig, ax = plt.subplots(2,5,figsize=(16,12))



for c in np.unique(dev_y):
    c_feat_dev_X = feat_dev_X[dev_y == c]
    idx = int(c/5)
    idy = int(c%5)

    for i in range(0,10):
        ax[idx, idy].fill_between(x= np.arange(len(c_feat_dev_X[i,0::5])),
                                y1= c_feat_dev_X[i,0::5]-c_feat_dev_X[i,1::5], # The␣
 ↪mean minus the standard deviation
                                y2=c_feat_dev_X[i,0::5]+c_feat_dev_X[i,1::5], # The␣
 ↪mean plus the standard deviation
                               alpha=0.1)
        ax[idx, idy].plot(c_feat_dev_X[i,0::5]) # The mean values
        ax[idx, idy].set_title(f"Class {c}")
plt.show()
```

```
plt.close()
```



Are you able to identify some pattern that characterize the different classes? Maybe looking at more indexes the classes can be even more separated and a classification model can be able to learn these differences.

Then, apply the same feature engineering process to the evaluation set as well.

[21]:
```
feat_eval_X = compute_statistical_features(eval_X, length_95_perc, step=step)
feat_eval_X.shape
```

[21]: (500, 80)

2.3 Classification

To build a robust and reliable classification model a cross validation on the data available in the development set is performed. In the baseline solution only the Random Forest classifier is exploited, performing a grid search on some of the most obvious parameters i.e. `n_estimators`and `max_depth`.

[22]:
```
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report

from sklearn.model_selection import GridSearchCV

from sklearn.ensemble import RandomForestClassifier
```

The following function allows you to perform a grid search over a parameter grid applying the cross validation process to each configuration. In particular, `GridSearchCV` is the function that performs the exploration of all the possible parameters combinations trying to optimize the scoring function specified with the `scoring`attribute.

[23]:
```
# Inspired by SK-learn documentation https://scikit-learn.org/stable/
 ↪auto_examples/model_selection/plot_grid_search_digits.html
def build_classifier(X_train, y_train, X_test, y_test, clf_to_evaluate, scores,␣
 ↪param_grid, n_folds=3 ):
    print("# Tuning hyper-parameters for %s" % score)
    print()

    clf = GridSearchCV(clf_to_evaluate, param_grid, cv=n_folds,
                       scoring=score, verbose=True, n_jobs=4, iid=False)
    clf.fit(X_train, y_train)

    print("Best parameters set found on development set:")
    print()
    print(clf.best_params_)
    print()
    print("Grid scores on development set:")
    print()
    means = clf.cv_results_['mean_test_score']
    stds = clf.cv_results_['std_test_score']
    for mean, std, params in zip(means, stds, clf.cv_results_['params']):
        print("%0.3f (+/-%0.03f) for %r"
              % (mean, std * 2, params))
    print()

    print("Detailed classification report:")
    print()
    print("The model is trained on the full development set.")
    print("The scores are computed on the full evaluation set.")
    print()
    y_true, y_pred = y_test, clf.predict(X_test)
    print(classification_report(y_true, y_pred))
    print()
    return clf
```

2.3.1 Build model with statistical features

The development set is split into train and test dataset. The train partition is exploited to perform the grid search of the parameters with cross validation. The test partition is then used to test the best configuration found during the training process.

[24]:
```
X_train, X_test, y_train, y_test = train_test_split(feat_dev_X, dev_y,␣
↪test_size=0.4, random_state=0)
X_train.shape, X_test.shape, len(y_train), len(y_test)
```

[24]:
```
((900, 80), (600, 80), 900, 600)
```

Let's perform the grid search with cross validation. The score that we want to optimize is the `f1_macro`. The parameters grid is specified in the `params_grid` variable. The only classifier taken into account in this baseline solution is the `RandomForestClassifier`.

[25]:
```
score = 'f1_macro'

n_estimators = [100, 400, 500, 1000, 2000]
max_depth = [100, 200]
params_grid = {
                'n_estimators': n_estimators,
                'max_depth': max_depth
            }

clf_to_evaluate = RandomForestClassifier()

best_clf = build_classifier(X_train, y_train, X_test, y_test, clf_to_evaluate,␣
↪score, params_grid)
```

```
# Tuning hyper-parameters for f1_macro

Fitting 3 folds for each of 10 candidates, totalling 30 fits

[Parallel(n_jobs=4)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=4)]: Done  30 out of  30 | elapsed:   31.8s finished

Best parameters set found on development set:

{'max_depth': 200, 'n_estimators': 2000}

Grid scores on development set:

0.608 (+/-0.022) for {'max_depth': 100, 'n_estimators': 100}
0.617 (+/-0.044) for {'max_depth': 100, 'n_estimators': 400}
0.619 (+/-0.042) for {'max_depth': 100, 'n_estimators': 500}
0.618 (+/-0.025) for {'max_depth': 100, 'n_estimators': 1000}
0.623 (+/-0.034) for {'max_depth': 100, 'n_estimators': 2000}
0.605 (+/-0.005) for {'max_depth': 200, 'n_estimators': 100}
```

```
0.612 (+/-0.033) for {'max_depth': 200, 'n_estimators': 400}
0.619 (+/-0.037) for {'max_depth': 200, 'n_estimators': 500}
0.621 (+/-0.041) for {'max_depth': 200, 'n_estimators': 1000}
0.628 (+/-0.037) for {'max_depth': 200, 'n_estimators': 2000}

Detailed classification report:

The model is trained on the full development set.
The scores are computed on the full evaluation set.

              precision    recall  f1-score   support

         0.0       0.62      0.73      0.67        55
         1.0       0.57      0.62      0.59        52
         2.0       0.61      0.74      0.67        58
         3.0       0.59      0.52      0.56        61
         4.0       0.56      0.54      0.55        63
         5.0       0.60      0.64      0.62        50
         6.0       0.87      0.74      0.80        70
         7.0       0.83      0.64      0.72        69
         8.0       0.74      0.78      0.76        55
         9.0       0.70      0.73      0.72        67

    accuracy                           0.67       600
   macro avg       0.67      0.67      0.67       600
weighted avg       0.68      0.67      0.67       600
```

Once that the best configuration has been identified, the classifier can be trained with all the available data in the development set. In this way you can be sure that all the information available are exploited by the model when performing a prediction of new unknown data.

The best configuration found by the grid search is available in `best_clf.best_params_`

[26]:
```
clf = RandomForestClassifier(**best_clf.best_params_)

clf.fit(feat_dev_X, dev_y)
```

[26]:
```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
                       max_depth=200, max_features='auto', max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, n_estimators=2000,
                       n_jobs=None, oob_score=False, random_state=None,
                       verbose=0, warm_start=False)
```

2.3.2 Model performance on the evaluation set

WARNING: This section is for demonstration only! You do not have access to the evaluation set ground truth since these labels are used during the competition to evaluate your submitted solutions.

[27]:
```
eval_solution_path = "./dataset/eval_solution.csv"
eval_solution = np.loadtxt(eval_solution_path, delimiter =',', skiprows=1,␣
↪dtype={'names': ('Id', 'Expected', 'Usage'),
                                        'formats': ('int', 'int', 'object')})
eval_solution.shape
```

[27]:
```
(500,)
```

Apply the model built with development data to the evaluation data

[28]:
```
eval_pred_y = clf.predict(feat_eval_X)
eval_pred_y.shape
```

[28]:
```
(500,)
```

[29]:
```
eval_pred_y, eval_true_y = eval_pred_y, eval_solution["Expected"]
eval_pred_y.shape, eval_true_y.shape
```

[29]:
```
((500,), (500,))
```

Scores on public evaluation set

[30]:
```
print(classification_report(
    eval_true_y[eval_solution["Usage"] == "Public"],
    eval_pred_y[eval_solution["Usage"] == "Public"])
     )
```

```
              precision    recall  f1-score   support

           0       0.72      0.84      0.78        25
           1       0.68      0.60      0.64        25
           2       0.69      0.72      0.71        25
           3       0.62      0.52      0.57        25
           4       0.69      0.72      0.71        25
           5       0.83      0.60      0.70        25
           6       0.76      0.88      0.81        25
           7       0.83      0.76      0.79        25
           8       0.77      0.68      0.72        25
           9       0.62      0.84      0.71        25

    accuracy                           0.72       250
   macro avg       0.72      0.72      0.71       250
```

```
weighted avg       0.72      0.72      0.71       250
```

Scores on private evaluation set

[31]:
```
print(classification_report(
    eval_true_y[eval_solution["Usage"] == "Private"],
    eval_pred_y[eval_solution["Usage"] == "Private"])
     )
```

```
              precision    recall  f1-score   support

           0       0.78      0.84      0.81        25
           1       0.71      0.80      0.75        25
           2       0.56      0.72      0.63        25
           3       0.65      0.44      0.52        25
           4       0.67      0.56      0.61        25
           5       0.79      0.76      0.78        25
           6       0.78      0.72      0.75        25
           7       0.84      0.84      0.84        25
           8       0.73      0.76      0.75        25
           9       0.81      0.88      0.85        25

    accuracy                           0.73       250
   macro avg       0.73      0.73      0.73       250
weighted avg       0.73      0.73      0.73       250
```