Dataset Name: Free Spoken Digit Dataset (FSDD)

Team member

No	Name	code
1	Abdullah Khaled Abdullah	190090
2	Ahmed Mohammed Abdel Fattah	190015
3	Badr Mohammed Mohammed	190031
4	kerolos Atia Ghale	190118
5	Khaled Mahmoud Abdel Hamid	190048
6	Melad Malek Farg	190191

Speech Processing

Supervised by:

Dr. Hany Elnashar

Eng. Rawan

❖Abstract

- The **Free Spoken Digit Dataset (FSDD)** is a collection of audio recordings of spoken digits in wav files at 8kHz. It is a simple and easy-to-use dataset for speech recognition and audio processing tasks. Think MNIST (Modified National Institute of Standards and Technology database) for audio.
- The MNIST database (Modified National Institute of Standards and Technology database) is **a large database of handwritten digits that is commonly used for training various image processing systems.**
- The dataset consists of **3,000 recordings** of 10 digits (0-9) spoken by **6 speakers**. The recordings are trimmed so that they have near minimal silence at the beginnings and ends. The dataset is open and can grow over time as data is contributed.
- The dataset can be used for various applications such as speech recognition, speaker identification, audio classification, etc. It can also be used as a benchmark for comparing different models and methods on a simple and common task.

❖Introduction

- A simple audio/speech dataset consisting of recordings of spoken digits in `wav` files at 8kHz. The recordings are trimmed so that they have near minimal silence at the beginnings and ends.
- FSDD is an open dataset, which means it will grow over time as data is contributed. In order to enable reproducibility and accurate citation the dataset is versioned using Zenodo DOI as well as `git tags`.
- The test set officially consists of the first 10% of the recordings. Recordings numbered 0-4 (inclusive) are in the test and 5-49 are in the training set.

❖Dataset & The Goal with it

- **Dataset:** free-spoken-digit-dataset-master(FSDD).
- **Goal:** The goal of this dataset is to correctly identify the digit being uttered in each recording.

❖Current status

- 6 speakers
- 3,000 recordings (50 of each digit per speaker)
- English pronunciations

❖Organization

Files are named in the following format: {digitLabel}_{speakerName}_{index}.wav Example : 7_jackson_32.wav.

❖Contributions

- Please contribute your homemade recordings. All recordings should be mono 8kHz wav files and be trimmed to have minimal silence. Don't forget to update metadata.py with the speaker meta-data.
- To add your data, follow the recording instructions in acquire_data/say_numbers_prompt.py and then run split_and_label_numbers.py to make your files.

❖Metadata

metadata.py contains meta-data regarding the speakers gender and accents.

❖Included utilities

- trimmer.py Trims silences at beginning and end of an audio file. Splits an audio file into multiple audio files by periods of silence.

- fsdd.py A simple class that provides an easy to use API to access the data.

- spectogramer.py Used for creating spectrograms of the audio data. Spectrograms are often a useful pre-processing step.

❖**Coding**

Import Necessary Libraries

```
import numpy as np
from matplotlib import pyplot as plt
import seaborn as sns

from os import listdir
from os.path import join
from scipy.io import wavfile

import IPython.display as ipd
from librosa.feature import melspectrogram
from librosa import power_to_db
from librosa.effects import trim

# plotting utilities
plt.rcParams["figure.figsize"] = (8, 4)
plt.rcParams["figure.titleweight"] = 'bold'
plt.rcParams["figure.titlesize"] = 'large'
plt.rcParams['figure.dpi'] = 120
plt.style.use('fivethirtyeight')
rs = 99
```

NMPAY : is an extension to the Python programming language, used to handle large arrays and multi-level fields, as well as providing a large library of high-level mathematical functions to work on these fields and arrays.

Matplotlib : is a Python two-dimensional drawing gallery, which uses various printout formats and interactive cross-platform environments to create high-quality graphics.

Seaborn : helps you explore and understand your data.

listdir : is used to get the list of all files and directories in the specified directory.

os.path.join : combines path names into one complete path.

WAV files : can specify arbitrary bit depth, and this function supports reading any integer PCM depth from 1 to 64 bits.

IPython.display : module that runs the appropriate dunder method to get the appropriate data to ... display.

Spectrograms : are immensely useful tools that we can use to help dissect information from audio files and process it into images.

ower_to_db : Convert a **power** spectrogram (amplitude squared) to decibel (**dB**) units. This computes the scaling 10 * log10 (S / ref) in a numerically stable way.

librosa.effects.trim : is a function in the `librosa` library that trims leading and trailing silence from an audio signal.

Load data

The Free Spoken Digit Dataset is a collection of audio recordings of utterances of digits (“zero” to “nine”) from different people.

```
files = '/kaggle/input/free-spoken-digit-dataset-fsdd/recordings/'
ds_files = listdir(files)

X = []
y = []
for file in ds_files:
    label = int(file.split("_")[0])
    rate, data = wavfile.read(join(files, file))
    X.append(data.astype(np.float16))
    y.append(label)

len(x), len(y)
```

output:

```
(3000, 3000)
```

Basic EDA(Exploratory data analysis)

```
np.unique(y, return_counts = True)
```

output:

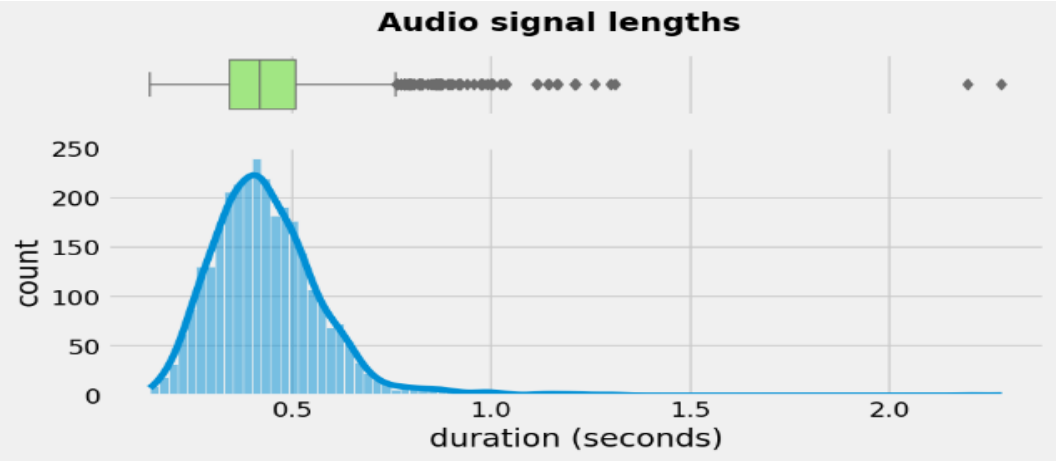
```
(array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]),
 array([300, 300, 300, 300, 300, 300, 300, 300, 300, 300]))
```

The problem is well balanced: for each of the classes we have 300 samples in dataset. All recordings are sampled at the rate of 8 kHZ

Audio signals have different length.
Some of them have leading and silence intervals. Let's analyze that first.

```
rate = 8000
def show_length_distribution(signals, rate = 8000):
    sampel_times = [len(x)/rate for x in signals]


    f, (ax_box, ax_hist) = plt.subplots(2, sharex=True, grid
spec_kw={"height_ratios": (.20, .80)})

    # Add a graph in each part
    sns.boxplot(x = sampel_times, ax=ax_box, linewidth = 0.9
, color=  '#9af772')
    sns.histplot(x = sampel_times, ax=ax_hist, bins = 'fd',
kde = True)

    # Remove x axis name for the boxplot
    ax_box.set(xlabel='')


    title = 'Audio signal lengths'
    x_label = 'duration (seconds)'
    y_label = 'count'

    plt.suptitle(title)
    ax_hist.set_xlabel(x_label)
    ax_hist.set_ylabel(y_label)
    plt.show()
    return sampel_times


lengths = show_length_distribution(X)
```

output:




```
q = 90
np.percentile(lengths, q)
```

output:

```
0.604525
```

```
tot_outliers = sum(map(lambda x: x > np.percentile(lengths, q), lengths))
print(f'Values outside {q} percentile: {tot_outliers}')
```

output:

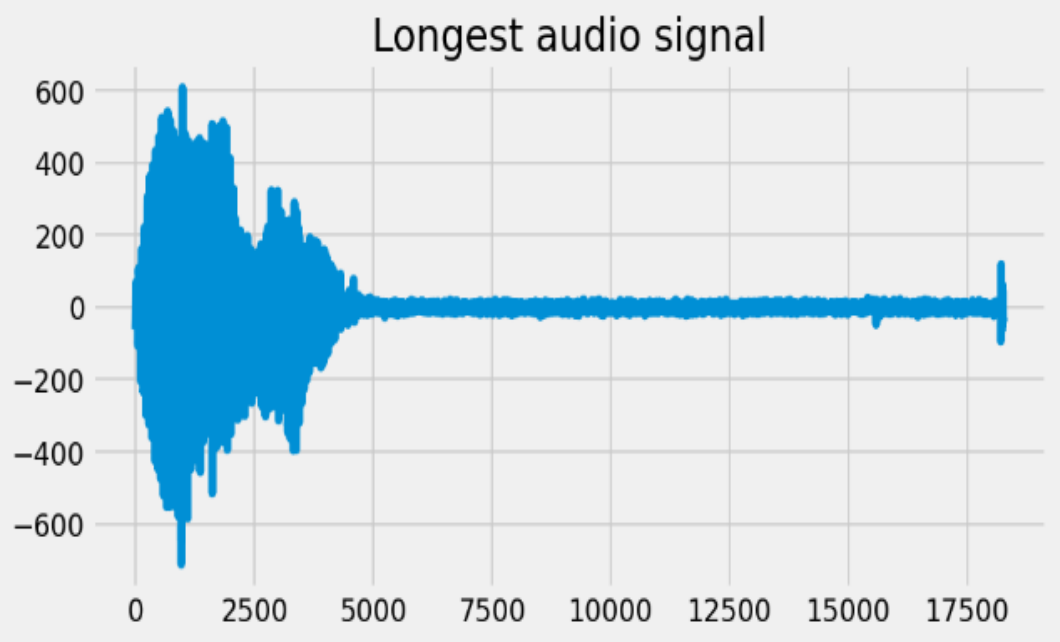
```
Values outside 90 percentile: 300
```

These outliers will be later handled according to the proposed solutions.

We can look at some extreme cases:

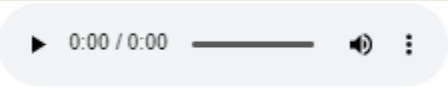
```
Longest_audio = np.argmax([len(x) for x in X])
plt.plot(X[Longest_audio])
plt.title("Longest audio signal");

ipd.Audio(X[Longest_audio], rate=rate)
```

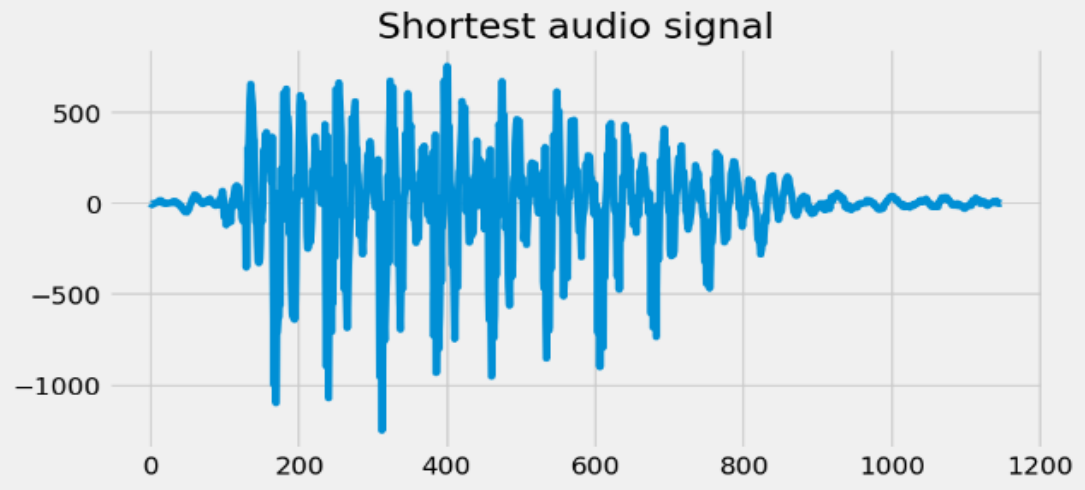
output:




```
Shortest_audio = np.argmin([len(x) for x in X])
plt.plot(X[Shortest_audio])
plt.title("Shortest audio signal");
ipd.Audio(X[Shortest_audio], rate=rate)
```

output:





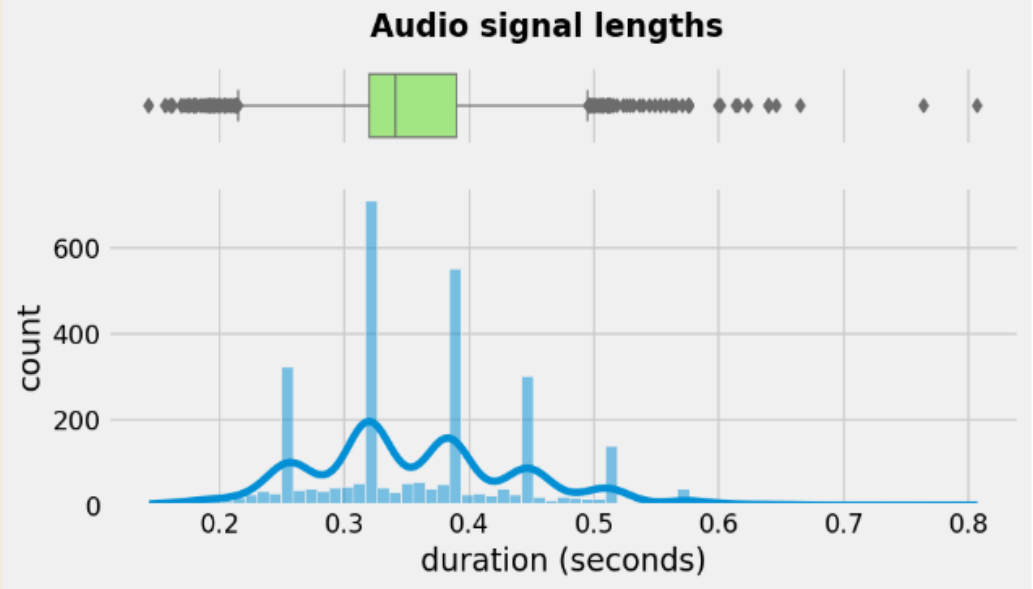
Time domain analysis

Feature Extraction from time domain:

99 percentile of audio length is around 0.92 seconds.
We will remove the leading and trailing silence from signals to see if we get different distribution of length.

```
# by default anything below 10 db is considered as silence
def remove_silence(sample, sr= 8000, top_db = 10):
    """This function removes trailing and leading silence pe
riods of audio signals.
    """
    y = np.array(sample, dtype = np.float64)
    # Trim the beginning and ending silence
    yt, _ = trim(y, top_db= top_db)
    return yt

X_tr = [remove_silence(x) for x in X]

show_length_distribution(X_tr);
```

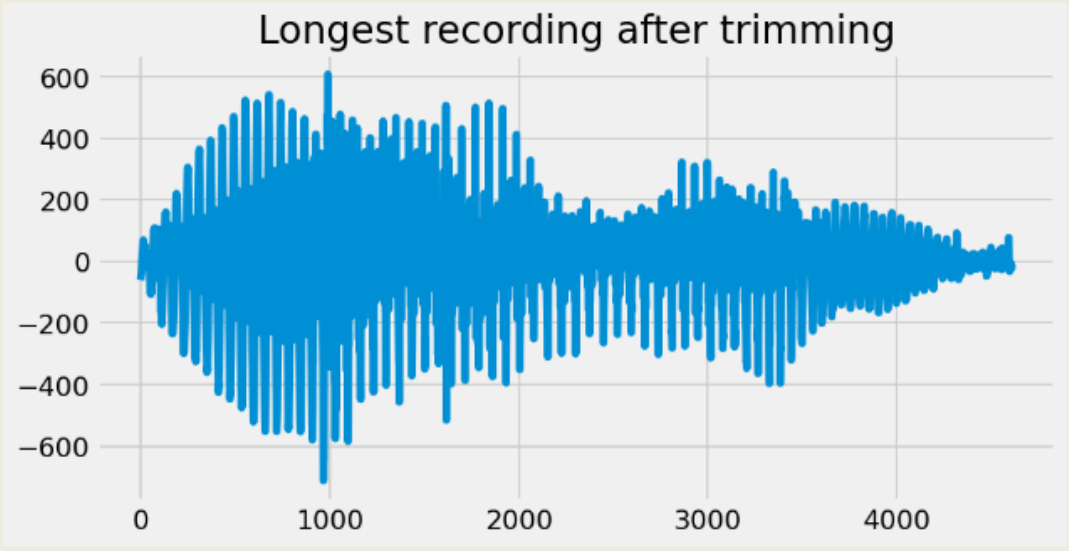
output:



We can explore different recordings to see how they are trimmed.

```
plt.plot(X_tr[Longest_audio])
plt.title("Longest recording after trimming");

ipd.Audio(X_tr[Longest_audio], rate=rate)
```

output:



We will create a matrix with uniform length of columns to allign all recordings.
All signals will have rate*0.8 data points.

```
N = int(rate * 0.8) # 0.8 is the upper limit of trimmed audi
o length
X_uniform = []
for x in X_tr:
    if len(x) < N:
        X_uniform.append(np.pad(x, (0, N - len(x)), constant
_values = (0, 0)))
    else:
        X_uniform.append(x[:N])
```

```
def into_bins(X, bins = 20):
    """This functions creates bins of same width and compute
s mean and standard deviation on those bins
    """
    X_mean_sd = []
    for x in X:
        x_mean_sd = []
        As = np.array_split(np.array(x), 20)
        for a in As:
            mean = np.round(a.mean(dtype=np.float64), 4)
            sd = np.round(a.std(dtype=np.float64), 4)
            x_mean_sd.extend([mean, sd])

        X_mean_sd.append(x_mean_sd)
    return np.array(X_mean_sd)
```

Model building

```
from sklearn.ensemble import RandomForestClassifier as RFC
from sklearn.metrics import accuracy_score, precision_recall
_fscore_support, classification_report
```

```
from sklearn.model_selection import train_test_split, GridSe
archCV, cross_val_score

from sklearn import svm
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
```

RandomForestClassifier :A random forest is a meta estimator that fits a number of decision tree classifiers on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting.

The sklearn.metrics.accuracy_score: is used to calculate the accuracy ratio of the classification model.

sklearn.metrics.precision_recall_fscore_support: is a function in the scikit-learn library that computes precision, recall, F-measure and support for each class.

sklearn.metrics.classification_report: is a function in the scikit-learn library that generates a text report showing the main classification metrics.

sklearn.model_selection.train_test_split: is a way to split the dataset into two sets: a training set and a test set.

sklearn.model_selection. GridSearchCV: is a way to optimize automated forms by looking for the best set of parameters that achieve the highest accuracy.

sklearn.model_selection .cross_val_score: is a way to evaluate the performance of a machine learning model using cross-verification.

sklearn.svm: is a package that provides support vector machine algorithms for classification and other tasks in Python.

sklearn.preprocessing.StandardScaler: is to convert numerical data into standard form, where the mean is zero

sklearn.pipeline.Pipeline: is a way to organize several steps of data processing and machine learning into a single chain. he standard deviation is one.

Number of bins is an hyperparameter.
We will try different n. of bins with default configurations of Random Forest Classifier.

```
for bins in range(20,101,20):
    X_mean_sd = into_bins(X_uniform, bins)
    X_train, X_test, y_train, y_test = train_test_split(X_me
an_sd, y, test_size = 0.20, random_state = rs)
    clf = RFC()
    clf.fit(X_train, y_train)
    y_pred = clf.predict(X_test)
    acc = accuracy_score(y_test, y_pred)
    p,r,f,s = precision_recall_fscore_support(y_test, y_pred
)
    print(f"for {bins} bins, f-macro average:{f.mean()}, acc
uracy: {acc}")
```

output:

```
for 20 bins, f-macro average:0.5642843931371829, accuracy: 0.5716666666666667
for 40 bins, f-macro average:0.5742353303644151, accuracy: 0.5816666666666667
for 60 bins, f-macro average:0.5740077439400117, accuracy: 0.5816666666666667
for 80 bins, f-macro average:0.5687754122280697, accuracy: 0.575
for 100 bins, f-macro average:0.5667429269520836, accuracy: 0.575
```

With 60 bins we are able to get comparable results.
Now we will train two models via grid search to optimize the configuration.

Hyperperameter tuning

```
X_time = into_bins(X_uniform, 60)
X_train, X_test, y_train, y_test = train_test_split(X_time,
y, test_size = 0.20, random_state = rs)
```

Random Forest Classifier

```
param_grid = {
    "n_estimators": [100,150,200],
    "criterion": ["gini", "entropy"],
    "min_impurity_decrease": [0.0,0.05,0.1]
}

clf = RFC(random_state = rs, n_jobs = -1 )
grid_search = GridSearchCV(clf, param_grid, scoring = "f1_ma
cro", cv= 5)
grid_search.fit(X_train, y_train)
print("best Parameters for RF model:\n", grid_search.best_pa
rams_)
print("best score:", grid_search.best_score_)
print("\n\n Results on test dataset:\n\n")
y_pred = grid_search.predict(X_test)
print(classification_report(y_test, y_pred))
```

output:

```
best Parameters for RF model:
 {'criterion': 'gini', 'min_impurity_decrease': 0.0, 'n_esti
mators':150}
best score: 0.5819761083994751


 Results on test dataset:
```

```
              precision    recall  f1-score   support

           0       0.66      0.72      0.69        60
           1       0.56      0.42      0.48        60
           2       0.58      0.54      0.56        61
           3       0.41      0.45      0.43        56
           4       0.33      0.38      0.35        58
           5       0.61      0.37      0.46        62
           6       0.69      0.84      0.76        49
           7       0.75      0.67      0.71        67
           8       0.68      0.77      0.72        66
           9       0.59      0.72      0.65        61

    accuracy                           0.59       600
   macro avg       0.59      0.59      0.58       600
weighted avg       0.59      0.59      0.58       600
```

Support Vector machines

```
steps = [('scaler', StandardScaler()), ('SVM', svm.SVC())]
pipeline = Pipeline(steps)

parameteres = {'SVM__C':[5,10,20], 'SVM__kernel':["linear",
"poly", "rbf"]}
grid_search = GridSearchCV(pipeline, param_grid=parameteres,
cv=5)
grid_search.fit(X_train, y_train)

print("best Parameters for RF model:\n", grid_search.best_pa
rams_)
print("best score:", grid_search.best_score_)
print("\n\n Results on test dataset:\n\n")
y_pred = grid_search.predict(X_test)
print(classification_report(y_test, y_pred))
```

output:

```
best Parameters for RF model:
 {'SVM__C': 20, 'SVM__kernel': 'rbf'}
best score: 0.412

Results on test dataset:

              precision    recall  f1-score   support

           0       0.44      0.48      0.46        60
           1       0.46      0.27      0.34        60
           2       0.35      0.28      0.31        61
           3       0.41      0.32      0.36        56
           4       0.51      0.33      0.40        58
           5       0.59      0.31      0.40        62
           6       0.20      0.86      0.32        49
           7       0.77      0.40      0.53        67
           8       0.52      0.41      0.46        66
           9       0.68      0.43      0.53        61

    accuracy                           0.40       600
   macro avg       0.49      0.41      0.41       600
weighted avg       0.50      0.40      0.41       600
```

Results:

We are able to set a baseline for other models. The baseline accuracy and f1 macro average are 0.59 and 0.58 respectively.

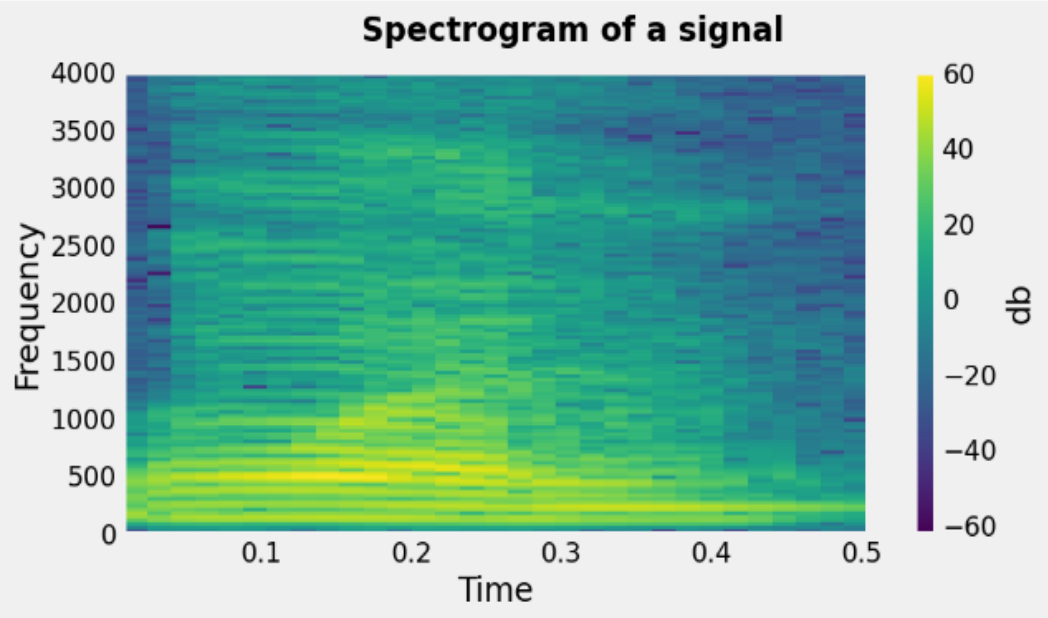
Spectorgrams

In a spectoral representation of audio signals, we get time on x-axis and different frequencies on y-axis. Values in the matrix represent different properties of audio singal related to particular time and frequency. (amplitude, power ecc)

```
# Plot the spectrogram of power on log scale

# fig, ax = plt.subplots(figsize = (8,6))
```

```
powerSpectrum, freqenciesFound, time, imageAxis = plt.specgr
am(X[np.random.randint(100)], Fs=rate, scale = "dB")
cbar = plt.gcf().colorbar(imageAxis)
cbar.set_label('db')
plt.grid()
plt.suptitle("Spectrogram of a signal")
plt.xlabel('Time')
plt.ylabel('Frequency')
plt.show()
```

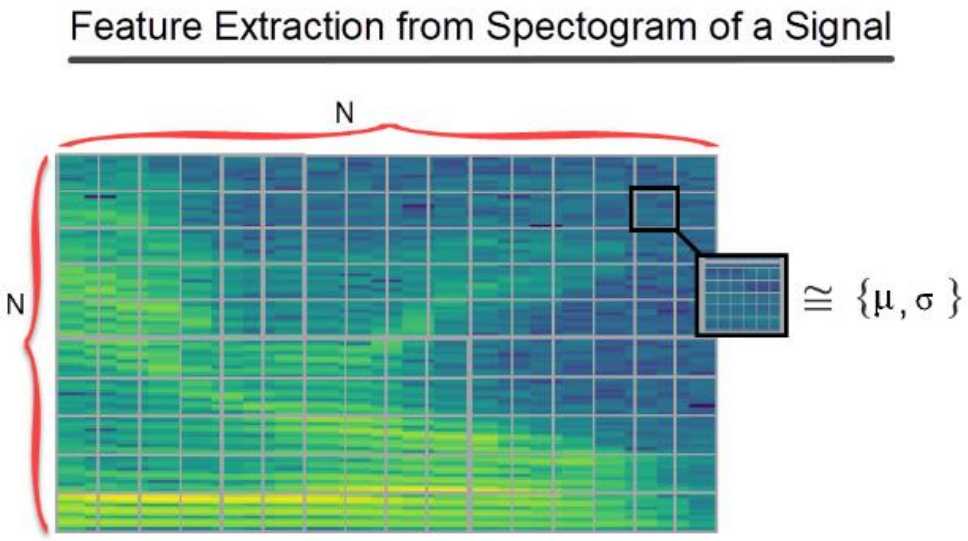
output:



Feature extraction from Power spectogram

We have seen that both time and frequency domains contain useful information regarding the recordings.
We can leverage both by using the spectrogram of each signal.

To extract features from a specogram of given signal, we divide it into N x N sub matrices of nearly identical shape.
Later, we compute mean and standard deviation of these submatrices and consider them as features set.
Number of sub matrices is considered as an hyperparameter for classifier.




```
      def ft_mean_std(X, n, f_s = 8000):
    """Computes mean and std of each n x n block of spectrog
rams of X
       empty bins contains mean values of that column matric
es

    Parameters:
        X: 2-d sampling array
        n: number of rows or columns to split spectogram
    Returns:
        A 2-d numpy array - feature Matrix with n x 2 x n fe
atures as columns
    """
    X_sp = [] #feature matrix
    for x in X:
```

```
        sp = power_to_db(melspectrogram(x, n_fft= len(x)), n
p.mean)
        x_sp = [] #current feature set
        # split the rows
        for v_split in np.array_split(sp, n, axis = 0):
            # split the columns
            for h_split in np.array_split(v_split, n, axis =
1):
                if h_split.size == 0: #happens when number o
f columns < n
                    m = np.median(v_split).__round__(4)
                    sd = np.std(v_split).__round__(4)
                else:
                    m = np.mean(h_split).__round__(4)
                    sd = np.std(h_split).__round__(4)
                x_sp.extend([m,sd])

        X_sp.append(x_sp)

    return np.array(X_sp)
```

```
X_ft = ft_mean_std(X, 10)
len(X_ft)
```

```
output:
    3000
```

Hyperparameters tuning

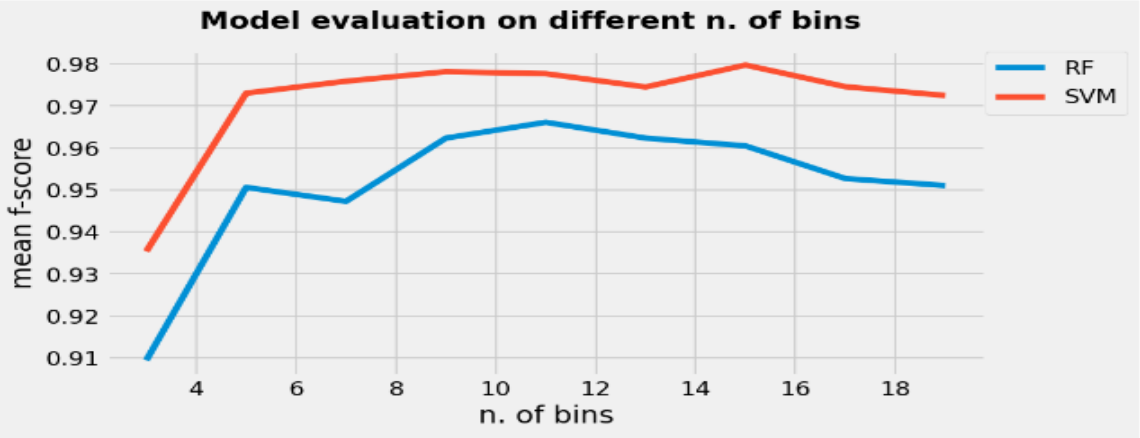
Number of bins

```
      models = {
    "rfc": RFC(random_state=rs),
    "svm": Pipeline([('scaler', StandardScaler()), ('SVM', s
vm.SVC())])
}
scores = {}
```

```
for n in range(3,20,2):
    X_ft = ft_mean_std(X, n)
    X_train, X_test, y_train, y_test = train_test_split(X_ft
, y, test_size = 0.20, random_state = rs)
    score = []
    for model in models:
        clf = models[model]
        clf.fit(X_train, y_train)
        y_pred = clf.predict(X_test)
        p,r,f,s = precision_recall_fscore_support(y_test, y_
pred)
        score.append((model, np.mean(f)))
    scores[n] = score
```

```
      rf_scores = [x[0][1] for x in scores.values()]
svm_scores = [x[1][1] for x in scores.values()]
x = scores.keys()

plt.plot(x, rf_scores, label = 'RF')
plt.plot(x, svm_scores, label= 'SVM')

plt.legend(loc = (1,.8))
plt.suptitle("Model evaluation on different n. of bins")
plt.xlabel("n. of bins")
plt.ylabel('mean f-score')
plt.show()
```

output:



We can select 10 as initial number of bins. Both models are stable in the neighborhood of 10.
we can check the performance of models with their optimal configurations.

```
X_ft = ft_mean_std(X, 10)
X_train, X_test, y_train, y_test = train_test_split(X_ft, y,
test_size = 0.20, random_state = rs)
```

Classification models

Random forest classifier

```
param_grid = {
    "n_estimators": [100,150,200],
    "criterion": ["gini", "entropy"],
    "min_impurity_decrease": [0.0,0.05,0.1]
}

clf = RFC(random_state = rs, n_jobs = -1 )
rf_search = GridSearchCV(clf, param_grid, scoring = "f1_macr
o", cv = 5)
rf_search.fit(X_train, y_train)

print("best Parameters for RF model:\n", rf_search.best_para
ms_)
print("best score:", rf_search.best_score_)
print("\n\n Results on test dataset:\n\n")
y_pred = rf_search.predict(X_test)
print(classification_report(y_test, y_pred))
```

output:

```
best Parameters for RF model:
 {'criterion': 'entropy', 'min_impurity_decrease': 0.0, 'n_e
stimators': 200}
best score: 0.9580000183550087
```

```
 Results on test dataset:


              precision    recall  f1-score   support

           0       0.98      0.97      0.97        60
           1       1.00      1.00      1.00        60
           2       0.98      0.95      0.97        61
           3       0.89      0.98      0.93        56
           4       1.00      1.00      1.00        58
           5       1.00      1.00      1.00        62
           6       0.93      0.88      0.91        49
           7       0.96      0.96      0.96        67
           8       0.97      0.97      0.97        66
           9       1.00      1.00      1.00        61

    accuracy                           0.97       600
   macro avg       0.97      0.97      0.97       600
weighted avg       0.97      0.97      0.97       600
```

```
rfc = RFC(n_estimators= 200, criterion= 'gini', min_impurity
_decrease= 0.0,random_state = rs, n_jobs = -1 )
scores = cross_val_score(rfc, X_ft, y, cv=10, scoring = 'acc
uracy', n_jobs = -1)
report = f"""Average accuracy of Random Forest model: {np.me
an(scores):.2f}
with a standard deviation of {np.std(scores):.2f}
"""
print(report)
```

output:

```
Average accuracy of Random Forest model: 0.96
with a standard deviation of 0.01
```

Support vector classifier

```
steps = [('scaler', StandardScaler()), ('SVM', svm.SVC())]
pipeline = Pipeline(steps)

parameteres = {'SVM__C':[5,10,20], 'SVM__kernel':["linear",
"poly", "rbf"]}
svm_search = GridSearchCV(pipeline, param_grid=parameteres,
cv=5)
svm_search.fit(X_train, y_train)

print("best Parameters for RF model:\n", svm_search.best_par
ams_)
print("best score:", svm_search.best_score_)
print("\n\n Results on test dataset:\n\n")
y_pred = svm_search.predict(X_test)
print(classification_report(y_test, y_pred))
```

output:

```
best Parameters for RF model:
 {'SVM__C': 5, 'SVM__kernel': 'rbf'}
best score: 0.9820833333333333


 Results on test dataset:


              precision    recall  f1-score   support

           0       0.98      0.98      0.98        60
           1       0.98      1.00      0.99        60
           2       0.98      0.98      0.98        61
           3       0.96      0.93      0.95        56
           4       1.00      1.00      1.00        58
           5       0.98      0.97      0.98        62
           6       0.92      0.96      0.94        49
           7       0.98      0.97      0.98        67
           8       0.97      0.98      0.98        66
```

```
           9       0.98      0.98      0.98        61

    accuracy                           0.98       600
   macro avg       0.98      0.98      0.98       600
weighted avg       0.98      0.98      0.98       600
```

```
steps = [('scaler', StandardScaler()), ('SVM', svm.SVC(C= 20
, kernel= 'rbf'))]
pipeline = Pipeline(steps)
scores = cross_val_score(pipeline, X_ft, y, cv=10, scoring =
'accuracy', n_jobs = -1)
report = f"""Average accuracy of SVM model: {np.mean(scores)
:.2f}
with a standard deviation of {np.std(scores):.2f}
"""
print(report)
```

output:

```
Average accuracy of SVM model: 0.98
with a standard deviation of 0.01
```

Results

Although results are quite satisfactory, we can use other techniques to split the spectogram matrix.
One other way of splitting spectorgram is to *pad* it such that each sub matrix has identical shape.
This way we also avoid the for-loops which is performance killer.

```
def split(array,w_bins):
    """Split a matrix into sub-matrices of equal size."""

    # original dimensions
    rows, cols = array.shape
    # size of sub matrices
```

```
    sub_rows = rows//w_bins + 1 * rows%w_bins
    sub_cols = cols//w_bins + 1 * cols%w_bins
    # padding to properly fit
    pad_rows = sub_rows*w_bins - rows
    pad_cols = sub_cols*w_bins - cols
    padded_array = np.pad(array, ((0,pad_rows), (0, pad_cols
)))

    rows, cols = padded_array.shape
    return (padded_array.reshape(rows//sub_rows, sub_rows, -
1, sub_cols)
                .swapaxes(1, 2)
                .reshape(-1, sub_rows, sub_cols))


def split_ft_mean_std(X, n):
    """ Computes mean and std of each n x n block of spectro
grams of X
        bins are padded with zeros to equaly divide in n x n
matrices.

    Parameters:
        X: 2-d sampling array
        n: number of rows or columns to split spectogram
    Returns:
        A 2-d numpy array - feature Matrix with n x n x 2 fe
atures
    """
    f_s = 8000
    X_sp = [] #feature matrix
    for x in X:
        sp = power_to_db(melspectrogram(x, n_fft= len(x)), n
p.mean)
        blocks = split(sp,n)
        mean = blocks.mean(axis = (-1,-2))
        std = blocks.std(axis = (-1,-2))
        X_sp.append(np.hstack((mean,std)))
    return np.array(X_sp)
```

```
%timeit -n2 -r1 ft_mean_std(X, 10)
```

Output:

```
55.6 s ± 0 ns per loop (mean ± std. dev. of 1 run, 2 loops each) In [35]:
```

```
%timeit -n2 -r1 split_ft_mean_std(X, 10)
```

Output:

```
28.7 s ± 0 ns per loop (mean ± std. dev. of 1 run, 2 loops each)
```

As expected, this new method is twice as fast as the previous one.
Let's compare the results:

```
steps = [('scaler', StandardScaler()), ('SVM', svm.SVC())]
pipeline = Pipeline(steps)

parameteres = {'SVM__C':[5,10,20], 'SVM__kernel':["linear", "poly", "rbf"]}
```

```
X_ft = split_ft_mean_std(X, 10)
X_train, X_test, y_train, y_test = train_test_split(X_ft, y, test_size = 0.20, random_state = rs)

svm_search = GridSearchCV(pipeline, param_grid=parameteres, cv=5)
svm_search.fit(X_train, y_train)
```

```
print("best Parameters for RF model:\n", svm_search.best_par
ams_)
print("best score:", svm_search.best_score_)
print("\n\n Results on test dataset:\n\n")
y_pred = svm_search.predict(X_test)
print(classification_report(y_test, y_pred))
```

output:

```
best Parameters for RF model:
 {'SVM__C': 20, 'SVM__kernel': 'rbf'}
best score: 0.8891666666666665

 Results on test dataset:


              precision    recall  f1-score   support

           0       0.92      0.92      0.92        60
           1       0.87      0.80      0.83        60
           2       0.92      0.90      0.91        61
           3       0.89      0.89      0.89        56
           4       0.95      0.95      0.95        58
           5       0.89      0.95      0.92        62
           6       0.80      0.90      0.85        49
           7       0.88      0.85      0.86        67
           8       0.94      0.95      0.95        66
           9       0.90      0.85      0.87        61

    accuracy                           0.90       600
   macro avg       0.90      0.90      0.90       600
weighted avg       0.90      0.90      0.90       600
```

```
steps = [('scaler', StandardScaler()), ('SVM', svm.SVC(C= 20
, kernel= 'rbf'))]
pipeline = Pipeline(steps)
```

```
scores = cross_val_score(pipeline, X_ft, y, cv=10, scoring =
'accuracy', n_jobs =-1)
report = f"""Average accuracy of SVM model: {np.mean(scores)
:.2f}
with a standard deviation of {np.std(scores):.2f}
"""
print(report)
```

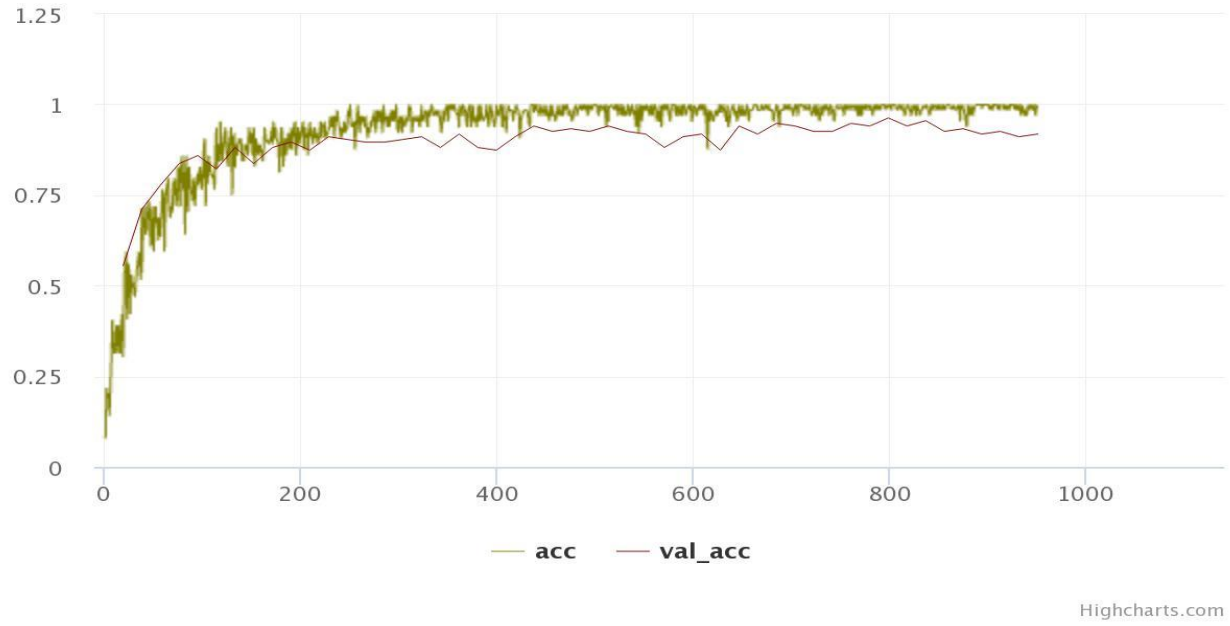
output:

```
Average accuracy of SVM model: 0.90
with a standard deviation of 0.02
```

We get comparable results but model is more efficient now.
The reuslts can be improved by tuning the appropriate number of splits (as we did earlier)

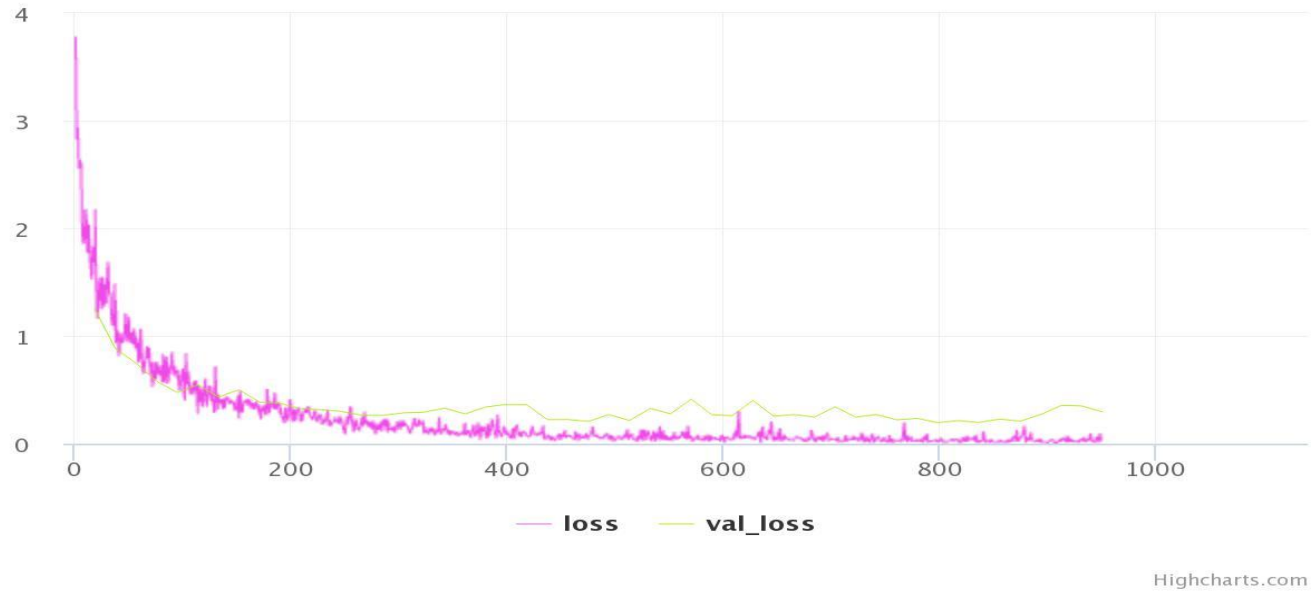
training

- Accuracy



Model Accuracy

- Loss



Model Loss

We get 98% validation accuracy!

Concluisons

The proposed approach obtains results that are outperforming naive baseline we defined in the beginning.
It does so by leveraging both time and frequency-based features. We have empirically shown that the selected classifiers perform similarly for this specific task, achieving satisfactory results in terms of macro f1 score and accuracy. We can further improve by using different set of hyper parameters. The results obtained, however, are already very promising. This classification problem is indeed quite easy and the datasets available are very limited.

Next, let's try the following steps to improvise:

1. We can go with the hybrid approach. We can take pre-trained embeddings that act as the base vectors and build a classifier on top of it a transfer learning approach. We explore this concept in later chapters of this book.

2. We can also try different and advanced deep learning architectures like attention mechanism instead of simple LSTM with dropout