EX1:

In this exercise we experimented with the different value data-types including FloatList, Int64List and BytesList. The temperature is 0-50 and the humidity is 20-90, python can use int8, so the Int64List type performs better as it covers int8 so it's more efficient compared to others. We tested the usage of Normalization on both Temperature and Humidity features and we reached the point that the best way to compress the data set without losing info is not using normalization and setting Temperature as Int, Humidity as Int and Time as Float. The results obtained after using the mentioned configuration was a TFRecord file with size of 405 B for 5 records which csv size was 135 B, and we considered a larger file with size of 3807 B for 46 records which csv size was 1269 B, the reason that the TFRecords are larger than csv files is that TFRecord format duplicates the feature key in each record, so we would expect the uncompressed TFRecord file to actually be bigger. The efficient way to write the data is without normalization, and if the normalization is required it should be included in the model preprocessing pipeline, because we are not using computed values from the training data and we are using stable values like min or max for Temperature and Humidity, we can apply the same preprocessing pipeline for training and inference.

Ex2:

We began by creating the pipeline to obtain MFCC-slow according to the parameters provided in the instructions and attained a shape of (W=124,nb_bins=40) but we made it (124,10) by taking only 10 coefficients. Where $(W = \frac{L-l}{s} + 1)$. Next, we identified all the parameters that we could potentially tune in order to satisfy all 3 conditions related to MFCC-fast. First, We tried to study the effects of l (window length) and s (stride). To maintain the required output shape that we got from MFCC-slow, we needed to satisfy the following equation: $(l = 1 - 123*s)$. This indicates that both parameters are inversely proportional, which implies low contribution to the overall execution time and quality. We tested it by trying to increase l and decrease s accordingly, and vice versa. However, the impact on the execution time and quality were negligible. As for the number of coefficients, we decided to keep it at 10 to maintain the shape of the MFCC-fast.As for the remaining parameters, we decided to run a grid search on them. These parameters were: nb_bins, lower_freq, upper_freq, and sampling rate. We also made sure to respect some constraints:

1) Lower frequency < Upper Frequency
2) Upper frequency <= Nyquist frequency (which is half of the sampling rate).

We focused on having more lower frequency values since they hold more information, and since the human voice frequency lies in the range of 60 Hz – 300 Hz. Finally, for the number of Mel filters (bins), we tried all even values between 20 and 40. (40 was the upper limit since we used it for MFCC-slow and more bins --> Higher SNR but also higher execution time). For the grid search, we only selected 2 values for the sampling rate: 8 kHz and 16 kHz since we knew that decreasing the sampling rate would speed up the execution, but it would negatively affect the quality. The frequency values provided for the grid search are provided in the python script. The results of the grid search are summarized below:

	# bins	lower_freq	upper_freq	sampling_rate	snr	time
1	32	200	3000	16000	16.854	17.989
2	32	110	2600	16000	16.217	17.983
3	32	110	2200	16000	15.830	17.989
4	32	80	2000	16000	15.256	17.989
5	32	400	4000	16000	15.048	17.986

We selected the configuration that maximized the SNR while respecting the time constraint: Length = 0.016 ms , Stride = 0.008 ms # Bins = 32 , Lower_freq = 200 Hz , Upper_freq = 3 kHz , sampling_rate = 16 kHz ,nb_coef= 10. SNR = 16.854 dB , Execution_time_fast = 17.989 ms, Execution_time_slow = 25.756 ms.