2. Questions

Considering the content of the three input files *s1.csv*, *s2.csv*, and *s3.csv*, one for each hypothetical Biometric system, please answer the following questions. You may leverage and adapt the functions and metrics available in *metrics.py*.

2.1. For each one of the three Biometric systems, what score threshold (a.k.a. operating point) should you use? Please explain your answer and describe how you have obtained each one of the respective system thresholds. (2 points)

The following is the threshold I would use for each system:

Name	Threshold
s1.csv	62.958
s2.csv	0.5849
s3.csv	2.3469

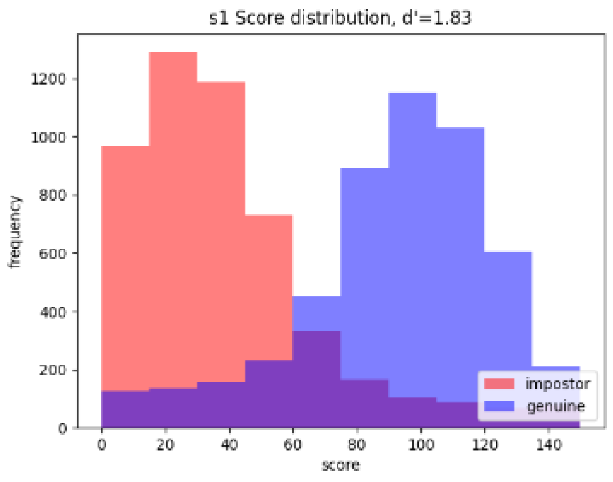
I arrived at these numbers for the threshold by computing the value at which the FMR and FNMR values were equal. In other words, I computed the threshold that would be needed to achieve EER (equal error rate).

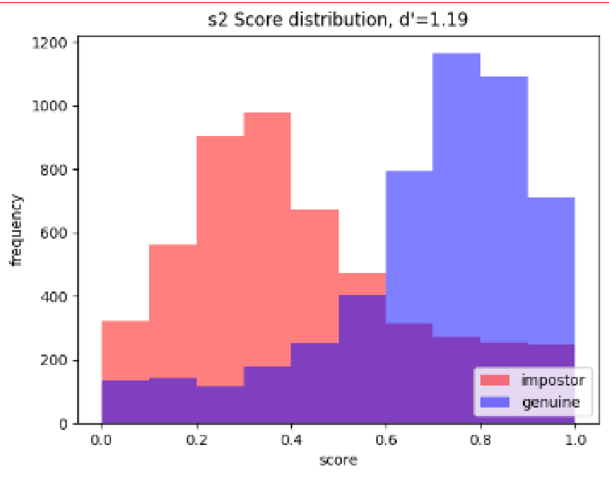
For models s1 and s2, these values could be computed by utilizing the given function, `compute_sim_fmr_fnmr_eer` provided in the metrics python file. The value for s3 was computed by adapting the functions for computing FMR and FNMR at a given threshold for distances instead of similarities. The process for computing the threshold at EER is nearly identical to s1 and s2, with the only difference being the utilization of the adapted functions.

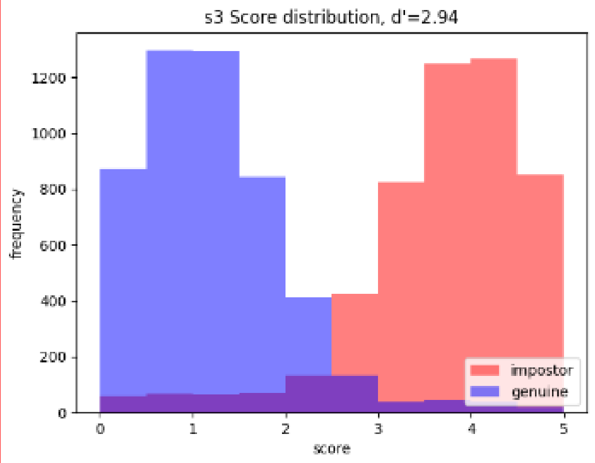
I decided to use the threshold at EER simply because not much is known about the nature of these models, so it would be safe to choose a value at which the errors are equivalent. However, as discussed in class, with more information it might be preferable for one type of error to be smaller than the other.

2.2. For each system, plot and provide a graph with the distributions of their respective scores. (1 point)

Graphs:





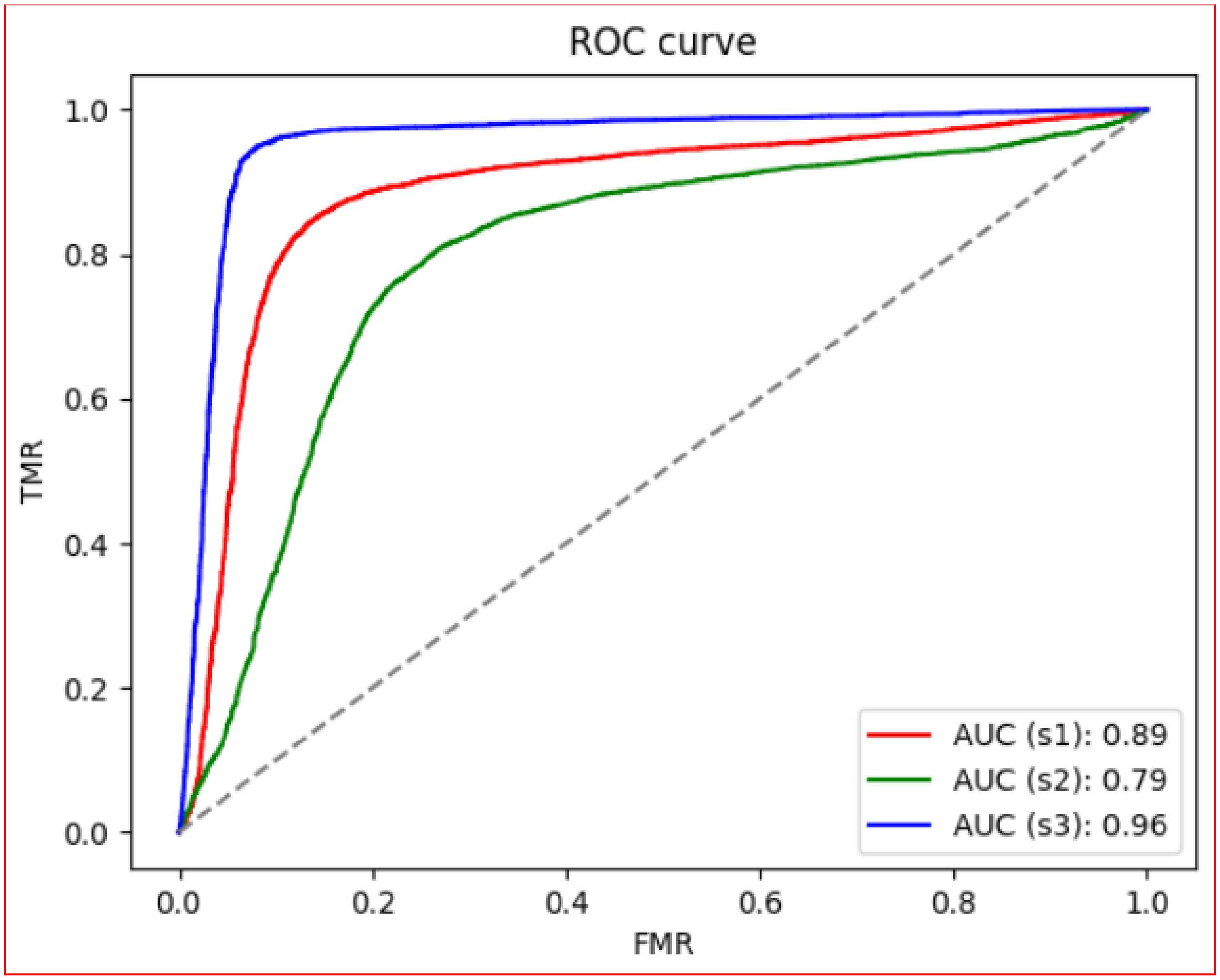


2.3. According to the d' (d-prime) values that one might compute for each system, which of the three should you use if you had to select only one for identification? Please justify your answer. (2 points)

According to the d' values of the three systems, System 3 is the most ideal because it has the highest d' value, indicating that there is a greater separation between the impostor and genuine distributions, meaning that this system is better at distinguishing between the two groups.

2.4. Plot and provide a single graph with the ROC curves and AUCs of all the three systems together. A reference to help you: https://bit.ly/3rrnSPR. (1 point)

Graph:



2.5. According to the ROC curves and AUC values, which one of the three systems should you use if you had to select only one for identification? Please justify your answer. (2 points)

I would choose s3 because it has the highest AUC value (as seen in the graph above). A higher AUC value indicates a lower FMR and higher TMR (or lower FNMR), and this can be seen with the actual curves as well, since the blue curve, representing s3, is closest to the edges.

2.6. Please provide your modified version of *metrics.py*, adapted to deal with scores that are distances rather than similarities. (2 points)

```
import math
import matplotlib.pyplot as plt

# Sums all the values in the given list, using pairwise summation to reduce round-off error.
def _pairwise_sum(values):
    if len(values) == 0:
        return 0.0  # nothing to sum, return zero

    elif len(values) == 1:
        return float(values[0])  # one element, returns it

    elif len(values) == 2:
        return float(values[0]) + float(values[1])  # two elements, returns their sum

    else:
        i = int(len(values) / 2)
        return _pairwise_sum(values[0:i]) + _pairwise_sum(values[i:len(values)])  # recursive call


# Computes the variance of the given values.
def _compute_var(values):
    if len(values) == 0:
        return float('NaN')  # nothing to compute, returns not-a-number

    # mean of values
    mean = _pairwise_sum(values) / len(values)

    # deviations
    deviations = [(v - mean) ** 2.0 for v in values]

    # variance
    var = _pairwise_sum(deviations) / len(deviations)
    return var
```

```python
# Loads data from the CSV file stored in the given file path.
# Expected file line format: <label>,<score>
# Comment lines starting with "#" will be ignored.
# Output: array of (<label>,<score>) elements.
def load_data(file_path):
    # output
    output = []

    # reads each line of the file, ignoring empty lines and the ones starting with '#'
    with open(file_path) as f:
        for line in f:
            content = line.strip().split(',')
            if len(content) > 0 and len(content[0]) > 0 and content[0][0] != '#':
                label = int(content[0])
                score = float(content[1])

                output.append((label, score))

    return output


# Computes d-prime for the given observations.
# Observations must be an array of (<label>,<score>) elements.
# Labels must be either 0 (impostor) or something else (genuine).
# If either the number of impostors or genuine observations is zero, it returns 'NaN' as d-prime.
def compute_d_prime(observations):
    # separates genuine and impostor scores
    genuine_scores = []
    impostor_scores = []

    for obs in observations:
        if obs[0] == 0:  # impostor observation
            impostor_scores.append(obs[1])
        else:  # genuine observation
            genuine_scores.append(obs[1])

    if len(genuine_scores) == 0 or len(impostor_scores) == 0:
        return float('NaN')

    # computes mean values
    genuine_mean = _pairwise_sum(genuine_scores) / len(genuine_scores)
    impostor_mean = _pairwise_sum(impostor_scores) / len(impostor_scores)

    # computes variances
    genuine_var = _compute_var(genuine_scores)
    impostor_var = _compute_var(impostor_scores)

    # computes d-prime
    d_prime = math.sqrt(2.0) * abs(genuine_mean - impostor_mean) / math.sqrt(genuine_var + impostor_var)
    return d_prime

#######################################################
##       MODIFIED METHODS THAT ACCOUNT FOR USING     ##
##         DISTANCE AS WELL AS SIMILARITY SCORE      ##
#######################################################
def compute_fmr(observations, threshold, metric):
    impostor_count = 0
    false_match_count = 0

    for obs in observations:
        if obs[0] == 0:  # impostor observation
            impostor_count = impostor_count + 1

            if metric == 'score': # Check which metric is being used and adjust comparison accordingly
                if obs[1] >= threshold:
                    false_match_count = false_match_count + 1

            if metric == 'distance':
                if obs[1] < threshold:
                    false_match_count = false_match_count + 1

    if impostor_count > 0:
        return float(false_match_count) / impostor_count
    else:
        return float('NaN')


def compute_fnmr(observations, threshold, metric):
    genuine_count = 0
    false_non_match_count = 0
    for obs in observations:
        if obs[0] != 0:  # genuine observation
            genuine_count = genuine_count + 1

            if metric == 'score': # Check which metric is being used and adjust comparison accordingly
                if obs[1] < threshold:
                    false_non_match_count = false_non_match_count + 1

            if metric == 'distance':
                if obs[1] >= threshold:
                    false_non_match_count = false_non_match_count + 1

    if genuine_count > 0:
        return float(false_non_match_count) / genuine_count
    else:
        return float('NaN')


def compute_fmr_fnmr_eer(observations, metric):
    # computed FNMR and FMR at EER, and their difference, and EER threshold
    output_fnmr = float('inf')
    output_fmr = float('inf')
    fnmr_fmr_diff = float('inf')
    output_threshold = float('inf')

    # sorted list of scores
    scores = [obs[1] for obs in observations]
    scores = sorted(scores)
    if len(scores) == 0:
        return float('NaN'), float('NaN'), float('NaN')  # nothing to do here

    # for each score taken as threshold
    for threshold in scores:
        current_fnmr = compute_fnmr(observations, threshold, metric) # FMR and FNMR are calculated based
        current_fmr = compute_fmr(observations, threshold, metric)   # on which metric we want to use

        # cancels computation if any of the FNMR or FMR values are 'NaN' (not possible to compute them)
        if not float('-inf') < current_fnmr < float('inf') or
           not float('-inf') < current_fmr < float('inf'):
            return float('NaN'), float('NaN'), float('NaN')  # nothing to do here

        # updates the difference between FNMR and FMR, if it is the case
        current_diff = abs(current_fnmr - current_fmr)
        if current_diff <= fnmr_fmr_diff:
            output_fnmr = current_fnmr
            output_fmr = current_fmr
            fnmr_fmr_diff = current_diff
            output_threshold = threshold

        else:
            # difference will start to increase, nothing to do anymore
            break

    return output_fnmr, output_fmr, output_threshold


def compute_fmr_tmr_auc(observations, metric):
    # sorted list of scores
    scores = [obs[1] for obs in observations]
    scores = sorted(scores)
    if len(scores) == 0:
        return float('NaN'), None, None  # nothing to do here

    # holds the computed FMR and FNMR values
    fmr = []
    tmr = []

    # for each score taken as threshold
    for threshold in scores:
        current_fmr = compute_fmr(observations, threshold, metric)   # FMR and FNMR are calculated based
        current_fnmr = compute_fnmr(observations, threshold, metric) # on which metric we want to use

        # cancels computation if any of the FNMR or FMR values are 'NaN' (not possible to compute them)
        if not float('-inf') < current_fmr < float('inf') or not float('-inf') < current_fnmr <
float('inf'):
            return float('NaN'), None, None  # nothing to do here

        # adds the computed values to the proper lists
        fmr.append(current_fmr)
        tmr.append(1.0 - current_fnmr)

    # computes the auc
    auc_parts = []
    for i in range(len(fmr) - 1):
        auc_parts.append(abs(fmr[i] - fmr[i + 1]) * (tmr[i] + tmr[i + 1]) / 2.0)
    auc = _pairwise_sum(auc_parts)

    return auc, fmr, tmr


def plot_all_fmr_tmr_auc(fmrs1, tmrs1, auc1, fmrs2, tmrs2, auc2, fmrs3, tmrs3, auc3):
    if float('-inf') < auc1 < float('inf') and float('-inf') < auc2 < float('inf')
       and float('-inf') < auc3 < float('inf'):
        plt.xlabel('FMR')
        plt.ylabel('TMR')
        plt.plot(fmrs1, tmrs1, 'r', label='System 1 AUC: ' + '{:.2f}'.format(auc1))
        plt.plot(fmrs2, tmrs2, 'g', label='System 2 AUC: ' + '{:.2f}'.format(auc2))
        plt.plot(fmrs3, tmrs3, 'b', label='System 3 AUC: ' + '{:.2f}'.format(auc3))
        plt.plot([0, 1], [0, 1], color='gray', linestyle='--')
        plt.legend(loc='lower right')
        plt.title('ROC curve')
        plt.show()

# Plots the histograms of the scores of the impostors and of the genuine observations together.
# Observations must be an array of (<label>,<score>) elements.
# Labels must be either 0 (impostor) or something else (genuine).
def plot_hist(observations, x_axis_label):
    impostors = []
    genuine = []

    for item in observations:
        if item[0] == 0:
            impostors.append(item[1])
        else:
            genuine.append(item[1])

    plt.xlabel(x_axis_label)
    plt.ylabel('frequency')
    plt.hist(impostors, facecolor='red', alpha=0.5, label='impostor', align='mid')
    plt.hist(genuine, facecolor='blue', alpha=0.5, label='genuine', align='mid')

    plt.legend(loc='lower right')

    dprime = compute_d_prime(observations)
    if float('-inf') < dprime < float('inf'):
        plt.title("Score distribution, d'=" + '{:.2f}'.format(dprime))
    else:
        plt.title('Score distribution')

    plt.show()
```