In [27]:
```python
import os
import glob

import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import seaborn as sns
%matplotlib inline
```

In [23]:
```python
cwd = os.getcwd()
plot_dir = os.path.join(cwd, 'plots')
sns.set_theme(style="whitegrid")
```

Technical Approach

a) Bayesian Estimation

Consider the training set D1 and strategy 1,

For each class, compute the covariance Σ of the class-conditional, and the posterior mean μ_1 , and covariance Σ_1 of

$$P_{\mu|T}(\mu|D1) = N(\mu, \mu 1, \Sigma_1)$$

$$p(x|\mu) \sim N(\mu, \Sigma)$$
$$p(\mu) \sim N(\mu_0, \Sigma_0)$$

where Σ, Σ_0 and μ_0 are assumed to be known.

We cheat a little bit by simply replacing the true covariance matrix by the sample covariance of the training set D.

$$\Sigma = \hat{\Sigma} = \frac{1}{N}\sum_{k=1}^{n}\Big(x_k - \frac{1}{N}\sum_{i=1}^{N}x_i\Big)\Big(x_k - \frac{1}{N}\sum_{i=1}^{N}x_i\Big)^T$$

For the covariance of gaussian prior Σ_0, we assume a diagonal matrix with

$$(\Sigma_0)_{ii} = \alpha\omega_i$$

$$\mu_n = \Sigma_0\big(\Sigma_0 + \frac{1}{n}\Sigma\big)^{-1}\hat{\mu_n} + \frac{1}{n}\Sigma\big(\Sigma_0 + \frac{1}{n}\Sigma\big)^{-1}\mu_0$$

Where $\hat{\mu_n}$ is the sample mean

$$\hat{\mu_n} = \frac{1}{n}\sum_{k=1}^{n}x_k$$

$$\Sigma_n^{-1} = n\Sigma^{-1} + \Sigma_0^{-1}$$

$$\implies \Sigma_n = \Sigma_0\big(\Sigma_0 + \frac{1}{n}\Sigma\big)^{-1}\frac{1}{n}\Sigma$$

Since the sum of two independent, normally distibuted vectors is again a normally distributed vector.

whose mean is the sum of the means

and whose covariance matrixis is the sum of the covariance matrices

$$p(x|D) \sim N(\mu_n, \Sigma + \Sigma_n)$$

We can compute the predictive distribution and plug into the BDR.

b) MLE

This part is same as what we do in the previous homework.

ML-BDR:

pick i if

$$i^* = \argmax_i P_{X|Y}(x|i;\theta_i^*)P_Y(i)$$

where

$$\theta_i^* = \argmax_\theta P_{X|Y}(D|i,\theta)$$

In the gaussian case, we obtain the posterior mean and covariance matrix as sample mean and sample covariance.

$$\mu = \hat{\mu} = \frac{1}{n}\sum_{k=1}^{n}x_k$$

$$\Sigma = \frac{1}{N}\sum_{k=1}^{n}\Big(x_k - \mu\Big)\Big(x_k - \mu\Big)^T$$

c) MAP estimate of μ

$$P_{X|T}(x|D1) = P_{X|\mu}(x|\mu_{MAP})$$

where

$$\begin{aligned}\mu_{MAP} &= \argmax_\mu P_{\mu|T}(\mu|D1)\\ &= \argmax_\theta P_{T|\mu}(D1|\mu)P_\Theta(\theta)\end{aligned}$$

and corresponds to approximating the prior by a delta function centered at its maximum.

In this case, we have

$$\begin{aligned}\mu_{MAP} = \Sigma_0\big(\Sigma_0 + \frac{1}{n}\Sigma\big)^{-1}\hat{\mu_n} + \frac{1}{n}\Sigma\big(\Sigma_0 + \frac{1}{n}\Sigma\big)^{-1}\mu_0\\ = \mu_n\end{aligned}$$

$$\Sigma_{MAP} = \Sigma = \hat{\Sigma}$$

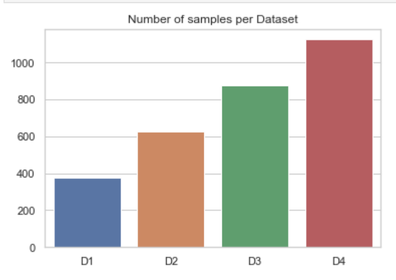
The difference between ML and Bayes MAP is non-negligible only when the dataset is small.

In [29]:
```python
data = {
    'D1': [75 ,300],
    'D2': [125,500],
    'D3': [175,700],
    'D4': [225,900],
    }
df = pd.DataFrame.from_dict(data, orient='index', columns=['FG', 'BG'])
df.head()
```

Out[29]:

	FG	BG
D1	75	300
D2	125	500
D3	175	700
D4	225	900

In [31]:
```python
ax = sns.barplot(x=["D1", "D2", "D3", "D4", ], y=[sum(data['D1']), sum(data['D2']), sum(data['D3']), sum(data['
plt.title("Number of samples per Dataset")
plt.show()
```



Discussion

Strategy 1:

μ_0 is smaller for the (darker) cheetah class (μ_0 = 1) and larger for the (lighter) grass class (μ_0 = 3).

Bayesian Estimation

Since strategy 1 is using a godd prior. we can see when α is small, the error is small as well. This can be confirm through all dataset's training result of strategy 1.

α controls the variance of gaussian prior. when the variance is small (α is small), we are very certain that assumption prior close to true prior of parameter. Increasing the size of data will have a smaller effect on the result, because we rely more on the prior belief of parameter.

On the other hand, when α is large, we are uncertain about the prior of parameter. In this case, we will rely more on the data. As the size of data increases, the results is approching to the ML result, but it does not guarante to converge to the ML result.

MLE

The MLE error is a horiziontal line since it use a determerministic parameter as opposed to Bayesian estimation method which use a stochastic parameter. The MLE is more accurate as the size of data increase, since the estimator $\hat{\mu}$ and $\hat{\Sigma}$ converge asymptotically to true mean and covariance.

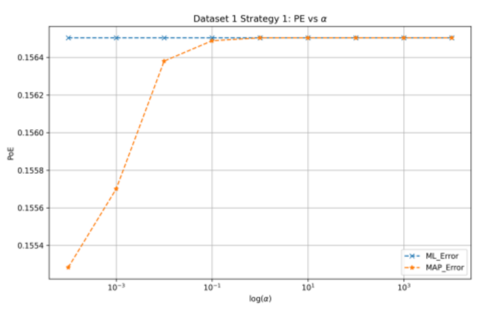
Bayes MAP Approximation

The MAP error curve starts at lower error. We can see it start very close to Bayesian estimation results since it is picking one model with largest probability instead of a weighted sum of every model. Then it increases as uncertantiy (α) increase and eventually becomes a horizontal curve and converge to ML error.

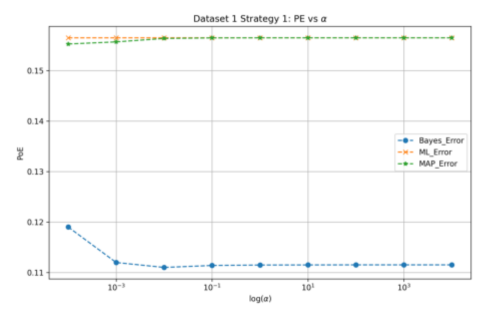
The difference between ML and Bayes MAP is non-negligible only when the dataset is small.

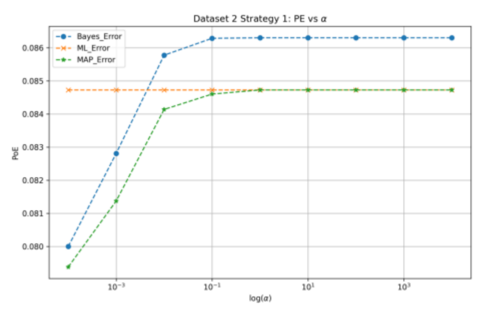
As α increase, the prior information has less effect on μ_{MAP}, the μ_{MAP} nearly equals to the μ_{ML} over the sample data.

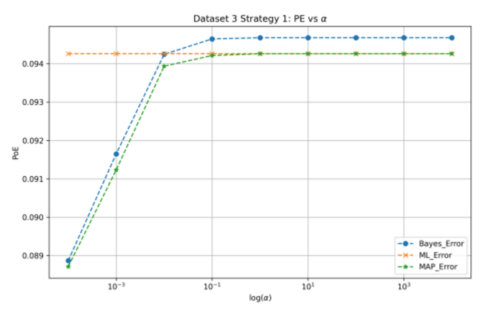
In [34]:
```python
# Strategy 1:
'''Zoom in for D1 Strategy 1 with ML and MAP only'''
plt.figure(figsize=(12,6))
img = mpimg.imread(plot_dir + "/only.png")
plt.imshow(img)
plt.axis('off')
plt.show()
```

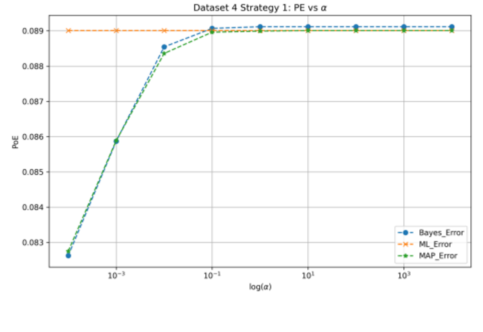


In [35]:
```python
for fig in sorted(glob.glob(plot_dir + "/*Strategy 1.png")):
    plt.figure(figsize=(12,6))
    img = mpimg.imread(fig)
    plt.imshow(img)
    plt.axis('off')
    plt.show()
```









Strategy 2:

μ_0 is equal to half the range of amplitudes of the DCT coefficient for both classes (μ_0 = 2) This time our prior is not good.

Bayesian Estimation

All implementation logic is the same as strategy 1. We can see now our prior is not as good as the one in stragegy 1. As the vairance of prior increase, the Bayesian estiamtion rely more on data which act as a heuristic to correct our biased prior assumption.

MLE

The trend of MLE error is not affect by the strategy but since the parameter is deterministic. The MLE is more accurate as the size of data increase, since the estimator $\hat{\mu}$ and $\hat{\Sigma}$ converge asymptotically to true mean and covariance.

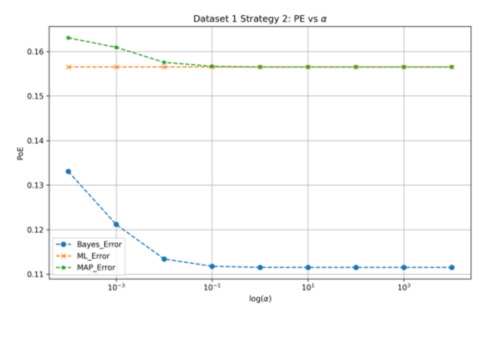
Bayes MAP Approximation

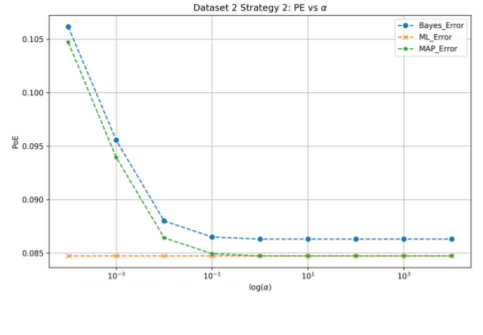
Form D1, we see a huge gap between Bayesian estimation and Bayes MAP. Since we are picking only one model with maximum posterior to approximate the Bayesian estimiation. All other information except the MAP one is lost.

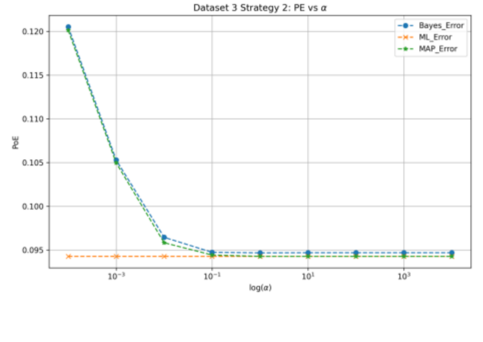
The difference between ML and Bayes MAP is non-negligible only when the dataset is small.

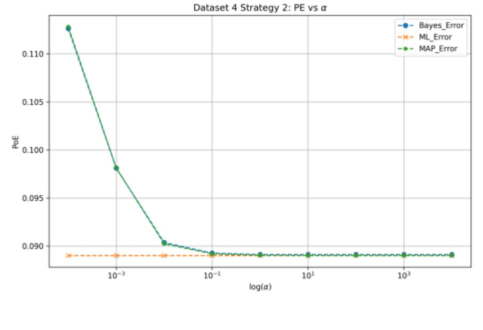
As mentioned above, Bayes MAP error will eventually converge to ML error when the dataset is large and variance of prior is also large.

In [18]:
```python
# Strategy 2:
for fig in sorted(glob.glob(plot_dir + "/*Strategy 2.png")):
    plt.figure(figsize=(12,6))
    img = mpimg.imread(fig)
    plt.imshow(img)
    plt.axis('off')
    plt.show()
```









```
import argparse
import os
import pathlib
import math

from multiprocessing import Process

import numpy as np
from numba import jit
import scipy.io as sio
import matplotlib.pyplot as plt
from PIL import Image
from tqdm import tqdm

import utils

try:
    from icecream import ic
except ImportError:  # Graceful fallback if IceCream isn't installed.
    ic = lambda *a: None if not a else (a[0] if len(a) == 1 else a)  # noqa


def mu_n(sample_mean, prior_mu0, sample_cov, prior_sigma0, n: int, a_inv:
np.ndarray):
    """
    a_inv = np.linalg.inv(Sigma_0 + Sigma / n)
    """
    # (64,64) @ (64,64) @ (64, 1) + (64, 64) @ (64, 64) @ (64, 1) -> (64,
1) + (64,1)
    return (prior_sigma0 @ a_inv @ sample_mean) + (sample_cov @ a_inv @
prior_mu0) / n


@jit(nopython=True)
def sigma_n(sigma, sigma_0, n: int, a_inv: np.ndarray):
    return sigma_0 @ a_inv * sigma / n


@jit(nopython=True)
def g(x, W, w, w0):
    """
    Decision boundary function g_i(x).
    """
    return x.T @ W @ x + w.T @ x + w0


def ML_result(BG, FG):
    current_dir = pathlib.Path(__file__).parent.resolve()
    data_dir = current_dir / "data"
    TrainsampleDCT_BG = BG
    TrainsampleDCT_FG = FG
```

```
    m_FG_ML, n_FG = TrainsampleDCT_FG.shape
    m_BG_ML, n_BG = TrainsampleDCT_BG.shape

    P_FG_ML = m_FG_ML / (m_FG_ML + m_BG_ML)
    P_BG_ML = m_BG_ML / (m_FG_ML + m_BG_ML)
    assert P_FG_ML + P_BG_ML == 1

    #
=============================================================================
============
    # ML mean mu
    mu_FG_ML = np.mean(TrainsampleDCT_FG, axis=0).reshape(-1, 1)
    mu_BG_ML = np.mean(TrainsampleDCT_BG, axis=0).reshape(-1, 1)

    # ML covariance Sigma
    cov_FG_ML, cov_BG_ML = np.cov(TrainsampleDCT_FG.T),
np.cov(TrainsampleDCT_BG.T)

    img = np.asarray(Image.open(os.path.join(data_dir, "cheetah.bmp"),
"r"))

    # Convert to double and / 255
    img = utils.im2double(img)
    assert img.min() == 0 and img.max() <= 1

    ground_truth = np.asarray(
        Image.open(os.path.join(data_dir, "cheetah_mask.bmp"), "r")
    )

    processed_img = np.zeros([img.shape[0] - 8, img.shape[1] - 8],
dtype=bool)

    # constants

    logp_FG_ML = np.log(P_FG_ML)
    logp_BG_ML = np.log(P_BG_ML)

    logdet_FG_ML = np.log(np.linalg.det(cov_FG_ML))
    logdet_BG_ML = np.log(np.linalg.det(cov_BG_ML))

    W_FG = np.linalg.inv(cov_FG_ML)
    W_BG = np.linalg.inv(cov_BG_ML)

    w_FG = -2 * W_FG @ mu_FG_ML
    w_BG = -2 * W_BG @ mu_BG_ML

    w0_FG = mu_FG_ML.T @ W_FG @ mu_FG_ML + logdet_FG_ML - 2 * logp_FG_ML
    w0_BG = mu_BG_ML.T @ W_BG @ mu_BG_ML + logdet_BG_ML - 2 * logp_BG_ML

    # Feature vector 64 x 1
    x_64 = np.zeros((64, 1), dtype=np.float64)
    for i in range(processed_img.shape[0]):
        for j in range(processed_img.shape[1]):
            # 8 x 8 block
```

```
            block = img[i : i + 8, j : j + 8]
            # DCT transform on the block
            block_DCT = utils.dct2(block)
            # zigzag pattern mapping
            for k in range(block_DCT.shape[0]):
                for p in range(block_DCT.shape[1]):
                    loc = utils.zigzag[k, p]
                    x_64[loc, :] = block_DCT[k, p]

            if g(x_64, W_FG, w_FG, w0_FG) >= g(x_64, W_BG, w_BG, w0_BG):
                processed_img[i, j] = 0
            else:
                processed_img[i, j] = 1
    errors_ML, _, _ = utils.calculate_error(processed_img, ground_truth,
verbose=False)
    return errors_ML


def run(strategy, plot: bool = True, save: bool = False, test: bool =
False):
    current_dir = pathlib.Path(__file__).parent.resolve()
    data_dir = current_dir / "data"
    prior1_fname = data_dir / "Prior_1.mat"
    prior2_fname = data_dir / "Prior_2.mat"
    alpha_fname = data_dir / "Alpha.mat"
    mat_fname_subset = data_dir / "TrainingSamplesDCT_subsets_8.mat"
    plot_dir = current_dir / "plots"

    # Create the directory if it does not exist
    for d in [data_dir, plot_dir]:
        if not os.path.exists(d):
            os.mkdir(d)

    #
===========================================================================
============
    # Load the data
    subsets8 = sio.loadmat(mat_fname_subset)
    prior_1 = sio.loadmat(prior1_fname)
    prior_2 = sio.loadmat(prior2_fname)
    alpha_dict = sio.loadmat(alpha_fname)

    # weights
    alpha = alpha_dict["alpha"].ravel()

    #
===========================================================================
============
    # Load Images
    # load Image (original_img has dtype=uint8)
    img = np.asarray(Image.open(os.path.join(data_dir, "cheetah.bmp"),
"r"))
    # Convert to double and / 255
    img = utils.im2double(img)
```

```
    assert img.min() == 0 and img.max() <= 1
    ground_truth = np.asarray(
        Image.open(os.path.join(data_dir, "cheetah_mask.bmp"), "r")
    )

    #
=============================================================================
============
    # Handle stragtegys
    if strategy == 1:
        prior = prior_1
    elif strategy == 2:
        prior = prior_2
    else:
        raise ValueError("Invalid strategy. Choice:[1, 2]")

    print(f"Strategy: {strategy}")
    err_bayes = []
    err_mle = []
    err_map = []

    #
=============================================================================
============
    pbar_idx = 1
    for subset_idx in tqdm(
        range(1, 5), dynamic_ncols=True, desc=f"Dataset ({pbar_idx})"
    ):
        pbar_idx += 1
        D_BG = subsets8[f"D{subset_idx}_BG"]
        D_FG = subsets8[f"D{subset_idx}_FG"]

        # n_samples and m_features
        n_BG, m_BG = D_BG.shape
        n_FG, m_BG = D_FG.shape

        # prior
        total_samples = n_BG + n_FG
        P_BG = n_BG / total_samples
        P_FG = n_FG / total_samples
        print(f"\tprior_BG: {P_BG}")
        print(f"\tprior_FG: {P_FG}")

        prior_mu0_BG = prior["mu0_BG"].reshape(-1, 1)
        prior_mu0_FG = prior["mu0_FG"].reshape(-1, 1)

        # \hat{\mu_n} sample mean
        BG_mean = np.mean(D_BG, axis=0).reshape(-1, 1)
        FG_mean = np.mean(D_FG, axis=0).reshape(-1, 1)

        # \Sigma sample covariance using unbiased estimator
        BG_cov = np.cov(D_BG.T, bias=False)
        FG_cov = np.cov(D_FG.T, bias=False)
```

```
        # log prior
        logp_FG = math.log(P_FG)
        logp_BG = math.log(P_BG)

        # a) Bayesian Estimation
        img_lst = []
        print(f"\tBayesian Estimation with Strategy {strategy}")
        for a in range(alpha.shape[0]):
            processed_img = np.empty([img.shape[0] - 8, img.shape[1] - 8],
dtype=bool)
            # Sigma_0 (with weight = alpha[i] )
            prior_sigma0 = np.diag((alpha[a] * prior["W0"]).flat)

            # import ipdb; ipdb.set_trace()
            # * pre-compute the inverse of Sigma_0 + (Sigma / n)
            a_BG_inv = np.linalg.inv(prior_sigma0 + BG_cov / n_BG)
            a_FG_inv = np.linalg.inv(prior_sigma0 + FG_cov / n_FG)

            # Parameter Distribution
            mu_n_BG = mu_n(BG_mean, prior_mu0_BG, BG_cov, prior_sigma0,
n_BG, a_BG_inv)
            mu_n_FG = mu_n(FG_mean, prior_mu0_FG, FG_cov, prior_sigma0,
n_FG, a_FG_inv)

            cov_n_BG = sigma_n(BG_cov, prior_sigma0, n_BG, a_BG_inv)
            cov_n_FG = sigma_n(FG_cov, prior_sigma0, n_FG, a_FG_inv)

            # Sum of two independent Gaussian is again a Gaussian
            # where mean is the sum of the means
            mu_BG = mu_n_BG
            mu_FG = mu_n_FG
            # and whose covariance matrixis is the sum of the covariance
matrices
            cov_BG = cov_n_BG + BG_cov
            cov_FG = cov_n_FG + FG_cov

            # guassian decsison bounday
            logdet_BG = math.log(np.linalg.det(cov_BG))
            logdet_FG = math.log(np.linalg.det(cov_FG))

            W_BG = np.linalg.inv(cov_BG)
            W_FG = np.linalg.inv(cov_FG)

            w_BG = -2 * W_BG @ mu_BG
            w_FG = -2 * W_FG @ mu_FG

            w0_FG = mu_FG.T @ W_FG @ mu_FG + logdet_FG - 2 * logp_FG
            w0_BG = mu_BG.T @ W_BG @ mu_BG + logdet_BG - 2 * logp_BG

            # Feature vector 64 x 1
            x_64 = np.zeros((64, 1), dtype=np.float64)
            for i in range(processed_img.shape[0]):
                for j in range(processed_img.shape[1]):
                    # # 8 x 8 block
```

```
                    block = img[i : i + 8, j : j + 8]
                    # DCT transform on the block
                    block_DCT = utils.dct2(block)
                    # zigzag pattern mapping
                    for k in range(block_DCT.shape[0]):
                        for p in range(block_DCT.shape[1]):
                            loc = utils.zigzag[k, p]
                            x_64[loc, :] = block_DCT[k, p]

                    if g(x_64, W_FG, w_FG, w0_FG) > g(x_64, W_BG, w_BG,
w0_BG):
                        processed_img[i, j] = 0
                    else:
                        processed_img[i, j] = 1
            img_lst.append(processed_img)
        error_lst_bayes = [
            utils.calculate_error(img, ground_truth, verbose=False)[0]
            for img in img_lst
        ]
        err_bayes.append(error_lst_bayes)

        print(f"\tMaximum Likelihood Estimation with Strategy {strategy}")
        error_lst_ml = ML_result(BG=D_BG, FG=D_FG)
        error_lst_ml = [error_lst_ml] * alpha.shape[0]
        err_mle.append(error_lst_ml)

        # b) Bayes MAP Approximation
        img_lst_MAP = []
        print(f"\tBayesian Estimation with MAP Approximation with Strategy
{strategy}")
        for a in range(alpha.shape[0]):
            processed_img = np.empty([img.shape[0] - 8, img.shape[1] - 8],
dtype=bool)

            # Sigma_0 (with alpha[:, i])
            prior_sigma_0 = np.diag((alpha[a] * prior["W0"]).flat)

            # * pre-compute the inverse of Sigma_0 + Sigma / n
            a_BG_inv = np.linalg.inv(prior_sigma_0 + BG_cov / n_BG)
            a_FG_inv = np.linalg.inv(prior_sigma_0 + FG_cov / n_FG)

            # Parameter Distribution
            mu_n_BG = mu_n(BG_mean, prior_mu0_BG, BG_cov, prior_sigma_0,
n_BG, a_BG_inv)
            mu_n_FG = mu_n(FG_mean, prior_mu0_FG, FG_cov, prior_sigma_0,
n_FG, a_FG_inv)

            # Sum of independent gaussian -- MAP
            mu_BG = mu_n_BG
            mu_FG = mu_n_FG

            cov_BG = BG_cov
            cov_FG = FG_cov
            #
```

```
=====================================================================================
==============
            # guassian decsison rule
            logdet_BG = np.log(np.linalg.det(cov_BG))
            logdet_FG = np.log(np.linalg.det(cov_FG))

            W_BG = np.linalg.inv(cov_BG)
            W_FG = np.linalg.inv(cov_FG)

            w_BG = -2 * W_BG @ mu_BG
            w_FG = -2 * W_FG @ mu_FG

            w0_BG = mu_BG.T @ W_BG @ mu_BG + logdet_BG - 2 * logp_BG
            w0_FG = mu_FG.T @ W_FG @ mu_FG + logdet_FG - 2 * logp_FG

            # Feature vector 64 x 1
            x_64 = np.empty((64, 1), dtype=np.float64)
            for i in range(processed_img.shape[0]):
                for j in range(processed_img.shape[1]):
                    # # 8 x 8 block
                    block = img[i : i + 8, j : j + 8]
                    # DCT transform on the block
                    block_DCT = utils.dct2(block)
                    # zigzag pattern mapping
                    for k in range(block_DCT.shape[0]):
                        for p in range(block_DCT.shape[1]):
                            loc = utils.zigzag[k, p]
                            x_64[loc, :] = block_DCT[k, p]

                    if g(x_64, W_FG, w_FG, w0_FG) >= g(x_64, W_BG, w_BG,
w0_BG):
                        processed_img[i, j] = 0
                    else:
                        processed_img[i, j] = 1
            img_lst_MAP.append(processed_img)
        error_lst_MAP = [
            utils.calculate_error(img, ground_truth, verbose=False)[0]
            for img in img_lst_MAP
        ]
        err_map.append(error_lst_MAP)

        if test:
            break

    #
=====================================================================================
==============
    # plot result
    assert len(err_bayes) == len(err_mle) == len(err_map)
    for idx in range(len(err_bayes)):
        plt.figure(figsize=(10, 6), dpi=300)
        plt.plot((alpha.flat), err_bayes[idx], "--o", label="Bayes_Error")
        plt.plot((alpha.flat), err_mle[idx], "--x", label="ML_Error")
        plt.plot((alpha.flat), err_map[idx], "--*", label="MAP_Error")
```

```
        plt.xlabel(r"$\log (\alpha)$")
        plt.ylabel("PoE")
        plt.xscale("log")
        plt.grid()
        plt.title(f"Dataset {idx+1} Strategy {strategy}: " + r"PE vs
$\alpha$")
        plt.legend()
        if save:
            plt.savefig(plot_dir / f"Dataset {idx+1} Strategy
{strategy}.png")
    if plot:
        plt.show()


def main():
    parser = argparse.ArgumentParser(description="HW2")
    parser.add_argument("--plot", "-p", action="store_true", help="Plot the
data")
    parser.add_argument("--save", "-s", action="store_true", help="Save
plots")
    parser.add_argument("--test", "-t", action="store_true", help="Test the
code")
    parser.add_argument("--strategy", type=int)
    args = parser.parse_args()

    if args.strategy == 1 or args.strategy == 2:
        run(args.strategy, args.plot, args.save, args.test)
    else:
        strategies = [1, 2]
        procs = []

        # instantiating process with arguments
        for s in strategies:
            # print(name)
            proc = Process(target=run, args=(s, args.plot, args.save,
args.test))
            procs.append(proc)
            proc.start()

        # complete the processes
        for proc in procs:
            proc.join()

    print("Done!")


if __name__ == "__main__":

    main()
```