# Kode

## 1 utils.py

```python
import numpy as np
import os
import cv2 as cv
import copy


def read_dataset(idx):
    project_dir = os.path.dirname(os.path.dirname((__file__)))
    file_path = project_dir + f"/data/ds-{idx}.txt"
    data_array = np.loadtxt(file_path)
    targets, obs = data_array[:, 0].copy(), data_array[:, 1:].copy()
    return targets, obs


def split_data(obs, targets):
    train_obs, train_targets = obs[1::2], targets[1::2]
    test_obs, test_targets = obs[0::2], targets[0::2]
    return train_obs, test_obs, train_targets, test_targets


def least_params(train_obs, train_targets):
    bias = np.ones((len(train_obs), 1))
    ext_train_obs = np.concatenate((bias, train_obs), axis=1)

    b = np.where(train_targets == 1, 1, -1)

    params = np.linalg.inv(ext_train_obs.T @ ext_train_obs) @ ext_train_obs.T @ b
    return params


def least_discriminant(params):
    def discriminant(test_obs):
        bias = np.ones((len(test_obs), 1))
        ext_test_obs = np.concatenate((bias, test_obs), axis=1)
        return np.where(ext_test_obs @ params > 0, 1, 2)

    return discriminant


def create_dataset(pixels):
```

```python
    dataset = []
    for i in range(len(pixels)):
        pixels[i] = pixels[i].reshape(-1, 3)
        pixels[i] = np.concatenate(
            (np.ones((pixels[i].shape[0], 1)) * (i + 1), pixels[i]), axis=1
        )
        dataset.extend(pixels[i])
    return np.array(dataset)


def normalize_dataset(pixels):
    r = pixels[:, 1]
    g = pixels[:, 2]
    b = pixels[:, 3]

    t1 = r / np.sum(pixels[:, 1:], axis=1)
    t2 = g / np.sum(pixels[:, 1:], axis=1)
    t3 = b / np.sum(pixels[:, 1:], axis=1)

    pixels[:, 1] = t1
    pixels[:, 2] = t2
    pixels[:, 3] = t3

    return pixels


def estimate_pixels_apriori(pixels):
    probs = []
    for i in range(np.int64(np.max(pixels[:, 0], axis=0))):
        prob = np.sum(pixels[:, 0] == (i + 1)) / pixels.shape[0]
        probs.append(prob)
    return np.array(probs)


def estimate_pixels_mean(pixels):
    means = []
    for i in range(np.int64(np.max(pixels[:, 0], axis=0))):
        est_mean = pixels[pixels[:, 0] == (i + 1)].mean(axis=0)
        means.append(est_mean)
    return np.array(means)


def estimate_pixels_cov(pixels, pixel_class_means):
    covs = []
    for i in range(np.int64(np.max(pixels[:, 0], axis=0))):
        N_class = np.sum(pixels[:, 0] == (i + 1))
        class_dev = pixels[pixels[:, 0] == (i + 1), 1:] - pixel_class_means[i, 1:]
        class_cov = (class_dev.T @ class_dev) / (N_class - 1)
        covs.append(class_cov)

    return np.array(covs)


def pixels_discriminants(pixel_means, pixel_covs, pixel_apriori):
    discriminants = []
```

```
    for i in range(len(pixel_means)):
        discriminants.append(
            class_discriminant(pixel_means[i], pixel_covs[i], pixel_apriori[i], pixels=True)
        )
    return discriminants


def segment_image(image_path, discriminants):
    img = cv.imread(image_path)
    if img.shape[:2] > (800,800):
        img = cv.resize(img, (600,600))
    seg_img = np.zeros_like(img)
    colors = np.array(
        [
            [255, 0, 0],
            [0, 255, 0],
            [0, 0, 255],
            [255, 255, 0],
            [255, 0, 255],
            [0, 255, 255],
            [128, 0, 128],
            [255, 165, 0],
            [0, 128, 0],
            [128, 128, 128],
        ]
    )
    for x in range(img.shape[0]):
        for y in range(img.shape[1]):
            cl = np.argmax([disc(img[x, y]) for disc in discriminants])
            seg_img[x,y] = colors[cl]
    return seg_img


def measure_dist(obs_1, obs_2):
    distance = np.linalg.norm(obs_1 - obs_2)
    return distance


def nearest_neighbour(train_obs, train_targets, test_obs):
    c_test_obs = np.zeros((len(test_obs), 1))

    for i in range(len(test_obs)):
        near_neigh = np.argmin(
            [
                measure_dist(test_obs[i], train_obs[j])
                for j in range(len(train_obs))
                if i != j
            ]
        )
        c_test_obs[i] = train_targets[near_neigh]

    return c_test_obs.flatten()


def estimate_a_priori(train_targets):
```

```python
    class_one = np.sum(train_targets == 1)
    prob_one = class_one / train_targets.shape[0]
    prob_two = 1.0 - prob_one
    return prob_one, prob_two


def estimate_class_mean(train_obs, train_targets):
    class_one_mean = train_obs[train_targets == 1].mean(axis=0)
    class_two_mean = train_obs[train_targets == 2].mean(axis=0)
    return class_one_mean, class_two_mean


def estimate_class_cov(class_one_mean, class_two_mean, train_obs, train_targets):
    N_one = train_obs[train_targets == 1].shape[0]
    N_two = train_obs.shape[0] - N_one
    class_one_dev = train_obs[train_targets == 1] - class_one_mean
    class_two_dev = train_obs[train_targets == 2] - class_two_mean
    cov_one = (class_one_dev.T @ class_one_dev) / (N_one - 1)
    cov_two = (class_two_dev.T @ class_two_dev) / (N_two - 1)

    return cov_one, cov_two


def class_discriminant(class_mean, class_cov, a_priori_prob, pixels=False):
    W = -(1 / 2) * np.linalg.inv(class_cov)

    w = np.linalg.inv(class_cov) @ class_mean

    det_cov = np.log(np.linalg.det(class_cov))
    det_cov = det_cov if det_cov > 1e-5 else 0

    w_0 = (
        -(1 / 2) * class_mean @ np.linalg.inv(class_cov) @ class_mean
        - (1 / 2) * det_cov
        + np.log(a_priori_prob)
    )
    if pixels:
        return lambda test_obs: test_obs.T @ W @ test_obs + test_obs @ w + w_0
    else:
        return lambda test_obs: np.sum(test_obs @ W * test_obs, axis=1) + test_obs @ w + w_0


def minimum_error(train_obs, train_targets):
    class_one_mean, class_two_mean = estimate_class_mean(train_obs, train_targets)
    cov_one, cov_two = estimate_class_cov(
        class_one_mean, class_two_mean, train_obs, train_targets
    )

    a_priori_one, a_priori_two = estimate_a_priori(train_targets)

    discriminant_one = _class_discriminant(class_one_mean, cov_one,
```

```
     a_priori_one)
    discriminant_two = _class_discriminant(class_two_mean, cov_two, a_priori_two)

    return gen_discriminant(discriminant_one, discriminant_two)


def gen_discriminant(c1_discr, c2_discr):
    return lambda test_obs: np.where(c1_discr(test_obs) - c2_discr(test_obs) > 0, 1, 2)
```

## 2 oblig1.py

```
from snutils import *
import numpy as np
import matplotlib.pyplot as plt

dim_combinations_list = [
    [(0,), (1,), (2,), (3,)],
    [(0, 1), (0, 2), (0, 3), (1, 2), (2, 3)],
    [(0, 1, 2), (0, 2, 3), (1, 2, 3)],
    [(0, 1, 2, 3)],
]

dim_combinations_for_dataset_2_list = [
    [(0,), (1,), (2,)],
    [(0, 1), (0, 2), (1, 2)],
    [(0, 1, 2)],
]

for dataset_idx in (1, 2, 3):
    print(f"======== Dataset {dataset_idx} =========")
    targets, obs = read_dataset(dataset_idx)
    train_obs, test_obs, train_targets, test_targets = split_data(obs, targets)

    for dim_combinations in (
        dim_combinations_list
        if dataset_idx != 2
        else dim_combinations_for_dataset_2_list
    ):
        print(
            f"========= Now testing for dimension {len(dim_combinations[0])} ============"
        )
        best_fail_rate = 1
        best_dim = None
        for dimensions in dim_combinations:
            preds = nearest_neighbour(
                train_obs[:, dimensions], train_targets, train_obs[:, dimensions]
            )
            fail_rate = (
                np.sum(np.where(preds != train_targets, 1, 0)) / train_targets.shape[0]
            )

```

```python
        # print(f"{dimensions=} {fail_rate=}")

        if fail_rate < best_fail_rate:
            best_dim = dimensions
            best_fail_rate = fail_rate
    if dataset_idx == 2 and len(best_dim) == 2:
        plt.scatter(
            test_obs[:, best_dim[0]][test_targets == 1],
            test_obs[:, best_dim[1]][test_targets == 1],
        )
        plt.scatter(
            test_obs[:, best_dim[0]][test_targets == 2],
            test_obs[:, best_dim[1]][test_targets == 2],
        )
        plt.show()

    print(f"Lowest fail rate was {best_fail_rate:.3f}, for features: {best_dim}")

    # ------------ here starts the actual grog way ------------

    print("\n\nNow testing all methods on test set:")
    # Nearest neighbour
    preds_test_nn = nearest_neighbour(
        train_obs[:, best_dim], train_targets, test_obs[:, best_dim]
    )
    fail_rate_test_nn = (
        np.sum(np.where(preds != test_targets, 1, 0)) / test_targets.shape[0]
    )

    # Linear discriminant
    linear_discriminant = least_discriminant(
        least_params(train_obs[:, best_dim], train_targets)
    )

    preds = linear_discriminant(test_obs[:, dimensions])
    fail_rate_test_lindisc = (
        np.sum(np.where(preds != test_targets, 1, 0)) / test_targets.shape[0]
    )
    # Minimum error
    minimum_error_discriminant = minimum_error(
        train_obs[:, best_dim], train_targets
    )

    preds = minimum_error_discriminant(test_obs[:, best_dim])
    fail_rate_test_minerror = (
        np.sum(np.where(preds != test_targets, 1, 0)) / test_targets.shape[0]
    )
    print(
        f"Fail rates: NN: {fail_rate_test_nn:.3f} LINDISC: {fail_rate_test_lindisc:.3f} MINERROR: {fail_rate_test_minerror:.3f}"
    )
```