

**МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ
КАФЕДРА КОМП'ЮТЕРНИХ НАУК**

**Обов'язкове домашнє завдання
з дисципліни
«Основи проектування інтелектуальних систем»**

Виконав

Тарасов О. О.

Перевірив(ла)

Барченко Н. Л.

Довбиш А. С.

СУМИ 2020

Зміст

| | |
|-------------------------|----|
| Опис..... | 3 |
| Роботи..... | 3 |
| Практична робота 1..... | 3 |
| Практична робота 2..... | 6 |
| Практична робота 3..... | 11 |
| Практична робота 4..... | 17 |
| Практична робота 5..... | 20 |
| Практична робота 6..... | 22 |
| Практична робота 7..... | 24 |
| Практична робота 8..... | 24 |
| Практична робота 9..... | 27 |
| Висновки..... | 33 |
| Додаток..... | 34 |

Опис

У даному ОДЗ зібрані всі 9 лабораторних робіт із дисципліни «Основи проектування інтелектуальних систем».

Роботи

Практична робота 1

Завдання

Знайти в інтернеті кадри текстури різних матеріалів розмірами 50 на 50. Написати програму, що читує зображення та зберігає в масив значення кольорів пікселя. В результаті роботи потрібно вивести дані значення на екран.

В процесі виконання

В результаті праці над даним завданням з'явилися такі уточнення :

- Зображення завантажуються в теку /images
- Використовується 4 класи зображень :
 - дерево
 - цегла
 - плитка
 - тканина
- Ім'я зображення використовує паттерн КласНомер.jpg . Для прикладу **wood1.jpg**
- Не шукатимемо зображення розміром **50 * 50**, а в процесі обробки вирізатимемо їх із центру наявного зображення
- Братимемо всі три кольорові компоненти
- Матрицю зображення розміром **50 * 50 * 3** послідовно переформуємо у векторний вигляд розміром **1 * 7500** .
- В результаті роботи програми отримаємо таблицю із 7500 колонками значень пікселів і однією колонкою класу.

Код програми

Код програми написано мовою програмування Python3. Окрім стандартних модулів(itertools, fnmatch, os) необхідно встановити такі бібліотеки :

- numpy(numpy)
- pandas(pandas)
- scikit-image(skimage)

```
from itertools import product
from skimage import io
import pandas as pd
import numpy as np
import fnmatch
import os

# high level function for building dataset
def read_dataset(source_dir, classes, file_pattern, standard_shape):

    source_class_gen = make_images_sources(source_dir, classes, file_pattern)
    dataset = build_dataset(source_class_gen, standard_shape)
    return dataset

# read the source directory content and return all pathes of classes images_for_labs
def make_images_sources(source_dir, classes, file_pattern):
    sources = list()
    for class_name in classes:
        full_paths_files_source_dir = (os.path.abspath("{} / {}".format(source_dir, file_name)))
        for file_name in os.listdir(source_dir)
            class_sources = fnmatch.filter(full_paths_files_source_dir, file_pattern.format(class_name))
            sources.extend(class_sources)
    yield from product(class_sources, [class_name])

# read each image, cut it for standard size, reshape for feature vector form (1 * features)
# and insert into the DataFrame also with the class label
def build_dataset(source_class_gen, std_shape):
    columns = make_columns(std_shape) + ["class"]
    rows = list()

    for image_path, class_name in source_class_gen:
        feature_vector_length = np.prod(std_shape)
        image_matrix = io.imread(image_path)
        cut_img = cut_image(image_matrix, std_shape)
        feature_vector = cut_img.reshape(feature_vector_length)
        rows.append((feature_vector, class_name))
    return pd.DataFrame(data=rows, columns=columns), columns[:-1]

def make_columns(std_shape):
    return ["{}:{}:{}{}".format(*triple) for triple in product(range(1, std_shape[0] + 1),
                                                               range(1, std_shape[1] + 1),
                                                               range(1, std_shape[2] + 1))]
```

```

# cut standard image size from the center
def cut_image(image_matrix, std_shape):
    real_shape = image_matrix.shape

    height_border = calc_border(std_shape, real_shape, 0)
    width_border = calc_border(std_shape, real_shape, 1)

    height_size = calc_size(std_shape, real_shape, 0)
    width_size = calc_size(std_shape, real_shape, 1)

    result_matrix = np.zeros(std_shape)
    result_matrix[:height_size, :width_size] = image_matrix[height_border:height_border+height_size,
                                                          width_border:width_size+width_border]

    return result_matrix

# calc the border value
def calc_border(std_vals, real_vals, axis_index):
    return only_positive_int_numbers((real_vals[axis_index] - std_vals[axis_index]) / 2)

# return positive number or 0
def only_positive_int_numbers(val):
    return int(val) if val > 0 else 0

# if the real image size param(width or height through the axis_index) is bigger than
# standard return standard value if not return real value of the image
def calc_size(std_vals, real_vals, axis_index):
    real_param = real_vals[axis_index]
    std_param = std_vals[axis_index]
    return std_param if real_param - std_param > 0 else real_param

source_dir = "./images"
classes = ["wood", "cloth", "tile", "brick"]
file_pattern = "*{}*.jpg"

# get the dataset from files in pandas data frame format
dataset, features = read_dataset(source_dir, classes, file_pattern)

```

Результати роботи

datataset — зчитані дані представлені у форматі pandas DataFrame(майже як таблиця excel). Даний формат був обраний через легкість його використання і наявність вже запрограмованих методів знаходження статистичних оцінок :

- максимум
- мінімум
- середнє значення
- мода

| | 1:1:1 | 1:1:2 | 1:1:3 | 1:2:1 | ... | 50:50:1 | 50:50:2 | 50:50:3 | class |
|----|-------|-------|-------|-------|-----|---------|---------|---------|-------|
| 0 | 85.0 | 68.0 | 52.0 | 80.0 | ... | 125.0 | 100.0 | 80.0 | wood |
| 1 | 80.0 | 35.0 | 14.0 | 81.0 | ... | 157.0 | 95.0 | 48.0 | wood |
| 2 | 88.0 | 77.0 | 59.0 | 106.0 | ... | 90.0 | 82.0 | 63.0 | wood |
| 3 | 133.0 | 104.0 | 74.0 | 148.0 | ... | 173.0 | 142.0 | 111.0 | wood |
| 4 | 145.0 | 113.0 | 114.0 | 126.0 | ... | 114.0 | 94.0 | 96.0 | wood |
| 5 | 130.0 | 80.0 | 45.0 | 123.0 | ... | 114.0 | 68.0 | 35.0 | wood |
| 6 | 70.0 | 61.0 | 62.0 | 69.0 | ... | 90.0 | 86.0 | 85.0 | cloth |
| 7 | 145.0 | 153.0 | 166.0 | 154.0 | ... | 62.0 | 64.0 | 85.0 | cloth |
| 8 | 118.0 | 119.0 | 121.0 | 157.0 | ... | 93.0 | 94.0 | 98.0 | cloth |
| 9 | 78.0 | 76.0 | 77.0 | 68.0 | ... | 61.0 | 59.0 | 62.0 | cloth |
| 10 | 66.0 | 55.0 | 59.0 | 40.0 | ... | 159.0 | 149.0 | 137.0 | cloth |
| 11 | 166.0 | 168.0 | 165.0 | 123.0 | ... | 99.0 | 104.0 | 97.0 | cloth |
| 12 | 122.0 | 167.0 | 208.0 | 121.0 | ... | 115.0 | 159.0 | 196.0 | tile |
| 13 | 193.0 | 190.0 | 183.0 | 196.0 | ... | 186.0 | 182.0 | 171.0 | tile |
| 14 | 223.0 | 207.0 | 182.0 | 221.0 | ... | 215.0 | 194.0 | 163.0 | tile |
| 15 | 135.0 | 115.0 | 90.0 | 162.0 | ... | 95.0 | 83.0 | 67.0 | tile |
| 16 | 31.0 | 48.0 | 94.0 | 51.0 | ... | 16.0 | 29.0 | 61.0 | tile |
| 17 | 184.0 | 163.0 | 134.0 | 189.0 | ... | 197.0 | 170.0 | 143.0 | tile |
| 18 | 146.0 | 99.0 | 91.0 | 164.0 | ... | 180.0 | 119.0 | 98.0 | brick |
| 19 | 218.0 | 107.0 | 61.0 | 208.0 | ... | 120.0 | 71.0 | 31.0 | brick |
| 20 | 130.0 | 104.0 | 103.0 | 122.0 | ... | 150.0 | 46.0 | 35.0 | brick |
| 21 | 148.0 | 72.0 | 56.0 | 154.0 | ... | 119.0 | 53.0 | 39.0 | brick |
| 22 | 181.0 | 89.0 | 64.0 | 178.0 | ... | 166.0 | 84.0 | 63.0 | brick |
| 23 | 162.0 | 95.0 | 69.0 | 160.0 | ... | 94.0 | 59.0 | 57.0 | brick |

features — масив колонок із даними.

```
[ '1:1:1', '1:1:2', '1:1:3', '1:2:1', '1:2:2', '1:2:3', '1:3:1', '1:3:2', '1:3:3', '1:4:1', '1:4:2', '1:4:3', '1:5:1',
  '1:5:2', '1:5:3', '1:6:1', '1:6:2', '1:6:3', '1:7:1', '1:7:2', '1:7:3', '1:8:1', '1:8:2', '1:8:3', '1:9:1', '1:9:2',
  '1:9:3', '1:10:1', '1:10:2', '1:10:3', '1:11:1', '1:11:2', '1:11:3', '1:12:1', '1:12:2', '1:12:3', '1:13:1', 2
  '1:13:2', '1:13:3', '1:14:1', '1:14:2', '1:14:3', '1:15:1', '1:15:2', '1:15:3', '1:16:1', '1:16:2', '1:16:3', 2
  '1:17:1', '1:17:2', '1:17:3', '1:18:1', '1:18:2', '1:18:3', '1:19:1', '1:19:2', '1:19:3', '1:20:1', '1:20:2', 2
  '1:20:3', '1:21:1', '1:21:2', '1:21:3', '1:22:1', '1:22:2', '1:22:3', '1:23:1', '1:23:2', '1:23:3', '1:24:1', 2
  '1:24:2', '1:24:3', '1:25:1', '1:25:2', '1:25:3', '1:26:1', '1:26:2', '1:26:3', '1:27:1', '1:27:2', '1:27:3', 2
  '1:28:1', '1:28:2', '1:28:3', '1:29:1', '1:29:2', '1:29:3', '1:30:1', '1:30:2', '1:30:3', '1:31:1', '1:31:2', 2
  '1:31:3', '1:32:1', '1:32:2', '1:32:3', '1:33:1', '1:33:2', '1:33:3', '1:34:1'... ]
```

Практична робота 2

Завдання

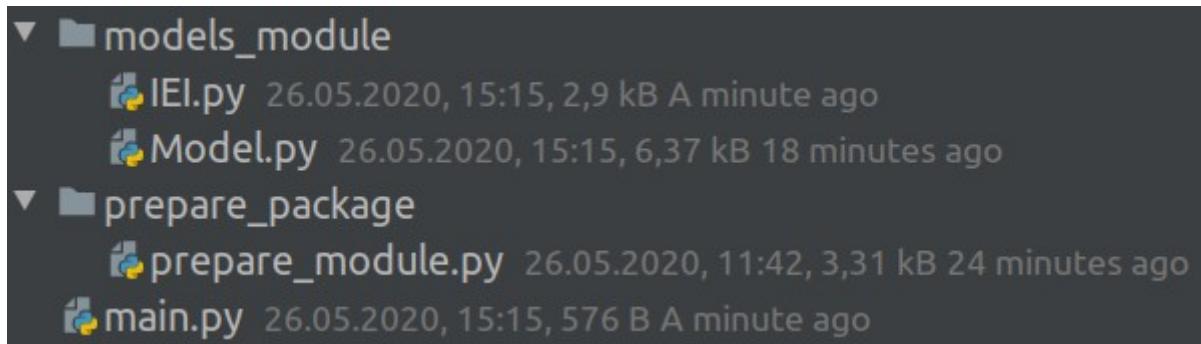
За отриманою в 1 практиці інформацією визначити базовий клас і обчислити :

- бінарні матриці класів
- центри гіперсферичних контейнерів класів

при $\delta = 20$. Вивести дані на екран.

Код програми

В результаті всієї роботи ОДЗ необхідно буде створити готову до роботи модель класифікації. Для цього застосуємо програмно-проектувальні здібності і створимо таку систему каталогів :



- **prepare_package** — каталог для напрацювань із 1 лабораторної роботи. Містить лише файл prepare_module
- **models_module** — каталог для основних класів моделі :
 - ➔ **IEI.IEIModelAPI** — клас-обгортка, що створює API моделі для користувачів
 - ➔ **Model.Model** — основний клас для навчання.
- **main** — файл для запуску всієї програми, тобто використання всієї написаних модулів відбудуватиметься саме тут.

В процесі подальшої роботи над ОДЗ ми будемо вдосконалювати саме ці класи.

main.py

```
from prepare_package.prepare_module import read_dataset
from models_module.IEI import IEIModelAPI

source_dir = "./images"
classes = ["wood", "cloth", "tile", "brick"]
file_pattern = "*{}*.jpg"

# get the dataset from files in pandas data frame format
dataset, features = read_dataset(source_dir, classes, file_pattern)

# split it into train and exam
train = dataset.sample(frac=0.8)
test = dataset.drop(train.index)

# make the model for predictions
model = IEIModelAPI(train, features, "class")
```

Код у даному файлі отримує датасет за назвами класів та паттерном класів із директорії *images*. За допомогою бібліотеки *pandas* даний датасет буде поділено на дві частини : тренувальні та екзаменаційні дані у пропорції 4:1. Далі відбувається створення API для моделі.

IEI.py

```
import numpy as np
from models_module.Model import Model

class IEIModelAPI:
    __tolerance_dist_start = 20

    # main constructor
    def __init__(self, train_data, features, target):
        # save all key properties
        self.__features = features
        self.__target = target
        self.__classes = {*np.unique(train_data[self.__target].values)}

        # start training
        self.__train(train_data)

    def __train(self, train_data):
        # define the most common class
        base_class = train_data[self.__target].mode().values[0]

        # calc mean feature values for base class
        mean_base_class_val = self.__mean_base_class_values(train_data, base_class)

        # make the deltas array for the features
        features_len = len(mean_base_class_val)
        features_deltas = np.array([self.__tolerance_dist_start] * features_len)
        model = Model(train_data, self.__features, self.__target, mean_base_class_val, features_deltas)

    def __mean_base_class_values(self, dataset, most_freq_class):
        # get base class feature values from dataset
        values_matrix = dataset.loc[dataset[self.__target] == most_freq_class, self.__features].values

        # get number of rows
        measure_number = values_matrix.shape[0]
        mean_feature_vals = np.sum(values_matrix, axis=0) / measure_number
        return mean_feature_vals
```

В даному класі зберігаються всі необхідні властивості даних(ознаки розпізнавання, колонка лейблу класу). Також знаходиться базовий клас, його середні значення ознак. Окремо формується вектор значень дельт для обчислення системи контрольних допусків. Всі отримані дані передаються в конструктор класу моделі, де і буде вестися обчислення бінарних матриць та центрів класів.

Model.py

В даному файлі відбувається нраві вся робота :

- визначення контрольних інтервалів
- виведення бінарних матриць і центрів контейнерів

```
import numpy as np
import pandas as pd
from math import log2

class Model:
    __sel_level = 0.5

    def __init__(self, train_data, features, target, mean_base_class_vals, tol_interval_delta):
        # save key properties
        self.__features = features
        self.__target = target
        self.__classes = {*np.unique(train_data[self.__target].values)}

        # calc tol interval
        self.tol_dist_field_bottom_vals = mean_base_class_vals - tol_interval_delta
        self.tol_dist_field_top_vals = mean_base_class_vals + tol_interval_delta
        self.tol_dists = tol_interval_delta

        # start building the model
        self.__learn(train_data)

    def __learn(self, train_data):
        binarized_dataset = self.__build_bin_feature_matrix(train_data)
        self.__centers = self.__build_standard_bin_classes_vectors(binarized_dataset)
        self.__practice_report(binarized_dataset)

    def __build_bin_feature_matrix(self, dataset):
        val_matrix = dataset[self.__features].values

        # make new bin matrix
        bin_val_matrix = np.zeros(shape=val_matrix.shape)

        # assert which numbers are in tolerance interval
        positions = np.where((val_matrix > self.tol_dist_field_bottom_vals) &
                             (val_matrix < self.tol_dist_field_top_vals))

        # set 1 to numbers that are in tolerance interval
        bin_val_matrix[positions] = 1

        # create new DataFrame but with binary data
        bin_dataset = dataset.copy()
        bin_dataset[self.__features] = bin_val_matrix
        return bin_dataset
```

```

def __build_standard_bin_classes_vectors(self, binarized_dataset):
    centers = {}
    for class_name in self.__classes:

        # get bin data for each class
        class_matrix = binarized_dataset.loc[binarized_dataset[self.__target] == class_name, self.__features].values

        # calc the container center
        cont_center = self.__calc_container_center(class_matrix)

        # add to all centers
        centers[class_name] = cont_center

    # make the DataFrame of centers
    return pd.DataFrame.from_dict(centers, orient="index", columns=self.__features)

def __calc_container_center(self, matrix):
    # calc mean bin value for each feature
    mean_bin_vals = np.sum(matrix, axis=0) / matrix.shape[0]

    # create new center array
    cur_cont_center = np.zeros(matrix.shape[1])

    # find where mean values are bigger than selection level
    units_positions = np.where(mean_bin_vals > self.__sel_level)

    # set 1 to that positions
    cur_cont_center[units_positions] = 1
    return cur_cont_center

def __practice_report(self, bin_dataset):
    print("bin_data:")
    print(bin_dataset)
    print("centers:")
    print(self.__centers)

```

Результатами роботи

| bin_data: | | | | | | | | | | |
|-----------|-------|-------|-------|-------|-----|---------|---------|---------|-------|--|
| | 1:1:1 | 1:1:2 | 1:1:3 | 1:2:1 | ... | 50:50:1 | 50:50:2 | 50:50:3 | class | |
| 12 | 0.0 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 0.0 | tile | |
| 20 | 0.0 | 1.0 | 0.0 | 0.0 | ... | 1.0 | 0.0 | 1.0 | brick | |
| 16 | 0.0 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 1.0 | tile | |
| 17 | 1.0 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 0.0 | tile | |
| 0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | 1.0 | 0.0 | 0.0 | wood | |
| 7 | 1.0 | 0.0 | 0.0 | 1.0 | ... | 0.0 | 1.0 | 0.0 | cloth | |
| 14 | 0.0 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 0.0 | tile | |
| 10 | 0.0 | 0.0 | 1.0 | 0.0 | ... | 0.0 | 0.0 | 0.0 | cloth | |
| 23 | 1.0 | 1.0 | 1.0 | 1.0 | ... | 0.0 | 1.0 | 1.0 | brick | |
| 15 | 0.0 | 0.0 | 1.0 | 1.0 | ... | 0.0 | 1.0 | 1.0 | tile | |
| 18 | 1.0 | 1.0 | 1.0 | 1.0 | ... | 0.0 | 0.0 | 0.0 | brick | |
| 4 | 1.0 | 1.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 0.0 | wood | |
| 22 | 1.0 | 1.0 | 1.0 | 1.0 | ... | 0.0 | 1.0 | 1.0 | brick | |
| 2 | 0.0 | 1.0 | 1.0 | 0.0 | ... | 0.0 | 1.0 | 1.0 | wood | |
| 21 | 1.0 | 0.0 | 1.0 | 1.0 | ... | 1.0 | 1.0 | 1.0 | brick | |
| 8 | 0.0 | 0.0 | 0.0 | 1.0 | ... | 0.0 | 0.0 | 0.0 | cloth | |
| 13 | 0.0 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 0.0 | tile | |
| 3 | 0.0 | 1.0 | 1.0 | 1.0 | ... | 0.0 | 0.0 | 0.0 | wood | |
| 19 | 0.0 | 1.0 | 1.0 | 0.0 | ... | 1.0 | 1.0 | 0.0 | brick | |

| centers: | | | | | | | | | |
|----------|-------|-------|-------|-------|-----|---------|---------|---------|---------|
| | 1:1:1 | 1:1:2 | 1:1:3 | 1:2:1 | ... | 50:49:3 | 50:50:1 | 50:50:2 | 50:50:3 |
| tile | 0.0 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 0.0 | 0.0 |
| brick | 1.0 | 1.0 | 1.0 | 1.0 | ... | 0.0 | 0.0 | 1.0 | 1.0 |
| wood | 0.0 | 1.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 0.0 | 0.0 |
| cloth | 0.0 | 0.0 | 0.0 | 1.0 | ... | 0.0 | 0.0 | 0.0 | 0.0 |

Практична робота 3

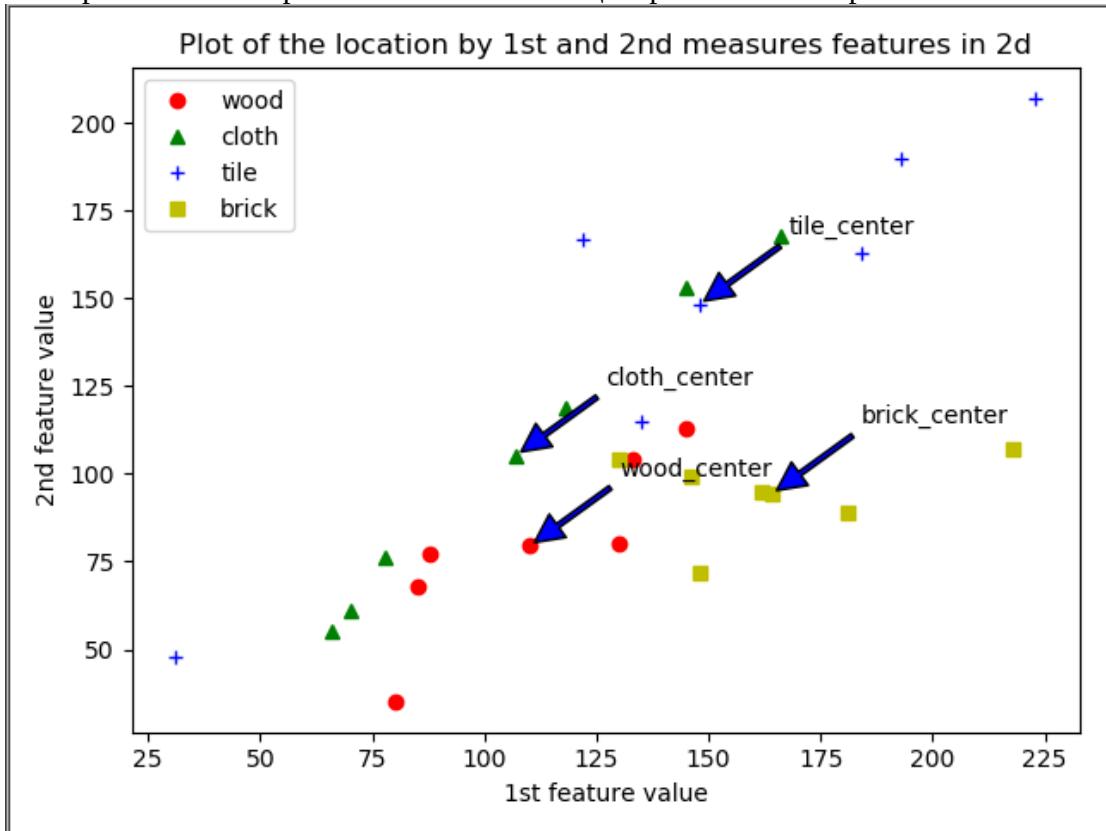
Завдання

Дане таке завдання :

- Створити функція обрахунку відстаней Хеммінга
- Порахувати відстані :
 - ➔ Центр — своя реалізація
 - ➔ Центр — Центр
 - ➔ Центр — чужі реалізації
- Створити графік із даних із даних із роботи 1 при цьому кожен клас позначити власним кольором і спроектувати центри на графіку.

Код програми

Спочатку побудуємо такий графік розташування вимірювань у 2д за першими двома ознаками розпізнавання разом із обчисленими центрами контейнерів:



Для цього ми використаємо наступний код:

```
from prepare_package.prepare_module import read_dataset
from models_module.IEI import IEIModelAPI
from matplotlib import pyplot as plt

def plot_class(data, classname, style):
    class_data = data[data["class"] == classname]
    x = class_data.iloc[:, 0]
    y = class_data.iloc[:, 1]
    x_mean = x.mean()
    y_mean = y.mean()
    plt.annotate("{}_center".format(classname), xy=(x_mean, y_mean),
                 xytext=(x_mean+20, y_mean+20), textcoords="data",
                 arrowprops=dict(facecolor="b", shrink=0.05, width=2))
    plt.plot([*x, x_mean], [*y, y_mean], style, label=classname)

source_dir = "./images"
classes = ["wood", "cloth", "tile", "brick"]
file_pattern = "*{}*.jpg"
styles = ["ro", "g^", "b+", "ys"]

# get the dataset from files in pandas data frame format
dataset, features = read_dataset(source_dir, classes, file_pattern)

for classname, style in zip(classes, styles):
    plot_class(dataset, classname, style)

plt.title("Plot of the location by 1st and 2nd measures features in 2d")
plt.xlabel("1st feature value")
plt.ylabel("2nd feature value")
plt.legend()
plt.show()
```

Отже за даними із лабораторної роботи 1 було побудовано графіки класів у 2д просторі ознак розпізнавання, були позначені центри контейнерів класів.

Тепер перейдемо до знаходження необхідних відстаней.

Для початку внесемо у модуль код, що здатний вирахувати сусіда для кожного класу. У складі даного коду є метод calc_hemming_distance, який використовується для обчислення відстані Хеммінга.

```

    def __define_neighbours(self):
        neighbours = dict()

        # define neighbour for each class
        for class_name in self.__classes:

            # define classes set without current class
            other_classes = self.__classes.difference({class_name})

            # get the neighbour
            neighbour_name = self.__find_neighbour_for_current_class(class_name, other_classes)

            # save the neighbour
            neighbours[class_name] = neighbour_name
        return neighbours

    def __find_neighbour_for_current_class(self, target_class_name, other_classes):
        this_class_center = self.__centers.loc[target_class_name]
        distances = dict()

        # we have current class and than go for each other class and calc
        # hemming distance.
        for current_other_class_name in other_classes:
            current_other_class_center = self.__centers.loc[current_other_class_name]
            hemming_distance = self.__calc_hemming_distance(this_class_center,
                                                            current_other_class_center)

            # save the distances
            distances[current_other_class_name] = hemming_distance
        dists_frame = pd.DataFrame.from_dict(distances, orient="index", columns=[0])

        # class with the less distance will be the neighbour
        return dists_frame[0].idxmin()

    def __calc_hemming_distance(self, vector_1, vector_2):
        return len(np.where(vector_1 != vector_2)[0])

```

Тепер можемо перейти до вирішення основних завдань практики, а саме :

- Обчислення відстаней від класу до власних реалізацій
- Обчислення відстаней від класу до чужиші реалізацій
- Обчислення міжцентркових відстаней класу

Формати індексів дистанцій у таблицях:

центр-реалізація : {клас_центр}_[{клас_реалізації}]_{індекс_реалізації}

центр-центр : {клас1}_{клас2}

```

    def __practice_work(self, bin_data):
        # make tables for data
        columns = ["hemming_distance"]
        self_to_self = pd.DataFrame(columns=columns)
        self_to_another = pd.DataFrame(columns=columns)
        center_to_center = pd.DataFrame(columns=columns)

        for class_name in self.__classes:

            # add distances to self class
            sel_to_self_dict = self.__calc_dists_from_center_to_class(bin_data, class_name, class_name)
            self_to_self = pd.concat([self_to_self,
                                     pd.DataFrame.from_dict(sel_to_self_dict, orient="index", columns=columns)])

            # take another classes
            other_classes = self.__classes.difference({class_name})

            for other_class in other_classes:

                # assert is this dist between centers exists
                index_variant_first = "{}_{}".format(class_name, other_class)
                index_variant_second = "{}_{}".format(other_class, class_name)
                indexes = center_to_center.index
                if index_variant_first not in indexes and index_variant_second not in indexes:
                    # add if not
                    this_class_center = self.__centers.loc[class_name, self.__features]
                    other_class_center = self.__centers.loc[other_class, self.__features]
                    center_to_center_dist = self.__calc_hemming_distance(this_class_center,
                                                                           other_class_center)
                    center_to_center.loc[index_variant_first] = [center_to_center_dist]
                # add distance to all another realizations
                self_to_another_dict = self.__calc_dists_from_center_to_class(bin_data, class_name, other_class)
                self_to_another = pd.concat([self_to_another,
                                             pd.DataFrame.from_dict(self_to_another_dict, orient="index",
                                                                   columns=columns)])
            print("Self to Self")
            print(self_to_self)
            print("\nSelf to Another")
            print(self_to_another)
            print("\nCenters")
            print(center_to_center)

    def __calc_dists_from_center_to_class(self, bin_data, from_center_class, to_class_realizations):
        dists = dict()
        class_center = self.__centers.loc[from_center_class]
        realizations = bin_data[bin_data[self.__target] == to_class_realizations]

        for realiz_index in realizations.index:
            realiz_vals = realizations.loc[realiz_index, self.__features]
            dist = self.__calc_hemming_distance(class_center.values, realiz_vals.values)
            dists["{}-{}-{}".format(from_center_class, to_class_realizations, realiz_index)] = dist

        return dists

```

Дана функція `__practice_work` викликається із функції `__learn`.

```

    def __learn(self, train_data):
        binarized_dataset = self.__build_bin_feature_matrix(train_data)
        self.__centers = self.__build_standard_bin_classes_vectors(binarized_dataset)
        neighbours = self.__define_neighbours()
        self.__practice_work(binarized_dataset)

```

Результатами роботи

| Self to Self | |
|----------------|------------------|
| | hemming_distance |
| brick_brick_23 | 1634 |
| brick_brick_19 | 1306 |
| brick_brick_22 | 1576 |
| brick_brick_21 | 1406 |
| cloth_cloth_11 | 2401 |
| cloth_cloth_7 | 2167 |
| cloth_cloth_9 | 2401 |
| cloth_cloth_6 | 1493 |
| cloth_cloth_8 | 2899 |
| cloth_cloth_10 | 2396 |
| tile_tile_12 | 667 |
| tile_tile_17 | 518 |
| tile_tile_15 | 1168 |
| tile_tile_13 | 149 |
| wood_wood_4 | 2600 |
| wood_wood_2 | 1806 |
| wood_wood_0 | 1524 |
| wood_wood_1 | 1848 |
| wood_wood_3 | 1897 |

| Self to Another | |
|-----------------|------------------|
| | hemming_distance |
| brick_cloth_11 | 3058 |
| brick_cloth_7 | 2758 |
| brick_cloth_9 | 2962 |
| brick_cloth_6 | 1832 |
| brick_cloth_8 | 3464 |
| brick_cloth_10 | 3117 |
| brick_tile_12 | 1284 |
| brick_tile_17 | 1123 |
| brick_tile_15 | 1655 |
| brick_tile_13 | 758 |
| brick_wood_4 | 3315 |
| brick_wood_2 | 4199 |
| brick_wood_0 | 3629 |
| brick_wood_1 | 2095 |

| | |
|---------------|------|
| cloth_tile_15 | 1928 |
| cloth_tile_13 | 1125 |
| cloth_wood_4 | 3614 |
| cloth_wood_2 | 4240 |
| cloth_wood_0 | 3712 |
| cloth_wood_1 | 2418 |
| cloth_wood_3 | 3113 |
| tile_brick_23 | 1879 |
| tile_brick_19 | 1629 |
| tile_brick_22 | 2121 |
| tile_brick_21 | 1719 |
| tile_cloth_11 | 2975 |
| tile_cloth_7 | 2547 |
| tile_cloth_9 | 2821 |
| tile_cloth_6 | 1443 |
| tile_cloth_8 | 3373 |
| tile_cloth_10 | 2944 |
| tile_wood_4 | 3426 |
| tile_wood_2 | 4534 |
| tile_wood_0 | 3800 |
| tile_wood_1 | 1998 |
| tile_wood_3 | 2945 |
| wood_brick_23 | 3489 |
| wood_brick_19 | 2741 |
| wood_brick_22 | 2461 |
| wood_brick_21 | 3323 |
| wood_cloth_11 | 3523 |
| wood_cloth_7 | 3545 |
| wood_cloth_9 | 3605 |
| wood_cloth_6 | 3443 |
| wood_cloth_8 | 3739 |
| wood_cloth_10 | 3602 |
| wood_tile_12 | 3197 |
| wood_tile_17 | 3650 |
| wood_tile_15 | 3818 |
| wood_tile_13 | 3291 |

| Centers | |
|----------------|------------------|
| | hemming_distance |
| brick_cloth | 1467 |
| brick_tile | 631 |
| brick_wood | 3083 |
| cloth_tile | 1014 |
| cloth_wood | 3358 |
| tile_wood | 3298 |
| brick_wood_3 | 2946 |
| cloth_brick_23 | 2373 |
| cloth_brick_19 | 2139 |
| cloth_brick_22 | 2419 |
| cloth_brick_21 | 2265 |
| cloth_tile_12 | 1585 |
| cloth_tile_17 | 1424 |

Практична робота 4

Завдання

У практичній роботі 4 завданням є створити метод для визначення чисельних характеристик моделі та обчислення критерію інформаційної міри Кульбака. Як радує використовувати половину кількості ознак розпізнавання.

Уточнення

Для початку побудуємо **confusion matrix** для ефективної візуалізації чисельних характеристик даної моделі.

| | | Confusion matrix | | |
|--------------------|-------|------------------|-------|----|
| | | Real class | | |
| | | True | False | |
| Predicted class | True | k1 | k3 | n3 |
| | False | k2 | k4 | n4 |
| | | n1 | n2 | |

Червоним кольором на даному малюнку позначено помилки класифікації. Але для вирахування критерію Кульбака ми будемо використовувати саме ймовірності, тому обчислимо відношення значення клітинки на сумарне значення стовпчика(n_1 та n_2 відповідно)

| | | Real class | | |
|-----------------|-------|------------|---------|-------|
| | | True | False | |
| Predicted class | True | D1 | β | n_3 |
| | False | α | D2 | n_4 |
| | | n_1 | n_2 | |

Для обчислення критерію кульбака використовуватиме наступну формулу :

$$\log_2 \left(\frac{2 - (\alpha_m^{(k)}(d) + \beta_m^{(k)}(d))}{\alpha_m^{(k)}(d) + \beta_m^{(k)}(d)} \right) * \left[1 - (\alpha_m^{(k)}(d) + \beta_m^{(k)}(d)) \right].$$

При цьому для уникнення ділення на 0 і взяття логарифма від 0 в чисельник і знаменник підлогарифмічного виразу додамо нескінченно мале число(10 в степені -10), яке буде мало впливатиме на результат обчислень, але дозволить уникнути проблем під час роботи комп'ютерної програми.

Код програми

Зміни вносимо до класу **Model**. Для даної практики проредагуємо метод `__practice_work` та додамо наступні методи:

```
def __practice_work(self, bin_data):
    radius = len(self.__features) / 2

    for cur_class in self.__classes:
        # get class center
        cur_class_center = self.__centers.loc[cur_class].values

        # get class values
        cur_class_matrix = bin_data.loc[bin_data[self.__target] == cur_class, self.__features].values

        # get not class values
        other_classes_matrix = bin_data.loc[bin_data[self.__target] != cur_class, self.__features].values

        print("Class {}".format(cur_class))
        self.__calc_inf_efficiency_coefficient(radius, cur_class_center, cur_class_matrix, other_classes_matrix)

def __calc_inf_efficiency_coefficient(self, radius, goal_class_center,
                                         cur_class_matrix, nghbr_class_matrix):
    # calc k1, k2
    cur_in, cur_out = self.__calc_how_many_in_out_measures(radius, goal_class_center, cur_class_matrix)

    # calc k3, k4
    nghbr_in, nghbr_out = self.__calc_how_many_in_out_measures(radius, goal_class_center, nghbr_class_matrix)

    alpha = cur_out/(cur_in + cur_out)
    beta = nghbr_in/(nghbr_in + nghbr_out)
    KFE_criteria = self.__calc_KFE_criteria(alpha, beta)

    print("k1:{}, k2: {}, k3: {}, k4: {}, KFE: {}".format(cur_in, cur_out, nghbr_in, nghbr_out, KFE_criteria))

    return KFE_criteria

def __calc_how_many_in_out_measures(self, radius, center, measures):
    in_measures = 0
    out_measures = 0

    for measure in measures:
        hemming_dist = self.__calc_hemming_distance(measure, center)
        if hemming_dist <= radius:
            in_measures += 1
        else:
            out_measures += 1
    return in_measures, out_measures

def __calc_KFE_criteria(self, alpha, beta):
    r = -10
    criteria_result = log2((2 + pow(10, r) - alpha - beta) / (alpha + beta + pow(10, r))) * (1 - alpha - beta)
    return criteria_result
```

Результати роботи

```
Class cloth
k1:5, k2: 0, k3: 13, k4: 1, KFE: 0.014746491246193608
Class brick
k1:5, k2: 0, k3: 14, k4: 0, KFE: 0.0
Class tile
k1:4, k2: 0, k3: 14, k4: 1, KFE: 0.01284300519487159
Class wood
k1:5, k2: 0, k3: 11, k4: 3, KFE: 0.1345781191174793
```

Практична робота 5

Завдання

Написати програму обчислення максимально ефективних радіусів контейнерів класів.

Уточнення

У програмному коді будемо підбирати радіус у проміжку [1, N], де N - кількість ознак розпізнавання. Радіус із максимальним значенням KFE буде прийнятий за основу. Для тестування функціональної ефективності братимемо лише реалізації власного класу і класу-сусіда.

Код програми

Зміни вноситимемо у клас **Module**.

```
def __build_optimal_classes_radiuses(self, bin_dataset, neighbours):
    # containers for radiuses and KFE
    radiuses = dict()
    func_eff_coefs = dict()

    # styles for plotting
    styles = ["g", "b", "r", "y"]

    for cur_class_name, style in zip(self.__classes, styles):
        nghbr_class_name = neighbours[cur_class_name]

        cur_class_center = self.__centers.loc[cur_class_name].values
        cur_class_msrs_matrix = self.__get_measures(cur_class_name, bin_dataset)
        nghbr_class_msrs_matrix = self.__get_measures(nghbr_class_name, bin_dataset)

        radiuses[cur_class_name], func_eff_coefs[cur_class_name] = self.__calc_optimal_radius(cur_class_center,
                                                                                           cur_class_msrs_matrix,
                                                                                           nghbr_class_msrs_matrix,
                                                                                           style,
                                                                                           cur_class_name)

    plt.title("Dependency of KFE from radius value")
    plt.xlabel("Radius")
    plt.ylabel("KFE value")
    plt.legend()
    plt.show()

    return pd.DataFrame.from_dict(radiuses, orient="index", columns=["radius"]),
           pd.DataFrame.from_dict(func_eff_coefs, orient="index", columns=["KFE"])
```

```

def __get_measures(self, class_name, bin_dataset):
    return bin_dataset.loc[bin_dataset[self.__target] == class_name, self.__features].values

def __calc_optimal_radius(self, goal_class_center, goal_class_matrix, neighbour_class_matrix, style, class_name):
    # func for building plot
    def build_plot():
        radiusses = np.array(list(cases.keys()))
        KFEs = np.array(list(cases.values()))
        plt.plot(radiusses, KFEs, style, label=class_name)

    cases = dict()
    feature_number = goal_class_matrix.shape[1]

    for radius in range(1, feature_number+1):
        best_func_efficiency_coof = self.__calc_inf_efficiency_coefficient(radius,
                                                                           goal_class_center,
                                                                           goal_class_matrix,
                                                                           neighbour_class_matrix)
        cases[radius] = best_func_efficiency_coof

    build_plot()

    result_radius = pd.DataFrame.from_dict(cases, orient="index").idxmax().values[0]
    return result_radius, cases[result_radius]

```

Даний код викликається для обчислення максимально ефективних радіусів і КФЕ при них. Дані методи викликаються із методу `__learn`.

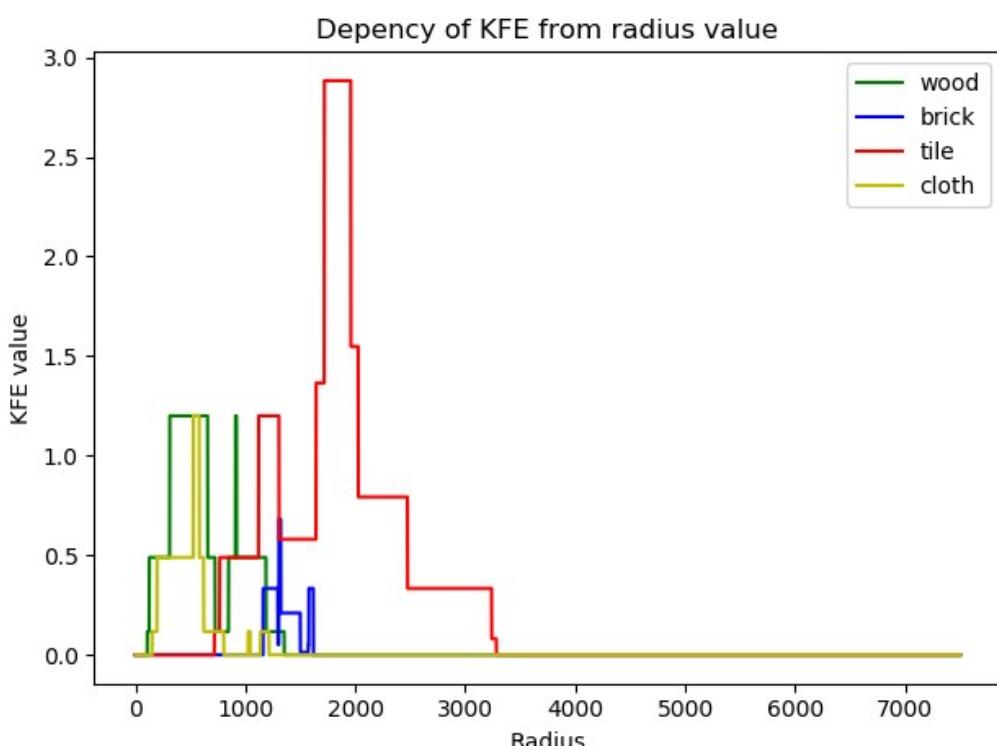
```

def __learn(self, train_data):
    binarized_dataset = self.__build_bin_feature_matrix(train_data)
    self.__centers = self.__build_standard_bin_classes_vectors(binarized_dataset)
    neighbours = self.__define_neighbours()
    self.__radiusses, self.__coefs = self.__build_optimal_classes_radiusses(binarized_dataset, neighbours)

    for class_name in self.__classes:
        print("Class {} with radius {} and KFE {}".format(class_name, self.__radiusses.loc[class_name, "radius"],
                                                          self.__coefs.loc[class_name, "KFE"]))

```

Результатом роботи



```

Class wood with 312 radius and 1.1999999998376967 KFE
Class brick with 1305 radius and 0.6810680886170737 KFE
Class tile with 1718 radius and 2.8828596815419782 KFE
Class cloth with 526 radius and 1.1999999998376967 KFE

```

Практична робота 6

Завдання

Організувати механізм екзамену.

Уточнення

Приналежність реалізації до якогось із класу будемо вираховувати за формулою:

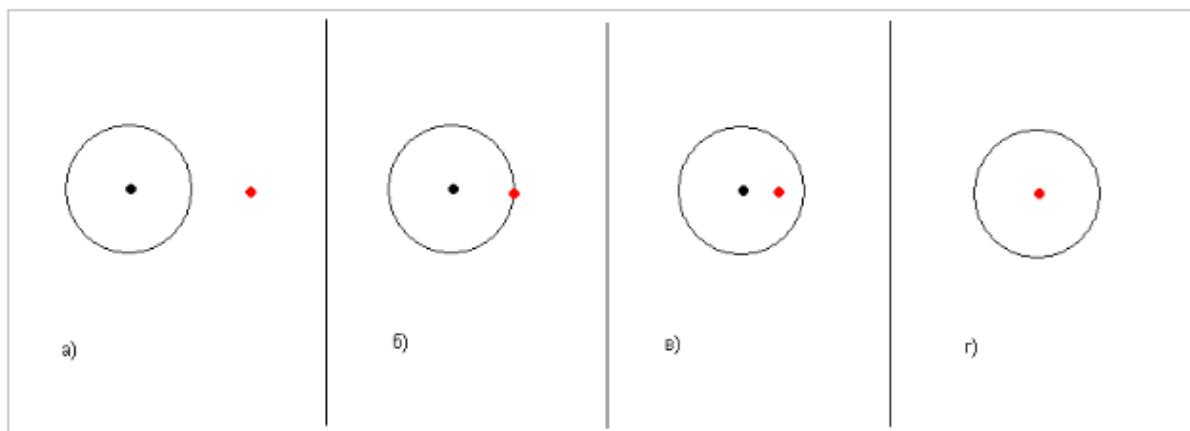
$$\mu_m = \left[1 - \frac{d(x_m \oplus x^{(j)})}{d_m^*} \right]$$

Тут μ_m - значення функції належності до m-того класу розпізнавання(стану об'єкту),

В чисельнику виразу знаходиться кодова відстань від екзаменаційної реалізації до еталонного вектора m-того класу, в знаменнику-оптимальний радіус контейнера m- того класу.

В результаті можна отримати наступні значення функції належності в залежності від позиції реалізації(На рисунку 6 зображені характерні ситуації положення реалізації відносно контейнера. Реалізація позначена червоним кольором, а контейнер - чорним)

- 1) Реалізація знаходитьться поза контейнером(Рис а)-тоді $\mu_m < 0$.
- 2) Реалізація знаходитьться на межі контейнера(Рис б)-тоді $\mu_m = 0$.
- 3) Реалізація знаходитьться в контейнері (Рис в)-тоді $\mu_m \in (1,0)$.
- 4) Реалізація знаходитьться в центрі контейнера (Рис г)-тоді $\mu_m > 1$.



Код програми

Нагадаємо, що в лабораторній роботі 2 було прийняте рішення кожного разу випадково розбивати вибірку на дві частини у відношенні 4 : 1.

```
# split it into train and exam
train = dataset.sample(frac=0.8)
test = dataset.drop(train.index)
```

Тому із самого початку роботи у зчитаний датасет закладено можливість тесту, тому відсутня необхідність дозавантажувати додаткові зображення текстури.

Основний метод класифікації розташуємо у класі **Module**.

```
def classify(self, dataset):
    predicted_classes = list()

    bin_data = self.__build_bin_feature_matrix(dataset)

    for data_index in bin_data.index:
        # get feature vector of current measure
        feature_vector = bin_data.loc[data_index, self.__features].values

        # all values of belonging class function
        belong_frame = self.__define_class(feature_vector)

        # define what class is it
        max_belong_val = belong_frame["btc"].max()
        if max_belong_val < 0:
            predicted_classes.append("no class")
        else:
            class_name = belong_frame["btc"].idxmax()
            predicted_classes.append(class_name)
    return predicted_classes

def __define_class(self, feature_vector):
    class_belong_to_class = dict()
    for cur_class in self.__classes:
        # calc current radius and center
        cur_class_center = self.__centers.loc[cur_class]
        cur_class_radius = self.__radiuses.loc[cur_class, "radius"]

        distance = self.__calc_hemming_distance(cur_class_center, feature_vector)

        # calc and add class belonging function
        class_belonging_function = 1 - distance / cur_class_radius
        class_belong_to_class[cur_class] = class_belonging_function

    return pd.DataFrame.from_dict(class_belong_to_class, orient="index", columns=["btc"])
```

Додамо метод до **IEIModelAPI**

```
def predict(self, data):
    return self._model.classify(data)
```

І нарешті у тестовій інформації створимо колонку **predicted** для порівняння реального і визначеного класу у файлі **main.py**

```
test["predicted"] = model.predict(test)

print(test)
```

Результати роботи

| | 1:1:1 | 1:1:2 | 1:1:3 | 1:2:1 | ... | 50:50:2 | 50:50:3 | class | predicted |
|----|-------|-------|-------|-------|-----|---------|---------|-------|-----------|
| 7 | 145.0 | 153.0 | 166.0 | 154.0 | ... | 64.0 | 85.0 | cloth | no class |
| 9 | 78.0 | 76.0 | 77.0 | 68.0 | ... | 59.0 | 62.0 | cloth | no class |
| 11 | 166.0 | 168.0 | 165.0 | 123.0 | ... | 104.0 | 97.0 | cloth | no class |
| 17 | 184.0 | 163.0 | 134.0 | 189.0 | ... | 170.0 | 143.0 | tile | tile |
| 22 | 181.0 | 89.0 | 64.0 | 178.0 | ... | 84.0 | 63.0 | brick | no class |

Практична робота 7

Завдання

Створити механізм визначення сусіднього класу.

Вирішення

Даний механізм був уже створений і використаний у лабораторній роботі 3 для визначення сусідніх класів за міжцентровою відстанню. Даний код економить час і пам'ять при обробці даних при визначенні оптимального радіусу і КФЕ, беручи лише реалізації даного і його сусіднього класів.

Практична робота 8

Завдання

Створити функціонал із підбору оптимального радіусу поля контрольних допусків. Вивести на екран графік залежності КФЕ від значення дельти.

Уточнення

Для підбору оптимального значення дельти поля контрольних допусків використовуватимемо робочу область із нижньою межею 20. Враховуючи, що всі ознаки розпізнавання мають одну і ту ж шкалу вимірювання, дельти поля контрольних допусків будуть спільними. Верхньою межею даної величини будемо вважати середнє значення всіх ознак розпізнавання центру контейнера базового класу.

Код програми

Використаємо вже створену структуру класів. Клас **Model** є закінченою моделлю із визначеними наперед середніми значеннями базового класу і дельтами поля допусків, які передаються класу в конструкторі під час створення. Оптимізацію поля допусків будемо проводити у класі **IEIModelAPI** у методі **__train**. Для оптимізації використаємо паралельну обробку даних на багатопроцесорному комп’ютері у методі **__parallel_delta_optimising**.

```
def __train(self, train_data):
    # define the most common class
    base_class = train_data[self.__target].mode().values[0]

    # calc mean feature values for base class
    mean_base_class_val = self.__mean_base_class_values(train_data, base_class)

    # get top border for delta optimizing process
    top_delta_border = int(mean_base_class_val.mean())

    # if mean top delta less than start delta than
    # use only start delta and start delta + 1 values for optimization
    if top_delta_border <= self.__tolerance_dist_start:
        top_delta_border = self.__tolerance_dist_start+1

    # number of cpu
    chunks = 4

    # prepare data for parallel processing
    generators = self.__split_deltas_range(self.__tolerance_dist_start, top_delta_border+1, chunks)
    futures = list()
    delta_kfe_pairs = dict()

    # make parallel processing
    with ThreadPoolExecutor(max_workers=chunks) as executor:
        for delta_gen in generators:
            futures.append(executor.submit(self.__parallel_delta_optimising,
                                            train_data, mean_base_class_val, delta_gen))

    # get together the parallel processing results
    for result in as_completed(futures):
        delta_kfe_pairs.update(result.result())

    # format the result data of processing
    deltas_kfe_frame = pd.DataFrame.from_dict(delta_kfe_pairs, orient="index", columns=["KFE"]).sort_index()
    best_delta = deltas_kfe_frame["KFE"].idxmax()

    # construct the final model through the optimal delta
    self.__model = self.__create_model(train_data, mean_base_class_val, best_delta)

    print("Model builded with {} delta and {} KFE ".format(best_delta, self.__model.get_overall_KFE()))

    plt.plot(deltas_kfe_frame.index, deltas_kfe_frame["KFE"].values)
    plt.title("Dependence of KFE from the delta value")
    plt.xlabel("delta")
    plt.ylabel("KFE")
    plt.show()
```

```

# split the whole delta range for parallel processing
def __split_deltas_range(self, start, finish, chunks):
    list_of_chunks = list()
    chunk_step = int((finish - start) / chunks)

    cur_start = start
    cur_finish = cur_start + chunk_step

    while cur_start < finish:
        if cur_finish + chunk_step - 1 >= finish:
            cur_finish = finish
        gen = range(cur_start, cur_finish)

        list_of_chunks.append(gen)

        cur_start = cur_finish
        cur_finish += chunk_step
    return list_of_chunks

# main data for the parallel processing
def __parallel_delta_optimising(self, data, mean_base_class_vals, deltas_generator):
    delta_kfe = dict()

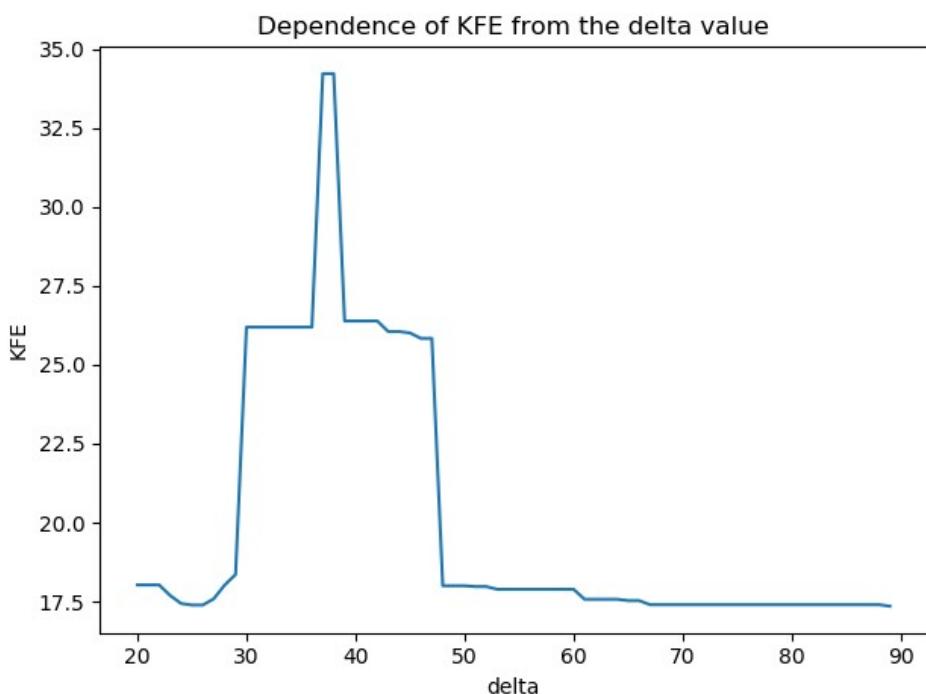
    for cur_delta in deltas_generator:

        # create model for current delta abd get KFE
        model = self.__create_model(data, mean_base_class_vals, cur_delta)
        kfe = model.get_overall_KFE()
        delta_kfe[cur_delta] = kfe
    return delta_kfe

def __create_model(self, data, mean_base_class_vals, delta):
    features_len = len(mean_base_class_vals)
    deltas_vector = np.array([delta] * features_len)
    return Model(data, self.__features, self.__target, mean_base_class_vals, deltas_vector)

```

Результатами роботи



Model builded with 37 delta and 34.21928094894576 KFE

Практична робота 9

Завдання

Створити нову модель розпізнавання, яка працюватиме із зображеннями аерофотозйомки і класифікуватиме квадрати місцевості. При цьому в результаті навчання вивести на екран графіки залежності КФЕ від радіусів контейнерів та КФЕ від значення дельти.

Виконання

Для початку завантажимо наступне зображення аерофотозйомки :

aerophoto.jpg



Дане зображення використаємо для того, щоб створити навчальну вибірку чотирьох класів : **[field, road, town, water]**. Для цього створимо файли:

- **split_image.py** , який зберігатиме у собі функціонал для розрізання загальних картинок на менші квадратики
- **make_train_aero_photo.py** для створення тестової вибірки.

Далі напишемо функціонал, що розріже дане фото на блоки розмірами **50 * 50** і збереже дані фото у директорію **aero_photo_train** , де кожному фото присвоємо за ім'я точні дату і час збереження. Після даних дій необхідно продивитися наявні фото і розподілити їх по класах

розділення. Клас розпізнавання присвоюється перейменуванням фото у вигляді паттерну `{назва класу}{довільний номер}.jpg`. Приклад імені : field1.jpg

make_train_aero_photo.py

```
from split_image import split_the_image_in_boxes
from skimage import io
from datetime import datetime as dt

parent_dir = "aero_photo_train"
image_name = "aerophoto.jpg"
std_size = (50, 50, 3)

image_boxes = split_the_image_in_boxes(image_name, std_size)

for small_image in image_boxes:
    path_to_save = "{}/{}.jpg".format(parent_dir, dt.now())
    io.imsave(path_to_save, small_image)
```

split_image.py

```
from skimage import io
from skimage.util.shape import view_as_blocks
import numpy as np

def split_the_image_in_boxes(image_path, standard_size):
    standard_high, standard_width, number_of_colors = standard_size

    # read the image
    all_pixels = io.imread(image_path)

    # calc params for extract the compatibility image
    real_height = all_pixels.shape[0]
    real_width = all_pixels.shape[1]
    compatible_height_border = standard_high * (real_height // standard_high)
    compatible_width_border = standard_width * (real_width // standard_width)

    # split the image and divide it into blocks
    compatible_image = all_pixels[:compatible_height_border, :compatible_width_border, :]
    image_blocks_matrix = view_as_blocks(compatible_image, standard_size)

    # reshape for comfortable processing
    rows, cols = image_blocks_matrix.shape[:2]
    image_blocks_list = np.reshape(image_blocks_matrix, newshape=(rows * cols, *standard_size))

    return image_blocks_list
```

Загалом в результаті роботи даного коду велике зображення було розбито на 2340 невеликих фото розміром 50 * 50 і з них було відібрано 44 фото(11 на один клас) для побудови моделі розпізнавання.

Далі створюємо файл **making_model_of_aero_photo.py** , у якому відбувається створення моделі розпізнавання. Враховуючи, що створення моделі займає багато часу — збережемо всі параметри даної моделі у директорії **model_bin** у вигляді бінарного файлу із ім'ям **model.pkl**.

making_model_of_aero_photo.py

```
from prepare_package.prepare_module import read_dataset
from models_module.IEI import IEIModelAPI
import pickle as pkl

# where small images situated
source_dir = "aero_photo_train"

# properties for extracting dataset
classes = ["field", "road", "town", "water"]
file_pattern = "*{}*.jpg"
standard_shape = (50, 50, 3)

# where save the model
target_dir_for_bin_model = "model_bin"

# get the dataset from files in pandas data frame format
dataset, features = read_dataset(source_dir, classes, file_pattern, standard_shape)

model = IEIModelAPI(dataset, features, "class")

# serialize the model
with open("{}_{}.pkl".format(target_dir_for_bin_model, "model"), "wb") as bin_model_file:
    pkl.dump(model, bin_model_file, protocol=pkl.HIGHEST_PROTOCOL)
```

Весь інший функціонал, використаний у даному лістингу коду вже відомий із попередніх робіт.

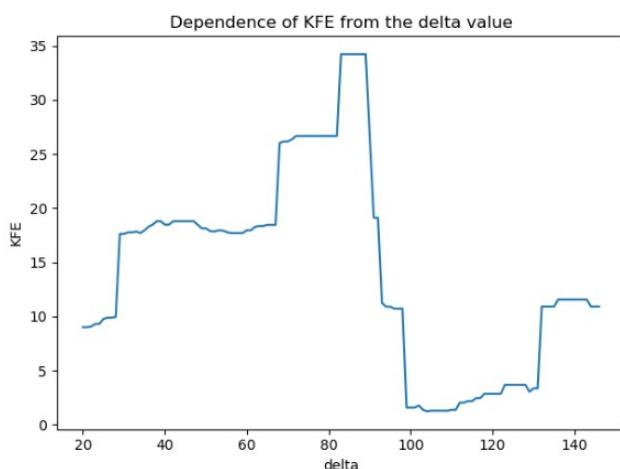
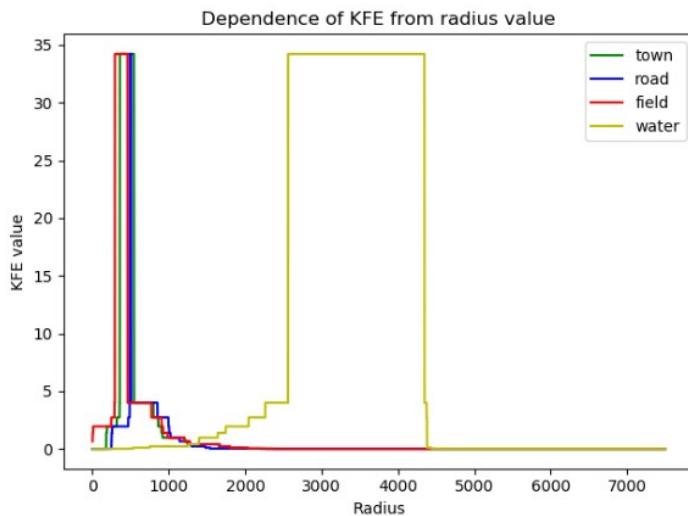
Але необхідно зробити уточнення : стандартна процедура побудови моделі знаходиться у класі **Model.Model** і нерозривно пов'язана із побудовою графіків залежності і візуалізацією результатів. Даний клас створюється багато разів при підборі оптимальної дельти, тому було прийняте рішення у процесі оптимізації програмно вимикати у даному класі побудову графіків. Так до конструктора класу додано спеціальній прапорець із логічним значенням True або False.

```
def __init__(self, train_data, features, target, mean_base_class_vals, tol_interval_delta, with_report=False):
    self.__with_report = with_report
```

Кожен раз при необхідності у екземплярі даного класу побудувати графік перевіряється значення внутрішньої змінної **__with_report**.

В кінці підбору дельти створюється фінальний об'єкт **Model**, який вже буде відповісти із ввімкненою візуалізацією.

Результати побудови моделі :



```
Class town with radius 359 and KFE 34.21928082957664
Class road with radius 489 and KFE 34.21928082957664
Class field with radius 294 and KFE 34.21928094894576
Class water with radius 2562 and KFE 34.21928094894576
Model builded with 83 delta and 34.2192808892612 KFE
```

Для тестування даної моделі створимо файл **make_test_of_the_aero_photo_model.py**.

Врахуємо також, що із зображення **aerophoto.jpg** із 2340 блоків $50 * 50$ для навчання системи ми брали лише 44. Тому дане фото також використовуємо для нашої моделі у дії. Знову ж таки розріжемо його на блоки $50 * 50$, перетворимо у **pandas DataFrame** у вигляді векторів кольору розміром 7500($50 * 50 * 3$) і за допомогою моделі, яку ми розпакуємо із файла **model_bin/model.pkl** зробимо висновок про приналежність даних реалізацій до одного з класів [**field**, **road**, **town**, **water**] або жодного із них (no class лейбл). Кожне із наявних

зображенъ збережемо у директорії **aero_photo_predict** з паттерном імені **{номер_y_таблиці}_{присуджений клас}**.

make_test_of_the_aero_photo_model.py

```
import pickle as pkl
from prepare_package.prepare_module import make_columns
from split_image import split_the_image_in_boxes
import pandas as pd
from skimage.io import imsave
import numpy as np

def make_frame_of_images_matrices(matrices, features_columns):
    features_len = len(features_columns)

    # container for collecting all feature vectors
    index_measure_dict = dict()

    for i in range(matrices.shape[0]):
        feature_vector = matrices[i].reshape(features_len)
        index_measure_dict[i] = feature_vector

    return pd.DataFrame.from_dict(index_measure_dict, orient="index", columns=features_columns)

# path of the aero photo image
aero_photo_image = "aerophoto.jpg"

# standard size of the small images
std_size = (50, 50, 3)

# make columns names for dataset
columns = make_columns(std_size)

# cut the big image
small_images = split_the_image_in_boxes(aero_photo_image, std_size)

# make the data frame for testing
test_dataset = make_frame_of_images_matrices(small_images, columns)

# free memory
del small_images

# path to model
model_path = "model_bin/model.pkl"

with open(model_path, "rb") as model_file:
    model = pkl.load(model_file)
# create column inside dataset with predicted results
test_dataset["predicted"] = model.predict(test_dataset)

for index in test_dataset.index:
    # define the name of the image
    path_where_to_save = "aero_photo_predict/{}_{}.jpg".format(index, test_dataset.loc[index, "predicted"])

    # convert to the std size from feature vector size and save
    imsave(path_where_to_save, np.array(test_dataset.loc[index, columns].values, dtype="int").reshape(std_size))
```

Після роботи даного коду ми отримаємо 2340 зображень розміру 50 * 50 у директорії **aero_photo_predict**. Для підтвердження роботи алгоритму приведемо приклад огляду каталогів із програмами огляду каталогів.



Детальніше із повним переліком даних фото можна буде ознайомитися у репозиторії проекту, що буде вказаний у пункті результатів роботи.

Висновки

Під час даної роботи були отримані знання реалізації IEI алгоритму класифікації. Ідея даної моделі полягає у розташуванні контейнерів класів, що не перетинаються у просторі ознак розпізнавання. Процес навчання відбувається за допомогою підбору оптимальних значень радіусів контейнерів класів і поля контрольних допусків за найвищим коефіцієнтом ефективності. Ефективність даної моделі визначається за критерієм Кульбака за допомогою точнісних характеристик прийняття рішення.

Модель IEI алгоритму була реалізована на мові програмування Python. Під час кодування программи було отримано досвід роботи із пакетами, що не входять до стандартної бібліотеки Python 3.7:

- Pandas
- Numpy
- Skimage

Також було поглиблено знання із ООП, мультипоточного програмування та стандартної бібліотеки Python 3.7.

Під час роботи над даним проектом було створено репозиторій у системі контролю версій GitHub за адресою : <https://github.com/JavaStudentAlex/IntelligenceSystems.git>

Додаток

Система каталогів проекту

```
▼ └── InteligenceSystems ~/PycharmProjects/InteligenceSystems
    ├── aero_photo_predict
    ├── aero_photo_train
    ├── images_for_labs
    ├── model_bin
    └── models_module
        ├── IEI.py 29.05.2020, 07:46, 4,54 kB Today 12:29
        ├── Model.py 28.05.2020, 17:11, 10,46 kB 59 minutes ago
    └── prepare_package
        ├── prepare_module.py 29.05.2020, 12:10, 3,14 kB Today 13:23
        ├── aerophoto.jpg 28.05.2020, 09:35, 1,19 MB
        ├── main.py 28.05.2020, 16:29, 687 B 12 minutes ago
        ├── make_test_of_the_aero_photo_model.py 29.05.2020, 14:39, 1,7 kB 12 minutes ago
        ├── make_train_aero_photo.py 29.05.2020, 11:26, 391 B 12 minutes ago
        ├── making_model_of_aero_photo.py 29.05.2020, 11:34, 754 B 12 minutes ago
        ├── plotting.py 28.05.2020, 13:26, 1,02 kB 12 minutes ago
        └── split_image.py 28.05.2020, 16:07, 992 B 3 minutes ago
```

- aero_photo_predict — директорія для збереження класифікованих зображень в лабораторній роботі 9
- aero_photo_train — директорія для тестових 44 зображень із лабоаторної роботи 9
- images_for_labs — директорія для фото текстур із перших 8 лабораторних
- model_bin — директорія для зберігання бінарного файлу моделі із 9 лабораторної роботи
- models_module — пакет для зберігання основних класів IEI моделі
- prepare_package — пакет для класів підготовки датасету до навчання

IEI.py — API моделі. Підбирає дельту поля контрольних допусків і містить у собі базовий клас моделі.

```
import numpy as np
from models_module.Model import Model
from matplotlib import pyplot as plt
from concurrent.futures import ThreadPoolExecutor, as_completed
import pandas as pd

class IEIModelAPI:
    _tolerance_dist_start = 20

    # main constructor
    def __init__(self, train_data, features, target):
        # save all key properties
        self.__features = features
        self.__target = target
        self.__classes = {*np.unique(train_data[self.__target].values)}

        # start training
        self.__train(train_data)

    def __train(self, train_data):
        # define the most common class
        base_class = train_data[self.__target].mode().values[0]

        # calc mean feature values for base class
        mean_base_class_val = self.__mean_base_class_values(train_data, base_class)

        # get top border for delta optimizing process
        top_delta_border = int(mean_base_class_val.mean())

        # if mean top delta less than start delta than
        # use only start delta and start delta + 1 values for optimization
        if top_delta_border <= self.__tolerance_dist_start:
            top_delta_border = self.__tolerance_dist_start+1

        # number of cpu
        chunks = 4

        # prepare data for parallel processing
        generators = self.__split_deltas_range(self.__tolerance_dist_start, top_delta_border+1, chunks)
        futures = list()
        delta_kfe_pairs = dict()

        print(self.__tolerance_dist_start, top_delta_border)

        # make parallel processing
        with ThreadPoolExecutor(max_workers=chunks) as executor:
            for delta_gen in generators:
                futures.append(executor.submit(self.__parallel_delta_optimising,
                                                train_data, mean_base_class_val, delta_gen))

            # get together the parallel processing results
            for result in as_completed(futures):
                delta_kfe_pairs.update(result.result())

        # format the result data of processing
        deltas_kfe_frame = pd.DataFrame.from_dict(delta_kfe_pairs, orient="index", columns=["KFE"]).sort_index()
        best_delta = deltas_kfe_frame["KFE"].idxmax()
```

```

# construct the final model through the optimal delta
self.__model = self.__create_model(train_data, mean_base_class_val, best_delta, with_report=True)

print("Model builded with {} delta and {} KFE ".format(best_delta, self.__model.get_overall_KFE()))

plt.plot(deltas_kfe_frame.index, deltas_kfe_frame["KFE"].values)
plt.title("Dependence of KFE from the delta value")
plt.xlabel("delta")
plt.ylabel("KFE")
plt.show()

def __mean_base_class_values(self, dataset, most_freq_class):
    # get base class feature values from dataset
    values_matrix = dataset.loc[dataset[self.__target] == most_freq_class, self.__features].values

    # get number of rows
    measure_number = values_matrix.shape[0]
    mean_feature_vals = np.sum(values_matrix, axis=0) / measure_number
    return mean_feature_vals

# split the whole delta range for parallel processing
def __split_deltas_range(self, start, finish, chunks):
    list_of_chanks = list()
    chunk_step = int((finish - start) / chunks)

    cur_start = start
    cur_finish = cur_start + chunk_step

    while cur_start < finish:
        if cur_finish + chunk_step - 1 >= finish:
            cur_finish = finish
        gen = range(cur_start, cur_finish)

        list_of_chanks.append(gen)

        cur_start = cur_finish
        cur_finish += chunk_step
    return list_of_chanks

# main data for the parallel processing
def __parallel_delta_optimising(self, data, mean_base_class_vals, deltas_generator):
    delta_kfe = dict()

    for cur_delta in deltas_generator:
        print(cur_delta)

        # create model for current delta and get KFE
        model = self.__create_model(data, mean_base_class_vals, cur_delta)
        kfe = model.get_overall_KFE()
        delta_kfe[cur_delta] = kfe
    return delta_kfe

def __create_model(self, data, mean_base_class_vals, delta, with_report=False):
    features_len = len(mean_base_class_vals)
    deltas_vector = np.array([delta] * features_len)
    return Model(data, self.__features, self.__target, mean_base_class_vals, deltas_vector, with_report)

def predict(self, data):
    return self.__model.classify(data)

```

Model.py — базовий программний клас моделі з певним базовим класом розпізнавання і полем допусків. Підбирає лише радіус.

```
import numpy as np
import pandas as pd
from math import log2
from matplotlib import pyplot as plt

class Model:
    _sel_level = 0.5

    def __init__(self, train_data, features, target, mean_base_class_vals, tol_interval_delta, with_report=False):
        self.__with_report = with_report

        # save key properties
        self.__features = features
        self.__target = target
        self.__classes = {*np.unique(train_data[self.__target].values)}

        # calc tol interval
        self.tol_dist_field_bottom_vals = mean_base_class_vals - tol_interval_delta
        self.tol_dist_field_top_vals = mean_base_class_vals + tol_interval_delta
        self.tol_dists = tol_interval_delta

        # start building the model
        self.__learn(train_data)

    def __learn(self, train_data):
        binarized_dataset = self.__build_bin_feature_matrix(train_data)
        self.__centers = self.__build_standard_bin_classes_vectors(binarized_dataset)
        neighbours = self.__define_neighbours()
        self.__radiuses, self.__coefs = self.__build_optimal_classes_radiuses(binarized_dataset, neighbours)

        if self.__with_report:
            for class_name in self.__classes:
                print("Class {} with radius {} and KFE {}".format(class_name,
                                                                    self.__radiuses.loc[class_name, "radius"],
                                                                    self.__coefs.loc[class_name, "KFE"]))

    def __build_bin_feature_matrix(self, dataset):
        val_matrix = dataset[self.__features].values

        # make new bin matrix
        bin_val_matrix = np.zeros(shape=val_matrix.shape)

        # assert which numbers are in tolerance interval
        positions = np.where((val_matrix > self.tol_dist_field_bottom_vals) &
                             (val_matrix < self.tol_dist_field_top_vals))

        # set 1 to numbers that are in tolerance interval
        bin_val_matrix[positions] = 1

        # create new DataFrame but with binary data
        bin_dataset = dataset.copy()
        bin_dataset[self.__features] = bin_val_matrix
        return bin_dataset
```

```

def __build_standard_bin_classes_vectors(self, binarized_dataset):
    centers = {}
    for class_name in self.__classes:

        # get bin data for each class
        class_matrix = binarized_dataset.loc[binarized_dataset[self.__target] == class_name, self.__features].values

        # calc the container center
        cont_center = self.__calc_container_center(class_matrix)

        # add to all centers
        centers[class_name] = cont_center

    # make the DataFrame of centers
    return pd.DataFrame.from_dict(centers, orient="index", columns=self.__features)

def __calc_container_center(self, matrix):
    # calc mean bin value for each feature
    mean_bin_vals = np.sum(matrix, axis=0) / matrix.shape[0]

    # create new center array
    cur_cont_center = np.zeros(matrix.shape[1])

    # find where mean values are bigger than selection level
    units_positions = np.where(mean_bin_vals > self.__sel_level)

    # set 1 to that positions
    cur_cont_center[units_positions] = 1
    return cur_cont_center

def __define_neighbours(self):
    neighbours = dict()

    # define neighbour for each class
    for class_name in self.__classes:

        # define classes set without current class
        other_classes = self.__classes.difference({class_name})

        # get the neighbour
        neighbour_name = self.__find_neighbour_for_current_class(class_name, other_classes)

        # save the neighbour
        neighbours[class_name] = neighbour_name
    return neighbours

def __find_neighbour_for_current_class(self, target_class_name, other_classes):
    this_class_center = self.__centers.loc[target_class_name]
    distances = dict()

    # we have current class and than go for each other class and calc
    # hemming distance.
    for current_other_class_name in other_classes:
        current_other_class_center = self.__centers.loc[current_other_class_name]
        hemming_distance = self.__calc_hemming_distance(this_class_center,
                                                       current_other_class_center)

```

```

        # save the distances
        distances[current_other_class_name] = hemming_distance
    dists_frame = pd.DataFrame.from_dict(distances, orient="index", columns=["0"])

    # class with the less distance will be the neighbour
    return dists_frame["0"].idxmin()

def __calc_hemming_distance(self, vector_1, vector_2):
    return len(np.where(vector_1 != vector_2)[0])

def __build_optimal_classes_radiuses(self, bin_dataset, neighbours):
    # containers for radiuses and KFE
    radiuses = dict()
    func_eff_coefs = dict()

    # styles for plotting
    styles = ["g", "b", "r", "y"]

    for cur_class_name, style in zip(self.__classes, styles):
        nghbr_class_name = neighbours[cur_class_name]

        cur_class_center = self.__centers.loc[cur_class_name].values
        cur_class_msrs_matrix = self.__get_measures(cur_class_name, bin_dataset)
        nghbr_class_msrs_matrix = self.__get_measures(nghbr_class_name, bin_dataset)

        radiuses[cur_class_name], func_eff_coefs[cur_class_name] = self.__calc_optimal_radius(cur_class_center,
                                                                                           cur_class_msrs_matrix,
                                                                                           nghbr_class_msrs_matrix,
                                                                                           style,
                                                                                           cur_class_name)

    # if user need report then show the graph
    if self.__with_report:
        plt.title("Dependence of KFE from radius value")
        plt.xlabel("Radius")
        plt.ylabel("KFE value")
        plt.legend()
        plt.show()

    return pd.DataFrame.from_dict(radiuses, orient="index", columns=["radius"]),
           pd.DataFrame.from_dict(func_eff_coefs, orient="index", columns=["KFE"])

def __get_measures(self, class_name, bin_dataset):
    return bin_dataset.loc[bin_dataset[self.__target] == class_name, self.__features].values

def __calc_optimal_radius(self, goal_class_center, goal_class_matrix, neighbour_class_matrix, style, class_name):
    # func for building plot
    def build_plot():
        radiuses = np.array(list(cases.keys()))
        KFEs = np.array(list(cases.values()))
        plt.plot(radiuses, KFEs, style, label=class_name)

    cases = dict()
    feature_number = goal_class_matrix.shape[1]

    for radius in range(1, feature_number+1):
        best_func_efficiency_coof = self.__calc_inf_efficiency_coefficient(radius,
                                                                           goal_class_center,
                                                                           goal_class_matrix,
                                                                           neighbour_class_matrix)
        cases[radius] = best_func_efficiency_coof

```

```

# it is not optimizing iteration, it is general container
if self.__with_report:
    build_plot()

result_radius = pd.DataFrame.from_dict(cases, orient="index").idxmax().values[0]
return result_radius, cases[result_radius]

def __calc_inf_efficiency_coefficient(self, radius, goal_class_center,
                                       cur_class_matrix, nghbr_class_matrix):
    # calc k1, k2
    cur_in, cur_out = self.__calc_how_many_in_out_measures(radius, goal_class_center, cur_class_matrix)

    # calc k3, k4
    nghbr_in, nghbr_out = self.__calc_how_many_in_out_measures(radius, goal_class_center, nghbr_class_matrix)

    alpha = cur_out/(cur_in + cur_out)
    beta = nghbr_in/(nghbr_in + nghbr_out)
    KFE_criteria = self.__calc_KFE_criteria(alpha, beta)

    return KFE_criteria

def __calc_how_many_in_out_measures(self, radius, center, measures):
    in_measures = 0
    out_measures = 0

    for measure in measures:
        hemming_dist = self.__calc_hemming_distance(measure, center)
        if hemming_dist <= radius:
            in_measures += 1
        else:
            out_measures += 1
    return in_measures, out_measures

def __calc_KFE_criteria(self, alpha, beta):
    r = -10
    criteria_result = log2((2 + pow(10, r) - alpha - beta) / (alpha + beta + pow(10, r))) * (1 - alpha - beta)
    return criteria_result
def get_overall_KFE(self):
    return self.__coefs["KFE"].mean()

def classify(self, dataset):
    predicted_classes = list()

    bin_data = self.__build_bin_feature_matrix(dataset)

    for data_index in bin_data.index:
        # get feature vector of current measure
        feature_vector = bin_data.loc[data_index, self.__features].values

        # all values of belonging class function
        belong_frame = self.__define_class(feature_vector)

        # define what class is it
        max_belong_val = belong_frame["btc"].max()
        if max_belong_val < 0:
            predicted_classes.append("no class")
        else:
            class_name = belong_frame["btc"].idxmax()
            predicted_classes.append(class_name)
    return predicted_classes

```

```
def __define_class(self, feature_vector):
    class_belong_to_class = dict()
    for cur_class in self.__classes:
        # calc current radius and center
        cur_class_center = self.__centers.loc[cur_class]
        cur_class_radius = self.__radiuses.loc[cur_class, "radius"]

        distance = self.__calc_hamming_distance(cur_class_center, feature_vector)

        # calc and add class belonging function
        class_belonging_function = 1 - distance/cur_class_radius
        class_belong_to_class[cur_class] = class_belonging_function

    return pd.DataFrame.from_dict(class_belong_to_class, orient="index", columns=["btc"])
```

prepare_module.py — модуль для читання та підготовки даних.

```
from itertools import product
from skimage import io
import pandas as pd
import numpy as np
import fnmatch
import os

# high level function for building dataset
def read_dataset(source_dir, classes, file_pattern, standard_shape):

    source_class_gen = make_images_sources(source_dir, classes, file_pattern)
    dataset = build_dataset(source_class_gen, standard_shape)
    return dataset

# read the source directory content and return all pathes of classes images_for_labs
def make_images_sources(source_dir, classes, file_pattern):
    sources = list()
    for class_name in classes:
        full_paths_files_source_dir = (os.path.abspath("{} / {}".format(source_dir, file_name)))
        for file_name in os.listdir(source_dir)
            class_sources = fnmatch.filter(full_paths_files_source_dir, file_pattern.format(class_name))
            sources.extend(class_sources)
            yield from product(class_sources, [class_name])

# read each image, cut it for standard size, reshape for feature vector form (1 * features)
# and insert into the DataFrame also with the class label
def build_dataset(source_class_gen, std_shape):
    columns = make_columns(std_shape) + ["class"]
    rows = list()

    for image_path, class_name in source_class_gen:
        feature_vector_length = np.prod(std_shape)
        image_matrix = io.imread(image_path)
        cut_img = cut_image(image_matrix, std_shape)
        feature_vector = cut_img.reshape(feature_vector_length)
        rows.append((*feature_vector, class_name))

    return pd.DataFrame(data=rows, columns=columns), columns[:-1]

def make_columns(std_shape):
    return ["{}:{}:{}:{}".format(*triple) for triple in product(range(1, std_shape[0] + 1),
                                                               range(1, std_shape[1] + 1),
                                                               range(1, std_shape[2] + 1))]

# cut standard image size from the center
def cut_image(image_matrix, std_shape):
    real_shape = image_matrix.shape

    height_border = calc_border(std_shape, real_shape, 0)
    width_border = calc_border(std_shape, real_shape, 1)

    height_size = calc_size(std_shape, real_shape, 0)
    width_size = calc_size(std_shape, real_shape, 1)
```

```
result_matrix = np.zeros(std_shape)
result_matrix[:height_size, :width_size] = image_matrix[height_border:height_border+height_size,
                                                       width_border:width_size+width_border]
return result_matrix

# calc the border value
def calc_border(std_vals, real_vals, axis_index):
    return only_positive_int_numbers((real_vals[axis_index] - std_vals[axis_index]) / 2)

# return positive number or 0
def only_positive_int_numbers(val):
    return int(val) if val > 0 else 0

# if the real image size param(width or height through the axis_index) is bigger than
# standard return standart value if not return real value of the image
def calc_size(std_vals, real_vals, axis_index):
    real_param = real_vals[axis_index]
    std_param = std_vals[axis_index]
    return std_param if real_param - std_param > 0 else real_param
```

main.py — файл для керування навчанням у 1-8 лабораторних роботах.

```
from prepare_package.prepare_module import read_dataset
from models_module.IEI import IEIModelAPI

source_dir = "images_for_labs"
classes = ["wood", "cloth", "tile", "brick"]
file_pattern = "*{}*.jpg"
standard_shape = (50, 50, 3)

# get the dataset from files in pandas data frame format
dataset, features = read_dataset(source_dir, classes, file_pattern, standard_shape)

# split it into train and exam
train = dataset.sample(frac=0.8)
test = dataset.drop(train.index)

# make the model for predictions
model = IEIModelAPI(train, features, "class")

test["predicted"] = model.predict(test)
```

make_test_of_the_aero_model_photo.py — модуль для тестування моделі у лабораторній роботі 9

```
import pickle as pkl
from prepare_package.prepare_module import make_columns
from split_image import split_the_image_in_boxes
import pandas as pd
from skimage.io import imsave
import numpy as np

def make_frame_of_images_matrices(matrices, features_columns):
    features_len = len(features_columns)

    # container for collecting all feature vectors
    index_measure_dict = dict()

    for i in range(matrices.shape[0]):
        feature_vector = matrices[i].reshape(features_len)
        index_measure_dict[i] = feature_vector

    return pd.DataFrame.from_dict(index_measure_dict, orient="index", columns=features_columns)

# path of the aero photo image
aero_photo_image = "aerophoto.jpg"

# standard size of the small images
std_size = (50, 50, 3)

# make columns names for dataset
columns = make_columns(std_size)

# cut the big image
small_images = split_the_image_in_boxes(aero_photo_image, std_size)

# make the data frame for testing
test_dataset = make_frame_of_images_matrices(small_images, columns)

# free memory
del small_images

# path to model
model_path = "model_bin/model.pkl"

with open(model_path, "rb") as model_file:
    model = pkl.load(model_file)

# create column inside dataset with predicted results
test_dataset["predicted"] = model.predict(test_dataset)

for index in test_dataset.index:
    # define the name of the image
    path_where_to_save = "aero_photo_predict/{}_{}.jpg".format(index, test_dataset.loc[index, "predicted"])

    # convert to the std size from feature vector size and save
    imsave(path_where_to_save, np.array(test_dataset.loc[index, columns].values, dtype="int").reshape(std_size))
```

make_train_aero_photo.py — файл для створення тестових зображень аерозйомки у лабораторній роботі 9

```
from split_image import split_the_image_in_boxes
from skimage import io
from datetime import datetime as dt

parent_dir = "aero_photo_train"
image_name = "aerophoto.jpg"
std_size = (50, 50, 3)

image_boxes = split_the_image_in_boxes(image_name, std_size)

for small_image in image_boxes:
    path_to_save = "{}_{}.jpg".format(parent_dir, dt.now())
    io.imsave(path_to_save, small_image)
```

making_model_of_aero_photo.py — модуль для навчання та збереження моделі у лабораторній роботі 9

```
from prepare_package.prepare_module import read_dataset
from models_module.IEI import IEIModelAPI
import pickle as pkl

# where small images situated
source_dir = "aero_photo_train"

# properties for extracting dataset
classes = ["field", "road", "town", "water"]
file_pattern = "*{}*.jpg"
standard_shape = (50, 50, 3)

# where save the model
target_dir_for_bin_model = "model_bin"

# get the dataset from files in pandas data frame format
dataset, features = read_dataset(source_dir, classes, file_pattern, standard_shape)

model = IEIModelAPI(dataset, features, "class")

# serialize the model
with open("{} / {}.pkl".format(target_dir_for_bin_model, "model"), "wb") as bin_model_file:
    pkl.dump(model, bin_model_file, protocol=pkl.HIGHEST_PROTOCOL)
```

plotting.py — модуль для виведення на екран графіку реалізацій у 2д за першими двома ознаками розпізнавання у лабораторній роботі 3.

```
from prepare_package.prepare_module import read_dataset
from matplotlib import pyplot as plt

def plot_class(dataset, class_name, style):
    class_data = dataset[dataset["class"] == class_name]
    x = class_data.iloc[:, 0].values
    y = class_data.iloc[:, 1].values
    x_mean = x.mean()
    y_mean = y.mean()
    plt.annotate("{}_center".format(class_name),
                 xy=(x_mean, y_mean), xycoords="data",
                 xytext=(x_mean+20, y_mean+20), textcoords="data",
                 arrowprops=dict(facecolor='b', shrink=0.05, width=2))
    plt.plot([*x, x_mean], [*y, y_mean], style, label=class_name)

source_dir = "images_for_labs"
classes = ["wood", "cloth", "tile", "brick"]
plotting_styles = ["ro", "g^", "b+", "bs"]
file_pattern = "*{}*.jpg"

# get the dataset from files in pandas data frame format
dataset, features = read_dataset(source_dir, classes, file_pattern)

for class_name, style in zip(classes, plotting_styles):
    plot_class(dataset, class_name, style)

plt.legend()
plt.show()
```

split_image.py — модуль для розділення великого зображення на квадрати певної форми у лабораторній 9

```
from skimage import io
from skimage.util.shape import view_as_blocks
import numpy as np

def split_the_image_in_boxes(image_path, standard_size):
    standard_high, standard_width, number_of_colors = standard_size

    # read the image
    all_pixels = io.imread(image_path)

    # calc params for extract the compatibility image
    real_height = all_pixels.shape[0]
    real_width = all_pixels.shape[1]
    compatible_height_border = standard_high * (real_height // standard_high)
    compatible_width_border = standard_width * (real_width // standard_width)

    # split the image and divide it into blocks
    compatible_image = all_pixels[:compatible_height_border, :compatible_width_border, :]
    image_blocks_matrix = view_as_blocks(compatible_image, standard_size)

    # reshape for comfortable processing
    rows, cols = image_blocks_matrix.shape[:2]
    image_blocks_list = np.reshape(image_blocks_matrix, newshape=(rows * cols, *standard_size))

    return image_blocks_list
```