Data Science and Machine Learning Lab
Lab 03 - Build Your Own Classifier

Politecnico di Torino

Intro

In this laboratory, you will build your own version of the K-Nearest Neighbors algorithm (a.k.a. KNN) using the NumPy library.

Note that exercises marked with a (*) are optional; you should focus on completing the other ones first.

1 Preliminary steps

1.1 Datasets

For this lab, you will need two of the datasets you have already met: Iris and MNIST. Please refer to Laboratory 1 for a complete description of the datasets. In addition, a third dataset will be used in the last part of the laboratory to test your KNN implementation.

Iris. You can download it from:

`https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data`

MNIST. You can download it from:

`https://raw.githubusercontent.com/dbdmg/data-science-lab/master/datasets/mnist_test.csv`

Ames Housing. This dataset contains detailed information on residential properties sold in Ames, Iowa. The dataset can be downloaded from:

`https://www.kaggle.com/datasets/shashanknecrothapa/ames-housing-dataset`

Each record represents a house, described through various numerical and categorical features such as lot size, number of rooms, quality ratings, and other property characteristics.

Target variable. In this exercise, we will consider the `House Style` column as the label we want to predict. This feature describes the architectural style of the house:

- `1Story`: single-story house;
- `1.5Fin`: one and a half story with a finished upper level;
- `1.5Unf`: one and a half story with an unfinished upper level;
- `2Story`: two full stories;
- `2.5Fin`: two and a half stories with a finished attic;
- `2.5Unf`: two and a half stories with an unfinished attic;
- `SFoyer`: split foyer house;
- `SLvl`: split level house.

2 KNN design and implementation

As you might remember from Lab. 1, the Iris dataset collects the measurements of different Iris flowers, and each data point is associated with a Iris species (*Setosa*, *Versicolor*, or *Virginica*). In this exercise, you will implement your own version of the the K-Nearest Neighbors algorithm, and you will use it to assign a Iris species (i.e. a label) to flowers whose species is unknown.

The KNN algorithm is straightforward. Suppose that some measurements (i.e. records) and their relative species are known in advance. Then, whenever we want to label a new flower, we look at the K *most similar* points (a.k.a. neighbors) and assign a label accordingly. The simplest solution is using a majority voting scheme: if the majority of the neighbors *votes* for a label, we will go for it. This approach is naive only at first sight: the local similarity assumed by KNN happens to be roughly true. Even though this reasoning does not generalize well[1], the KNN provides a valid baseline for your tasks.

1. Load the Iris dataset.

> **Info:** You can use the `pandas.read_csv` method to easily parse and store the dataset into a pandas DataFrame. It applies for both locally stored data and remote files:
>
> ```
> df = pd.read_csv(
>     "https://archive.ics.uci.edu/ml/machine-learning-databases/
>    iris/iris.data",
>     header=None
> )
> ```

2. Let's identify a portion of our data for which we will try to guess the species. Randomly select 20% of the records and store the first four columns (i.e. the features representing each flower) into a two-dimensional numpy array of shape $N \times C$, you can call it `X_test`.
   For the same records, store the last column (i.e. the one with the species values) into another array, namely `y_test`. This is the data that will be used to test the accuracy of your KNN implementation and its correct functioning (i.e. the testing data).

> **Info:** Each pandas DataFrame stores its data as a multi-dimensional numpy array internally. You can access it through the DataFrame property `values`.

3. Store the remaining 80% of the records in the same way. In this case, use the names `X_train` and `y_train` for the arrays. This is the data that your model will use as ground-truth knowledge (i.e. the training data).

4. Focus now on the KNN technique. You can find a thorough explanation of the algorithm on the course slides on classification techniques (slides [60,69]).

   Starting from the next laboratory, you will use the `scikit-learn` package. Many of its functionalities are exposed via an object-oriented interface. With this paradigm in mind, implement now the KNN algorithm and expose it as a Python class. The bare skeleton of your class should look like this (you are free to add as many methods as you want):

```
class KNearestNeighbors:
    def __init__(self, k, distance_metric="euclidean"):
        self.k = k
        self.distance_metric = distance_metric

    def fit(self, X, y):
        """
        Store the 'prior knowledge' of you model that will be used
```

[1] Concepts like generalization capabilities and model capacity will be studied later in the course. Also, you will learn and use more complex estimators and classifiers that overcome KNN limitations.

```
        to predict new labels.

        :param X : input data points, ndarray, shape = (R,C).
        :param y : input labels, ndarray, shape = (R,).
        """
        pass  # TODO: implement it!

    def predict(self, X):
        """Run the KNN classification on X.

        :param X: input data points, ndarray, shape = (N,C).
        :return: labels : ndarray, shape = (N,).
        """
        pass   # TODO: implement it!
```

Implement the `fit` method first. Here, you should only keep track of the main attributes that will be used by the algorithm.

- In this version of the algorithm, does the KNN need to store all the samples of `X_train` and `y_train`?

5. To identify the K closest points, or neighbors, a notion of distance is required. Your implementation must support three different distance definitions. Given two n-dimensional points $p = (p_1, p_2, \ldots, p_n)$ and $q = (q_1, q_2, \ldots, q_n)$, the euclidean distance is defined as

$$ed(p, q) = \sqrt{\sum_{i=1}^{n} (p_i - q_i)^2} \tag{1}$$

The second distance function is the cosine distance, which is defined as:

$$cd(p, q) = 1 - |cs(p, q)| \tag{2}$$

where

$$cs(p, q) = \frac{\sum_{i=1}^{n} p_i q_i}{\sqrt{\sum_{i=1}^{n} p_i^2} \cdot \sqrt{\sum_{i=1}^{n} q_i^2}} \tag{3}$$

is known as cosine similarity.

> **Info:** The math lovers among you have probably noted that $cs(p, q)$ is the cosine of the angle between p and q, hence the name. When the two vectors are orthogonal, the cosine is equal to 0 and distance in Equation 2 is equal to 1. Otherwise, if the two vectors are parallel (read here *similar*), the distance is 0.

The third distance is the Manhattan distance. The distance is defined as:

$$md(p, q) = \sum_{i=1}^{n} |p_i - q_i| \tag{4}$$

Write three functions that receive two numpy arrays and use NumPy to compute the respective distance. As you might have noticed from the class constructor, we will use the euclidean distance by default.

6. Implement the `predict` method. The function receives as input a numpy array with N rows and C columns, corresponding to N flowers. The method assigns one of the three Iris species to each row using the KNN algorithm, and returns them as a numpy array. For the actual implementation, identify K neighbors using the distance specified by the parameters `k` and `distance` passed to the constructor.

Then, assign the label using a majority voting scheme[2].

[2] If K is even, assign the label arbitrarily

7. Now let's `fit` the KNN model with the `X_train` and `y_train` data. Then, try to use your KNN model to `predict` the species for each record in `X_test` and store them in a nupy array called `y_pred`.

   Check how many Iris species in the array `y_pred` have been guessed correctly with respect to the ones in `y_test`. A prediction is correct if `y_pred[i] == y_test[i]`. The ratio between the number of correct guesses and the total number of guesses is known as accuracy. If all labels are assigned correctly (`(y_pred == y_test).all() == True`), the accuracy of the model is 100%. Instead, if none of the guessed species corresponds to the real one (`(y_pred == y_test).any() == False`), the accuracy is 0%.

8. (*) As a software developer, you might want to increase the functionalities of your product and publish newer versions over time. The better your code is structured and organized, the lower the effort to release updates.

   As such, extend now your KNN implementation by adding the parameter *weights* to the constructor, as shown below:

```
class KNearestNeighbors:
    def __init__(self, k, distance_metric="euclidean", weights="
   uniform"):
        self.k = k
        self.distance_metric = distance_metric
        self.weights = weights
```

   Change your KNN implementation to accept a new weighting scheme for the labels. If `weights="distance"`, weight neighbor votes by the inverse of their distance (for the distance, again, use `distance_metric`). The weight for a neighbor of the point p is:

$$w(p, n) = \frac{1}{\textit{distance\_metric(p, n)}} \tag{5}$$

   Instead, if the default is chosen (`weights="uniform"`), use the majority voting you already implemented in Exercise 6.

9. (*) Test the modularity of the implementation applying it on a different dataset. Ideally, you should not change the code of your KNN python class.

   - Download the MNIST dataset and sample only 100 points per digit. You will end up with a dataset of 1000 samples.
   - Define again four numpy arrays as you did in Exercises 2 and 3.
   - Apply your KNN as you did for the Iris dataset.
   - Evaluate the accuracy on MNIST's `y_test`.

10. (*) The choice of parameters like `k`, `distance_metric` or `weights` is part of your data analysis pipeline: the process is typically known as validation. You will learn more about algorithm validation in theory lectures.

    For now, evaluate the accuracy scores achieved by your KNN with different parameter values. Then, try to identify which is the best configuration, for both Iris and MNIST.

11. Load the **Ames Housing** dataset, select only the numerical features, and define the target variable as `House Style`. Replace missing values with the mean of each column. Then, split the data into train and test sets.

    Info: For the selection of the numerical features, you can use `pandas.select_dtypes`

```
df = pd.read_csv("AmesHousing.csv")
df = df.select_dtypes(include=np.number)
```

12. Train your KNN classifier on the dataset and evaluate its accuracy. Then, compare your result with a naive baseline where all houses are predicted as belonging to the *majority class* (i.e., the most frequent "*House Style*" in the dataset).

13. Before re-training, standardize the dataset to remove scale effects between features. Use the normalization formula:

$$X' = \frac{X - \mu}{\sigma} \tag{6}$$

where μ and σ are the mean and standard deviation computed on each feature of the training set. Train the KNN again and compare the accuracy. Is it improved? Why?

14. Repeat the experiment for different values of k and evaluate how the parameter k affects model performance.

15. (*) For each value of k, repeat the random split between training and testing data multiple times (e.g., 20 runs). Compute the **mean** and **standard deviation** of the accuracy across runs to evaluate the model's stability.

16. Rerun the KNN experiments using the different distance definitions implemented (euclidean, cosine, and manhattan distances). For each distance metric, vary k and record the achieved accuracy. Identify which combination of distance metric and k achieves the best overall performance.