Machine Learning Methods

Exercise 2

Last updated: 18.1.24

1 General Description

In this exercise, you will explore the practical application of two fundamental machine learning concepts: Distance-Based Methods and Decision Trees. We will use a dataset of 2D map locations of cities in the USA, which consists of 30K data samples. Using different methods, you will examine how cities are classified into states and identify anomalous cities.

You are expected to hand in a report, no longer than 8 pages, describing your experiments and answers to the questions in the exercise. The submission guidelines are described in Sec. 7, please read them carefully.

Note 1: You **must** explain your results in each question! Unless specified otherwise, an answer without explanation will **not** be awarded marks.
Note 2: When you are asked to plot something, we expect that plot to appear in the report.

2 Seeding

As in the previous exercise, we would like to reproduce your experiments. This can be done by random seeding. Specifically, all that it takes is to put the following line of code at the beginning of your main file: *np.random.seed(0)* (where *np* stands for the *numpy* library).

3 Provided Code Helpers

You are free to choose how to implement this exercise. We do provide you with some helpful helper functions. These helpers include visualizations, data processing and demos for working with specific models. They can be found in `helpers.py`. These functions are meant to help you and we recommend you to use them. Saying that, they are optional, meaning **if you find them confusing you may simply ignore them**.

We also provide you with a skeleton code for the kNN clasifier, found in `knn.py`. Unlike the prevoius helpers, this skeleton is mandatory and you must fill-in the blanks for Sec. 5.

4 Data

The dataset used for our exploration consists of three distinct files: *train.csv, validation.csv, test.csv*. Each file is structured as a table, with three columns: longitude (`long`), latitude (`lat`) and `state`. The samples in our dataset are represented as 2D coordinates of cities in the USA, with the corresponding labels indicating the state. For instance, if we consider a 2D city coordinate such as $(41.8375, -87.6866)$ – representative of Chicago – its label would be *'IL'* (short for *'Illinois'*). To make it easier, we already encoded the `state` column into integers.

The samples in our dataset can be denoted as $\mathcal{X} \in \mathbb{R}^{N \times d}$, where N represents the number of samples, and d is the representation dimension. In our specific case, $d = 2$ as we are dealing with longitude and latitude coordinates.

We will use this spatial dataset for exploring distance-based methods and decision trees, by classifying cities into states based on their geographical coordinates.

5 Classification with k-Nearest Neighbors (kNN)

kNN embodies the principle that similar instances tend to belong to the same class. However, the choice of the hyperparameter k, representing the number of neighbors considered, can significantly influence classification outcomes and requires careful consideration.

To facilitate your implementation of supervised kNN classification, we offer a pseudo-code template in Algorithm. 1. This pseudo-code outlines the key steps involved in the kNN algorithm, making it a valuable reference for your implementation.

Moreover, we recommend exploring the `faiss` library, which offers an efficient kNN implementation. The library is particularly useful when dealing with large datasets or scenarios where computational efficiency is critical. Additionally, you can find an example of how to extract kNN distances for a given test sample in the provided code helper. To install `faiss` make sure you already have the `numpy` library installed, then simply run:
`pip install faiss-cpu`.

5.1 Task

Implement the `KNNClassifier` class given in the `knn.py` file. The required inputs and outputs are given in the code. The kNN search should be done using `faiss` (in a similar way to the code you saw in class).

Algorithm 1: kNN Classification

Data: Training dataset $\mathcal{X}_{train} = \{(x_i, y_i)\}_{i=1}^{N}$, where $x_i \in \mathbb{R}^2$ is the feature vector and y_i corresponds to its class label; Test instance $\hat{x}_{\text{test}}$; Number of neighbors k.
Result: Predicted class label $\hat{y}_{\text{test}}$ for the test instance.
for *each training instance* (x_i, y_i) **do**
  Calculate the distance d_i between x_i and x_{test}.
Sort the distances d_i in ascending order and select the first k instances.
for *each class* c **do**
  Count the occurrences of c in the selected k instances.
Assign the class label $\hat{y}_{\text{test}}$ to the one with the highest count.
return $\hat{y}_{test}$

1. Your code should allow for two distance metrics, L_1 and L_2, it should also allow the user to specify the number of neighbors k.
2. Use *train.csv* as training data *test.csv* as the testing data.
3. Now, try every combination for $k \in \{1, 10, 100, 1000, 3000\}$ and distance metrics in $\{L_1, L_2\}$ (feel free to use nested loops). Present the results in a table of dimensions 5×2, containing the test accuracy scores for each combination of k and distance metric. Report this table in your PDF.

You should save all 10 kNN models and accuracies. For each model you should also save its hyper-parameters (distance metric, k) and its test accuracy. We will later choose between the models, according to these values.

5.2 Questions

1. Look at the 5×2 table of results from task 3. What is the trend you see when the number of k increases? Does it changes between different distance metrics?
2. We will now visualize the differences between k values and distance metrics:
   - Choose the k value with the highest test accuracy when using the L_2 distance metric. We will call it k_{max}.
   - Choose the k value with the lowest test accuracy when using the L_2 distance metric. We will call it k_{min}.
   - Using the given visualization helper (`plot_decision_boundaries`), plot the test data and color the space according to the prediction of the following 3 models, each in a separate plot (overall 3 plots): (i) distance_metric $= L_2$, $k = k_{max}$. (ii) distance_metric $= L_2$, $k = k_{min}$. (iii) distance_metric $= L_1$, $k = k_{max}$.

NOTE: Running `plot_decision_boundaries` may take up to a few minutes when using high k values (e.g. 1000, 3000). If it takes substantially more (15 minutes or more) your `predict` implementation is probably not efficient enough.

(a) Look at the plots of the (i) k_{max} with L_2 and (ii) k_{min} with L_2. What is different between the way each one divides the space? Why does k_{max} results in better accuracy? Explain.

(b) Look at the plots of the k_{max} with L_2 distance metric and k_{max} with L_1 distance metric. How does the choice of distance metric affect the classification space? Explain.

5.3 Anomaly Detection Using kNN

In this phase, our exploration turns towards anomaly detection, utilizing the kNN algorithm to identify anomalies. For this specific task, we deviate from supervised classification. Instead, we treat the training set as a single class, ignoring individual class labels. The primary objective is to identify anomalies within the test set by calculating kNN distances.
More specifically:

- You were given an additional test file specifically for this task, named: *AD_test.csv*. Your train set remains unchanged.
- Find the 5 nearest neighbors from the train set for each test sample of *AD_test.csv* using `faiss`. Use the L_2 distance metric. Save the distances to the neighbors as well.
- Sum the 5 distances to the nearest neighbors for each test sample. We will refer to these summations as the anomaly scores.
- Find the 50 test examples with the highest anomaly scores. We will define these points as anomalies, while the rest of the points will be defined as normal.
- Using the `matplotlib` library, plot the points of *AD_test.csv*. According to your prediction, color the normal points in blue, and the anomalous points in red. Additionally, in the same plot, include the data points from *train.csv* colored in black and with an opacity of 0.01 (the parameter controlling the opacity is named `alpha`). You might find the `plt.scatter` function useful for this visualization.

5.4 Questions

1. What can you tell about the anomalies your model found? How are they different from the normal data? Explain.

1 A Greedy Boosting Implementation

This part of the exercise is for you to experiment with an implementation of a boosting strategy, which is described below:

You will first create a multi-class decision stump. In this case, the decision stump predicts over k classes. To handle this, the space will be divided into two regions. For one side of the split, the decision stump will predict the label of class c, represented as $[0, \ldots, 0, 1, 0, \ldots, 0]$, where the one is at index c. For the other side, it will predict the label $[1/k, \ldots, 1/k]$ (a uniform distribution over all classes). To find the best index that splits the space most effectively, you will iterate over all possible labels and choose the one that yields the best accuracy.

How does the ensemble come into play? For each sample in your dataset, you will maintain a `score` array of size `num_classes`, initialized as an array of zeros. This `score` array will be updated incrementally as more stumps are added (see the pseudocode). The predicted label of a sample is determined as the `argmax` of its corresponding `score`. Once a stump iterates over all possible features, thresholds, and classes, it selects the configuration that best divides the space and updates the `score` array as described below.

Algorithm 1 Decision Stump for Multi-Class Greedy Ensemble

Require: $X \in R^{n \times m}$: Feature matrix with n samples and m features
Require: $y \in R^n$: Ground truth labels for n samples
Require: $score \in R^{n \times k}$: Score array for each label, initialized to zero
Initialize $best\_accuracy \leftarrow 0$
for $d \leftarrow 1$ to m **do** ▷ Iterate over all feature dimensions
  $thresholds \leftarrow X[:, d]$ ▷ All possible thresholds for dimension d
  for $\theta \in thresholds$ **do** ▷ Iterate over sampled thresholds
    for $c \leftarrow 1$ to k **do** ▷ Iterate over all possible classes
      for $s \in \{left, right\}$ **do** ▷ Iterate over sides of the split
        $samples\_to\_update \leftarrow$ samples selected based on threshold θ and side s
        $current\_score\_diff \leftarrow \mathbf{0} \in R^{n \times k}$
        $current\_score\_diff[samples\_to\_update] \leftarrow one\_hot(c)$
        $updated\_score \leftarrow score + current\_score\_diff$
        $predictions \leftarrow \text{argmax}(updated\_score, \text{axis} = 1)$ ▷ Predicted class per sample
        Compute $current\_accuracy$ by comparing $predictions$ and y
        if $current\_accuracy > best\_accuracy$ **then**
          $best\_accuracy \leftarrow current\_accuracy$
          $selected\_score \leftarrow updated\_score$
        end if
      end for
    end for
  end for
end for
return $selected\_score$

—

1.1 Task

Implement a decision stump that takes as input the `score` array (this will be either the output of the previous decision stump or an array of zeros if this is the first stump) and the data X and y.

Inference: For inference, each stump needs to store the following: - The dimension along which it splits the data. - The threshold value for the split. - The direction of the split. - The label it assigns to the subspace.

When predicting the label for a new sample, each stump assigns a class label based on the sample's position relative to the split. The final prediction is determined by a majority vote among all the stumps.

You are free to design the implementation as you see fit!

—

1.2 Implementation Notes

- Split the data into 90% for training and 10% for testing.
- Implement this boosting strategy with 25 stumps.
- Iterating over all possible thresholds for `thresholds` can be slow. Instead, sample a subset of possible thresholds (at least 20) to reduce computational overhead.
- To test your implementation, we **strongly recommend** creating a small synthetic dataset and verifying that your algorithm behaves as expected.
- Ensure the concept is clear before writing code.

—

1.3 Questions

1. Present a graph of the accuracy as a function of the number of decision stumps, showing results for both the training set and the test set on the same plot.
2. Create a scatter plot of the training data, with the color of each sample corresponding to its predicted class for iterations $[1, 5, 15, 25]$.
3. The final accuracy of your method is likely far from perfect. Explain why this is the case. If you could design a more sophisticated weak learner that splits the space differently, could you achieve 100% accuracy? Explain.
4. Compare your results with those of KNN. Which method performs better, and why? Provide an intuitive explanation rather than a numerical one.

1.4 Experimenting with XGBoost (Bonus 5 pts)

- **A Word on Boosting.** While random forests are useful and powerful models, it is often more beneficial to use boosting methods, specifically gradient boosting methods such as XGBoost [**?**]. Boosting works by fitting a series of models sequentially, where each model attempts to correct the mistakes of the previous one. The process begins with a base model, and subsequent models are trained to predict the errors (residuals) of the prior model. This iterative process continues, progressively adding "corrective" models, until a stopping criterion is met. For the purpose of this exercise, this intuition is sufficient.
- **Operating XGBoost in Code.** We will use XGBoost via the official XGBoost library. Its API is similar to the scikit-learn models used for the decision trees. To implement XGBoost, use the `XGBClassifier` class.

- **Question.** Train an XGBoost model with a maximum tree depth of 1 (`max_depth=1`) and a learning rate of 0.1 (`learning_rate=0.1`). Evaluate and plot the test accuracy as a function of (`n_estimators`). For each value of `n_estimators`, you will need to refit the model. XGBoost's efficient implementation ensures this process is fast.
  After training, visualize the predictions made by XGBoost. Compare the predictions of XGBoost with those made by the boosting method you've implemented in the previous question after training with the same number of stumps as in the previous question:

  – How do the predictions of XGBoost differ from those of the boosting you implemented?

6 Ethics

We will not tolerate any form of cheating or code sharing. You may only use the `numpy`, `pandas`, `faiss`, `mathplotlib`, `sklearn` and `xgboost` libraries.

7 Submission Guidelines

You should submit your report, code and README for the exercise as $ex2\_\{YOUR\_ID\}.zip$ file. **Other formats (e.g. *.rar*) will not be accepted!**.

The README file should include your name, cse username and ID.

Reports should be in PDF format and be no longer than 6 pages in length. They should include any analysis and questions that were raised during the exercise. Please answer the questions in Sec. 5, 5.3& **??** in sequential order. **You should submit yourr code (including your modified helpers file), Report PDF and README files alone without any additional files.**

7.1 ChatGPT / Github Copilot

For this exercise specifically, we advise you to avoid ChatGPT / Github Copilot, as one of the purposes is to get familiar with **numpy**. Saying that, we do not prohibit using them. If you do choose to use them, write at the end of your report a paragraph detailing for which parts /functionalities you used them.

7.2 Submission

Please submit a zip file named $ex2\_\{YOURID\}.zip$ that includes the following:

1. A README file with your name, cse username and ID.
2. The *.py* files with your filled code.
3. A PDF report.