CITS5508 Machine Learning
Semester 1, 2024
Lab sheet 3
(Not assessed)

You will develop Python code to continue the classification tasks in the previous lab sheet. Certify that the presentation of your Python notebook is good and that you used the Markdown cells well. Make sure you properly format your plots and results. For instance, all your diagrams/plots should have proper axis labels and titles to help the reader understand what you are plotting. Another example is the confusion matrix; not showing the class names makes the confusion matrix completely useless. Use the lab sheets to learn how to improve the presentation of your notebook, as you will need this in the assessments.

Forest type mapping

In this part, we will use the *training.csv* and *testing.csv* data files supplied on LMS. This is the same data used in lab sheet 2; you should look there for more details about the data. This lab sheet builds on your work in the previous lab. Therefore, it assumes you completed the tasks of the lab sheet 2 and will continue to work on the same Jupyter notebook.

Tasks

Your tasks for this lab sheet are described below. To perform these tasks, use all features (b1-b9). We will continue to use Logistic Regression and the binary classification on this dataset using examples from two classes: 's' ('Sugi' forest) and 'd' ('Mixed deciduous' forest).

1. Split your training data into two sets: training and validation. Select randomly 80% for training and 20% for validation. You must set the random generator seed to 42 before the splitting. Ensure your validation set is fixed and does not change each time you run your algorithm.

2. Use `sklearn.linear_model.SGDClassifier` together with `sklearn.model_selection.GridSearchCV` to find an optimal value of the regularisation hyperparameter C and the learning rate η in the training set using Grid Search and a 10-fold cross-validation. Be careful how `scikit-learn` defines the Grid Search.

3. Show the optimal values for C and η according to Grid Search.

4. Train your model on the entire training set using these optimal values. Using the validation set, plot precision versus recall and comments on the results. How does the performance measure behave? What threshold would you choose and why?

5. Given now these optimal three values (optimal values for C and η and the threshold you selected), inspect some performance indicators on the test. Show the results for:

   (a) The accuracy values for the training set and the test set.

   (b) The confusion matrix on the training and testing set.

   (c) The plot of precision versus recall for the training set.

   What do you see from inspecting these performance indicators? How do they compare with your previous lab sheet results, where we didn't use cross-validation and Grid Search?

Optional, but may be covered in assessments.

6. Examine the estimated probabilities and decision boundary of the Logistic Regression model.

   Consider all features. Train your model as before (using Grid Search to find the optimal values for C and η). Plot the estimated probabilities and decision boundary. To build your plot, you must use the score value of the linear part from the logistic regression model in the x-axis. Then, choose randomly 10 instances of the testing set, add them to your plot, and verify if you have made a right or wrong decision (how would you classify the test instances) regarding the decision boundary of 50%. Comment about your results.

Based on your plots, comment about:

- What threshold would you choose for classification based on the predicted probabilities?
- What is the impact of changing the threshold for performance indicators such as precision and recall?
- What can you say about the overlap between classes, and how does this impact classification performance?

Implementing the *k-nearest neighbours* (k-NN) algorithm to do regression

You should create a new Jupyter notebook for this part of the lab sheet.

We will learn how to use the *k-nearest neighbours* (k-NN) algorithm to do regression. This is an *instance-based learning* method (see Figure 1-15 in the textbook for an example of k-NN classification). Suppose that the features $\mathbf{X}_i$, for $i = 1, \cdots, n$, of our training data are n-dimensional vectors and each feature vector has an associated y_i scalar value. Our objective is to predict the scalar y_{test} value for a given test instance $\mathbf{X}_{\text{test}} \in \mathbb{R}^n$. The way how the k-NN regression works can be summarized as follows:

1. finds the k nearest neighbours of the test instance $\mathbf{X}_{\text{test}}$ in the feature space ($\mathbb{R}^n$) . This step gives a set of k neighbours $\{\mathbf{X}_{(1)}, \mathbf{X}_{(2)}, \cdots, \mathbf{X}_{(k)}\}$, where k is a positive integer supplied by the user. From the training data, the associated scalar values $\{y_{(1)}, y_{(2)}, \cdots, y_{(k)}\}$ of these neighbours can be extracted.
2. computes y_{test} as the weighted sum of the y values of these neighbours, i.e., $y_{\text{test}} = \sum_{i=1}^{k} w_i y_{(i)}$.

In Step 1, we need to use a distance function that defines the *nearness* of points in the feature space. Distance functions that are commonly used include: Manhattan distance (ℓ_1 norm), Euclidean distance (ℓ_2 norm), and Minkowski distance (ℓ_p norm, for some integer $p > 2$).

In Step 2, the value of each weight w_i needs to be defined. The simplest formula is to set $w_i = 1/k, \forall i$, so y_{test} is just the simple average of $\{y_{(1)}, \cdots, y_{(k)}\}$. A more complex formula is to set w_i as the inverse of the distance or squared distance of each neighbour $\mathbf{X}_{(i)}$ to the test instance $\mathbf{X}_{\text{test}}$, i.e., let

$$w_i = \frac{1}{d(\mathbf{X}_{(i)}, \mathbf{X}_{\text{test}}) + \epsilon} \text{ or } w_i = \frac{1}{d^2(\mathbf{X}_{(i)}, \mathbf{X}_{\text{test}}) + \epsilon}$$

where $d(\cdot, \cdot)$ can be the Manhattan distance, Euclidean distance, Minkowski distance, or any user-defined distance between the two input items. The term ϵ in the denominator is a small constant to avoid the division-by-zero problem when $\mathbf{X}_{(i)}$ and $\mathbf{X}_{\text{test}}$ happen to be the same point and $d(\mathbf{X}_{(i)}, \mathbf{X}_{\text{test}})$ becomes

zero. These more complex formulae allow neighbouring points that are closer to the test instance $\mathbf{X}_{\text{test}}$ to have larger weights. If the test instance $\mathbf{X}_{\text{test}}$ happens to coincide with one of the neighbours, then the weight for that neighbour would be 1 and the weights for all other neighbours would be 0.

After appropriately defining w_i, it is important to normalize all the weight values as follows:

$$w_i \leftarrow \frac{w_i}{\sum_{j=1}^{k} w_j}$$

so that they sum to 1.

The Scikit-Learn library includes the class `KNeighborsRegressor` that performs k-nearest neighbours regression. See

https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsRegressor.html

The bottleneck of k-NN is in Step 1 – how to quickly find the k nearest neighbours. This is a computationally expensive process especially when the dimension n of the features is large and when the training set size is large also (having more training data is actually good for the regression task so we should not complain that training set is too large). The class `KNeighborsRegressor` includes the implementation of various algorithms (e.g., building a *ball-tree* or a *kd-tree* data structure) to speed up the neighbour search step.

We will apply the k-NN algorithm to the California Housing Prices dataset described in Chapter 2 to predict the house prices. You can use the parameter `n_neighbors` to set your desired k value. Your tasks for this part of the lab sheet are:

1. Randomly split the data into a training set (say 80%) and a test set (say 20%), apply the k-NN regressor, and evaluate how good the regressor performs by computing the *root mean square error* (RMSE) of the predicted house prices of the test set.

2. Experiment with a selection of columns (avoid those columns that contain text) from the dataset to form your feature vectors. The number of columns that you choose would become the dimension n of your feature vectors. Note that you would need to do some normalization so that your data is not dominated by some columns that have large magnitudes.

3. Try also setting the parameter `weights` to '`uniform`' (this is equivalent to setting $w_i = 1/k, \forall i$) and to '`distance`' (this is equivalent to setting $w_i = 1/(d(\mathbf{X}_{(i)}, \mathbf{X}_{\text{test}}) + \epsilon)$) and then compare the *root mean squared errors* (RMSEs) of their predictions.

Let $\{\mathbf{X}_1, \mathbf{X}_2, \cdots, \mathbf{X}_n\}$ and $\{y_1, y_2, \cdots, y_n\}$ be the ground-truth instances and their corresponding ground truth house price values in the test set. Let $\{\hat{y}_1, \hat{y}_2, \cdots, \hat{y}_n\}$ be the predicted house price values for these instances. Then the RMSE is given by:

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n} (\hat{y}_i - y_i)^2}.$$