Machine Learning homework 1 solution

k-Nearest Neighbors and Decision Trees

Wiktor Jurasz - M.Nr. 03709419

October 28, 2018

1 Introduction

I have implemented both algorithms in Python3.

This pdf contains the source code and solutions to all given problems.

The code can also be found here https://github.com/WiktorJ/machin-learning/tree/master/hw01

I have done the assignments by my own using mostly *"Machine Learning: A Probabilistic Perspective" by Murphy* and other internet resources.

2 Decision Trees

2.1 Problem 1

2.1.1 Solution in Python3

```python
import numpy as np

from node import Node


def get_thresholds(data):
    """
    :param data: 2d numeric array
    :return: A 2d array with with columns containing unique values from corresponding columns from `data`
    """
    return [sorted(set(data[:, c])) for c in range(0, data.shape[1])]


def get_predictions(data, labels):
    """
    Calculates the "Likelihood" for each label to occur
    :param data: data for which predictions are calculated
    :param labels: set of labels
    :return: dict {label_id: "Likelihood"}
    """
    return {int(l): list(data[:, -1]).count(l) / len(data[:, -1]) for l in labels}


def get_gini(predictions):
    """
    Calculates gini index based on predictions
```

```
    :param predictions:
    :return: gini index
    """
    from functools import reduce
    return reduce(lambda acc, val: acc - val ** 2, predictions.values(), 1)


def split(data, thresholds):
    """
    Tries to split data at every point in `thresholds`. Returns split with the lowest gini index
    :param data: data to split
    :param thresholds: possible points to split
    :return: split data, value and its column on which data was split
    """
    left_data = None
    right_data = None
    min_cost = 1
    column = -1
    pivot = None
    for i, feature in enumerate(thresholds[:-1]):
        for val in feature:
            data_l = np.empty((0, data.shape[1]))
            data_r = np.empty((0, data.shape[1]))
            for row in data:
                if row[i] <= val:
                    data_l = np.vstack((data_l, row))
                else:
                    data_r = np.vstack((data_r, row))

            gini_left = get_gini(get_predictions(data_l, thresholds[-1])) if data_l.shape[0] > 0 else 0
            gini_right = get_gini(get_predictions(data_r, thresholds[-1])) if data_r.shape[0] > 0 else 0
            cost = gini_left * (data_l.shape[0] / data.shape[0]) + gini_right * (data_r.shape[0] / data.shape
                [0])
            if min_cost is None or cost < min_cost:
                left_data = data_l
                right_data = data_r
                column = i
                pivot = val
                min_cost = cost

    return left_data, right_data, column, pivot


def fit_tree(data, depth):
    """
    Recursively creates decision tree
    :param data: data at current node
    :param depth: current depth
    :return: root of (sub)tree
    """
    root = Node()
    thresholds = get_thresholds(data)
    root.prediction = get_predictions(data, thresholds[-1])
    root.misclassification_rate = get_gini(root.prediction)
    data_left, data_right, root.column, root.pivot = split(data, thresholds)
    if depth >= 2 or len(root.prediction) == 1:
        return root
    root.left = fit_tree(data_left, depth + 1)
    root.right = fit_tree(data_right, depth + 1)
```

```
    return root


def print_tree(root, indentation):
    """
    :param root: Root of the tree
    :param indentation: Each level in tree adds indentation string for clearer output
    """
    if root is not None:
        prediction = {el: round(root.prediction[el], 3) for el in root.prediction}
        print_tree(root.right, "    {}->".format(indentation))
        print("{} \\textcolor{{blue}}{{ Distribution: ({}), gini index: {} }} {}"
              .format(indentation,
                      prediction,
                      round(root.misclassification_rate, 3),
                      "column: {}, pivot: {}".format(root.column, root.pivot) if not root.is_leaf() else "
                          LEAF"))
        print_tree(root.left, "    {}->".format(indentation))


def get_class(prediction):
    """

    :param prediction: predictions for each label
    :return: the most likely label and its prediction
    """
    label = max(prediction, key=prediction.get)
    return label, round(prediction[label],3)


def classify(root, vector):
    """
    Recursively travers a tree to find label of `vector`
    :param root: root of (sub)tree
    :param vector: vector to classify
    :return: label
    """
    if root.left is None and root.right is None:
        return get_class(root.prediction)
    pivot = vector[root.column]
    if pivot <= root.pivot:
        return classify(root.left, vector)
    else:
        return classify(root.right, vector)


data = np.loadtxt("data/01_homework_dataset.csv", delimiter=",", skiprows=1)
tree_root = fit_tree(data, 0)
print_tree(tree_root, "")

x_a = [4.1, -0.1, 2.2]
x_b = [6.1, 0.4, 1.3]
print("x_a class: {}".format(classify(tree_root, x_a)))
print("x_b class: {}".format(classify(tree_root, x_b)))
```

2.1.2 Helper DTO class

```
class Node:
    def __init__(self) -> None:
        self.left = None
        self.right = None
        self.prediction = {}
        self.misclassification_rate = None
        self.cost = 1
        self.column = None
        self.pivot = None

    def is_leaf(self):
        return self.left is None and self.right is None
```

2.1.3 Result tree

After running above code with {**X**, **y**} data, following tree is created:

```
        ->->  Distribution: (0: 1.0), gini index: 0.0  LEAF
    ->  Distribution: (0: 0.556, 2: 0.444), gini index: 0.494  column: 0, pivot: 6.9
        ->->  Distribution: (0: 0.333, 2: 0.667), gini index: 0.444  LEAF
Distribution: (0: 0.333, 1: 0.4, 2: 0.267), gini index: 0.658  column: 0, pivot: 4.1
    ->  Distribution: (1: 1.0), gini index: 0.0  LEAF
```

2.2 Problem 2

Tree predicted:
For vector $\vec{x_a} = \begin{bmatrix}4.1 & -0.1 & 2.2\end{bmatrix}^T$ class **1** with probability **1.0**
For vector $\vec{x_b} = \begin{bmatrix}6.1 & 0.4 & 1.3\end{bmatrix}^T$ class **2** with probability **0.67**

3 k-Nearest Neighbors

3.1 Programming assignment 1: k-Nearest Neighbors classification

```
In [1]: import numpy as np
        from sklearn import datasets, model_selection
        import matplotlib.pyplot as plt
        %matplotlib inline
```

3.1.1 Introduction

For those of you new to Python, there are lots of tutorials online, just pick whichever you like best :)

If you never worked with Numpy or Jupyter before, you can check out these guides * https://docs.scipy.org/doc/numpy-dev/user/quickstart.html * http://jupyter.readthedocs.io/en/latest/

3.1.2 Your task

In this notebook code to perform k-NN classification is provided. However, some functions are incomplete. Your task is to fill in the missing code and run the entire notebook.

In the beginning of every function there is docstring, which specifies the format of input and output. Write your code in a way that adheres to it. You may only use plain python and `numpy` functions (i.e. no scikit-learn classifiers).

3.1.3 Exporting the results to PDF

Once you complete the assignments, export the entire notebook as PDF and attach it to your homework solutions. The best way of doing that is 1. Run all the cells of the notebook. 2. Download the notebook in HTML (click File > Download as > .html) 3. Convert the HTML to PDF using e.g. https://www.sejda.com/html-to-pdf or `wkhtmltopdf` for Linux (tutorial) 4. Concatenate your solutions for other tasks with the output of Step 3. On a Linux machine you can simply use `pdfunite`, there are similar tools for other platforms too. You can only upload a single PDF file to Moodle.

This way is preferred to using `nbconvert`, since `nbconvert` clips lines that exceed page width and makes your code harder to grade.

3.1.4 Load dataset

The iris data set (https://en.wikipedia.org/wiki/Iris_flower_data_set) is loaded and split into train and test parts by the function `load_dataset`.

```
In [2]: def load_dataset(split):
            """Load and split the dataset into training and test parts.

            Parameters
            ----------
            split : float in range (0, 1)
                Fraction of the data used for training.

            Returns
            -------
            X_train : array, shape (N_train, 4)
                Training features.
            y_train : array, shape (N_train)
                Training labels.
            X_test : array, shape (N_test, 4)
                Test features.
```

```
            y_test : array, shape (N_test)
                Test labels.
            """
            dataset = datasets.load_iris()
            X, y = dataset['data'], dataset['target']
            X_train, X_test, y_train, y_test =\
                model_selection\
                    .train_test_split(X, y, random_state=123, test_size=(1 - split))
            return X_train, X_test, y_train, y_test
```

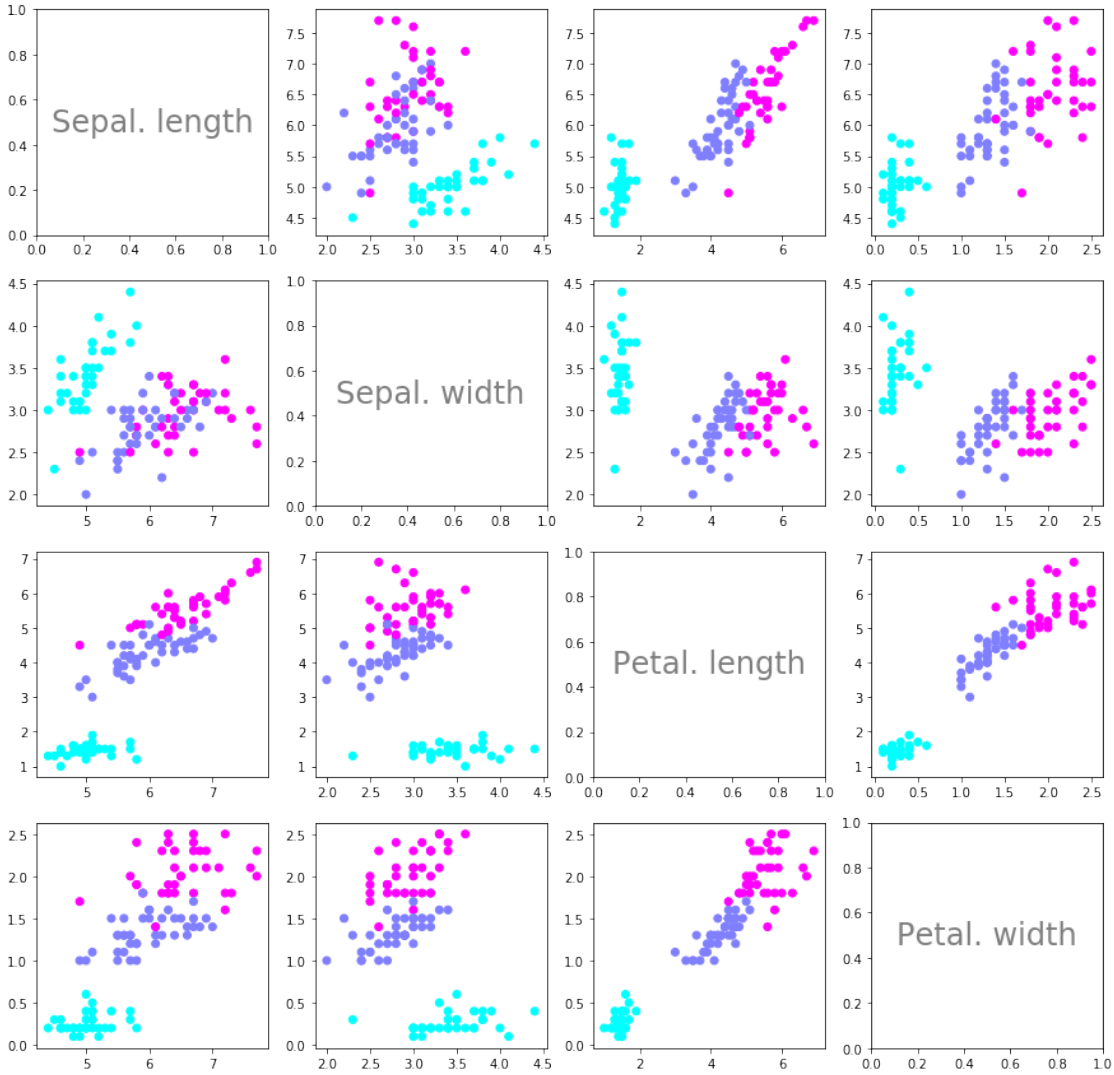
```
In [3]: def load_csv_dataset(split, path):
            dataset = np.loadtxt(path, delimiter=",", skiprows=1)
            X, y = dataset[:,:3], dataset[:,-1]
            X_train, X_test, y_train, y_test = \
                model_selection\
                        .train_test_split(X, y, random_state=123, test_size=(1 - split))
            return X_train, X_test, y_train, y_test
```

```
In [4]: # prepare data
        split = 0.75
        X_train, X_test, y_train, y_test = load_dataset(split)
```

3.1.5 Plot dataset

Since the data has 4 features, 16 scatterplots (4x4) are plotted showing the dependencies between each pair of features.

```
In [5]: f, axes = plt.subplots(4, 4,figsize=(15, 15))
        for i in range(4):
            for j in range(4):
                if j == 0 and i == 0:
                    axes[i,j].text(0.5, 0.5, 'Sepal. length', ha='center', va='center', size=24, alpha=
                elif j == 1 and i == 1:
                    axes[i,j].text(0.5, 0.5, 'Sepal. width', ha='center', va='center', size=24, alpha=.5
                elif j == 2 and i == 2:
                    axes[i,j].text(0.5, 0.5, 'Petal. length', ha='center', va='center', size=24, alpha=
                elif j == 3 and i == 3:
                    axes[i,j].text(0.5, 0.5, 'Petal. width', ha='center', va='center', size=24, alpha=.5
                else:
                    axes[i,j].scatter(X_train[:,j],X_train[:,i], c=y_train, cmap=plt.cm.cool)
```



3.1.6 Task 1: Euclidean distance

Compute Euclidean distance between two data points.

```
In [6]: def euclidean_distance(x1, x2):
            """Compute Euclidean distance between two data points.

            Parameters
            ----------
            x1 : array, shape (4)
                First data point.
            x2 : array, shape (4)
                Second data point.

            Returns
            -------
            distance : float
```

```
        Euclidean distance between x1 and x2.
    """
    from functools import reduce
    return reduce(lambda acc, cords: acc + (cords[1] - cords[0]) ** 2,
                  zip(x1, x2), 0) ** (1 / 2)
```

3.1.7 Task 2: get k nearest neighbors' labels

Get the labels of the k nearest neighbors of the datapoint *x_new*.

```
In [7]: def get_neighbors_labels(X_train, y_train, x_new, k):
            """Get the labels of the k nearest neighbors of the datapoint x_new.

            Parameters
            ----------
            X_train : array, shape (N_train, 4)
                Training features.
            y_train : array, shape (N_train)
                Training labels.
            x_new : array, shape (4)
                Data point for which the neighbors have to be found.
            k : int
                Number of neighbors to return.

            Returns
            -------
            neighbors_labels : array, shape (k)
                Array containing the labels of the k nearest neighbors.
            """
            distances = \
                list(
                    map(lambda el: (euclidean_distance(el[0], x_new), el[1]),
                        zip(X_train, y_train))
                )
            distances.sort(key=lambda element: element[0])
            return list(map(lambda element: element[1], distances[:k]))
```

3.1.8 Task 3: get the majority label

For the previously computed labels of the k nearest neighbors, compute the actual response. I.e. give back the class of the majority of nearest neighbors. In case of a tie, choose the "lowest" label (i.e. the order of tie resolutions is $0 > 1 > 2$).

```
In [8]: def get_response(neighbors_labels, num_classes=3):
            """Predict label given the set of neighbors.

            Parameters
            ----------
            neighbors_labels : array, shape (k)
                Array containing the labels of the k nearest neighbors.
            num_classes : int
                Number of classes in the dataset.

            Returns
            -------
```

```
            y : int
                Majority class among the neighbors.
            """
            return np.argmax([neighbors_labels.count(y) for y in range(num_classes)])
```

```
In [9]: def get_response_real(neighbors_values):
            """Predict label given the set of neighbors.

            Parameters
            ----------
            neighbors_values : array, shape (k)
                Array containing the labels of the k nearest neighbors.
            num_classes : int
                Number of classes in the dataset.

            Returns
            -------
            y : int
                Majority class among the neighbors.
            """
            return sum(neighbors_values)/float(len(neighbors_values))
```

3.1.9 Task 4: compute accuracy

Compute the accuracy of the generated predictions.

```
In [10]: def compute_accuracy(y_pred, y_test):
             """Compute accuracy of prediction.

             Parameters
             ----------
             y_pred : array, shape (N_test)
                 Predicted labels.
             y_test : array, shape (N_test)
                 True labels.
             """
             values, counts = np.unique(y_pred == y_test, return_counts=True)
             return dict(zip(values, counts))[True] / len(y_pred)
```

```
In [11]: # This function is given, nothing to do here.
         def predict(X_train, y_train, X_test, k, real=False):
             """Generate predictions for all points in the test set.

             Parameters
             ----------
             X_train : array, shape (N_train, 4)
                 Training features.
             y_train : array, shape (N_train)
                 Training labels.
             X_test : array, shape (N_test, 4)
                 Test features.
             k : int
                 Number of neighbors to consider.

             Returns
```

```
        -------
        y_pred : array, shape (N_test)
            Predictions for the test data.
        """
        y_pred = []
        for x_new in X_test:
            neighbors = get_neighbors_labels(X_train, y_train, x_new, k)
            y_pred.append(get_response(neighbors, k) if not real
                          else get_response_real(neighbors))
        return y_pred
```

3.1.10 Testing

Should output an accuracy of 0.9473684210526315.

```
In [12]: # prepare data
         split = 0.75
         X_train, X_test, y_train, y_test = load_dataset(split)
         print('Training set: {0} samples'.format(X_train.shape[0]))
         print('Test set: {0} samples'.format(X_test.shape[0]))

         # generate predictions
         k = 3
         y_pred = predict(X_train, y_train, X_test, k)
         accuracy = compute_accuracy(y_pred, y_test)
         print('Accuracy = {0}'.format(accuracy))
```

```
Training set: 112 samples
Test set: 38 samples
Accuracy = 0.9473684210526315
```

```
In [13]: split = 1
         X_train, _, y_train, _ = load_csv_dataset(split, "data/01_homework_dataset.csv")
         k = 3
         x_a = [4.1, -0.1, 2.2]
         x_b = [6.1, 0.4, 1.3]
         X_test = [x_a, x_b]
         y_pred = predict(X_train, y_train, X_test, k)
         print("x_a class = {}".format(y_pred[0]))
         print("x_b class = {}".format(y_pred[1]))
```

```
x_a class = 0
x_b class = 2
```

```
In [14]: split = 1
         X_train, _, y_train, _ = load_csv_dataset(split, "data/01_homework_dataset.csv")
         k = 3
         x_a = [4.1, -0.1, 2.2]
         x_b = [6.1, 0.4, 1.3]
         X_test = [x_a, x_b]
         y_pred = predict(X_train, y_train, X_test, k, real=True)
         print("x_a solution = {}".format(y_pred[0]))
         print("x_b solution = {}".format(y_pred[1]))
```

```
x_a solution = 1.0
x_b solution = 1.3333333333333333
```

In [ ]:

3.2 Problem 4

After adjusting load data method (reading from csv) following results were obrained (see *In [13]*):
For vector $\vec{x_a} = \begin{bmatrix}4.1 & -0.1 & 2.2\end{bmatrix}^T$ class **0**
For vector $\vec{x_b} = \begin{bmatrix}6.1 & 0.4 & 1.3\end{bmatrix}^T$ class **2**

3.3 Problem 5

After adjusting *predict(X_train, y_train, X_test, k, real=False)* method (see *In [11]*) and implementing *get_response_real(neighbors_values)* (see *In [9]*) following results were obrained (see *In [14]*):
For vector $\vec{x_a} = \begin{bmatrix}4.1 & -0.1 & 2.2\end{bmatrix}^T$ mean **1**
For vector $\vec{x_b} = \begin{bmatrix}6.1 & 0.4 & 1.3\end{bmatrix}^T$ mean **1.333**

3.4 Problem 6

When looking at above results it can be seen that for vector $\vec{x_a}$ label is assigned "randomly" (As mean is 1 and class is 0 we can deduce that 3 nearest neighbors of the vector have values 0, 1 and 2).
To overcome this, number of neighbors can be increased, but for such small data set (15 elements and 3 dimensions) this can lead to underfitting (we are "approaching" mean of whole data set).
For $k = 4$ $\vec{x_a}$ has label **2** and $\vec{x_b}$ has label **0**

Better solution would be to introduce weighted majority label with euclidean distance as weight. With this improvement there wouldn't be the necessity for assigning label "randomly".

Yet another solution for this type of problems would be to increase dataset size. Sample size of 15 seems small for 3 dimensional data. Although in this particular case all samples are concentrated in one region (column 1 and 3 greater than 0 and column 2 oscillating very close to 0). So curse of dimensionality shouldn't be decisive factor.

This problem does not occur when using decision trees. Samples can be a great distance apart, but they may have one very similar coordinate that decides to which class new sample is assigned.