Machine Learning: Homework #1

Due on October 30, 2017 at 10:00am

Professor Dr. Stephan Guennemann

Marc Meissner

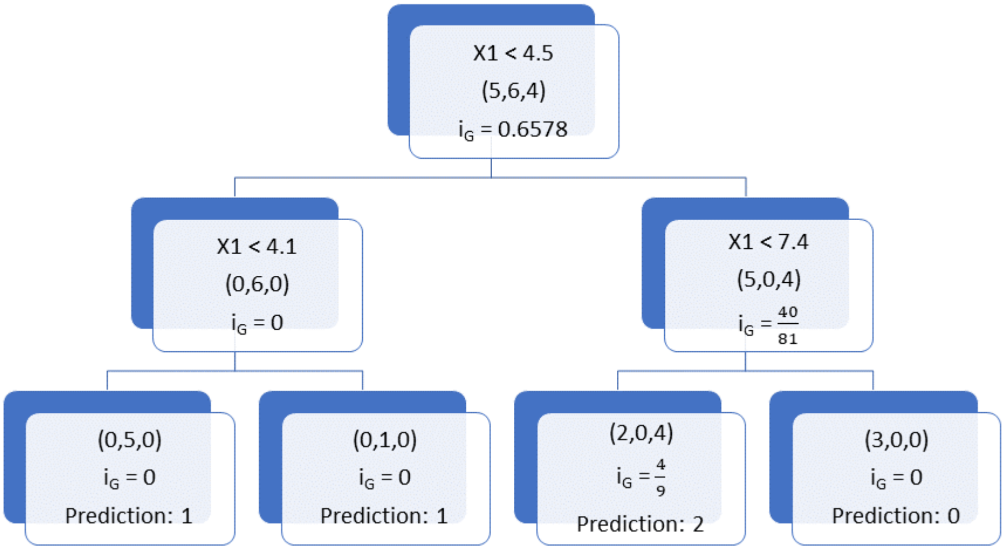


Figure 1: The decision tree built for Problem 1.

Problem 1

Build a decision tree T for your data X, y. Consider all possible splits for all features and use the Gini index to build your tree. Build the tree only to a depth of two! Provide at least the value of the final Gini index at each node and the distribution of classes at each leaf.

Solution The data is plotted in Figure 2. The code used to solve this problem is inspired by the tutorial available at *https://machinelearningmastery.com/implement-decision-tree-algorithm-scratch-python/* and is appended to the document. The resulting tree is showcased in Figure 1. A few interesting observations:

- The tree only uses feature $X1$ to split the data. It seems like the classes are primarily separated by this feature, especially class '1'.
- The Gini index is minimized by having "pure classes", as the index then turns to 0. The algorithm tries to achieve that property at the leaf nodes.
- The unstandardized data did not have an effect on the algorithm. More on this in Problem 6.

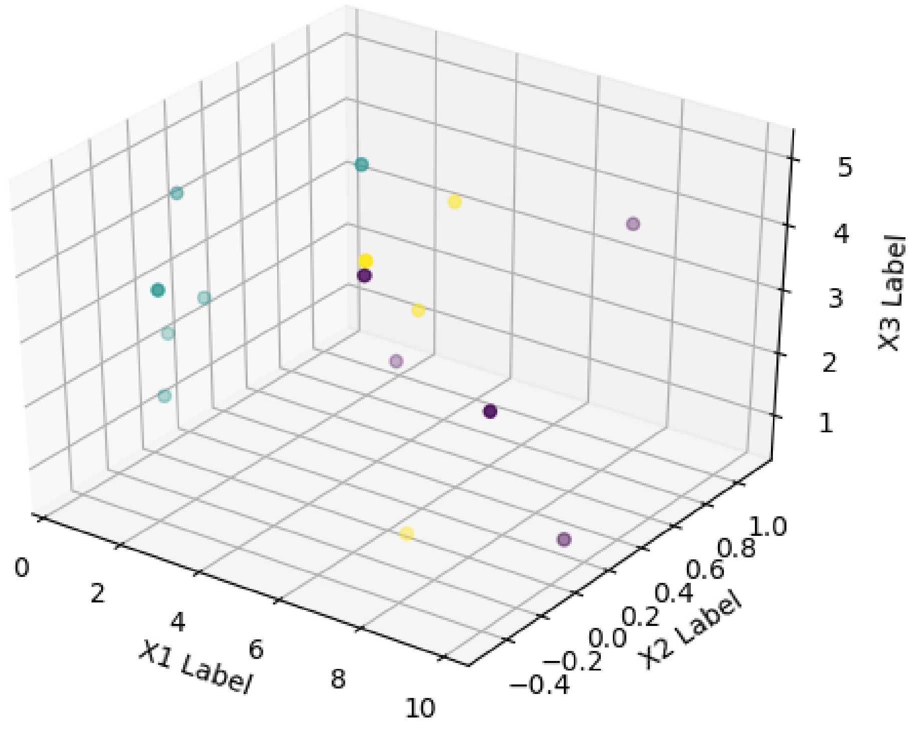


Figure 2: The data visualized in its three dimensions(features).

Problem 2

Use the final tree T from the previous problem to classify the vectors $x_a = \begin{pmatrix} 4.1 \\ -0.1 \\ 2.2 \end{pmatrix}^T$ and $x_a = \begin{pmatrix} 6.1 \\ 0.4 \\ 1.3 \end{pmatrix}^T$. Provide both your classification y_a and y_b and their respective probabilities $P\left(c = y_a | x_a, T\right)$ and $P\left(c = y_b | x_b, T\right)$.

Solution
x_a was classified as '1' with a probability of $P\left(c = 1 | x_a, T\right) = 1$. x_b was classified as '2' with a probability of $P\left(c = 2 | x_b, T\right) = \frac{2}{3}$.

Code Output:

```
[X1 < 4.500]
  [X1 < 4.100]
    [1.0]
    [1.0]
  [X1 < 7.400]
    [2.0]
    [0.0]

pred =
[1.0, 2.0]
```

Problem 3

Load the notebook 01_*homework*_*kNN.ipynb* from piazza. Fill in the missing code and run the notebook. Convert the evaluated notebook to pdf and add it to the printout of your homework.

Solution
The evaluated notebook is attached as a pdf document.

Problem 4

Classify the two vectors x_a and x_b given in Problem 2 with the k-nearest neighbors algorithm. Use k = 3 and Euclidean distance.

Solution
The 3-nearest neighbor algorithm with euclidean distance from the notebook classifies x_a and x_b as '1' and '2', respectively.

Problem 5

Now, consider y_i to be real-valued targets rather than classes. Perform 3-NN regression to label the vectors from Problem 2.

Solution

For (weighted) kNN regression, it is not sufficient anymore to have only the labels of the neighbors. One also needs the distances to the new sample. The following code changes account for that.

```
def get_neighbors(X_train, y_train, x_new, k):
    dist = np.zeros(np.size(y_train))
    for i in range(np.size(y_train)):
        dist[i] = euclidean_distance(X_train[i, :], x_new)
    maxDist = np.amax(dist) + 1.0
    neighbors_labels = np.empty(k)
    neighbors_distances = np.empty(k)
    for i in range(k):
        minInd = np.argmin(dist)
        neighbors_distances[i] = dist[minInd]
        neighbors_labels[i] = y_train[minInd]
        dist[minInd] = maxDist
    return neighbors_labels, neighbors_distances

def get_response(neighbors, distances, num_classes=3):
    sum_dist = np.sum(1/distances)
    y_hat = 0
    y_hat += np.sum((1/distances) * neighbors)
    y_hat /= sum_dist
    return y_hat
def predict(X_train, y_train, X_test, k):
    y_pred = []
    for x_new in X_test:
        neighbors, distances = get_neighbors(X_train, y_train, x_new, k)
        y_pred.append(get_response(neighbors,distances))
    return y_pred

dataset = np.genfromtxt('01_homework_dataset.csv', delimiter=',', skip_header=1,
                 skip_footer=0)
k = 3
X_train = dataset[:,0:3]
y_train = dataset[:,3]
X_test = np.array([[4.1, -0.1,2.2],[6.1,0.4,1.3]])
y_pred = predict(X_train, y_train, X_test, k)
print("y_pred␣=␣")
print(y_pred)
```

The resulting regression and output leads to:

```
y_pred =
[0.56101642597440038, 1.3959245132894498]
```

Problem 6

Look at the data. Which problem do you see w.r.t. building a Euclidean distance-based k-NN model on X? How can you compensate for this problem? Does this problem also arise when training a decision tree?

Solution
The mean values of the different features deviate strongly from each other. The euclidean value metric makes use of all features and is therefore susceptible to these deviations. One can compensate for this by standardizing the data and letting e.g. all values range from 0 to 1. Decision trees only split in regards to one feature when making their local decisions. For local changes, it does not matter how far nodes are placed apart according to the feature, but rather what the resulting split means for the distribution of nodes. Therefore, this effect is not dominant as it is in the case of kNN.

DECISION TREE CODE:

```
import numpy as np

# Split a dataset based on an attribute and an attribute value
def test_split(index, value, dataset):
    left, right = list(), list()
    for row in dataset:
        if row[index] < value:
            left.append(row)
        else:
            right.append(row)
    return left, right


# Calculate the Gini index for a split dataset
def gini_index(groups, classes):
    # count all samples at split point
    n_instances = float(sum([len(group) for group in groups]))
    # sum weighted Gini index for each group
    gini = 0.0
    for group in groups:
        size = float(len(group))
        # avoid divide by zero
        if size == 0:
            continue
        score = 0.0
        # score the group based on the score for each class
        for class_val in classes:
            p = [row[-1] for row in group].count(class_val) / size
            score += p * p
        # weight the group score by its relative size
        gini += (1.0 - score) * (size / n_instances)
    return gini


# Select the best split point for a dataset
def get_split(dataset):
    class_values = list(set(row[-1] for row in dataset))
    b_index, b_value, b_score, b_groups = 999, 999, 999, None
    for index in range(len(dataset[0]) - 1):
        for row in dataset:
            groups = test_split(index, row[index], dataset)
            gini = gini_index(groups, class_values)
            if gini < b_score:
                b_index, b_value, b_score, b_groups = index, row[index], gini, groups
    return {'index': b_index, 'value': b_value, 'groups': b_groups}
```

```
# Create a terminal node value
def to_terminal(group):
    outcomes = [row[-1] for row in group]
    return max(set(outcomes), key=outcomes.count)


# Create child splits for a node or make terminal
def split(node, max_depth, min_size, depth):
    left, right = node['groups']
    del (node['groups'])
    # check for a no split
    if not left or not right:
        node['left'] = node['right'] = to_terminal(left + right)
        return
    # check for max depth
    if depth >= max_depth:
        node['left'], node['right'] = to_terminal(left), to_terminal(right)
        return
    # process left child
    if len(left) <= min_size:
        node['left'] = to_terminal(left)
    else:
        node['left'] = get_split(left)
        split(node['left'], max_depth, min_size, depth + 1)
    # process right child
    if len(right) <= min_size:
        node['right'] = to_terminal(right)
    else:
        node['right'] = get_split(right)
        split(node['right'], max_depth, min_size, depth + 1)


# Build a decision tree
def build_tree(train, max_depth, min_size):
    root = get_split(train)
    split(root, max_depth, min_size, 1)
    return root


# Make a prediction with a decision tree
def predict(node, row):
    if row[node['index']] < node['value']:
        if isinstance(node['left'], dict):
            return predict(node['left'], row)
        else:
            return node['left']
    else:
        if isinstance(node['right'], dict):
            return predict(node['right'], row)
        else:
            return node['right']

def print_tree(node, depth=0):
    if isinstance(node, dict):
        print('%s[X%d < %.3f]' % ((depth * ' ', (node['index'] + 1), node['value'])))
        print_tree(node['left'], depth + 1)
        print_tree(node['right'], depth + 1)
    else:
        print('%s[%s]' % ((depth * ' ', node)))

# Classification and Regression Tree Algorithm
def decision_tree(train, test, max_depth, min_size):
    tree = build_tree(train, max_depth, min_size)
    print_tree(tree)
    predictions = list()
    for row in test:
        prediction = predict(tree, row)
        predictions.append(prediction)
    return (predictions)

max_depth = 2
min_size = 0

dataset = np.genfromtxt('01_homework_dataset.csv', delimiter=',', skip_header=1,
                    skip_footer=0)
datalist = dataset.tolist()
test = [[4.1,-0.1,2.2],[6.1,0.4,1.3]]

pred = decision_tree(datalist,test,max_depth,min_size)
print(pred)
```

01_homework_knn

October 26, 2017

1 Programming assignment 1: k-Nearest Neighbors classification

```
In [14]: import numpy as np
         from sklearn import datasets, model_selection
         import matplotlib.pyplot as plt
         %matplotlib inline
```

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/

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).

Once you complete the assignments, export the entire notebook as PDF using nbconvert and attach it to your homework solutions. 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.

1.3 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`.

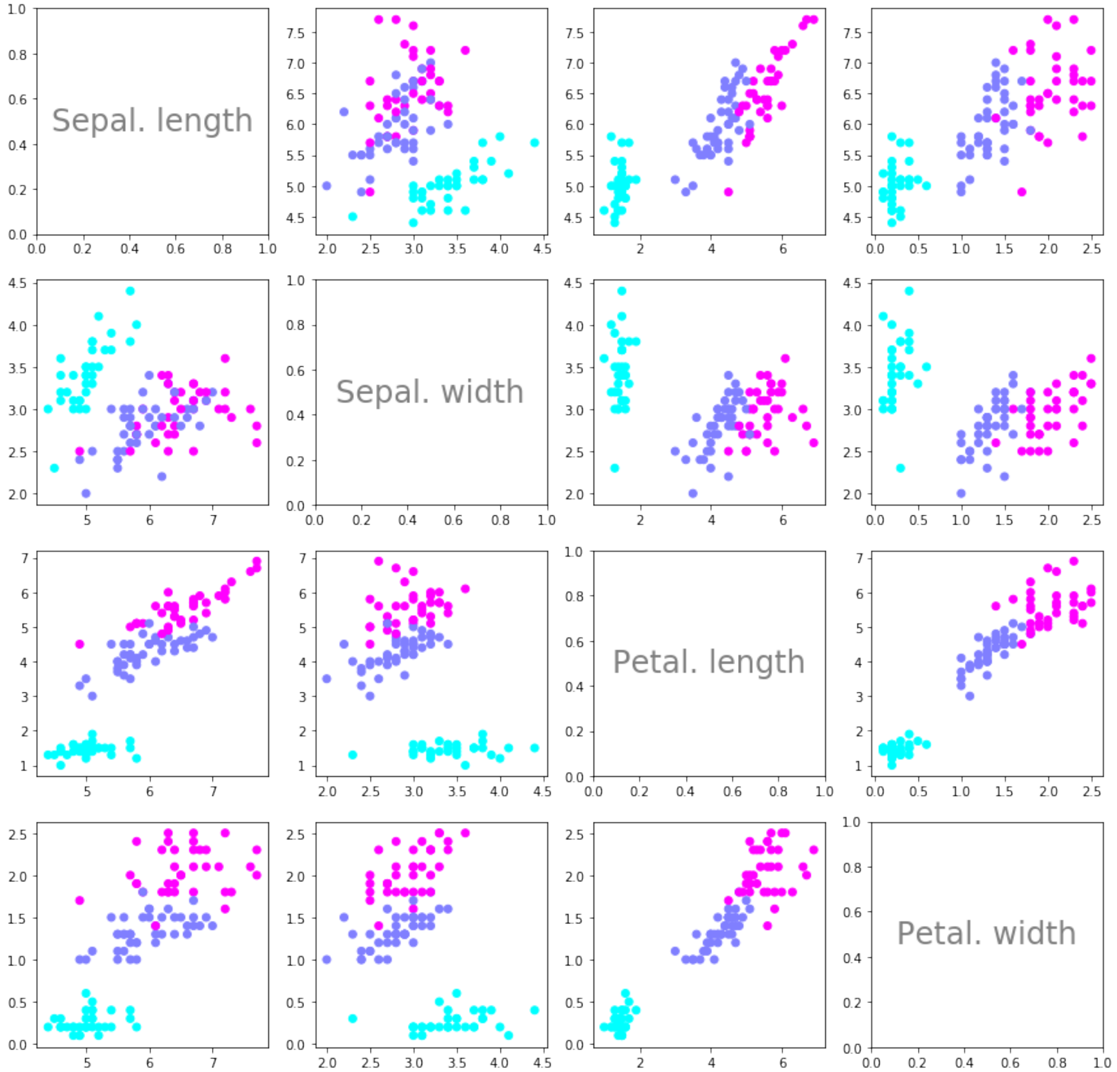
```
In [15]: def load_dataset(split):
             """Load and split the dataset into training and test parts.

             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_st
             return X_train, X_test, y_train, y_test
```

```
In [16]: # prepare data
         split = 0.67
         X_train, X_test, y_train, y_test = load_dataset(split)
```

1.4 Plot dataset

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

```
In [17]: 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
                 elif j == 1 and i == 1:
                     axes[i,j].text(0.5, 0.5, 'Sepal. width', ha='center', va='center', size=24,
                 elif j == 2 and i == 2:
                     axes[i,j].text(0.5, 0.5, 'Petal. length', ha='center', va='center', size=24
                 elif j == 3 and i == 3:
                     axes[i,j].text(0.5, 0.5, 'Petal. width', ha='center', va='center', size=24,
                 else:
                     axes[i,j].scatter(X_train[:,j],X_train[:,i], c=y_train, cmap=plt.cm.cool)
```



1.5 Task 1: Euclidean distance

Compute Euclidean distance between two data points.

```
In [18]: 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.
    """
    # TODO
    distance = 0
    for index, elem in enumerate(x1):
        distance += np.square(x1[index]-x2[index])
    distance = np.sqrt(distance)
    return distance
```

1.6 Task 2: get k nearest neighbors' labels

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

```
In [19]: 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.
             """
             dist = np.zeros(np.size(y_train))
             for i in range(np.size(y_train)):
                 dist[i] = euclidean_distance(X_train[i,:],x_new)
             maxDist = np.amax(dist)+1.0
             neighbors_labels = []
             for i in range(k):
                 minInd = np.argmin(dist)
                 dist[minInd] = maxDist
                 neighbors_labels.append(y_train[minInd])
             return neighbors_labels
```

1.7 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. Think about how a tie is handled by your solution.

```
In [20]: def get_response(neighbors, 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.
             """
             # TODO
             class_votes = np.zeros(num_classes)
             for elem in neighbors:
                 class_votes[elem] += 1
             return np.argmax(class_votes)
```

1.8 Task 4: compute accuracy

Compute the accuracy of the generated predictions.

```
In [21]: 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.
             """
             # TODO
             n_right = 0
             for ind, elem in enumerate(y_pred):
                 if elem == y_test[ind]:
                     n_right += 1
             return n_right/np.size(y_pred)
```

```
In [22]: # This function is given, nothing to do here.
         def predict(X_train, y_train, X_test, k):
```

```
    """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))
    return y_pred
```

1.9 Testing

Should output an accuracy of 0.9473684210526315.

```
In [23]: # prepare data
         split = 0.67
         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
```