Week 2 Homework Submission

Iuliia Skobleva

October 27, 2019

kNN Classification

Problem 1

The visualization for our data can be seen in , where Blue stands for Class 1 and red stands for Class 2 (I have left out the marks' names). We identify the nearest neighbors by eye.

(a) We start with the L_1-norm defined as $L_1 = \sum_{i=1}^{2} |x_i|$

(b) We continue with the L_2-norm defined as $L_2 = \sqrt{\sum_{i=1}^{2} |x_i|^2}$

(c) We see that the nearest neighbors calculated using the L_2-norm automatically mean they will be the nearest neighbors in the L_1-norm but not vice versa!

In the table below we find the values for the distances calculated for two different norms.

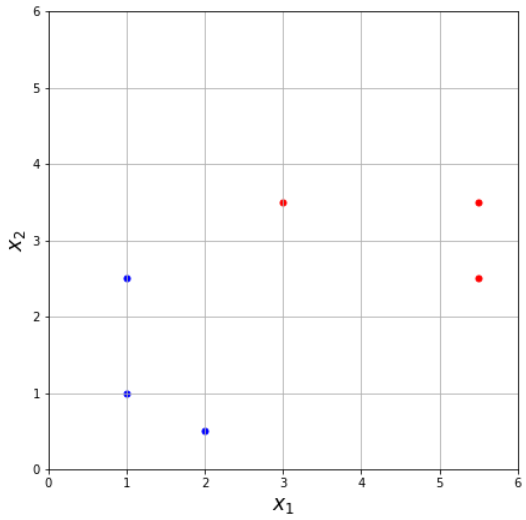
point	its closest Neighbor	L_1-distance	L_2-distance
A	B	1.50	1.11
B	A	1.50	1.11
C	A	1.50	1.50
D	C	3.00	2.24
E	F	1.00	1.00
F	E	1.00	1.00

Problem 2

First off, $k = 112$.

(a) A new point x_{new} is 14% (16/112) likely to be identified as 'A' class, 29% (32/112) to be classified as 'B' and 57% (64/112) to be identified as 'C'.

(b) If we used weighted kNN-classification the probability with which a new point would be identified as a certain class, would be reversely proportionate to the distance to the neighboring points. So, if x_{new} is closest to points of class 'A', even though class 'A' is only 1/7 of the set, x_new will most likely be identified as 'A'.



Decision Trees

Problem 3

You cannot build a decision tree of depth 1 by using a single feature like: "is $x_1 > 0.5$? if yes - Class Blue, if no - Class Red". However, it is possible to ask this question using the two features simultaneously.

Is $\mathbf{x_2/x_1} \geq \mathbf{1}$? If **yes**, then Class Blue. If **no**, then Class Red.

This is possible because a line with slope equal to 1 splits the data perfectly. Everything above that line is blue and everything below is red.

Problem 4

(a) We calculate the entropy using the formula from the lecture $i_H(t) = -\sum_{c_i \in C} \pi_{c_i} \log \pi_{c_i}$, where π_{c_i} is a probability of a point being in a certain class. In our case:

$$i_H(t) = -\pi_{win} \log \pi_{win} - \pi_{lose} \log \pi_{lose} \tag{1}$$

With $\pi_{win} = 0.4$ and $\pi_{lose} = 0.6$ the entropy is $i_H(t) = 0.29$.

(b) By looking at the data it seems right to split on x_3. Let's prove it. The change in entropy after splitting at node t is:

$$\Delta i(s,t) = i(t) - p_L \cdot i(t_L) - p_R \cdot i(t_R) \tag{2}$$

with $i_E(t)$ being defined as:

$$i_E(t) = 1 - \max p(y = c|t) \tag{3}$$

1) if we split on x_1 we would have 2 wins and 3 losses in each category - team & individual. Here, $\Delta i_E(t) = 0.4 - 0.5 \cdot 0.4 - 0.5 \cdot 0.4 = 0$.

0.4 comes from 1 - 0.6 (0.6 being the probability of losing in this data set). 0.5 is there because in each node we have 50% of the data. 0.4, again, is 1 - 0.6(probability of a loss in a node).[1]

2) if we split on x_2 we would have 4 instances in node 'mental' (2 wins and 2 losses) and 6 instances in node 'physical' (2 wins and 4 losses). the change in entropy here is $\Delta i_E(t) = 0.4 - 0.4 \cdot 0.5 - 0.6 \cdot 0.3 = 0.02$. as we see, the change is rather small.

3) if we split on x_3, we have 5 instances in 'skill'(3 wins and 2 losses) and 5 instances in 'chance'(1 win and 4 losses) with $\Delta i_E(t) = 0.4 - 0.5 \cdot 0.4 - 0.5 \cdot 0.2 = 0.1$. here we have the maximal change of entropy and therefore, when splitting on this condition we get the purest nodes.

[1] These comments are mostly for me so that I understand what I did here later. Probably just ignore them.

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
```

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/

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).

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.

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`.

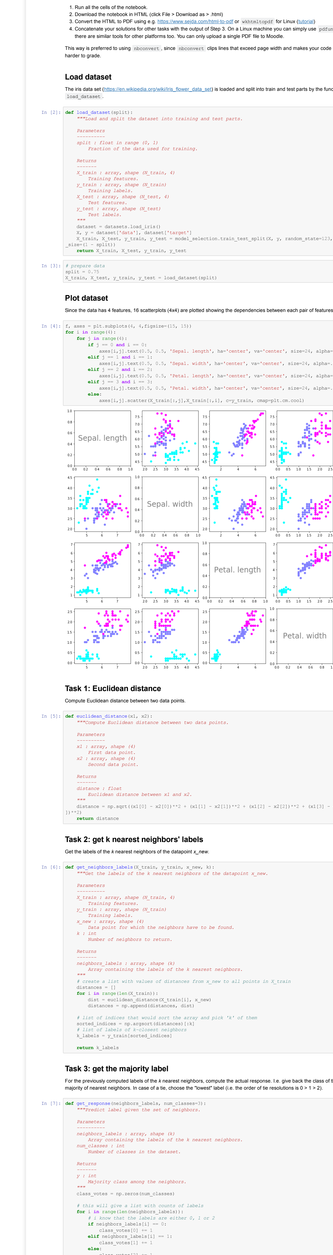
In [2]:
```
def load_dataset(split):
    """Load and split the dataset into training and test parts.

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

Plot dataset

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

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



Task 1: Euclidean distance

Compute Euclidean distance between two data points.

In [5]:
```
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.
    """
    distance = np.sqrt((x1[0] - x2[0])**2 + (x1[1] - x2[1])**2 + (x1[2] - x2[2])**2 + (x1[3] - x2[3])**2)
    return distance
```

Task 2: get k nearest neighbors' labels

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

In [6]:
```
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.
    """
    # create a list with values of distances from x_new to all points in X_train
    distances = []
    for i in range(len(X_train)):
        dist = euclidean_distance(X_train[i], x_new)
        distances = np.append(distances, dist)

    # list of indices that would sort the array and pick 'k' of them
    sorted_indices = np.argsort(distances)[:k]
    # list of labels of k-closest neighbors
    k_labels = y_train[sorted_indices]

    return k_labels
```

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 [7]:
```
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.
    """
    class_votes = np.zeros(num_classes)

    # this will give a list with counts of labels
    for i in range(len(neighbors_labels)):
        # Note that the labels are either 0, 1 or 2
        if neighbors_labels[i] == 0:
            class_votes[0] += 1
        elif neighbors_labels[i] == 1:
            class_votes[1] += 1
        else:
            class_votes[2] += 1

    return np.argmax(class_votes)
```

Task 4: compute accuracy

Compute the accuracy of the generated predictions.

In [8]:
```
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.
    """
    count = 0
    for i in range(len(y_pred)):
        if y_pred[i] == y_test[i]:
            count += 1
    accuracy = count/len(y_pred)

    return accuracy
```

In [9]:
```
# 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
```

Testing

Should output an accuracy of 0.9473684210526315.

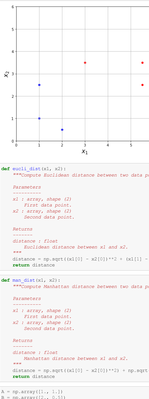
In [10]:
```
# prepare data
split = 0.75
X_train, X_test, y_train, y_test = load_dataset(split)
print("Training set: {0} samples".format(X_train.shape[0]))
print("Test set: {0} samples".format(X_test.shape[0]))

# 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
```

Problem 1

In [11]:
```
x_1 = np.array([1.,2.,1.])
x_2 = np.array([1.,2.5,3.5])
y_1 = np.array([1.,2.5,2.5])
y_2 = np.array([3.5,3.5,2.5])
```

In [12]:
```
plt.figure(figsize = (7,7))
plt.grid()
plt.scatter(x_1, y_1, s=25.5, color="blue")
plt.scatter(x_2, y_2, s=25.5, color="red")
plt.xlim(0,5)
plt.ylim(0,5)
plt.xlabel("$x_1$", size = 17)
plt.ylabel("$x_2$", size = 17)
plt.savefig("kNNneighbors.png")

plt.show()
```



In [13]:
```
def eucli_dist(x1, x2):
    """Compute Euclidean distance between two data points.

    Parameters
    ----------
    x1 : array, shape (2)
        First data point.
    x2 : array, shape (2)
        Second data point.

    Returns
    -------
    distance : float
        Euclidean distance between x1 and x2.
    """
    distance = np.sqrt((x1[0] - x2[0])**2 + (x1[1] - x2[1])**2)
    return distance
```

In [14]:
```
def man_dist(x1, x2):
    """Compute Manhattan distance between two data points.

    Parameters
    ----------
    x1 : array, shape (2)
        First data point.
    x2 : array, shape (2)
        Second data point.

    Returns
    -------
    distance : float
        Manhattan distance between x1 and x2.
    """
    distance = np.sqrt((x1[0] - x2[0])**2) + np.sqrt((x1[1] - x2[1])**2)
    return distance
```

In [15]:
```
A = np.array([1., 1.])
B = np.array([1., 0.5])
C = np.array([1., 2.5])
D = np.array([2., 3.5])
E = np.array([3.5, 3.5])
F = np.array([3.5, 2.5])
```

In [16]:
```
print(eucli_dist(C, D))
```

```
2.23606797749979
```

In [17]:
```
print(man_dist(A, B))
```

```
1.5
```