# Assignment 1 – K-NN algorithm

This assignment is worth 20% of your course grade. You will implement the K-Nearest-Neighbors algorithm in Python. The task is split into several functions, as specified in the section **Tasks**.

You should define all the required functions in a single Python file, named `knn_functions.py`. You should upload this file as individual assignment via Blackboard.

The file should only contain **function definitions** (possibly with comments), but no other executable code.

For this assignment you can use any standard Python modules, but do not use any external libraries not included with Python.

## Code reuse rules

Remember that assignment 1 is individual work. You are **not allowed** to collaborate or share code with other students. **Submissions will be checked for plagiarism.**

You are welcome to study and learn from code found in online sources such as books, tutorials and blogs. You are **not allowed** to copy fragments of such code into your assignment. The only form of code reuse allowed is to call library functions.

If you are found breaking the above rules you will be reported to the Examination Board for fraud.

## Grading

This assignment will be checked and graded using an automated process. Make sure you spell the function names correctly. Make sure the file you submit is accepted with no error by the Python interpreter. If your file gives errors when loaded into the interpreter, you may get zero credit. Make sure your functions give the correct answers by testing them thoroughly before submitting your assignment.

## Tasks

### Task 0 (example)

Define function `words` which takes a string as an parameter and returns a list of space-delimited words in the input string. Once defined, your function should work like this:

Examples:

```
>>> words('this is an example')
['this', 'is', 'an', 'example']
>>> words('xyz')
['xyz']
```

Answer:

```
def words(input):
   "Split input string into space-separated list of words."
   return input.split(" ")
```

## Task 1

### *1 point*

Define function `distance` which computes the Euclidean distance between two points. The function takes two arguments:

- point *a*
- point *b*

and returns the distance between them. A point is represented as a list of numbers. You can assume that both lists have the same length, but this length can be any positive whole number.

For example:

```
>>> a = [1.2, 3.2, 0.0]
>>> b = [1.2, -3.2, -1.0]
>>> print distance(a, b)
6.47765389628
>>> x = [1, 2]
>>> y = [-2, -1]
>>> print distance(x, y)
4.24264068712
```

## Task 2

### *1 point*

For this task you should define the function `argsort`, which take a list of values, and returns a list of the indexes (positions) ordered according the the corresponding values. That is, the index of the smallest value should appear as the first element of the returned list, the index of the next smallest value as the second element etc. For example:

```
>>> A = [0.0, 90.0, -20.0, -4.0]
>>> print argsort(A)
[2, 3, 0, 1]
>>> B = ['x', 'y', 'z', 'a', 'b']
>>> argsort(B)
[3, 4, 0, 1, 2]
```

## Task 3

### *1 point*

In this task we will define the function `histogram` which we will use to count how many of each type of target there are in list containing the targets of the *k*-nearest neighbors. The function `histogram` should take a list of values (possibly with repetitions) and should return a dictionary with the values as keys, and their frequencies as values. For example:

```
>>> neighbors = ['versicolor', 'setosa', 'setosa']
>>> print histogram(neighbors)
{'setosa': 2, 'versicolor': 1}
>>> L = ['a', 'b', 'b', 'a']
>>> histogram(L)
{'a': 2, 'b': 2}
```

## Task 4

### *1 point*

Define function `keymax` which takes a dictionary, and returns the key associated with the largest value. We can use this function to find the most frequent target from the frequencies computed by `histogram` (if there are more than one keys with the same highest value, you should only return one of them). For example:

```
>>> freq = {'setosa': 2, 'versicolor': 1}
>>> print keymax(freq)
setosa
>>> d = {'a': -2, 'b': 72, 'c': 100}
>>> print keymax(d)
c
```

## Task 5

### *1 point*

The K-NN algorithm is a memory-based method which means that it memorizes all the training examples. For this task we will define the function `train` which takes two arguments:

- *inputs* - a list of inputs (where a single input is a list of numbers)
- *targets* - a list of target values

It should create a single new list which contains input-target pairs, and return this list. For example:

```
>>> X = [ [6.4, 3.2, 5.3, 2.3], [4.9, 3.1, 1.5, 0.1] ] # List of two inputs
>>> Y = ['virginica', 'setosa'] # List of two targets
>>> model = train(X, Y)
>>> for pair in model:  print pair
...
([6.4, 3.2, 5.3, 2.3], 'virginica')
([4.9, 3.1, 1.5, 0.1], 'setosa')
```

## Task 6

### *2 points*

In this task we implement the main functionality of the K-NN algorithm. You need to code the function `predict` which takes two normal arguments:

- *model* - this is the memorized list of training examples, as returned by the function `train`,
- *new* - this is the new input (represented as a list of numbers) for which we will predict the target.

And additionally, it should also accept the keyword argument *k*, which specifies how many nearest neighbors we consider.

This function should return the target which is most frequent among the *k* nearest neighbors of *new*. You can use the following steps to code this function:

a. compute the list of distances between *new* and each input in *model* (using `distance`)
b. compute the list of indexes of this list, sorted by smallest distance (using `argsort`)
c. keep the first *k* indexes of this sorted list

d. find the targets corresponding to these *k* indices in *model*

e. calculate the frequencies of these *k* targets (using `histogram`)

f. find and return the most frequent target (using `keymax`)

For example:

```
>>> # Tiny training set of two examples
>>> X = [[6.4, 3.2, 5.3, 2.3], [4.9, 3.1, 1.5, 0.1]]
>>> Y = ['virginica', 'setosa']
>>> model = train(X, Y)
>>> new = [6.3, 3.1, 5.2, 2.3]
>>> new_target = predict(model, new, k=1)
>>> print new_target
virginica
```

## Task 7

### *1 point*

In this task you'll implement the function `error_rate` which we'll use to calculate the proportion of mistakes made by the classifier. Error rate should take two lists of the same length:

- *gold* - list of true targets
- *predicted* - list of predicted targets

It should count in how many cases the elements in the same position are not the same. It should return three values:

- the number of mistakes
- the total size of the *gold* list
- the ratio of number of mistakes to the size

For example:

```
>>> A = ['setosa', 'setosa', 'virginica']
>>> B = ['setosa', 'versicolor', 'virginica']
>>> print error_rate(A,B)
(1, 3, 0.3333333333333333)
>>> Q = ['a','a','a','a']
>>> W = ['b','b','b','b']
>>> print error_rate(Q,W)
(4, 4, 1.0)
```

## Testing complete algorithm

Once you have correctly defined all the functions, you will be able to run the K-NN algorithm on the *iris* dataset. In order to do this, get the script `knn.py` and the datafiles `iris-train.txt` and `iris-dev.txt` from the assignment page on BlackBoard. Put your assignment file named `knn_functions.py` in the same directory as `knn.py` and the data files. If you execute the script `knn.py` it will run the algorithm on the data and report the results.