Exercise 1

The first exercise simply requires loading the dataset using pandas. You can either download the dataset locally and load it from there, or you can use pandas to access a remote file through its url.

This second option is simpler, but only recommended if you only run your code once. If you have to re-run your code multiple times, you might want to consider downloading the file once and storing it locally.

In [1]:
```
import pandas as pd
import numpy as np

df = pd.read_csv("https://archive.ics.uci.edu/ml/machine-learning-databases/
iris/iris.data", header=None)
len(df)
```

Out[1]:
```
150
```

Notice that, since the iris.data file does not have a headers row (i.e. a row with the column names in it), we are passing the `header=None` parameter -- meaning that Pandas will expect to find a valid data point in the first row.

If we do not specify this parameter, the default behavior for Pandas is to use the first column as the column names -- resulting in a dataset of 149 points, instead of 150.

As a general rule, you always want to inspect the files you are working with, to assess this kind of things before blindly approaching them.

In [2]:
```
df
```

Out[2]:

	0	1	2	3	4
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa
...	...	...	...	...	...
145	6.7	3.0	5.2	2.3	Iris-virginica
146	6.3	2.5	5.0	1.9	Iris-virginica
147	6.5	3.0	5.2	2.0	Iris-virginica
148	6.2	3.4	5.4	2.3	Iris-virginica
149	5.9	3.0	5.1	1.8	Iris-virginica

150 rows × 5 columns

Exercise 2 and 3
Before building our matrices X_train, X_test and vectors y_train, y_test, we will first build the design matrix *X* and the labels to be predicted *y*.

A design matrix is a canonical way of representing a dataset: each row of the design matrix contains a data point, each column represents one of the features of our data points.

For the iris dataset, the design matrix will have 150 rows (one for each iris measured) and 4 columns (one for each measurement -- petal width and length, sepal width and length). The labels vector will instead be a row of values: the i-th value will be the target label for the i-th element of our design matrix.

We can access the matrix representation underneath the DataFrame df using the `values` attribute:

```
In [3]: df.values
```

```
Out[3]: array([[5.1, 3.5, 1.4, 0.2, 'Iris-setosa'],
                [4.9, 3.0, 1.4, 0.2, 'Iris-setosa'],
                [4.7, 3.2, 1.3, 0.2, 'Iris-setosa'],
                [4.6, 3.1, 1.5, 0.2, 'Iris-setosa'],
                [5.0, 3.6, 1.4, 0.2, 'Iris-setosa'],
                [5.4, 3.9, 1.7, 0.4, 'Iris-setosa'],
                [4.6, 3.4, 1.4, 0.3, 'Iris-setosa'],
                [5.0, 3.4, 1.5, 0.2, 'Iris-setosa'],
                [4.4, 2.9, 1.4, 0.2, 'Iris-setosa'],
                [4.9, 3.1, 1.5, 0.1, 'Iris-setosa'],
                [5.4, 3.7, 1.5, 0.2, 'Iris-setosa'],
                [4.8, 3.4, 1.6, 0.2, 'Iris-setosa'],
                [4.8, 3.0, 1.4, 0.1, 'Iris-setosa'],
                [4.3, 3.0, 1.1, 0.1, 'Iris-setosa'],
                [5.8, 4.0, 1.2, 0.2, 'Iris-setosa'],
                [5.7, 4.4, 1.5, 0.4, 'Iris-setosa'],
                [5.4, 3.9, 1.3, 0.4, 'Iris-setosa'],
                [5.1, 3.5, 1.4, 0.3, 'Iris-setosa'],
                [5.7, 3.8, 1.7, 0.3, 'Iris-setosa'],
                [5.1, 3.8, 1.5, 0.3, 'Iris-setosa'],
                [5.4, 3.4, 1.7, 0.2, 'Iris-setosa'],
                [5.1, 3.7, 1.5, 0.4, 'Iris-setosa'],
                [4.6, 3.6, 1.0, 0.2, 'Iris-setosa'],
                [5.1, 3.3, 1.7, 0.5, 'Iris-setosa'],
                [4.8, 3.4, 1.9, 0.2, 'Iris-setosa'],
                [5.0, 3.0, 1.6, 0.2, 'Iris-setosa'],
                [5.0, 3.4, 1.6, 0.4, 'Iris-setosa'],
                [5.2, 3.5, 1.5, 0.2, 'Iris-setosa'],
                [5.2, 3.4, 1.4, 0.2, 'Iris-setosa'],
                [4.7, 3.2, 1.6, 0.2, 'Iris-setosa'],
                [4.8, 3.1, 1.6, 0.2, 'Iris-setosa'],
                [5.4, 3.4, 1.5, 0.4, 'Iris-setosa'],
                [5.2, 4.1, 1.5, 0.1, 'Iris-setosa'],
                [5.5, 4.2, 1.4, 0.2, 'Iris-setosa'],
                [4.9, 3.1, 1.5, 0.1, 'Iris-setosa'],
                [5.0, 3.2, 1.2, 0.2, 'Iris-setosa'],
                [5.5, 3.5, 1.3, 0.2, 'Iris-setosa'],
                [4.9, 3.1, 1.5, 0.1, 'Iris-setosa'],
                [4.4, 3.0, 1.3, 0.2, 'Iris-setosa'],
                [5.1, 3.4, 1.5, 0.2, 'Iris-setosa'],
                [5.0, 3.5, 1.3, 0.3, 'Iris-setosa'],
                [4.5, 2.3, 1.3, 0.3, 'Iris-setosa'],
                [4.4, 3.2, 1.3, 0.2, 'Iris-setosa'],
                [5.0, 3.5, 1.6, 0.6, 'Iris-setosa'],
                [5.1, 3.8, 1.9, 0.4, 'Iris-setosa'],
                [4.8, 3.0, 1.4, 0.3, 'Iris-setosa'],
                [5.1, 3.8, 1.6, 0.2, 'Iris-setosa'],
                [4.6, 3.2, 1.4, 0.2, 'Iris-setosa'],
                [5.3, 3.7, 1.5, 0.2, 'Iris-setosa'],
                [5.0, 3.3, 1.4, 0.2, 'Iris-setosa'],
                [7.0, 3.2, 4.7, 1.4, 'Iris-versicolor'],
                [6.4, 3.2, 4.5, 1.5, 'Iris-versicolor'],
                [6.9, 3.1, 4.9, 1.5, 'Iris-versicolor'],
                [5.5, 2.3, 4.0, 1.3, 'Iris-versicolor'],
                [6.5, 2.8, 4.6, 1.5, 'Iris-versicolor'],
                [5.7, 2.8, 4.5, 1.3, 'Iris-versicolor'],
                [6.3, 3.3, 4.7, 1.6, 'Iris-versicolor'],
                [4.9, 2.4, 3.3, 1.0, 'Iris-versicolor'],
                [6.6, 2.9, 4.6, 1.3, 'Iris-versicolor'],
                [5.2, 2.7, 3.9, 1.4, 'Iris-versicolor'],
                [5.0, 2.0, 3.5, 1.0, 'Iris-versicolor'],
                [5.9, 3.0, 4.2, 1.5, 'Iris-versicolor'],
                [6.0, 2.2, 4.0, 1.0, 'Iris-versicolor'],
                [6.1, 2.9, 4.7, 1.4, 'Iris-versicolor'],
                [5.6, 2.9, 3.6, 1.3, 'Iris-versicolor'],
[6.7, 3.1, 4.4, 1.4, 'Iris-versicolor'],
[5.6, 3.0, 4.5, 1.5, 'Iris-versicolor'],
[5.8, 2.7, 4.1, 1.0, 'Iris-versicolor'],
[6.2, 2.2, 4.5, 1.5, 'Iris-versicolor'],
[5.6, 2.5, 3.9, 1.1, 'Iris-versicolor'],
[5.9, 3.2, 4.8, 1.8, 'Iris-versicolor'],
[6.1, 2.8, 4.0, 1.3, 'Iris-versicolor'],
[6.3, 2.5, 4.9, 1.5, 'Iris-versicolor'],
[6.1, 2.8, 4.7, 1.2, 'Iris-versicolor'],
[6.4, 2.9, 4.3, 1.3, 'Iris-versicolor'],
[6.6, 3.0, 4.4, 1.4, 'Iris-versicolor'],
[6.8, 2.8, 4.8, 1.4, 'Iris-versicolor'],
[6.7, 3.0, 5.0, 1.7, 'Iris-versicolor'],
[6.0, 2.9, 4.5, 1.5, 'Iris-versicolor'],
[5.7, 2.6, 3.5, 1.0, 'Iris-versicolor'],
[5.5, 2.4, 3.8, 1.1, 'Iris-versicolor'],
[5.5, 2.4, 3.7, 1.0, 'Iris-versicolor'],
[5.8, 2.7, 3.9, 1.2, 'Iris-versicolor'],
[6.0, 2.7, 5.1, 1.6, 'Iris-versicolor'],
[5.4, 3.0, 4.5, 1.5, 'Iris-versicolor'],
[6.0, 3.4, 4.5, 1.6, 'Iris-versicolor'],
[6.7, 3.1, 4.7, 1.5, 'Iris-versicolor'],
[6.3, 2.3, 4.4, 1.3, 'Iris-versicolor'],
[5.6, 3.0, 4.1, 1.3, 'Iris-versicolor'],
[5.5, 2.5, 4.0, 1.3, 'Iris-versicolor'],
[5.5, 2.6, 4.4, 1.2, 'Iris-versicolor'],
[6.1, 3.0, 4.6, 1.4, 'Iris-versicolor'],
[5.8, 2.6, 4.0, 1.2, 'Iris-versicolor'],
[5.0, 2.3, 3.3, 1.0, 'Iris-versicolor'],
[5.6, 2.7, 4.2, 1.3, 'Iris-versicolor'],
[5.7, 3.0, 4.2, 1.2, 'Iris-versicolor'],
[5.7, 2.9, 4.2, 1.3, 'Iris-versicolor'],
[6.2, 2.9, 4.3, 1.3, 'Iris-versicolor'],
[5.1, 2.5, 3.0, 1.1, 'Iris-versicolor'],
[5.7, 2.8, 4.1, 1.3, 'Iris-versicolor'],
[6.3, 3.3, 6.0, 2.5, 'Iris-virginica'],
[5.8, 2.7, 5.1, 1.9, 'Iris-virginica'],
[7.1, 3.0, 5.9, 2.1, 'Iris-virginica'],
[6.3, 2.9, 5.6, 1.8, 'Iris-virginica'],
[6.5, 3.0, 5.8, 2.2, 'Iris-virginica'],
[7.6, 3.0, 6.6, 2.1, 'Iris-virginica'],
[4.9, 2.5, 4.5, 1.7, 'Iris-virginica'],
[7.3, 2.9, 6.3, 1.8, 'Iris-virginica'],
[6.7, 2.5, 5.8, 1.8, 'Iris-virginica'],
[7.2, 3.6, 6.1, 2.5, 'Iris-virginica'],
[6.5, 3.2, 5.1, 2.0, 'Iris-virginica'],
[6.4, 2.7, 5.3, 1.9, 'Iris-virginica'],
[6.8, 3.0, 5.5, 2.1, 'Iris-virginica'],
[5.7, 2.5, 5.0, 2.0, 'Iris-virginica'],
[5.8, 2.8, 5.1, 2.4, 'Iris-virginica'],
[6.4, 3.2, 5.3, 2.3, 'Iris-virginica'],
[6.5, 3.0, 5.5, 1.8, 'Iris-virginica'],
[7.7, 3.8, 6.7, 2.2, 'Iris-virginica'],
[7.7, 2.6, 6.9, 2.3, 'Iris-virginica'],
[6.0, 2.2, 5.0, 1.5, 'Iris-virginica'],
[6.9, 3.2, 5.7, 2.3, 'Iris-virginica'],
[5.6, 2.8, 4.9, 2.0, 'Iris-virginica'],
[7.7, 2.8, 6.7, 2.0, 'Iris-virginica'],
[6.3, 2.7, 4.9, 1.8, 'Iris-virginica'],
[6.7, 3.3, 5.7, 2.1, 'Iris-virginica'],
[7.2, 3.2, 6.0, 1.8, 'Iris-virginica'],
[6.2, 2.8, 4.8, 1.8, 'Iris-virginica'],
[6.1, 3.0, 4.9, 1.8, 'Iris-virginica'],
[6.4, 2.8, 5.6, 2.1, 'Iris-virginica'],
[7.2, 3.0, 5.8, 1.6, 'Iris-virginica'],
[7.4, 2.8, 6.1, 1.9, 'Iris-virginica'],
[7.9, 3.8, 6.4, 2.0, 'Iris-virginica'],
[6.4, 2.8, 5.6, 2.2, 'Iris-virginica'],
[6.3, 2.8, 5.1, 1.5, 'Iris-virginica'],
[6.1, 2.6, 5.6, 1.4, 'Iris-virginica'],
[7.7, 3.0, 6.1, 2.3, 'Iris-virginica'],
[6.3, 3.4, 5.6, 2.4, 'Iris-virginica'],
[6.4, 3.1, 5.5, 1.8, 'Iris-virginica'],
[6.0, 3.0, 4.8, 1.8, 'Iris-virginica'],
[6.9, 3.1, 5.4, 2.1, 'Iris-virginica'],
[6.7, 3.1, 5.6, 2.4, 'Iris-virginica'],
[6.9, 3.1, 5.1, 2.3, 'Iris-virginica'],
[5.8, 2.7, 5.1, 1.9, 'Iris-virginica'],
[6.8, 3.2, 5.9, 2.3, 'Iris-virginica'],
[6.7, 3.3, 5.7, 2.5, 'Iris-virginica'],
[6.7, 3.0, 5.2, 2.3, 'Iris-virginica'],
[6.3, 2.5, 5.0, 1.9, 'Iris-virginica'],
[6.5, 3.0, 5.2, 2.0, 'Iris-virginica'],
[6.2, 3.4, 5.4, 2.3, 'Iris-virginica'],
[5.9, 3.0, 5.1, 1.8, 'Iris-virginica']], dtype=object)
```

From this matrix it is clear that our X is contained in the first 4 columns of `df`, whereas our y is the 5th column. We can slice this numpy matrix as follows:

```
In [4]: X = df.values[:, :4].astype(float) # all rows (:), columns 0 -> 3 (:4)
        y = df.values[:, 4] # all rows (:), 4th column (4)
        X.shape, y.shape

Out[4]: ((150, 4), (150,))
```

Indeed, the shapes of the extracted arrays are in accordance with the expectations.

Notice how we are changing the type of X to float. This is because the original matrix (`df.values`) uses a single type, "object", for all values. Numpy arrays support single types only (for efficiency reasons) and, since the "label" column is a string (object), the "object" type is used to represent the entire `df.values` .

We now need to extract a training set (X_train, y_train) and a test set (X_test, y_test). An important property is that there cannot be any point shared between the two (i.e. the two sets cannot overlap). This is because we need to evaluate the performance of our classification model on data that has never seen before by our model (i.e. data that is not in the training set). If this is not the case, the model would have an unfair advantage and we would over-estimate the model's capability to generalize to new data.

To guarantee this property, we can use a boolean mask to select only a portion of the dataset. Then, using the negation operator, we can "flip" the mask so as to select all remaining values. The following code snippet shows how this works on a toy example.

```
In [5]: a = np.array([0, 1, 2, 3, 4])
        mask = np.array([True, True, True, False, False])
        print("mask", mask)
        print("~mask", ~mask)
        print("a[mask]", a[mask])
        print("a[~mask]", a[~mask])

        mask [ True  True  True False False]
        ~mask [False False False  True  True]
        a[mask] [0 1 2]
        a[~mask] [3 4]
```

Notice how we use `mask` to extract the first three values of `a` : we set to True the positions of the values we are interested in extracting. Then, using the ~ operator, we negate `mask` . With this approach, we extract all values that were not previously selected.

We can apply this approach to our dataset, by first selecting X_train with a mask and then flipping the mask to select all remaining values (to put in X_test).

Our mask will need to contain 150 values: 120 of which (80% of 150) will be true (i.e. will be placed in X_test), whereas the remaining 20% will be set to false (X_test). The 80/20 ratio is a common one for train/test splits. You typically want a larger proportion of data for your training set and a smaller one for the testing (other common splits could be 60/40, 75/25 -- depending on the availability of data).

Clearly, we will not write a mask of 150 values. We can use the "repetition" operator ("") *in python (e.g. `[1]* 3 = [1,1,1]` `). We can repeat the True value 120 times, and the False value 30 times -- then concatenate the two ("+" operator) (e.g.` [1,2,3] + [4,5,6] = [1,2,3,4,5,6]`).

```
In [6]: mask = np.array([True] * 120 + [False] * 30)
        mask
```

```
Out[6]: array([ True,  True,  True,  True,  True,  True,  True,  True,  True,
                True,  True,  True,  True,  True,  True,  True,  True,  True,
                True,  True,  True,  True,  True,  True,  True,  True,  True,
                True,  True,  True,  True,  True,  True,  True,  True,  True,
                True,  True,  True,  True,  True,  True,  True,  True,  True,
                True,  True,  True,  True,  True,  True,  True,  True,  True,
                True,  True,  True,  True,  True,  True,  True,  True,  True,
                True,  True,  True,  True,  True,  True,  True,  True,  True,
                True,  True,  True,  True,  True,  True,  True,  True,  True,
                True,  True,  True,  True,  True,  True,  True,  True,  True,
                True,  True,  True,  True,  True,  True,  True,  True,  True,
                True,  True,  True,  True,  True,  True,  True,  True,  True,
                True,  True,  True,  True,  True,  True,  True,  True,  True,
                True,  True,  True, False, False, False, False, False, False,
               False, False, False, False, False, False, False, False, False,
               False, False, False, False, False, False, False, False, False,
               False, False, False, False, False, False])
```

If we use this mask, though, we will select the frist 120 values of X as our training set, and the last 30 values as test. Since, however, our data is ordered by label, we would get all setosa and all versicolor points (+ 20 virginica points) in our training set and 30 for our test set. This is obviously not desirable. Instead, we can shuffle `mask` so that the selected points will be randomly distributed.

`np.random.shuffle()` shuffles an array in-place. By calling it we can get the shuffled version of mask we are interested in.

```
In [7]: np.random.shuffle(mask)
        mask
```

```
Out[7]: array([ True, False,  True,  True, False,  True,  True,  True,  True,
                True,  True, False, False,  True,  True,  True,  True,  True,
                True,  True,  True,  True,  True, False,  True,  True,  True,
                True, False,  True,  True, False, False,  True,  True,  True,
               False, False,  True,  True, False,  True, False,  True,  True,
                True,  True,  True,  True,  True, False,  True,  True,  True,
                True,  True,  True,  True,  True,  True,  True,  True,  True,
                True,  True,  True, False,  True,  True,  True,  True,  True,
                True,  True,  True,  True,  True, False,  True, False,  True,
                True,  True,  True,  True,  True,  True,  True,  True,  True,
                True, False,  True,  True,  True,  True,  True,  True,  True,
                True, False,  True, False, False,  True, False, False, False,
                True,  True,  True,  True,  True,  True,  True,  True,  True,
                True,  True,  True,  True, False,  True,  True,  True,  True,
                True,  True,  True,  True,  True,  True,  True,  True,  True,
               False,  True, False,  True,  True, False,  True, False, False,
                True,  True,  True,  True, False,  True])
```

Now, we can extract our X_train and X_test, as well as y_train and y_test (notice how we can leverage the same mask for both X and y).

```
In [8]: X_train = X[mask]
        X_test = X[~mask]

        y_train = y[mask]
        y_test = y[~mask]
        (X_train.shape, X_test.shape, y_train.shape, y_test.shape)
```

```
Out[8]: ((120, 4), (30, 4), (120,), (30,))
```

To make sure that we are approximately retaining the same proportion of labels in the training and test set, we can count how many points of each class we are using for training and for testing. To this end, we can use `Counter` (which takes an iterable as input and returns a dictionary where keys are elements of the input list and values are the number of occurences for each element).

```
In [9]: from collections import Counter
        Counter(y_train), Counter(y_test)
```

```
Out[9]: (Counter({'Iris-setosa': 38, 'Iris-versicolor': 45, 'Iris-virginica': 37}),
         Counter({'Iris-setosa': 12, 'Iris-versicolor': 5, 'Iris-virginica': 13}))
```

We can verify that, to some extent, our random sampling has maintained some balance across classes (in later labs you will see how you can use stratification to maintain a better balance among classes).

Finally, splitting a dataset into a training and a test set is an operation you will often need to do. In this lab, we have seen how it can be done from scratch. In later exercises, you will leverage existing functions (e.g. from scikit-learn) to achieve the same result more easily. In other words, you will not need to reinvent the wheel every time, just this one time!

Exercise 4
In this exercise, we will start building the `KNearestNeighbors` class. We already have a skeleton to start from, which is as follows:

In [10]:
```
class KNearestNeighbors:
    def __init__(self, k, distance_metric="euclidean"):
        self.k = k
        self.distance_metric = distance_metric

    def fit(self, X, y):
        pass

    def predict(self, X):
        pass
```

For the KNN algorithm, the "fit" part is particularly straightforward: since all KNN does is computing the distance between training points and test points, the "fit" step will only need to store the training set for later use.

Notice how the syntax used for this class (which is also the syntax used for all sklearn models) does not distinguish between train and test arrays (i.e. it always uses X and y). The context in which does arrays are used should be enough to understand whether we are referring to a training or a test set. For this solution, though, we will disambiguate between training and test.

In [11]:
```
class KNearestNeighbors:
    def __init__(self, k, distance_metric="euclidean"):
        self.k = k
        self.distance_metric = distance_metric

    def fit(self, X_train, y_train):
        self.X_train = X_train
        self.y_train = y_train

    def predict(self, X_test):
        pass
```

Exercise 5
We now need to implement three different distances: euclidean, cosine and Manhattan. We will then need to compute these distances between all points in the training set and all points in the test set. Since distances are scalar values, the output should be a distance matrix of M rows and N columns, where M is the number of points in our training set, and N is the number of points in our test set (for the Iris case, therefore, this will be a 120x30 matrix).

The "naive" approach (i.e. an approach that does not leverage numpy) would be to iterate over all points in X_train and over each point in X_test, and compute the distance between each possible pair (and store it in a matrix). The following is an example:

In [12]:
```
def naive_distance(X_train, X_test, distance):
    distance_matrix = np.zeros((X_train.shape[0], X_test.shape[0]))
    for i in range(X_train.shape[0]):
        for j in range(X_test.shape[0]):
            distance_matrix[i,j] = distance(X_train[i], X_test[j])
    return distance_matrix
```

Where `distance(a,b)` is a function that computes the distance between two n-dimensional points. However, we can do better using Numpy. We will now approach each of the three distances separately.

Euclidean distance Given two vectors $p, q \in \mathbb{R}^n$, we can compute their euclidean distance as $\sqrt{\sum_i (p_i - q_i)^2}$.

We can optimize computing $p_i - q_i$ by leveraging operations between numpy arrays (`p-q`). We can then square the result element-wise (`(p-q)**2`), sum the elements ((`((p-q)**2).sum()`) and compute the square root of the result (`((p-q)**2).sum()**2`). The following functions implement the euclidean distance without and with numpy.

In [13]:
```
def euclidean_non_numpy(p, q):
    cumul = 0
    for i in range(len(p)):
        cumul += (p[i] - q[i])**2
    return cumul ** 0.5

def euclidean_numpy(p, q):
    return ((p-q)**2).sum()**.5
```

We can now use the timeit function to compute the distance between two random vectors:

In [14]:
```
from timeit import timeit

# p and q are 200-dimensional random vectors
p = np.random.random(200)
q = np.random.random(200)

time_nnp = timeit(lambda: euclidean_non_numpy(p,q), number=10000)
time_np = timeit(lambda: euclidean_numpy(p,q), number=10000)
print("euclidean_non_numpy", time_nnp)
print("euclidean_numpy", time_np)
print("Ratio", time_nnp/time_np)
```

```
euclidean_non_numpy 1.7033853250000002
euclidean_numpy 0.05013339499999958
Ratio 33.97705910401668
```

Indeed, the numpy-based version is approximately 30x faster than the "cumulative" version.

We could now plug the `euclidean_numpy` function into `naive_distance` and compute the distance matrix. However, we have just shown how more efficient numpy can be with respect to python's for loops. So, instead of using those two nested loops fo `naive_distance` , we can probably develop a more efficient solution.

First, we need to compute the difference between all X_train points and X_test points. `X_train - X_test` is not as valid solution, from both a shape-wise and a logical perspective. Instead we need, for each of the M points in X_train, to compute N subtractions (one for each of the N points in X_test).

Ideally, therefore, our broadcasted training set should have shape (M, N, n) (we replicate each of the M points N times). Since we can leverage broadcasting, we can have a training set with shape (M, 1, n) be broadcasted when subtracting the (N, n) matrix from it (indeed, those two shape are compatible and broadcasted to shape (M, N, n).

In [15]:
```
X_train_reshaped = np.expand_dims(X_train, 1)
X_train_reshaped.shape
```

Out[15]:
```
(120, 1, 4)
```

From this, we can subtract X_test and obtain the (M, N, n) matrix we are loo:

In [16]:
```
X_diff = X_train_reshaped - X_test
X_diff.shape
```
Out[16]:
```
(120, 30, 4)
```

On the third dimension we have the difference across all dimensions. We can square those values (to obtain the squared distance) and sum them (along the correct axis). Then, we can take the (element-wise) square root of the matrix: this will be our distance matrix.

In [17]:
```
dist_matrix = ((X_diff**2).sum(axis=2))**.5
dist_matrix.shape
```
Out[17]:
```
(120, 30)
```

To make sure that the values are indeed distances, we can take the i-th X_train element and j-th X_test element and make sure that their distance (as computed by euclidean_numpy) is the same as the (i,j) position in `dist_matrix` .

In [18]:
```
i = 25
j = 14
dist_matrix[i,j], euclidean_numpy(X_train[i], X_test[j])
```
Out[18]:
```
(4.253234063627348, 4.253234063627348)
```

That looks about right. Now, for a performance comparison, we can compute two random X_train and X_test and test how fast `naive_distance` and `numpy_distance` are.

In [19]:
```
def euclidean(X_train, X_test):
    X_train_reshaped = np.expand_dims(X_train, 1)
    X_diff = X_train_reshaped - X_test
    dist_matrix = ((X_diff**2).sum(axis=2))**.5
    return dist_matrix

X_tr = np.random.random((120, 4))
X_te = np.random.random((30, 4))

time_naive = timeit(lambda: naive_distance(X_tr, X_te, euclidean_numpy), num
ber=100)
time_numpy = timeit(lambda: euclidean(X_tr, X_te), number=100)
print("Naive", time_naive)
print("Numpy", time_numpy)
print("Ratio", time_naive / time_numpy)
```
```
Naive 1.7557359340000005
Numpy 0.011007946999999518
Ratio 159.4971282111076
```

I rest my case.

> Note that we can compute the euclidean distance as square root of the dot product of the difference vector with itself: $\sqrt{(p-q)^T(p-q)}$. This makes the computations faster still, but introduces some additional problems in obtaining the desired output. As a further deep dive into numpy, you may wish to explore this option as well!

Cosine distance For the cosine distance, the situation is significantly easier. Given a vector p and a vector q, we know that the dot product between the two corresponds to $||p||_2||q||_2 cos\theta$, where θ is the angle between the two vector, whose absolute value is referred to as the cosine similarity. The cosine distance is 1 - cosine similarity.

We can build the dot product of each vector of X_train with each vector of X_test by computing the matrix multiplication of the two (with X_test transposed).

(make sure you understand the rationale behind the previous statement, as well as the reason why we need to transpose X_test).

```
In [20]: dot_prods = X_train @ X_test.T # the @ operator is a short-hand for the matr
         ix multiplication function
         dot_prods.shape

Out[20]: (120, 30)
```

We know that we need to normalize the previous result by the norms of the vectors. We can leverage broadcasting to this end (by first dividing by a column vector with all norms of X_train points, then dividing by a row vector with the norms of X_test points).

```
In [21]: X_train_norm = ((X_train**2).sum(axis=1)**.5).reshape(-1,1)
         X_test_norm = ((X_test**2).sum(axis=1)**.5).T
```

```
In [22]: dist_matrix = 1 - abs(dot_prods / X_train_norm.reshape(-1,1) / X_test_norm)
```

Note that, when computing `X_train_norm` , we need to reshape it into a column vector (`reshape(-1,1`). This is needed because the sum() collapses a dimension, thus making the result 1-dimensional (i.e. a row vector). By reshaping it, we convert the row into a column.

We can put together the previous pieces and build a `cosine` function that that computes the desired matrix.

```
In [23]: def cosine(X_train, X_test):
             X_train_norm = ((X_train**2).sum(axis=1)**.5).reshape(-1,1)
             X_test_norm = ((X_test**2).sum(axis=1)**.5)
             dot_prods = X_train @ X_test.T
             dist_matrix = 1 - abs(dot_prods / X_train_norm.reshape(-1,1) / X_test_no
         rm)
             return dist_matrix
```

Manhattan distance
Based on the solution for the euclidean distance, we can easily work out a function that computes the Manhattan distance. Instead of squaring the values and taking the square root, we can compute the absolute value of the differences.

```
In [24]: def manhattan(X_train, X_test):
             X_train_reshaped = np.expand_dims(X_train, 1)
             X_diff = X_train_reshaped - X_test
             dist_matrix = abs(X_diff).sum(axis=2)
             return dist_matrix
```

```
In [25]: manhattan(X_train, X_test)
```

```
Out[25]: array([[0.7, 0.2, 0.6, ..., 6.9, 8.6, 7.3],
                [0.5, 0.8, 0.6, ..., 7.1, 8.8, 7.9],
                [0.5, 1. , 0.6, ..., 6.9, 8.8, 7.9],
                ...,
                [7.2, 7.7, 7.5, ..., 0.8, 2.5, 1.8],
                [7.2, 7.7, 7.5, ..., 1.2, 1.5, 1.2],
                [6.3, 6.8, 6.6, ..., 0.5, 2.4, 1.5]])
```

We can now embed the various distance functions we built into our class.

```
In [26]: class KNearestNeighbors:
             def __init__(self, k, distance_metric="euclidean"):
                 self.k = k
                 self.distance_metric = distance_metric

             def fit(self, X_train, y_train):
                 self.X_train = X_train
                 self.y_train = y_train

                 self.X_train_reshaped = np.expand_dims(self.X_train, 1)
                 self.X_train_norm = ((self.X_train**2).sum(axis=1)**.5).reshape(-1,
         1)

             def _euclidean(self, X_test):
                 X_diff = self.X_train_reshaped - X_test
                 dist_matrix = ((X_diff**2).sum(axis=2))**.5
                 return dist_matrix

             def _cosine(self, X_test):
                 X_test_norm = ((X_test**2).sum(axis=1)**.5)
                 dot_prods = X_train @ X_test.T
                 dist_matrix = 1 - abs(dot_prods / self.X_train_norm.reshape(-1,1) /
         X_test_norm)
                 return dist_matrix

             def _manhattan(self, X_test):
                 X_diff = self.X_train_reshaped - X_test
                 dist_matrix = abs(X_diff).sum(axis=2)
                 return dist_matrix


             def predict(self, X_test):
                 pass
```

Notice that there are some arrays (`X_train_norm` and `X_train_reshaped`) that can be worked out from X_train alone. We can move those operations into the fit() function: in this way, they will only be computed once (at training time) and can be used for multiple predictions, with no need to recompute them every time.

Exercise 6 We now need to make a prediction for each point in X_test. Before we do this, we need to pass from an MxN matrix (i.e. the distance matrix computed before) to an NxK matrix (i.e. a matrix with the indices of the K nearest neighbors for each point in X_test).

To this end, we can use `argsort` (https://numpy.org/doc/stable/reference/generated/numpy.argsort.html), which returns the indices of the input data in sorted order. In other words, given an unsorted array, argsort returns an index that can be used to access the target array and obtain a sorted version of it. The following is such an example:

In [27]:
```
a = np.array([52, 12, 99, 4])
ndx = np.argsort(a)
print("ndx", ndx)
print("a[ndx]", a[ndx])
```

```
ndx [3 1 0 2]
a[ndx] [ 4 12 52 99]
```

If we select the first K values of argsort (applied on the j-th column of the distance matrix), we will get the indices of the K closest training points to the j-th test point.

Additionally, argsort allows specifying an axis along which the sorting is done. So, given a distance matrix `dist_matrix` , we can do the following:

In [28]:
```
k = 3
knn = dist_matrix.argsort(axis=0)[:k, :].T
knn
```

Out[28]:
```
array([[ 33,  16,  26],
       [ 36,   0,  15],
       [ 23,  35,   2],
       [  7,  26,  16],
       [ 21,  31,  33],
       [ 37,  22,   1],
       [ 33,  14,  37],
       [ 25,  34,  15],
       [ 27,  37,   1],
       [  7,  26,  16],
       [ 13,   0,   1],
       [ 15,  34,   4],
       [ 60,  66,  76],
       [ 68,  56,  46],
       [ 73,  40,  63],
       [ 76,  52,  57],
       [ 77,  63,  50],
       [111,  91,  84],
       [ 88,  89, 117],
       [ 67,  93,  94],
       [ 85,  98, 106],
       [ 84,  83, 111],
       [ 85,  96, 106],
       [ 86, 115,  92],
       [117,  88,  99],
       [ 94,  93, 119],
       [ 97, 115,  86],
       [ 83,  84, 104],
       [ 83,  84,  97],
       [111,  86,  92]])
```

In this matrix, at row j, we will find the K nearest neighbors of th j-th test point.

We can now leverage numpy's indexing to access the actual labels for those nearest neighbors.

```
In [29]: y_train[knn]
```

```
Out[29]: array([['Iris-setosa', 'Iris-setosa', 'Iris-setosa'],
                ['Iris-setosa', 'Iris-setosa', 'Iris-setosa'],
                ['Iris-setosa', 'Iris-setosa', 'Iris-setosa'],
                ['Iris-setosa', 'Iris-setosa', 'Iris-setosa'],
                ['Iris-setosa', 'Iris-setosa', 'Iris-setosa'],
                ['Iris-setosa', 'Iris-setosa', 'Iris-setosa'],
                ['Iris-setosa', 'Iris-setosa', 'Iris-setosa'],
                ['Iris-setosa', 'Iris-setosa', 'Iris-setosa'],
                ['Iris-setosa', 'Iris-setosa', 'Iris-setosa'],
                ['Iris-setosa', 'Iris-setosa', 'Iris-setosa'],
                ['Iris-setosa', 'Iris-setosa', 'Iris-setosa'],
                ['Iris-setosa', 'Iris-setosa', 'Iris-setosa'],
                ['Iris-versicolor', 'Iris-versicolor', 'Iris-versicolor'],
                ['Iris-versicolor', 'Iris-versicolor', 'Iris-versicolor'],
                ['Iris-versicolor', 'Iris-versicolor', 'Iris-versicolor'],
                ['Iris-versicolor', 'Iris-versicolor', 'Iris-versicolor'],
                ['Iris-versicolor', 'Iris-versicolor', 'Iris-versicolor'],
                ['Iris-virginica', 'Iris-virginica', 'Iris-virginica'],
                ['Iris-virginica', 'Iris-virginica', 'Iris-virginica'],
                ['Iris-versicolor', 'Iris-virginica', 'Iris-virginica'],
                ['Iris-virginica', 'Iris-virginica', 'Iris-virginica'],
                ['Iris-virginica', 'Iris-virginica', 'Iris-virginica'],
                ['Iris-virginica', 'Iris-virginica', 'Iris-virginica'],
                ['Iris-virginica', 'Iris-virginica', 'Iris-virginica'],
                ['Iris-virginica', 'Iris-virginica', 'Iris-virginica'],
                ['Iris-virginica', 'Iris-virginica', 'Iris-virginica'],
                ['Iris-virginica', 'Iris-virginica', 'Iris-virginica'],
                ['Iris-virginica', 'Iris-virginica', 'Iris-virginica'],
                ['Iris-virginica', 'Iris-virginica', 'Iris-virginica'],
                ['Iris-virginica', 'Iris-virginica', 'Iris-virginica']],
               dtype=object)
```

For the 20th test point (index 19), for example, we can see that the K=3 nearest neighbors are labelled as `['Iris-versicolor', 'Iris-virginica', 'Iris-virginica']`. With a majority voting, the label assigned to the first test point will therefore be Iris-virginica. We can check `y_test[19]` to see if we got it right.

```
In [30]: y_test[27]
```

```
Out[30]: 'Iris-virginica'
```

> Note that, since we only need to find the K smallest values (not necessarily in a sorted order), we could also use `argpartition` (https://numpy.org/doc/stable/reference/generated/numpy.argpartition.html?highlight=argpartition#numpy.argpartition). For simplicity's sake, we will avoid getting into that aspect as well. Do notice, though, that argpartition has a time complexity of O(N), whereas argsort is O(N*logN).

Now, for each row in the knn matrix, we should do a majority voting to assign a label. A simple solution to this problem is counting the number of times each label appears in each row. For this task, the Counter class comes to mind. We can iterate over each row of `knn` and apply Counter to them.

```
In [31]: list(map(Counter, y_train[knn]))

Out[31]: [Counter({'Iris-setosa': 3}),
          Counter({'Iris-setosa': 3}),
          Counter({'Iris-setosa': 3}),
          Counter({'Iris-setosa': 3}),
          Counter({'Iris-setosa': 3}),
          Counter({'Iris-setosa': 3}),
          Counter({'Iris-setosa': 3}),
          Counter({'Iris-setosa': 3}),
          Counter({'Iris-setosa': 3}),
          Counter({'Iris-setosa': 3}),
          Counter({'Iris-setosa': 3}),
          Counter({'Iris-setosa': 3}),
          Counter({'Iris-versicolor': 3}),
          Counter({'Iris-versicolor': 3}),
          Counter({'Iris-versicolor': 3}),
          Counter({'Iris-versicolor': 3}),
          Counter({'Iris-versicolor': 3}),
          Counter({'Iris-virginica': 3}),
          Counter({'Iris-virginica': 3}),
          Counter({'Iris-versicolor': 1, 'Iris-virginica': 2}),
          Counter({'Iris-virginica': 3}),
          Counter({'Iris-virginica': 3}),
          Counter({'Iris-virginica': 3}),
          Counter({'Iris-virginica': 3}),
          Counter({'Iris-virginica': 3}),
          Counter({'Iris-virginica': 3}),
          Counter({'Iris-virginica': 3}),
          Counter({'Iris-virginica': 3}),
          Counter({'Iris-virginica': 3}),
          Counter({'Iris-virginica': 3})]
```

Notice how we are now using lists and for loops instead of using numpy altearnatives. This is because we are treading with strings (labels) and numpy is typically recommended for operations with numbers. Indeed, one could use apply_along_axis to apply Counter to a specific axis. There have been, however, problems in the past (e.g. truncated strings because of problematic type inferences) (e.g. this open issue (https://github.com/numpy/numpy/issues/8352), which makes the adoption of apply_along_axis not trivial). Other solutions, such as not using Counter altogether, but rather use numpy's `bincount` , once again, require some careful preparative operations (mapping string labels to natural numbers). For simplicity's sake, we have limited these kinds of optimizations and prioritized focusing on other ones. Feel free to further optimize the code.

This shows the votes cast for each label. We can also build a function that directly returns the predicted label.

```
In [32]: def majority_voting(votes):
             count = Counter(votes)
             return count.most_common(1)[0][0] # most_common(n) returns a list with t
         he n most recurring votes (n=1 -> top vote)

         np.array(list(map(majority_voting, y_train[knn])))
         np.array([ majority_voting(y_train[knn][i]) for i in range(len(y_train[kn
         n])) ])
```

```
Out[32]: array(['Iris-setosa', 'Iris-setosa', 'Iris-setosa', 'Iris-setosa',
                'Iris-setosa', 'Iris-setosa', 'Iris-setosa', 'Iris-setosa',
                'Iris-setosa', 'Iris-setosa', 'Iris-setosa', 'Iris-setosa',
                'Iris-versicolor', 'Iris-versicolor', 'Iris-versicolor',
                'Iris-versicolor', 'Iris-versicolor', 'Iris-virginica',
                'Iris-virginica', 'Iris-virginica', 'Iris-virginica',
                'Iris-virginica', 'Iris-virginica', 'Iris-virginica',
                'Iris-virginica', 'Iris-virginica', 'Iris-virginica',
                'Iris-virginica', 'Iris-virginica', 'Iris-virginica'], dtype='<U15')
```

We can now condense all previous steps into our predict() method.

```
In [33]: def _majority_voting(votes):
             count = Counter(votes)
             return count.most_common(1)[0][0] # most_common(n) returns a list with t
         he n most recurring votes (n=1 -> top vote)

         class KNearestNeighbors:
             def __init__(self, k, distance_metric="euclidean"):
                 self.k = k
                 self.distance_metric = distance_metric

             def fit(self, X_train, y_train):
                 self.X_train = X_train
                 self.y_train = y_train

                 self.X_train_reshaped = np.expand_dims(self.X_train, 1)
                 self.X_train_norm = ((self.X_train**2).sum(axis=1)**.5).reshape(-1,
         1)

             def _euclidean(self, X_test):
                 X_diff = self.X_train_reshaped - X_test
                 dist_matrix = ((X_diff**2).sum(axis=2))**.5
                 return dist_matrix

             def _cosine(self, X_test):
                 X_test_norm = ((X_test**2).sum(axis=1)**.5).T
                 dot_prods = X_train @ X_test.T
                 dist_matrix = 1 - abs(dot_prods / self.X_train_norm.reshape(-1,1) /
         X_test_norm)
                 return dist_matrix

             def _manhattan(self, X_test):
                 X_diff = self.X_train_reshaped - X_test
                 dist_matrix = abs(X_diff).sum(axis=2)
                 return dist_matrix

             def predict(self, X_test):
                 if self.distance_metric == "euclidean":
                     dist_matrix = self._euclidean(X_test)
                 elif self.distance_metric == "cosine":
                     dist_matrix = self._cosine(X_test)
                 elif self.distance_metric == "manhattan":
                     dist_matrix = self._manhattan(X_test)
                 else:
                     raise Exception("Unknown distance metric")
                 knn = dist_matrix.argsort(axis=0)[:self.k, :].T
                 y_pred = np.array([ majority_voting(self.y_train[knn][i]) for i in r
         ange(len(self.y_train[knn])) ])
                 return y_pred
```

We can now instantiate our classifier, "train" it and get a prediction out of it!

```
In [34]: knn_model = KNearestNeighbors(3, "cosine")
         knn_model.fit(X_train, y_train)
         y_pred = knn_model.predict(X_test)
         y_pred
```

```
Out[34]: array(['Iris-setosa', 'Iris-setosa', 'Iris-setosa', 'Iris-setosa',
                'Iris-setosa', 'Iris-setosa', 'Iris-setosa', 'Iris-setosa',
                'Iris-setosa', 'Iris-setosa', 'Iris-setosa', 'Iris-setosa',
                'Iris-versicolor', 'Iris-versicolor', 'Iris-versicolor',
                'Iris-versicolor', 'Iris-versicolor', 'Iris-virginica',
                'Iris-virginica', 'Iris-virginica', 'Iris-virginica',
                'Iris-virginica', 'Iris-virginica', 'Iris-virginica',
                'Iris-virginica', 'Iris-virginica', 'Iris-virginica',
                'Iris-virginica', 'Iris-virginica', 'Iris-virginica'], dtype='<U15')
```

Exercise 7 We can now proceed with computing our model's accuracy. We can build an accuracy_score() function that returns the fraction of correctly predicted occurences as follows.

```
In [35]: def accuracy_score(y_true, y_pred):
             return (y_true==y_pred).sum()/len(y_true)
```

This function leverages the fact that booleans (the output of the equality check) are stored as 0's and 1's in numpy (therefore, using sum() on a boolean array returns the number of True values).

Let's now measure the accuracy of our model, comparing `y_test` and `y_pred`.

```
In [36]: accuracy_score(y_test, y_pred)
```

```
Out[36]: 1.0
```

Our model has exceedingly good performance (100% of correct guesses!). This is typically not the case for real-world problems. For this specific case, two factors caused this outstanding performance:

1. Trivial task (the Iris dataset is a so-called toy dataset, given how simple it is to approach)
2. Small dataset (our test set is comprised of 30 samples -- it is not particularly hard to predict 30 samples right, specifically with a simple problem such as this one!)

Exercise 8
This task requires slightly changing the way our predict() function works. Instead of weighting each vote with a standard weight of "1" (as previously done), we should now weight each vote with the inverse of the distance between the training and the test points.

Luckily, our code is well-structured, and this change only slightly impacts the previously written code.

The `_majority_voting()` function should no longer only count frequencies, but rather sum weights. We will therefore add an additional parameter to it, `w` (representing the weight of each of the votes in `votes`).

In [37]:
```python
from collections import defaultdict

def _weighted_majority_voting(votes, weights):
    # we now compute `count` as a sum of weights
    # (no longer through Counter -- which effectively
    # weighted all votes as "1")
    count = defaultdict(lambda: 0)
    for vote, weight in zip(votes, weights):
        count[vote] += weight
    return max(count.items(), key=lambda x: x[1])[0] # return the max value
(use a custom key extractor)
```

Next, we need to modify predict() to pass the weights along with the labels. We can do this by using the already available `knn` matrix. Indeed, by using [`take_along_axis` (https://numpy.org/devdocs/reference/generated/numpy.take_along_axis.html)](https://numpy.org/devdocs/reference/generated/numpy.take_along_axis.html), we can extract the distances corresponding to the neighbors in `knn`. By taking the inverse of those values, we should the weights needed.

But, if we stop for a moment, we might realize something unexpected would happen if we simply take the inverse the distances. Indeed, if we had a point with distance 0 from our target point (i.e., we know a point in our training set that is identical to the one we want to predict), we would get a 0 denominator, with the problems that come with it.

So, instead, we can add a small delta (e.g. 10^-5) to the distances before inverting them. This, along the fact that distances are non-negative, guarantees that we will never have divisions by zero.

In [38]:
```python
weights = 1/(np.take_along_axis(dist_matrix, knn.T, 0)+1e-5)
[ _weighted_majority_voting(y_train[knn][i], weights[:, i]) for i in range(l
en(y_train[knn])) ]
```

Out[38]:
```
['Iris-setosa',
 'Iris-setosa',
 'Iris-setosa',
 'Iris-setosa',
 'Iris-setosa',
 'Iris-setosa',
 'Iris-setosa',
 'Iris-setosa',
 'Iris-setosa',
 'Iris-setosa',
 'Iris-setosa',
 'Iris-setosa',
 'Iris-versicolor',
 'Iris-versicolor',
 'Iris-versicolor',
 'Iris-versicolor',
 'Iris-versicolor',
 'Iris-virginica',
 'Iris-virginica',
 'Iris-virginica',
 'Iris-virginica',
 'Iris-virginica',
 'Iris-virginica',
 'Iris-virginica',
 'Iris-virginica',
 'Iris-virginica',
 'Iris-virginica',
 'Iris-virginica',
 'Iris-virginica',
 'Iris-virginica']
```

Notice that the delta we are adding to each distance should not significantly alter the value. We can verify the average value of distances with the following approach:

```
In [39]: np.take_along_axis(dist_matrix, knn.T, 0).max()

Out[39]: 0.001280191091580396
```

So, taking a value that is 2 orders of magnitude lower than the largest distance should not affect the results significantly (clearly, though, it does affect how much we are weighting the case for distance zero).

We can now put together everything we have come up with and build a new version of KNN (this time with an extra parameter, `weights` .

In [40]:
```
from collections import defaultdict

def _weighted_majority_voting(votes, weights):
    # we now compute `count` as a sum of weights
    # (no longer through Counter -- which effectively
    # weighted all votes as "1")
    count = defaultdict(lambda: 0)
    for vote, weight in zip(votes, weights):
        count[vote] += weight
    return max(count.items(), key=lambda x: x[1])[0] # return the max value
(use a custom key extractor)


def _majority_voting(votes):
    count = Counter(votes)
    return count.most_common(1)[0][0] # most_common(n) returns a list with t
he n most recurring votes (n=1 -> top vote)

class KNearestNeighbors:
    def __init__(self, k, distance_metric="euclidean", weights="uniform"):
        self.k = k
        self.distance_metric = distance_metric
        self.weights = weights

    def fit(self, X_train, y_train):
        self.X_train = X_train
        self.y_train = y_train

        self.X_train_reshaped = np.expand_dims(self.X_train, 1)
        self.X_train_norm = ((self.X_train**2).sum(axis=1)**.5).reshape(-1,
1)

    def _euclidean(self, X_test):
        X_diff = self.X_train_reshaped - X_test
        dist_matrix = ((X_diff**2).sum(axis=2))**.5
        return dist_matrix

    def _cosine(self, X_test):
        X_test_norm = ((X_test**2).sum(axis=1)**.5).T
        dot_prods = X_train @ X_test.T
        dist_matrix = 1 - abs(dot_prods / self.X_train_norm.reshape(-1,1) /
X_test_norm)
        return dist_matrix

    def _manhattan(self, X_test):
        X_diff = self.X_train_reshaped - X_test
        dist_matrix = abs(X_diff).sum(axis=2)
        return dist_matrix

    def predict(self, X_test):
        if self.distance_metric == "euclidean":
            dist_matrix = self._euclidean(X_test)
        elif self.distance_metric == "cosine":
            dist_matrix = self._cosine(X_test)
        elif self.distance_metric == "manhattan":
            dist_matrix = self._manhattan(X_test)
        else:
            raise Exception("Unknown distance metric")

        knn = dist_matrix.argsort(axis=0)[:self.k, :].T
        if self.weights == "uniform":
            y_pred = np.array([ majority_voting(self.y_train[knn][i]) for i
in range(len(self.y_train[knn])) ])
        elif self.weights == "distance":
            weights = 1/(np.take_along_axis(dist_matrix, knn.T, 0)+1e-5)
            y_pred = np.array([ _weighted_majority_voting(y_train[knn][i], w
eights[:, i]) for i in range(len(y_train[knn])) ])
        return y_pred
```

Let's make sure everything is still working as expected.

```
In [41]: knn_model = KNearestNeighbors(3, "euclidean", "uniform")
         knn_model.fit(X_train, y_train)
         y_pred = knn_model.predict(X_test)
         accuracy_score(y_test, y_pred)

Out[41]: 0.9666666666666667
```

We can verify whether KNN is correctly weighting samples with one simple trick. If we use a value of k greater than (or close to) the number of samples in the training set, we can expect the following two behaviors:

- from `weights=uniform` , we should always get the label of the class that has a majority of labels (because we are selecting all labels, hence the winning class will always be the most populous one
- from `weights=distance` , on the other hand, we should be able to limit the "damage" done by this unacceptable value of k, since further away points will be weighted less than closer ones.

Based on the expected result for `uniform` , we can make a prediction of the accuracy on the test set. This will be the fraction of values in `y_test` that belong to most populous class in `y_train` (since KNN should always predict the most common class, and will only get right (and get points when computing the accuracy) those entries that indeed belong to the most common training class.

```
In [42]: list(y_test).count(Counter(y_train).most_common(1)[0][0]) / len(y_test)

Out[42]: 0.16666666666666666
```

In the code above, we first compute the most common value of y_train (using Counter's `most_common()` function we have already seen -- `Counter(y_train).most_common(1)[0][0]` . Then, we compute the number of elements in y_test with that sepcific value (using list's count() function), then compute the fraction of elements in y_test with that value (by dividing by `len(y_test)` .

```
In [43]: knn_model = KNearestNeighbors(120, "euclidean", "uniform")
         knn_model.fit(X_train, y_train)
         y_pred = knn_model.predict(X_test)
         accuracy_score(y_test, y_pred)

Out[43]: 0.16666666666666666
```

As expected. As for the `distance` scheme:

```
In [44]: knn_model = KNearestNeighbors(120, "euclidean", "distance")
         knn_model.fit(X_train, y_train)
         y_pred = knn_model.predict(X_test)
         accuracy_score(y_test, y_pred)

Out[44]: 0.9333333333333333
```

A much better result. This tells us that, in a way, using this weighting scheme makes up for a poorly selected (too high) value of K. Note that, in this trivial case, we are getting perfect results even with an absurdly large value of K. In more reasonable scenarios (i.e. not for toy datasets), you will get better results than using a uniform scheme, but probably not as good as when using a reasonable value of K.

Exercise 9
We can now proceed to loading the MNIST dataset, extract 80% of the data for training and 20% for testing, and see how well KNN fares on a slightly harder problem.

The code required for this exercise is pretty much the same we used throughout the code except for the fact that we now need to sample 100 points per digit. We can do this sampling creating a mask for each of the 10 digits (`y==Digit`) and selecting the first 100 values for each of them (as for `y` , given that we are selecting 100 '0's, 100 '1's, 100 '2's etc., we can simply create an array with 100 '0's, 100 '1's, etc).

In [45]:
```
df = pd.read_csv("mnist_test.csv", header=None)
X = df.values[:, 1:].astype(float) # all cols but the first one (784 cols in
this case)
y = df.values[:, 0] # first column, with the label info

# here, we use vstack and hstack to stack (vertical and horizontally, respec
tively)
# different arrays/lists.
X_100 = np.vstack([ X[y==d][:100] for d in range(10) ])
y_100 = np.hstack([ [d]*100 for d in range(10) ])

# generalized version of the previous code (generates 80/20 split)
mask = np.array([True] * int(len(X_100)*.8) + [False] * (len(X_100)-int(len
(X_100)*.8)))
np.random.shuffle(mask)

X_train = X_100[mask]
X_test = X_100[~mask]

y_train = y_100[mask]
y_test = y_100[~mask]
(X_train.shape, X_test.shape, y_train.shape, y_test.shape)
```

Out[45]:
```
((800, 784), (200, 784), (800,), (200,))
```

In [46]:
```
knn_model = KNearestNeighbors(5, "euclidean", "uniform")
knn_model.fit(X_train, y_train)
y_pred = knn_model.predict(X_test)
accuracy_score(y_test, y_pred)
```

Out[46]:
```
0.85
```

All is well. Notice that we are undersampling our dataset to obtain more reasonable training times. For larger problems, KNN will take (much) longer. It is not a particularly efficient algorithm, since it does not build any meaningful structure that allows for faster computations.

We can run a few more tests (e.g. the "distance" vs "uniform" test with a large K).

In [47]:
```python
knn_model = KNearestNeighbors(800, "euclidean", "uniform")
knn_model.fit(X_train, y_train)
y_pred = knn_model.predict(X_test)
print("uniform", accuracy_score(y_test, y_pred))

knn_model = KNearestNeighbors(800, "euclidean", "distance")
knn_model.fit(X_train, y_train)
y_pred = knn_model.predict(X_test)
print("distance", accuracy_score(y_test, y_pred))
```

```
uniform 0.055
distance 0.535
```

As previously mentioned, this unreasonable value for K leads to more reasonable results when wheighting each vote (though, in this case, we have a significant degradation in performance, compared to the case where K has a more reasonable value).

Exercise 10

Let's now try to find some meaningful configurations of parameters for our KNN classifier.

Since the Iris dataset is so simple, it will well with pretty much any configuration. We will instead focus on MNIST, which poses a slightly more difficult challenge.

When searching for a specific configuration of parameters, you may end up running a "grid search". This means trying out different combinations of parameters selected from lists of possible values (e.g. if `K` can be either of [2, 10, 50] and `weight` one of ["uniform", "distance"], you will try all possible combinations: [2, "uniform"], [2, "distance"], [10, "uniform"], [10, "distance"], [50, "uniform"], [50, "distance"].

The number of configurations does not scale well with the number of parameters being tuned, nor does it scale well with the number of possible values each parameter can have, but this may often be a reasonable approach (if the parameters to be tuned and their values are chosen wisely).

You will run such grid searches in future labs. For the time being, we will analyze how KNN performs as we vary value of K. More specifically, let's try running KNN with values from 3 to 103, with step 10

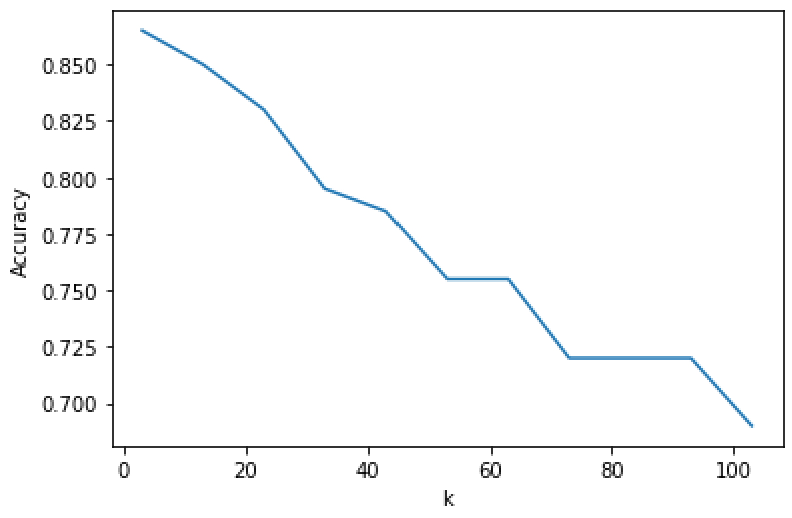
In [48]:
```python
accuracies = []
k_values = list(range(3, 104, 10))
for k in k_values:
    knn_model = KNearestNeighbors(k, "euclidean", "distance")
    knn_model.fit(X_train, y_train)
    y_pred = knn_model.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred)
    accuracies.append(accuracy)
    print(k, accuracy)
```

```
3 0.865
13 0.85
23 0.83
33 0.795
43 0.785
53 0.755
63 0.755
73 0.72
83 0.72
93 0.72
103 0.69
```

We can already see that large values of K are not particularly suitable. Let's plot this result:

```
In [49]: import matplotlib.pyplot as plt
         %matplotlib inline
         plt.plot(k_values, accuracies)
         plt.xlabel("k")
         plt.ylabel("Accuracy")
```

```
Out[49]: Text(0, 0.5, 'Accuracy')
```



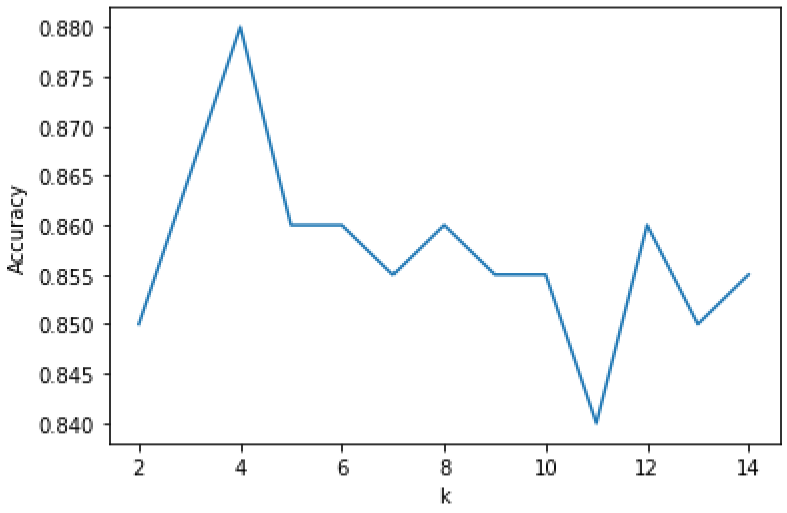
This lets us know that low values of K should be preferrable, for this problem. So let's now try a narrower range of values, for example 2 to 10, with a step of 1.

In [50]:
```
accuracies = []
k_values = list(range(2, 15))
for k in k_values:
    knn_model = KNearestNeighbors(k, "euclidean", "distance")
    knn_model.fit(X_train, y_train)
    y_pred = knn_model.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred)
    accuracies.append(accuracy)
    print(k, accuracy)

plt.plot(k_values, accuracies)
plt.xlabel("k")
plt.ylabel("Accuracy")
```

```
2 0.85
3 0.865
4 0.88
5 0.86
6 0.86
7 0.855
8 0.86
9 0.855
10 0.855
11 0.84
12 0.86
13 0.85
14 0.855
```

Out[50]:
```
Text(0, 0.5, 'Accuracy')
```

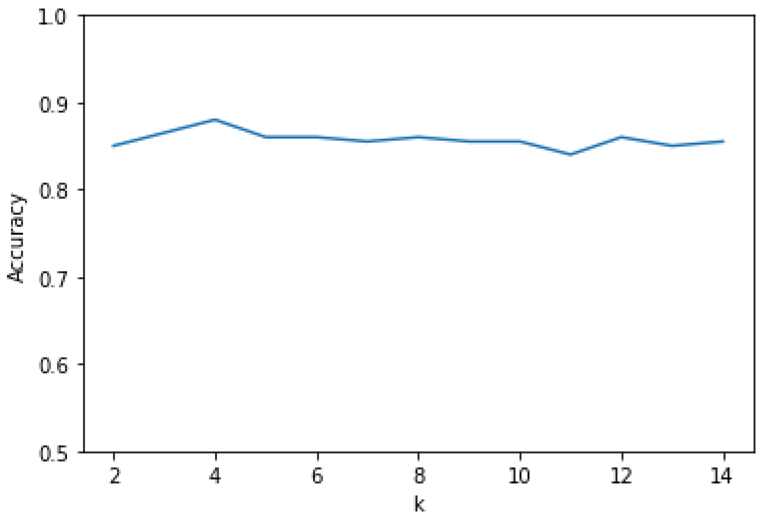


We get the best results for small values of K (approx. $2 \le K \le 6$). Notice how we reached this result without actually trying all values from 2 to 103, but rather running a "coarse" analysis first, identifying promising ranges of values, then running a fine-grained analysis on the range of interest.

Finally, notice how the second plot appears to be quite "bumpy". We might be tempted to take some decisions based on this (e.g. 4 is a much better value for K than 3, or 5). However, also notice that the y axis range is particularly narrow. There actually isn't much of a difference in the values we have. Indeed, if we plot the results on a more meaningful range of values (e.g. 0.5-1), we get the following:

In [51]:
```
plt.plot(k_values, accuracies)
plt.xlabel("k")
plt.ylabel("Accuracy")
plt.ylim(0.5,1)
```

Out[51]:
```
(0.5, 1.0)
```



So, nothing really to worry about.

Finally, we can run some simple grid search. We have identified a reasonable range for K, so we can do the following.

In [52]:
```
for k in range(2, 7):
    for weights in ["uniform", "distance"]:
        for distance in [ "euclidean", "manhattan", "cosine"]:
            knn_model = KNearestNeighbors(k, distance, weights)
            knn_model.fit(X_train, y_train)
            y_pred = knn_model.predict(X_test)
            accuracy = accuracy_score(y_test, y_pred)
            print(accuracy, k, weights, distance)
```

```
0.85 2 uniform euclidean
0.825 2 uniform manhattan
0.84 2 uniform cosine
0.85 2 distance euclidean
0.825 2 distance manhattan
0.84 2 distance cosine
0.865 3 uniform euclidean
0.85 3 uniform manhattan
0.875 3 uniform cosine
0.865 3 distance euclidean
0.85 3 distance manhattan
0.875 3 distance cosine
0.875 4 uniform euclidean
0.86 4 uniform manhattan
0.885 4 uniform cosine
0.88 4 distance euclidean
0.86 4 distance manhattan
0.885 4 distance cosine
0.85 5 uniform euclidean
0.855 5 uniform manhattan
0.88 5 uniform cosine
0.86 5 distance euclidean
0.85 5 distance manhattan
0.875 5 distance cosine
0.865 6 uniform euclidean
0.84 6 uniform manhattan
0.885 6 uniform cosine
0.86 6 distance euclidean
0.84 6 distance manhattan
0.88 6 distance cosine
```

All models perform fairly well. Notice, however, that for values of k in [4, 5, 6] and for the cosine distance, the model performs slightly better (0.88 - 0.885), regardless of the weighting scheme. This should therefore give us a better idea of reasonable parameters for our model.

> Note that, in this case, we are using the test set to draw conclusions on which parameters (called hyperparameters) we should use for our classifier. Therefore, the performance obtained on the test set are not representative of how well our model performs on unseen data (since we are making decisions based on the best results). This set is what we typically refer to as "validation set". An additional test set would be needed to assess how well the trained classifier does on new data.