# ENGI E1006 - Fall 2016
# Homework 5: KNN on the IRIS flower dataset

**Assignment due on 12/04/2016 at 11:59PM**
**135 points + 15 bonus**

**Academic Honesty Policy:**
You are expected to submit **your own work**. Assignments are to be completed individually, NOT in groups. No collaboration is permitted. Please do not include any of your code snippets or algorithms in public Piazza posts. You cannot not use solutions from any external source. If you have any questions at all about this policy please ask the course staff before submitting an assignment.

We'll have one main Piazza post for the homework overall, as well as more specific posts for each component of the KNN classifier.
General questions should be asked as follow-up discussions beneath the main Piazza post:
https://piazza.com/class/isq8q3q8iev6dp?cid=175

Please note that we will be modifying this main Piazza post with clarifications on the homework where necessary, and you will be responsible for adhering to them.

If you have a question, please post a follow-up discussion *on the appropriate post.* Any relevant discussions will be incorporated as a clarification in the main post.

**For all problems:**

- **A skeleton of each function has been provided for you on Canvas. DO NOT modify the function signatures or return variables for any reason. Your job is to complete the body of each function. We've included test cases along with expected output so you can check your solutions. These test cases are under the line:**
  ```
  if __name__ == "__main__":
  ```
  **You can modify these test cases within the skeleton code.**
  **NOTE: you may have to add import statements to the skeleton code.**

- **Use Python 3.5 for all Python parts. To check your Python version, enter "python --version" (no quotes) in a terminal.**

- **Make a comment at the top of each file explaining the high-level logic of your code. Make in-line comments detailing specific decisions and logic where necessary.**

- **You will be graded on your code style. Please refer to the Python style guide available on Courseworks in the resources folder.**

- **Download the skeletons from Canvas and <u>keep names the same</u>:**

- **calculate_error_rate.py**
- **create_data.py**
- **euclidean_distance.py**
- **find_k_nearest_neighbors.py**
- **integerize_labels.py**
- **knn.py**
- **main.py**
- **majority_vote.py**
- **plot_data.py**
- **split.py**
- **weighted_knn.py**
- **weighted_majority_vote.py**

**For submitting, put all files into a single folder named UNI_hw5. Compress your UNI_hw5 folder into a zip file named UNI_hw5.zip (eg. abc1234_hw5.zip). Make sure to use an underline, NOT a hyphen. Submit that zip file via courseworks.**

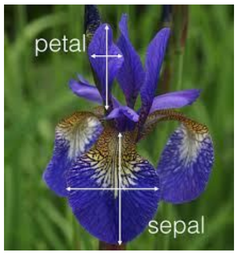
Recall from the class lecture the k nearest neighbor classification algorithm. In this assignment, you will implement the KNN algorithm in python and test it on a real dataset.
The dataset you will use is a well-known benchmark dataset in the pattern recognition literature.

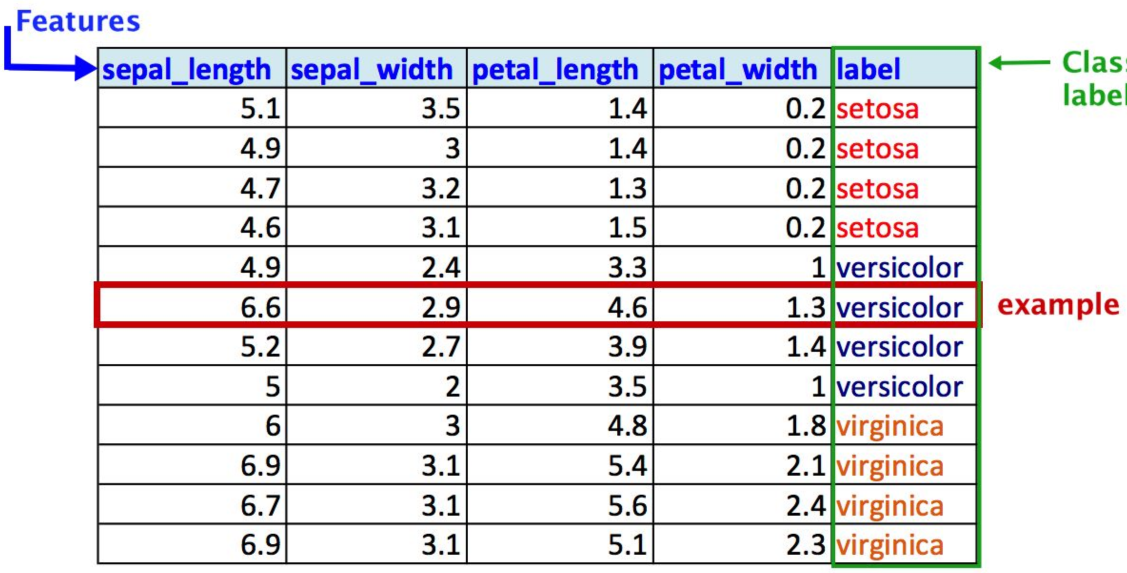
Iris Setosa

Iris Virginica

Iris Versicolor

The data set contains 3 classes of 50 examples each, where each class refers to a type of iris flower. We want to predict the class of the iris plant based on its measures in centimeters of: sepal length, sepal width, petal length and petal width. The question is: given a flower measurement, can we predict its species?




Features
Class label
example

| sepal_length | sepal_width | petal_length | petal_width | label |
|---|---|---|---|---|
| 5.1 | 3.5 | 1.4 | 0.2 | setosa |
| 4.9 | 3 | 1.4 | 0.2 | setosa |
| 4.7 | 3.2 | 1.3 | 0.2 | setosa |
| 4.6 | 3.1 | 1.5 | 0.2 | setosa |
| 4.9 | 2.4 | 3.3 | 1 | versicolor |
| 6.6 | 2.9 | 4.6 | 1.3 | versicolor |
| 5.2 | 2.7 | 3.9 | 1.4 | versicolor |
| 5 | 2 | 3.5 | 1 | versicolor |
| 6 | 3 | 4.8 | 1.8 | virginica |
| 6.9 | 3.1 | 5.4 | 2.1 | virginica |
| 6.7 | 3.1 | 5.6 | 2.4 | virginica |
| 6.9 | 3.1 | 5.1 | 2.3 | virginica |

Your mission is to build a KNN classifier that can predict the class of iris flowers based on their measurements.

Recall, KNN uses the distance between examples to assess how similar they are. The most common distance used is the *Euclidian distance.*

Given two iris flowers $x_i$ and $x_j$: the distance between them is defined as follows:

$$d(x_i, x_j) = \sqrt{\sum_{k=1}^{4} (x_{ik} - x_{jk})^2}$$

$X_{ik}$ denotes the value of the kth feature for the ith example.

For example, given two flowers $x_1$ and $x_2$:

> $x_1$: [74, 21, 58, 24, “versicolor”]
> $x_2$: [82, 21, 13, 98, “virginica”]

The distance between the two flowers is:

$$d(x1, x2) = \sqrt{(74-82)^2 + (21-21)^2 + (58-13)^2 + (24-98)^2} = 86.98$$

Note: the distance between two examples with exactly the same feature values is 0.

Given an unlabeled example $x_{new}$ to classify, which is a flower for which we have the measurements but we don’t know to which species of iris it belongs, KNN will calculate all the distances between $x_{new}$ and all the labeled examples in the dataset. The nearest neighbors are the examples that have the shortest distance to $x_{new}$. The predicted class for $x_{new}$ is simply whatever label is assigned to the majority of $x_{new}$’s k nearest neighbors.

For more details: https://en.wikipedia.org/wiki/**K-nearest_neighbors_**algorithm

Variables that are used throughout this homework (listed here for your convenience)
**m -** the total number of data points (number of rows in the data file)
**n** - the total number of dimensions / features of the data i.e. number of columns (note that the label is not a dimension)
**k** - the number of nearest neighbors to be considered in this algorithm

## I- Data preparation

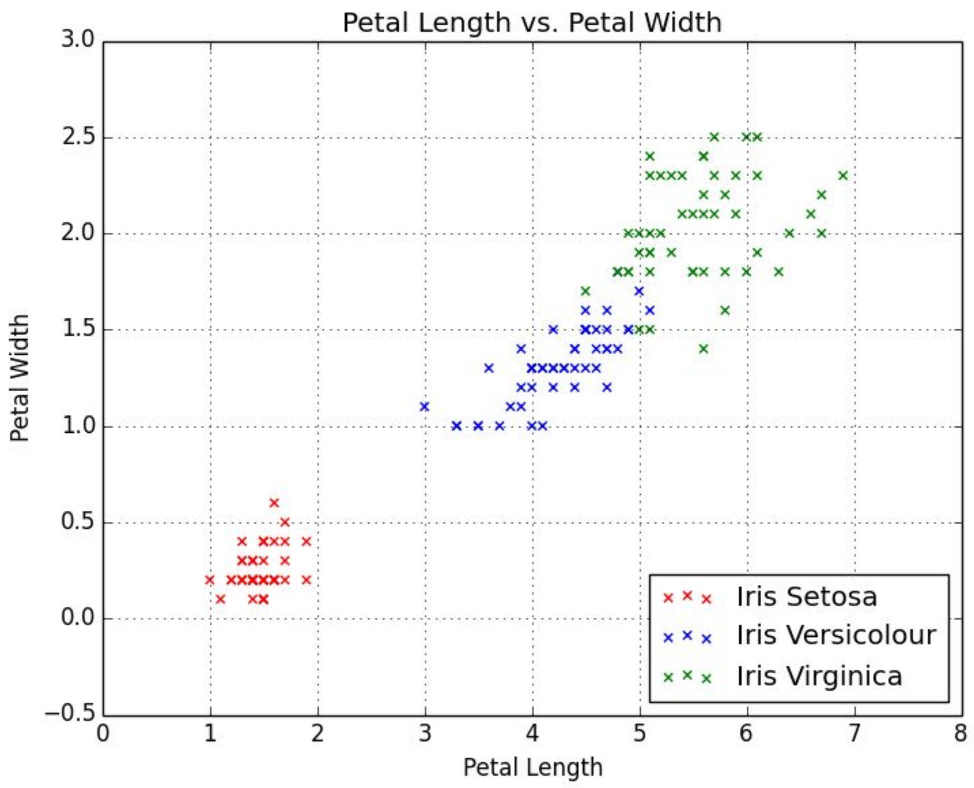
**Download the data from [https://archive.ics.uci.edu/ml/datasets/Iris](https://archive.ics.uci.edu/ml/datasets/Iris)**
Go to Data Folder → iris.data

**1)**
**create_data(input_data_file) -- 10 pts**
Reads in a file named **input_data_file**
Returns a m x (n + 1) numpy matrix **data**, where the first n columns correspond to the n features of the data set: sepal length in cm, sepal width in cm, petal length in cm, and petal width in cm. The last column corresponds to the label of the row.

**2)**
**plot_data(data) -- 20 pts**
**data** is the output of create_data()
Go through every unique way of pairing the four features:

- sepal length vs. sepal width
- sepal length vs. petal length
- sepal length vs. petal width
- sepal width vs. petal length
- sepal width vs. petal width
- petal length vs. petal width

for each one, use matplotlib to create a scatter plot of **data** along those two dimensions. Every chart should have an appropriate plot title, axis title, and legend. Ensure that the axis lengths are enough to display all of the points appropriately. The points in the scatter plot should be clearly colored in accordance with their label. You should produce six plots for each of the dimensional combinations, save them as .png, and name them appropriately (i.e. petal_length_vs_petal_width.png )
For example:


Petal Length vs. Petal Width
Petal Width
Petal Length
3.0
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
0
1
2
3
4
5
6
7
8
Iris Setosa
Iris Versicolour
Iris Virginica

**3)**
**integerize_labels(data) -- 5 pts**
Takes in a m x (n + 1) numpy matrix **data** (which is the result of create_data)
Creates a new m x (n + 1) numpy matrix that is identical to **data** except that all of the labels are converted into unique integers that correspond to those labels.

For example, the input:

```
[
    [32, 61, 18, 95, “setosa”],
    [74, 21, 58, 24, “versicolor”],
    [82, 21, 13, 98, “virginica”],
    [43, 16, 89, 28, “setosa”],
]
```

Would produce the output

```
[
    [32, 61, 18, 95, 0],
    [74, 21, 58, 24, 1],
    [82, 21, 13, 98, 2],
    [43, 16, 89, 28, 0],
]
```

First, it encounters setosa as a label, so it converts it to a 0, since that is the first label it sees. Then it encounters versicolour, which it hasn’t seen before, so it assigns it a new label, which is the value of the previous new label + 1. Then it encounters virginica, which is another new label. The previous new label is 1, which means that virginica should be represented by 2. Finally, it sees setosa again, so it stays consistent and assigns it 0 again.
Returns a tuple (**integerized_data, label_dict),** where **label_dict** is composed of { label_string: label_int} pairs.

**4)**
**split(integerized_data) -- 15 pts**
**integerized_data** is the m x (n + 1) matrix returned from integerize_labels().
Randomly sample 15 instances from each of the three classes in integerized_data, and put them into a 45 x (n + 1) numpy matrix named **test_data** This will be your test set. Put the (m - 45) instances that you didn’t select into a different matrix **train_data**, which will be your training set. The function will return the tuple (**train_data, test_data**).

## II- Make a classification

**1)**
**euclidean_distance(x1, x2) -- 10 pts**
**x1** and **x2** are both 1 x (n + 1) vectors representing an individual data point (i.e. [32, 61, 18, 95, 0] ). Keeping in mind that the first n columns represent dimensions, and that the last represents the label, write a function that returns euclidean **distance** between x1 and x2. Note that while our data set has four dimensions, this function should be able to handle any n dimensional data point, where the last column is the label. Assume that **x1** and **x2** are of the same length.

**2)**
**find_k_nearest_neighbors(test_point, train_data, k) -- 25 pts**
**test_point** is one of the rows from the test_data matrix that is returned from split()
**train_data** is the (m - 45) x n matrix from split()
**k** is an integer
Return a **k** x (n + 1) matrix named **k_nearest_neighbors** that contains **k** nearest neighbors of **test_point** from **train_data.** Use euclidean distance to calculate the distance of each point in **train_data** to **test_point.**

**3)**
**majority_vote(neighbors) -- 10 pts**
**neighbors** is the k x (n + 1) matrix returned by find_k_nearest_neighbors()
Returns an integer representing the majority label of **neighbors.** In the event of a tie, you may break it arbitrarily.

**4)**
**knn(train_data, test_data, k) -- 30 pts**
**train_data** and **test_data** are the output of split(), and **k** is an integer.
For every row in **test_data**, predict its label using **k** nearest neighbors. Create a 45 x 1 (note that this means vertical! ) matrix **predicted_labels** composed of all of these labels, such that every $i^{th}$ row in **test_data** corresponds to the $i^{th}$ predicted label in **predicted_labels**.
Return **predicted_labels**

**III- Evaluation**
**1)**
**calculate_error_rate(predicted_labels, test_data) -- 10 pts**
**predicted_labels** is the 45 x 1 matrix returned by knn().
**test_data** is the 45 x (n + 1) matrix that represents the test set from split()

Calculate the error rate (the proportion of predictions that are incorrect) for the test set. Every index in **predicted_labels** should correspond to the appropriate row in **test**. If the i$^{th}$ predicted label is not equal to the actual i$^{th}$ label in **test,** then that prediction was incorrect (it was an error).
Error rate = (total number of errors)/(total number of predictions)
Return the error rate.

**IV- Bringing it all together**
We have written a main file for you that will tie all of these functions together. Make sure that you do not change any function signatures or filenames or you will have import errors.

Go through the compare_errors() function in main.py and implement all of the commented out lines under the TODOs as you implement the respective functions.
**compare_errors(k_vals, input_data_file)** performs knn on the data in the **input_data_file** and records the classification error for each value of **k** in **k_vals** in the **errors** dict. The **errors** dict is composed of
{
    **k_val:** error_rate
}
pairs.
compare_errors() will return **errors.**

**V- Bonus**
**1)**
**weighted_majority_vote(test_point, neighbor_labels) -- 10 pts**
Write a new majority vote function such that the distance between the neighbors and the example to classify is taken into account. In other words, you should weight the votes depending on the distance of the point that is voting(by the way, this will do a better job than randomly breaking ties).

**2)**
**weighted_knn(train_data, test_data, k) -- 5 pts**
weighted_knn() operates exactly like knn() except that it uses **weighted_majority_vote()** instead of **majority_vote()** to determine the labels for all of the points in **test_data**.

In **compare_errors(),** comment out the lines for performing **weighted_knn()** so that you may compare the weighted errors to the unweighted errors.