Week 4 Tasks

1. knn =>

a)how we can use images directly without extracting the features from it

So to understand this we have to understand:

**How does the k-NN classifier work?**

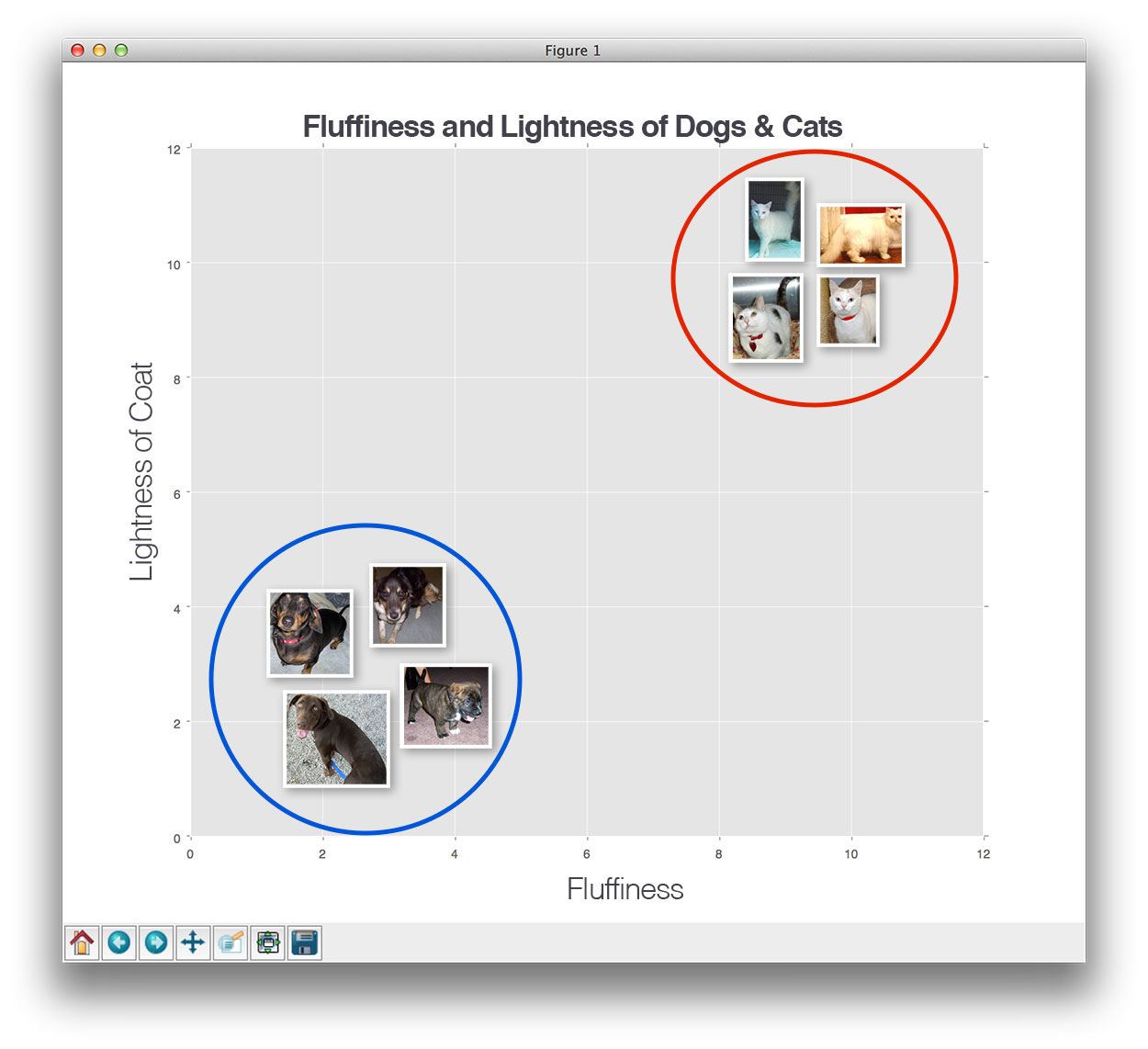
The [k-Nearest Neighbor](https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm) classifier is *by far* the most simple machine learning/image classification algorithm. In fact, it’s *so simple* that it doesn’t actually “learn” anything.

Inside, this algorithm simply relies on the distance between feature vectors, [much like building an image search engine](https://www.pyimagesearch.com/2014/12/01/complete-guide-building-image-search-engine-python-opencv/) — only this time, we have the *labels* associated with each image so we can predict and return an actual *category* for the image.

Simply put, the k-NN algorithm classifies unknown data points by finding the *most common class* among the ***k-closest examples.***Each data point in the *k* closest examples casts a vote and the category with the most votes wins!

Or, in plain english: *“Tell me who your neighbors are, and I’ll tell you who you are”*

To visualize this, take a look at the following toy example where I have plotted the “fluffiness” of animals along the *x-axis* and the lightness of their coat on the *y-axis*:



**Figure 2:** Plotting the fluffiness of animals along the *x-axis* and the lightness of their coat on the *y-axis*.

Here we can see there are two categories of images and that each of the data points within each respective category are grouped relatively close together in an *n-*dimensional space. Our dogs tend to have dark coats which are not very fluffy while our cats have very light coats that are extremely fluffy.

This implies that the distance between two data points in the *red circle* is *much smaller* than the distance between a data point in the *red circle* and a data point in the *blue circle*.

In order to apply the k-nearest Neighbor classification, we need to define a [distance metric or similarity function](https://www.pyimagesearch.com/2014/02/17/building-an-image-search-engine-defining-your-similarity-metric-step-3-of-4/). Common choices include the [Euclidean distance](https://en.wikipedia.org/wiki/Euclidean_distance):

Figure 2: The Euclidean distance.

**Figure 3:** The Euclidean distance.

And the [Manhattan/city block distance](https://en.wikipedia.org/wiki/Taxicab_geometry):

Figure 3: The Manhattan/city block distance.

So, to answer the question we are basically using labels on each images and then sorting them based on the characteristics of that species.

b)loading the ENTIRE dataset to memory => RAM issues

It depends on how much bigger your index is in comparison to the memory. Here are some ideas:

1. Supposing it was tens of times the size of the RAM, try to cluster the data using, for instance, hierarchical clustering trees (implemented in FLANN). Modifying the implementation of the trees so that they keep the branches in memory and save the leaves (the clusters) on the disk. Therefore, the appropriate cluster would have to be loaded each time. This can be optimized in different ways.
2. If it was not that bigger (let's say twice the size of the RAM), separate the dataset in two parts and create one index for each. In that case, finding the nearest neighbor in each dataset and then choosing between them is the most appropriate thing to do.

c)2 hyperparameters used in KNN => 1. Low value of k and high value of k. Oversmoothing?

2. Actual distance metric used: Euclidean distance? Manhattan distance?\

As mentioned above,

In order to apply the k-nearest Neighbor classification, we need to define a [distance metric or similarity function](https://www.pyimagesearch.com/2014/02/17/building-an-image-search-engine-defining-your-similarity-metric-step-3-of-4/). Common choices include the [Euclidean distance](https://en.wikipedia.org/wiki/Euclidean_distance):

Figure 2: The Euclidean distance.

**Figure 3:** The Euclidean distance.

And the [Manhattan/city block distance](https://en.wikipedia.org/wiki/Taxicab_geometry):

Figure 3: The Manhattan/city block distance.

For high dimensional vectors Manhattan works better than the Euclidean distance.  
  
The reason for this is quite simple to explain. Consider the case where we use the l∞l∞ norm that is the Minkowski distance with exponent = infinity. Then the distance is the highest difference between any two dimensions of your vectors. We can see this doesn't make sense in many dimensions as we would be ignoring most of the dimensionality and measuring distance based on a single attribute.   
  
Thus reducing the exponent makes the other features play a bigger role in the distance calculation. The lower the exponent the less relevant a high difference in some given dimension will be.  
  
Something interesting is that distances with exponent <1 might work even better than Manhattan: (||x−y||p)1/p(||x−y||p)1/p with p<1. This distances are curiously not valid metrics as the triangular inequality doesn't hold but they can be used anyway.

In 2d the Euclidean distance measures the shortest distance in the plane, the Manhattan metric the shortest path if you are only allowed to move horizontally or vertically.

So both measure shortest paths but the Euclidean metric does not have any restrictions while the Manhattan metric only allows paths constant in all but one dimension.

Drawbacks :

i)How no actual learning takes place in knn? If it makes mistake, it has no way to "correct" or "improve" itself for later classification.

The model representation for KNN is the entire training dataset. It is as simple as that.

KNN has no model other than storing the entire dataset, so there is no learning required.

The main **disadvantage** of the **KNN algorithm** is that it is a lazy learner, i.e. it does not learn anything from the training data and simply uses the training data itself for classification, hence it can’t increase its accuracy or improve or even correct itself for later classification.

Knn is non-parametric and hence KNN makes no assumptions about the functional form of the problem being solved.

ii)High dimensional data? knn becomes questionable, training time increases with O(n)

It is worth noting that the minimal training phase of KNN comes both at a *memory cost*, since we must store a potentially huge data set, as well as a *computational cost* during test time since classifying a given observation requires a run down of the whole data set. Practically speaking, this is undesirable since we usually want fast responses.

So for a huge dataset as the knn algorithm has to analyse large no. of cases and characteristics of the dataset, therefore the training time increase with O(n).

e)How since NO PARAMETERS are learnt, classification TIME is more. Increases with the training data size => O(n)

As classification time depends on dataset and knn requires a large dataset that’s why it increases with the training data size.

f)Given these cons why study knn? Algorithm is simple. It is easy to understand. And most importantly, it gives us a baseline that we can use to compare neural networks and CNN.

Given these cons still knn is used because of the following:

* + - * Simple to implement
      * Flexible to feature / distance choices
      * Naturally handles multi-class cases
      * Can do well in practice with enough representative data.

g)Introduction to PARAMETERISED LEARNING => Linear Classifier

In the field of [machine learning](https://en.wikipedia.org/wiki/Machine_learning), the goal of [statistical classification](https://en.wikipedia.org/wiki/Statistical_classification) is to use an object's characteristics to identify which class (or group) it belongs to. A **linear classifier** achieves this by making a classification decision based on the value of a [linear combination](https://en.wikipedia.org/wiki/Linear_combination) of the characteristics. An object's characteristics are also known as [feature values](https://en.wikipedia.org/wiki/Features_(pattern_recognition)) and are typically presented to the machine in a vector called a [feature vector](https://en.wikipedia.org/wiki/Feature_vector). Such classifiers work well for practical problems such as [document classification](https://en.wikipedia.org/wiki/Document_classification), and more generally for problems with many variables ([features](https://en.wikipedia.org/wiki/Feature_vector)), reaching accuracy levels comparable to non-linear classifiers while taking less time to train and use.