IAML - Week 6

Antonio León Villares

December 2021

Contents

1 Nearest Neighbours

- kNN uses the k closest points in training to classify a new instance
- kNN can be used for classification and regression
- We can select k by using a validation set
- Since kNN is slow for large datasets, techniques like KD-Trees, Locally-Sensitive Hashing or Inverted Lists can be used to speed the process up

1.1 Overview and Intuition

- **What is the idea behind nearest neighbours classification?**
  - we classify a new instance based on which instances in the training data it is closest to

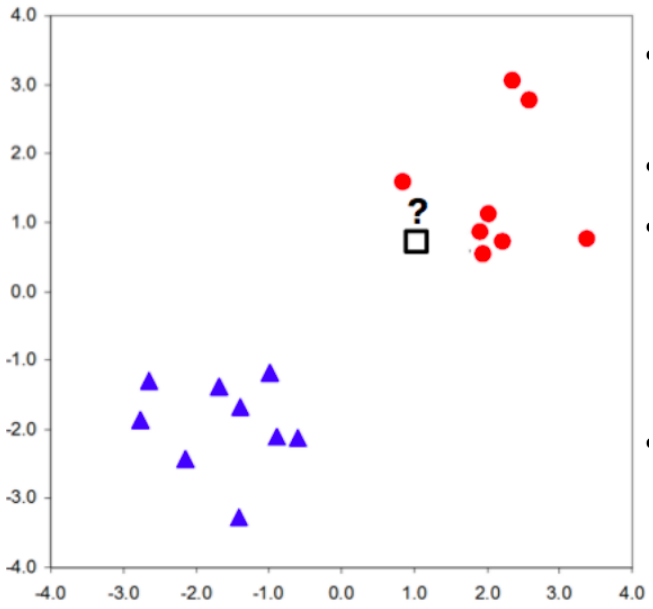


Figure 1: We would classify the unknown instance as "red" since we can see that it is close to training instances which are "red"

  - this is more intuitive than other methods: we don't need to consider prior probabilities of classes; nor do we compute the maximum margin hyperplane to use SVM

- **What is 1NN?**
  - in 1NN, we classify $\underline{x}$ based on the class of its nearest neighbour in the training data, $\underline{x}$'
  - if we employ 1NN, we can partition the data space using a **Voronoi Tesselation** (since classification of a point is just based on which point in the training is closest to it)
  - the **decision boundary**
  - in this case, the decision boundary will just be all the points in the space which are equally distanced from any 2 training examples

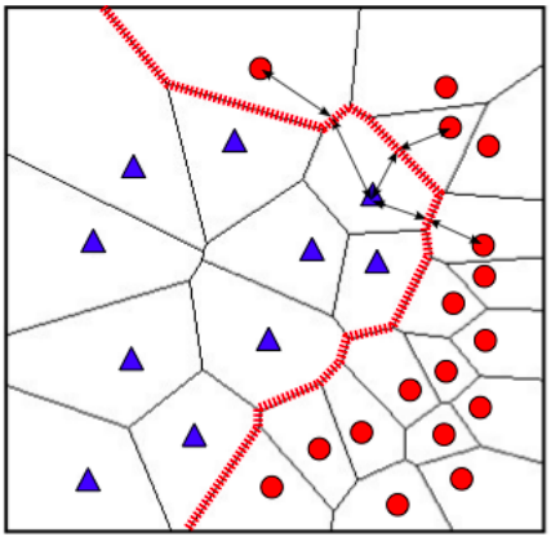
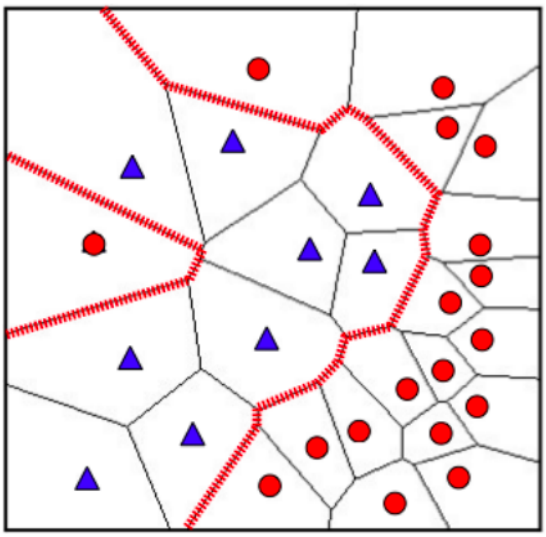


Figure 2: In this case the DB splits the data very nicely. Notice it is much more complex (non-linear) when compared to other classifiers like SVM or NB.

- **How do outliers affect 1NN?**
  - the diagram above shows that 1NN will be highly sensitive to outliers
  - changing one blue dot for a red dot will significantly affect the decision boundary, and entire regions of space will suddenly be classified as red
  - this is not desirable: changing a point by a bit should not drastically affect its classification
  - 1NN is insensitive to class priors: it doesn't know if one class is more frequent than another



- **How can we solve the issues associated to 1NN?**
  - instead of considering the single nearest neighbours, make a more robust decision based on k of its nearest neighbours
  - doing this means that a single circle outlier will be outweighed by many triangles
  - this will also implicitly include class priors (if a class if more requent, it is more likely to be a neighbour)
  - this is known as kNN

1.2 kNN Classification Algorithm

1.2.1 The Algorithm

- **What is the training phase of kNN?**
  - kNN doesn't need to train: it doesn't learn weight, it just has to find distances between a new instance $\underline{x}$ and the set of all training points
- **What is the kNN algorithm?**
  - we consider:
    - $\underline{x}$ (a instance to classify)
    - $\{\underline{x}_i, y_i\}$ (training instance and class label)
  - the algorithm is just:
    1. $\forall \underline{x}_i$, compute $D(\underline{x}, \underline{x}_i)$ (here D is just a distance metric)
    2. select the k instances in $\{\underline{x}_i, y_i\}$ with the smallest $D(\underline{x}, \underline{x}_i)$ (that is, the closest training instances to $\underline{x}$)
    3. do a majority vote with the class labels

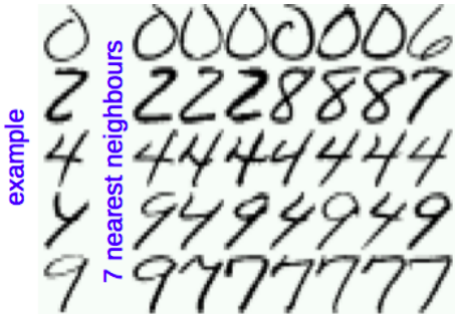
1.2.2 Performance: MNIST Dataset

If we apply kNN to the MNIST handwritten digit dataset, we get impressive accuracy:

- kNN: 95.2%
- SVM: 95.8%
- humans: 97.5%

As a distance metric, we can consider the pixelwise difference between 2 images A, B:

$$D(A, B) = \sqrt{\sum_r \sum_c (A_{rc} - B_{rc})^2)}$$



1.3 kNN Regression Algorithm

1.3.1 The Algorithm

This is very similar as with classification, with the difference that the class labels in the training set $\{\underline{x}, y_i\}$ become real numbers ($y_i \in \mathbb{R}$). Once we have the nearest neighbours (call them $\underline{x}_{i1}, \underline{x}_{i2}, \ldots, \underline{x}_{ik}$), the regression algorithm returns the **average** of the class labels:

$$y^* = \frac{1}{k} \sum_{j=1}^{k} y_{ij}$$

1.3.2 Regression vs Extrapolation

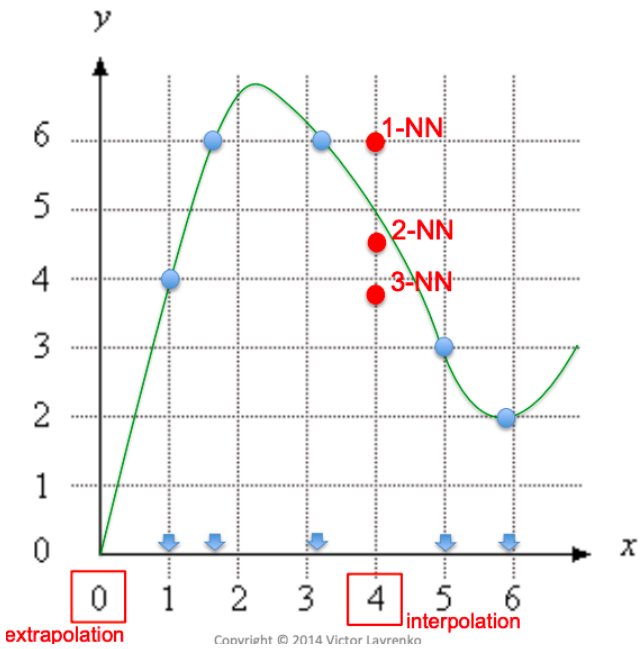


Figure 3: We can use kNN regression, to model the function, taking x as the attribute value, and y as the class label. If we regress $x = 4$ with $k = 1$, we will just predict the y value associated with $x = 3.1$. If we increase $k = 2$, we can consider $x = 3.1, 5$, which gives us a much better prediction. Notice, using $k = 3$ leads to worse prediction. This is one of the first pointers to make about kNN: increasing k need not lead to a better classification (for regression, this means that the model would always output the average value of the ys). If we attempt to use regression on $x = 0$, this is **extrapolation**: $x = 0$ is not within the range of the training data (since it we consider pints with $x \in [1, 5.9]$). In such a case, the prediction will never approach the real value of the function at 0.

1.4 Determining k

- **How does k affect the classification accuracy of kNN?**
  - if k is **small**, the classifier will be extremely sensitive and have high variance (think of 1NN: one small change completely alters the decision boundary)
  - if k is **big**, the classifier will be biased to the most probable class
- **How can we find k?**
  - use a **validation** set, and test how changing k alters the performance.

- the validation set is a subset of the training set
- select the k which leads to the highest (validation) testing accuracy (we iterate over the validation set and classify using a fixed k)
- notice, if we used the full training set, this would just give us $k = 1$ (since this produces 100% accuracy, as a point will always be classified correctly if it considers itself as the nearest neighbour)

1.5 Distance Measures

- **Euclidean Distance**

$$D(\underline{x}, \underline{x}') = \|\underline{x} - \underline{x}'\| = \sqrt{\sum_d |x_d - x'_d|^2}$$

  - symmetric
  - easy to differentiate
  - spherical
  - treats all dimensions equally
  - sensitive to extreme differences in a single attribute

- **Hamming Distance**

$$D(\underline{x}, \underline{x}') = \sum_d \mathcal{X}_{x_d}$$

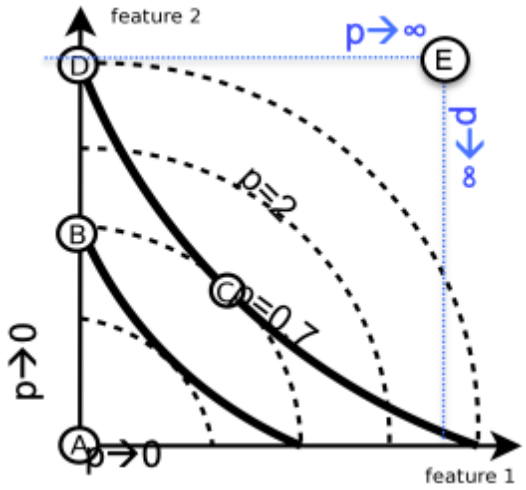
  where $\mathcal{X}_{x_d} = 1$ if $x_d == x'_d$ and 0 otherwise.

  - distance measure for categorical attributes
  - gives the number of attributes in which $\underline{x}$ and $\underline{x}'$ differ

- **Minkowski Distance**

$$D_p(\underline{x}, \underline{x}') = \sqrt[p]{\sum_d |x_d - x'_d|^p}$$

  - also known as **p-norm**
  - $p = 2$ is **Euclidean** distance
  - $p = 1$ is **Manhattan** distance
  - $p = 0$ number of non-zero distances
  - $p = \infty$ is $\max_d |x_d - x'_d|$
  - as $p \to 0$, the p-norm becomes smaller if $\underline{x}$ and $\underline{x}'$ differ in less attributes



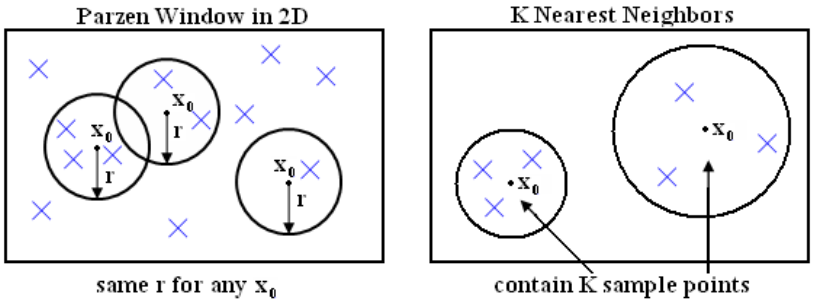
1.6 Issues with kNN

- **What are the 2 issues associated with kNN?**
  - a given k can lead to a **tie** in the class of the neighbours, in which case a class can't be selected by majority vote (for example, a point can have 2 neighbours that are red, 2 neighbours that are blue and 2 neighbours that are green)
  - the instance $\underline{x}$ can have a mising attribute value, in which case we can't compute the distance
- **How can we resolve ties in kNN?**
  - if we are performing **binary** classification, we can just pick an odd k (this won't work for multi-class classification)
  - more generally:
    - pick randomly
    - pick class with greatest prior
    - pick the nearest of the nearest neighbours
- **How can we fill in missing data?**
  - we don't want to fill in witha value that can affect the distance too much
  - we can use the average attribute value across all of the data

1.7 kNN and Parzen Windows

1.7.1 Comparing kNN and Parzen Windows

In kNN, we classify based on the k neighbours which are closest to the instance we want to classify. In the **Parzen Window**, we clasify based on the class most present a given radius away from the instance we want to classify.



1.8 Relation Between kNN and SVM

- **How can we mathematically formulate the kNN classifier?**
  - consider 2 classes, labelled as +1 and -1
  - we can classify a new instance $\underline{x}$ via (assuming there is an odd number of points x_i):

$$f(x) = sgn\left[\sum_{i:x_i \in R(x)} y_i\right] = sgn\left[\sum_i y_i \mathcal{X}_{R(x)}(x_i)\right]$$

    where $R(x)$ represents the set of all nearest neighbours of x, and $\mathcal{X}_{R(x)}(x_i) = 1$ if and only if $x_i \in R(x)$, and 0 otherwise.

- **Is the above classifier robust?**
  - we can plot how $f(x)$ would look like, by noticing that $x_i \in R(x) \iff \|x - x_i\| \leq R$, where R is the radius of the neighbourhood of x. For example:

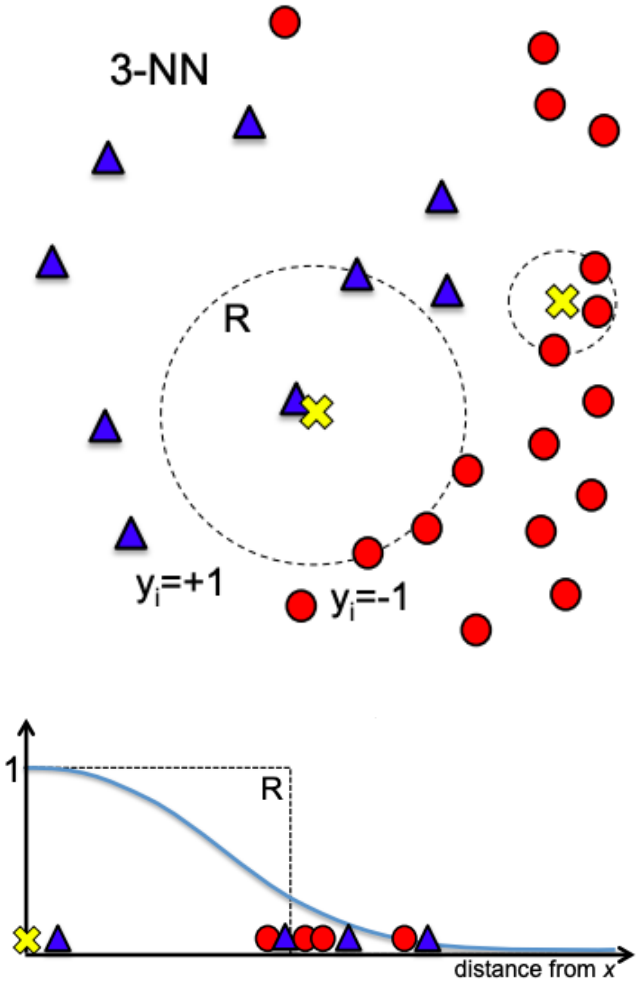


Figure 4: The R indicates the radius of the neighbourhood. If an instance is within the radius, is considered a neighbour.

  - however, notice that changing R by a bit can alter the classification: for example, if we shift it slightly to the left, we could classify the cross as either red or blue

- **How can we make the classifier more robust?**
  - instead of using such a strict decision boundary, we can smooth it out using a **kernel** (this is represented by the smooth blue line)
  - instead of immediately discarding all points outside the radius, we can consider **all** points, and assign a weight to each instance, depending on how close they are to the point we are testing x

- this weight can be represented by a **kernel**:

$$f(x) = sgn\left[\sum_i y_i K(x, x_i)\right]$$

- but recall the SVM classifier:

$$f(x) = sgn\left[w_0 + \sum_i \alpha_i y_i K(x, x_i)\right]$$

- the kernelised kNN is extremely similar to the kernelised SVM! The main difference is that SVM is **sparse** (most of the α_i are 0), whilst the kNN will consider all points in the training set.

1.9 Evaluating kNN

✓ very little assumptions made (proximity $\iff$ same label)

✓ non-parametric (no training; let the data speak for itself)

✓ classifier can be easily updated (add item to training set)

✗ need to account for missing values

✗ sensitive to outliers (can alter decision boundary)

✗ irrelevant attributes affect the distance, but don't aid in classification

✗ extremely **computationally expensive** (since no model is learned)

- need to store **all** training instances
- if there are n samples each of d dimensions, computing the distances (testing) is an $\mathcal{O}(nd)$ operation
- this is bad, because more accurate algorithms require more training data, but this will slow kNN

1.10 Speeding Up kNN

1.10.1 The Issue

Testing kNN is a $\mathcal{O}(nd)$ operation. This is because it needs to consider **all** the datapoints in teh training set to classify a new instance. In order to speed up, there are 2 possibilities:

- reduce **dimensionality** (i.e feature selection to remove irrelevant attributes)
- reduce **training instances** (i.e perform classification based on a subset of size $m << n$)

We will consider how to reduce the number of training instances required to classify, using 3 techniques:

- KD-Trees
- Locally Sensitive Hashing
- Inverted Lists

Some resources to help better understand KDT and LSH:

- On why LSH is less affected than KDT by curse of dimensionality
- KDT using many diagrams
- A more mathematical approach as to how the curse of dimensionality affects KDT

1.10.2 KD-Trees

- **On what type of data are KD-Trees useful?**
  - data with **low dimensionality**
  - data which is **real valued**
- **What is the runtime of KD-Trees?**
  - $\mathcal{O}(d \log_2 n)$
- **How do KD-Trees reduce the number of training instances required?**
  - the algorithm iterates over all the attributes/features of the data:
    1. Pick a random feature
    2. Compute the median of the data along that feature (for example, if the feature is height, we would compute the median height across all instances)
    3. Split the dataset based on those instances which are less or greater than the median (we can think of drawing a hyperplane across the value of the median height)
    4. Repeat, by picking a different feature, and applying the algorithm on each partition of the dataset (for example, if we have split the data based on height into 2 halves, we can then further split these halves based on their median weight)
  - we can stop the algorithm once the space partitioned has a given number of training instances
  - overall this results in a binary tree, with the leaves as the instances of the dataset which have similar feature values

– to classify $\underline{x}$, we can just search down the tree, until we reach a leaf. Then, we would apply kNN on $\underline{x}$ with all the training instances at the leaf node.

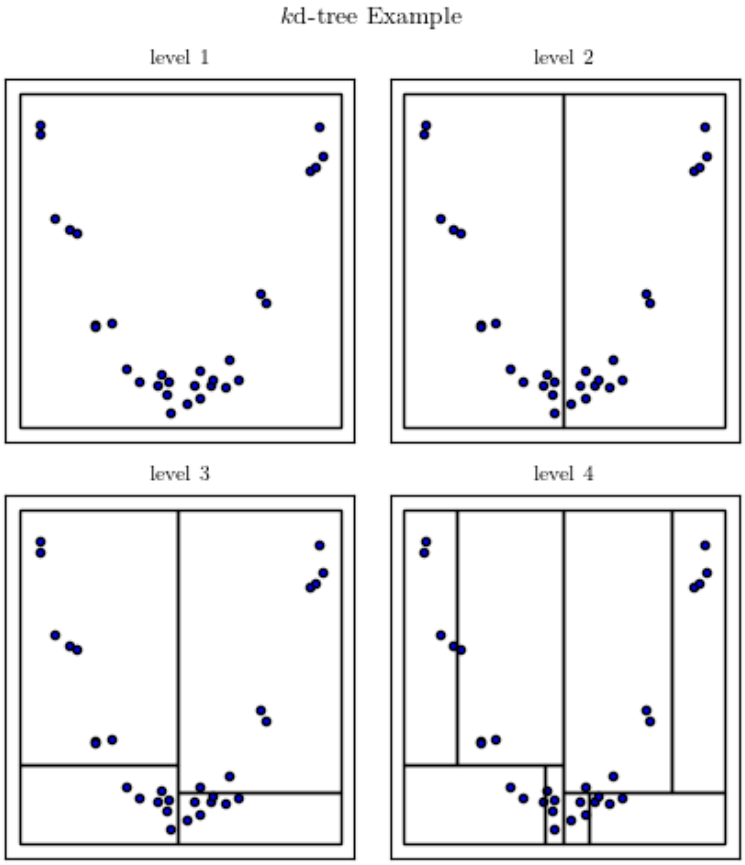


Figure 5: At each step, we split the data into 2, so each partition should have around $\log_2 n$ instances, hence the runtime improvement. We just need to focus on the points which are on the same region as the point we want to classify.

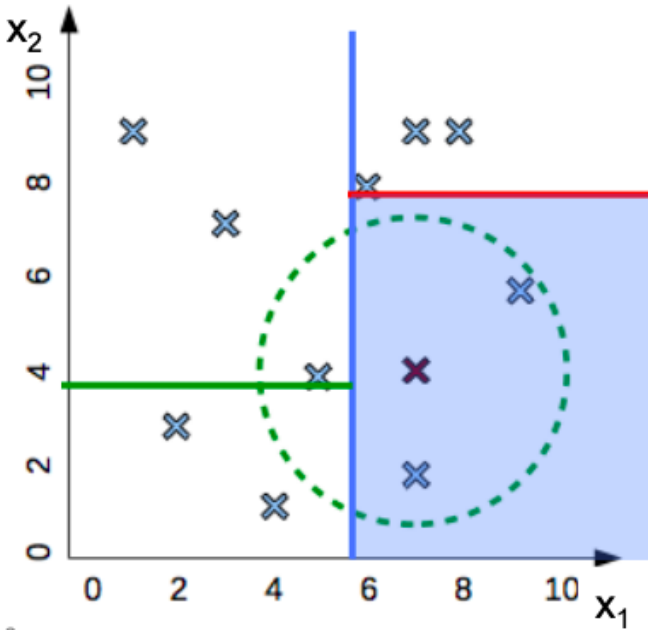


Figure 6: We have 2 attributes, so we split twice. The point which we want to classify is in the bottom right quadrant.

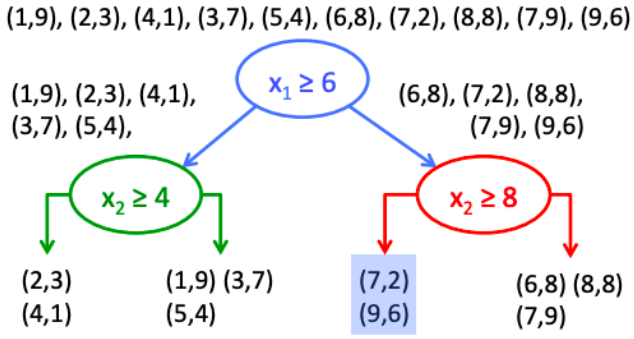
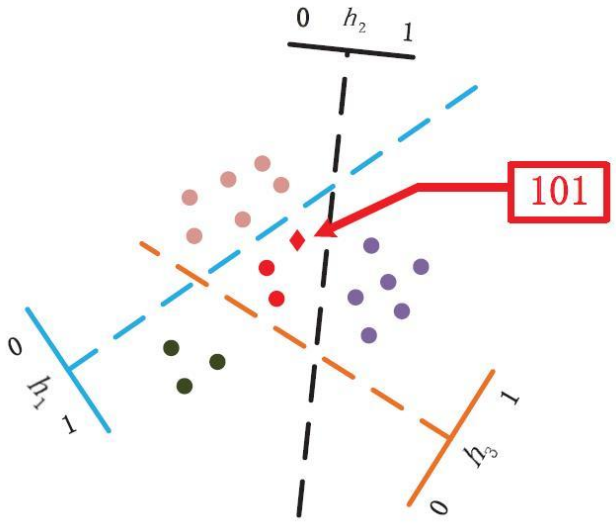


Figure 7: To classify the point, we notice that its x_1 attribute is greater than 6, and its x_2 attribute is less than 8, so we perform kNN using the instances $(7, 2)$ and $(9, 6)$

- notice, the above examples showcases one flaw of KDT: it can miss nearest neighbours. In the situation above, the point $(5, 4)$ is the closest point in the dataset to the point we want to classify, but due to the partition it won't be considered.

1.10.3 Locality-Sensitive Hashing

- **On what type of data is Locality-Sensitive Hashing useful?**
  - data with **high dimensionality**
  - data which is **real valued** or **discrete**
- **What is the runtime of Locality-Sensitive Hashing?**
  - $\mathcal{O}(kd + dn/2^k)$
  - here 2^k is the number of regions into which LSH splits the data space, with $k << n$
- **How does Locality-Sensitive Hashing reduce the number of training instances required?**
  1. Draw k random hyperplanes, splitting the space into 2^k regions
  2. Compare $\underline{x}$ only to the instances found in its region
  3. For more robust results, repeat using different random hyperplanes, to ensure that neighbours are not missed
  - these random splits are done with the idea that if points remain in the same region after all the splits, they should be similar
  - however, it is highly likely that a split separates 2 points which are close, hence why it is recommended to run LSH many times
  - it is also possible for the point which we want to classify to be in a region with no training instances



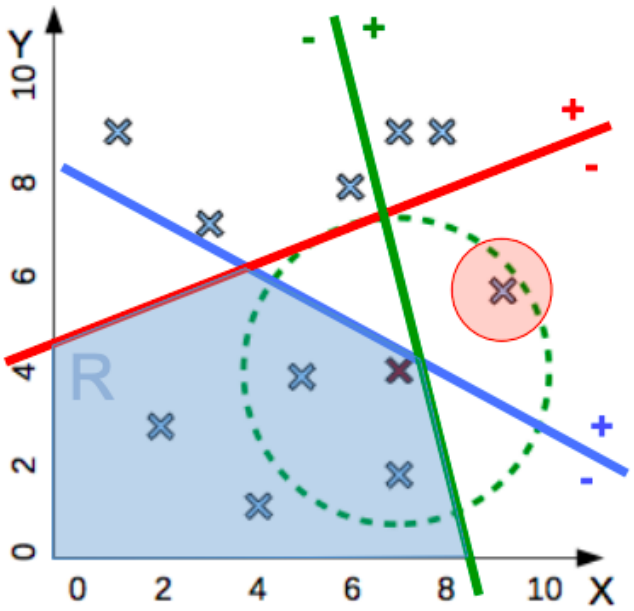


Figure 8: Notice wehn trying to classify the black cross (in the blue region

, a nearest neighbour was missed (circled in red). Moreover, notice how the green hyperplane separates a group of points which are very close together.

- **Why does Locality-Sensitive Hashing work for high dimensional data, whilst KD-Trees doesn't?**
  - the curse of dimensionality makes it so that in higher dimensions, distances become less meaningful
  - to think why, in higher dimensions, the value of an attribute tells us much less about the possible position of a point than if the point were in low dimensions (for example, it is easier to locate (1,2) than (1,8,2,9,4,6,3,1,2,2,3,5,9))
  - what this means is that since KDT split based on each attribute, if we have a high number of attributes, the splits will be less informative, since a single attribute in high dimensions doesn't carry as much "weight" as the same attribute in lower dimensions. What this means is that if KDT are applied with high dimensional data, they won't provide a significant performance improvement when compared to just looking at each training instance.
  - LSH is not affected by the value of the attributes, since it just splits the data using random hyperplanes

1.10.4 Inverted Lists

- **On what type of data are Inverted Lists useful?**
  - data which is **high dimensional**
  - data which is **discrete** and **sparse** (for example, text)
- **What is the runtime of Inverted Lists?**
  - $\mathcal{O}(d'\sqrt{n})$, where d' is the number of non-zero attributes
- **How do Inverted Lists reduce the number of training instances required?**
  - we can create a sort of `HashMap`, in which the attribute is a **key**, and the **value** is a list of all the training instances which contain the attribute
  - for example, we can think of words being represented as one-hot vectors, so sentences can be thought as sparse vectors. In this case, the key will be the word, and the values will be a list of all sentences which contain the word
  - if the data weren't sparse, then the lists assocaited to each attribute would be of more or less the same length, so it won't be useful
  - to classify using **inverted lists**, we just consider the attributes which appear in a new data instance. In order to make a classification, we then consider the classes of the values which contain said attributes.

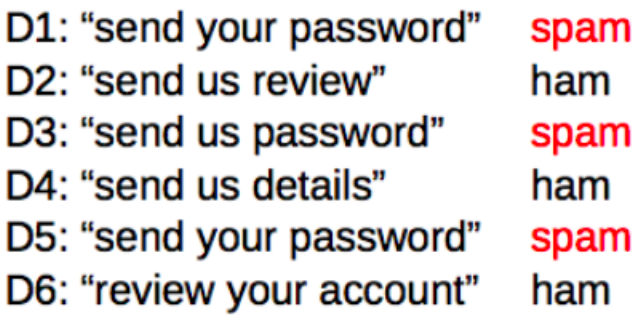


Figure 9: We have the following sentences, each labelled with "ham" or "spam".

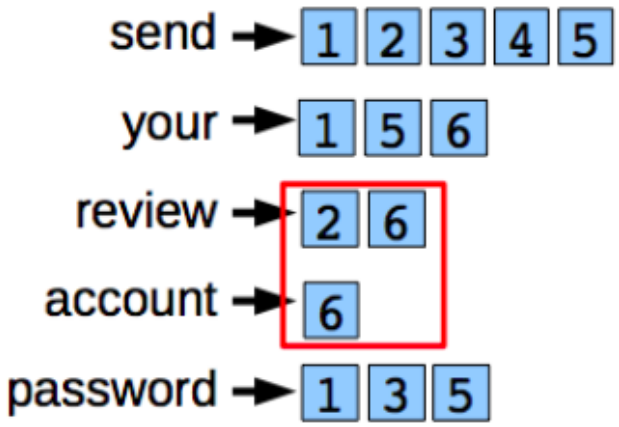


Figure 10: We can then create the inverted lists. For example, the attribute "your" appears in sentences 1,5 and 6, so we create the mapping "your" $\longrightarrow$ 1,5,6.

- for the above example, if we got a new sentence "account review", we would classify it based on the class of sentences 2 and 6 (since these are the sentences which contain "account" and "review"). In this case, we would classify "account review" as **ham**
- inverted lists are **exact**, in the sense that if 2 sentences are neighbours/similar, it will always use them for classification