Classification

As previously discussed,

- **Regression** means predicting a continuous value.
- **Classification** means predicting a discrete value.

In classification, we are given an object and we predict to which of a finite (and usually small) set of classes the object belongs. So we formulate it as follows:

- We assume we have a finite set of labels, $C$.
- Given an object $x$, our task is to assign one of the labels $\hat{y} \in C$ to the object.

We will often use integers for the labels.

- For example, given an email, a spam filter predicts $\hat{y} \in \{0, 1\}$, where 0 means ham and 1 means spam.
- But a classifier should not treat these as continuous, e.g. it should never output 0.5.
- Furthermore, where there are more than two labels, we should not assume a relationship between the labels.
  - Suppose there are three classes $\{1, 2, 3\}$ and we are classifying object $x$. One model predicts $\hat{y} = 1$ and another model predicts $\hat{y} = 2$. But the actual label for $x$ is $y = 3$. Which model has done better?
  - This problem of the unrelatedness of the labels in the case where there are more than two labels is one reason why we can’t use regression directly for classification.

```python
In [1]: load_ext autoreload
%autoreload 2
%matplotlib inline

In [2]: import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```
A Variation of Classification

Given an object \( x \), a classifier outputs a label, \( \hat{y} \in C \). Instead, a classifier could output a probability distribution over the labels \( C \).

- E.g. given an email, a spam filter might output \( \langle 0.2, 0.8 \rangle \) meaning \( P(y = \text{ham} | x) = 0.2 \) and \( P(y = \text{spam} | x) = 0.8 \).
- The probabilities must sum to 1.
- We can convert such a classifier into a more traditional one by taking the probability distribution and selecting the class with the highest probability:

\[
\arg \max_{\hat{y} \in C} P(\hat{y} | x)
\]

Types of Classification

We distinguish two types of classification:

- **binary classification**, in which there are just two classes, i.e. \( |C| = 2 \), e.g. fail/pass, ham/spam, benign/malignant;
- **multiclass classification**, where there are more than two classes, i.e. \( |C| > 2 \), e.g. let's say that a post to a forum can be a question, an answer, a clarification or an irrelevance.

Binary classification

In binary classification, it is common to refer to one class (the one labelled 0) as the **negative class** and the other (the one labelled 1) as the **positive class**. It doesn't really matter which is which. But, usually, we treat the class we're trying to identify, or the class that requires special action, as the positive class.

- So in spam filtering, ham is the negative class; spam is the positive class.
- What about tumour classification?

This terminology is extended to other things too, e.g. we can refer to **negative examples** and **positive examples**.

Multiclass classification

Note that many classifiers can only do binary classification. How can we use a binary classifier for a multiclass problem? Two methods are common:

- One-versus-Rest; and
- One-versus-One.

One-versus-Rest
**One-versus-Rest** involves training $|C|$ classifiers, one per class. It relies on the idea that these classifiers can output probabilities, as above. You can think of it as taking the training set, producing $|C|$ copies of it but with modified labels, and training a classifier on each copy, as follows:

- **for each** class $c \in C$
  - create a copy of the training set in which you replace examples $\langle x, c \rangle$ by $\langle x, 1 \rangle$ and examples $\langle x, c' \rangle$ where $c' \neq c$ by $\langle x, 0 \rangle$
  - train classifier $L_c$ on this modified training set

After all these classifiers have been trained, to classify $x$, you run all the classifiers $L_c$ for each $c \in C$. The predicted class of $x$ is the class $c$ whose classifier $L_c$ predicts 1 with highest probability.

This is a good approach provided the binary classifier is one that produces reliable probabilities.

Another name for this is 'One-versus-All'. We will stick with One-versus-Rest on the basis that this is the name of a scikit-learn class that can do this for us.

There are some variations of One-versus-Rest, e.g. the use of error-correcting codes. We will not study these because they are more complicated but also there are restrictions on their applicability (e.g. there need to be at least four classes and there may be problems using them for kNN classifiers).

**One-versus-One**

In **One-versus-One**, you build a classifier for every pair of classes using only the training data for those two classes.

After all these classifiers have been trained, when you want to classify $x$, you run all the classifiers and, for each class $c \in C$, you count how many of the classifiers predict that class. The predicted class of $x$ is the one that is predicted most often.

One-versus-One's advantage over One-versus-Rest is that the individual classifiers do not need to produce probabilities. But the disadvantage is the number of individual classifiers it must train.

- In terms of $|C|$, how many?
  - This sounds like a lot of work. What offsets this, so that in practice it may be quite feasible?

Sometimes, even when you have a classifier that can directly handle multiclass classification, using it in a One-versus-One fashion can be more accurate! This is because it is like an 'ensemble' learning method, which we will discuss properly in a later lecture.

Here again there is another name in the literature, 'pairwise classification', but we stick with scikit-learn's nomenclature.

There are some other methods with similarities to One-versus-One, such as ensembles of nested dichotomies, which we will not study in this introductory module.

**Multiclass classification in scikit-learn**
scikit-learn provides classes that make it easy to implement One-versus-Rest, One-versus-One and error-correcting codes. But we don't need them!

Even when a classification algorithm can only do binary classification, scikit-learn's implementation comes with one of the above methods built-in already. For example, we will see that logistic regression typically only does binary classification, but scikit-learn's implementation of it already makes use of One-versus-Rest.

As we proceed, I will make clear which classifiers are restricted to binary classification. But, from an implementation point of view, we can ignore this issue from now on.

Other Types of Classification

Here are two variations of classification that we will not be studying any further in this module.

In **multilabel classification**, the classifier can assign \( x \) to more than one class.

- In other words, it outputs a set of labels, \( \hat{y} \subseteq C \).
- E.g. consider a movie classifier where the classes are genres, e.g.
  \[
  C = \{ \text{comedy, action, horror, musical, romance} \}.
  \]
  - The classifier's output for *The Blues Brothers* should be \( \{ \text{comedy, action, musical} \} \).
- Do not confuse this with **multiclass** classification

In **ordered classification**, there is an ordering defined on the classes. The ordering matters in measuring the performance of the classifier.

- E.g. consider a classifier that predicts a student's degree class, i.e.
  \[
  C = \{ \text{Ordinary, 3rd, 2ii, 2i, 1st} \}.
  \]
- Suppose we are classifying student \( x \). One model predicts \( \hat{y} = 2ii \) and another model predicts \( \hat{y} = 2i \). But the actual label for \( x \) is \( y = 1st \). Which model has done better?

Nearest-Neighbours for Classification

Recall the CS1109 dataset: it has 342 examples of students; 3 features (\( \text{lect, lab, cao} \)); the dependent variable's values are either 0 (the student fails) or 1 (the student passes). Let's load it and, because we'll be using nearest-neighbour methods, let's scale it:
In [5]:
```
# Read CSV file
df = pd.read_csv("dataset-cs1109.csv")

# lect, where min = 0 and max = 100 (it's a percentage)
# Subtract the min
# df['lect'] -= 0
# Divide by the range, i.e. max - min
df['lect'] /= 100

# lab, where min = 0 and max = 100 (it's a percentage)
# Subtract the min
# df['lab'] -= 0
# Divide by the range, i.e. max - min
df['lab'] /= 100

# cao, where min = 200 and max = 600
# Subtract the min
df['cao'] -= 200
# Divide by the range, i.e. max - min
df['cao'] /= 400

# Get the feature-values into a separate numpy arrays of numbers and
# the target values into a separate
# numpy arrays of ints
X = df[['lect', 'lab', 'cao']].values
y = df['outcome'].values
```

1NN

In 1NN, we predict the class of the nearest-neighbour. We'll do this for Craig, who is described by this vector

\[
\begin{bmatrix}
60 \\
45 \\
500
\end{bmatrix}
\]

but, when scaled, is described by this vector

\[
\begin{bmatrix}
0.6 \\
0.45 \\
0.75
\end{bmatrix}
\]

In [13]:
```
from sklearn.neighbors import KNeighborsClassifier

estimator = KNeighborsClassifier(n_neighbors = 1)
estimator.fit(X, y)
estimator.predict([[0.6, 0.45, 0.75]])
```

Out[13]: array([1])

His nearest-neighbour is \( \begin{bmatrix} 0.39 \\ 0.5 \\ 0.7 \end{bmatrix} \) (scaled), which is \( \begin{bmatrix} 39 \\ 50 \\ 480 \end{bmatrix} \) (unscaled). We predict 1 — he will pass.

kNN
Unweighted

With $k > 1$, for regression we returned the mean of the neighbours’ $y$-values.

Question:

- What will we do for classification?
- Hence, for unweighted kNN ($k > 1$), if someone picks $k$ manually (rather than using model selection), s/he often chooses an odd number, e.g. $k = 3, k = 5$. Why?

\[
\text{In [18]:} \quad \text{estimator = KNeighborsClassifier(n_neighbors = 3)} \\
\text{estimator.fit(X, y)} \\
\text{estimator.predict([[0.6, 0.45, 0.75]])}
\]

Out[18]: array([0])

Craig’s 3 nearest-neighbours are $\langle 0.39, 0.5, 0.7 \rangle$, $\langle 0.48, 0.36, 0.575 \rangle$, and $\langle 0.68, 0.64, 0.6375 \rangle$. Hence, now we predict 0 (fail).

kNN

Distance-weighted

In distance-weighted kNN, the closer you are, the greater your vote.

The distances of the neighbours from Craig, to three decimal places, are 0.222, 0.230 and 0.234, respectively. The weights are \( 1/(0.0001 + 0.222) = 4.502 \), \( 1/(0.0001 + 0.230) = 4.346 \) and \( 1/(0.0001 + 0.234) = 4.271 \) respectively. So the votes for class 0 (fail) are \( 4.346 + 4.271 = 8.617 \); and the votes for class 1 (pass) are 4.502. So we predict Craig fails.

\[
\text{In [19]:} \quad \text{def inv_distances(dists):} \\
\quad \quad \text{return 1 / (0.0001 + dists)} \\
\text{estimator = KNeighborsClassifier(n_neighbors = 3, weights = inv_distances)} \\
\text{estimator.fit(X, y)} \\
\text{estimator.predict([[0.6, 0.45, 0.75]])}
\]

Out[19]: array([0])
Shepard's Method

Here \( k = m = 342 \) and we must use distance-weighting. (Why?)

```
In [21]: estimator = KNeighborsClassifier(n_neighbors = 342, weights = inv_distances)
estimator.fit(X, y)
estimator.predict([[0.6, 0.45, 0.75]])
```

```
Out[21]: array([1])
```

So now we're once more predicting he passes!

kNN

Variations

What if we want to use nearest-neighbour methods to build the kind of classifier that outputs a probability distribution?

- We can't do this when \( k = 1 \). Why not?
- In the unweighted 3NN example above, the probabilities would be \( \left( \frac{2}{3}, \frac{1}{3} \right) \) meaning
  \[ P(\text{fail}|x) = 0.66 \quad \text{and} \quad P(\text{pass}|x) = 0.33 \]  How are these computed?
- In the weighted 3NN example above, the probabilities would be \( \langle 0.657, 0.343 \rangle \) meaning
  \[ P(\text{fail}|x) = 0.657 \quad \text{and} \quad P(\text{pass}|x) = 0.343 \]  How are these computed?

What about multiclass classification?

- Can nearest-neighbour classifiers be used for multiclass classification?