Practical 3

Jumping Rivers

During the lecture we fit a logistic regression model to the breast cancer data for classifying tumors in patients. We are going to fit a KNN classifier to the same data.

- Construct the pipeline ready for fitting the model
- We want to find the best value of K for the classifier when optimising for recall, our motivation is that we want to correctly identify as many of the malignant tumours as possible. Start with a grid search over `k = [1,5,10,20,50,100]`
- Create a plot of the K parameter against the average recall score found in the cross validation grid search
- What region of K looks like it will give the best value?
- Re-run your grid search across that region
- What is the best parameter choice and the corresponding recall score?
- Is this better than the Logistic regression in the notes?

Other techniques

- Try the other classification algorithms that we explored, can you find any that perform better?

Response Optimisation

In this question we aim to use machine learning as a route to optimising a response. In particular, the data we have at hand are mixtures of concrete together with there measured compressive strength. We want to use the available data to propose new formulations of concrete that might be better than those used in the experiment.

The data can be loaded as

```
import jrpyml
concrete = jrpyml.datasets.mixtures.load_data()
```

- This isn't a traditional experimental design set up, there are some mixture formulations that have repeated measures. A sensible thing to do in situations like this is to average over the repeated responses.

```
concrete = concrete.groupby([
  item for item in concrete.columns if item != 'CompressiveStrength'
  ]).agg({
    'CompressiveStrength': 'mean'
  }).reset_index()
```

- Try any number of models aiming to find one that gives good predictive performance

- Once you have found a model you are happy with, we can aim to use it to optimise mixtures of concrete at different ages. This optimisation problem also has some constraints, namely that the sum of all components must be equal to 1 (as we have proportions of a mixture). Further no input can be less than 0 or greater than 1. One way to impose these constraints is to define an objective function in terms of 6 of the mixture components and infer the final one. Such an objective function might be defined as below. We return the negative model prediction here as out optimisation routine will look to minimise the function.

```
import numpy as np

def obj(x, age=28):
  # add some constraints, # index 7 is age
  # sum of components excluding age must be 1
  # there are 7 components in total

  if np.any(x < 0) | np.any(x > 1):
    return 1e50
  x = np.append(x, [1-x.sum(), age])
  return -1*model.predict(x.reshape(1,-1))
```

- Given an objective function to minimize, **scipy** has a number of routines to do so. We try to minimize the function from a number of random start points, the aim being to have a better chance at finding a global minimum.

```
from scipy.optimize import minimize
import random

## 20 random rows of data to start from
start_idx = [random.randint(0,X_train.shape[0]) for _ in range(20)]

outputs = [minimize(obj, X_train[i,:-2], method = 'Nelder-Mead') for i in start_idx]

## Extract the results of the objective function
## once the optimisation routine completes.
np.array([o.fun.flatten()[0] for o in outputs])

## For a given output we might also extract the input array
## (The mixture of concrete in our case)

outputs[0].x
```