AIML430/COMP309: ML Tools and Techniques
Lecture 10: Data Preprocessing

Ali Knott
School of Engineering and Computer Science, VUW

The plan for this week and next week

This week and next, we're finishing up with data preprocessing.

Today a bit more on a few things we've already touched on:

- Categorical data: how to *encode* it
- Numerical data
  - More on *scaling* (normalisation and standardisation)
  - More on *splitting* (more formally, discretisation).

Tomorrow: a more detailed look at *dimensionality reduction* methods.

- *Global methods* for changing feature space (PCA, & new ones)
- *Local methods* for removing features.

Next week (with Marcus Frean): 'Feature engineering'.

- In particular: how to *construct* useful new features.

Before we start - a few more points on data splitting!

Data splitting is when you divide your dataset into training, validation and test sets.

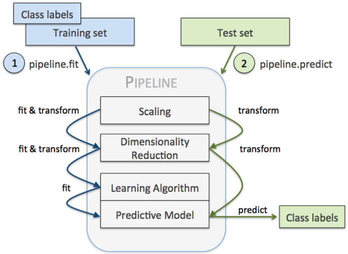
1. What proportions should you divide your dataset into?
   - No hard and fast rules...
   - A reasonable range is
     60-80% training set
     10-20% validation set
     10-20% test set.

Before we start - a few more points on data splitting!

2. An important rule: *split first*, and then preprocess the separate datasets separately!

- If you preprocess *before* splitting, information from the training set can *leak* into the validation/test sets.
- That data leakage compromises the validation/test sets.

This hygiene is built into Scikit-learn's pipeline structures. . .

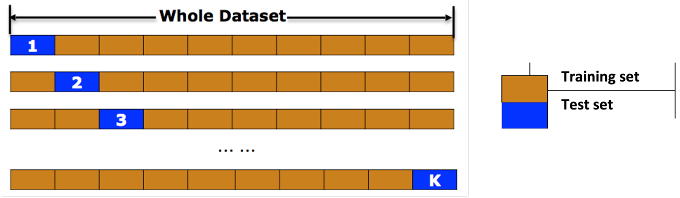


Before we start - a few more points on data splitting!

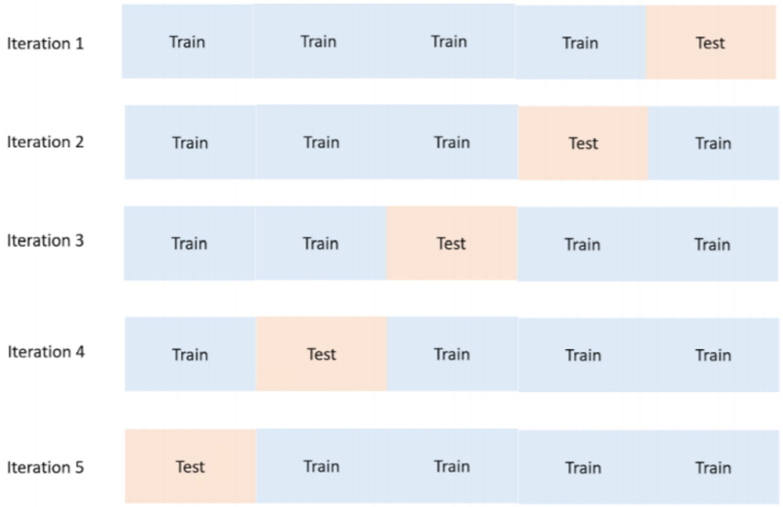
3. We briefly mentioned a useful method called cross-validation: here's some more on that.

If you don't have much data, leaving some out for testing feels wasteful.

- *k*-fold cross-validation is a way of training on all the data.
- Method: split the dataset into *k* equal-sized folds.
  - For each fold *F*:
    - Set aside *F* for testing, and train on all the other folds.
- To evaluate the learned model, take the *average* test performance.



An example: 5-fold cross-validation



Iteration 1	Train	Train	Train	Train	Test
Iteration 2	Train	Train	Train	Test	Train
Iteration 3	Train	Train	Test	Train	Train
Iteration 4	Train	Test	Train	Train	Train
Iteration 5	Test	Train	Train	Train	Train

Cross-validation notes

There's no 'leakage' in cross-validation, because each model trains separately.

But: there's no real *validation set* in the method just sketched.

- If you want to tune model parameters, you can hold one fold aside as a true 'test set', and do cross-validation (with tuning) on the remaining $k - 1$ folds.

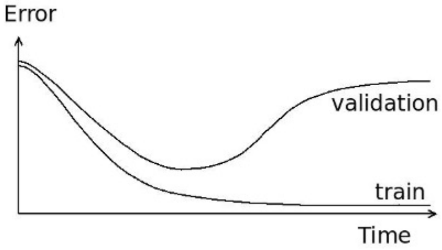
One more point on training/testing. . . about overfitting

4. Some training processes take time, and increase the complexity of the learned model over time.

- A good example is a neural network.
  - This trains incrementally, over *n* iterations. . .
  - Weights are gradually adapted to training items as time passes.

In this kind of case, *overfitting happens at a point in time*.

- To identify that point, test on the validation set regularly *during training*, and see when validation performance *stops improving*.



And now to data preprocessing!

Recap: there are two broad types of feature:

- Categorical features (with discrete alternative values)
- Numerical features (with continuous values).

For categorical data, I'll say more about how it should be *encoded*.

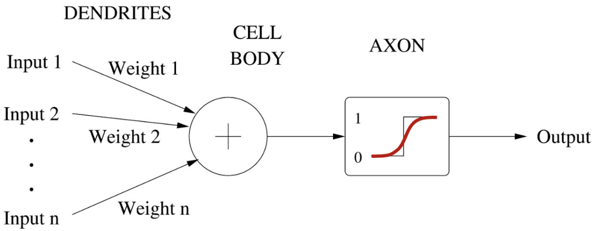
For numerical data, I'll say more about how it should be *scaled*, and *discretised*.

1. Encoding categorical data

1.1. ‘One-hot’ encoding

When we introduced neural network classifiers, we focussed on binary classifiers, that make decisions about a *single class*.

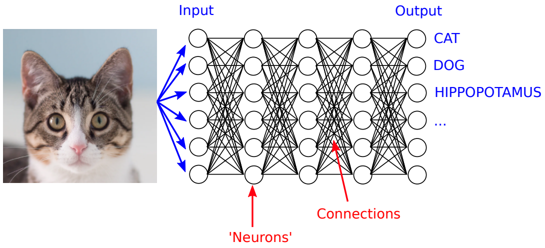
- 1 means ‘yes’, 0 means ‘no’.
- We saw a classifier that responds discretely, and one that responds with a *probability*.



1.1. ‘One-hot’ encoding

Neural networks can also be *multi-class* classifiers.

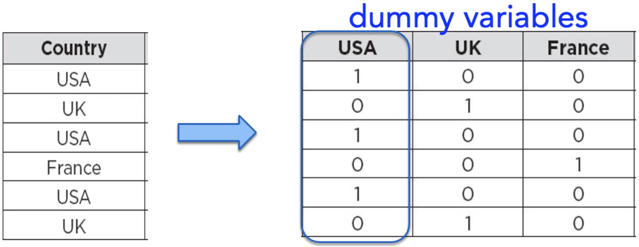
- If there are *n* classes, the network has *n* output units.
- Here’s an image classifier, that recognises discrete object classes.
- Activity in the output layer can be interpreted as a probability distribution—if we normalise it to sum to 1.



1.1. ‘One-hot’ encoding

This kind of network is trained on a one-hot probability distribution: 1 for the correct class, and 0 for all the others.

- In the dataset, the output feature must be *encoded* in this one-hot notation.
- We often do that by creating a dummy variable for each output class.

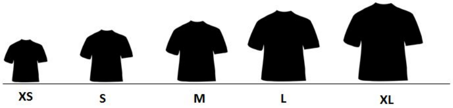

dummy variables

Country
USA
UK
USA
France
USA
UK

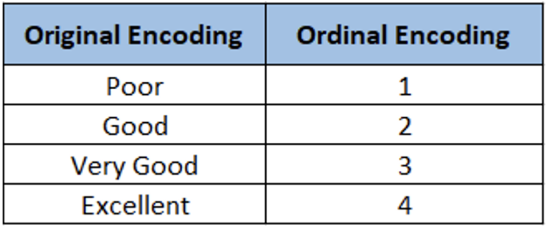
USA	UK	France
1	0	0
0	1	0
1	0	0
0	0	1
1	0	0
0	1	0

1.2. Ordinal encoding

Some categorical features are unordered; others are ordinal.

- An ordinal feature has values with a natural *ranking*:
  - E.g. *poor*, *good*, *very good*, *excellent*
  - E.g. 

An ordinal encoding just assigns *integers* to the ranked categories.



Original Encoding	Ordinal Encoding
Poor	1
Good	2
Very Good	3
Excellent	4

2. Encoding numerical data

Two topics here

- Scaling
- Discretisation.

2.1. Scaling numerical data

We have already discussed how numerical features should be *scaled*:

- If you don't scale, features with a larger *range* will dominate...
- Especially in *distance* computations.

I'll discuss two scaling methods in some more detail.

- Linear scaling
- Standardisation.

2.1.1. Linear scaling

In linear scaling for a feature f, we scale the whole dataset by a constant factor.

Often, we scale so that datapoints fall in the range [0,1].

- We begin by finding the *maximum* and *minimum* values for f (x_{max}, x_{min}).
- Then, for any value x, the scaled value x' is given by $\frac{x - x_{min}}{x_{max} - x_{min}}$.
- In this scheme, the min values are scaled to 0, and the max values are scaled to 1.

We can also scale to a specified range, $[New_{min}, New_{max}]$. In this case,

$$x' = \frac{x - x_{min}}{x_{max} - x_{min}} \times (New_{max} - New_{min}) + New_{min}.$$

In scikit-learn, the MinMaxScaler object does these operations.

2.1.2. *z*-score standardisation

If a feature happens to be normally distributed, there's a more informative way of scaling.

z-score standardisation, or centre scaling, scales the *distribution* of points, so they have a *mean of 0*, and a *standard deviation of 1*.

- This kind of scaling is useful, because its units are standard deviations. (Value 1 is 1 standard deviation above the mean.)
- Values map easily to probabilities (see next page).

For a feature with mean μ and standard deviation σ, the scaled value z for a raw value x is given by

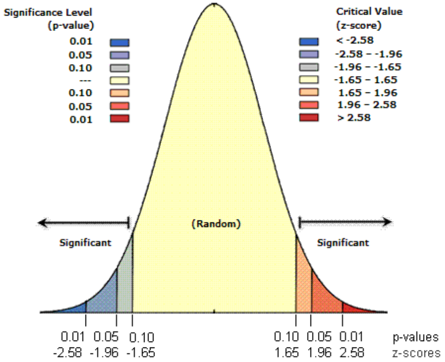
$$z = \frac{x - \mu}{\sigma}$$

To apply this kind of scaling, use the scikit-learn object StandardScaler.

z-scores and *p* values

With a normally distributed variable, we can compute the probability that values fall within a given range.

- The *p* value for a given range is the area under the normal curve for that range. There's a mapping from *z* ranges to *p* values.
- You can compute *p* values with the scipy.stats fn norm.ppf.



Linear scaling or *z*-score standardisation?

The basic rule:

- If your numerical feature is normally distributed, use *z*-score standardisation.
- Otherwise, use linear scaling.

If you have a *mixture* of normal and non-normal input features, you should use linear scaling on all of them—so you can scale them all to *the same range*.

2.2. Discretisation

Discretisation (or binning) is the process of converting continuous numeric values (e.g. price, age, weight) into discrete intervals.

Two key purposes for discretisation:

- It's required for some ML algorithms—in particular, decision trees.
- It's used in many visualisation routines.

There are two types of discretisation:

- Supervised—splitting is guided by class labels. (E.g. in decision tree learning.)
- Unsupervised—no reference to class labels.

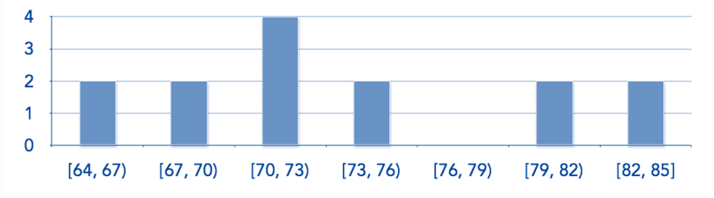
2.2.1. Unsupervised discretisation methods

Uniform (equal-width) discretisation creates equal-sized bins for a numerical variable.

- This is what we use to make a histogram.
- The basic process: find the Max and Min values, then divide Max-Min by the desired number of bins.

Say we have this set of temperature values, and we want 7 bins.

- 85, 80, 83, 70, 64, 65, 68, 71, 69, 72, 75, 75, 81,72
- The width of each bin (85-64)/7=3 .
- KBinsDiscretizer(n_bin=7, encode='ordinal', strategy='uniform')

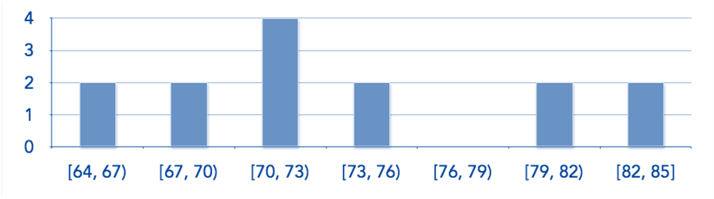


2.2.1. Unsupervised discretisation methods

Quantile (equal-depth) discretisation divides numerical data into bins that each hold (approximately) the same number of items.

- KBinsDiscretizer(n_bin=4, encode='ordinal', strategy='quantile')

For the temperature values shown before, we get:



I don't see what this is useful for!

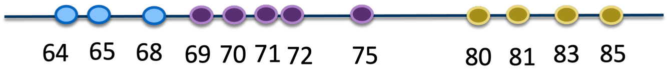
Useful to note: discretisation creates an ordinal categorical variable.

2.2.1. Unsupervised discretisation methods

Clustering can be used to choose bins for discretisation.

- KBinsDiscretizer(n_bin=3, encode='ordinal', strategy='kmeans')
- This runs *k*-means clustering *in one dimension*, with three centroids.
  - A bin is created for each cluster.

For the temperature values shown before, we get:



2.2.2. Supervised discretisation methods

Supervised discretisation methods aim to find splits for a numerical variable that best match values of a discrete output class.

The basic idea is to search through *all possible split points*, to find the points that best separate classes.

- Often, we look for *binary* splits, that distinguish one class from the others.
- Within each binary split we must *recurse*, looking for further splits:
  - To separate other classes;
  - To refine splits for each class.

The process is similar to decision-tree learning.

2.2.2. Supervised discretisation methods

There are various methods for assessing the ‘goodness’ of a candidate split.

Some methods compute the total error of a given split.

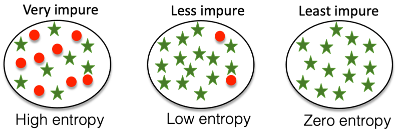
- For each split, assign the most frequent class label...
- Then count the number of items that are of other classes
- And sum over all splits.
- This is used by 1R discretisation.
  - In R’s dprep package, use the function disc.1r.

2.2.2. Supervised discretisation methods

Other supervised methods compute the entropy (or ‘purity’) of a given split.

Entropy is a probabilistic measure: it quantifies the ‘sharpness’ of a probability distribution. (A sharp distribution has low entropy.)

- For a probability distribution over N classes $c_1 \ldots c_N$, the entropy is given by $E = -\sum_{i=1}^{N} p_c \times log_2(p_c)$.
- For each candidate split, we can estimate probabilities by counting, and compute entropies...



- We are looking for the split with the *lowest entropy*.
- In R’s dprep package, use the function disc.mentr.