Applications of Data Science

Xander Byrne

January 2, 2024

1 Learning from Data

Data science is about learning a model from data. In order to do this, the data must accurately represent the real world. The data must therefore be:

- **Accurate**: the data should reflect what is observed, and systematic errors should be absent
- **Complete**: the data should include data from all relevant classes, and across a wide range of parameter space;
- **Time-invariant**: otherwise our models will quickly fall out of date and we'll have to start over

When creating a model from data, there are three things one should do:

1.1 Inspect the Data Structure

- **Quantitative/Qualitative**: e.g. height, or country of birth
- **Continuous/Categorical**: e.g. weight, or birthday
- **Range**: If quantitative, what are the feature's maximum and minimum values? If categorical, how many categories can the feature be in?

1.2 Preprocess the Data

Data may be in an awkward form and need to be "cleaned up" before a model can be created from it.

- **Missing Data**: A datum may have missing features, perhaps even being unlabelled.
- **Redundance**: Features may be co-dependent and hence redundant. Some data instances may be repeated.

Data can be awkward in less boring ways that might also make it difficult for modelling.

1.2.1 Qualitative Features

Many types of models are based on numbers, so how might one encode a qualitative class, e.g. birth country? It makes no sense to assign, for example, Afghanistan $\to 1$, Albania $\to 2$ etc., as this implies some form of quantitative ordering which is meaningless. There is no reason to assign instances with this feature being Afghanistan as "closer" to those with Albania than to those with Zimbabwe.

A better approach is *one-hot encoding*, where rather than assigning a qualitative feature an integer from 1 to C, we instead assign a C-dimensional vector whose entries are all 0 except for one. We would thus assign Afghanistan $\to (1, 0, 0, ...)$, Albania $\to (0, 1, 0, ...)$ such that all the classes are equidistant.

1.2.2 Heterogeneous Ranges

If the range of some feature is orders of magnitude larger than others, the construction of a model may over-focus on this feature. To avoid this, we need to somehow standardise this feature; there are several ways of achieving this.

We could rescale all features to the range $[0, 1]$ by subtracting the minimum and dividing by the range. This has the advantage that the distribution shape of that feature is unchanged. However, if that feature has outliers then they will be at 0/1 and the rest of the features will be smushed together at the other end. Sometimes this is mitigated by taking a $\log_2$ transformation first (though this will alter the feature distribution shape), or by introducing a cutoff for that feature (though this may be removing important information)

An alternative is to make a Z-transformation, taking the mean μ and standard deviation σ of that feature, and assigning each datum a feature $Z_i = (x_i - \mu)/\sigma$, where x_i was the value of the original feature. This handles small numbers of outliers well: they won't affect μ much, though they will affect σ, causing to the Z distribution to be narrower. A disadvantage is that if this transformation is applied to every feature, any information on the relative variances/sizes of the features is lost.

A variation on the Z-transformation uses the median absolute deviation (MAD), given by:

$$\text{MAD} = \text{median}\Big[\big|x_i - \text{median}[x_i]\big|\Big] \qquad \Rightarrow \qquad Z_\text{m} \equiv \frac{x_i - \text{median}[x_i]}{\text{MAD}}$$

This transformation is more robust to outliers.

1.2.3 Near-Zero Variance

If every datum has the same value for one feature, it is hardly a feature. Features with near-zero variance can be excluded.

1.3 Ensure Robustness

Once the model has been created, we must ensure that it is textitrobust: that it can perform well on new/unseen data. If not, it is likely that the model is *overfit* on the data it was trained on. To test this, we need a separate and independent dataset on which to evaluate the model's generalisability.

This can be achieved by splitting the available data into a *training set* and a *test set*; often this is done in a 90:10 or 95:5 ratio. One then trains the model on the training set, evaluates it on the test set, and optimises the model parameters based on how it performs on the test.

A problem with this is that the model may then overfit on the *test* set instead. A way of escaping this information leaking is to create a third set in between: the *validation set*, with a split of perhaps 80:10:10. The model is trained on the training set, optimised based on performance on the validation set, and then evaluated on the test set, which was completely unused in training.

A disadvantage is the reduced amount of data we can train the model on. This may be mitigated by *cross-validation*, where at each training stage the training and validation sets are re-partitioned, allowing all of the data to be used at some point to train the data. For example, k-fold cross-validation, where the dataset is partitioned into k equal "folds" of size N/k; at each training iteration, one fold is selected as the validation set, the model is trained on the rest of the data, and then evaluated on this validation fold. At the next iteration, a different fold is selected.

A common issue with validation splitting, as well as data collection in general, is the *balance* of the data. Do members of one class contribute 90% of the data? In which case, a model would achieve a 90% success rate just by guessing this class every time! This can be mitigated by *downsampling*, where overrepresented entries are removed from the training data, though this can remove useful generalisable features. The opposite approach is *upweighting*, where underrepresented entries are artificially replicated in the training data (perhaps with some variation), though this might result in an artificially lower variance for that class.

1.3.1 Evaluating Models

For supervised learning tasks, where the output of the model for each datum is known, one can create a *confusion matrix*, containing the *true positives* (TP) which were correctly classified as "positive"[1], *true negatives* (TN) which were correctly classified as "negative"[2], and false negatives and false positives off the diagonal. There are loads of other metrics which can be used to evaluate models depending on what one wants to optimise, such as the *false discovery rate*, or *negative predictive value*.

1.3.2 Bias-Variance Trade-Off

A model's *bias* is, vaguely speaking, errors on the test set. A model's *variance* is its sensitivity to small fluctuations, as learned from the training set. Bias is due to having learned insufficiently completely, such as by *underfitting* the training set. Variance is a result of modelling random noise in the training set, i.e. *overfitting*.

These two are usually in conflict, and it is usually impossible to get both down to a low level: this is the *bias-variance trade-off*. One must make a political decision as to how much of each is permissible.

[1] This might be successful classification in a particular class

[2] which might be successful classification in *another* class

2 Supervised Learning

2.1 Linear Regression

Regression is a continuous-output problem; the features can be of any kind but is usually continuous (if not they can be one-hot encoded). Linear regression assumes a natural law $Y = \beta_0 + \beta_1 X + \epsilon$ (for one feature and one target variable), and attempts to find the best estimates $(\hat{\beta}_0, \hat{\beta}_1)$ for the real coefficients (β_0, β_1). It can answer the the following questions:

- Is there a linear relationship between any features and the target variable(s)?
- If so, how strong are these relationships?
- Which feature explains the target best?
- How accurately can we predict the target variable?

One might implement a linear regression by minimising the residual sum of squares

$$\text{RSS} = \sum_i^N (y_i - \hat{y}_i)^2 = \sum_i^N \left(y_i - \hat{\beta}_0 - \hat{\beta}_1 x_i\right)^2$$

Minimising the RSS with respect to $\hat{\beta}_0$ and $\hat{\beta}_1$ turns out to give the following estimates:

$$\hat{\beta}_1 = \frac{\sum_i^N (y_i - \bar{y})(x_i - \bar{x})}{\sum_i^N (x_i - \bar{x})^2} \qquad\qquad \hat{\beta}_0 = \bar{y} - \hat{\beta}_1 \bar{x}$$

There are various ways of evaluating a regression. One way is to calculate a t-statistic (effectively a signal-to-noise ratio on $\hat{\beta}_1$), and a p-value giving the probability of obtaining a test statistic $\geq |t|$: a low p-value signifies a strong correlation. Another test statistic is the *coefficient of determination* R^2, which quantifies the fraction of variability in the data that a given linear model predicts. It is given by

$$R^2 = \text{corr}[x, y]^2 = \frac{\left[\sum (x_i - \bar{x})(y_i - \bar{y})\right]^2}{\sum (x_i - \bar{x})^2 \sum (y_i - \bar{y})^2} \in [0, 1]$$

$R^2 \approx 0$ means that a linear model is bad; $R^2 \approx 1$ means it explains most of the data variability.

Other metrics include the *mean absolute error* (or L1 loss): $\text{MAE} = \frac{1}{N}\sum |y_i - \hat{y}_i|$, and the *mean square error* (L2): $\text{MSE} = \frac{1}{N}\sum (y_i - \hat{y}_i)^2$. L2 is less robust to outliers, amplifying quadratically the influence of data that are away from a linear fit. However, L1 is less stable to noise than L2, which reacts more smoothly. The *Huber loss* combines the two, as an L2 loss for small residuals and an L1 for larger residuals.

2.1.1 Multiple Regression

When using multiple features to predict a target variable, some features may be more useful than others. There are 2^F subsets of features (for F features), so an "all subsets regression" may be impractical for $F \gtrsim 25$.

One option is *forward selection*:

1. Begin with the null model: $\hat{y}_i = \hat{\beta}_0$

2. Using linear regression, fit all F features individually: $y_i = \hat{\beta}_0 + \hat{\beta}_f x_{if}$, where x_{if} is the fth feature of datum x_i.

3. Choose the feature with the lowest RSS, and incorporate that into the working model

4. Fit all remaining $F-1$ features on top of that (allowing the first feature to vary), finding the best (lowest RSS) two-variable model

5. Continue until e.g. adding any of the remaining features would have a minimal improvement on the model

Another option is *backward selection*, which is essentially the reverse: we start by fitting all of the features together, then repeatedly prune the one with the least significance. An intermediate method to alleviate the *greediness* of forward selection (whereby a better solution may be found by not necessarily doing whatever is the best step straight away) is called *mixed selection*: we begin by forward selection, but if one of the features already incorporated becomes insignificant it is removed in a backward selection step.

2.1.2 Regularisation

Sometimes a regression may lead to some parameter estimates being enormous, cancelling each other out somewhat in order to give a good fit but giving a weird and complex model. To mitigate against this and get simpler models, *regularisation* can be employed to penalise large parameters. For example, rather than minimising the RSS, we might instead minimise:

$$\text{RSS} + \lambda \sum_f \hat{\beta}_f^{\,2}$$

where λ is a regularisation hyperparameter which must be chosen.

2.1.3 Simpson's Paradox

A word of warning about doing linear regression on data with inherent substrata. A linear trend may appear in an overall dataset, but then disappear or even reverse when the groups are examined individually. It is therefore crucial to inspect the dataset beforehand to identify any groupings.

2.2 Classification

Classification problems involve data where each datum is a member of a particular class C_κ, where κ runs from 1 to K. Modelling can either be *discriminative* or *generative*. Discriminative models simply attempt to calculate the conditional probabilities $p(C_\kappa|\mathbf{x})$ that a given datum belongs to the classes. Generative models attempt to model the process behind the data, finding a way one might *generate* data from a given class, i.e. looking for $p(\mathbf{x}|C_\kappa)$.

2.2.1 Discriminative Models

Discriminative models divide the data space into disjoint *decision regions* separated by *decision boundaries*, thus assigning each possible datum to a specific class. They are simpler, more

interpretable, and computationally cheaper, but usually need a lot of training data to generalise well.

These models are often realised using *discriminant functions*. For a two-class problem, we might use a discriminant function:

$$y(\mathbf{x}) = \phi(\mathbf{w}^\intercal \mathbf{x} + \mathbf{w}_0)$$

(where ϕ is an *activation function*) and that if $y \geq 0$ then $\mathbf{x}$ is assigned to class C_0 and otherwise class C_1. Here, the decision boundary $y = 0$ corresponds to a $(F-1)$-dimensional hyperplane in the data space.

To find the optimal decision boundary for a two-class problem, one could use linear regression, but this would give a load of weird space in between 0 and 1, and beyond those it would also look weird as it would be assigning classes which don't exist. A better way forward is *logistic regression*, which rather than fitting a function of the form $y = \beta_0 + \beta_1 x$ fits the logistic function[3]:

$$p(x; \beta_i) = \frac{e^{\beta_0 + \beta_1 x}}{1 + e^{\beta_0 + \beta_1 x}} \qquad \Rightarrow \qquad \ln\left(\frac{p(x)}{1 - p(x)}\right) = \beta_0 + \beta_1 x$$

where the function $p(x)$ then gives the probability that the datum x lies in class C_1, rather than C_0. The coefficients β_0 and β_1 must be estimated from the data, for instance by maximising the likelihood function

$$\mathcal{L}(\beta_0, \beta_1) = \prod_{x_i \in C_0} [1 - p(x_i; \beta_i)] \prod_{x_i \in C_1} p(x_i; \beta_i)$$

– the product of the successful evaluation probabilities of the C_0 and C_1 data – with respect to the β_i.

For a larger number of classes $K > 2$, we can choose one particular class (wlog, K) as a baseline and write:

$$p(C_{\kappa<K}|x) = \frac{e^{\beta_{\kappa 0} + \beta_{\kappa 1} x}}{1 + \sum_{k=1}^{K-1} e^{\beta_{k0} + \beta_{k1} x}} \qquad p(C_K|x) = \frac{1}{1 + \sum_{k=1}^{K-1} e^{\beta_{k0} + \beta_{k1} x}}$$

2.2.2 Generative Models

Generative models model the production of data in each of the classes, i.e. what went on under the hood to create the data, the $p(\mathbf{x}|C_\kappa)$. One then estimates the probabilities $p(C_\kappa|\mathbf{x})$ that a datum $\mathbf{x}$ is in class C_κ using Bayes' Theorem: if a generative model deduces that a given class C_κ has a probability density function $f_\kappa(\mathbf{x})$, and the prior probability of a datum being from class κ is[4] π_κ, then we would give our final class probabilities as

$$p(C_\kappa|\mathbf{x}) = \frac{\pi_\kappa f_\kappa(\mathbf{x})}{\sum_k^K \pi_k f_k(\mathbf{x})}$$

For example, *linear discriminant analysis* (LDA) (a) is used where the data x only have one feature, (b) assumes that the generative distributions f_κ are Gaussians about means μ_κ, and (c) assumes that these Gaussians all have the same variances σ^2. This gives:

$$\ln p(C_\kappa|x) = [\text{const. in } \kappa] + \ln \pi_\kappa - \frac{\mu_\kappa^2}{2\sigma^2} + \frac{\mu_k}{\sigma^2} x$$

[3]This shows the logistic function for one feature but generalises naturally for $F > 1$.

[4]For example, we might estimate $\pi_\kappa = N_\kappa / N$, where N_κ is the number of instances in class C_κ in the training data

We then assign x to the class C_κ with the largest $\ln p(C_\kappa|x)$ (which is why we don't care about the leading constant). For a two-class problem ($\kappa \in \{0, 1\}$), we would hence assign x to be in class 0 provided

$$(\mu_0 - \mu_1)\left(x - \frac{\mu_0 + \mu_1}{2}\right) > 2\sigma^2 \ln\left(\frac{\pi_1}{\pi_0}\right)$$

which is eminently interpretable: for example we are less likely to assign things to class 0 if $\pi_1 \gg \pi_0$, as then the RHS will be larger. With a uniform prior $\pi_0 = \pi_1$, we assign to whichever side x is of $\frac{1}{2}(\mu_0 + \mu_1)$, which makes sense.

LDA is extendable to multifeature data, by assuming a multivariate Gaussian for $f_\kappa(\mathbf{x})$.

2.3 k-Nearest Neighbours (kNN)

kNN can be used for both classification and regression, and can handle categorical and continuous data (though usually continuous).

To classify using kNN, the idea is the following. For each new datum, if we want to classify it, let the k nearest neighbours vote on the classification.

We first need to define "nearest". One option is the Euclidean distance, though clearly this is inappropriate if one feature has much more variation than others, in which case this feature will be the most important in classification. One could overcome this by standardisation preprocessing, or alternatively by a different choice of distance formula (e.g. Manhatten).

k is a hyperparameter which requires optimisation. For $k = 1$, the nearest neighbour has all the say, which gives the noise too much influence (overfitting). This can create "islands of influence", where regions around exclave outliers get misclassified. For $k = N$, every datum will simply be assigned to the largest class (underfitting). k ought to be somewhere in between, as clearly it can influence the classification of points.

Another issue is that ties may emerge, depending on k. This can be overcome by applying a weighting factor: nearer neighbours' votes count for more, by say $1/d^2$. This needs to be finely tuned however, to avoid effectively a $k = 1$ situation.

To regress, one can look at the k nearest neighbours and take a (perhaps weighted) average.

2.4 Support Vector Machines (SVMs)

SVMs are based on Maximum Margin Classifiers, which seek a *maximal margin hyperplane*, which maximises the perpendicular distance in data space between the hyperplane and the nearest data instances of either group, which are the *support vectors*. Data which are not support vectors have no effect on the position of the hyperplane. This process requires the data to be linearly separable, otherwise misclassification is unavoidable. If not linearly separable, one might either allow this at a cost (which would be a *support vector classifier*), or apply a nonlinear kernel to project the data into a higher dimension in which the data *are* linearly separable. These kernels might be polynomials, the radial basis function, etc.; choosing the right one necessitates looking at the data first.

2.5 Decision Trees

Decision trees segment the feature space into disjoint hyperrectangles. New data instances can be classified according to the most common class in their region, or regressed using the average value in their region. They don't require any feature standardisation beforehand, and have the

advantage of allowing a mixture of categorical and continuous features. They are also readily interpretable, and seem more like how humans think. However, they are not very robust to small changes in the input data, and other methods are usually better at prediction.

A decision tree is conceptually created as follows:

1. Look at every feature, and decide both which feature is best to split on and where would be the best place to split it.
2. Create a "branch" at that split, dividing the data space into two regions.
3. Look at all of the remaining features. Repeat.
4. Finish when a given stopping rule is satisfied.

2.5.1 Choosing a Split

The general goal for a split is to increase the homogeneity of the sub-regions of data space, so we'd like a split to maximise that.

For a continuous target variable, we have a regression tree. For a given split, the prediction in each sub-region $\hat{y_r}$ is simply the mean of all the members of that region. We can then derive a loss function based on the RSS error of that region, and sum over all the regions:

$$\text{RSS} = \sum_r \sum_{x_i \in r} (\hat{y_r} - y_i)^2$$

For a categorical target (classification tree), each datum is the member of a class C_κ. There are several options for the splitting criteria. Consider a region r: let the fraction of data in region r that are in class κ be $p_{r\kappa}$. The *Gini impurity* of this region is given by:

$$G_r = \sum_\kappa^K p_{r\kappa}(1 - p_{r\kappa}) = 1 - \sum_\kappa^K p_{r\kappa}^2$$

where the sum runs over all classes. The ideal situation is for all the $p_{r\kappa}$ to be 1 and hence G_r to be 0; if randomly classified we expect $p_{r\kappa} = 1/K \forall \kappa, r$ and hence $G_r = 1 - 1/K$. The overall Gini score for a particular split is the weighted average of the two new regions r and s:

$$G = \frac{G_r}{N_r} + \frac{G_s}{N_s}$$

where N_r is the number of data in region r etc. We would thus choose the split which minimises G. Another loss function is the *classification error* $E_r = 1 - \max_\kappa p_{r\kappa}$, which in an ideal scenario would also be 0; a random classifier would also give $E_r = 1 - 1/K$. The total score for each split is calculated in the same weighted average way. Yet another is the *entropy*:

$$H_r = -\sum_\kappa^K p_{r\kappa} \log_2 p_{r\kappa}$$

Again the ideal is 0 but now the worst-case is $\log_2 K$. The choice of loss function is somewhat political, but generally we would prefer a partition to cause $p_{r\kappa} = 0.8 \to 0.9$ than $p_{r\kappa} = 0.5 \to 0.6$, because of how our brains think about purity. In this case, the Gini impurity and the entropy are superior to the classification error.

2.5.2 Stopping Rules

Small trees underfit the data; large trees overfit it; there's a bias-variance trade-off as always. The intuitive stopping rule is to stop growing the tree when the improvement (in RSS, or Gini, etc.) is below some threshold. However, because this algorithm is greedy, a bad split might be followed by a really good one.

A solution to this is *pruning*. Pruning involves growing a very large tree before pruning some of the branches. Starting with a large tree T_0, we gradually remove branches to minimise:

$$\sum_{r}^{|T|} \sum_{x_i \in r} (\hat{y}_r - y_i)^2 + \alpha |T|$$

(or the equivalent with e.g. a Gini index leading), where $|T|$ is the number of leaves of the tree = number of regions the data space is split into, and α is a complexity hyperparameter.

T_0 may have a large number of subtrees, making computation hard. A solution is to start with $\alpha = 0$ (in which the optimal tree is by definition T_0) and gradually increase it; as α gets very large eventually the tree will only have a single leaf and there will be no splits. One can choose an α by applying each of these intermediate pruned trees to some validation data (or using cross-validation), and seeing which gives the lowest error (i.e. which generalises the best).

2.5.3 Ensemble Methods

A single decision tree is usually quite bad, but combining many decision trees together can be incredibly powerful. There are several ways of combining trees.

Bagging (*bootstrap aggregating*) consists of training a series of decision trees on bootstraps of the original dataset (samples from the original datasets taken with replacement). The data not used in a particular bootstrap (*out-of-bag* data) can be used as a validation set, e.g. for pruning as above. For regression, one can then take the average of the predictions of the ensemble of trees. For classification, one can let the trees vote.

Random Forests use a similar procedure to bagging, training on bootstrap samples, but with a restriction. In training the trees, at each branching stage the split is only allowed to be on a subset of the features. For F features, one might only be permitted to split on $f \approx \sqrt{F}$ of them. This forces some trees to look at features that they otherwise wouldn't, and generates a diverse forest which is less likely to become stuck in local optima.

Boosting uses a sequence of trees to predict the error of the previous tree. The first tree predicts the target variable $\hat{y}_1$ for each datum. The second tree predicts the residual $\hat{r}_1 = y - \hat{y}_1$ (often by minimising the MSE), at which point the prediction becomes $\hat{y}_2 = \hat{y}_1 + \lambda \hat{r}_1$, where λ is a small (~ 0.01) "shrinkage" parameter. After B trees, the overall prediction is then:

$$\hat{y}_B = \hat{y}_1 + \lambda \sum_{b=1}^{B-1} \hat{r}_b$$

This has been extended to great effect in systems such as *AdaBoost* and *XGBoost*.

An issue with ensemble methods is the loss of interpretability. Some can be retained by calculating the importance of each feature across the whole ensemble, e.g. the average amount that the loss improves as a result of splits on that feature.

3 Unsupervised Learning

Sometimes we don't have labels for our data, and the best we can do is group together similar-looking data points.

3.1 Dimensionality Reduction

One way of grouping together data with many features is to perform a dimensionality reduction: projecting the data points into a lower-dimensional space while preserving any clustering inherent in the data space. That is, preserving as well as possible the (relative) pairwise distances of all the data points. This is not possible to do precisely: in 3D the four vertices of a tetrahedron are equidistant to all the others, but good luck trying to get four equidistant points in 2D. Hence the "as well as possible". Dimensionality reduction exploits the common phenomenon that data rarely occupy every corner of the data space: data usually exist on or near a submanifold of the data space, as features often have complicated interrelations.

3.1.1 Principal Component Analysis (PCA)

Principal components (PCs) are orthonormal vectors: the mth PC is along a line which best fits the data while being perpendicular to the first $m-1$ PCs. Another way of describing this is to find a line where, when all the data are projected along this line, there is the largest variance of distances along the line. If we choose the vector $\mathbf{w}$ as the first PC, the projection of data point $\mathbf{x}_i$ along the line $\mathbf{w}$ is $\mathbf{w}\cdot\mathbf{x}_i$, so the variance which we seek to maximise is given by $V[\mathbf{w}\cdot\mathbf{x}]$. Now $E[\mathbf{w}\cdot\mathbf{x}] = \mathbf{w}\cdot\bar{\mathbf{x}}$, so

$$V[\mathbf{w}\cdot\mathbf{x}] = \frac{1}{N}\sum_i (\mathbf{w}\cdot\mathbf{x}_i - \mathbf{w}\cdot\bar{\mathbf{x}})^2 = \frac{1}{N}\sum_i [\mathbf{w}\cdot(\mathbf{x}_i - \bar{\mathbf{x}})]^2 = \mathbf{w}^\intercal\mathbf{Q}\mathbf{w}$$

where $Q_{jk} = \frac{1}{N}\sum_i (\mathbf{x}_{ij} - \bar{\mathbf{x}}_j)(\mathbf{x}_{ik} - \bar{\mathbf{x}}_k)$ is (proportional to) the covariance matrix of the data $\mathbf{x}_i$. Being a quadratic form, extremising this variance is equivalent to finding the eigenvectors of the quadratic matrix. The first PC is then the eigenvector with the largest eigenvalue. Subsequent PCs are eigenvectors with successively smaller eigenvalues. $\mathbf{Q}$ is a positive semi-definite matrix, so its eigenvalues are all at least 0, and are only 0 if there is a direction in data space where no data lie, i.e. the data points all lie exactly on a hyperplane in the data space.

One can plot a *scree plot*, of λ_m against m. The first n PCs account for a fraction $\sum_m^n \lambda_m / V[\mathbf{x}]$ of the variance in the data, where we note that the sum of *all* the eigenvalues $\sum_m^N \lambda_m = \mathrm{Tr}\ \mathbf{Q} = V[\mathbf{x}]$.

PCA is sometimes sensitive to outliers; "robust PCA" variations exist to mitigate this. An advantage of PCA is that new points can easily be mapped: you can just project the new data points against the PCs already found, without needing to rerun the whole thing again (though it will then be slightly suboptimal).

When data is not distributed roughly linearly, but for example concentrically, PCA is bad at finding a good submanifold. An alternative is to use *kernel PCA*, where the data are mapped using a *kernel map*, a vector function $\mathbf{\Phi}(\mathbf{x})$. We then use PCA but instead of $\mathbf{Q}$ we look for the eigenvalues/vectors of the matrix $\mathbf{K} = \mathbf{\Phi}^\intercal\mathbf{\Phi}$. This might be a projection into a higher dimension in which the data are linearly separable.

3.1.2 Nonlinear Dimensionality Reduction

There exist various other dimensionality reduction techniques which are nonlinear in nature.

Isomap constructs a locally-roughly-Euclidean neighbourhood graph, allowing twisted-up submanifolds (e.g. Swiss roll) to be unfolded.

***t*-distributed stochastic neighbour embedding (*t*SNE)** similarly converts data points $\mathbf{x}$ into embedded vectors $\mathbf{y}$ in 2 or 3 dimensions by considering the probabilities $p_{i|j}$ that $\mathbf{x}_j$ would consider $\mathbf{x}_i$ to be a neighbour, according to how far away they are. This is typically weighted according to a Gaussian $\propto \exp(-\|\mathbf{x}_i - \mathbf{x}_j\|/2\sigma^2)$ where σ^2 is a hyperparameter called the *perplexity*. High perplexity means that points will be antisocial and not have many neighbours; low perplexity gives well-connected neighbourhoods. The procedure minimises the difference between the neighbourhood distributions in the original data space and the projected data space, which is quantified by the Kullback-Leibler divergence.

Uniform Manifold Approximation and Projection (UMAP) is another method, similar to *t*SNE, but more mathematically complicated. UMAP and *t*SNE are generally better at nonlinear dimensionality reduction than PCA, but are both non-deterministic and don't allow new points to be mapped in.

3.1.3 Self-Organising Maps (SOMs)

SOMs are 2D rectangular or hexagonal grids in the data space, trained to distort over the data space. Initially, the SOM nodes are chosen at random, usually a flat grid somewhere in data space. A data point is picked at random, and the node closest to this data point is called the *best-matching unit*. All of the SOM nodes are then dragged by varying amounts towards this data point, according to some *neighbourhood function*. The neighbourhood function might, for example, cause nodes to move differently according to how far away the node in question is from the best-matching unit.

3.2 Clustering

Assuming that the data were generated from a number of different classes, we can try and separate them out. Data points that are nearby in data space should ideally be in the same cluster.

3.2.1 *k*-Means

In k-means clustering, each cluster is represented by a *centroid* $\mathbf{c}_\kappa$ for that cluster. Initially, these are K random points in the data space. Each data point is then assigned to a cluster according to which centroid it is closest to. Then, the centroids are re-plotted as the mean of all data points just assigned to their respective clusters. This assignment–update procedure (called *Lloyd's algorithm*) is repeated until convergence.

Variations on k-means might use distance metrics other than Euclidean to assign clusters, or may use a different kind of average rather than the mean (e.g. k-medians).

k-means ultimately seeks to minimise the following loss function:

$$\sum_{\kappa=1}^{K} \sum_{\mathbf{x}_i \in C_\kappa} \|\mathbf{x}_i - \mathbf{c}_\kappa\|^2$$

but it usually instead finds a local minimum, and is quite sensitive to the initialisation. Rather than randomly initialising the centroids, one might choose specific data points to be the initial centroids, or do a random assignment step first.

3.2.2 Fuzzy c-means

Fuzzy c-means is an example of a *soft clustering* method, where rather than strictly assigning data points to one cluster, one assigns each data point k weights to each cluster. These weights can be thought of as probabilities that the data point belongs to each cluster. One can make a soft clustering method hard by "fluffing", which consists simply of picking the cluster with the highest weight.

As with k-means, there are assignment and centroid-update steps, though the assignment step is now "fuzzy". The data point $\mathbf{x}_i$ is assigned weights $w_{i\kappa}$ to each of the K clusters, where

$$w_{i\kappa} = \left[\sum_{k=1}^{K} \left(\frac{\|\mathbf{x}_i - \mathbf{c}_\kappa\|}{\|\mathbf{x}_i - \mathbf{c}_k\|} \right)^{2/(m-1)} \right]^{-1}$$

where m is a "fuzziness hyperparameter": larger m leads to $w_{i\kappa}$ being more similar for each κ, that is, more agnostic assignment. We see that if $\mathbf{x}_i$ is closer to $\mathbf{c}_\kappa$ than to the rest of the $\mathbf{c}_k$, then the fraction will be smaller and the weight w_{ik} will be larger. The position of the centroid is then updated to be the *weighted* average of all the data points:

$$\mathbf{c}_\kappa = \frac{\sum_{i=1}^{N} w_{i\kappa}^m \mathbf{x}_i}{\sum_{i=1}^{N} w_{i\kappa}^m}$$

3.2.3 Hierarchical Clustering

Hierarchical clustering seeks a hierarchy of sub-clusters which represent the data well.

Agglomerative clustering begins with each data point in its own cluster, and at each stage merges the two clusters which have the best *linkage*, to form a binary tree. The linkage describes the similarity between two clusters, and there are many options. We could use:

- The distance between cluster centroids
- The average, or minimum, or maximum pairwise distance between members of the two clusters
- The clusters which, if merged, would cause the minimum increase in the average distance to the new centroid (*Ward linkage*)

Divisive clustering works the other way around: all the data start in one big cluster, which is hollowed out by progressively making smaller clusters by removing outliers.

3.2.4 Gaussian Mixture Models (GMMs)

GMMs are an example of *generative models*, where we assume that the data have been drawn from some parametrised distribution $P(\mathbf{x}; \boldsymbol{\theta})$, and in a Bayesian way try to find the parameters $\boldsymbol{\theta}$, by essentially maximum-likelihood estimation. If we assume that the data have come from one of a number of different classes, this distribution would be the sum of a number of

distributions $P = P_1 + P_2 + \cdots + P_\kappa + \cdots + P_K$, each with their own parameters – for example, GMMs assume the distribution is the sum of several Gaussian distributions, which may be differently normalised. Each data point can then be assigned probabilities P_κ for each class: this is therefore a soft clustering method.

With enough distributions one can describe any distribution, so they are powerful, but they are quite difficult to optimise. This is partly because the data may actually not have come from normal distributions, or because the GMM is overfit with the assumption of too many models.

3.2.5 Spectral Clustering

Consider converting a dataset into a graph, where each data point is a node. We might choose to connect nodes if their respective data points are within a certain distance of each other, or if they are one of the k-nearest neighbours of each other. We can then define an *adjacency matrix* $\mathbf{A}$, where $A_{ij} = 1$ if $\mathbf{x}_i$ and $\mathbf{x}_j$ are connected, and 0 otherwise. Another matrix describing this graph is the *diagonal matrix* $\mathbf{D}$ of the graph, whose diagonal terms are the degrees (number of edges) of the nodes. (This is also equal to the sum of each row/column of the adjacency matrix: $D_{ii} = \sum_j A_{ij}$.) The *Laplacian matrix* of the graph is $\mathbf{L} = \mathbf{D} - \mathbf{A}$; the *normalised Laplacian matrix* $\boldsymbol{\ell} = \mathbf{I} - \mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}$ apparently has all its diagonal terms equal to 1. We then find the m eigenvectors of $\boldsymbol{\ell}$ corresponding to the m smallest positive eigenvalues, and put them into a matrix $\mathbf{V}$, which will be of shape $N \times m$. These eigenvectors (of shape $1 \times N$) can be thought of as important features in the data space, so if we transform the data points using this matrix $\mathbf{V}$ we will obtain vectors in m-dimensional space (where we would typically choose $m < \dim\mathbf{x}$) which are apparently easier to cluster than the original data points. If we just look at the first eigenvector (called the *Fiedler* vector), then each data point is mapped to a scalar value (effectively their projections onto the Fiedler vector), and we are then just clustering scalars which can be done with the humble histogram.

3.2.6 DBSCAN

DBSCAN (density-based spatial clustering of applications with noise) clusters together points which can be easily reached from each other by making small jumps between data points, and labels points which are far away from all others as outliers. DBSCAN classifies all points as a *core point*, a *border point*, or an outlier, in the following way:

- If there are at least `minPts` other data points within a distance ϵ in data space of the point in question, that point is classed as a core point. These points are in regions of high density in data space
- If there are less than `minPts` other data points within ϵ, but there is still a core point that *is* within ϵ, then that point is classed as a border point.
- Points not classified as core points or border points are classified as outliers.

where ϵ and `minPts` are hyperparameters which must be optimised.

DBSCAN is nice because there is no need to specify the number of clusters, and one can find clusters of arbitrary shape. However, if there are two clusters where one is much more dense than the other, DBSCAN may not be able to identify the latter as a cluster unless ϵ is tuned very finely.

3.2.7 Evaluating Clustering

There is no universal approach for how good a clustering algorithm has performed, as this is ultimately subjective. *Internal evaluation* uses the data that were clustered to evaluate the clustering (as opposed to external evaluation which uses data that were not used in clustering, e.g. ground-truth labels). This typically involves quantifying similarity between members of the same cluster, and between members of different clusters; beware that if the clustering was performed using the same similarity metric as one is using to evaluate the clustering, then this will naturally be evaluated pretty well.

The *silhouette coefficient method* assigns each data point a silhouette value $s_i \in [-1, 1]$ according to how similar it is to other members of its own cluster C_i, compared to members of the next-most-similar cluster C_j. Each data point is assigned first *cohesion* and a *separation* values:

$$a_i = \frac{1}{|C_i| - 1} \sum_{k \in C_i} \|\mathbf{x}_i - \mathbf{x}_j\| \qquad b_i = \frac{1}{|C_j|} \sum_{k \in C_j} \|\mathbf{x}_i - \mathbf{x}_k\|$$

where the next-most-similar cluster C_j is decided by whichever cluster gives the smallest b_i. The silhouette coefficient is then given by $s_i = 1 - a_i/b_i$ if $a_i < b_i$ (indicating that this point is in the right cluster), and $s_i = -1 + b_i/a_i$ if $a_i > b_i$ (indicating this point is in the wrong cluster). The overall silhouette coefficient S is then the mean of the s_i.

For clustering algorithms where the number of clusters K is a hyperparameter, the *elbow method* can help you decide. Run the clustering for several values of K, and evaluate the fraction of explained variance – that is, the ratio of inter-cluster variance to total variance. $K + 1$ clusters will always improve this quantity with respect to K clusters, but if we are starting to overfit then the improvement will be much less than from going from $K - 1$ to K, in which case K would be the ideal number of clusters. Plotting the explained variance fraction against K, we would find a bit of an "elbow" at the ideal value of K, though again this is often subjective.

If the clustering algorithm is generative and has a likelihood function $\mathcal{L}$ (e.g. GMMs), we can use *information criteria* to assess the clustering. For example, the Akaike information criterion is given by:

$$\text{AIC} = 2K - 2 \ln \mathcal{L}$$

where K is the number of clusters. A lower AIC means a better model: we see that large K and low $\mathcal{L}$ mean a bad model.

3.2.8 Problems with Clustering

- Clustering techniques sometimes overzealously find clusters which are not really there, but are merely a consequence of noise.
- Clustering is often very sensitive to the choice of hyperparameters and distance metrics
- Distance metrics are sometimes quite expensive to calculate, especially with high-dimensional data. Further, many algorithms require calculating the distance metric relative to all of the other points in the dataset, which can make the algorithm complexity $\sim N^2$
- The curse of dimensionality – the phenomenon whereby high-dimensional data are always quite far away from each other just because there are a lot of dimensions to add up – means that clustering gets difficult in higher dimensions

- Hard clustering algorithms can be upset by the presence of outliers, as they are forced to try and assign them a cluster membership.
- Clustering is often not stable to perturbations, e.g. the removal of a random subset of the data

4 Outliers & Missing Data

Outliers are data points which are considerably different from the rest of the dataset. They are sometimes a nuisance (e.g. due to noise), but are sometimes exactly what we are looking for (e.g. rare diseases, lensed high-redshift quasars [Byrne+24]).

Outliers can be identified by data-based procedures (e.g. DBSCAN) or model-based procedures (e.g. points which significantly distort the model).

Missing data (e.g. missing fields in partially-complete data)may be:

- Missing completely at random (MCAR): there is no known reason why the data should be missing. This is annoying but at least introduces no bias
- Missing at random (MAR): the data are missing as a probabilistic result of the values of some other features. For example, if feature 2 is more likely to be missing if feature 1 is positive (but still recorded), feature 2 is MAR If the data are MAR, they can often be *imputed* by a generative model
- Missing not at random (MNAR): the data are missing because of its value. For example, feature 2 might be missing because its true value is negative and that breaks the recording device or something

Missing data may be omitted if they're being awkward, or alternatively we may decide not to use a particular feature if it is absent from some of the data points.