Data Science Toolbox Portfolio

MATHM0029

Thomas Pagulatos

School of Mathematics
University of Bristol
January 2024

Suppose we have a linear regression model of the form $\mathbf{Y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}$, where $\mathbf{X}$ is a real $N \times P$ matrix, $\boldsymbol{\beta} = (\beta_1, ..., \beta_P)$ is a P–vector of parameters, $\mathbf{Y} = (y_1, ..., y_N)$ is an N–vector of observations, and $\boldsymbol{\epsilon}$ is the error. The least squares estimate of $\boldsymbol{\beta}$ is given by $\widehat{\boldsymbol{\beta}} = (\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'\mathbf{Y}$ [1]. Below, we use the resources of the 'Machine Learning' course at Carnegie Mellon University from fall 2005 [2].

The estimator for $\mathbf{Y}$ is given by $\widehat{\mathbf{Y}} = \mathbf{X}\widehat{\boldsymbol{\beta}} = \mathbf{H}\mathbf{Y}$, where $\mathbf{H} = \mathbf{X}(\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'$ is the 'hat matrix.'

The least square estimator minimises the sums of squared errors (SSE), given by

$$\text{SSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2, \tag{1}$$

The leave-one-out cross-validation (LOOCV) statistic is defined by

$$\text{CV} = \frac{1}{N}\sum_{i=1}^{N}\left(y_i - \hat{y}_i^{(-i)}\right)^2, \tag{2}$$

where $\hat{y}_i^{(-i)}$ is the predicted value of y_i obtained after deleting the i–th observation. That is, it minimises $\sum_{j\neq i}(y_j - \hat{y}_i^{(-i)})^2$. The estimator of $\mathbf{Y}$ after removing the i–th observation is given by $\widehat{\mathbf{Y}}^{(-i)}$.

Recall that $\widehat{\mathbf{Y}} = \mathbf{H}\mathbf{Y}$. Thus, the i–th element of $\widehat{\mathbf{Y}}$ is $\hat{y}_i = \sum_j h_{ij}y_j$.

It turns out that $\widehat{\mathbf{Y}}^{(-i)}$ is also the estimator that minimises the sum of squared errors for $\mathbf{Z}$, whose elements are

$$z_j = \begin{cases} y_j, & j \neq i; \\ \hat{y}_i{}^{(-i)}, & j = i. \end{cases} \tag{3}$$

Proof. We have that $\mathbf{Y}^{(-i)} \triangleq \operatorname{argmin}\sum_{j\neq i}(y_j - \hat{y}_i^{(-i)})^2$. Furthermore, $\sum_{j\neq i}(y_j - \hat{y}_i^{(-i)})^2$ is equivalent to $\sum_j(z_j - \hat{y}_i^{(-i)})^2$ since $z_i = \hat{y}_i{}^{(-i)}$. Hence, $\widehat{\mathbf{Y}}^{(-i)} = \operatorname{argmin}\sum_j(z_j - \hat{y}_i^{(-i)})^2$. □

Now, we claim that $\hat{y}_i{}^{(-i)} = \hat{y}_i - h_{ii}y_i + h_{ii}\hat{y}_i{}^{(-i)}$.

Proof. From above, we know that $\hat{y}_i = \sum_j h_{ij}y_j$, and $\hat{y}_i{}^{(-i)} = \sum_j h_{ij}z_j$. Furthermore, $\hat{y}_i - \hat{y}_i{}^{(-i)} = \sum_j h_{ij}(y_j - z_j) = h_{ii}y_i + h_{ii}\hat{y}_i{}^{(-i)}$. Therefore, $\hat{y}_i{}^{(-i)} = \hat{y}_i - h_{ii}y_i + h_{ii}\hat{y}_i{}^{(-i)}$. Note that h_{ii} are the diagonal elements of $\mathbf{H}$. □

Plugging this result into the definition of the LOOCV statistic above yields the following result.

$$\text{CV} = \frac{1}{N}\sum_{i=1}^{N}\left(\frac{y_i - \hat{y}_i^{(-i)}}{1 - h_{ii}}\right)^2. \tag{4}$$

Therefore LOOCV can be computed after estimating the model once on the complete data set.

There are various implications to the fact that the LOOCV error can be cheaply computed for linear regression for datasets such as the Temperature and Diamonds datasets in R. Namely, a cheap computation means that LOOCV, which is a robust cross-validation method, is more easily implemented in models. This can lead to improved model validation, as well as mitigating overfitting and aiding feature selection and parameter tuning.

The unweighted pair group method with arithmetic mean (UPGMA), also known as average linkage, is an algorithm used in data science primarily for hierarchical clustering and phylogenetics – the study of evolutionary relatedness among groups of organisms [3]. UPGMA clusters data points in a bottom-up fashion based on distance. At each step, the closest pair of clusters are merged, forming a new cluster. As suggested by its name, the algorithm relies on measuring the arithmetic mean of distances between cluster points and is therefore 'more robust than single-linkage methods' [4].

Clustering plays a crucial role in the automatic processing of large datasets [4]. Hierarchical clustering methods classify data points into a nested structure of clusters, forming a tree. Its uses range from classifying patterns in images to stock prediction [4].

Suppose we have two clusters of data points $\mathcal{A}$ and $\mathcal{B}$ of size (cardinality) $|\mathcal{A}|$ and $|\mathcal{B}|$ respectively. Let $d(x, y)$, or d_{xy}, represent the distance between a pair of data points, x and y, in $\mathcal{A}$ and $\mathcal{B}$, respectively. The mean distance between elements across clusters is given by

$$\frac{1}{|\mathcal{A}| \cdot |\mathcal{B}|} \sum_{x \in \mathcal{A}} \sum_{y \in \mathcal{B}} d(x, y). \tag{5}$$

At each step of clustering, the distance between the cluster $\mathcal{A} \cup \mathcal{B}$ and a new cluster $\mathcal{X}$ is given by

$$d_{(\mathcal{A} \cup \mathcal{B}),\mathcal{X}} = \frac{|\mathcal{A}| \cdot d_{\mathcal{A},\mathcal{X}} + |\mathcal{B}| \cdot d_{\mathcal{B},\mathcal{X}}}{|\mathcal{A}| + |\mathcal{B}|}. \tag{6}$$

An issue is that UPGMA, in its 'vanilla' form, uses an entire similarity (or distance or dissimilarity) matrix in memory. Such a requirement is prohibitive to scaling the algorithm to large datasets. The number of pairwise relations grows quadratically in relation to the number of cluster points [4]. The distance matrix is scanned and recalculated at each iteration. Therefore, holding the matrix in memory becomes infeasible in datasets of even moderate size [5]. These factors render the UPGMA algorithm impractical for many applications. A naive implementation of UPGMA has a time complexity of $O(N^3)$ [6], but this can be reduced to an optimised time complexity of $O(N^2)$ [7], such as in *sparse* UPGMA [4].

Some notation and definitions [4]

Let $\mathbb{V}$ be the set of data points (or vertices), and $\mathbb{E} \subset \mathbb{V} \times \mathbb{V}$ be the edge set. Note that $d : \mathbb{E} \to \mathbb{R}^+$. UPGMA inputs an undirected graph $G = (\mathbb{V}, \mathbb{E})$. Formally, we say the $\mathbb{E}$ is *sparse* if 'not all possible pairs exist in the input graph' [4]. That is, there exists $i, j \in \mathbb{V}$ such that $e_{ij} \notin \mathbb{E}$, where e_{ij} denotes the edge from point i in one cluster to point j in another.

Sparse UPGMA provides an improvement over standard UPGMA since not *all* of the pairwise distances between cluster points are needed (or known) – such as in protein sequence data [4]. By leveraging the sparsity of the data, sparse UPGMA operates more efficiently than standard UPGMA and uses less memory since it does not need to store the entire similarity matrix in memory. Therefore, it allows for the clustering of larger sets, and it is also efficient for non-sparse inputs [4].

Equation (1) is not well-defined for sparse inputs. Therefore, the domain of d is expanded to all vertex pairs by introducing a missing value completion rule [4]. Putting $\psi := \max(\mathbb{E})$, the distance $d : \mathbb{V} \times \mathbb{V} \to \mathbb{R}^+$, over an expanded domain, is given by d_{ij} if $e_{ij} \in \mathbb{E}$ and ψ otherwise.

Sparse UPGMA uses binary heaps (which only handle necessary subsets of the data) to structure the data [4], therefore reducing time complexity and memory requirements. Efficient clustering can then be done on large, sparse data.

In summary, sparse UPGMA provides a more efficient method for hierarchical clustering of large datasets and overcomes the limitations of higher memory usage and computational complexity encountered in vanilla UPGMA.

Kernel Methods in machine learning are pivotal in transforming input data for machine learning models. Using methods such as 'polynomial kernels,' input data can be transformed into a higher-dimensional feature space – essential for data not linearly separable in the original feature space. By mapping the data into this new feature space, it may be possible to apply linear separation methods to inherently non-linear datasets, which can improve the performance of models.

As discussed in the paper 'Kernel Methods in Machine Learning' [8], the polynomial kernel kernel is defined by its degree and coefficients. Linear algorithms such as support vector machines (SVM) can work on the data in the transformed space, addressing the challenges of non-linear problems. Adjusting the degree of the polynomial controls the decision boundary.

Suppose we have some empirical data in the form $(x_1, y_1), ..., (x_n, y_n) \in \mathcal{X} \times \mathcal{Y}$, where $\mathcal{X}$ is a non-empty set of predictor variables, x_i, and each $y_i \in \mathcal{Y}$ is a corresponding target variable, or class, where $i \in \{1, 2, ..., n\}$.

In the case of binary classification, given unseen data $x \in \mathcal{X}$, we want to be able to predict the corresponding target class $y \in \{0, 1\}$. That is, we require a y such that (x, y) is in some way similar to the training data. Therefore, we need a way of measuring the similarity in $\mathcal{X}$ and $\{0, 1\}$. Since two target values can only be either the same or different. In the former measure, we must have a function $K : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$, $(x, x') \mapsto K(x, x')$ such that for all $x, x' \in \mathcal{X}$,

$$K(x, x') = \langle \phi(x), \phi(x') \rangle = \phi(x)^T \phi(y). \tag{7}$$

K is a kernel corresponding to an inner product in the feature space, and ϕ is feature map. Using such kernels as a similarity measure can enable one to create algorithms in dot product spaces [8]. Computing the dot product in high-dimension feature spaces can be computationally challenging, and the kernel greatly aids in performing these computations. The class of kernels that satisfy the correspondence in (1) are *positive definite* (Definition 3 [8]).

Definition 4.1 For a polynomial of degree d, where $d \in \mathbb{N}$, the polynomial kernel is defined as

$$K(x, y) = (x^T y + c)^d \tag{8}$$

where x and y are $n-$dimensional vectors in the input space of features and $c \geq 0$ is a parameter that balances the influence of higher and lower order terms in the polynomial. When $c = 0$ the kernel is called homogeneous [9].

Let us consider the homogeneous polynomial kernel of degree two (i.e. the quadratic polykernel). Consider also a dataset with features that are not linearly separable, such as that defined in the blog 'Kernels and feature maps: theory and intuition' [10]. Let $x = (x_1, x_2)^T$ and $x' = (x'_1, x'_2)^T$. Then

$$\begin{aligned} K(x, x') &= (x_1 x'_1 + x_2 x'_2)^2 = 2x_1 x'_1 x_2 x'_2 + (x_1 x'_1)^2 + (x_2 x'_2)^2 \\ &= \begin{pmatrix} \sqrt{2} x_1 x_2 & x_1^2 & x_2^2 \end{pmatrix} \begin{pmatrix} \sqrt{2} x'_1 x'_2 & x_1'^2 & x_2'^2 \end{pmatrix}^T =: \phi(x)^T \phi(x') \end{aligned} \tag{9}$$

where ϕ is our feature map defined by the mapping $(x_1, x_2) \to (z_1, z_2, z_3)$; $z_1 = \sqrt{2} x_1 x_2$, $z_2 = x_1^2$, $z_3 = x_2^2$. This 'kernel trick' has mapped original points into a higher-dimensional feature space where the data are now linearly separable. The SVM algorithm can then be applied to the transformed data to create a linear hyper-plane boundary (or decision boundary).

We saw that polynomial kernels can be used in binary classification. Other specific uses include multi-classification, regression, and image processing. In the latter case, polykernels can be used to model the complex non-linear relationships that exist in image data, such as those between pixel patterns and image categories. Polykernels are a vital tool in machine learning; however, care must be taken since their use often requires careful parameter tuning in order to avoid overfitting.

Bagging (or bootstrap aggregation) is an ensemble learning method that can mitigate variance in noisy datasets. Suppose that we have a dataset of size n and divide it into a training set and a validation set with a ratio of 70:30, which is typical in modeling. In bagging, the training set is randomly sampled without replacement, meaning that each data point can be chosen more than once, creating diverse samples – this is bootstrapping. We call these data 'bags,' since they are not strictly speaking subsets. The bags are typically of size $n' < n$ and are used to train (independently and in parallel) different models in an ensemble. These models can often be weak (or base) learners. Finally, each model's output is aggregated meaningfully to provide a final output, such as averaging the individual outputs in regression or taking a majority vote in classification.

The bagging algorithm, with the three basic steps of bootstrapping, parallel training, and aggregation, was introduced by Leo Breiman in 1996 [11]. While bagging can significantly improve a model's performance (e.g. by reducing overfitting and enhancing accuracy), it is not free of costs and limitations. We shall explore these next and discuss whether bagging can be useful.

A common problem in data science is overfitting, which prevents a model from generalising accurately to new unseen data. High-variance models are more prone to overfitting. By aggregating outputs from multiple models, each trained on slightly different datasets, bagging can decrease overall variance, reducing overfitting and improving accuracy, which is particularly beneficial for models such as decision trees [12].

Bagging increases model stability and leads to more robust predictions through averaging or majority voting. Here, outlier results and errors from individual models are cancelled out, outvoted, or absorbed into an average weighting that increases accuracy and reduces variance overall.

Since each model is built independently and trained on a unique bag of data, the process is highly parallelisable. Employing parallel computing can result in a significant reduction in training time, and improve efficiency [13]. Since many modern computational models are complex and sophisticated, parallel computation can be beneficial.

The use of bagging comes with its share of costs and limitations. Although parallelisation can improve training time with large datasets (and more complex models), it can, in fact, lead to an increase in computational resources, costs, and training time [14]. Moreover, it will likely result in higher memory usage, especially with many models in an ensemble. In addition, aggregating the results from multiple models can result in a loss of interpretability. For example, a single decision tree is easy to interpret, but a vast ensemble may not.

If the base model is already robust and does not suffer from overfitting, bagging may not be beneficial, as in low-variance linear models [15]. Bagging may not be helpful for inherently biased base models since it is more effective at reducing variance than bias. In addition, improvements in bagging can be marginal and there are diminishing returns when using more models. Moreover, choosing the optimal number of models in an ensemble is a hyperparameter that can require time-consuming and computationally expensive fine-tuning. For large data sets and complex models, a slight increase in accuracy with a significant increase in cost is potentially unjustifiable.

In summary, we have seen that bagging has various vital benefits, including mitigating overfitting, increasing accuracy, and parallelisation. In contrast, we have identified some limitations of the bagging, which are not insignificant. These include increases in computational cost, high memory usage, possible ineffectiveness, and the loss of interpretability in some instances. It is essential to weigh these factors against the specific needs of a project, such as the nature of the data and any constraints, e.g. resource availability and budget. With careful consideration of the trade-offs discussed, bagging can be a valuable tool for enhancing the performance of a model.

Here, we look at how principal component analysis (PCA) can be used to construct features for decision trees, and we consider its application to a dataset for classification using a decision tree model.

'Decision trees align decision boundaries with features,' meaning that decision trees (used in classification and regression) make decisions based on the values of features. Each node has a question or condition on a feature, and each subsequent branch represents the outcome of that test. In classification, the decision boundary is the region in the feature space where the model favours predicting one class over another. Each feature represents a dimension in the feature space. Depending on the dimensions of the data, decision boundaries separating the classes could be a line, a plane, or even a hyperplane – we call these axis-parallel splits since they partition the space parallel to feature axes [16].

PCA can be used in conjunction with decision trees. In some cases, the original features do not effectively capture the variance of the data. PCA transforms the feature space such that the new uncorrelated axes (principal components) are aligned in directions with maximum variance across all linear transformations of the dataset. These directions often align with the greatest 'information gain.'

Furthermore, PCA is often used as a technique for dimension reduction. Fewer decision layers can make decision trees in high-dimension datasets run faster and mitigate overfitting, a problem common in decision trees where many additional features are less relevant and contribute to noise in the data. Overall, this can increase the accuracy of a model.

Mathematically, we take an $N \times n$ mean-centred data matrix, $\mathbf{X}$, with n features and N rows of data. Then, we create an $n \times n$ covariance matrix $\mathbf{S}$ such that $\mathbf{S} = \mathbf{X}^\top \mathbf{X}/(n-1)$. The matrix $\mathbf{S}$ is symmetric and, therefore, diagonalisable, so we can write $\mathbf{S} = \mathbf{V}\mathbf{D}\mathbf{V}^\top$, where $\mathbf{V}$ is the eigenmatrix whose columns are given by eigenvectors $\mathbf{v}_i$ corresponding to eigenvalues λ_i and $\mathbf{D}$ is a diagonal matrix with entries λ_i corresponding to each $\mathbf{v}_i$, which we call principal axes. Principal components are projections of the original data values onto these axes, and act as new features. The principal components are arranged such that the first few explain the most variance in the data. Finally, we can build a decision tree that uses these principal components as features.

Now, let us look at an application of this technique in R. In Appendix 1, we can see that PCA can be used to successfully reduce the dimensions of a dataset and create a decision tree model. The eight features in the 'Pulsar' dataset were used to classify whether a celestial object is a pulsar. A pulsar is a fast rotating neutron star that emits regular pulses of electromagnetic radiation [17]. After applying PCA to the pulsar dataset, we can see that the first five principal components explain 97.6% of the variance. Therefore, it is reasonable to perform dimension reduction by reducing the dimension by three to five principal component features. After applying a decision tree model to the PCA-transformed data, the classifier recorded an accuracy of 96.81%. In comparison, a model trained only on scaled data saw an accuracy of 97.96%. This is not significantly different, and we can therefore see how PCA can reduce the number of dimensions needed to train a model, while still maintaining high accuracy. If a much larger dataset were used, along with a more sophisticated classification model, this use of PCA is likely to reduce training time.

These theoretical notions notwithstanding, one should note that the blind application of PCA will not always lead to better-performing models, especially in scenarios where the original features are already well-aligned with class boundaries, or the variance explained by the features is pretty equally distributed. In the latter case, PCA cannot successfully reduce the dimensions of the data. In Project 1, we saw a case where PCA was not productive in the pursuit of dimension reduction and the random forest tree classifier did not perform better with its use.

Layer-wise relevance propagation (LRP) is a technique used in deep learning for interpreting neural network decisions. LRP works by analysing how input features contribute to outputs and decisions. Through a backward propagation technique, the output is passed through the network's layers giving scores to each neuron based on their contribution to the output, therefore revealing the most influential aspects of the input data to the network's decision-making process. LRP is crucial for elucidating and interpreting sophisticated neural network models. How does it work?

Figure 11 in section 5 of 'Methods for interpreting and understanding deep neural networks' [18] provides a clear visual depiction of the LRP procedure applied to a deep neural network.

Firstly, a forward pass of the network is performed, collecting the activations at each layer. Next, according to propagation rules (Section 5.1 [18]), the score generated at the output of the network is sent back through the network. Suppose j and k are the indices for the neurons of two consecutive layers where j is lower than k, and R_k is the relevance of neuron k for predicting the outcome $f(\mathbf{x})$. The share of R_k that is redistributed to neuron j is $R_{j\leftarrow k}$. The relevance of neuron k is redistributed to the lower layer in equal amounts, so we have the conservation property $\sum_j R_{j\leftarrow k} = R_k$, which is analogous to Kirchoff's conservation laws for electrical circuits. Similarly, $\sum_k R_{j\leftarrow k} = R_j$. Combining these equations shows that relevance is conserved between layers. Proving this is simple: $\sum_j R_j = \sum_j \sum_k R_{j\leftarrow k} = \sum_k \sum_j R_{j\leftarrow k} = \sum_k R_k$. This property holds globally [18], $\sum_{i=1}^{d} R_i = \ldots = \sum_j R_j = \sum_k R_k = \ldots = f(\mathbf{x})$. This shows that LRP decomposes the output, $f(\mathbf{x})$, and relates it to the input variables. The implementation of the conservation property of LRP must follow specific rules.

One of the most common applications of LRP is that of deep neural networks with rectifier (ReLU) non-linearities. Specific examples include image recognition models such as VGG-16 [19] and Inception v3 [20]. Such deep rectifier models have neurons of the type $a_k = \max(0, \sum_{0,j} a_j w_{jk})$. This sum is over all lower-layer activations a_j and a neuron that represents the bias. Several relevance distribution rules, each suited to specific types of layers in neural networks or network architectures. Some common rules are listed.

LRP-0: $R_j = \sum_j \frac{a_j w_{jk}}{\sum_{0,j} a_j w_{jk}} R_k$. This rule redistributes the relevance proportionally to the amount each input neuron contributes to the neuron activation in the subsequent layer. It has been shown [21] that uniformly applying this rule across an entire neural network results in noise, hence more robust propagation rules are required.

LRP-$\boldsymbol{\epsilon}$: $R_j = \sum_j \frac{a_j w_{jk}}{\epsilon + \sum_{0,j} a_j w_{jk}} R_k$. This is very similar to LRP-0, but a small term ϵ is added to the denominator to absorb some relevance when the activation of neuron k is influenced by weak or inconsistent contributions [22]. The larger ϵ becomes the more it filters out less salient explanation factors, leading to 'explanations that are sparser in terms of input features and less noisy' [22].

The rule **LRP-$\boldsymbol{\gamma}$** [22] introduces a parameter γ to control the favouring of positive contributions, and **LRP-$\boldsymbol{\alpha\beta}$** (originally proposed in [23]) involves two parameters, α and β, that individually deal with positive and negative contributions.

Implementing LPR efficiently requires using different rules on different types of layers (e.g., convolutional, pooling, and fully-connected layers), and some adaptions to the basis LRP rules to guarantee successful relevance propagation.

Finally, following these rules, once the backward propagation reaches the input layer the process ends and each input feature is given a relevance score. These scores indicate how much that input – possibly a specific pixel or feature (collection of pixels) in an image – contributed to the final output, such as a prediction or classification. The higher the score, the more this indicates that a feature played an important role in the network's decision.

Consider a set of latent (or hidden) variables $\mathbf{z}_{1:m}$, and observations $\mathbf{x}_{1:n}$, with joint probability density $p(\mathbf{z}, \mathbf{x}) = p(\mathbf{z})p(\mathbf{x} \mid \mathbf{z})$, omitting constants such as hyperparameters in the notation.

In Bayesian modelling, latent variables play a key role in determining distributions. Initially drawn from a *prior density*, $p(\mathbf{z})$, they are linked to observations via the *likelihood*, $p(\mathbf{x} \mid \mathbf{z})$. Here, making inferences involves conditioning on data and then calculating the *posterior* $p(\mathbf{z} \mid \mathbf{x})$ [24]. Often, we wish to sample from $p(\mathbf{z} \mid \mathbf{x})$. Using Bayes' theorem, we can write the conditional density as $p(\mathbf{z} \mid \mathbf{x}) = p(\mathbf{z}, \mathbf{x})/p(\mathbf{x}) = p(\mathbf{z})p(\mathbf{x} \mid \mathbf{z})/p(\mathbf{x})$, where $p(\mathbf{x}) = \int p(\mathbf{x} \mid \mathbf{z})\, p(\mathbf{z})\, \mathrm{d}\mathbf{z}$ is the marginal density (or *evidence*) of the observations. In many models, this integral cannot be expressed in a closed form. Such integrals can be very slow to compute due to their often high dimensionality and complexity. Therefore, Bayesian inference using sampling methods like Markov chain Monte Carlo (MCMC) methods can be very slow.

Nevertheless, we need the evidence to calculate our desired posterior, which is a practical barrier; however, by reframing things as an optimisation problem using *variational inference* (VI), we can approximate such distributions in a more tractable fashion. The aim is to find, within a family of simple (often Gaussian) distributions, $\mathcal{Q}$, a distribution, $q^\star(\mathbf{z})$, that is as close to $p(\mathbf{z} \mid \mathbf{x})$ as possible. The right choice of $\mathcal{Q}$ is critical. It needs to be tractable, but not too simple that each $q \in \mathcal{Q}$ misses capturing the nuances of the true posterior – this can be a challenge in practice, and it naturally introduces bias. Now, how can we determine the 'closeness' of two distributions?

The *Kullback-Leibler (KL) divergence* (see Definition 8.1) provides a measure for how far two distributions are apart – a concept rooted in information theory [25]. Crucially, we can use optimisation to minimise the KL divergence between approximate and true distributions. Namely, where $\mathrm{D_{KL}}$ is the KL divergence, $q^\star(\mathbf{z}) = \underset{q(\mathbf{z}) \in \mathcal{Q}}{\operatorname{argmin}}\, \mathrm{D_{KL}}\left(q(\mathbf{z}) \,||\, p(\mathbf{z} \mid \mathbf{x})\right)$.

Definition 8.1 Suppose we have two distribution functions, P and Q, of a random variable, X, defined on the same sample space, $\mathcal{X}$. The KL divergence from Q to P is defined [26] as

$$\mathrm{D_{KL}}(P \,||\, Q) = \underset{X \sim P}{\mathbb{E}}\left[\log \frac{P(X)}{Q(X)}\right] = \begin{cases} \sum_{x \in \mathcal{X}} p(x) \log \frac{p(x)}{q(x)}, & \text{X discrete, p and q pmfs;} \\ \int_{\mathcal{X}} p(x) \log \frac{p(x)}{q(x)} \mathrm{d}x, & \text{X continuous, p and q pdfs.} \end{cases} \tag{10}$$

By convention, $0 \cdot \log(0) = 0$, and $\mathrm{D_{KL}}(P \,||\, Q) = \infty$ if $\exists x \in X : q(x) = 0,\ p(x) > 0$. The KL divergence is not symmetric. Namely, $\mathrm{D_{KL}}(P \,||\, Q) \neq \mathrm{D_{KL}}(Q \,||\, P)$. We refer to these as the forward KL and reverse KL, respectively. Furthermore, $\mathrm{D_{KL}}$ does not satisfy the triangle inequality, so it is not a true distance metric. Also, $\mathrm{D_{KL}}(P \,||\, Q) \geq 0$, with equality if and only if $P(x) = Q(x)$ for all x [27]. The more the distributions differ, the higher the KL divergence. For our problem, we have

$$\mathrm{D_{KL}}\left(q(\mathbf{z}) \,||\, p(\mathbf{z} \mid \mathbf{x})\right) = \underset{q(\mathbf{z})}{\mathbb{E}}\left[\log(q(\mathbf{z})\right] - \underset{q(\mathbf{z})}{\mathbb{E}}\left[\log(p(\mathbf{z} \mid \mathbf{x})\right] = \underset{q(\mathbf{z})}{\mathbb{E}}\left[\log(q(\mathbf{z})\right] - \underset{q(\mathbf{z})}{\mathbb{E}}\left[\log(p(\mathbf{z}, \mathbf{x})\right] + \log p(\mathbf{x})$$

We can see that $\log p(\mathbf{x})$ is constant with respect to $q(\mathbf{z})$, and $\mathrm{D_{KL}}(q(\mathbf{z}) \,||\, p(\mathbf{z} \mid \mathbf{x}))$ is dependent on $\log p(\mathbf{x})$, which we recall is hard to compute – a practical barrier. Alternatively, we can optimise the *evidence lower bound* (ELBO), which is equivalent to $\mathrm{D_{KL}}$ up to an additive constant. It is defined as $\mathrm{ELBO}(q) = \mathbb{E}_{q(\mathbf{z})}\left[\log p(\mathbf{z}, \mathbf{x})\right] - \mathbb{E}_{q(\mathbf{z})}\left[\log q(\mathbf{z})\right] = \log p(\mathbf{x}) - \mathrm{D_{KL}}(q(\mathbf{z}) \,||\, p(\mathbf{z} \mid \mathbf{x}))$. Since $\log p(\mathbf{x})$ is constant with respect to $q(\mathbf{z})$, maximising $\mathrm{ELBO}(q)$ is the same as minimising $\mathrm{D_{KL}}$. This gives an approximation of $q(\mathbf{z})$ for the latent variables, $\mathbf{z}$, which is as similar as possible to the true posterior under the KL divergence measure. Since the KL divergence is non-negative, we have that $\log p(\mathbf{x}) \geq \mathrm{ELBO}(q)$. That is, it gives a lower bound for the (log) evidence. In Project 2, we saw that minimising the *cross-entropy* is equivalent to minimising the KL divergence.

While VI it is often more efficient than MCMC methods, it can still be computationally intensive for large datasets. Other challenges include trouble determining convergence and difficulty tuning sensitive hyperparameters (which affect model performance).

Quicksort is an efficient general-purpose sorting algorithm in the field of computer science. It was first developed by Tony Hoare in 1959 and published in 1961 [28]. The algorithmic complexity of Quicksort varies depending on its application, particularly between its worse-case performance of $O(n^2)$ and average-case performance of $O(n \log n)$. This fast performance makes it an appealing choice for many datasets.

Quicksort is a 'divide-and-conquer algorithm' for sorting arrays and works in the following way [29]: Firstly, if the input array has fewer than two elements, there is nothing to sort. Otherwise, choose a particular key (or pivot) and divide the array into two subarrays. Secondly, sort the subarrays recursively.

It is crucial to make the right choice when choosing a pivot since making a wrong choice may lead to the worst-case quadratic time complexity [30].

Lemma 9.1 The worst-case time complexity of Quicksort is $\Omega(n^2)$.

Proof. (from [30])

The partitioning step has at least $n-1$ comparisons. At each step for $n \geq 1$, the number of comparisons is one less, so $T(n) = T(n-1) + (n-1), where T(1) = 0$, and the $T(n)$ factor is the time required to sort a list of size n. Now, we perform 'telescoping.' Note that $T(n) - T(n-1) = n-1$, hence

$$\begin{aligned} &T(n) + T(n-1) + T(n-2) + ... + T(3) + T(2) \\ &\quad - T(n-1) - T(n-2) - ... - T(3) - T(2) - T(1) \\ &\quad = (n-1) + (n-2) + ... + 2 + 1 + 0 = n(n-1)/2 = T(n) \end{aligned}$$

Therefore $T(n) \in \Omega(n^2)$. □

Lemma 9.2 The average-case time complexity of Quicksort is $\Theta(n \log n)$.

Proof. (again from [30])

For a given pivot position i, $i \in 0, 1, 2, ..., n-1$, the time for partitioning an array in cn. Here, c represents the constant time required to process a single element. The head and tail subarrays, respectively, contain i and $n-1-i$ items. Hence $T(n) = cn + T(i) + T(n-1-i)$. The Average runtime for sorting (a more complex recurrence) is

$$\begin{aligned} T(n) &= \tfrac{1}{n} \sum_{i=0}^{n-1} (T(i) + T(n-1-i) + cn) \\ &= \tfrac{2}{n} n(T(0) + T(1) + ... + T(n-2) + T(n-1)) + cn \\ nT(n) &= (T(0) + T(1) + ... + T(n-2) + T(n-1)) + cn^2 \\ (n-1)T(n-1) &= 2(T(0) + T(1) + ... + T(n-2)) + c(n-1)^2 \end{aligned}$$

Hence, $nT(n) - (n-1)T(n-1) = 2T(n-1) + 2cn - c \approx 2T(n-1) + 2cn$. Thus $nT(n) \approx (n+1)T(n-1) + 2cn$, or $\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2c}{n+1}$. Putting $\frac{T(n)}{n+1} - \frac{T(n-1)}{n} = \frac{2c}{n+1}$, we can performing 'telescoping' like before to yield

$$\begin{aligned} &\tfrac{T(n)}{n+1} + \tfrac{T(n-1)}{n} + \tfrac{T(n-2)}{n-1} + ... + \tfrac{T(2)}{3} + \tfrac{T(1)}{2} - \left(\tfrac{T(n-1)}{n} + \tfrac{T(n-2)}{n-1} + ... + \tfrac{T(2)}{3} \tfrac{T(1)}{2} \right) \\ &\quad = \tfrac{2c}{n+1} + \tfrac{2c}{n} + \tfrac{2c}{n-1} + ... + \tfrac{2c}{3} + \tfrac{2c}{2} \end{aligned}$$

Hence $\frac{T(n)}{n+1} = \frac{T(0)}{1} + 2c\left(\frac{1}{2} + \frac{1}{3} + ... + \frac{1}{n} + \frac{1}{n+1}\right) \approx 2c(H_{n+1} - 1) \approx c' \log n$, where H_n is the n–th harmonic number [30]. Therefore, $T(n) \approx c'(n+1)\log n \in \Theta(n \log n)$. □

The paper 'Understanding the Efficiency of GPU Algorithms for Matrix-Matrix Multiplication' [31] provides insights into the differences between matrix multiplication when implemented on a central processing unit (CPU) versus a massively parallel graphics processing unit (GPU). We shall explore these contrasts and then discuss what we can glean from this comparison.

Matrix multiplication is an essential operation in computing, and it is used in many sophisticated models in machine learning, such as in convolutional neural networks, which we implemented in Project 2. Matrix-matrix multiplication is often performed as a sequential process on CPUs. This is fine for simple models; however, with larger datasets and more sophistical models (e.g. many-layered deep neural networks), this can be limiting. Conversely, the GPU architecture is ideal for parallel computation, and matrix multiplication is a highly parallelisable task. Their high memory bandwidth and superior ability to perform floating-point arithmetic compared to conventional CPUs make them naturally appealing for highly parallel tasks. GPUs can excel at handling large-scale matrix operations simultaneously, significantly reducing training time in deep learning models.

Nevertheless, the paper found that, the parallel processing capabilities of GPUs notwithstanding, 'near-optimal GPU implementations' for matrix-matrix multiplication were markedly less efficient than various 'cache-aware CPU approaches.' They found this despite the 'regular access pattern' and 'highly parallel computational requirements' of matrix-matrix multiplication. The inefficiency of GPU performance can be attributed to its being less well-suited at fetching data and its executing more arithmetic operations per clock cycle than a CPU. Moreover, a 'lack of high-bandwidth access to cached data' reduces GPU performance in tasks requiring significant input reuse, resulting in a marked performance disparity to CPU implementations. Here, 'input reuse' in matrix multiplication refers to the frequency with which each element of an input matrix is used in an operation. In the paper, it is noted that in dense matrix-matrix multiplication, each element of the input matrices is reused $O(n)$ times [31]. This factor is notable since, although GPUs can handle parallel computations efficiently, GPUs are less effective than CPUs at tasks that require this frequent input reuse of data. GPUs have less bandwidth for accessing this type of data compared to CPUs. In the paper, it was discovered that 'available floating point bandwidth from the closest cache on a GPU is up to several times lower than that of current CPUs to their L1 caches.'

Although CPUs have fewer cores than GPUs, their individual cores have higher processing capabilities than each individual GPU core. Their cache hierarchy can be exploited, and increased memory bandwidth allows for efficient data reuse, which is vital for operations such as matrix-matrix multiplication. Conversely, GPUs have less sophisticated cache systems and do not manage memory as efficiently; hence, they can perform worse at matrix-matrix multiplication.

When multiplying matrices $\mathbf{A}$ and $\mathbf{B}$, blocking strategies separate matrix $\mathbf{A}$ into smaller submatrices (or blocks) that can fit in processor caches, decomposing into smaller computations of these blocks. Moreover, matrix $\mathbf{B}$ can be transposed – a more 'cache–friendly representation' when tracing columns. The ATLAS [32] software package was used in the paper to test the system cache size of the host CPU and tune its algorithms to achieve 'higher effective bandwidth and hence numerical efficiency' and employ optimised cache-aware matrix-matrix multiplication. ATLAS outperformed all but one GPU implementation and was much more efficient overall. In Project 2, we compared using a GPU to an unoptimised CPU for matrix-matrix multiplication and saw that the GPU was much faster. In the future, we might like to re-perform the experiments with a cache-aware CPU.

As a key take-home message, it is crucial to understand these concepts when choosing whether to use CPU or GPU in applying machine learning models. This decision depends heavily on the specifics of the tasks, namely the data access pattern, what requirements are needed for parallelism, and the hardware architecture available.

Machine learning algorithms like neural networks are typically trained on enormous representative datasets. Such datasets are often crowd-sourced and contain sensitive private information. It should be the aim of a model to avoid revealing confidential information from datasets. To this end, an algorithm has been developed [33] for training as and a refined analysis of the privacy costs and implications using the concept of differential privacy.

Differential privacy [33] is a robust standard for privacy guarantees that ensures privacy in algorithms used on aggregate databases. It is defined by the notion of adjacent databases. For example, experiments have been performed [33] so that each training set comprises pairs of images and labels. Two such sets are considered adjacent if they differ by only one entry.

In Project 2, we explained the stochastic gradient descent (SGD) optimisation algorithm and used it for image classification in a convolutional neural network. Its differentially private counterpart, Algorithm 1, from Deep Learning with Differential Privacy[33] is a variant of SGD; although there are both optimisation algorithms, they have some key differences.

The objective of SGD is to minimise a loss function (such as cross-entropy loss) to increase the accuracy of a model. It does this by tuning parameters (or weights) to minimise loss when iterating through a training set.

Differentially private SGD shares this goal while also introducing measures in its algorithm that ensure that data privacy is not compromised during the training process of models. The algorithm is specifically designed to achieve this goal by adding noise in gradient calculations, which attempts to mask the data to ensure a training set cannot be identified. In contrast, basic SGD does not consider factors of data privacy. Hence, data identification can occur during model learning.

Both algorithms start by initialising random parameters (or weights) θ_0. From a random batch. Then the gradient of the loss function is calculated with respect to the parameters θ, $\nabla\mathcal{L}(\theta)$. Next, a method of 'gradient clipping' [33] is introduced. Its purpose is to bound the influence of individual data points on the gradient calculation and, therefore, the learning process, in order to avoid revealing too much information about that specific observation. Norm-clipping, using the ℓ_2 norm, replaces a gradient vector $\mathbf{g}$ with $\mathbf{g}/\max\left(1, \frac{\|g\|_2}{C}\right)$, where C is the clipping threshold (or the gradient norm bound). In this way, $\|g\|_2 \leq C$ ensures that $\mathbf{g}$ is preserved. If $\|g\|_2 > C$, it's scaled down. It is worth noting that gradient clipping is used in deep neural networks for reasons other than ensuring privacy.

Noise is then added to further ensure the gradient is not too revealing of the features of individual data points. Naturally, the introduction of noise can lead to a reduction in the accuracy of a model. This may necessitate the use of more training data requirements in differentially private SGD than in models that use the traditional SGD algorithm. So, increased computational costs can result from attempts to obscure the identifiability of data – this is a trade-off that must be considered when deciding whether or not to use differentially private SGD. Nevertheless, in sectors such as healthcare and finance (where data is highly sensitive), differentially private SGD is essential.

The data sets used in Project 1 and Project 2 may have privacy concerns. The medical data used in Project 1 pose problems regarding identifiability and privacy. Medical information can reveal intimate and confidential information about an individual's health status and medical history. Furthermore, image data such as that used in Project 2 pose problems of identifiability and consent. Crowd-sourced or open-source datasets are often collected without consent, and their widespread use is a potential ethical problem.

Both SGD and differentially private SGD share the goal of optimisation by minimising the loss function; however, the latter method introduces important mechanisms to guarantee the privacy of individuals whose details are represented in training data. However, these additions can negatively impact a model's performance.

References

[1] P. K. Bhattacharya and P. Burman, “Probability theory,” in *Theory and Methods of Statistics*, pp. 1–24, Elsevier, 2016.

[2] T. Mitchell and A. W. Moore, “Machine learning, 10-701 and 15-781, 2005s, school of computer science, carnegie mellon university.” http://www.cs.cmu.edu/~awm/10701/.

[3] N. Ziemert and P. R. Jensen, *Phylogenetic Approaches to Natural Product Structure Prediction*, p. 161–182. Elsevier, 2012.

[4] Y. Loewenstein, E. Portugaly, M. Fromer, and M. Linial, “Efficient algorithms for accurate hierarchical clustering of huge datasets: tackling the entire protein space,” *Bioinformatics*, vol. 24, p. i41–i49, July 2008.

[5] S. F. Altschul, T. L. Madden, A. A. Schäffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman, “Gapped BLAST and PSI-BLAST: a new generation of protein database search programs,” *Nucleic Acids Research*, vol. 25, pp. 3389–3402, 09 1997.

[6] R. Sokal, C. Michener, and U. of Kansas, *A Statistical Method for Evaluating Systematic Relationships*. University of Kansas science bulletin, University of Kansas, 1958.

[7] R. C. Edgar, “Muscle: multiple sequence alignment with high accuracy and high throughput,” *Nucleic acids research*, vol. 32, no. 5, pp. 1792–1797, 2004.

[8] T. Hofmann, B. Schölkopf, and A. J. Smola, “Kernel methods in machine learning,” *The Annals of Statistics*, vol. 36, June 2008.

[9] A. Shashua, “Introduction to machine learning: Class notes 67577,” 2009.

[10] X. B. Sicotte, “Kernels and feature maps: Theory and intuition.” https://xavierbourretsicotte.github.io/Kernel_feature_map.html.

[11] L. Breiman, “Bagging predictors,” *Machine Learning*, vol. 24, p. 123–140, Aug. 1996.

[12] P. Bühlmann and B. Yu, “Analyzing bagging,” *The Annals of Statistics*, vol. 30, Aug. 2002.

[13] J. Han and Q. Liu, “Bootstrap model aggregation for distributed statistical learning,” 2016.

[14] H.-J. Yoon, H. B. Klasky, J. P. Gounley, M. Alawad, S. Gao, E. B. Durbin, X.-C. Wu, A. Stroup, J. Doherty, L. Coyle, L. Penberthy, J. Blair Christian, and G. D. Tourassi, “Accelerated training of bootstrap aggregation-based deep information extraction systems from cancer pathology reports,” *Journal of Biomedical Informatics*, vol. 110, p. 103564, Oct. 2020.

[15] V. Kurama, “Introduction to bagging and ensemble methods.” https://blog.paperspace.com/bagging-ensemble-methods/.

[16] D. Wickramarachchi, B. Robertson, M. Reale, C. Price, and J. Brown, “Hhcart: An oblique decision tree,” *Computational Statistics amp; Data Analysis*, vol. 96, p. 12–23, Apr. 2016.

[17] J. R. Fair and H. Z. Kister, “Absorption (chemical engineering),” in *Encyclopedia of Physical Science and Technology*, pp. 1–25, Elsevier, 2003.

[18] G. Montavon, W. Samek, and K.-R. Müller, "Methods for interpreting and understanding deep neural networks," *Digital Signal Processing*, vol. 73, p. 1–15, Feb. 2018.

[19] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2014.

[20] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 2818–2826, 2016.

[21] A. Shrikumar, P. Greenside, A. Shcherbina, and A. Kundaje, "Not just a black box: Learning important features through propagating activation differences," 2016.

[22] G. Montavon, A. Binder, S. Lapuschkin, W. Samek, and K.-R. Müller, *Layer-Wise Relevance Propagation: An Overview*, p. 193–209. Springer International Publishing, 2019.

[23] S. Bach, A. Binder, G. Montavon, F. Klauschen, K.-R. Müller, and W. Samek, "On pixel-wise explanations for non-linear classifier decisions by layer-wise relevance propagation," *PLOS ONE*, vol. 10, p. e0130140, July 2015.

[24] D. M. Blei, A. Kucukelbir, and J. D. McAuliffe, "Variational inference: A review for statisticians," *Journal of the American Statistical Association*, vol. 112, p. 859–877, Apr. 2017.

[25] "Introduction to data warehousing and data mining." 2.4.8 Kullback-Leibler Divergence, University of Illinois, CS412 Fall 2008.

[26] D. J. C. MacKay, *Information theory, inference and learning algorithms.* Cambridge, England: Cambridge University Press, Sept. 2003.

[27] Thomas M. Cover and J. A. Thomas, *Elements of Information Theory.* Nashville, TN: John Wiley & Sons, 2 ed., June 2006.

[28] C. A. R. Hoare, "Algorithm 64: Quicksort," *Communications of the ACM*, vol. 4, p. 321, July 1961.

[29] "Algorithms and data structures: Average-case analysis of quicksort, university of edinburgh." https://www.inf.ed.ac.uk/teaching/courses/ads/Lects/lecture8.pdf.

[30] G. Gimel'farb, "Algorithm quicksort: Analysis of complexity, compsci 220 algorithms and data structures, university of auckland." https://www.cs.auckland.ac.nz/courses/compsci220s1c/lectures/2016S1C/CS220-Lecture10.pdf.

[31] K. Fatahalian, J. Sugerman, and P. Hanrahan, "Understanding the efficiency of gpu algorithms for matrix-matrix multiplication," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, GH04, ACM, Aug. 2004.

[32] R. Clint Whaley, A. Petitet, and J. J. Dongarra, "Automated empirical optimizations of software and the atlas project," *Parallel Computing*, vol. 27, p. 3–35, Jan. 2001.

[33] M. Abadi, A. Chu, I. Goodfellow, H. B. McMahan, I. Mironov, K. Talwar, and L. Zhang, "Deep learning with differential privacy," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS'16, ACM, Oct. 2016.

Appendix

Appendix 1

```
# Download the dataset from kaggle (link below)
# https://www.kaggle.com/datasets/spacemod/pulsar-dataset/data
# Alternative, you can import the dataset from sklearn in python and save it

pulsar <- read.csv("/Users/thomaspagulatos/Downloads/pulsar_stars.csv", header=TRUE)
```

```
# Rename the columns
names(pulsar)
```

```
## [1] "Mean.of.the.integrated.profile"
## [2] "Standard.deviation.of.the.integrated.profile"
## [3] "Excess.kurtosis.of.the.integrated.profile"
## [4] "Skewness.of.the.integrated.profile"
## [5] "Mean.of.the.DM.SNR.curve"
## [6] "Standard.deviation.of.the.DM.SNR.curve"
## [7] "Excess.kurtosis.of.the.DM.SNR.curve"
## [8] "Skewness.of.the.DM.SNR.curve"
## [9] "target_class"
```

```
names(pulsar) <- c("mean_profile", "std_profile", "kurtosis_profile","skewness_profile","mean_dmsnr_cur
names(pulsar)
```

```
## [1] "mean_profile"         "std_profile"          "kurtosis_profile"
## [4] "skewness_profile"     "mean_dmsnr_curve"     "std_dmsnr_curve"
## [7] "kurtosis_dmsnr_curve" "skewness_dmsnr_curve" "category"
```

```
# Inspect the data
str(pulsar)
```

```
## 'data.frame':    17898 obs. of  9 variables:
##  $ mean_profile        : num  140.6 102.5 103 136.8 88.7 ...
##  $ std_profile         : num  55.7 58.9 39.3 57.2 40.7 ...
##  $ kurtosis_profile    : num  -0.2346 0.4653 0.3233 -0.0684 0.6009 ...
##  $ skewness_profile    : num  -0.7 -0.515 1.051 -0.636 1.123 ...
##  $ mean_dmsnr_curve    : num  3.2 1.68 3.12 3.64 1.18 ...
##  $ std_dmsnr_curve     : num  19.1 14.9 21.7 21 11.5 ...
##  $ kurtosis_dmsnr_curve: num  7.98 10.58 7.74 6.9 14.27 ...
##  $ skewness_dmsnr_curve: num  74.2 127.4 63.2 53.6 252.6 ...
##  $ category            : int  0 0 0 0 0 0 0 0 0 0 ...
```

```r
# Split data into train and test set

trainIndex <- createDataPartition(pulsar$category, p = 0.8,
                                  list = FALSE,
                                  times = 1)
# Sub-setting into training data
train <- pulsar[ trainIndex,]

# Sub-setting into testing data
test <- pulsar[-trainIndex,]

# Let us inspect these new dataframes using frequency tables
as.data.frame(table(train$category))
```

```
##   Var1  Freq
## 1    0 12993
## 2    1  1326
```

```r
as.data.frame(table(test$category))
```

```
##   Var1 Freq
## 1    0 3266
## 2    1  313
```

```r
train1 <- subset(train, select = -c(category))
test1 <- subset(test, select = -c(category))

# Mean-centre and normalise the 8 features, doing nothing to the target
# category, of course
preProcValues <- preProcess(train1, method = c("center", "scale"))

# Now, we transform the test data set using the same transformation parameters
# that we used on the training set -- this is important.
train.transformed <- predict(preProcValues, train1)
test.transformed <- predict(preProcValues, test1)

# Recombine into full scaled datasets
train.scaled <- cbind(subset(train, select = c(category)), train.transformed)
test.scaled <- cbind(subset(test, select = c(category)), test.transformed)
```

```r
#train.scaled
#test.scaled

x.train <- subset(train.scaled, select = -category)
y.train <- subset(train.scaled, select = category)
x.test <- subset(test.scaled, select = -category)
y.test <- subset(test.scaled, select = category)
```

```r
# Perform PCA on x.train, since we do not target category
pca <- prcomp(x.train)
pca$rotation
```

```
##                             PC1        PC2         PC3         PC4          PC5
## mean_profile          0.3597029 -0.3621000  0.01069667 -0.29250163 -0.742962016
## std_profile           0.2124969 -0.4293729 -0.43173985  0.76024146 -0.019947980
## kurtosis_profile     -0.4161905  0.3190080 -0.09309957  0.30309913 -0.143900885
## skewness_profile     -0.4013112  0.3064873 -0.08584431  0.17510531 -0.645265745
## mean_dmsnr_curve     -0.3440155 -0.2505756 -0.57115589 -0.33540987  0.077821153
## std_dmsnr_curve      -0.3867257 -0.3181481 -0.23587449 -0.24866933  0.064332286
## kurtosis_dmsnr_curve  0.3705750  0.4101683 -0.27562449 -0.03834983 -0.002820744
## skewness_dmsnr_curve  0.2931969  0.3956988 -0.58277339 -0.19528912  0.018205586
##                              PC6         PC7          PC8
## mean_profile          0.01906882  0.31704252  0.030882164
## std_profile           0.03855270 -0.06495525  0.003762874
## kurtosis_profile      0.04706614  0.77229905  0.071514142
## skewness_profile     -0.02448828 -0.53729073 -0.036535084
## mean_dmsnr_curve     -0.60463370  0.02399938  0.089063852
## std_dmsnr_curve       0.75328748 -0.06368735  0.236886101
## kurtosis_dmsnr_curve  0.01332467 -0.06173575  0.782943019
## skewness_dmsnr_curve  0.24926630  0.04116254 -0.561725807
```

```
# The main source of inspiration for this code was found here:
# http://www.sthda.com/english/articles/31-principal-component-methods-in-r-practical-guide/112-pca-pri

# calculate total variance explained by each principal component
var.explained = pca$sdev^2 / sum(pca$sdev^2)

# Let us inspect the variance explained
# This should be monotonically decreasing -- and it is
var.explained
```

```
## [1] 0.515263090 0.268618198 0.101820346 0.057404691 0.032467559 0.019877081
## [7] 0.002565172 0.001983863
```

```
#install.packages(c("FactoMineR", "factoextra"))
library("FactoMineR")
library("factoextra")
```

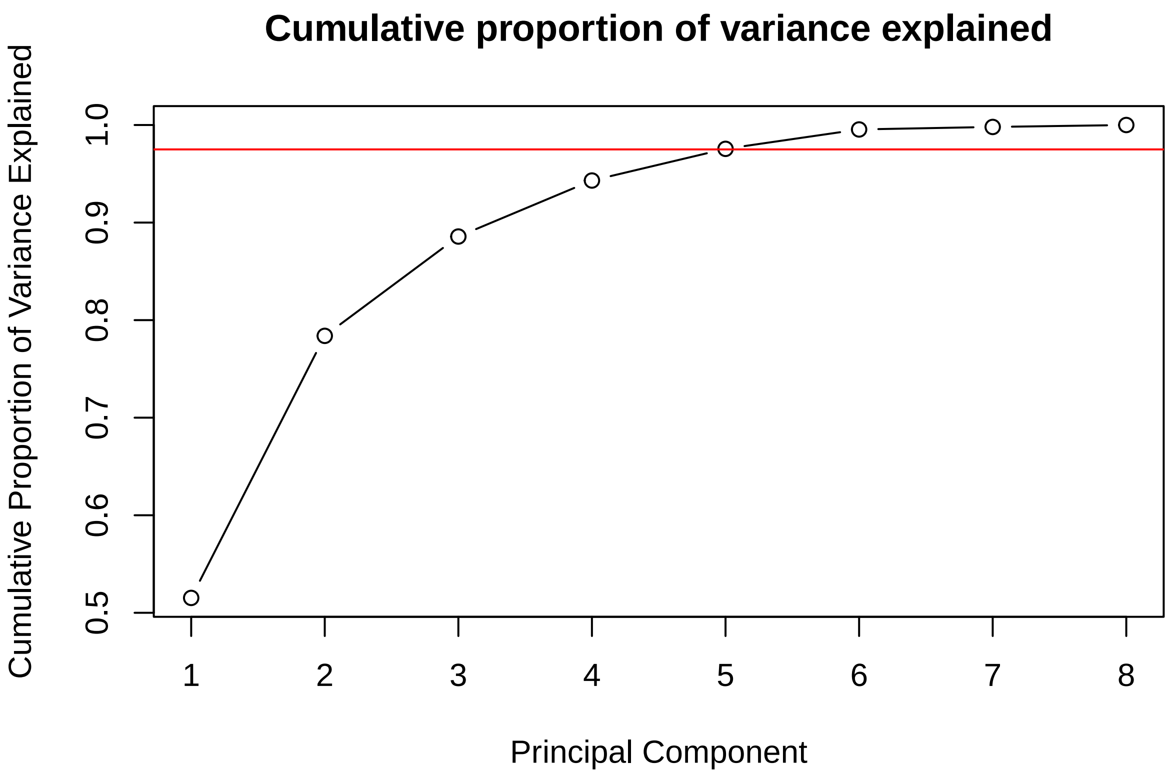
```
## Welcome! Want to learn more? See two factoextra-related books at https://goo.gl/ve3WBa
```

```
# If we look at the cumulative proportion, we can see that 97.5% of the total
# variance is explained by the first 5 principal components.
summary(pca)
```

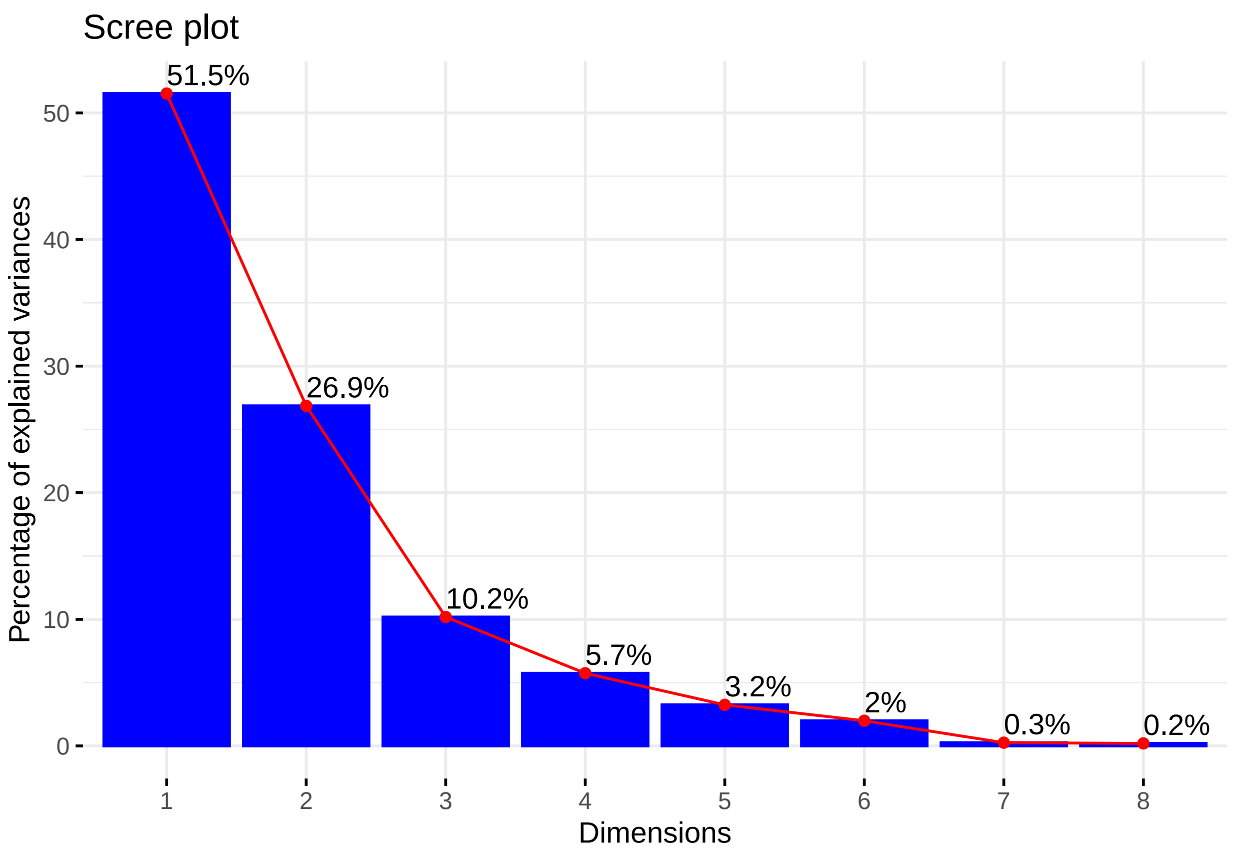
```
## Importance of components:
##                           PC1    PC2    PC3    PC4     PC5     PC6     PC7
## Standard deviation     2.0303 1.4659 0.9025 0.6777 0.50965 0.39877 0.14325
## Proportion of Variance 0.5153 0.2686 0.1018 0.0574 0.03247 0.01988 0.00257
## Cumulative Proportion  0.5153 0.7839 0.8857 0.9431 0.97557 0.99545 0.99802
##                            PC8
## Standard deviation     0.12598
## Proportion of Variance 0.00198
## Cumulative Proportion  1.00000
```

```r
# Let us plot the cumulative proportion of variance explained
cumulative.plot.data <- cumsum(var.explained)

plot(cumulative.plot.data, xlab = "Principal Component",
     ylab = "Cumulative Proportion of Variance Explained",
     type = "b",
     main = "Cumulative proportion of variance explained")
abline(h = 0.975, col="red", v=10)
```



```r
# Screeplot
fviz_screeplot(pca, addlabels = TRUE, barfill = "blue", barcolor = "blue",
               linecolor = "red")
```



```
# This shows that the first 5 PCs capture enough of the variance in the data
# that we can reduce the number of dimensions by 3.
```

```
training.data <- cbind(subset(train.scaled, select = c(category)), pca$x)

# We use the first 5 PCs
training.data <- training.data[,1:6]

# Perform the same transformation to test data as was done of train data,
# including mean centring and scaling.
# Again, we select the first 5 PCs
testing.data <- as.data.frame(predict(pca, newdata = x.test))[,1:5]

# Convert int to factor for use in decsion tree
training.data$category = as.factor(training.data$category)
y.test$category = as.factor(y.test$category)
```

```
# Train tree classifier on pca-transformed data

rpart.model = train(category ~ .,
                  data=training.data,
                  method="rpart",
                  trControl = trainControl(method = "cv"))

# Make predictions
```

```r
predictions <- predict(rpart.model, testing.data)

cm <- confusionMatrix(predictions, y.test$category)
cm
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction    0    1
##          0 3254  102
##          1   12  211
##
##                Accuracy : 0.9681
##                  95% CI : (0.9619, 0.9737)
##     No Information Rate : 0.9125
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.7706
##
##  Mcnemar's Test P-Value : < 2.2e-16
##
##             Sensitivity : 0.9963
##             Specificity : 0.6741
##          Pos Pred Value : 0.9696
##          Neg Pred Value : 0.9462
##              Prevalence : 0.9125
##          Detection Rate : 0.9092
##    Detection Prevalence : 0.9377
##       Balanced Accuracy : 0.8352
##
##        'Positive' Class : 0
##
```

```r
# Convert int to factor for use in decsion tree
train.scaled$category = as.factor(train.scaled$category)

# Train tree classifier just on scaled data
rpart.model1 = train(category ~ .,
                  data=train.scaled,
                  method="rpart",
                  metric="Accuracy",
                  trControl = trainControl(method = "cv"))

# Make predictions

predictions1 <- predict(rpart.model1, x.test)

cm1 <- confusionMatrix(predictions1, y.test$category)
cm1
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction    0    1
##          0 3240   47
##          1   26  266
##
##                Accuracy : 0.9796
##                  95% CI : (0.9744, 0.984)
##     No Information Rate : 0.9125
##     P-Value [Acc > NIR] : < 2e-16
##
##                   Kappa : 0.8682
##
##  Mcnemar's Test P-Value : 0.01924
##
##             Sensitivity : 0.9920
##             Specificity : 0.8498
##          Pos Pred Value : 0.9857
##          Neg Pred Value : 0.9110
##              Prevalence : 0.9125
##          Detection Rate : 0.9053
##    Detection Prevalence : 0.9184
##       Balanced Accuracy : 0.9209
##
##        'Positive' Class : 0
##
```