Contents

Chapter 1

Background Information

1.1 Definitions

Monotonic Function: A function whose range is entirely non-decreasing or non-increasing over its domain. It has no global or local minima or maxima anywhere in the range except for at the start and end of the domain. A monotonic function's first derivative will not change sign over the domain.

1.2 Equations

1.2.1 Mean (Expectation)

$$E[X] = \mu$$

For a discrete data set X:

$$E[X] = \frac{1}{n}\sum_{i=1}^{n} x_i$$

1.2.2 Variance

Variance is defined as the expected value of the squared deviation from the mean and is a measure of the spread of the data about the mean.

$$Var(X) = \sigma^2$$
$$Var(X) = cov(X, X)$$
$$Var(X) = E[(X - E[X])^2] = E[X^2] - E[X]^2$$

1.2.3 Covariance

Covariance is a generalization of variance between any two random variables.

$$cov(X,Y) = E[(X - E[X])(Y - E[Y])] = E[XY] - E[X]E[Y]$$

1.2.4 Correlation

Correlation (in this case, Pearson's product-moment correlation) is just a normalized covariance measure and measures how related two variables are.

$$corr(X,Y) = \rho_{X,Y}$$
$$corr(X,Y) = \frac{cov(X,Y)}{\sigma_X \sigma_Y}$$

1.2.5 Bayes' Theorem

Bayes' theorem is a conditional distribution theorem that states, for $A = (a_1, ..., a_p)$ and $B = (b_1, ..., b_q)$:

$$P(A|B) = \frac{P(A)P(B|A)}{P(B)}$$

Where $P(A|B)$ is our **posterior**, $\frac{P(B|A)}{P(B)}$ is our **evidence** or the 'support B provides for A' and $P(A)$ is our **prior**.

Chapter 2

Definitions

2.1 General

Input Variables: Input variables are the set of data points that are fed in to the model and are generally referred to as X. Each individual data point is usually indexed as x_i for the i^{th} data point, each of which can be a vector indexed with j, x_{ij}, for the j^{th} element of the i^{th} data point. Input variables can also be referred to as the **predictors**, **independent variables** or the **features**.

Output Variables: Output variables are the result or outcome of a given event and are (we hope!) related to a corresponding set of input variables. Output variables are referred to as Y and each individual point can be indexed as y_i for the i^{th} data point. As with input variables, each data point could also be a vector indexed with j, y_{ij}. Input and output variables are coupled as the data point x_i is used to predict the output y_i and are sometimes represented as $((x_1, y_1), (x_2, y_2), ..., (x_n, y_n))$. Output variables are also known as the **dependent variable** or the **response**.

Statistical Learning: Statistical learning (and, in general, machine learning) aims to take (x_i, y_i) pairs and find the relationship between them, expressed formally as:

$$Y = f(X) + \epsilon$$

Where ϵ is the irreducible error and f represents the **function**, **relationship** or **mapping** between X and Y. Given this, our aim is to learn f. There are two types of error present here, the **irreducible error** and the **reducible error**.

Irreducible Error: Irreducible error, ϵ, represents error we cannot account for (and therefore not reduce) such as our data not being representative of the true relationship or inherent noise in the relationship. In theory you could reduce this error, in certain cases, but you would have to go outside the scope of the problem (collect more data, understand more about the underlying relationship) which is not always feasible.

Reducible Error: Reducible error is found within f and is the difference between the true f and our predicted f, $\hat{f}$. It can arise from $\hat{f}$ being the wrong form (a linear $\hat{f}$ approximating

a non-linear f), being too flexible or not flexible enough, errors in the learning algorithm (rounding errors, approximations for computational tractability) and a number of other ways. Importantly, reducible error is named as so because we can take steps to reduce this error, relying on our domain knowledge or experience in selecting the proper approach.

Supervised vs Unsupervised Learning: In supervised learning each x_i has a corresponding y_i and thus there is a complete set of responses for all predictors. We therefore aim to learn the relationship between the two. Unsupervised learning is different in that we do not have corresponding y_i for each x_i. We lack the data to learn $\hat{f}$ and either y_i do not exist or are otherwise unobtainable so there is no approach to tackle $\hat{f}$. In such a case we instead focus on learning relationships *within* the set of predictors such as relative differences between each data point. Semi-supervised learning also exists where we have a set of y_i for $i < n$ where n is our number of data points. In such a case, we can apply supervised learning for the set of predictors that have responses and incorporate the remaining predictors with another method.

Flexibility vs Interpretability: The flexibility of a model is proportional to how many degrees of freedom it contains (how many learnable parameters it contains) and flexible models tend to learn complex relationships better. Interpretable models are models that allow us to understand the learned relationships in the data, for example learning the values of the parameters in linear regression lets us understand the influence of each predictor on the response. These two concepts are usually opposed in that by increasing one we reduce the other. Approaches that can accurately model complex behaviour (flexible) tend to be very hard to interpret for a human (multi layer neural nets) and vice versa.

Prediction vs Inference: When undertaking a statistical learning task you must decide what the goal is, be that prediction or inference. Prediction focuses on the product of the statistical learning, namely the learned Y values, and does not care for how they were obtained or the particular form of $\hat{f}$. Due to this, prediction places little emphasis on interpretability and prefers as accurate as model as possible. Inference on the other hand does not care so much about what the predicted Y values are and focuses more on the relationship between X and Y, $\hat{f}$, and how each predictor influences the outcome. It therefore concerns itself a great deal more with interpretability, potentially at the expense of (some) flexibility. Prediction would be using the weather today to predict whether it will rain tomorrow, inference would be working out what aspects of the weather today, and with what importance, will influence whether it rains tomorrow.

Parametric vs Non-Parametric Approaches: When approaching a statistical learning problem you must decide how you will learn $\hat{f}$. A parametric approach seeks to simplify the problem to learning a set of parameters that define the behaviour of $\hat{f}$ by selecting a form for $\hat{f}$ manually and parametrising it. A common example would be a linear equation for $\hat{f}$:

$$\hat{f}(x_i) = Ax_{i1} + Bx_{i2} + ... + Cx_{in} + D$$

Where A, B, C and D are the parameters. A non-parametric approach would instead try to learn the form of $\hat{f}$ directly. Parametric approaches tend to be more interpretable and easier to train, requiring less data and less computation. Conversely, non-parametric approaches require considerably more data and computational power and are usually less interpretable but offer gains in accuracy, are more flexible (can better model highly complex relationships) and reduce the need to know what the correct form for $\hat{f}$ is beforehand.

Over/Under Fitting: Overfitting refers to a $\hat{f}$ which models the training data too accurately, to the point where it is modelling random noise in the dataset as if it were part of the true relationship in the data. The knock on effect of this is that when you move to a test (or production) data set where that random noise does not exist, you are predicting outcomes based on false relationships. Overfitting can come from an overly flexible model (too many degrees of freedom for the true relationship so some get filled by noise) or a model trained for too long. Underfitting refers to the opposite problem where the model does not (or cannot, if the model is too simplistic) learn enough of the true relationship in the data and therefore produces incorrect predictions.

Regression vs Classification: Regression problems involve response variables that are numeric and continuous whereas classification problems involve response variables that are discrete and fall in to categories or classes. In either case, the form of the predictor variables usually assumes the same form as the response but this is not always the case and is not particularly important either way.

Bias: Bias is a type of error that is introduced when approximating complex real world problems with simpler models. Bias is a **systematic** error inherent in the model and is defined as the difference between the expected sample value and the population (true) value. In the case of population statistics, bias is defined as the difference between the expectation (average) of the sample statistic (mean, variance) and the population statistic. In general, more flexible models reduce the bias in the system as they better fit the underlying relationship. Important to note here, bias is not a measure of how wrong the value is *in practice* but rather whether the model/equation is systematically over/under estimating the true value. For example, with the sample variance, defined as $S^2 = \frac{1}{n}\sum_{i=1}^{n}(X_i - \bar{X})^2$, the expectation value $E[S^2]$ will equal $(1 - \frac{1}{n})\sigma^2$ which is less than the true statistic σ^2 (unless n goes to infinity), which means that S^2 is a biased estimator. This can be fixed by setting $\frac{1}{n} \rightarrow \frac{1}{n-1}$ in the definition of S^2.

Variance: Variance is mathematically understood as a measure of squared deviation from the mean and acts as a metric for 'spread'. Variance can be understood as the amount the model's predictions would change if we estimated it with a different training set - how 'stable' or generalized is our solution or does it depend heavily on each individual data point for its shape. Flexible models tend to introduce variance as the model's expressiveness allows for small changes in the training set to lead to large changes in the learned model.

The Bias-Variance Trade-Off: Flexible models reduce bias but increase variance, restrictive models do the opposite. The bias-variance trade-off is the balancing act between flexibility and accuracy that minimizes both measures. Specifically, the expected test MSE can be decomposed in to three parts: $E[y_0 - \hat{f}(x_0)] = Var(\hat{f}(x_0)) + [Bias(\hat{f}(x_0))]^2 + Var(\epsilon)$. We want to minimize the test MSE and looking at this equation, ϵ cannot be reduced and so we must reduce the variance and (square) bias. Though as we just saw, we struggle to reduce one without increasing the other and so we must make a trade-off.

l_1 **Norm**: The l_1 norm of a vector, v, of length n is defined as:

$$||v||_1 = \sum_{i=1}^{n} |v_i|$$

Where $|x|$ is the absolute value of x. When applied as regularization, the l_1 norm tends to set

small weights equal to zero.

l_2 **Norm**: The l_2 norm of a vector, v, of length n is defined as:

$$||v||_2 = \sqrt{\sum_{i=1}^{n} v_i^2}$$

The l_2 norm measures the distance of v from zero. The l_2 norm is also known as the **Euclidean norm**. When applied as regularization, the l_2 norm tends to penalize large weights and tends all weights towards zero (but usually not equal to).

Kernel Functions: A kernel function is a function which defines an analogous 'inner product' for objects other than vectors. In other words, a kernel function is a generalization of the inner product and thus it quantifies the similarity of two observations. A kernel function is therefore a function of two objects (e.g. vectors) and outputs a similarity measure which can be tailored to the problem. A linear kernel might look something like $K(x_i, x_{i'}) = \sum_{j=1} x_{ij} x_{i'j}$ while the **radial kernel** may look like $K(x_i, x_{i'}) = exp(-\gamma \sum_{j=1} (x_{ij} - x_{i'j})^2)$ where the radial kernel is good at generating circular decision boundaries. The radial kernel is of particular note, it is a commonly used kernel function. It works because observations far away from the test point ($(x_{ij} - x_{i'j})^2$ is large) will produce very small similarity measures (small K) and so it has a **local behaviour** which allows it to 'wrap around' groups of observations.

Chapter 3

Algorithms

3.1 Dimensionality Reduction

Dimensionality reduction works by transforming the predictors X in to a set of new predictors Z where $|Z| \leq |X|$. For example, if we let Z represent linear combinations of X, with $|X| = p$ and $|Z| = M$, $M < p$:

$$Z_m = \sum_{j=1}^{p} \phi_{jm} X_j$$

With ϕ_{jm} being some matrix of constants. These can then be used in a linear regression problem:

$$y_i = \Theta_0 + \sum_{m=1}^{M} \Theta_m z_{im} + \eta_i$$

Where z_{im} is the ith component of Z_m and Θ are our new coefficients analogous to β but for the Z predictors. It can be shown that:

$$\beta_j = \sum_{m=1}^{M} \Theta_m \phi_{jm}$$

Now, we have a regression problem with only M predictors as opposed to the original p predictors ($M < p$) which means we have a less flexible model, need less training data and can train faster. The problem of dimensionality reduction is then in selecting the correct Z_ms (selecting ϕ).

3.1.1 Principal Component Analysis

One way of selecting Z is to use Principal Component Analysis (PCA). PCA works by building the data X from its 'principal' components in descending order (Z_1 would be the most principal, and so on). 'Principal' here means the direction of the data with which observations vary the most. For a simple 2-dimensional linear regression, the first principal component (Z_1) may be the line that best describes the relationship. The second principal component (Z_2) would be perpendicular to this line. In fact, this is a rule: all principal components must be uncorrelated with all other principal components or, in other words, they must be orthogonal.

PCA is an unsupervised approach. When used in regression, we may be selecting coefficients based on a known response Y but for finding the principal components themselves, we only have access to the features X.

The ith principal component of a set of features X of length p is:

$$Z_i = \sum_{j=1}^{p} \phi_{ji} X_j$$

Where ϕ_{ji} is the jth component of the **loadings** of principal component i. These ϕ are normalized: $\sum_{j=1}^{p} \phi_{ji}^2 = 1$ for each i. The **principal component loading vector** for principal component i is thus: $\phi_i = (\phi_{1i}, ..., \phi_{pi})^T$.

Given a dataset of size n with p features (a $n \times p$ matrix $\mathbf{X}$) and normalized to have zero mean and a standard deviation of 1, we can find the first principal component through the optimisation problem:

$$\text{maximize}_{\phi_1} \left\{ \frac{1}{n} \sum_{i=1}^{n} \left(\sum_{j=1}^{p} \phi_{j1} x_{ij} \right)^2 \right\} \text{ subject to } \sum_{j=1}^{p} \phi_{j1}^2 = 1$$

Which is possible since we wish to maximize the variance captured by the first principal component. We can rewrite the objective as $\frac{1}{n} \sum_{i=1}^{n} z_{i1}^2$ and since $\frac{1}{n} \sum_{i=1}^{n} x_{ij} = 0$ (zero mean constraint) the average of all z will be zero as well. Thus, the objective is simply maximizing the sample variance of the n values of z_{i1}. The set of z_i values are called the **scores** of the ith principal component. This optimisation problem can be solved with eigen decomposition.

The loading vectors and scores can be interpreted as the direction with which the data varies the most and the projection of each data point on to these directions, respectively.

The second principal component (and all others after it) can be found in a very similar way. Each time you wish to find the next principal component, a new constraint must be added: the ith principal component ($i > 1$) must be orthogonal to all previous ($i - 1$) principal components: the second principal component must be orthogonal to the first, the third must be orthogonal to the first and second, and so on.

With these principal components we can reduce the dimensionality of our original problem to

an arbitrary dimension M ($M \leq min(n-1, p)$, $M \geq 1$) which could be useful in regression (reducing the number of features helps prevent overfitting) or analysis (being able to visualize high dimensionality data in 2 or 3 dimensions). In fact, if we plot the score vectors Z_i against each other, this amounts to projecting the data down onto the subspace spanned by ϕ_i. One could, for example, project the dataset in to two dimensions by plotting the first and second principal component scores for each data point on a graph and drawing in the loading vector lines for each feature. You could then roughly which features are related (if a subset of the features are explained by the same loading vector more than the other, they may be related), what features each principal component is explaining, where each data point lies in relation and so on.

Principal components can also be interpreted as providing low dimensionality linear surfaces that are 'closest' to the data. The first principal component has the property that it is the line in p-dimensional space that is closest to to the n observations. The first and second principal component define a plane in p-dimensional space that is closest to the n observations, and so on.

How Much Variance Does Each PC Explain

There is a set amount of variance in the dataset and each principal component accounts for some amount of that variance, with $M = p$ accounting for all of it. The total variance present in the data (assuming normalization) is defined as:

$$\sum_{j=1}^{p} \text{Var}(X_j) = \sum_{j=1}^{p} \frac{1}{n} \sum_{i=1}^{n} x_{ij}^2$$

And the variance explained by the mth principal component is:

$$\frac{1}{n} \sum_{i=1}^{n} z_{im}^2 = \frac{1}{n} \sum_{i=1}^{n} \left(\sum_{j=1}^{p} \phi_{jm} x_{ij} \right)^2$$

Using this we can define the **proportion of variance explained**, or **PVE**, of the mth principal component as:

$$\frac{\sum_{i=1}^{n} \left(\sum_{j=1}^{p} \phi_{jm} x_{ij} \right)^2}{\sum_{j=1}^{p} \sum_{i=1}^{n} x_{ij}^2}$$

The PVE is always positive and the sum of all PVE's (over all principal components) is 1. In general, a $n \times p$ data matrix $\mathbf{X}$ has $\min(n-1, p)$ principal components.

Generally, there is no rule as to how many principal components are necessary and there is no automatic way to discover it (such as cross validation). One subjective method is a **scree plot** which is a plot of PVE for each principal component. You would then look for an 'elbow' - the point where the decrease in PVE starts to taper off. If the principal components are to

be used in a supervised model (e.g. for principal component regression) then more objective techniques (cross validation) can be used.

3.2 Linear Regression

When performing regression it is important to note that there are actually two equations defining the relationship, the **least squares line**:

$$\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 x_1$$

Which is our best estimate for the relationship, and the **population regression line**:

$$y = \beta_0 + \beta_1 x_1 + \epsilon$$

Which is the true relationship. The distinction here is that the population regression line includes ϵ which cannot be calculated from data and the parameters are all 'exact' (they are as theoretically good as they can be). We therefore care that $\hat{\beta}_i$ is an unbiased estimator for β_i which would imply that $\hat{y}$ is an unbiased estimator for y.

3.2.1 Least Squares Regression

Linear regression is a parametric algorithm where we construct an equation as so for n predictors:

$$\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 x_1 + ... + \hat{\beta}_n x_n$$

Where our task is to learn all $\hat{\beta}$s.

In order to find the optimal value for these we need to define a measure of error, the **residual sum of squares (RSS)**:

$$RSS = e_1^2 + e_2^2 + ... + e_n^2$$

Where $e_i^2 = (y_i - \hat{y}_i)^2 = (y_i - \hat{\beta}_0 - \hat{\beta}_1 x_i)^2$ and is known as the i^{th} **residual**. Note, the RSS is the mean squared error (MSE) without the normalizing ($\frac{1}{n}$) factor.

We can then derive an optimisation equation for all $\hat{\beta}$s by differentiating the RSS with respect to each individual $\hat{\beta}$ and setting equal to 0. We find that our first $\hat{\beta}$ will be a function of only x and y and subsequent $\hat{\beta}$s will be a function of x, y and previously calculated $\hat{\beta}$s which allows us to sequentially solve for all parameters. For a simple regression case where we only have one predictor, we find:

$$\hat{\beta}_1 = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{n}(x_i - \bar{x})^2}$$

$$\hat{\beta}_0 = \bar{y} - \hat{\beta}_1 \bar{x}$$

Where $\bar{x}$ means the average value of x (the sample means). The RSS and the update equations can be trivially extended to multiple ($n > 1$) predictors if we need to.

The quality of a linear regression fit is usually assessed with the **residual standard error (RSE)** and the R^2 statistic.

Linear regression works natively with quantitative data but can be made to work with qualitative data also with an encoding scheme. For 3 predictors you need 2 'dummy variables' where, for example, you set x_1 = -1 and x_2 = 1, while x_0 is represented as the absence of the other two. From this you could read a learned model as the difference from the baseline (the missing variable): β_1 is the difference between x_1 and x_0 and so on.

Assumptions Of The Linear Model

The linear model makes two assumptions, the **additive** and **linear** assumptions. The additive assumption states that the effect a predictor X_j has on response Y is independent of the values of all other predictors $X_{\neq j}$. The linear assumption states that a one-unit change in X_j leads to the same change in Y ($\pm\beta_j$) regardless of the current value of X_j. If at all possible, we would like to relax these assumptions to make our model more robust.

Relaxing The Additive Assumption: The additive assumption can be relaxed by encoding for (and thus removing) interaction/synergy between predictors, or in other words, dependency. This can be achieved by adding a new parameter that encodes the joint relationship, $\beta_3 x_1 x_2$. We would then have a model such as $Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_1 X_2 + \epsilon$ which can be rewritten as $Y = \beta_0 + \widetilde{\beta}_1 X_1 + \beta_2 X_2 + \epsilon$ where $\widetilde{\beta}_1 = \beta_1 + \beta_3 X_2$. With this, we have now dealt with the interaction between predictors and thus relaxed the additive assumption. The **hierarchical principle** states that if we are going to include the interaction term (if it is statistically significant) we should also include the 'main effects' - the base predictors that are interacting, even if they are not statistically relevant.

Relaxing The Linear Assumption: The linear assumption can't be directly removed (obviously, or it wouldn't be linear regression) but it can be manipulated to allow for expression of non-linear relationships. Using **polynomial regression** we can express non-linear relationships in a linear framework: we can add extra parameter-predictor pairs to the model, modifying the predictor with some non-linear function, e.g. we could add $\beta_j x_i^2$, $\beta_j \sqrt{x_i}$, and so on. This is still linear but allows us to incorporate non-linearities. This is trivially extended to multiple linear regression and can even be used with interaction terms, possibly even with non-linear interaction terms, although it gets intractably complicated quite fast.

Potential Problems

- **Non-linearity of the response-predictor relationship:** If there is a complex non-linear relationship then there is nothing we can realistically do to fix our linear model (short of making it non-linear). The task is instead to identify that this is the case. We can use a **residual plot** to identify a non-linearity, checking for patterns or anything that would suggest that the residuals are not randomly distributed.
- **Correlation of error terms:** If error terms are correlated then the standard error we calculate tends to underestimate the true standard error. This means that our confi-

dence and prediction intervals are narrower than they should be (they actually represent less % than the X% interval they are intended to represent) along with the p-values which are smaller than they should be, perhaps leading us to erroneously conclude that a parameter is statistically significant. This is most prevalent in time series data where nearby residuals may have similar values, known as **tracking**.

- **Non-constant variance of error terms:** Once again, standard errors, hypothesis tests and confidence intervals rely on a constant variance, $Var(e_i) = \sigma^2$. It can be easy to diagnose this problem by looking at a **residual plot** and looking for a funnel shape which would imply **heteroscedasticity** (a funnel shape suggests that variability increases/decreases as you cycle across the $\hat{Y}$ domain). One solution is a transformation of the response Y to counteract the shape: $log(Y)$ or $\sqrt{Y}$ are common transformations. It could also be the case that we know the variance of each response and we can therefore perform the regression weighted by the inverse variance to counteract the funnel.

- **Outliers:** Outliers don't tend to have a large effect on the regression itself but do have a noticeable effect on our performance metrics. Spotting outliers is usually just a case of plotting the graph (or residual graph) and spotting them by hand. This is subjective of course and instead we can plot the **studentized residuals** which should allow us to better spot outliers.

- **High Leverage Points:** High leverage points are similar to outliers but where outliers are outliers in the response, high leverage points are outliers in the predictors. That is to say, high leverage points have an unusual combination of predictor values even if the resultant response is normal - if X_1 and X_2 are positively correlated then it would not be surprising to see a (relatively) large x_1 with a large x_2, but it would be surprising (and thus a high leverage point) to see a large x_1 with a small x_2 or vice versa. We can use a **leverage statistic** to identify these high leverage points.

- **Collinearity:** Collinearity is the linear correlation of two predictors. It reduces the accuracy of parameter estimates and causes their standard error to grow and thus a decline in the t-statistic. Diagnosing collinearity can sometimes be a case of just looking at a correlation matrix and identifying correlated predictors, either including a interaction term/combining the offending predictors in to a single new predictor or removing one of the predictors if it is deemed redundant. However if multiple predictors (> 2) are collinear with each other we have a case of **multi-collinearity** which may not necessarily be seen in a first-order correlation matrix. It would be possible to generate a second-order correlation matrix (looking at correlations of predictor pairs with other predictors/pairs) but it quickly becomes unfeasible especially if the correlation is hidden in third, fourth, etc order graphs. Instead we can compute the **variance inflation factor (VIF)** to diagnose collinearity.

3.2.2 Subset Selection

Subset selection aims to improve the regression by reducing the number of predictors p that need to be considered and therefore makes our dataset n go further (as in, the dataset has a limited amount of information so by reducing p we have more information to accurately predict the remaining predictors). Because of this, subset selection can be used for problems where $p > n$ or $p \approx n$ and should discover the maximum number of useful predictors given n (or if n is large enough, then it will discover all useful predictors). There are three

types of subset selection: **best subset selection**, **forward stepwise selection** and **backward stepwise selection**. The latter two can also be combined in to a **hybrid stepwise selection**.

Best Subset Selection

Best subset selection involves fitting a separate least squares regression (the high-level algorithm can be applied to any model however, for example logistic regression) for each possible combination of the p predictors which obviously scales very inefficiently as p increases. The algorithm selects the best model M_k for $k = 0, ..., p$ where M_k is the best performing model, scored with RSS or equivalently R^2, that contains k predictors. With these best models selected, the overall best model is selected from these p models using **cross-validation**, C_p, **AIC**, **BIC** or adjusted R^2. We use RSS within each M_k as it allows us to grade the models however RSS will decrease (R^2 will increase) as k increases and so we can't use it to compare across M_ks as it would always select for the models with more predictors. This is why we use the other measures when selecting the optimal p value.

Best subset selection allows us not only to find the most optimal model but it also allows us to work out which of the p predictors provide useful information for the regression (which predictors does the response depend on). As mentioned before it also allows us to find the predictors that give the most information when we have a limited data set ($n \approx p$) and need to drop predictors to find a closed solution. The obvious downside with this is the massive computing requirements, in general there are 2^p different models that must be considered. **Stepwise selection** is a solution to this.

Forward Stepwise Selection

Rather than considering all possible combinations of predictors p we instead consider the highest performing predictor at each step and add that to the model. We start with a **null model**, M_0 containing no predictors and for $k = 0, ..., p-1$ we define M_{k+1} which is the model M_k with the best performing predictor not already used added to the model, scored using RSS and R^2. In other words at $k = 0$ we select the best performing predictor and create a new model M_1 which has exactly one predictor (the best performing one). At $k = 1$ we test each remaining predictor with the model M_1 (one at a time, they are not tested together in the same model) and select the new best predictor, adding that to the model M_1 (which becomes M_2). We then repeat this until M_p is defined. With all M_ks defined, we select the best one with the same metrics as best subset selection.

This approach is considerably more computationally feasible than best subset selection, requiring only $1 + \sum_{k=0}^{p-1}(p-k)$ models (as opposed to 2^p). The downside is that we do not consider every possible combination of predictors and thus we have no guarantee that the best combination will be found (for example, if we have four predictors the optimal combination is 2 and 3, but on step $k = 0$ we find the best predictor to be 4, we will always have 4 in our final model which we can see doesn't appear in the optimal model). As with best subset selection, forward stepwise selection can be used with $n < p$ however it is not possible to evaluate models M_k for $k \geq n$.

Backward Stepwise Selection

Follows a very similar algorithm to forward stepwise selection but instead starts with the fully fitted model, M_p, and removes a predictor at each step by testing models M_{p-1} with each predictor removed, continuing until the null model M_0 is found. Unlike the previous two methods, backward stepwise selection requires that $n > p$.

Hybrid Stepwise Selection

One can combine forward and backward stepwise selection in to a hybrid solution: at each step perform forward stepwise selection and once the optimal model is obtained for each k, check if model performance can be improved by removing any one variable - perform backward stepwise selection. If model performance cannot be improved then move on to the next model M_{k+1} and if performance can be improved, remove that variable and perform forward stepwise selection again for model M_k. If a cycle is found (each time a variable is added a variable is removed and eventually a cycle forms) then either break it manually or conclude that the best subset has been found.

3.2.3 Shrinkage (Lasso/Ridge)

Shrinkage methods also aim to reduce p to the minimal optimal subset however unlike subset selection which explicitly removes predictors in a least squares model, shrinkage models modify the least squares optimization such that the optimization process itself leads to small or zero coefficients for certain predictors, otherwise known as **constrained** or **regularized** coefficients. For reference, the standard least squares update rule is to minimize:

$$RSS = \sum_{i=1}^{n}(y_i - \beta_0 - \sum_{j=1}^{p}\beta_j x_{ij})^2$$

Over all β_i.

Ridge Regression

Ridge regression modifies the least squares update rule to be:

$$RSS + \lambda \sum_{j=1}^{p}\beta_j^2$$

Where $\lambda \geq 0$ is a **tuning parameter**. Ridge regression works by optimizing βs to best model the relationship, much like least squares, but to also penalize large βs and thus **shrinks** the coefficient estimates toward zero. The second term in the above equation is known as the **shrinkage penalty** and the level of shrinkage is determined by λ which can be defined

manually or an optimal value discovered with cross validation. This shrinkage penalty is also known as l_2 **regularization** which has the effect of penalizing large weights.

It turns out that coefficients in least squares regression are **scale equivariant** - multiplying a predictors by c has the effect of multiplying its coefficient by $\frac{1}{c}$. Ridge regression does not have this property. Changes in the scale of predictors can have large and unpredictable effects to the scale and value of coefficients. We therefore want to **standardize** the predictors:

$$\widetilde{x}_{ij} = \frac{x_{ij}}{\sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_{ij} - \bar{x}_j)^2}}$$

Where the denominator is the estimated standard deviation of the jth predictor.

In ridge regression, λ acts as a way to modify model flexibility. By increasing λ we reduce the flexibility of the model and thus reduce variance (but increase bias). This is stronger than the standard least squares model as a ridge regression still allows for a least squares model if that is optimal ($\lambda = 0$) but some problems won't need the flexibility of even least squares. As such, ridge regression tends to work in problems where least squares estimates have high variance which ridge regression can reduce at the expense of increased bias - a balance would have to be found (a search over λ).

The alternate form for ridge is:

$$minimize_\beta \left(\sum_{i=1}^{n}(y_i - \beta_0 - \sum_{j=1}^{p}\beta_j x_{ij})^2 \right) \text{ subject to} \lambda \sum_{j=1}^{p}\beta_j^2 \leq s$$

The Lasso

Ridge regression does not solve the problem of some predictors being uncorrelated or having low information gain - all predictor coefficients tend toward zero in ridge as opposed to a select few. Ridge regression instead offers a variance reduction and prevents high-coefficient predictors dominating the other predictors. Lasso regression is very similar to ridge regression but instead of using l_2 regularization we use l_1:

$$RSS + \lambda \sum_{j=1}^{p}|\beta_j|$$

The key difference here is l_1 tends to set small weights equal to zero whereas l_2 tends all weights towards zero (but usually doesn't equal zero). This can be understood geometrically as l_1 regularization creates (in 2-dimensions) a diamond (of size given by λ) which leads to a situation where the optimal value of the above equation (where both lines intersect) occurs at the vertices. These vertices have one of the variables equal to zero. l_2 on the other hand creates a circle (in 2-dimensions) which means the chance of intersecting where one or more variables is zero is low. This effect is easier to see by looking at the alternate form for ridge (above) and lasso:

$$minimize_{\beta} \left(\sum_{i=1}^{n} (y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_{ij})^2 \right) \text{ subject to } \sum_{j=1}^{p} |\beta_j| \leq s$$

Where we can imagine drawing a circle/diamond of radius s and finding the closest point of intersection to the RSS minimum, which usually occurs at an axis (vertex) for l_1 and can occur anywhere for l_2.

3.2.4 Dimensionality Reduction

Principal Component Regression

Principal component regression (PCR) works by using principal component analysis to build a regression model on the first M principal components. The assumption is that the directions which show the most variation are the most associated with Y, that is to say, the first Z predictors. PCR is not a feature selection method (like best-set or lasso) as it still uses all X predictors, just some reduced linear combination of them. In this sense, it is more similar to ridge regression. When performing PCR it is a good idea to standardize predictors first so that scale does not play a role in selecting the principal components. M is then selected with cross validation.

3.3 Non-linear Regression

3.3.1 Basis Functions

Basis functions are the more generalized form for many non-linear regression approaches. Essentially they take the standard linear regression form:

$$y_i = \beta_0 + \beta_1 x_i + \eta_i$$

And replace the x_i with a **basis function**:

$$y_i = \beta_0 + \beta_1 b_1(x_i) + ... + \beta_K b_K(x_i) + \eta_i$$

Where we can see that, for simple linear regression, $K = 1$ and $b_1(x_i) = x_i$. For polynomial regression we may have, for example, $b_k(x_i) = x_i^k$. As this still resembles a linear regression problem, all our tools are still available: training algorithm, error measures and so on.

Selecting $b_k(x_i)$ functions is entirely up to the user and many different basis functions exist.

3.3.2 K-Nearest Neighbours

A non-parametric approach to estimate $f(X)$ using:

$$f(\hat{x}_0) = \frac{1}{K} \sum_{x_i \in \mathcal{N}_0} y_i$$

Which is read as $\hat{y}_0$ is equal to the sum of the K nearest Y values (calculated from the K nearest X values) around x_0 divided by $\frac{1}{K}$.

There exists the **curse of dimensionality** which is especially prevalent in K-NN as, for example, with 100 data points and 20 predictors, the closest point to any X point may actually be quite far away in the 20 p dimensions (if we think of volume, its not 100 split between 20, its more like 100 split between $\propto r^{20}$).

3.3.3 Polynomial Regression

Polynomial regression involves manually adding polynomial terms to the standard linear regression:

$$y_i = \beta_0 + \beta_1 x_i + \eta_i$$

Becomes:

$$y_i = \beta_0 + \beta_1 x_i + \beta_2 x_i^2 + ... + \beta_d x_i^d + \eta_i$$

For polynomial regression up to the dth order.

Since this is still effectively a linear regression problem all our tools with linear regression are still available: how we train it, our error metrics, etc.

We can also calculate error (variance) on the predicted value, $\hat{f}(x) = \hat{y}$, by using the variances of our predicted $\hat{\beta}$ values:

$$Var(\hat{f}(x_0)) = l_0^T \hat{C} l_0$$

For $l_0^T = (1, x_0, x_0^2, ..., x_0^d)$ and $\hat{C}$ is a $d \times d$ covariance matrix for $\hat{\beta}$. This only gives us the variance at a specific point x_0 so to calculate the variance across the entire domain we could create a grid of x points and calculate at each grid point. To obtain the error here, we simply root the variance. Generally, for normally distributed errors, the 95% confidence margin is approximately two times the standard error ($\sqrt{(Var)}$).

Polynomial regression can also be used with logistic regression to tackle non-linear classification problems without any changes to the base logistic regression algorithm (much the same way polynomial regression is essentially linear regression).

3.3.4 Step Functions

While polynomial regression imposes a global structure for f, step functions seek to split the domain in to some number of bins and fit a constant per bin. This amounts to converting a continuous variable in to an **ordered categorical variable**.

Essentially, we select some number, K, of cut points c_k in the range of X, usually uniformly distributed although not by necessity, and create new variables:

$$C_k(X) = I(c_k \leq X < c_{k+1})$$

With $c_0 = 0$ and $c_{K+1} = max(X)$. We then construct our response function as:

$$y_i = \beta_0 + \beta_1 C_1 + ... + \beta_K C_K + \eta_i$$

Since we have For a given value of x, only one C_k can be non-zero.

Once again, since this is essentially a linear regression problem, we have access to all our normal tools. It also adapts similarly to logistic regression.

3.3.5 Regression Splines

Regression splines are another basis function approach where we fit a polynomial function within each bin. In that sense, they are a combination of polynomial regression and step regression. The problem is that, when applying this naively, there is a discontinuity at the boundaries of each bin. We could require that $b_j(c_{j+1}) = b_{j+1}(c_{j+1})$ - that the function value should be equal in both bins at their shared boundary, known as continuity - but then we get odd behaviour such as instantaneous change in direction. To fix this added problem, we impose two extra conditions: that the function value (the original condition), its first derivative and second derivative all be continuous. Selecting the second derivative is a mathematically arbitrary selection however for humans this makes the discontinuity imperceptible.

In splines, the bin boundaries are known as **knots**. To create an optimisation function for regression splines, we start with a polynomial regression function:

$$y_i = \beta_0 + \beta_1 x_i + ... + \beta_K x_i^K + \eta_i$$

And, for each knot, we add a **truncated power basis function** defined as:

$$h(x, \xi) = (x - \xi)_+^n = \begin{cases} (x - \xi)^n & \text{if } x > \xi \\ 0 & \text{otherwise} \end{cases} \tag{3.1}$$

Where ξ is the knot location (in x) and n is the polynomial degree, where usually you use $n = 3$. In the end, for K knots and $n = 3$, we are fitting X, X^2, X^3, $h(x, \xi_1)$, ..., $h(x, \xi_K)$ with a β for each.

This gives us $n + K + 1$ degrees of freedom with 1 being the intercept.

One finds that at the boundaries, $x < \xi_1$ and $x > \xi_K$, there is high variance. A solution to that is **natural splines** where the function in each boundary area is restricted to be linear.

Deciding on the number of knots K, and where to place them, can be decided either with knowledge of the function behaviour (we may want more knots where the function behaves more non-linearly) or with cross validation and similar methods.

3.3.6 Smoothing Splines

Smoothing splines take a different approach to the basis function approach. They intend to find an arbitrary $g(x)$ that best fits the data with the added condition of smoothness. Without a smoothness condition, one could find a $g(x)$ that perfectly interpolates the training data but would obviously score abysmally on test data. We also want the smoothness restriction because it looks nice and makes more sense: would be strange to have a prediction be wildly different from the prediction directly next to it (in x). To achieve this the minimization function becomes:

$$\sum_{i=1}^{n}(y+i-g(x_i))^2+\lambda\int g''(t)^2dt$$

Where λ is a non-negative tuning parameter. The function $g(x)$ that minimizes this function is the **smoothing spline**. It can be shown that the function that satisfies this minimization is a piecewise cubic polynomial with knots at each unique x_i and has continuous first and second derivatives at each knot. Furthermore, it is linear beyond each boundary. This obviously resembles a natural spline with knots at all unique x_i however the tuning parameter λ shrinks the result such that it is not the same natural spline we would get if we applied the basis function approach.

Since the λ parameter smooths the line, it has the effect of reducing the **effective degrees of freedom** even though, according to the same degrees of freedom metrics in natural splines, it would have n (where n is the total number of data points) degrees of freedom (for each knot). It can be shown that the effective degrees of freedom reduces from $n \to 2$ as λ is increased. We can write the solution as:

$$\hat{g}_\lambda = S_\lambda y$$

Where $\hat{g}_\lambda$ is our predicted $g(x)$ for a given λ and S_λ is a $n \times n$ matrix of fitted values. Using this, we can define the effective degrees of freedom as:

$$df_\lambda = \sum_{i=1}^{n}(S_\lambda)_{ii}$$

Or, the **trace** of S_λ.

As it turns out, selecting λ, which could be done naively with standard cross validation, is computationally very easy as one can essentially perform a full LOOCV pass (which would require n fitted models) for each λ in a single pass with the equation:

$$RSS_{cv}(\lambda) = \sum_{i=1}^{n}(y_i - \hat{g}_\lambda^{-i}(x_i))^2 = \sum_{i=1}^{n}\left[\frac{y_i - \hat{g}_\lambda(x_i)}{1-(S_\lambda)_{ii}}\right]^2$$

Where $\hat{g}_\lambda^{-i}(x_i)$ is the fitted value of the spline evaluated at x_i with the ith data point left out of training (LOOCV). Here, the left hand side would be the standard LOOCV approach while the right hand side is our convenient trick.

3.3.7 Local Regression

Local regression involves computing a weighted regression line at each target point x_0 using some chosen number of nearby points.

Performing local regression requires the selection of regression type (constant, linear, poly-

nomial), the number of nearby points (span, s) and the weighting function. The weighting function can be any function, but generally is set such that being closer in Euclidean distance means having a higher weight, with the closest point having the largest weight and the furthest point (within the nearby points) having weight 0, and all other points outside of the nearby points having weight 0. For example, a simple weight function could be truncated Euclidean distance where any distance greater than the distance of the furthest point within the chosen points is set to zero and all others have weight set to their Euclidean distance.

The process of local regression involves finding a regression line at each x_0 minimizing (for linear regression):

$$\sum_{i=1}^{n} K_{i0}(y_i - \beta_0 - \beta_1 x_i)^2$$

Where $K_{i0} = K(x_i, x_0)$ is a weight function for point x_i relative to point x_0.

One can expand local regression, for example, in the case of multiple predictors $X_1, ..., X_p$ we could have a regression local in some predictors but global in others. Local regression scales with multiple predictors although as p becomes large we have *curse of dimensionality* problems.

3.3.8 Generalized Additive Models

Many of the non-linear regression approaches detailed above apply only to single-predictor regression ($p = 1$). Generalized additive models (GAMs) provide a generalized framework that allows for these techniques, and others, to be used in a multiple predictor setting ($p > 1$). As in the name, these preserve additivity found in linear regression.

A GAM takes the form:

$$y_i = \beta_0 + \sum_{j=1}^{p} f_j(x_{ij}) + \eta_i$$

Where for simple linear regression, $f_j(x_{ij}) = \beta_j x_{ij}$. GAMs learn a separate f_j for each X_j and are therefore additive.

For a regression spline approach, for example, GAMs provide a simple way of extending them to multiple predictors. Each f_j would be a whole regression spline and they could all be added to the function and solved with least squares simultaneously. Is it not enough to solve each regression spline individually however as each spline may take on a different form if variables are added to the model, which we cannot accommodate if they are trained individually and added at the end. Regression splines are useful in this sense as when formulated as a GAM they can still be trained with least squares. Smoothing splines, for example, cannot be fit like this and are instead fit with a method known as **backfitting**. Backfitting involves updating each model individually, keeping the others fixed, at each iteration.

3.4 Classification

3.4.1 The Bayes Classifier

The Bayes classifier is an ideal classifier that always places an unknown observation in the class for which is has the highest probability of membership. It assigns to a class j for which

$$Pr(Y = j|X = x_0)$$

is maximized. Where Y is the observation's unknown class and x_0 is the observations predictor values. Simple enough, but actually calculating this probability is usually not directly possible and we therefore either need to approximate it or use another method entirely.

We can use this to define a Bayes **decision boundary** where we can decide an observation's class by which side of the boundary it falls on.

The Bayes classifier also gives rise to an ideal, irreducible error rate which is the theoretical best performance any classification algorithm could obtain given the data available:

$$1 - E[max_j Pr(Y = j|X)]$$

Where max_j is the value of j which maximizes $Pr(...)$. The expectation is the average over all possible values of X. This error measure can be read as the error when all observations (all values of X) are assigned to their highest probability class which, if we knew the conditional distribution $Pr(Y|X)$, would be exactly the error we would obtain when using the Bayes classifier and since the Bayes classifier is ideal so to is the error (it is irreducible).

3.4.2 K-Nearest Neighbour Classifier

The K-NN algorithm seeks to classify an observation by setting its class to the majority class the K closest known observations. Specifically, K-NN provides a way to estimate class probabilities by relating them to the proportion of each class in the neighbourhood:

$$Pr(Y = j|X = x_0) = \frac{1}{K} \sum_{i \in \mathcal{N}_0} I(y_i = j)$$

Where $\mathcal{N}_0$ is the set of the K nearest data points, $I(...)$ is an indicator variable which equals 1 if true and 0 otherwise. For K-NN, 'nearest' can mean anything as long as its a consistent measure.

With these class probabilities, we can follow the Bayes classifier and set the unknown class to the class j which maximizes $Pr(...)$.

3.4.3 Logistic Regression

Logistic regression models the probability that a the response belongs to a particular category given the predictors, $P(class|predictors)$ for n classes. Following the Bayes classifier you can then either assign the response to the class i for which $P(class = i|predictors)$ is highest or for binary problems define a threshold for class membership, i.e. when $P(class = true|predictor) > threshold$.

The key to predicting probabilities is to ensure a result between $[0, 1]$ (or between any fixed boundary which can be mapped). For logistic regression we use a **logistic function** to achieve this:

$$P(Y = 1|X) = p(X) = \frac{e^{\beta_0+\beta_1 X}}{1 + e^{\beta_0+\beta_1 X}}$$

Where, in general, a logistic function is:

$$f(x) = \frac{L}{1 + e^{-k(x-x_0)}}$$

Where L is the maximum value, x_0 is the y-axis midpoint and k is a parameter (curve steepness). We can see that $\beta_1 = -k$ and $\beta_0 = kx_0$. The logistic function produces an S-shaped curve bounded between 0 and L.

We can rearrange $p(X)$ to produce

$$\frac{p(X)}{1 - p(X)} = e^{\beta_0+\beta_1 X}$$

Where the left hand side is known as the **odds** and can take any value between 0 and ∞. We can also take the logarithm of both sides to produce a **logit** or **log-odds**:

$$log\left(\frac{p(X)}{1 - p(X)}\right) = \beta_0 + \beta_1 X$$

Where the left hand side is our logit.

Our job is to then estimate the coefficients β for which we will use **maximum likelihood**. Maximum likelihood works by setting parameters such that the predicted class (given class probabilities) best matches the true class. The **likelihood function** that we wish to maximize is:

$$l(\beta_0, \beta_1) = \prod_{i:y_i=1} p(x_i) \prod_{i':y_{i'}=0} (1 - p(x_{i'}))$$

Maximization of this would proceed by differentiating with respect to each parameter and

setting equal to 0. Additionally, the likelihood function can be manipulated to make it easier to manage, for example by taking the logarithm and turning the products in to sums. Predicted values can be analysed in much the same way as linear regression, primarily with a standard error calculation and computing the z-statistic.

Logistic regression can also be extended in two areas: multiple logistic regression where there are multiple predictors ($X = (x_1, ..., x_n)$) and logistic regression for > 2 response classes. The former extension is a simple matter of modifying $p(X)$ to be:

$$p(X) = \frac{e^{\beta_0+\beta_1 X_1+...+\beta_n X_n}}{1 + e^{\beta_0+\beta_1 X_1+...+\beta_n X_n}}$$

And using this modified $p(X)$ with maximum likelihood optimisation as before. Extending to multiple response classes is a more difficult task and instead of using logistic regression for this (which can be done) you would tend to use **linear discriminant analysis** instead.

3.4.4 Linear Discriminant Analysis (LDA)

Logistic regression works by assuming a form for $p(X) = P(Y|X)$ and setting the parameters to best model the data. Linear discriminant analysis (LDA) instead models the conditional distribution for predictors given class, $P(X|Y)$ and then uses Bayes' theorem to find $p(X)$. The net result is an equation for $P(Y|X)$:

$$p_k(X) = P(Y = k|X = x) = \frac{\pi_k f_k(x)}{\sum_{l=1}^{K} \pi_l f_l(x)}$$

Where π_k is our **prior** and represents the probability that an observation at random comes from class k (i.e. the class distributions) and $f_k(x)$ is the density function of X for an observation that comes from the kth class (i.e. given a class what are the likely X values, equivalent to $P(X|Y)$). Usually we can estimate π_k by calculating the proportion of class k out of all classes in our training data. Estimating $f_k(X)$ is more difficult however and we have to assume a form for f.

For the one dimensional case where we only have 1 predictor ($p = 1$), we assume f takes the form of a one dimensional **Gaussian**:

$$f_k(x) = \frac{1}{\sqrt{2\pi}\sigma_k} exp\left(-\frac{1}{2\sigma_k^2}(x - \mu_k)^2\right)$$

Where μ_k and σ_k^2 are the mean and variance parameters for the kth class (mean/variance of the single predictor x for class k). We simplify this by assuming $\sigma_k = \sigma_1 = ... = \sigma_n$ which we denote σ. Plugging this in to $p_k(x)$ and taking the log we can find the form:

$$\delta_k(x) = x \cdot \frac{\mu_k}{\sigma^2} - \frac{\mu_k^2}{2\sigma^2} + log(\pi_k)$$

Where, under LDA, we assign class k to observation x for the class which maximizes $\delta_k(x)$. $\delta_k(x)$ is known as the **discriminant function**. The *linear* in LDA comes from the fact that the discriminant function is linear in x.

LDA approximates the unknown parameters π_k and σ with

$$\hat{\pi_k} = \frac{1}{n_k} \sum_{i:y_i=k} x_i$$

$$\hat{\sigma^2} = \frac{1}{n-K} \sum_{k=1}^{K} \sum_{i:y_i=k} (x_i - \hat{\mu_k})^2$$

For $p > 1$, we assume the predictor X is drawn from a **multi-variate Gaussian** where the class mean becomes a vector of size p (a mean per class k for each predictor X_i), $E(X_k) = \mu_k$, and the the common variance becomes a covariance matrix of size $p \times p$, $Cov(X) = \mathbf{\Sigma}$. With this, $f_k(x)$ becomes:

$$f_k(x) = \frac{1}{(2\pi)^{p/2}|\mathbf{\Sigma}|^{1/2}} exp\left(-\frac{1}{2}(x-\mu_k)^T\mathbf{\Sigma}^{-1}(x-\mu_k)\right)$$

We can then define an analogous discriminant function:

$$\delta_k(x) = x^T\mathbf{\Sigma}^{-1}\mu_k - \frac{1}{2}\mu_k^T\mathbf{\Sigma}^{-1}\mu_k + log\pi_k$$

The linearity of x can also be shown in the $p = 1$ case, however we show it here. When working through the derivation and eliminating all constant terms (since we only care about relative differences), we find a term $x^T\mathbf{\Sigma}^{-1}x$ which is quadratic in x. Since this is not dependent on k, it will be constant over all classes and can be eliminated and thus we eliminate the only non-linear x term. Note: $\delta_k(x) \neq logp_k(x)$! $\delta_k(x)$ is $logp_k(x)$ rearranged and with all constant terms eliminated as we only care about assigning class k where $p_k(x)$ is greatest, we don't care for the specific value of $p_k(x)$.

3.4.5 Quadratic Discriminant Analysis (QDA)

Quadratic Discriminant Analysis is similar to LDA but extends it to relax the assumption that all classes share a common covariance matrix. Specifically, QDA follows the exact same process as multi-predictor ($p > 1$) LDA but with each class having its own covariance matrix. The *quadratic* part comes arises from the discriminant function taking on a quadratic form as opposed to the linear discriminant function in LDA. QDA assumes that observations from the kth class is of the form $X \sim N(\mu_k, \mathbf{\Sigma}_k)$. The discriminant function is then (following the same derivation as LDA):

$$\delta_k(x) = -\frac{1}{2}(x-\mu_k)^T\Sigma_k^{-1}(x-\mu_k) - \frac{1}{2}log|\Sigma_k| + log\pi_k$$

Upon simplifying this equation we find a term $x^T \mathbf{\Sigma}_k^{-1} x$. Unlike the LDA case, this is now dependent on k and will be different between classes. We cannot eliminate it and therefore we introduce a quadratic term in x, hence *quadratic* in QDA.

QDA introduces a considerable number of parameters over LDA and thus is a more flexible model, at the expense of increased variance (but reduced bias) and complexity.

3.4.6 Decision Trees

Note: See the **tree based methods** section for advanced methods for tree based methods, which can be applied to the following discussion. This section only considers modifications required for classification.

Much the same as with regression, classification tree building requires that we provide some measure of error. This is used at each decision point to work out the optimal regions. Naively one would use the **most commonly occurring class** in place of the average response in the regression RSS or more specifically, we would define the error as the proportion of observations which are *not* members of the most commonly occurring class in that region, known as **classification error**. As it turns out, this is not a sensitive enough measure to properly build the tree.

Instead, tree building is done using two other measures. The **Gini index**:

$$G = \sum_{k=1}^{K} \hat{p}_{mk}(1 - \hat{p}_{mk})$$

Where $\hat{p}_{mk}$ represents the proportion of training observations in the mth region belonging to the kth class. The Gini index is a measure of total variance across the K classes. The Gini index would be small for regions where the vast majority of observations are dominated by one class ($\hat{p}_{mk}$ for all classes that are not the majority class are equal to 0 or 1) and is therefore regarded as a measure of *purity*.

Alternatively, one can use **entropy**:

$$S = -\sum_{k=1}^{K} \hat{p}_{mk} log(\hat{p}_{mk})$$

Entropy will also be small if most $\hat{p}_{mk}$s are 0 or 1. In fact, Gini and entropy measures are numerically very similar, although their mathematical justification differ.

When building a tree you would generally use Gini or entropy. When pruning a tree however the first option, classification error, tends to perform better if the accuracy of the pruned tree is your primary concern. Although for pruning any of the three will work.

3.5 Tree-based Methods (Class/Reg)

3.5.1 Decision Trees

Tree based methods involve segmenting the predictor space in to a number of simple regions defined by some set of rules. One can then traverse the tree, evaluating the current example against each rule, to further reduce the predictor space of interest (e.g. for x = 0.3 and a predictor space of [-1,1], the 'greater than 1' rule means we only have to look at half of our predictor space). Depending on how the rules are formulated, this could be in terms of reducing size (the first rule splits the entire predictor space equally in two, the second splits one of the halves in two again so it is clearly a smaller divide), in terms of maximal information gain (decisions which result in more information are performed first) or even in terms of evaluation complexity (compute cheap rules first to get a good approximation before expensive refinement).

Decision trees are called as such because they can be represented graphically as a tree where at each each 'internal node' we make a decision (evaluate a rule) to decide which 'branch' we go down (left or right for a 2-dimensional tree) which we repeat at the next internal node until we reach a 'terminal node' or a 'leaf' which is our decision given our measurement - this could be an explicit value, it could be the average value of the response in that segment of the predictor space, and so on.

The training procedure for stratifying a predictor space (building a decision tree) would be to find boxes (n-dimensional rectangles), $R_1, ..., R_J$ that minimize the RSS:

$$\sum_{j=1}^{J} \sum_{i \in R_j} (y_i - \hat{y}_{R_j})^2$$

This is computationally infeasible however as there are too many possible boxes considering the number of boxes J and the size of each box. Instead, we use a *top-down*, *greedy* approach known as **recursive binary splitting**. For J predictors:

- Select a predictor X_j and a cut point s that minimizes the above RSS for the regions $R_1(j,s) = X|X_j < s$ and $R_2(j,s) = X|X_j \geq s$, where $\hat{y}$ is the mean response for observations in that region.

- Repeat the process but instead of looking at the entire predictor space, perform the above split in each individual region, selecting the single split that minimizes the RSS (as in, while we find the optimal split in each region, we only use the better of the two). This algorithm continues, selecting the single best split at each iteration.

- Continue until some **stopping criteria** is met, such as a certain number of regions, requiring that each region contains no more than N observations each, and so on. It is important to note, a stopping criteria is necessary. Without one, the algorithm would continue until a single observation exists in each region which has the effect of overfitting to the point where the training set achieves 100% performance.

Pruning

One method of avoiding overfitting is by pruning the tree. Pruning involves growing the full tree and working bottom-up to remove nodes in order of smallest gain in accuracy (which could also be a *reduction*) on a validation set. **Cost complexity pruning** or **weakest link pruning** parametrises this process by reformulating the RSS measure to include a penalty on tree complexity parametrised by α:

$$\sum_{m=1}^{|T|} \sum_{i:x_i \in R_m} (y_i - \hat{y}_{R_m})^2 + \alpha |T|$$

For some subtree $T \subset T_0$. Here, $|T|$ is the number of terminal nodes (the complexity) in the tree. Selecting α is usually done with cross-validation whereby you create your K-folds and select K values for α. For each fold, generate a tree on the training set and perform cost complexity pruning to produce an optimal subtree for the given α - nodes are iteratively removed thereby reducing —T— and based on the $\alpha|T|$ term there will be some optimal complexity. This is repeated on the next fold and after all folds are trialled we average the results for a given α. This can be performed for a range of α values. It is important to note that because of the discrete nature of decision trees (you can't have 1.5 nodes!), there will be some range $\alpha_1 \leq \alpha < \alpha_2$ where the optimal subtree will be the same. One can then select the best α.

3.5.2 Bagging

Bagging is the process of applying the bootstrap method to decision trees. As per bootstrapping, we generate B datasets by randomly drawing from our original dataset with replacement and we usually draw the same number of elements that our original dataset had. These B datasets are not necessarily unique but will usually be so. We train a deep decision tree on each dataset without pruning. This is the bagged model. We can then make predictions using our bag of B trees by performing the inference with each tree individually and averaging the results. While normally a deep decision tree would have very high variance, it can be shown that the variance of a set of n variables that have their own variance σ is $\frac{\sigma}{n}$ (this is, of course, only if these are uncorrelated variables - this is not always the case as we will see next). As such, if B (equal to n here) is large our resultant bagged model will have low variance despite having deep trees. This, in essence, avoids the overfitting problems with deep trees.

We can even generate a valid estimate for the test error with very little added computation. Normally we would need to perform cross validation to estimate test error but due to how bootstrap datasets are drawn, it can be shown that each dataset will use, on average, two-thirds of the original dataset. This means, that for any observation, there exists $\frac{B}{3}$ generated trees that were not generated with that observation. In essence, this acts as a test set for that observation, much like cross validation. You can calculate the average error for that observation across those trees and then perform the same process across all observations, averaging that result also. The result will be a valid estimator for the test set MSE. This is known as **out-of-bag error**.

While bagging is less interpretable than a single decision tree, one can still generate a mea-

sure of variable importance. We could calculate the average MSE reduction for a split on a given variable across our entire bag, or equivalently calculate the average Gini index for classification problems. While our bagged model may not be as graphically interpretable we can still give a measure of which variables are most important or offer the most gains.

3.5.3 Random Forest

A random forest is very similar to the bagging approach. One problem with bagging is that for correlated trees - trees that are very similar - the reduction in variance may not be as great as desired. This could arise if there is one very dominant predictor. In such a case, almost all trees that contained that predictor would split on that predictor as the root which means most trees would look pretty similar. Random forest gets around this by randomly selecting a subset m, $m < p$, of predictors that are available to be split on on each split. This means that no matter how dominant a predictor is there will always be variation in the bag B. Usually, m is set such that $m = \sqrt{(}p)$. Doing this has the effect of *decoupling* the trees in the bag which means the variance reduction in cases where there is a dominant variable or similar is maintained.

3.5.4 Boosting

Boosting is another technique to reduce the variance inherent in single trees. Boosting works by building trees sequentially with each tree using information from the previously grown tree. In particular, each new tree is grown using the residuals left over from previous trees which can be seen as each new tree is explaining whatever behaviour previous trees could not explain. This process is parametrised with λ, d and B: the shrinkage parameter, λ, which controls the rate at which boosting learns; the number of splits, d, which is the maximum tree complexity; the total number of trees B.

The algorithm is as follows:

- Set all residuals $r_i = y_i$ and set $\hat{f}(x) = 0$
- For all B:
  - Fit a new tree $\hat{f}^b$ with d splits using the data (X, r)
  - Update $\hat{f}(x) = \hat{f}(x) + \lambda \hat{f}^b(x)$
  - Update $r_i = r_i - \lambda \hat{f}^b(x_i)$
- Output the final model $\hat{f}(x) = \sum_{b=1}^{B} \lambda \hat{f}^b(x)$

3.6 Support Vector Machines (Class)

3.6.1 Hyperplanes

The goal of all SVM and SVM related methods is to discover a hyperplane. A hyperplane in p dimensional space is a "flat affine subspace of dimension $p-1$" (**affine** means it does not need to pass through the origin) - essentially it cuts the space in to two segments: a line in 2-dimensional space, a plane in 3-dimensional space. Hyperplanes are restricted to being linear although non-linear analogues to hyperplanes exist. Since hyperplanes divide the space in to two segments, they only work for binary classification problems. There are ways to adapt hyperplane approaches for multi-class problems although fundamentally each individual problem is a binary problem.

Mathematically, a hyperplane in p dimensions can be defined as:

$$H(X) = \beta_0 + \sum_{i}^{p} \beta_i X_i$$

If we can find the values of β for our dataset, this can be used as a classifier by setting one class to be $H(X) > 0$ and one class to be $H(X) < 0$. Given a data point at test we can work out which side of the hyperplane it lies on. From this a classifier can be created based on the sign of $H(X)$ and a measure of certainty be derived from the distance of the data point from the hyperplane. This can also be formulated as:

$$y_i H(x_i) > 0$$

For y_i defined as 1 for one class and -1 for the other.

3.6.2 Maximal Margin Classifier

In general, if the data is separable (if a hyperplane that separates the dataset can be found) then there will be an infinite number of hyperplanes formed by shifting or rotating the hyperplane by infinitesimal amounts. There therefore needs to be a way to decide which hyperplane is the best - the maximal margin classifier.

The MMC, otherwise known as a **optimal separating hyperplane**, is the hyperplane which maximizes the margin, where **margin** refers to the minimum perpendicular distance over all points to the plane.

The MMC operates exactly as described in the hyperplane section. It only adds rules as to what hyperplane is regarded as optimal.

The location of the margin defines a set of **support vectors**. These are p dimensional vectors which 'support' the margin in that they lie on the margin boundaries and if they were to move, the margin would change. They therefore support the margin by defining its location. In fact, the hyperplane can be found using only the support vectors - hyperplane methods in

general place no weight whatsoever on data points that are not on (or within) the margin.

Constructing the MMC

For some data points $x_1, ..., x_n \in \mathbb{R}^p$ and class labels $y_1, ..., y_n \in \{-1, 1\}$, solving for the MMC hyperplane is an optimisation problem:

$$\begin{aligned} &\text{maximize}_{\beta_i, M}\, M \\ &\text{subject to } \sum_{j=1}^{p} \beta_j^2 = 1 \\ &y_i H(x_i) \geq M \, \forall i = 1, ..., n \end{aligned}$$

Here, M is actually the margin of the MMC. This is because the second equation means that $y_i H(x_i)$ is the perpendicular distance of observation x_i from the hyperplane and equation 3 requires this quantity to be at least M. Thus, we define all points to be on the correct side of the hyperplane at least M away from the hyperplane, and our margin is defined to be the minimum distance of any given point from the hyperplane, i.e. M. This is indeed the MMC since we are maximizing M.

The MMC is not without its flaws however. In the more realistic case where a linear boundary cannot be found to separate all the points (the **non-separable case**), no solution exists.

3.6.3 Support Vector Classifier

The support vector classifier, or a **soft margin classifier**, is simply the extension of the MMC to allow for some incorrectly classified data points (data points either within the margin or on the wrong side of the hyperplane entirely). The optimisation problem for SVC is very similar to the MMC:

$$\begin{aligned} &\text{maximize}_{\beta_i, M}\, M \\ &\text{subject to } \sum_{j=1}^{p} \beta_j^2 = 1 \\ &y_i H(x_i) \geq M(1 - \eta_i) \, \forall i = 1, ..., n \qquad\qquad \eta_i \geq 0, \sum_{i=1}^{n} \eta_i \leq C \end{aligned}$$

Where C is a non-negative tuning parameter. M is still the size of the margin and η_i are **slack variables** that allow individual observations to violate the margin or the hyperplane. We can see that if $\eta_i > 0$ then observation x_i will be on the wrong side of the margin and if $\eta_i > 1$ then observation x_i will be on the wrong side of the hyperplane. As such, η_i effectively tells us where x_i is located. Since the sum of all ηs is bounded, C represents how many incorrectly

classified data points we will accept - our *budget for violations*.

As with all algorithms, C controls the bias-variance trade off. High C reduces variance by creating a more lax classifier but with increased bias. A lower C increases the variance by being more strict and fitting the training data more closely but decreases bias as a result.

As we saw with the MMC and support vectors, it turns out that the SVC also has support vectors. Only data points that either lie on the margin or violate the margin actually affect the resulting hyperplane. These are support vectors. This property explains how C controls the bias-variance trade-off: as C increases, the number of support vectors increases and you are therefore using more of the data set and reducing variance as a result but contrarily increasing bias since your result is increasingly dependent on the training set. The opposite is true for decreasing C.

3.6.4 Support Vector Machines

As SVCs were an extension to MMCs, SVMs are an extension to SVCs. In particular, SVMs take SVCs and extend the decision boundary (the hyperplane) to be non-linear.

The simple way to extend the decision boundary to accommodate for non-linear relationships is the same as with other methods: enlarge the feature space to include quadratic, cubic and higher order terms. Interestingly, in the enlarged feature space such a decision boundary is still linear. It is when this learned decision function is used in the original feature space that it produces a non-linear boundary. You could also include interaction terms or different functions altogether (other than polynomials) to represent your relationships but there is obviously far too many possible non-linearities that must be accounted for and in most cases you couldn't realistically find the right ones to enlarge the feature space with (not to mention enlarging the feature space increases the computational complexity).

The more popular way to enable a non-linear decision boundary is with **kernel functions** (see definition). The support vector classifier can be written as:

$$f(x) = \beta_0 + \sum_{i \in S} \alpha_i \langle x, x_i \rangle$$

Where S is the set of support vector indices. The task is then to find β_0 and α. In particular, the inner product $\langle x, x_i \rangle$ produces a linear decision boundary. We can generalize this function with a kernel to produce:

$$f(x) = \beta_0 + \sum_{i \in S} \alpha_i K(x, x_i)$$

Where K is our kernel function and can be set to model the behaviour we are looking for.

Fitting with a kernel amounts to fitting the support vector classifier in higher dimensional space (which produces a linear boundary) and mapping back to the original space, as before. A support vector classifier using a non-linear kernel function is known as a **support vector machine**.

SVMs For More Than Two Classes

Hyperplane methods do not lend themselves well to multi-class classification as they only divine the decision space in to two parts. There are generally two ways to get around this limitation: **one-versus-one classification** and **one-versus-all classification**.

One-versus-one classification involves creating a SVM for each pair of classes. For a test observation, it is compared in each SVM and the 'votes' are tallied up for each class. The class which the majority of SVM pairs voted for is the classification for the test observation, and a measure of certainty can be obtained from how many votes the class received compared to other classes (if it only one by 1 vote in a 20-class classification, it probably isn't very reliable).

One-versus-all classification involves comparing one class against all other classes simultaneously. We represent the $K-1$ classes as a pseudo-class ('not class K') and compare it against class K. We generate K of these SVMs, each comparing one class against all others. The classification is performed by assigning a test observation to the class K for which $f_K(x)$ is largest.

3.7 Clustering

Clustering refers to a set of methods aimed at detecting subgroups within data - points that are collectively 'closer' (with whatever measure you use to determine closeness) to each other than to other points/subgroups. In general, clustering determines **distinct** subgroups (no overlaps) and thus each observation belongs to exactly one subgroup. There are advanced methods which potentially allow for joint association or no association. Clustering can be performed either by clustering the features based on the observations (what features do similar observations posses) or by clustering the observations based on the features (what observations share similar features). For a dataset consisting of a set of users who buy certain items from a website, clustering on the features could result in subgroups of users (observations) who buy similar items e.g. a subgroup of users who all buy technology goods. Clustering on the observations could result in subgroups of items (features) bought by similar users e.g. a subgroup of expensive items bought by users with a lot of disposable income.

3.7.1 K-Means Clustering

K-means clustering involves first defining the desired number of subgroups, K. A good clustering is defined as the clustering for which the **within-cluster variation** is as small as possible, or in other words, the amount by which observations in each cluster differ from each other is minimized across all clusters:

$$\text{minimize}_{C_K} \left\{ \sum_{k=1}^{K} W(C_k) \right\}$$

Where C_k is the kth cluster and $W(C_k)$ is a measure of how much observations within that cluster differ from each other.

$W(C_k)$ can be defined by the user. A popular choice is **squared Euclidean distance**:

$$W(C_k) = \frac{1}{|C_k|} \sum_{i,i' \in C_k} \sum_{j=1}^{p} (x_{ij} - x_{i'j})^2$$

In general however, this is a computationally intractable problem. Instead, we use an approximate algorithm that gives decent results:

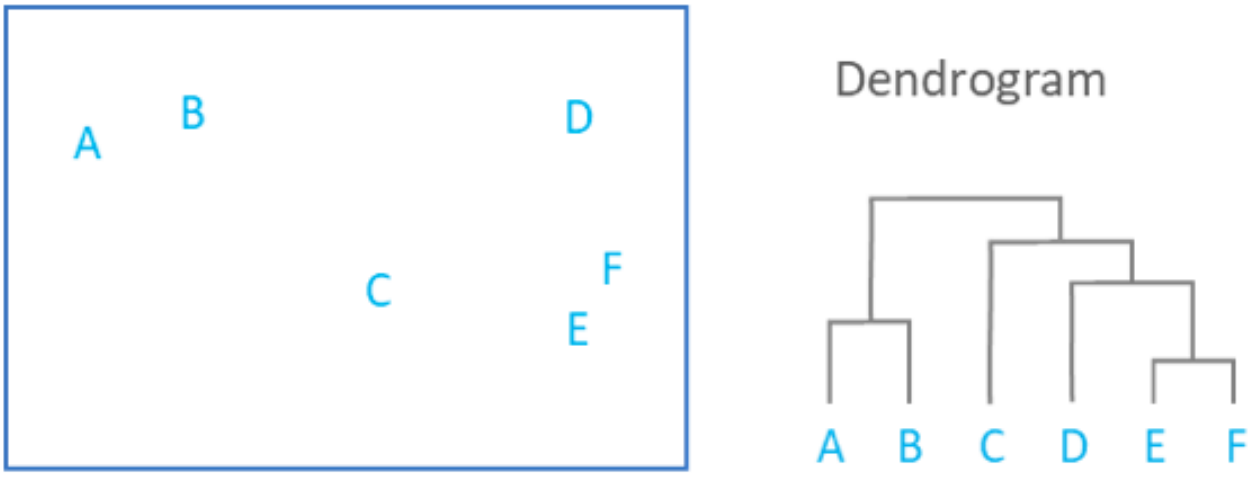
- Randomly assign all observations to one of the subgroups $1, ..., K$.
- Iterate until all cluster alignments stop changing:
  - For each of the K clusters, compute the cluster centroid - the mean of the p features within that cluster.
  - Assign each observation to the cluster with the closest centroid to that observation.

The algorithm is guaranteed to decrease to a local optimum however this is not guaranteed to be the global optimum - k-means clustering is dependent on the initial randomization step to

produce good (best) results! One can run the algorithm many times and select the result that has the lowest within-cluster variation over all runs. Another problem is selecting K. Once again, because its an unsupervised method there in general isn't an objective way to select the correct K. It could be domain dependent or could even require eyeballing the resultant clusters.

3.7.2 Hierarchical Clustering

Hierarchical clustering is a non-parametric approach to clustering that will produce all clusters from $1, ..., n$, as in, $K = 1$ (no clustering) to $K = n$ (each observation is its own cluster). Hierarchical clustering produces a **Dendrogram** which is a type of upside down tree diagram, as seen below.



It can be interpreted as each leaf being an observation and each fusion of two branches correspond to two observations/branches that are similar to each other. The height that these fusion occur corresponds to how similar they are. Using the image, We can say E and F are most similar. D is similar to E and F, but not as similar as A is to B, since it is higher up on the dendrogram. To derive clusters from a dendrogram we make a horizontal cut anywhere on the y-axis and the first set of fusions directly below the cut (so not fusions that are below another fusion) are our clusters. If our cut crosses two lines then that corresponds to $K = 2$ - we will have two clusters.

Hierarchical clustering assumes a hierarchical structure to the data - clusters obtained by cutting the dendrogram at a specific height are necessarily nested in clusters obtained by cutting the dendrogram at a greater height. E.g., a dataset of patients at a hospital who either do or do not have cancer and have type 1 or type 2 - with $K = 2$ we naturally split on has cancer and does not and on $K = 3$ we naturally split has cancer in to type 1 and type 2 - there is a hierarchy. Conversely, if we had a dataset of males and females from three different countries, splitting with $K = 2$ might yield male and female, but $K = 3$ would not split either male or female in to two new groups since nationality is not dependent on gender - there is no hierarchy in the data. When there is no hierarchy, hierarchical clustering can perform worse than k-means.

To build the dendrogram you first select a dissimilarity measure, e.g. Euclidean distance or correlation. The algorithm then treats each observation as its own cluster and finds the

cluster with the least dissimilarity (most similarity) and creates (fuses) a new cluster. The algorithm then iteratively repeats this process, treating the previous fusion as a new cluster and removing the two clusters that were fused in to it from the pool. It continues until there is only 1 cluster ($K = 1$) left. The dendrogram can be built from these fusions and uses how dissimilar the two clusters were when the fusion occurred to determine height.

The only problem here is determining the similarity between any object (single observation or a cluster) and a cluster. A cluster is made up of multiple observations so how do we determine similarity to a group. To do this there are 4 separate **linkage** schemes:

- **Complete**: Complete linkage is the maximal inter-cluster dissimilarity - compute all pairwise dissimilarities between all observations in both clusters and select the maximum.
- **Single**: Single linkage is the minimal inter-cluster dissimilarity - compute all pairwise dissimilarities between all observations and select the minimum. This can result in extended, trailing clusters in which single observations are fused one at a time (more emphasis put on observation-observation fusions than observation-cluster).
- **Average**: Average linkage is the mean inter-cluster dissimilarity - compute all pairwise dissimilarities between all observations and average them.
- **Centroid**: Centroid linkage is the dissimilarity between the centroid of each cluster. This can lead to undesirable **inversions** (where two clusters are fused at a height below either of the individual clusters).

In general, Average and complete are preferred over single linkage as they give more balanced dendrograms. Centroid is avoided due to the inversion issue but finds use in genomics.

Chapter 4

Assessing Performance

4.1 Measures

4.1.1 Statistical Errors

Standard Error

The standard error tells us how much, on average, our estimate for a statistic differs from the true value. This doesn't mean what the actual error is, but what the *expected* error is. For μ the standard error on $\hat{\mu}$ is given by:

$$Var(\hat{\mu}) = SE(\hat{\mu})^2 = \frac{\sigma^2}{n}$$

We can also calculate this for β_0 and β_1:

$$SE(\hat{\beta}_0)^2 = \sigma^2 \left[\frac{1}{n} + \frac{\bar{x}^2}{\sum_{i=1}^{n}(x_i - \bar{x}^2)^2} \right]$$

$$SE(\hat{\beta}_1)^2 = \frac{\sigma^2}{\sum_{i=1}^{n}(x_i - \bar{x}^2)^2}$$

For these to be strictly valid, we require that all residuals e_i are uncorrelated and all share a common variance σ^2.

Residual Standard Error

The residual standard error is essentially the sample deviation:

$$RSE = \hat{\sigma} = \sqrt{\frac{RSS}{n-2}}$$

It measures the 'lack of fit' of the model - high values indicate that the model does not fit well and as it approaches zero it indicates a good fit. The RSE implies that even if all parameters were known exactly, the random error ϵ would lead to an average deviation equal to the RSE. The RSE being non-normalized means that assessing performance is subjective. Additionally it is in the units of Y which may complicate matters further.

In general (for multiple regression) the RSE is:

$$RSE = \hat{\sigma} = \sqrt{\frac{RSS}{n-p-1}}$$

Confidence Intervals

A confidence interval is a range where we can say that true value of a parameter will lie within that range with a given probability. A 95% confidence interval would be an interval where we could say with 95% certainty that the true parameter value was within the boundaries. For simple linear regression, the confidence intervals are:

$$\hat{\beta}_1 \pm 2 \times SE(\hat{\beta}_1)$$
$$\hat{\beta}_0 \pm 2 \times SE(\hat{\beta}_0)$$

Prediction Intervals

Prediction intervals are similar to confidence intervals except they pertain to individual readings rather than the expected population value. They include both the reducible error (bias) and the irreducible error, which means they are larger than the corresponding confidence interval. They are still centred about the same point however (mean).

Hypothesis Testing

Hypothesis testing is a way to reject or accept certain beliefs about the model. You create a null hypothesis, or the hypothesis you assume to be true, and also an alternative hypothesis which you will accept if you can reject (prove false) the null hypothesis. In simple linear regression, we could have:

H_0: There is no relationship between X and Y, $H_0 : \beta_1 = 0$
H_a: There is some relationship between X and Y, $H_a : \beta_1 \neq 0$

We can then accept or reject the null hypothesis by testing whether our estimate for β_1, $\hat{\beta}_1$, is sufficiently far from 0. To determine what 'sufficiently' means we can use a **t-statistic**.

T-Statistic and P-Value

The t-statistic measures the number of standard deviations a parameter is away from some value. For the hypothesis test above, we want to test how far away $\hat{\beta}_1$ is from 0 which requires normalization as 'far from zero' is of course relative to how large the parameter actually is:

$$t = \frac{\hat{\beta}_1 - 0}{SE(\hat{\beta}_1)}$$

Generally:

$$t = \frac{\hat{\beta}_i - K}{SE(\hat{\beta}_i)}$$

For any parameter or statistic $\hat{\beta}_i$ where K can be any constant, usually for when a null hypothesis is of the form: $H_0 : \beta_i = K$.

From this we can calculate the probability of observing any number greater than or equal to $|t|$ assuming that the null hypothesis is correct by using the expected **t-distribution** for a correct null hypothesis (note, we use $|t|$ because this is a two-tailed test, if this was one tailed then we would treat t and $-t$ as different values). The resultant probability is called the **p-value**. If we observed a p-value of 0.01 (1%) we may decide to reject the null hypothesis in favour of the alternative hypothesis, thus stating that there is a high chance that there is a relationship.

Total Sum of Squares (TSS)

The total sum of squares (TSS) is a measure of the total variance in the response Y and can be thought of the amount of variability inherent in the response before any regression if performed. It is calculated as:

$$TSS = \sum (y_i - \bar{y})^2$$

R^2 Statistic

The R^2 statistic is a normalized statistic with a range of 0 to 1 with no units and acts as a performance metric for a regression. It is calculated as:

$$R^2 = \frac{TSS - RSS}{TSS}$$

The R^2 metric can be interpreted as the proportion of variability in Y that can be explained using X. As in, with TSS being the variability before regression and RSS being the variability

after it, R^2 is a measure for how much of the initial variability the model can account for (i.e. all Ys may be very spread but if given X values we can predict corresponding y values exactly with no variability, our model has entirely explained the relationship).

The R^2 statistic is highly related to correlation as it essentially measures the quality linear relationship between X and Y - if they are perfectly linear then, other than random error, the regression should be very accurate and thus explains a large amount of the initial variability. We find:

$$R^2 = r^2 = Corr(X, Y)^2$$

In a multiple linear regression setting, we find:

$$R^2 = Corr(Y, \hat{Y})^2$$

F-Statistic

For a multiple linear regression setting we want to perform the same hypothesis test where we check whether parameters are zero, except in this case we must check if they are all zero rather than just one. To do this we use a f-statistic:

$$F = \frac{\frac{TSS-RSS}{p}}{\frac{RSS}{n-p-1}}$$

If linear assumption is correct we find that

$$E[\frac{RSS}{n-p-1}] = \sigma^2$$

And if H_0 is correct

$$E[\frac{TSS-RSS}{p}] = \sigma^2$$

In both, n is the number of data points and p is the number of predictors. These give us an f-statistic of 1 if there is no linear relationship (H_0 is true). If H_a is correct in stead we expect $E[\frac{TSS-RSS}{p}] > \sigma^2$ and therefore the f-statistic will be greater than 1. Much like the t-statistic, one can construct an f-distribution in order to read off a **p-value** which we can use to determine significance.

It is also possible to compute an f-statistic between two models as opposed to one model vs no relationship at all. Suppose we want to compare a model with 10 predictors and the same model with q predictors removed (we want to test a less flexible model), we can calculate the f-statistic with:

$$F = \frac{\frac{RSS_0 - RSS}{q}}{\frac{RSS}{n-p-1}}$$

Where RSS_0 is the RSS for the model without the q predictors and RSS is the RSS value for the full model (all parameters including q). It turns out that the t-statistic and corresponding p-value for each individual predictor is related to the f-statistic for a model where RSS_0 is the model where all parameters exist except the one in question. In fact, the square of the t-statistic is equal to the f-statistic in that case. We still care about the overall f-statistic however as if you had 100 predictors with a 0.05 p-value cutoff and you calculated their significance (with a t-test) individually, you would find 5% of your predictors are statistically relevant by pure chance, even if they are not. The overall f-statistic does not fall victim to this as it scales with the number of predictors.

4.1.2 Response Errors

Residual Sum of Squares (RSS)

The residual sum of squares is simply the sum of the square residuals:

$$RSS = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

It resembles the mean squared error (MSE) without the normalization term. The RSS can be understood as the amount of variability that is left unexplained after performing the regression - it is a measure of the variability of the predicted response to the true response.

Mean Squared Error

Mean squared error is a error/loss measure used for continuous (regression) problems. It is defined as:

$$MSE = \frac{1}{n} \times \sum_{i=1}^{n} (y_i - \hat{f}(x_i))^2$$

Where $\hat{f}$ is our current estimate for f, n is our total number of data points and (x_i, y_i) is the predictor, response pair for the i^{th} data point. Mean squared error calculates the average of the squared difference between the true value y_i and the predicted value $\hat{f}(x_i)$ over all data points.

Brier Score

The Brier score is a error measure for discrete (classification) problems analogous to MSE for regression. It is defined as:

$$BS = \frac{1}{n} \times \sum_{i=1}^{n} (\hat{f}(x_i) - o_i)^2$$

Where $\hat{f}$ is our current estimate for f, n is our total number of data points, x_i is the predictors at the i^{th} data point and o_i is the 'outcome' (class label, binary true/false) at the i^{th} data point. Here, $\hat{f}(x_i)$ computes the predicted class/probability. This can be further simplified if $\hat{f}(x_i)$ and o_i are binary (or are set to binary) with:

$$BS = \frac{1}{n} \times \sum_{i=1}^{n} I(\hat{f}(x_i), o_i)$$

Where $I(x, y)$ is called an indicator variable as is 1 if $x = y$ and 0 otherwise. The simplified Brier score can be seen as the percentage of true classifications (true positive + true negative rate) while the full Brier score, if both predictor and response are continuous, generalizes to a form of MSE.

4.2 Diagnosing Performance

4.2.1 The Loss/Flexibility Plot

It is useful to plot training and test loss against flexibility. Flexibility here is a general concept but could specifically refer to a changing hyper parameter (number of neurons in a neural net). Across the entire domain you would expect the training performance to decrease below the irreducible error ϵ as it learns the noise (which is what ϵ is), while the test performance initially converges to an optimal value above the irreducible error and then diverges.

4.2.2 Determining Over/Under Fitting

If we plot our loss metric (such as MSE) for both the test and training set against whatever hyper parameter we wish to increase (or in general, plot against flexibility), any divergence suggests that, beyond the divergence, we have begun to overfit. Ideally we would select the point where performance on the test set is optimal (loss is at a minimum) and any point before this could be considered underfit. Additionally, if you knew what the irreducible error was (in general not possible) you could test for overfitting by whether the training loss is less than ϵ. To do so it must be learning the random noise in the data which is the hallmark of overfitting.

4.2.3 Residual Plots

A residual plot is simply a plot of each residual ($e_i = y_i - \hat{y}_i$). For a simple 1-predictor case this can be against the predictor, but in general it will be against $\hat{Y}$. A residual plot should have no pattern if our model is accurate. That is to say, the residuals should all be independent of each other and share a common variance and have a mean of zero. A perfectly uncorrelated residual plot would suggest that our model perfectly predicts the true population model, minus ϵ which is exactly what our residuals are. A pattern in the residual plot can suggest that some assumption made for our model is incorrect, usually a linear assumption for non-linearities. The shape of the residual plot can be useful itself as it can suggest what relationship does exist (a U-shape likely means a missing quadratic term) although for complex relationships this is unreliable.

Studentized Residual Plots

An extension to residual plots, these simply plot the same as residual plots but each residual is divided by its estimated standard error. This acts as a normalization step and makes spotting outliers or otherwise unexpected data points much easier.

4.2.4 The Leverage Statistic

To diagnose high leverage points we can use a leverage statistic. For simple linear regression this takes the form:

$$h_i = \frac{1}{n} + \frac{(x_i - \bar{x})^2}{\sum_{i'=1}^{n}(x_{i'} - \bar{x})^2}$$

The leverage statistic is always between $\frac{1}{n}$ and 1 and the average leverage statistic for all observations is $\frac{p+1}{n}$, if a given observation has a leverage statistic far larger than this we can likely conclude it is a high leverage point.

4.2.5 The Variance Inflation Factor (VIF)

For detection of multi-collinearity we can compute the VIF which is the ratio of the variance of $\hat{\beta}_j$ when fitting the full model divided by the variance of $\hat{\beta}_j$ if fit on its own. I.e. find the variance of $\hat{\beta}_j$ in both $y = \beta_0 + \beta_j x_j$ and $y = \beta_0 + ... + \beta_j x_j + ...$ The smallest value of the VIF is 1 and indicates no collinearity.

It can also be calculated with:

$$VIF(\hat{\beta}_j) = \frac{1}{1 - R^2_{X_j|X_{-j}}}$$

Where $R^2_{X_j|X_{-j}}$ is the R^2 statistic from a regression of X_j onto all of the other predictors.

4.2.6 Confusion Matrix

Used for binary classification problems (or anything that can be mapped to binary).

A confusion matrix is a matrix of true/false negative/positive results for a binary problem (any problem can be represented as a binary problem by looking at it as $class = k$ or $class \neq k$ for any number of classes). A **false positive** is when the algorithm incorrectly classifies a data point as a member of the target class, when it should be a different class. A **false negative** is when the algorithm incorrectly classifies a data point as *not* a member of the target class, when it should be part of the target class. A **true positive** or **true negative** is the opposite - where the algorithm correctly assigns the data point as a member or not member of the target class respectively. A confusion matrix is important as it allows you to detect when the algorithm is not performing in a specific way, e.g. if in a data set 99% of all data points are members of class k, we can obtain 99% accuracy by classifying all data points as class k. If our job is to identify data points that are not class k then we have completely failed our task despite our error measurements suggesting we are doing well - a confusion matrix (specifically the true positive rate/false negative rate: calculating the ratio of these two measures will tell us our true percentage classification of $class \neq k$) can be used to identify this.

Other names for these measures are the **sensitivity** and **specificity**. The **sensitivity** is the percentage of true positives recorded (true 'target class') or in other words, the ratio of correctly classified positive data points out of all positive data points ($100 \times \frac{TP}{TP+FN}$). The **specificity** is the opposite - the percentage of true negatives recorded (true not 'target class'), or the ratio of correctly classified negative data points out of all negative data points ($100 \times \frac{TN}{TN+FP}$).

4.2.7 AUC/ROC Curves

Used for binary classification problems (or anything that can be mapped to binary).

A **ROC Curve** or a **receiver operating characteristic curve** displays the true positive rate (sensitivity) against the false positive rate (1-specificity) across all possible thresholds (the value you use to assign classes, e.g. set observation x equal to class k if $P(class = k|x) > threshold$). This is achieved as, by setting the threshold, you can obtain any true positive rate (e.g. a threshold of 1 will obtain a perfect true positive rate at the expense of overall accuracy) and a corresponding false positive rate. By scanning over all thresholds you can plot each true positive rate against the corresponding false positive rate.

The performance of a classifier can be obtained by looking at the **AUC** or **area under the (ROC) curve**. A perfect classifier would hug the top left corner s.t. the area under the curve is maximized (as in, a right angled line with 0 false positive rate and 1 true positive rate across all thresholds). A classifier with a straight line where the true positive and false positive rates are equal is known as a 'no information' classifier and is equivalent to a random guess - in such a classifier the AUC would be 0.5, with a perfect classifier having an AUC of 1.

4.3 Re-sampling Methods

When scoring our model we want an unbiased estimate of the performance of our model on the entire population. That is to say, we want to estimate how well the algorithm would have performed if trained on the entire population. This is important as re-sampling methods aren't just for scoring model performance, although that is the first step. We wish to take this score and use it to select the best algorithm out of a collection of algorithms, or do a hyper parameter search, or estimate the error on some model parameter. For all of this we need an accurate measure of error (relative to the population error).

Ideally we would have a training set *and* a test set when we are building our model. We assume that both sets are representative of the population but are independently drawn, this ensures that when we test our model performance on the test set we are testing only on the underlying relationships in the population and not on the noise found in the training set - if the two sets were correlated, for example if we tested our model on the training set, then we run the risk that our metrics will underestimate the true error because the model better fits the random noise. If we satisfy this condition then we know that performance on the test set is an unbiased and accurate estimator for performance on the population. Most of the time however we won't have access to a test set and we have to devise some way to obtain one.

4.3.1 Cross-Validation

Building a **validation** (**hold-out set**) set and scoring the algorithm on that can take the place of a dedicated test set. A validation set is built by randomly splitting the training data in to two distinct, non-overlapping sets and using one as the new training set and one as the validation set. These sets are usually equal in size. While this method does solve the issue of a correlated test set, it introduces other issues which could work to make our analysis even worse:

- The random selection of data points for each set could lead to a point where there is not enough data in one or both of the sets to represent the relationship, for example if our relationship followed a sine curve and our random selection was such that one of the sets only had data points at each period. The net effect is that our analysis is heavily dependent on the initial random selection step.

- The accuracy of most algorithms scales with data set size and therefore splitting our data set will naturally reduce accuracy. In particular, we may be led to believe that our algorithm choice performs poorly for our dataset, perhaps suggesting we have made the wrong assumptions, while if trained on the full data set it may have performed well. In this case, the validation set error will *overestimate* the population error for a particular algorithm

Both of these things mean that we cannot guarantee that performance on the validation set will be a good estimator of performance on the population which at the end of the day is the only reason we even want to construct one.

k-fold Cross-Validation

K-fold cross-validation aims to solve this problem by reducing the size of the split in to k random, disjoint sets of equal size. The sets are then recombined with $k-1$ of them making the training set and the last set acting as the validation (test) set. Using these sets we can solve both problems. By increasing the size of the training set from $0.5\times$ to $1-\frac{1}{k}\times$ we allow the algorithm to train on as many examples as possible (while retaining the benefits of having a validation set) while reducing the chance of the training set not being representative. There is an issue though, now the validation set is very small. The second step to k-fold CV is to repeat this process k times and on each iteration, use a different set (from the initially generated k sets) for the validation set, with the training set being made up of all sets that weren't used for validation in that iteration. We can then obtain an error estimate that is resistant to the randomization procedure (as we will train and validate on all data points at some point) and in effect, doesn't reduce the size of the training set:

$$MSE_{overall} = \frac{1}{k}\sum_{i=1}^{k} MSE_k$$

There is a special case of k-fold CV called **leave-one-out cross validation (LOOCV)** where $k = n$, where n is the size of the data set - our validation set is 1 data point. LOOCV removes the randomization element entirely and essentially maximizes training set size (if $n-1 << n$ then your dataset was probably too small in the first place) thus avoiding all the issues inherent in CV while still having an accurate error estimate. The downside is in the computational complexity scales with k and setting $k = n$ means there is no value of k that costs more (for that dataset). There is a special case with linear least-squares regression where we have a computationally efficient trick for calculating $MSE_{overall}$:

$$MSE_{overall} = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{y_i - \bar{y}_i}{1-h_i}\right)^2$$

Where h_i is the leverage and $k = n$.

LOOCV is not strictly better in all scenarios however. By maximizing the size of the training set, LOOCV minimizes bias as much as possible without using the full training set and does so considerably more than $k < n$-fold CV. As is usual with reducing bias, LOOCV increases variance on the estimate. Since LOOCV averages n different, high (positively) correlated models (they all contain almost the same data minus one point) and we know that the variance of a collection of values increases with their (positive) correlation, thus the LOOCV average must have higher variance than a $k < n$ average since a dataset of $n-1$ is more correlated with itself than $n-k$.

This statement arises out of the idea of 'cancelling out'. If you calculate the variance of positively correlated quantities - which these are since the dataset containing $[1,...,n-1]$ *is almost identical to that of* $[2,...,n]$ *and thus they will produce very similar MSE values, or in other words, positively correlated - it will tend to be higher because mathematically variance is dependent on the covariance between data points which is high. Contrarily in* $k < n$ *the data points are less correlated since there is less overlap in the training sets. Intuitively it can be understood as, if our* $n-1$ *training set has a high mean estimate then its likely that all other* $n-1$ *sets will (positive*

correlation) and so there is nothing to 'cancel it out' whereas with $n-k$ datasets there is more chance of getting a low mean despite other values having high mean which cancels out the effect of the high mean.

The important point though is the variability of the chosen model on an unseen dataset. It is an example of overfitting. Since your training set effectively doesn't change, you will select the best performing algorithm/parameter based on which test data best matches the noise in the training data which is a type of overfitting. In $k < n$-fold the training set changes more noticeably and thus the noise does too, meaning you tend to select for the best performing algorithm overall, not just based on training set noise.

4.3.2 Bootstrap

Bootstrap can be used to estimate the uncertainty for a parameter or algorithm. Ideally, to quantify the uncertainty of a measurement we would draw multiple independent samples (multiple test sets) from the population and recalculate (or rescore in the case of an algorithm) the parameter on each sample, using the set of parameter estimates to estimate the uncertainty. In the real world though we can't generate multiple independent samples since we don't have the underlying population or its generation rules. Bootstrap is a method for generating these samples using the dataset we already have. For a set of size n, bootstrap involves sampling n (X, Y) pairs from the dataset at random while allowing duplicates (**'with replacement'**) - we may have duplicates of some data points while missing other data points entirely. We would do this i times to generate 'bootstrapped' sample sets, B_i. We can then estimate the variance of our parameter, ψ by calculating the parameter, $\hat{\psi}_i^*$, for each dataset B_i and finally estimating the uncertainty with:

$$SE(\hat{\psi}) = \sqrt{\frac{1}{i-1}\sum_{r=1}^{i}\left(\hat{\psi}_r^* - \frac{1}{i}\sum_{r'=1}^{i}\hat{\psi}_{r'}^*\right)^2}$$

Where $\hat{\psi}$ is our estimate for ψ.

4.4 Estimating Test-Set Error

Training set error can almost invariably be reduced by increasing the flexibility of the model. In a linear regression setting, this could mean increasing the number of predictors endlessly. As such, when we are doing feature selection we cannot rely on grading different models using error metrics that test performance on the training set as this will always select for larger models. Re-sampling methods, as detailed above, are one method for estimating performance on the test set. Another method is **adjusting the training set error** to take in to account overfitting bias.

It should be noted, the below measures are for least squares linear models. They can be defined for more general models if required.

4.4.1 C_p

For a least squares model containing d predictors, the C_p estimate of the test MSE is:

$$C_p = \frac{1}{n}(RSS + 2d\hat{\sigma}^2)$$

Where $\hat{\sigma}^2$ is an estimate of the variance of the error η, where this variance is typically estimated on the model containing all predictors.

4.4.2 Akaike Information Criterion (AIC)

AIC is defined for models fit by maximum likelihood.

$$AIC = \frac{1}{n\hat{\sigma}^2}(RSS + 2d\hat{\sigma}^2)$$

For least squares (which is equivalent to maximum likelihood with Gaussian errors) this is proportional to C_p.

4.4.3 Bayesian Information Criterion (BIC)

$$BIC = \frac{1}{n\hat{\sigma}^2}(RSS + log(n)d\hat{\sigma}^2)$$

BIC penalizes larger models more than C_p or AIC as usually $log(n) > 2$ (specifically for $n > 7$).

4.4.4 Adjusted R^2

The adjusted R^2 model, as the name implies, attempts to adjust the usual R^2 metric to account for increasing model size:

$$AdjustedR^2 = 1 - \frac{\frac{RSS}{(n-d-1)}}{\frac{TSS}{(n-1)}}$$

The intuition here is that for RSS by itself, adding noise variables will still reduce RSS but only very slightly. The adjusted R^2 therefore compensates for this very small increase for a considerably larger decrease as d increases in $\frac{RSS}{(n-d-1)}$.