Machine Learning

Matthew Chen

September 8, 2025

Contents

1 Preface

Machine learning is the field of using statistical algorithms to learn patterns from data and generalize those learned patterns to unseen data. I provide a brief overview of commonly used machine learning techniques and their different uses in pattern recognition, data mining, and artificial intelligence. Focus in these notes will be put towards the understanding of key concepts in Machine Learning and not on formal mathematical results or implementations in computer code. The second edition of Elements of Statistical Learning by Trevor Hastie, Robert Tibshirani, and Rerome Friedman will be a key reference.

1.1 Supervised Learning

The majority of these notes are dedicated to *supervised learning*, where in a typical scenario we build a model that uses some input data (features) is used to predict some other variable of interest (target/response/outcome). We have *training data*, which are observed sets of both features and targets in which we can build our model (the name supervised learning comes from having observed the targets that help guide the model training). The goal is to build a model that can successfully predict the outcome on new unseen data examples.

Unsupervised learning, which is less commonly discussed in the machine learning literature, refers to learning the structure and organization of the data based only on features and no targets. This is briefly introduced towards the end.

1.1.1 Empirical Risk

Consider a regression task where the random vector $x \in \mathbb{R}^p$ is used to predict the response variable y. The elements of x represent different predictors (also known as *features*), and y represents the target variable being predicted. We can formulate a *loss function*, denoted as $\ell(\theta; x, y)$, as a measure of how close the predicted value is to the true y, where θ is the set of model parameters.

We define Statistical Risk as the expected loss over the random input x and y:

$$R(\theta) = \mathbb{E}_{x,y}\ell(\theta; x, y)) = \int_{\mathbb{R}} p(x, y)\ell(\theta; x, y)\, dxdy$$

However, the true Risk is unknown since the joint distribution of the data $p(x, y)$ is unknown. Instead, risk is estimated using the observed dataset, taking the average of n individual losses corresponding to n training observations (also known as training *examples*). Denoting the observed data as $\{x_i, y_i\}_{i=1}^n$ we define the Empirical Risk as a Monte-Carlo estimate of the true Risk

$$R_n(\theta) = \frac{1}{n}\sum_{i=1}^{n} \ell(\theta; x_i, y_i)$$

Note that by the Law of Large Numbers, the Empirical Risk converges in probability to the true Risk. The estimation of the model parameters θ then, can be formulated as an optimization problem involving the Empirical Risk. This class of estimators are known as the Empirical Risk Minimizers (ERM).

$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}}\ R_n(\theta)$$

Unfortunately, in expectation, the empirical risk under the ERM is not equivalent to the parameter that minimizes the true risk. The intuition is that we tune the model on observed data that also includes some level of noise, which prevents us from learning the true relationship. This is the basic theoretical idea of *overfitting*.

$$\mathbb{E}R_n(\hat{\theta}) \neq \min_{\theta} R(\theta)$$

In practice, this problem is resolved by checking our model on out-of-sample data, i.e. in cross validation or using a train/test split. That is, we can split the data into two different components, a training set and a test set. Then we can estimate the parameters using the training data and validate the model's out-of-sample performance on the test data. In practice, the test error from a single train/test split has high variance, i.e., if we simply split the data differently the results would be different. We can reduce this variance by averaging errors systematically from different train/test partitions of the data. For example, in k-fold cross validation, the data is split equally into k partitions, and k models are trained. For each model, one of the data partitions is set aside and the rest are used to train the model. The errors from each of the k folds are averaged to stabilize the result, at the cost of training k models.

A brief note about notation throughout: often, the observed dataset $\{x_i, y_i\}_{i=1}^n$ is represented by the design matrix $\mathbf{X} \in \mathbb{R}^{nxp}$ where its rows are the n observation vectors of x_i, and $y \in \mathbb{R}^n$ is the corresponding response vector.

$$X = \begin{bmatrix} — & x_1 & — \\ \vdots & \vdots & \vdots \\ — & x_n & — \end{bmatrix} \quad y = \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix}$$

1.1.2 Bias-Variance Tradeoff

Consider a model $f(x)$ that represents the true data generating process

$$y = f(x) + \epsilon$$

where ϵ is some zero-mean error/noise term with constant variance σ^2. Our goal is to find an approximate function $\hat{f}$ that approximates the true function f as well as possible. This can be quantified by the MSPE (mean square prediction error) of $\hat{y}$, which is the expected square difference between predicted value of a never seen data sample x and its corresponding true y (that is, the sample was never used in training). Note that this is a theoretical quantity since we cannot compute MSPE as the true y is not known on our unseen sample. We define

$$MSPE := \mathbb{E}\left[(y - \hat{f}(x))^2\right]$$

MSPE can be decomposed into a variance term, a bias term, and an irreducible error term using simple algebra.

$$\begin{aligned} MSPE(\hat{f}(x)) &= \mathbb{E}\left[(y - \hat{f}(x))^2\right] = \mathbb{E}\left[(f(x) + \epsilon - \hat{f}(x))^2\right] \\ &= \mathbb{E}[(\hat{f}(x) - \mathbb{E}\hat{f}(x))^2] + \left(\mathbb{E}[f(x) - \hat{f}(x)]\right)^2 + \sigma_\epsilon^2 \\ &= Variance(\hat{f}(x)) + Bias(\hat{f}(x))^2 + \sigma_\epsilon^2 \end{aligned}$$

Intuitively, models that are too large and complex for the given task tend to have greater variance. That is, if we draw new data from the same underlying distribution as our original data and retrain the model, the estimates of the model parameters may vary significantly from the first model. Intuitively, this is because the first model had enough complexity to even learn the noise in the first dataset, so that the model performs poorly when new data is drawn with a new instance of observed noise. This is the *overfit* regime - where the training performance is much better than testing. Conversely, a model with too little complexity may not be able to capture the true data generating process and therefore have high bias, i.e. on average (in expectation), the model does poorly. This is the *underfit* regime. The best models balance the first and the second scenarios, and achieves the so-called bias-variance tradeoff. See Figure 1.

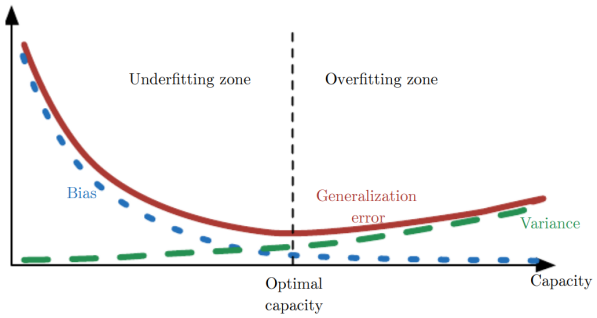

Figure 1: The Bias Variance Tradeoff. *Image Source: Deep Learning by Ian Goodfellow*

1.2 Loss Functions

In this section, I highlight popular loss functions: cross entropy loss for classification problems and square error loss for regression problems, and their connection to Maximum Likelihood Estimation (MLE).

1.2.1 Cross Entropy Loss

Consider a classification problem with $k = 1, ..., K$ classes and the cross-entropy loss function.

$$\ell(y_i, x_i; \theta) = -\sum_{k=1}^{K} y_{ik} log(f_k(x_i)), \ i = 1, ..., n$$

where y_i is the true class for the ith training observation, x_i is the corresponding input vector, θ is the parameter set of the model. Further, we write y_{ik} as shorthand for $\mathbf{1}\{Y_i = k\}$ and $f_k(x_i)$ represents the modeled probability of class k, $\mathbb{P}\{Y_i = k|X = x_i; \theta\}$. Combining all n losses, we have the empirical risk minimizer

$$\hat{\theta} = \underset{\theta}{\text{argmin}} R_n(\theta) = \underset{\theta}{\text{argmin}} -\sum_{i=1}^{n}\sum_{k=1}^{K} y_{ik} log(f_k(x_i))$$

Aside, let's write out the maximum likelihood estimator for θ. First, we write the joint likelihood, which is the joint distribution evaluated using the observed data. In maximum likelihood estimation (MLE) in statistics, we model the data in such a way where the joint likelihood is maximized.

$$\hat{\theta}_{MLE} = \underset{\theta}{\text{argmax}} \ \mathbb{P}\left(Y_1 = y_1, ...Y_n = y_n | X_1 = x_1, ..., X_n = x_n; \theta\right)$$

Assuming independence between observations, we have

$$\hat{\theta}_{MLE} = \underset{\theta}{\text{argmax}} \prod_{i=1}^{n} \mathbb{P}(Y_i = y_i | X = x_i; \theta) = \underset{\theta}{\text{argmax}} \prod_{i=1}^{n}\prod_{k=1}^{K} \mathbb{P}(Y_i = y_i | X = x_i; \theta)^{\mathbf{1}\{Y_i = k\}}$$

Rewriting the indicator using the notation defined earlier, and evaluating for the modeled probability

$$\hat{\theta}_{MLE} = \underset{\theta}{\text{argmax}} \prod_{i=1}^{n}\prod_{k=1}^{K} \mathbb{P}(Y_i = y_i | X = x_i; \theta)^{y_{ik}} = \underset{\theta}{\text{argmax}} \prod_{i=1}^{n}\prod_{k=1}^{K} f_k(x_i; \theta)^{y_{ik}}$$

Finally, taking the log-likelihood

$$\hat{\theta} = \underset{\theta}{\text{argmax}} \sum_{i=1}^{n}\sum_{k=1}^{K} y_{ik} log(f_k(x_i)) = \underset{\theta}{\text{argmin}} -\sum_{i=1}^{n}\sum_{k=1}^{K} y_{ik} log(f_k(x_i))$$

Thus, fitting the parameters by minimizing the cross entropy loss is the same as minimizing the negative log-likelihood of our observed data using a categorical distribution.

Cross-entropy comes from information theory, and measures the distance between two distributions p and q. We can think of p as the true distribution and q as the computed distribution. Then the cross-entropy $H(p,q)$ is

$$H(p,q) = -\mathbb{E}_p[log(q)] = -\sum_{k \in \chi} p(k) log(q(k))$$

where χ is the support of p and q.

For intuition, consider a ground truth distribution p that has probability 1 at the true class and 0 elsewhere. For example, if $y_i = 3$, then $p = [0, 0, 1, 0, 0]$ for a support of $K = 5$ possible classes, and let q be the computed probability, e.g. a softmax output from a neural network, be $q = [0.01, 0.09, 0.8, 0.05, 0.05]$. Then the goal is to have the computed probability be as close as possible to the ground truth.

1.2.2 Square Error Loss

Above, we saw the connection between cross entropy and the MLE for classification problems. It turns out, a similar connection exists for square error loss and regression problems for Gaussian distributed data.

The empirical risk with square error loss is as follows

$$R_n(\theta) = \frac{1}{n} \sum_{i=1}^{n} (y_i - f(x_i; \theta))^2$$

where y_i is the value of the ith training example, and $f(x_i; \theta)$ is the corresponding predicted value from the model. Consider modeling the response variable Y with a Gaussian distribution, with mean $f(X)$ and constant variance σ^2.

$$Y|X \sim \mathcal{N}(f(X; \theta), \sigma^2)$$

Let $\phi(f(x_i), \sigma^2)$ denote the normal probability density function with mean $f(x_i)$ and variance σ^2. Writing out the joint likelihood of the data, assuming independence between observations, we have

$$L\left(y_1, ..., y_n | x_1, ..., x_n; \theta\right) = \prod_{i=1}^{n} \phi(y_i | f(x_i; \theta), \sigma^2) = \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi\sigma^2}} exp\left(-\frac{1}{2\sigma^2}(y_i - f(x_i; \theta)^2\right)$$

Taking the log of the expression, we arrive at the log-likelihood

$$\ell(\theta) = \sum_{i=1}^{n} log(\frac{1}{\sqrt{2\pi\sigma^2}}) - \frac{1}{2\sigma^2}(y_i - f(x_i; \theta))^2$$

Writing out the maximum likelihood estimator

$$\hat{\theta}_{MLE} = \underset{\theta}{\text{argmax}} \sum_{i=1}^{n} log(\frac{1}{\sqrt{2\pi\sigma^2}}) - \frac{1}{2\sigma^2}(y_i - f(x_i; \theta))^2$$

Removing terms that do not depend on θ

$$\hat{\theta}_{MLE} = \underset{\theta}{\text{argmax}} \sum_{i=1}^{n} -(y_i - f(x_i; \theta))^2 = \underset{\theta}{\text{argmin}} \sum_{i=1}^{n} (y_i - f(x_i; \theta))^2$$

We see that minimizing the square error loss also gives the MLE estimate for Gaussian data.

2 Linear Regression

2.1 The Ordinary Least Squares

In linear regression, the model $f(x)$ takes on a linear form. Consider a model with p coefficients, so that $x_i \in \mathbb{R}^p$ with corresponding response variable y_i. An important nuance here is that the first element of x_i has $x_{i0} = 1$, which corresponds to the "intercept" of the model. Let $\beta \in \mathbb{R}^p$ be a vector of model coefficients, and let ϵ be a zero-mean constant variance random error terms so that we have

$$y_i = f(x_i) + \epsilon_i = \beta^T x_i + \epsilon_i$$

In matrix form, considering all n data observations, we can write

$$y = X\beta + \epsilon$$

To fit the model, we minimize the square error deviations between the predicted values, denoted as $\hat{y}$, and the observed ground truth y.

$$\hat{\beta}_{OLS} = \underset{\beta}{\text{argmin}} ||y - X\beta||_2^2$$

An analytical solution to the OLS problem can be written by taking the gradient of the least-squares objective and setting it to zero. When a multivariate function is composite of an affine function, i.e. $g(x) = f(Ax+b)$, we can use the Jacobian chain rule to obtain

$$\nabla_x g(x) = A^T \nabla_{Ax+b} f(Ax + b)$$

Applying this to the least squares criterion, we have that

$$\nabla_\beta \ell(\beta) = -X^T \nabla_{y-X\beta} ||y - X\beta||_2^2 = -2X^T(y - X\beta) = 0$$

so we have that

$$X^T y = X^T X\beta$$

so finally

$$\hat{\beta}_{OLS} = (X^T X)^{-1} X^T y$$

Note that for a solution to exist, $X^T X \in \mathbb{R}^{p,p}$ must be invertible. That is, it must be full rank, $Rank(X^T X) = p$ which requires $Rank(X) = p$. This requires that $n \geq p$ (if $n < p$, $Rank(X) \leq n < p$ which would make $X^T X$ rank deficient and invertible).

We can evaluate for the predicted values using

$$\hat{y} = X\hat{\beta}_{OLS} = X(X^T X)^{-1} X^T y = Hy$$

$H = X(X^T X)^{-1} X^T$ is known as the *hat matrix*, which has the form of a projection matrix to the column space of the design matrix X. This geometric interpretation is very natural to linear regression: we are finding a member of $R(X)$ (R here denotes the range, or column space), i.e. a linear combination of the columns of X, such that the distance to the ground truth y is as small as possible.

Linear regression has several classic assumptions:

- The relationship between y and x is linear (linearity assumption)
- The error terms are uncorrelated with each other
- The error terms have zero mean and equal variance (no heteroskedasticity)
- The error terms are normally distributed (normality assumption)

The last 3 assumptions can be summarized as $\epsilon \sim \mathcal{N}(0, \sigma^2 I)$ and are essential for statistical inference and testing of the model results. The Gaussian/normality assumption, however, is not needed if the goal is purely to build a predictive model and not use it for statistical inference.

We can decompose the total variance (total sum of squares, SSTO) into an explained variance part (regression of sum of squares, SSR) and an unexplained variance part (error sum of squares, SSE). This is called the sum of squares decomposition.

$$\sum_{i=1}^{n}(y_i - \bar{y})^2 = \sum_{i=1}^{n}(y_i - \hat{y_i})^2 + \sum_{i=1}^{n}(\hat{y_i} - \bar{y})^2$$
$$SSTO = SSE + SSR$$

As a measure of goodness-of-fit, we can use the coefficient of determination, R^2, to quantify the proportion of variance that the model "explains."

$$R^2 = \frac{SSR}{SSTO} = 1 - \frac{SSE}{SSTO}$$

If the Gaussian assumption is true, we can also use the sum of squares to test the statistical significance of the model fit. Noting that $SSTO$ has degrees of freedom $n-1$, SSE has degrees of freedom $n-p$, and SSR has degrees of freedom $p-1$, we can define the F statistic as

$$F^* = \frac{\frac{SSR}{p-1}}{\frac{SSE}{n-p}} = \frac{MSR}{MSE}$$

Where we test the hypothesis

$$H_0 : \beta = 0$$
$$H_1 : \beta \neq 0$$

Under the null hypothesis, the F-statistic follows a F-distribution with degrees of freedom $p-1$ and $n-p$, respectively. The p-value of the test measures how extreme our observed F^* statistic is under the null distribution. If it is too extreme then we have stronger evidence *against* the null hypothesis.

A common issue that arises in practice is the issue of multicollinearity; i.e., when different predictor variables are inter-correlated with each other. If this is the case, the regression coefficients represent the incremental contribution of that variable *given the other variables already in the model*. A new variable may not add much to the model if it is highly correlated with the existing predictor variables since most of its information is already in the model. Importantly, the presence of colinearity inflates the variance of the model, highlighting the importance of variable/feature selection to achieve the bias-variance tradeoff. We can also reduce variance by penalizing the size (the norm) of the coefficient vector (such as in Ridge and Lasso regression). This is called *regularization*. An added benefit of regularization is that it also provides solutions in the high-dimensional case of $p > n$ where the OLS solution does not exist.

2.2 Ridge Regression

The ridge solution regularizes the OLS by solving a penalized version of the residual sum of squares, using the Euclidean (L2) norm of the parameter vector. Overall, this penalty shrinks the size of the coefficients (toward 0) and trades some bias for reduced variance. Defining λ as a shrinkage parameter, we can formulate the problem as the following

$$\hat{\beta}_{ridge} = \underset{\beta}{\text{argmin}} ||y - X\beta||_2^2 + \lambda||\beta||_2^2 = \underset{\beta}{\text{argmin}} (y - X\beta)^T(y - X\beta) + \lambda\beta^T\beta$$

Solving for $\hat{\beta}_{ridge}$ we obtain

$$\hat{\beta}_{ridge} = (X^TX + \lambda I)^{-1}X^Ty$$

As a sanity check, note that if $\lambda = 0$, the result is the same as the ordinary least squares solution.

The mathematical interpretation of ridge regression becomes clear if we apply a thin singular value decomposition to X. Recall that the singular value decomposition (SVD) generalizes the eigenvalue/eigenvector decomposition to any rectangular matrix (rather than just square matrices with n independent eigenvectors), so that we can decompose the data matrix as $X = UDV^T$.

Here we briefly review the SVD. Suppose $X \in \mathbb{R}^{nxp}$. Here, the columns of V contain the p eigenvectors of the symmetric and square matrix $X^TX \in \mathbb{R}^{pxp}$ so that $V \in \mathbb{R}^{pxp}$. Recall that these eigenvectors span all of $\mathbb{R}^p$ and are mutually orthogonal (linearly independent). That is, the columns of V, known as the right singular vectors, form an orthogonal basis for $\mathbb{R}^p$.

The singular values are the square-root of the eigenvalues of X^TX, which we can denote $\sigma_i = \sqrt{\lambda_i}$. Suppose X is Rank r and therefore has r non-zero singular values (recall that out of the p real eigenvalues, the zero eigenvalues correspond to the null space). The columns of U, which are the left singular vectors, contain the normalized products of X and the right singular vectors v_i. Note that we will have r non-zero vectors of such products.

$$\{u_1, .., u_r\} = \left\{ \frac{Xv_1}{||Xv_1||}, .., \frac{Xv_r}{||Xv_r||} \right\}$$

Since

$$||Xv_i||^2 = (Xv_i)^T(Xv_i) = v_i^TX^TXv_i = \lambda_i v_i^Tv_i = \lambda_i$$

Then $||Xv_i|| = \sigma_i$, and we have

$$\{u_1, .., u_r\} = \left\{ \frac{Xv_1}{\sigma_1}, .., \frac{Xv_r}{\sigma_r} \right\}$$

We can show that the left singular vectors are orthogonal to each other and span the column space of X. For example, for $i \neq j$ we have

$$u_i^Tu_j = (Xv_i)^T(Xv_j) = v_i^TX^TXv_j = \lambda_j v_i^Tv_j = 0$$

and $u_1, ..., u_r$ are r linearly independent vectors in $R(X)$.

In standard SVD, we extend the nonzero vectors above $\{u_1, .., u_r\}$ to $\{u_1, .., u_n\}$ to be a orthonormal basis for $\mathbb{R}^n$ so that $U \in \mathbb{R}^{nxn}$. We define a block matrix $\Sigma \in \mathbb{R}^{nxp}$ where the top right block is the matrix $D = diag(\sigma_1, ..., \sigma_r)$ containing the r non-zero singular values σ of X, and all other elements are zero. Then we can show that $X = U\Sigma V^T$. Thin SVD is very similar, but instead of extending the columns of U and building a block matrix for Σ, we keep only the non-zero singular values and the corresponding left and right singular vectors. Then we have $U \in \mathbb{R}^{nxr}$, $D \in \mathbb{R}^{rxr}$, and $V \in \mathbb{R}^{pxr}$ where U and V are semi-orthogonal matrices (since they are not square) satisfying $U^TU = V^TV = I$.

Then we have that

$$XV = (Xv_1, ..., Xv_r) = (\sigma_1 u_1, ..., \sigma_r u_r) = UD$$

Since V is a orthogonal matrix, we have

$$X = XVV^T = UDV^T$$

Going back to Ridge regression: if we evaluate for $X = UDV^T$ in the fitted values then we have:

$$\hat{y} = X\hat{\beta}_{ridge} = (X^TX + \lambda I)^{-1}X^Ty = UD(D^2 + \lambda I)^{-1}DU^Ty = \sum_{j=1}^{p} u_j \frac{\sigma_j^2}{\sigma_j^2 + \lambda} u_j^Ty$$

By convention, we arrange the singular values in decreasing order so that $\sigma_1 \geq \sigma_2 \geq ... \geq \sigma_p$. Further, we can write the sample covariance matrix of X as

$$S_x = \frac{1}{n}X^TX$$

where the sample variance of the ith principal component (PCA is discussed in more detail later) is

$$Var(PC_i) = \frac{\sigma_i^2}{n}$$

Noting that the shrinkage term $\frac{\sigma_j^2}{\sigma_j^2+\lambda}$ is closer to 1 for larger σ_j^2, we see that the ridge solution shrinks the directions (i.e. features, columns of X) corresponding to the *principal components with the least variance the most.*

2.3 Lasso Regression

The lasso solution is similar to the ridge, except it employs an L1 penalty on the coefficients instead of an L2 penalty.

$$\hat{\beta}_{lasso} = \underset{\beta}{\text{argmin}} \sum_{i=1}^{n} (y_i - \beta^T x_i)^2 + \lambda \sum_{j=1}^{p} |\beta_j|$$

By the nature of this constraint, lasso sets coefficients exactly to zero, and can be seen as a type of continuous variable selection. That is, in LASSO we make an assumption of sparsity. Specifically, the ridge solution does a proportional shrinkage while the lasso shifts coefficients by λ and then truncates at zero (see Figure 2 below). This is called soft-thresholding. An advantage of lasso is that it does variable selection and provides estimates simultaneously, although the estimates are biased by λ.

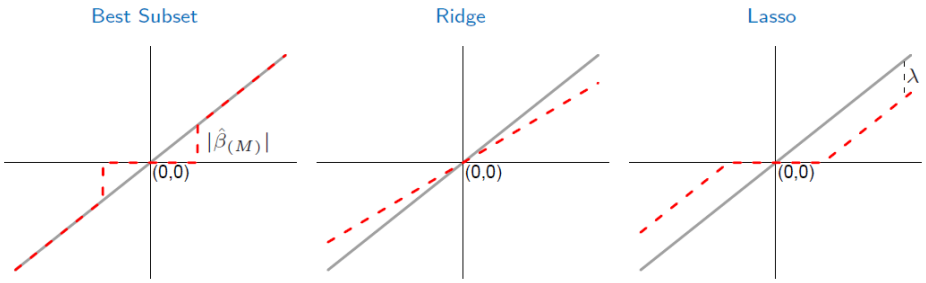

Figure 2: Proportional shrinking (ridge) and soft thresholding (lasso). Hard thresholding is represented by the best subset selection, which is much harder to solve in practice. The gray slopped line represents the unrestricted estimates. *Image Source: Elements of Statistical Learning*

The final question is how the shrinkage parameter λ is chosen for lasso and ridge. In practice, this typically done using leave-one-out or k-fold cross validation. Considering a set of candidates $\{\lambda_1, ..., \lambda_m\}$ we choose

$$\hat{\lambda} = \underset{\lambda}{\text{argmin}} CVError(\lambda)$$

The process of choosing an external parameter (like choosing the shrinkage λ) prior to fitting the model is called *hyperparameter tuning* in machine learning.

3 Classification Methods

3.1 Logistic Regression

Logistic regression is a linear supervised classification method, typically used a binary classifier. The idea is to model the log-odds ratio as a linear relationship with x. Let $p(x;\beta) = \mathbb{P}(Y = 1|X = x)$ be the probability that the outcome is $Y = 1$ conditioned on $X = x$. In the case of binary classification, we model

$$logit(p(x;\beta)) := log\left(\frac{p(x;\beta)}{1-p(x;\beta)}\right) = \beta^T x$$

Re-arranging the terms, this is equivalent to modeling the probability $p(x;\beta)$ as

$$p(x;\beta) = \frac{exp(\beta^T x)}{1 + exp(\beta^T x)}$$

We see that $p(x;\beta)$ ranges from $(0, 1)$, which makes it a valid probability. The set where the log-odds is 0 (i.e. the probabilities of both classes are equal) is called the *decision boundary*. In set-builder notation, we can write the decision boundary as $\{x : \beta^T x = 0\}$. On this decision boundary, the probability of predicting either $Y = 1$ or $Y = 0$ is both 0.5.

Note that in logistic regression, the decision boundary is linear, and thus has very similar assumptions to linear regression:

- We assume that the logit (log-odds) is linear in its relationship with the predictor variables x
- We assume that observations are independent of each other
- We assume that there is no perfect or complete separation

In the case of complete separation, the logistic coefficients will tend to (positive or negative) infinity due to an unstable maximum likelihood. This is intuitive since if $p = 1$ or $p = 0$, the log-odds does not exist. This problem can be addressed with penalized (e.g. L2) logistic regression.

We can also generalize the binary classification case to the multinomial case (with K classes instead of 2). The model has the following form, comparing every class $Y = 1, ..., K-1$ to class K. Here, β_k are the parameters (slope and intercepts) for class k.

$$\begin{aligned}
log\frac{\mathbb{P}(Y = 1|X = x)}{\mathbb{P}(Y = K|X = x)} &= \beta_1^T x \\
log\frac{\mathbb{P}(Y = 2|X = x)}{\mathbb{P}(Y = K|X = x)} &= \beta_2^T x \\
&... \\
log\frac{\mathbb{P}(Y = K-1|X = x)}{\mathbb{P}(Y = K|X = x)} &= \beta_{k-1}^T x
\end{aligned}$$

Rearranging the terms, we can write

$$\begin{aligned}
\mathbb{P}(Y = k|X = x) &= \frac{\beta_k^T x}{1 + \sum_{\ell=1}^{K-1} exp(\beta_\ell^T x)} \; for \; k = 1, ..., K-1 \\
\mathbb{P}(Y = K|X = x) &= \frac{1}{1 + \sum_{\ell=1}^{K-1} exp(\beta_\ell^T x)}
\end{aligned}$$

The probabilities for $k = 1, ..., K$ clearly sum to 1, so they are valid multinomial probabilities.

3.1.1 Fitting Logistic Regression

Logistic regression is typically fit using maximum likelihood estimation (recall that this corresponds with the cross-entropy loss function for classification). The log-likelihood for the multinomial case for n independent observations is as follows

$$\ell(\theta) = \sum_{i=1}^{n} logp_{y_i}(x_i;\theta)$$

where $p_{y_i} = \mathbb{P}(Y = y_i|X = x_i; \theta)$ and θ is the parameter set, $\theta = \{\beta_1, ..., \beta_{K-1}\}$. In the case of binary outcomes, we can write the log-likelihood as

$$\ell(\beta) = \sum_{i=1}^{n} y_i log p(x_i; \beta) + (1 - y_i) log(1 - p(x_i; \beta))$$

where $p = \mathbb{P}(Y = 1|X = x_i; \beta)$. We can optimize for the loss using some numerical method, i.e. Netwon-Raphson, gradient descent, etc.

3.2 Support Vector Machine (SVM)

Consider a binary classification problem with $y_i \in \{+1, -1\}$ with dataset $\{x_i, y_i\}_{i=1}^n$. If the data is perfectly separable, there a infinitely many choices for a decision boundary. In Support Vector Machine, the boundary hyperplane is chosen so that the two classes have the greatest separation (known as the maximum-margin hyperplane). Consider a classifier with weights/parameters w such that positive predictions are induced by the linear decision rule.

$$\hat{y_i} = \mathbf{1}\{w^T x_i > 0\}$$

A reasonable choice for the loss function of our classifier is the $0 - 1$ loss (0 if the prediction was correct, 1 otherwise). However, since the $0 - 1$ loss is not convex and difficult to optimize, we can consider the hinge loss as a proxy:

$$\ell(w; (x, y)) = \max(0, 1 - y_i w^T x_i)$$

That is, for positive $y = 1$, there is no penalty if $w^T x > 1$ and a penalty of $1 - w^T x$ if $w^T x < 1$. Similarly, for negative $y = -1$, there is no penalty if $w^T x < -1$ and a penalty of $1 + w^T x$ if $w^T x > -1$. Note how the penalty is a function of the distance from the margin lines $w^T x = 1$ or $w^T x = -1$ depending on the sign of y, which is different than the $0 - 1$ loss that does not take into account how far away a misclassified point is. We visualize this setup below in Figure 3.

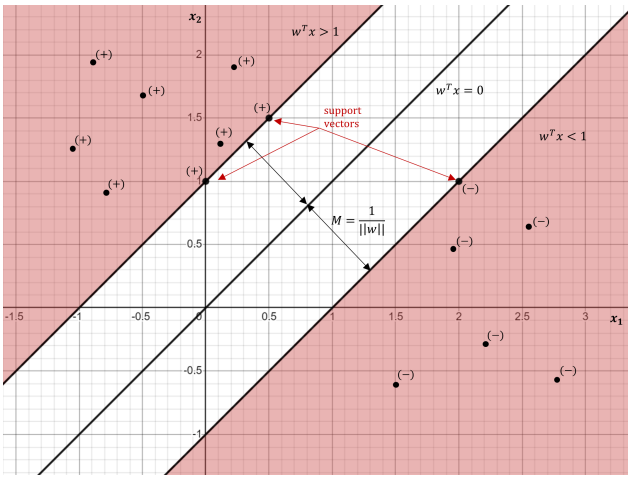

Figure 3: Support vector machine.

If we solve for

$$\hat{w} = \underset{w}{\text{argmin}} \frac{1}{n} \sum_{i=1}^{n} \ell(w; (x_i, y_i)) = \underset{w}{\text{argmin}} \frac{1}{n} \sum_{i=1}^{n} \max(0, 1 - y_i w^T x_i)$$

on perfectly separable data, we will achieve a total loss of 0. For every datapoint with positive y_i we will have $\hat{w}^T x_i > 1$, and for every datapoint with negative y_i we will have $\hat{w}^T x_i < -1$. That is, to achieve a loss of zero, we solve for separating hyperplanes such that every training datapoint is on the correct side of their respective separating hyperplane.

However, we not only want every data point to be on the correct side, but also that the margin width is as large as possible. Let the margin width be denoted as M, which is the distance between the separating hyperplanes $w^T x = 0$ and $w^T x = 1$. M is the length of the vector in the direction of w (which is orthogonal to the line $w^T x = 0$) that intersects $w^T x = 1$. Setting this vector as $x_{in} = cw$, we have that $(cw)^T w = 1$ so $c = \frac{1}{||w||^2}$. Then it easily follows that

$$M = ||x_{in}|| = c||w|| = \frac{1}{||w||}$$

Then our goal is to optimize

$$\hat{w} = \underset{w}{\operatorname{argmin}} \left(\lambda ||w||_2^2 + \frac{1}{n} \sum_{i=1}^{n} \max(0, 1 - y_i w^T x_i) \right)$$

where the first term encourages the margin width to be maximized, and the second term encourages the datapoints to be on the correct side of the margin. The hyperparameter λ controls the trade-off between the two terms which is important to achieve the bias-variance tradeoff. For example, if we want larger margins that are less sensitive to noise, we prioritize small $||w||$ and allow for some missclassification from the hinge loss.

3.3 k-Nearest Neighbors (kNN)

kNN is a simple classification algorithm based on the intuition that similar points are close to each other. The algorithm makes predictions by finding the k-closest points in X, and then taking either a majority vote for classification problems or taking the average of the neighbors for regression problems to predict the target y. Euclidean distance is the classic choice as the measure of distance. Note that in this model, the number of neighbors k is a model hyperparameter that needs to be tuned.

4 Tree Based Methods

4.1 Decision Trees (CART)

Tree based methods essentially divide the input feature space into rectangles, and fit simple models (often times a constant) for each partition region. To simplify the description of each rectangular region, we consider partitions that can be created with recursive binary splits. Consider the following example in Figure 4, where the partitioning can be made using the following steps

1. Split X_1 at t_1
2. Split the regions $(X_1 \leq t_1)$ at $X_2 = t_2$
3. Split the region $(X_1 > t_1)$ at $X_1 = t_3$
4. Split the region $(X_1 > t_3)$ at $X_2 = t_4$

These recursive splits result in 5 different regions: R_1, R_2, R_3, R_4, R_5.

Equivalently, we can represent the splits using a binary tree, with all the data beginning at the top. Then, at each junction node, data satisfying the condition are assigned to the left branch, and the rest of the data are assigned to the right branch. The terminal nodes, which are called leaves, are what define the final resulting regions (see Figure 5).

If we model each region with a constant, the response surface may look like the following in Figure 6.
Since the partitions for an entire feature space can be described by a single tree, one main advantage of CART (classification and regression trees) is their interpretability. This is true in particular for shallow trees.

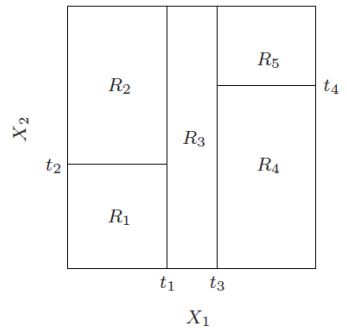

Figure 4: An example of a 2D input space partitioned using recusive binary splits *Image Source: Elements of Statistical Learning*

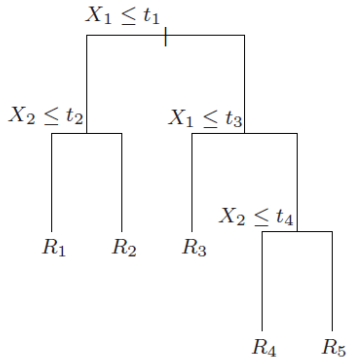

Figure 5: Rectangular partitions represented as a binary tree *Image Source: Elements of Statistical Learning*

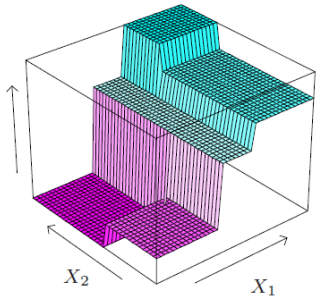

Figure 6: Response surface of a decision tree *Image Source: Elements of Statistical Learning*

4.1.1 Trees in Regression

As usual, consider the data $\{x_i, y_i\}_{i=1}^n$ of n data observations, with the features $x \in \mathbb{R}^p$ consisting of p variables $(x_1, ..., x_p)$. Suppose we have M regions, and we model the response in each region with a constant c_m:

$$f(x) = \sum_{m=1}^{M} c_m \mathbf{1}\{x \in R_m\}$$

c_m can be found such that the sum of squares is minimized.

$$c_m = \underset{c}{argmin} \sum_{i \in R_m} (y_i - c)^2$$

Setting the derivative of the objective with respect to c to zero

$$\left(\sum_{i \in R_m} (y_i - c)^2 \right)' = - \sum_{i \in R_m} 2(y_i - c) = 0$$

Then

$$c_m = \frac{1}{|R_m|} \sum_{i \in R_m} y_i$$

That is, the optimal c_m is simply the average of the points in that region. If we split variable j at split point s, then the regions defined after the split are

$$R_1(j, s) = \{x | x_j \leq s\}$$
$$R_2(j, s) = \{x | x_j > s\}$$

We seek to find the variable to split j and the split point s by minimizing an impurity criterion, such as the one formulated below for a regression problem. This impurity criterion is analogous to the error sum of squares in linear regression. We solve:

$$\underset{j,s}{min} \left[\sum_{x_i \in R_1(j,s)} (y_i - \hat{c}_1)^2 + \sum_{x_i \in R_2(j,s)} (y_i - \hat{c}_2)^2 \right]$$

Recall that the predictions $\hat{c}_1$ and $\hat{c}_2$ are the sample averages of the points belong to region 1 and 2, respectively. It turns out, for each splitting variable, the split point can be solved quickly. We can scan though all potential variables and find the best pair (j, s) that minimize the impurity measure the most for that split. Then this process is repeated iteratively (i.e. greedy search) for both of the resulting regions to continue growing the tree.

When should we stop growing the tree? A tree that is too large will have high variance and overfit the data. To answer this question, consider cost-complexity pruning. The strategy is to grow a large tree (T_0) and prune afterwards to achieve the bias-variance tradeoff.

Let T be a subtree of T_0, $T \subset T_0$. Denote $|T|$ as the number of terminal leaf nodes in T, indexed by m. Let N_m be the number of data points in the region R_m, and let $\hat{c}_m$ be the average of all data points in R_m, as before. Defining the square error impurity for the mth region as unexplained variance that region, we have

$$Q_m(T) = \frac{1}{N_m} \sum_{x_i \in R_m} (y_i - \hat{c}_m)^2$$

To account for all regions, we can define the cost-complexity criterion:

$$C_\alpha(T) = \sum_{m=1}^{|T|} N_m Q_m(T) + \alpha |T|$$

The first term is a measure of cost, and the second term penalizes model complexity - by balancing the two, we control the trade-off between goodness of fit and tree size. α is a hyperparameter that needs to be tuned. Overall, out of all candidate subtrees, we want to choose the subtree with the lowest $C_\alpha(T)$.

Let T_α be the smallest optimal subtree. To find T_α, we start from the full tree T_0 and successively collapse the internal node (i.e. non-leaf node) that produces the smallest per-node increase in $\sum_m N_m Q_m(T)$, the total impurity, until we are only left with one node. One can show that this obtains a finite sequence of subtrees that must contain T_α. This strategy is called weakest-link pruning.

4.1.2 Trees in Classification

Classification trees are very similar to regression trees except the square error impurity does not make sense. We need a new measure of $Q_m(T)$ to grow and prune trees. Consider a classification problem with $k = 1, ..., K$ different outcomes. Let

$$\hat{p}_{mk} = \frac{1}{N_m} \sum_{i \in R_m} \mathbf{1}\{y_i = k\}$$

That is, the proportion of observations belonging to class k in leaf node/region m. We classify observations in node m as $k(m) = \underset{k}{argmax}\ \hat{p}_{mk}$, or in other words, classifications for each region are made based on the majority class in that region. Different choices for node impurity $Q_m(T)$ are delineated below.

Missclassification rate, where $k(m)$ is the prediction for region m:

$$\frac{1}{N_m} \sum_{i \in R_m} \mathbf{1}\{y_i \neq k(m)\} = 1 - \hat{p}_{mk(m)}$$

Gini index:

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

Entropy or deviance:

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

It is common to grow a tree using cross-entropy or the Gini index, and then prune using the missclassification error. To gain intuition on the Gini index, consider classifying a randomly chosen observation to class k from region m with probability $\hat{p}_{mk}$. Recall that $\hat{p}_{mk}$ represents the proportion of class k in node m. Then the probability of missclassifying that observation, by the Law of Total Probability, is the Gini index.

$$\mathbb{P}\{missclassify\} = \sum_{k=1}^{K} \mathbb{P}\{missclassify \mid select\ class\ k\}\mathbb{P}\{select\ class\ k\} = \sum_{k=1}^{K} (1 - \hat{p}_{mk})(\hat{p}_{mk})$$

4.1.3 Pros and Cons of Decision Trees

Drawbacks:

1. *Instability / high variance:* A small change in the data can result in a very different series of splits as errors from higher splits are propagated down the entire tree. Consider bagging or a random forrest to reduce the varaince.
2. *Lack of smoothness:* Performance may suffer particularly for regression problems with smooth surfaces.
3. *Difficulty modeling an additive structure:* Suppose we have the model $y = c_1\mathbf{1}\{x_1 < t_1\} + c_2\mathbf{1}\{x_2 < t_2\} + \epsilon$. A decision tree might make a split at $x_1 = t_1$, but it will have difficulty splitting *both* the resulting nodes at $x_2 = t_2$

Advantages:

1. *Interpretable:* the entire input space is clearly and explicitly mapped. Note this is not true for methods built on top of trees such as random forest or gradient boosting - these methods change the structure of the model.
2. *Fast to construct*
3. *Handles mixed data*: CART supports both categorical and numerical variables
4. *Invariant to transformations of individual predictors*
5. *Performs internal feature selection*, i.e., features are selected automatically in the splitting process.
6. *Robust to outliers*

4.2 Random Forest

A random forest is a modified bagging technique so that many different decision trees can be averaged to reduce the overall variance of the model. Bagging is a strategy that averages many high-variance, low-bias models, which we will discuss later. Suppose we have random variables $X_1, ..., X_B$ with shared variance σ^2 and pairwise correlation ρ. In such a case, the variance of the sample mean is

$$\begin{aligned} Var\left(\frac{1}{B}\sum_{i=1}^{B} X_i\right) &= \frac{1}{B^2} Cov\left(\sum_{i=1}^{B} X_i, \sum_{i=1}^{B} X_i\right) = \frac{1}{B^2}\left(\sum_{i=1}^{B} Var(X_i) + \sum_{i\neq j} Cov(X_i, X_j)\right) \\ &= \frac{1}{B^2}\left(B\sigma^2 + \sum_{i\neq j} Cor(X_i, X_j)\sqrt{Var(X_i)Var(X_j)}\right) \\ &= \frac{1}{B^2}\left(B\sigma^2 + B(1-B)\rho\sigma^2\right) = \rho\sigma^2 + \frac{1-\rho}{B}\sigma^2 \end{aligned}$$

As $B \to \infty$, we are only left with the first term which never goes to zero so long as ρ is not zero. This highlights the basic idea of the random forest, that is, to reduce variance by averaging many *decorrelated* trees by randomly selecting $m \leq p$ input features at each split for each tree. The algorithm is as follows (see Algorithm 1)

Algorithm 1 Random Forest

for $b = 1, ..., B$ **do**

- Draw a bootstrap sample Z^* of size n from the training data
- Grow a random-forest tree T_b using Z^* by recursively repeating the following steps for each terminal leaf node of the tree, until the minimum node size n_{min} is reached.
  - Select m variables at random from the p variables
  - Identify the best variable and split-point from the m variables
  - Split the node into two daughter nodes

end for
return ensemble of trees $\{T_b\}_{b=1}^{B}$

To make a prediction on a new observation x, we return the average of the forest for regression and the majority vote for classification problems. Specifically, in regression, a prediction is made using

$$\hat{f}_{rf}^{B} = \frac{1}{B}\sum_{b=1}^{B} T_b(x)$$

In classification, we can take a majority vote across the entire forest. Let $\hat{C}_b(x)$ be the class prediction of the bth random-forest tree. Then the predicted class for x is

$$\hat{C}_{rf}^B = majority\ vote\ \{\hat{C}_b(x)\}_{b=1}^B$$

The bias-variance tradeoff in a random forest is controlled by m. Choosing a smaller m intuitively reduces the correlation between trees, and thus reduces the variance of the sample average as shown above. However, choosing a m to be too small introduces bias into the model. Overall, random forests generally perform well, are easy to train, not prone to overfit, and have decent interpretability.

4.2.1 Out-of-Bag (OOB) Samples

For each observation $z_i = (x_i, y_i)$, we can construct its RF predictor by averaging trees trained on bootstrap samples *that did not include* z_i. Note that this is very similar to the idea behind k-fold cross-validation. Hyperparameters, such as the number of forest trees, can be chosen when OOB error stabilizes.

4.2.2 Feature Importance

It is possible to easily rank feature importance after fitting a random forest model. One strategy is to keep track of how much a variable decreases impurity (Gini index, sample variance, etc...) at its respective splits and accumulate them over all the trees in the forest for each variable. This approach is computationally inexpensive, but tends to over-inflate the importance of continuous variables or high cardinality categorical variables. Additionally, consider two variables that are equally important but highly co-linear. The model can only select one of the variables to split higher up the tree, so in the subsequent lower splits, the other variable will be considered less important as its information was already captured in the first variable higher up the tree. Another strategy using OOB samples is called permutation feature importance. After growing the bth tree, we can calculate the prediction accuracy for its respective OOB samples. Then, for the jth variable, we can randomly permutate its values among the OOB samples and measure the resulting decrease in prediction accuracy. Intuitively, the larger the decrease in prediction accuracy, the more important variable j was. The results are averaged for all trees in the forest.

5 Ensemble Methods

5.1 Bagging

In bootstrap aggregation or bagging, we fit several models based on bootstrap samples from the data and average their results. By doing this, bagging can dramatically reduce the variance of many high variance models, like decision trees for example. Roughly speaking, these methods train models in parallel to reduce the variance whiling leaving bias largely unchanged. For example, if we were to bag decision trees in regression, we could average the predictions of all the individual trees for a given x. For classification, we could either average the estimates of class probabilities, or take the majority-vote for the predictions of each tree.

Consider the training data $Z = \{x_i, y_i\}_{i=1}^n$. Let $\{Z^{*b}\}_{b=1}^B$ represent B different bootstrap samples of the data, with corresponding predictions on the bootstrap samples $\hat{f}^{*b}(x)$ for a given x. Then in regression, the bagged estimate is

$$\hat{f}_{bag}(x) = \frac{1}{B}\sum_{b=1}^{B} \hat{f}^{*b}(x)$$

This expression is an estimate of the true bagging estimate, which we can express as $\mathbb{E}_{\hat{\mathcal{P}}}\hat{f}^*(x)$, where $\hat{\mathcal{P}}$ is an empirical distribution weighting each of the data samples (x_i, y_i) with equal probability $\frac{1}{n}$. Denote the bootstrap sample that $\hat{f}^*(x)$ is trained on as $Z^* = \{x_i^*, y_i^*\}_{i=1}^n$ where each (x_i^*, y_i^*) is drawn from the data Z based on $\hat{\mathcal{P}}$. As $B \to \infty$, the estimate converges to the true expectation by the law of large numbers.

In classification, we can consider a majority vote.

$$\hat{f}_{bag}(x) = \underset{k}{\text{argmax}}\{p_1(x), ..., p_k(x)\}$$

where $\{p_1(x), ..., p_k(x)\}$ represents the proportion of trees predicting each of the $1, ..., k$ classes. Note that these proportions are *not the class probability estimates*. For example, in a 0-1 binary classification where the class probability of predicting 1 is 0.75. Then if we had two trees that we bag, and both trees correctly predict 1, then $p_1(x) = 1$ which is different than the true class probability.

For motivation in the regression case, consider aggregation on the true population distribution. That is, instead of drawing from the empirical distribution $\hat{\mathcal{P}}$, consider drawing from the true distribution $\mathcal{P}$. Then the ideal aggregated estimator is $f_{ag}(x) = \mathbb{E}_{\mathcal{P}}\hat{f}^*(x)$ where $\hat{f}^*(x)$ is trained on the dataset Z^* drawn from the true distribution $\mathcal{P}$. Consider comparing the mean square error of $f_{ag}(x)$ and $\hat{f}^*(x)$:

$$\begin{aligned}
\mathbb{E}_{\mathcal{P}}[(y - \hat{f}^*(x))^2] &= \mathbb{E}_{\mathcal{P}}[(y - f_{ag}(x) + f_{ag}(x) - \hat{f}^*(x))^2] \\
&= \mathbb{E}_{\mathcal{P}}[(y - f_{ag}(x))^2] + \mathbb{E}_{\mathcal{P}}[(\hat{f}^*(x) - f_{ag}(x))^2] \\
&\geq \mathbb{E}_{\mathcal{P}}[(y - f_{ag}(x))^2]
\end{aligned}$$

Thus, at the population level, aggregation *never increases mean squared error*. Further, we know that in bootstrapping, the empirical distribution $\hat{\mathcal{P}}$ converges to the true distribution $\mathcal{P}$ for large samples. This suggests that bagging will often times decrease the overall mean squared error. This argument, however, does not hold under classification.

For the classification case, bagging a poor classifier creates a worse classifier, but bagging a good classifier can make it better. To understand this, we consider what is known as the Wisdom of Crowds, or the idea that the consensus of independent weak learners is stronger than any of the models alone. For example, consider a binary classification case. Let $\hat{f}_b^*(x)$ be one of the independent $b = 1, ..., B$ bagged models, and assume that the error rate e is constant for each model, and is less than 0.5 so $e < 0.5$. Suppose the true label is $y = 1$ at the given x. The consensus vote for predicting $\hat{y} = 1$ can be written

$$S_1(x) = \sum_{b=1}^{B} 1\{\hat{f}_b^*(x) = 1\}$$

Note that since we assumed the models are independent, we have

$$S_1(x) \sim Binomial(B, 1 - e)$$

Finally, note that the probability of getting the correct answer of $y = 1$ (in other words, $S_1 > B/2$) goes to 1 as B grows. In contrast, the upper-bound for the error of a single predictor is 0.5. Through this demonstration, we can also see that if we started with a poor model with $e > 0.5$, the probability of getting the desired answer does not increase but actually decreases. Note that an important assumption made here is the independence assumption between models - which is not true for bagged trees. Recall that the Random Forest improves on this issue by using bootstrap samples of both *data* and *features* for each tree.

5.2 Stacking

Stacking refers to the process of combining the strengths of several different models. After training several classifiers on the training data (called level 0 predictors), we can further combine the predictions using a logistic regression model (called a level 1 predictor), and evaluate the final model.

If we take a Bayesian interpretation of model stacking, we can view the posterior mean as a weighted average of level 0 predictions. Suppose that the prediction $f(x)$ is our quantity of interest, considering some fixed

feature variable x. Let Z represent out training data set. Let the level 0 models be $\mathcal{M}_m$ indexed by $m = 1, ..., M$. Then the posterior distribution of $f(x)$ is

$$P(f(x)|Z) = \sum_{m=1}^{M} P(f(x)|\mathcal{M}_m, Z)P(\mathcal{M}_m|Z)$$

taking the posterior mean, we have

$$\mathbb{E}(f(x)|Z) = \sum_{m=1}^{M} \mathbb{E}(f(x)|\mathcal{M}_m, Z)P(\mathcal{M}_m|Z)$$

Thus, the posterior mean (which presents our final prediction), is a weighted average of each individual prediction.

In the Frequentist viewpoint, given the predictions $\hat{f}_1(x), ..., \hat{f}_M(x)$ for fixed x, and let the dataset Z be distributed according to $\mathcal{P}$. We can search for weights to combine the predictions that minimize mean square error such that

$$\hat{w} = \underset{w}{argmin}\ \mathbb{E}_{\mathcal{P}}\left[\left(Y - \sum_{m=1}^{M} w_m \hat{f}_m(x)\right)^2\right]$$

At the population level, this problem is simply a linear regression of Y on $\{\hat{f}_1(x), ..., \hat{f}_M(x)\}$. Note that the mean square error in this case cannot be worse than the mean square error of any individual model, i.e., in the worst case, the other coefficients can all be set to 0 and the MSE is unchanged. Also note that while we discuss using a level 1 linear regression here, any other learning algorithm can be used to combine the level 0 models. In practice, stacking models typically perform better than any one single model.

5.3 Boosting

5.3.1 Gradient Boosting

In boosting, we want to iteratively/sequentially fit models (so called "weak learners") so that each model improves on the previous model's mistakes. Consider $f_m(x_i)$, the current model at the mth iteration. To improve on this model, we could add another model $h_m(x_i)$ as follows to obtain the $m+1$ model iteration:

$$f_{m+1}(x_i) = f_m(x_i) + h_m(x_i) = y_i$$

Equivalently, we can write $h_m(x_i) = y_i - f_m(x_i)$. Realistically, $h_m(x)$ may not be able to perfectly achieve this for all $i = 1, ..., n$ in the training data especially if it is a weak learner, but at least we can fit $h_m(x)$ such that it models the residuals of the current model as well as possible.

This sequential process also has a gradient descent interpretation. Consider the MSE loss function and its derivative w.r.t the function value $f_m(x_i)$. Here, we see that the negative gradient of the MSE loss is *proportional to the residual*.

$$\begin{aligned} L(y_i, f_m(x_i)) &= \frac{1}{n}\sum_{i=1}^{n}(y_i - f_m(x_i))^2 \\ -\frac{\partial L}{\partial f_m(x_i)} &= \frac{2}{n}(y_i - f_m(x_i)) = constant * (y_i - f_m(x_i)) \end{aligned}$$

In this view, a gradient descent update to f that minimizes MSE on the training data could look like the following, for some constant stepsize in the direction of the negative gradient:

$$f_{m+1}(x_i) = f_m(x_i) - constant * \frac{\partial L}{\partial f_m(x_i)} \approx f_m(x_i) + constant * h_m(x_i)$$

Similar to before, we can model the gradient using $h_m(x)$. In this case, the gradient of the loss (known as the pseudo-residual) is the signal that tells us how to improve the next model. Generalizing this example, we note that in gradient boosting, we are seeking a sequence of $m = 1, ..., M$ weak learners so that the final approximating function is a weighted sum of the sequence. That is,

$$\hat{f}(x) = \sum_{m=1}^{M} \gamma_m h_m(x)$$

The goal of $\hat{f}(x)$ is to minimize the loss over the training data $\{x_i, y_i\}_{i=1}^n$, in accordance with the principle of empirical risk minimization.

$$\hat{f}(x) = \underset{f}{argmin} \sum_{i=1}^{n} L(y_i, f(x_i))$$

We can solve for the components of $\hat{f}(x)$ in a "forward stage-wise" manner. That is, for each iteration m, we optimize for the next additive function and fix the previous $m-1$ function values. This process is initialized by fitting a constant (denoted γ below, not to be confused with γ_m). Consider a class of weak learners $\mathcal{H}$ and some arbitrary loss function L. Then we can write for $m = 1, ..., M$:

$$f_0(x) = h_0(x) = \underset{\gamma}{\text{argmin}} \sum_{i=1}^{n} L(y_i, \gamma)$$

$$f_m(x) = f_{m-1}(x) + \left(\underset{h_m \in \mathcal{H}}{\text{argmin}} \sum_{i=1}^{n} L\left(y_i, f_{m-1}(x_i) + h_m(x_i)\right) \right)(x)$$

This problem as stated above is computationally infeasible. Instead, we can consider the "steepest descent" approach to minimize the loss with a step size of γ_m, i.e. using functional gradient descent. Note that since a function space can be thought of as infinite dimensional, the gradient can also be thought of as an infinite dimensional vector.

$$f_m(x) = f_{m-1}(x) - \gamma_m \nabla_{f_{m-1}} \sum_{i=1}^{n} L(y_i, f_{m-1}(x_i))$$

where γ_m can be found using a line search

$$\gamma_m = \underset{\gamma}{\text{argmin}} \sum_{i=1}^{n} L(y_i, f_{m-1}(x) - \gamma \nabla_{f_{m-1}} L(y_i, f_{m-1}(x_i)))$$

The problem with the above equations is that the gradients are only defined on the observed training data, yet our goal is to apply the model to unseen data. To resolve this issue, we fit a weak learner $h_m(x)$, a small decision tree for example, to the computed (negative) derivatives over the training data (the pseudo-residuals). The pseudo-residuals r_{im} for the ith training data observation and the mth iteration can be written:

$$r_{im} = -\left(\frac{\partial L(y_i, f_{m-1}(x_i)}{\partial f_{m-1}(x_i)} \right)$$

Intuitively, these pseudo residuals represent the error of the model at the current iteration, and fitting a weak learner to them allows us to estimate the error at any input x beyond the training data. We can write a general gradient boosting algorithm as follows (see Algorithm 2).

5.3.2 Gradient Boosted Trees

The most common class of "weak learners" used in gradient boosting is the Decision Tree, hence the name. For the mth step, suppose that the decision tree $h_m(x)$ has J_m leaf nodes. That is, $h_m(x)$ divides the input space into J_m different regions, which we can denote as $R_{1m}, ..., R_{Jm}$, and models each region with a constant. We can write the decision tree as

$$h_m(x) = \sum_{j=1}^{J_m} b_{jm} \mathbf{1}\{x \in R_{jm}\}$$

Algorithm 2 General gradient boosting

$f_0(x) \leftarrow \underset{\gamma}{\text{argmin}} \sum_{i=1}^{n} L(y_i, \gamma)$

for $m = 1, ..., M$ **do**

- Compute the pseudo-residuals:

$$r_{im} = -\left(\frac{\partial L(y_i, f_{m-1}(x_i)}{\partial f_{m-1}(x_i)} \right)$$

  for all $i = 1, ..., n$

- Fit a weak learner $h_m(x)$ to the pseudo-residuals using the training set $\{x_i, r_{im}\}_{i=1}^{n}$
- Compute γ_m using line search:

$$\gamma_m = \underset{\gamma}{\text{argmin}} \sum_{i=1}^{n} L(y_i, f_{m-1}(x_i) + \gamma h_m(x_i))$$

- Perform the update step:

$$f_m(x) = f_{m-1}(x) + \gamma_m h_m(x)$$

end for
return $f_M(x)$

Then the update equations directly follow as

$$f_m(x) = f_{m-1}(x) + \gamma_m h_m(x)$$
$$\gamma_m = \underset{\gamma}{\text{argmin}} \sum_{i=1}^{n} L(y_i, f_{m-1}(x_i) + \gamma h_m(x_i))$$

Alternatively, instead of using γ_m for the entire tree, we can optimize for a constant γ_{mj} for *each tree region*, and b_{jm} can be discarded. In other words, solving the line search for γ_{jm} could look like

$$\gamma_{jm} = \underset{\gamma}{\text{argmin}} \sum_{x_i \in R_{jm}} L(y_i, f_{m-1}(x_i) + \gamma)$$

with the update equation

$$f_m(x) = f_{m-1}(x) + \sum_{j=1}^{J_m} \gamma_{mj} \mathbf{1}\{x \in R_{jm}\}$$

5.3.3 Regularization

The two obvious hyperparameters that control the bias variance tradeoff are the size of the trees J and the number of boosting rounds M. If we set $J = 2$ (i.e. only one split), we do not allow for the interaction between terms. $J = 3$ allows for two-variable interactions and so on and so forth. In practice, choosing J between 4 and 8 yields good results, and $J > 10$ is seldom needed. Note that setting J to be too large will incur high variance. The best strategy is to try several different J and choose the one with the highest performance on a hold-out validation set.

For M, note that each boosting iteration reduces the loss on the training set. If M is too large, we can overfit the training data and do poorly on the test data. A convenient way to choose an optimal M is to monitor the performance on a validation set and to choose M where the performance is best, similar to early stopping in neural networks.

Another strategy is called shrinkage, where we control the "learning rate" of the boosting step. For example, let ν be a constant between 0 and 1. Then the update equation can be written as

$$f_m(x) = f_{m-1}(x) + \nu\gamma_m h_m(x)$$

Note that there is a tradeoff between ν and M, that is, for the same training error smaller ν corresponds to larger M. However, empirically, it appears that the best strategy is to set a small $\nu < 0.1$ and then control M using early stopping.

Finally, similar to the idea of bootstrap averaging to reduce variance, we can sample a fraction η of the training data without replacement. This is known as sub-sampling, and is related to *stochastic gradient boosting*. Sub-sampling improves the performance of the model and improves the computational time required for training. A typical choice is $\eta = 0.5$ but even smaller choices are possible for large n.

6 Neural Networks and Deep Learning

6.1 The Feedforward Neural Network

Neural networks are nonlinear statistical models used in both regression and classification. Below we discuss a model with one "hidden layer," and then extend the result to obtain a deep feedforward network with mutiple hidden layers (multi-layer perceptron or MLP). Consider the model in Figure 7.

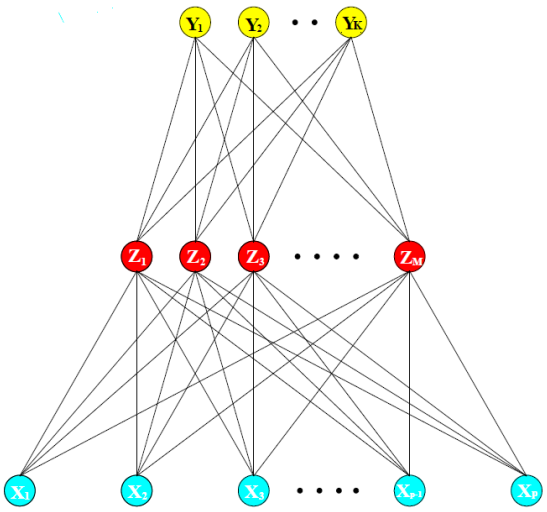

Figure 7: A schematic of a single layer neural network *Image Source: Elements of Statistical Learning*

The intuition for the hidden layer is that the model learns to represent the data in a higher-dimensional space, where predictions could potentially be made easier. This is accomplished through one or multiple non-linear transformations of the input features. In a "feed-forward" neural network, the data is processed sequentially.

1. A nonlinear function is applied to linear combinations of the input data, $x \in \mathbb{R}^p$. This nonlinear function, denoted f, is known as the activation function. The activation function is responsible for introducing non-linearity into the model; historically a common choice was the sigmoid function ($f(v) = \frac{1}{1+e^{-v}}$) but now the ReLU function, or rectified-linear unit ($f(v) = max\{0, v\}$), is a common default option.

$$z_m = f(\alpha_{0m} + \alpha_m^T x),\ m = 1, ..., M$$

Here, z_m for $m = 1, ..., M$ are the hidden units. We can think of them as derived features that serve as a basis expansion of the original $x_1, ..., x_p$ input features. In matrix notation, we can collect all M hidden units as $z = (z_1, ..., z_M) \in \mathbb{R}^M$, and defining the weight matrix $A \in \mathbb{R}^{M,p}$ having $\alpha_1^T, ..., \alpha_M^T$ as its rows and the intercept term $\alpha_0 = (\alpha_{01}, ..., \alpha_{0M}) \in \mathbb{R}^M$ we have

$$z = f(Ax + \alpha_0)$$

Sometimes, α_0 is called the *bias* term.

2. We continue to apply a linear transformation to the hidden units, and again apply a nonlinear transformation f_{out} to obtain the output. Note that f_{out} for the final layer is called the output function. For a regression problem, the output function is usually the identity function (called a linear activation), or another activation function. For classification, we use the softmax function so that each of the $k = 1, ..., K$ classes can be assigned a predicted probability that sum to one.

$$\hat{y}_k = f_{out}(\beta_{0k} + \beta_k^T z),\ k = 1, ..., K$$

Let $B \in \mathbb{R}^{K,M}$ be a weight matrix with $\beta_1^T, ..., \beta_K^T$ as its rows, $\beta_0 = (\beta_{01}, ..., \beta_{0K})$ be the bias term, and $\hat{y} = (\hat{y}_1, ..., \hat{y}_K) \in \mathbb{R}^K$ be the output vector. In matrix notation

$$\hat{y} = f_{out}(Bz + \beta_0)$$

This procedure can easily be extended to an arbitrary number of hidden layers to derive higher level and more complex hidden representations. For example, using the subscript (i) to denote the ith hidden layer and defining the weight matrices/bias terms similarly as above, we have in matrix form

$$\begin{aligned}
Hidden\ layer\ 1:\ & z_{(1)} = f(A_{(1)}x + \alpha_{0(1)}) \in \mathbb{R}^{M_1} \\
Hidden\ layer\ 2:\ & z_{(2)} = f(A_{(2)}z_{(1)} + \alpha_{0(2)}) \in \mathbb{R}^{M_2} \\
& ... \\
Hidden\ layer\ j:\ & z_{(j)} = f(A_{(j)}z_{(j-1)} + \alpha_{0(j)}) \in \mathbb{R}^{M_j} \\
Output\ layer:\ & \hat{y} = f_{out}(Bz_{(j)} + \beta_0) \in \mathbb{R}^K
\end{aligned}$$

Neural networks are mathematically special because of the Universal Approximation Theorem. That is, there exists a neural network that can approximate, to any desired degree of accuracy, any continuous function (on a closed and bounded subset of the reals), as long as we use a "squashing" activation function such as the sigmoid or ReLU. Note that while a neural network with sufficient representation exists in theory, it is not guaranteed that we obtain such a network in training.

6.2 Training Neural Networks

6.2.1 Backpropagation

Let θ be the complete set of weights/parameters, which include for one layer

$$\{\alpha_{0m}, \vec{\alpha}_m, \beta_{0k}, \vec{\beta}_k;\ m = 1, ..., M,\ k = 1, .., K\}$$

where M is the number of hidden units, and K is the output dimension. For regression problems, consider minimizing square error loss:

$$R_n(\theta) = \frac{1}{n}\sum_{i=1}^{n} R_i = \frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{K}(y_{ik} - \hat{y}_{ik}(x_i))^2$$

For classification problems, consider minimizing cross entropy loss:

$$R_n(\theta) = \frac{1}{n}\sum_{i=1}^{n} R_i = -\frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{K} y_{ik} log(\hat{y}_{ik}(x_i))$$

where y_i represents the true label (as a one-hot vector) for the ith data observation and $\hat{y}_i$ represents the softmax probability distribution which assigns a probability to each classification class. Note that for non-convex objectives (as is the case for most deep learning problems), gradient-based optimization algorithms do not find global solutions.

In order to use gradient descent to minimize loss, we need to take derivatives of the loss function with respect to the parameters. For example, in the regression case we have for a single layer network

$$R_n(\theta) = \frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{K}(y_{ik} - \hat{y}_{ik}(x_i))^2 = \frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{K}(y_{ik} - f_k(\beta_{0k} + \beta_k^T z_i))^2$$

The derivative of a single loss R_i with respect to the parameters β_{km}, the mth unit of the parameter vector β_k, and $\alpha_{m\ell}$, the ℓth unit of the parameter vector α_m is

$$\frac{\partial R_i}{\partial \beta_{km}} = -2(y_{ik} - f_k(\beta_{0k} + \beta_k^T z_i))\frac{\partial f_k(\beta_{0k} + \beta_k^T z_i)}{\partial(\beta_{0k} + \beta_k^T z_i)}\frac{\partial(\beta_{0k} + \beta_k^T z_i)}{\partial \beta_{km}}$$

$$\frac{\partial R_i}{\partial \alpha_{m\ell}} = -\sum_{k=1}^{K} 2(y_{ik} - f_k(\beta_{0k} + \beta_k^T z_i))\frac{\partial f_k(\beta_{0k} + \beta_k^T z_i)}{\partial(\beta_{0k} + \beta_k^T z_i)}\frac{\partial(\beta_{0k} + \beta_k^T z_i)}{\partial z_m}\frac{\partial z_m}{\partial(\alpha_{0m} + \alpha_m^T x_i)}\frac{\partial(\alpha_{0m} + \alpha_m^T x_i)}{\partial \alpha_{m\ell}}$$

We can rewrite this result as

$$\frac{\partial R_i}{\partial \beta_{km}} = \delta_{ki} z_{mi}$$
$$\frac{\partial R_i}{\partial \alpha_{m\ell}} = \delta_{mi} x_{i\ell}$$

for $i = 1, .., n$ observations, $m = 1, ..., M$ hidden units, $k = 1, ..., K$ output units, and $\ell = 1, ..., p$ input features. Here δ_{ki} and δ_{mi} are called "errors" and they represent the upstream gradient. Notice that the gradient calculations heavily rely on the use of the chain rule to calculate derivatives with respect to a model parameter after each function evaluation, but in reverse order. This is the essence of the backpropagation algorithm, the name of the algorithm used to calculate the gradients and train neural networks. We pass gradient information backwards using the chain rule to each model parameter so that they can be updated with a gradient based optimization algorithm.

Specifically, backpropagation consists of two steps. Here we let the superscript (t) denote the t^{th} iteration of parameter updates.

1. The Forward Pass: Fix the current weights $\theta^{(t)}$ and find the predicted values $\hat{y}^{(t)}(x_i)$. Calculate the loss at the current iteration.
2. The Backward Pass: Use the chain rule to find the errors δ_{ki} and δ_{mi}. Update the weights using a gradient descent scheme.

For example, one step of a gradient descent update with step-size η_t could look like

$$\beta_{km}^{(t+1)} = \beta_{km}^{(t)} - \eta_t \frac{\partial R_n}{\partial \beta_{km}^{(t)}} = \beta_{km}^{(t)} - \eta_t \frac{1}{n}\sum_{i=1}^{n}\frac{\partial R_i}{\partial \beta_{km}^{(t)}}$$

$$\alpha_{m\ell}^{(t+1)} = \alpha_{m\ell}^{(t)} - \eta_t \frac{\partial R_n}{\partial \alpha_{m\ell}^{(t)}} = \alpha_{m\ell}^{(t)} - \eta_t \frac{1}{n}\sum_{i=1}^{n}\frac{\partial R_i}{\partial \alpha_{m\ell}^{(t)}}$$

Collecting all such updates, we write

$$\theta^{(t+1)} = \theta^{(t)} - \eta_t \nabla_\theta R_n = \theta^{(t)} - \eta_t \frac{1}{n}\sum_{i=1}^{n}\nabla_\theta R_i$$

However, the problem with this approach is that each gradient descent step requires n gradient calculations, one for each data observation. We will find that with Stochastic Gradient Descent, it is possible to obtain good parameter updates with gradients calculated with only one or several data points at a time.

6.2.2 Stochastic Gradient Descent

Here, we take an aside to discuss Stochastic Gradient Descent, a useful optimization algorithm when standard Gradient Descent is slow due to large data. Recall the goal of machine learning is to solve for the minimizer of Risk

$$R(\theta) = \mathbb{E}\ell(\theta, X, Y)$$
$$\tilde{\theta} = \underset{\theta}{\text{argmin}} R(\theta)$$

Here, ℓ is the loss function, and X and Y and random variables for the features and targets, respectively. Note, the true risk cannot be evaluated since the probability distribution of the data X and Y is often times unknown. Instead, in focus on minimizing true risk, we usually minimize empirical risk (estimating the expectation with n iid draws of data),

$$\hat{\theta} = \underset{\theta}{\text{argmin}} R_n(\theta) = \underset{\theta}{\text{argmin}} \frac{1}{n} \sum_{i=1}^{n} \ell(\theta; x_i, y_i)$$

which gives us a deterministic optimization problem since now we are only dealing with the observed data $\{x_i, y_i\}_{i=1}^n$. The problem is, the gradient descent solution requires n gradient calculations at every step, which can be slow for large n. Instead, in stochastic gradient descent, we try to estimate the expectation of the gradient of the random loss directly instead of the expectation of the random loss itself. That is, we notice that

$$\theta^{(t+1)} = \theta^{(t)} - \eta_t \nabla_\theta \mathbb{E}\ell(\theta, X, Y) = \theta^{(t)} - \eta_t \mathbb{E}\left(\nabla_\theta \ell(\theta, X, Y)\right)$$

where $\nabla_\theta \ell(\theta, X, Y)$ is the "stochastic gradient." We then estimate the expectation of the stochastic gradient with m random draws of the data, where $m << n$. Practically, m data samples can be taken from the pool of n observed data points either with or without replacement (the former is easier to analyze theoretically). This is called mini-batch stochastic gradient descent.

$$\theta^{(t+1)} = \theta^{(t)} - \eta_t \frac{1}{m} \sum_{j=1}^{m} \nabla_\theta \ell(\theta, x_j, y_j)$$

Note how for each update step, we now only need m gradient calculations instead of n. Further, in many practical applications, only one sample is sufficient enough to estimate the stochastic gradient and obtain a good solution (intuitively, there is a summing of updates over many iterations which can help average results). In this case

$$\theta^{(t+1)} = \theta^{(t)} - \eta_t \nabla_\theta \ell(\theta, x_j, y_j)$$

only requires one gradient calculation per update step.

It can be shown that standard gradient descent has convergence guarantee (i.e., $||\theta^{(t)} - \theta^*|| \leq \epsilon$ where ϵ is an arbitrarily small error value and θ^* is the true optimum) on the order of $t \sim O(nlog(\frac{1}{\epsilon}))$ while stochastic gradient descent has a convergence rate of $t \sim O(\frac{1}{\epsilon})$. So, in the regime where $O(\frac{1}{\epsilon}) < O(nlog(\frac{1}{\epsilon}))$, which is true for large n, stochastic gradient descent performs better. As a technical detail, these guarantees are for strongly convex, L-smooth optimization problems.

6.2.3 Computational Graphs

In many deep learning frameworks, gradients are calculated automatically using computational graphs, which represent the mathematical operations of a model. For example, if we are calculating $y = (a + b)(b - c)$ then we would have the computational graph as shown in Figure 8. Computational graphs can be followed backwards to compute the gradients. For example, Figure 9 shows the backward derivatives needed from the chain rule to calculate

$$\frac{\partial y}{\partial a} = \frac{\partial y}{\partial d}\frac{\partial d}{\partial a} = e * 1 = e$$

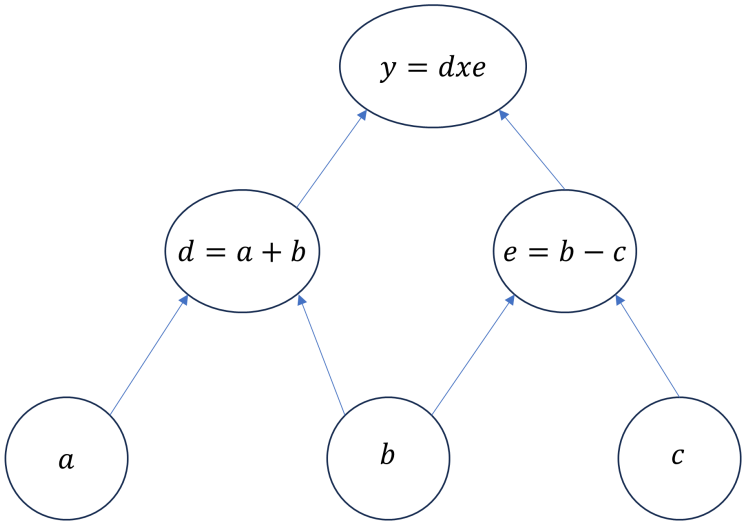

Figure 8: An example of a computational graph

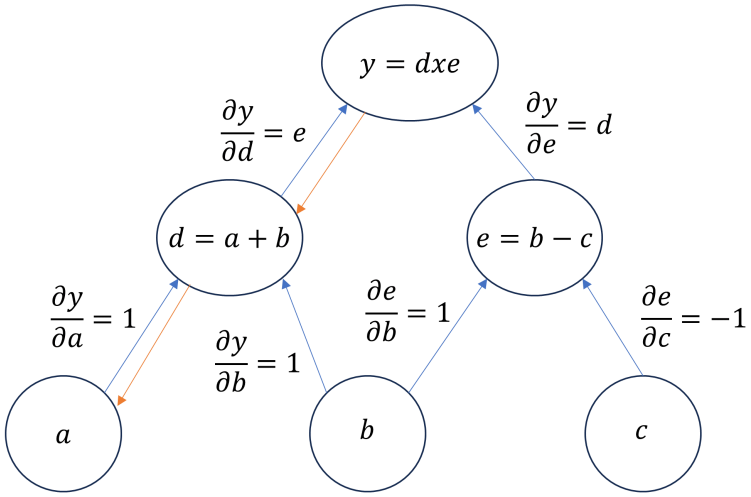

Figure 9: Gradient calculation with a computational graph

As an aside, note that if there are contributions to the gradient at multiple sources, these contributions are summed. That is, gradients are summed at outward branches. For example, if we were interested in $\frac{\partial y}{\partial b}$ then we would need to sum

$$\frac{\partial y}{\partial b} = \frac{\partial y}{\partial d}\frac{\partial d}{\partial b} + \frac{\partial y}{\partial e}\frac{\partial e}{\partial b}$$

Each node receives an "upstream gradient," and the goal is to pass on the correct "downstream" gradients by multiplying the upstream gradient with the corresponding "local gradient" (see Figure 10). For example, if we were calculating $z = Wx$ at a node and we are interested in the gradient of some upstream s, then we would have $\frac{\partial s}{\partial W} = \frac{\partial s}{\partial z}\frac{\partial z}{\partial W}$ where $\frac{\partial s}{\partial z}$ is the upstream gradient, $\frac{\partial z}{\partial W}$ is the local gradient, and $\frac{\partial s}{\partial W}$ is the desired downstream gradient. Similarly, $\frac{\partial s}{\partial x} = \frac{\partial s}{\partial z}\frac{\partial z}{\partial x}$.

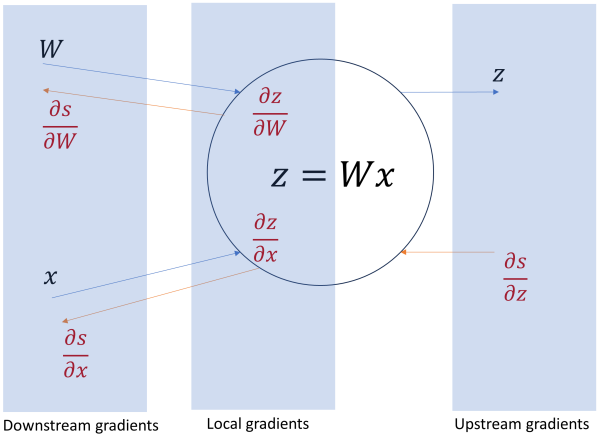

Figure 10: Backpropagation at a single node

6.3 Regularization

Two common ways to regularize deep neural networks are early stopping and dropout. In early stopping, the model is trained on a train dataset and losses are monitored on a separate validation dataset at every training epoch. Here, loss on the validation dataset is a proxy for generalization error. When validation loss does not improve for a set number of epochs (called the *patience*), training is stopped.

Dropout is a regularization technique that prevents feature co-adaptation, that is, where features cannot only be useful when used in conjunction with particular other features. Another view is that dropout allows us to train many different "sub-models" and is a form of bagging that achieves a similar effect to a random forest. To implement dropout, we randomly set inputs to each neuron to 0 using a set fixed probability p in training i.e. we randomly turn some connections in the neural network off during training. During prediction, we do not turn off model connections but instead scale all of them by p.

The classic view of regularization is striking a balance in the bias-variance tradeoff by preventing overfitting. However, the modern view of regularization is that it enables deep neural networks to generalize well even when the model is large and overfit. Specifically, it has been observed that large neural networks that have more parameters than datapoints (called *overparameterized* networks) that include regularization are able to interpolate and generalize well to unseen data even when they are hugely overfit (see literature on double descent). While in the overparameterized regime, there are infinitely possible solutions that perfectly fit the training data, these models appear to have a surprising bias towards solutions that generalize well.

7 Sequential Neural Networks

7.1 Recurrent Neural Networks (RNN)

Sometimes we want to model sequential data with temporal dependence. For example, a language model predicts the next word one at a time, where each next predicted words depend on the context given by the words that came before it. The problem with using a standard neural network to approach these problems is that it has no sense of memory to understand the context of previous events. It also has no built in concept of sequence order. To model temporal dynamics, we look beyond the feed forward network and introduce recurrent neural networks (RNN).

In a recurrent neural network, weights are repeatedly applied at every timestep, and hidden states are obtained sequentially, with each hidden state depending on the previous hidden state and the new input at the current timestep. This theoretically gives the network context of past events prior to each new timestep. Then, using each hidden state, we can optionally produce a sequence of outputs (many-to-many), or produce a single output (many-to-one) using only the last hidden state.

We begin with an initial hidden state $h^{(0)}$ which can be initialized as zeros. Here, the superscript denotes the timestep. At each following timestep, we calculate the hidden state based on the previous hidden state and the new input. Note that the weights the same for each timestep and are applied repeatedly.

$$h^{(t)} = tanh(W_h h^{(t-1)} + W_x x^{(t)} + b_1)$$

where W_h is the weight matrix corresponding to the previous hidden state, W_x is the weight matrix corresponding to the new input x and b_1 is the bias term. Then, we can apply a transformation to the hidden states to arrive at the output

$$\hat{y}^{(t)} = Uh^{(t)} + b_2$$

where U is the weight matrix corresponding to the output, and b_2 is another bias term.

For a classification problem, we apply a softmax transformation to use the last hidden state $h^{(T)}$. In the language model example, this produces a probability distribution of possible next words given the context or history of previous words.

$$\hat{y} = softmax(Uh^{(T)} + b_2)$$

The RNN is commonly represented as an unrolled graph, as below in Figure 11. Note how the weights are repeatedly applied at each timestep, and how each prediction is affected by the previous states that came before it.

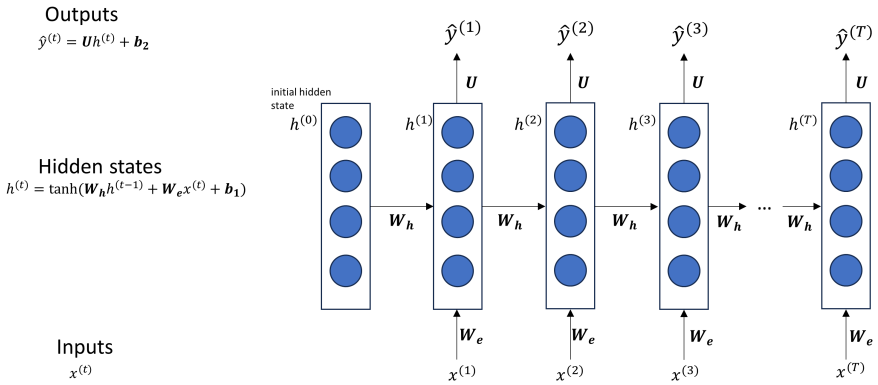

Figure 11: Graph of an unrolled RNN

A RNN can process any length input, and at timestep t, can theoretically use information from any previous timestep. Notably, the model size does not increase for longer inputs. However, recurrent computations are

slow, and in practice, it is difficult to access information from many timesteps back. In the next sections, we will discuss how to address these drawbacks.

7.1.1 Training a RNN

At each time t, we can calculate the loss for that timestep. For example, if we are doing a timeseries prediction problem on MSE loss, we can write for the timestep t

$$J^{(t)}(\theta) = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i^{(t)} - y_i^{(t)})^2$$

We can average losses overall all T timesteps to get the overall loss

$$J(\theta) = \frac{1}{T}\sum_{t=1}^{T} J^{(t)}(\theta)$$

To minimize the loss, we need to calculate the derivative of $J(\theta)$ with respect to the parameters of the model. For example, what is the derivative of $J^{(t)}(\theta)$ with respect to the repeated weight matrix W_h? Recall that in a computational graph, gradients sum at outward branches. That is,

$$\frac{\partial}{\partial t} f\left(x(t), y(t)\right) = \frac{\partial f}{\partial x}\frac{\partial x}{\partial t} + \frac{\partial f}{\partial y}\frac{\partial y}{\partial t}$$

Thus, the gradient with respect to a repeated weight matrix is the sum of the gradient with respect to each timestep it appears. That is,

$$\frac{\partial J^{(t)}}{\partial W_h} = \sum_{i=1}^{t} \frac{\partial J^{(t)}}{\partial W_h}\bigg|_{(i)}$$

Here, $\frac{\partial J^{(t)}}{\partial W_h}\Big|_{(i)}$ represent the gradient of the loss at time t, as it is used at time i. That is, at time i, how does changing W_h affect $J^{(t)}$? The effect is different depending on the time, so we need to consider the effect of each timestep. Applying the chain rule to compute gradients in such a manner is called "backpropagation through time." See Figure 12.

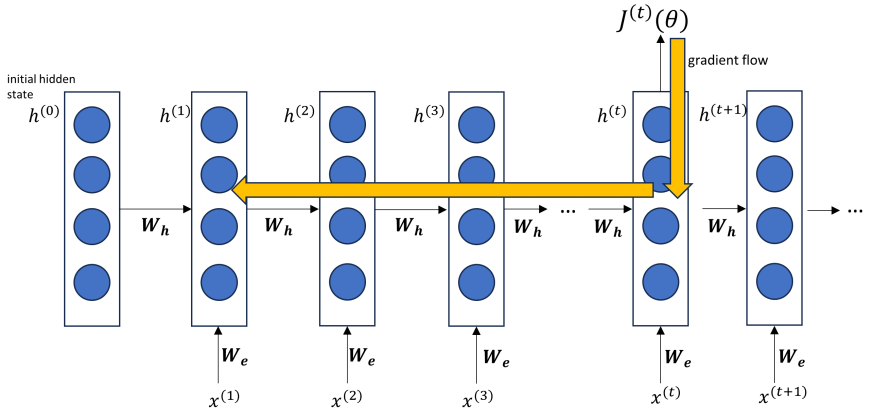

Figure 12: Backpropagation through time in a RNN

7.1.2 The Vanishing Gradient Problem

Recurrent Neural Networks suffer from the vanishing gradient problem, since the gradient signal (which instructs the model about what information to store in the hidden states) tends to vanish for longer-term dependencies. Intuitively, this is because long applications of the chain rules increase the risk of vanishing gradients, especially as we chain together derivatives that are less than 1. For example, consider the derivative of the loss at the 10th timestep with respect to the hidden state at the first timestep

$$\frac{\partial J^{(10)}}{\partial h^{(1)}} = \frac{\partial J^{(10)}}{\partial h^{(10)}} \frac{\partial h^{(10)}}{\partial h^{(9)}} \frac{\partial h^{(9)}}{\partial h^{(8)}} \frac{\partial h^{(8)}}{\partial h^{(7)}} ... \frac{\partial h^{(2)}}{\partial h^{(1)}}$$

The consequence of this is that the gradient signal from far away is lost because it is much smaller than the gradient signal close by. That is, model weights tend to be updated only with respect to near effects and not long term effects. It is too difficult for the RNN to preserve information over long periods, since in a vanilla RNN, the hidden state is constantly being rewritten at every timestep, and is not a suitable storage of long term information:

$$h^{(t)} = tanh(W_h h^{(t-1)} + W_x x^{(t)} + b_1)$$

This motivates us to consider a RNN that has a separate memory variable that can maintain information over many timesteps, which will lead us to the Long Short Term Memory (LSTM) model architecture.

As an aside, note that the vanishing gradient problem are not unique to only RNN's. Due to the chain rule, the gradient becomes vanishingly small as it backpropagates. This can be a problem for very large feed-forward or convolutional architectures. A common solution is to add more direct connections that allow the gradient to flow such as residual connections (ResNet). The figure below demonstrates ResNet or "skip-connections" (see Figure 13). In this case, the weight layers model the *residual* of the model instead of directly predicting the function from the input.

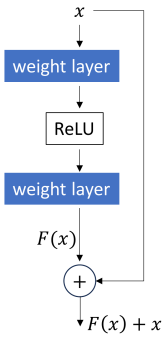

Figure 13: A residual connection

7.2 Long Short Term Memory (LSTM)

In a LSTM, every timestep t has a hidden state $h^{(t)}$ and a cell state $c^{(t)}$. The cell states store and maintain long term information, and at every timestep, the LSTM can "read", "erase", or "write" from the cell states. When the model reads from the cell state, it releases information from the cell state into the hidden state, where it is used for prediction. As new inputs arrive, the model can also write and erase from the cell state, i.e. deleting and adding information that may be useful to store for future predictions. The selection of information to be erased, written, or read is controlled by so called "gates" that can be open (value of 1), closed (value of 0), or somewhere in between. By adding a memory state, the LSTM is encouraged to learn more long-range dependencies compared to RNN, and is less susceptible to the vanishing gradient problem.

At timestep t, the first step would be to compute the gate values. The gate values all depend on the previous hidden state $h^{(t-1)}$ and the new input at the current timestep $x^{(t)}$. The sigmoid function σ ensures that the gates take values between 0 and 1:

1. The forget gate (erase), $f^{(t)}$, which controlled by parameters W_f, U_f, and b_f, controls what information is kept versus forgotten from the previous cell state. That is, it controls what information is erased at timestep t.
$$f^{(t)} = \sigma(W_f h^{(t-1)} + U_f x^{(t)} + b_f)$$
2. The input gate (write), $i^{(t)}$, controls what parts of the new input is written to the cell state at timestep t.
$$i^{(t)} = \sigma(W_i h^{(t-1)} + U_i x^{(t)} + b_i)$$
3. The output gate (read), $o^{(t)}$, controls what information from the cell we output to the hidden state at timestep t. That is, this gate controls what information is used to make a prediction at the current timestep.
$$o^{(t)} = \sigma(W_o h^{(t-1)} + U_o x^{(t)} + b_o)$$

After the gate values are computed for the current timestep, we can compute the new cell and hidden states:

1. We first represent new content using a candidate update, $\tilde{c}^{(t)}$. This is similar to the hidden state of a vanilla RNN, and is not to be confused with an LSTM hidden state. As with the hidden state of a vanilla RNN, the candidate update is a function of the previous hidden state and the new input features at the current timestep.
$$\tilde{c}^{(t)} = tanh(W_c h^{(t-1)} + U_c x^{(t)} + b_c)$$
2. We then can update the cell state, $c^{(t)}$, forgetting some content from the last cell state, and inputting some new cell content
$$c^{(t)} = f^{(t)} \odot c^{(t-1)} + i^{(t)} \odot \tilde{c}^{(t)}$$
3. Finally, some content is outputted from the cell state to the hidden state, $h^{(t)}$, where we derive the final prediction.
$$h^{(t)} = o^{(t)} \odot tanh(c^{(t)})$$

The $\odot$ symbol represents the Hadamard product (element-wise multiplication).

Once the hidden state are calculated, the final prediction can be derived depending on the task. For example, the softmax can be applied for a classification task. On the other hand, a linear layer or some feed-forward layers can produce the output for a regression task.

Overall, the LSTM architecture makes it much easier for an RNN to preserve information over many timesteps. For example, if the forget gate is 1 and the input gate is 0, information in the cell state is preserved indefinitely. In contrast, it is much harder for the vanilla RNN to learn a recurrent weight matrix W_h that preserves information in the hidden states. This behavior helps the LSTM to learn long term dependencies and circumvent the vanishing gradient problem.

As shown in Figure 14, LSTM's are commonly represented as a computational graph. This also makes it easy to see that multiple LSTM units applied over time make it easy to preserve information in the cell state, given the correct gate values (see Figure 15).

7.3 Sequence-to-sequence models

In this section, we explore how sequences can be used as both inputs and outputs using the encoder-decoder model architecture (see Figure 16). This is useful especially for input and target sequences of different lengths. Our motivating example will be a machine translation problem where we want to translate the 4 word French sentence "il a m' entarte"' into it's 6 word English translation "he hit me with a pie." The encoder-decoder models are two separate RNN's (or LSTM's) that are eventually trained simultaneously. The encoder RNN encodes all of the information from the input sentence into its final hidden state, where it is passed to the decoder RNN as its initial hidden state. In this case, the decoder RNN is a language model

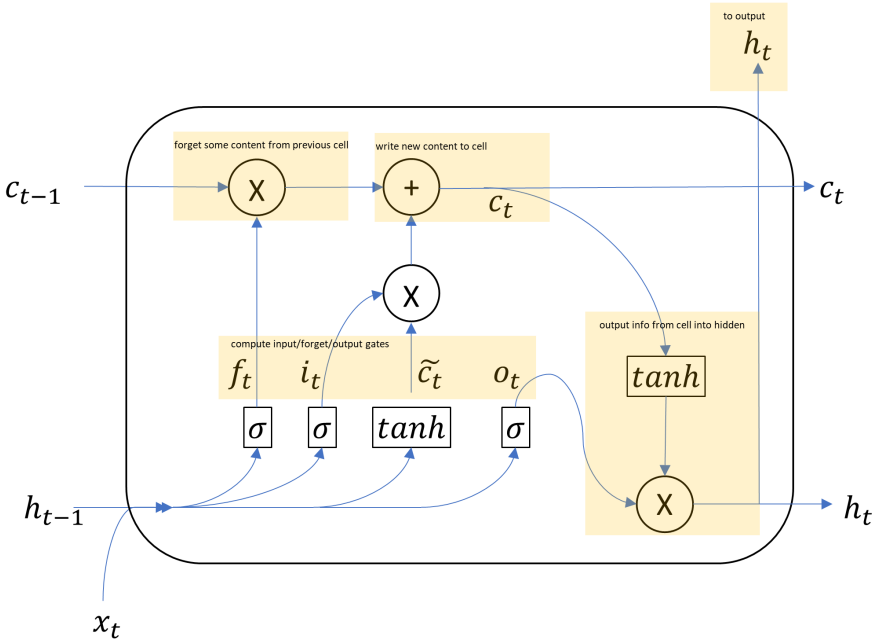

Figure 14: Computational graph for one LSTM unit

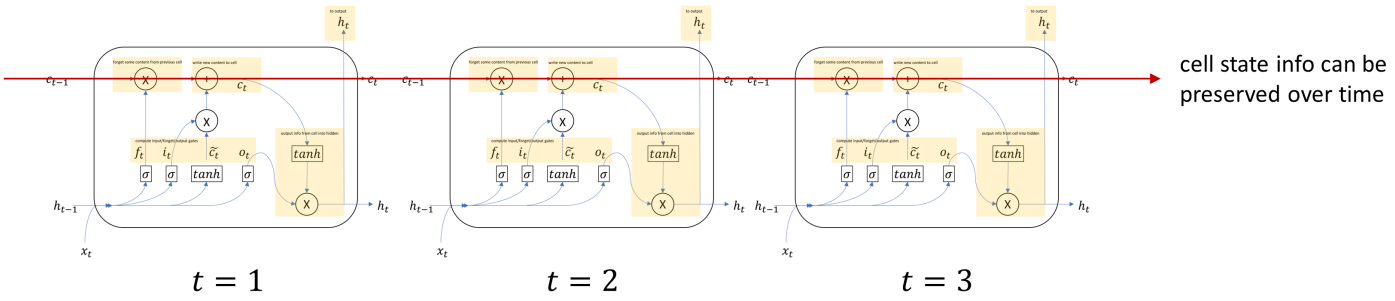

Figure 15: Demonstration of preserving cell state information over several LSTM units

that generates the target sentence given the output from the encoder RNN. As alluded to, a language model is a special natural language model that predicts the next word, given the words already predicted so far (in prediction) or the true target words so far (in training).

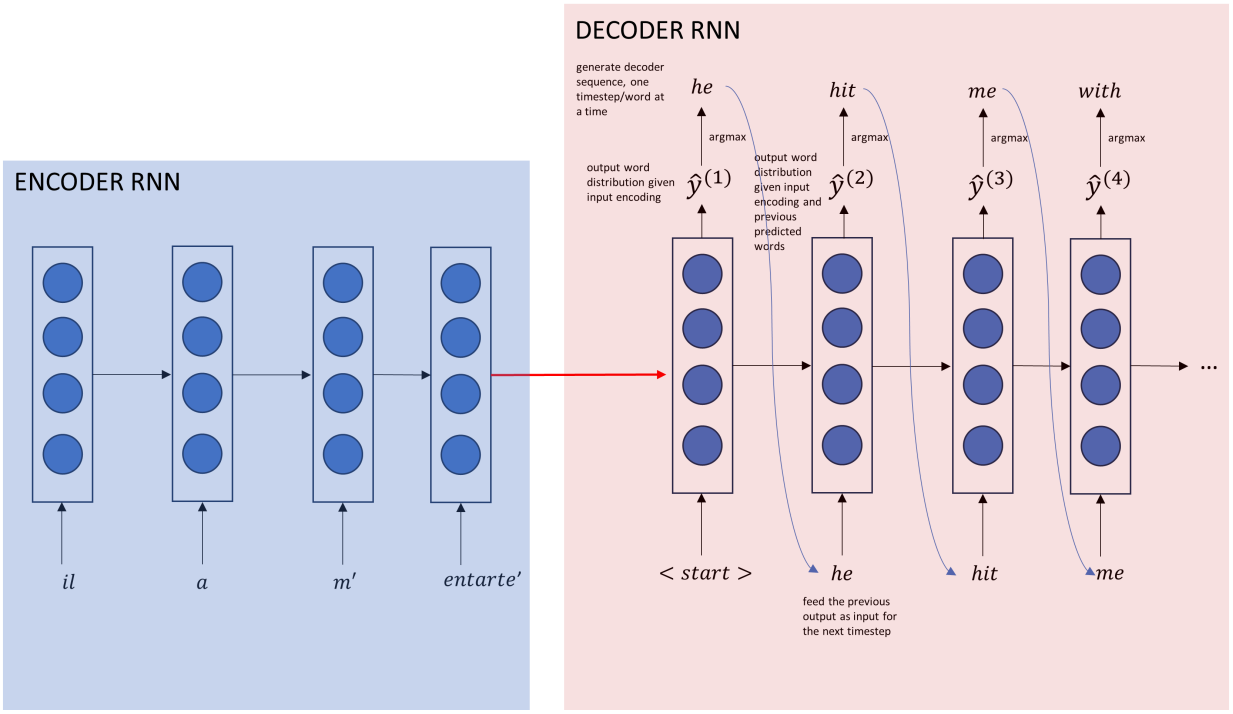

Figure 16: The encoder-decoder model as applied to a machine translation task

As an aside, note that in Figure 16 we sample from the decoder language model by taking the argmax of each output word distribution for each timestep. This is called greedy decoding, and usually does not produce the optimal result. Instead, "beam search decoding" is more commonly used, where at each timestep, we keep track of the k most probable partial translations, where k is the beam size. Typically, k is between 5 to 10 for machine translation problems.

The encoder-decoder architecture is trained simultaneously. That is, backprogagtion operates end-to-end and updates both the encoder and decoder RNN weights.

7.4 Attention

Previously, we discussed how the encoder-decoder model architecture can allow for input and target sequences of different lengths. However, one flaw of such a model is that the final hidden state of the encoder RNN needs to capture all of the information about the source sentence, creating an information bottleneck. The attention mechanism provides a solution to this problem. In attention, we use direct connections to the encoder from the decoder step to focus on a particular part of the source sentence for each step of the decoder. Figure 17 below demonstrates the attention mechanism.

Specifically, let's say we have an input sequence of length N and a target sequence of length T. Let $h_1, ..., h_N \in \mathbb{R}^h$ be the encoder hidden states and $s_t \in \mathbb{R}^h$ be the decoder hidden state for timestep t. The first step is to obtain the *attention scores*, $e_t \in \mathbb{R}^N$, for timestep t. Here, we use dot product similarity to compare the decoder hidden state with the encoder hidden states (a higher dot product means greater similarity between the hidden states).

$$e_t = [s_t^T h_1, ..., s_t^T h_N] \in \mathbb{R}^N$$

Next, we want to take the softmax to normalize the attention scores and obtain the *attention distribution*, α_t, for timestep t

$$\alpha_t = softmax(e_t) \in \mathbb{R}^N$$

Finally, we get the *attention output*, $a_t \in \mathbb{R}^h$, by taking a weighted sum of the encoder hidden states using the attention distribution.

$$a_t = \sum_{i=1}^{N} \alpha_{it} h_i \in \mathbb{R}^h$$

The attention output, a_t, and the decoder hidden state for time t, s_t, can be concatenated to proceed with the decoder step (e.g. make a prediction at timestep t).

$$[a_t; s_t] \in \mathbb{R}^{2h}$$

Figure 17 demonstrates the attention mechanism for a single decoder hidden state.

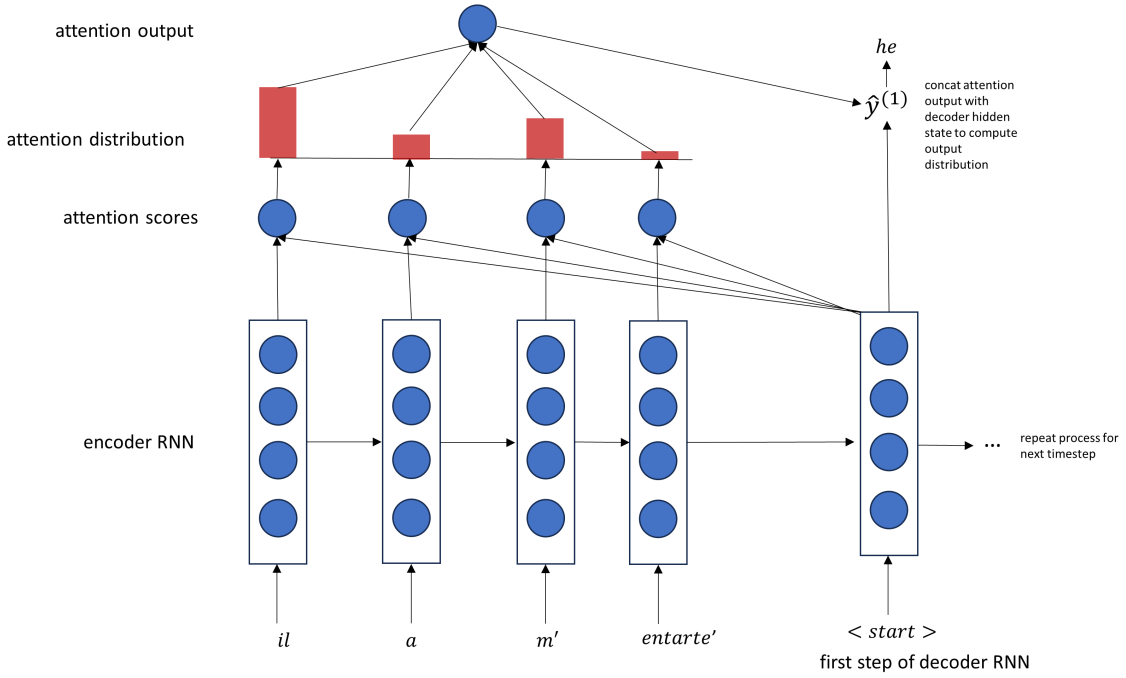

Figure 17: The attention mechanism

There are several variants of attention, but the main idea and terminology are the same. We always have *values*, *keys*, and *queries*. The attention scores are computed by comparing the *queries* and the *keys*, the attention distribution is computed by taking the softmax of the attention scores, and the attention output is obtained by taking a weighted sum of the *values*. In the dot product similarity example above, the encoder hidden states $h_1, ..., h_N$ were both the *values* and *keys*, and the decoder hidden state s_t was the *query*.

7.4.1 Other similarity measures

1. Dot product attention: $e_i = s^T h_i$. This assumes that every hidden unit has information about attention.
2. Multiplicative attention: $e_i = s^T W h_i$ where W is a weight matrix that models what parts of s and h to focus on. However, W introduces an entire matrix of extra parameters.

3. Reduced rank multiplicative attention: $e_i = s^T(U^TV)h_i = (Us)^T(Vh_i)$. Here, we model W using low rank "skinny matrices." The idea of using Us and Vh_i will be important for the Transformer architecture.

7.5 Transformer

So far, we have discussed different recurrent models and their variants. However, the biggest drawback for recurrent models is their lack of parallelizability. That is, future RNN hidden states cannot be computed in full before past hidden states have been computed. Instead of using a recurrent structure, the transformer model architecture focuses on stacking multiple (self) attention layers on top of each other. Since all words interact with each other at every layer, the number of unparallelizable operations does not increase with sequence length. Note that using attention weights for each layer is different than a fully connected layer because the weights are dynamic with the input, while in a feed-forward network, the weights are fixed after training. Figure 18 below demonstrates parallelizability for stacked attention layers.

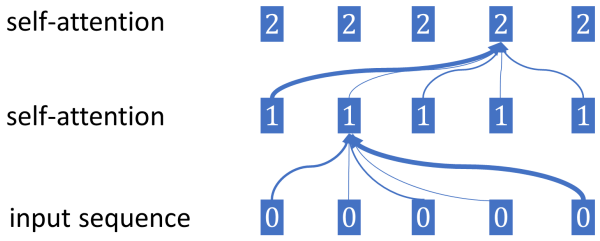

Figure 18: Parallelizability for stacked attention layers. The numbers indicate the minimum number of steps before a state can be computed.

7.5.1 Self-attention

In self-attention, the keys, queries, and values come from the same sequence, in contrast to before where we used attention to compare the encoder and decoder hidden states. As before, we will discuss Transformers in the context of language modeling, but the concepts can be applicable for many other sequence modeling tasks. Let $w_1, ..., w_n$ be a sequence of n words (as one-hot vectors). For each w_i, let x_i be its corresponding embedding where $x_i = Ew_i$ and $E \in \mathbb{R}^{dxV}$ is an embedding matrix. Here, d is the dimensionality of the embedding, and V is the vocabulary size of the model. The learned matrix E converts our input words into a dense representation, as opposed to using the one-hot words directly. In addition to dimensionality reduction, embeddings can capture semantic information, i.e., where similar words in meaning have similar embeddings. This is not true for using raw one-hot vectors.

The first step is to transform each word embedding with weight matrices $Q, K, V \in \mathbb{R}^{dxd}$ to obtain the queries, keys, and values.

$$q_i = Qx_i$$
$$k_i = Kx_i$$
$$v_i = Vx_i$$

Next, we obtain the pairwise similarities, i.e. the key-query affinities. Below, we compare word i to word j in the sequence to get attention scores.

$$e_{ij} = q_i^T k_j$$

We then normalize the attention scores with softmax to obtain an attention distribution

$$\alpha_{ij} = \frac{exp(e_{ij})}{\sum_{j'=1}^{n} exp(e_{ij'})}$$

Finally, we compute the attention output for each word i as a weighted sum of values according to the attention distribution

$$o_i = \sum_{j=1}^{n} \alpha_{ij} v_j$$

We can compute key-query-value attention in matrices. Let $X = [x_1, ..., x_n] \in \mathbb{R}^{nxd}$ be the concatenation of the whole sequence of input vectors. Note that XK, XQ, and $XV \in \mathbb{R}^{nxd}$ are the matrices containing the keys, queries, and values for each word in the sequence. Next, the query-key dot products can be represented by

$$XQK^TX^T \in \mathbb{R}^{nxn}$$

where the resulting matrix contains all pairs of attention scores. We then take the softmax (across rows) of this matrix to obtain the attention distributions, and then matrix multiply with the matrix of values to obtain the outputs.

$$output = softmax(XQK^TX^T)XV \in \mathbb{R}^{nxd}$$

7.5.2 Self-attention to replace recurrence

Our overall goal is to use attention layers stacked on top of each other as a "drop-in" replacement for recurrence. However, the naive implementation of using these stacked attention layers directly has several barriers to its success, which we will discuss one at a time.

7.5.3 Order information

The first barrier to the implementation of stacked attention layers is that a naive implementation would have no sense of sequence order. The solution to this problem is to encode position into vectors p_i for $i = 1, ..., n$ and add them to the inputs x_i:

$$\tilde{x_i} = x_i + p_i, \ i = 1, .., n$$

The original Transformer used sinusoidal position representations to create unique vectors for every index i, but a more recent approach is to let all p_i be learnable parameters. Note that both methods struggle to extrapolate to indices outside of $1, ..., n$ for a sequence of n words.

7.5.4 Adding nonlinearities

At this point, there are no nonlinearities introduced into the model, since attention outputs are simply linear combinations of the original inputs. To solve this issue, we add feedforward neural network layers in between the stacked attention layers to introduce nonlinearities by post-processing each attention output vector. See Figure 19.

$$m_i = MLP(output_i) = W_2 ReLU(W_1 output_i + b_1) + b_2$$

7.5.5 Hiding the future

In self-attention, the entire sequence interacts with each other. This a problem in sequence modeling because we cannot cheat by simply looking into the future. Note that seeing the future is not a problem in decoder RNN's since they process one word at a time in order by design. Here, we want the transformer decoder to be language model that predicts the next words (one at a time) given the *previous* predicted words, so it would defeat the purpose if the model had access to *all* target words simultaneously during training. To solve this problem, we mask out attention to future words by setting the attention scores corresponding to future timesteps to $-\infty$ (so that the future timesteps will be assigned a probability of 0 in the attention distribution). See Figure 20.

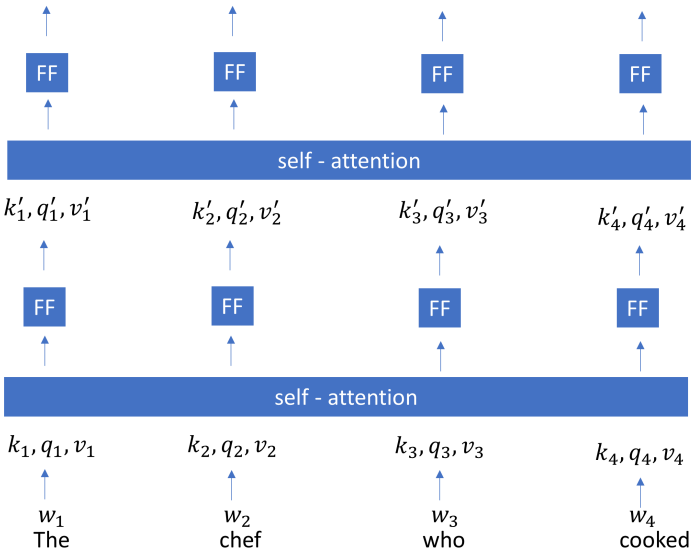

Figure 19: Stacked attention layers with feedforward networks

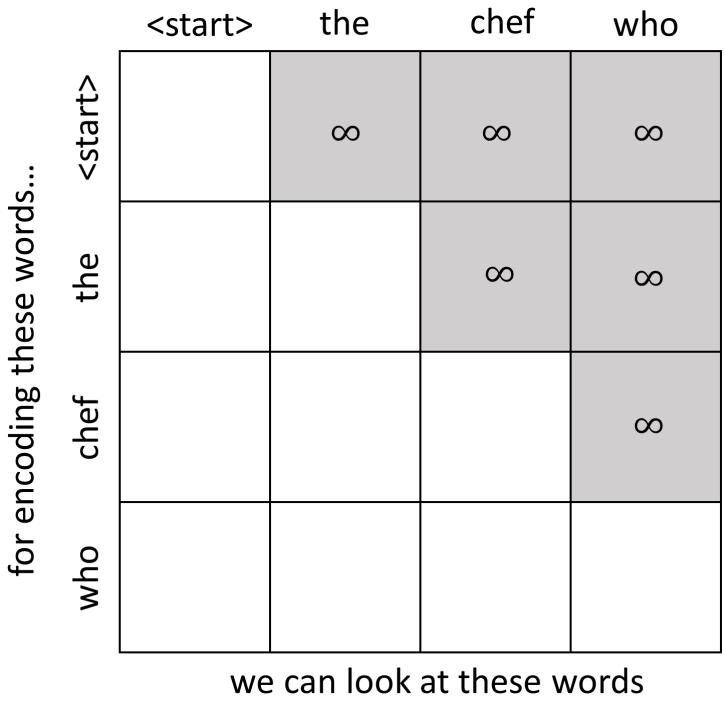

Figure 20: Transformer (decoder) masking for self attention

7.5.6 Multi-headed Attention

What if we want to look in multiple places in the sentence at once? For word i, self attention focuses on word j where $x_i^T Q^T K x_j$ is high, but what if we want to focus on different j for different reasons? To address this, we define multiple attention heads through multiple Q, K, V matrices. Each head "looks" at different things for different reasons, and constructs the value vectors differently from each other. Let $Q_\ell, K_\ell, V_\ell \in \mathbb{R}^{dx(d/h)}$ where h is the number of attention heads indexed by $\ell = 1, ..., h$. Each attention head performs self-attention independently from the others.

$$output_\ell = softmax(XQ_\ell K_\ell^T X^T) XV_\ell \in \mathbb{R}^{nx(d/h)}$$

Then, each of the outputs are combined

$$output = [output_1, ..., output_h] \in \mathbb{R}^{nxd}$$

7.5.7 Scaled dot product

In particular, when the dimensionality, d, becomes large, the dot products between vectors tend to be large. With large inputs into softmax, we run the risk of small gradients. Instead, to facilitate more stable training, we divide attention scores by $\sqrt{d/h}$ to stop attention scores from becoming large just as a function of d/h (that is, the embedding dimensionality / # of attention heads). That is, for the ℓth attention head, the attention output is

$$output_\ell = softmax\left(\frac{XQ_\ell K_\ell^T X^T}{\sqrt{d/h}}\right) XV_\ell$$

7.5.8 Residual connections

The Transformer architecture includes several residual connections to facilitate gradient flow. Recall that with residual connections (Figure 13), instead of modeling the output directly as a function of the inputs, we only model the residual from the previous layer. Residual connections allow the gradient to flow even if the gradient in the layer itself vanishes.

7.5.9 Layer normalization

The idea of layer normalization is to cut down on uninformative variation in hidden vector values by normalizing them to unit mean and standard deviation within each layer. Let $x \in \mathbb{R}^d$ be an individual hidden (word) vector in the model. Let $\mu = \frac{1}{d}\sum_{j=1}^d x_j$ and $\sigma = \sqrt{\frac{1}{d}\sum_{j=1}^d (x_j - \mu)^2}$. Then we compute

$$\frac{x - \mu}{\sigma + \epsilon} * \gamma + \beta$$

where ϵ, γ, β are optional bias parameters.

7.5.10 Cross-attention

Information from the encoder is passed to the decoder through cross attention. Let $h_1, ..., h_n$ be output vectors from the transformer encoder where $h_i \in \mathbb{R}^d$. Let $z_1, ..., z_n$ be input vectors from the transformer decoder where $z_i \in \mathbb{R}^d$. Keys and values are taken from the encoder, and queries are taken from the decoder.

$$k_i = Kh_i$$
$$v_i = Vh_i$$
$$q_i = Qz_i$$

7.5.11 Transformer architecture

Putting the above factors together, we complete the Transformer architecture as shown in Figure 21.

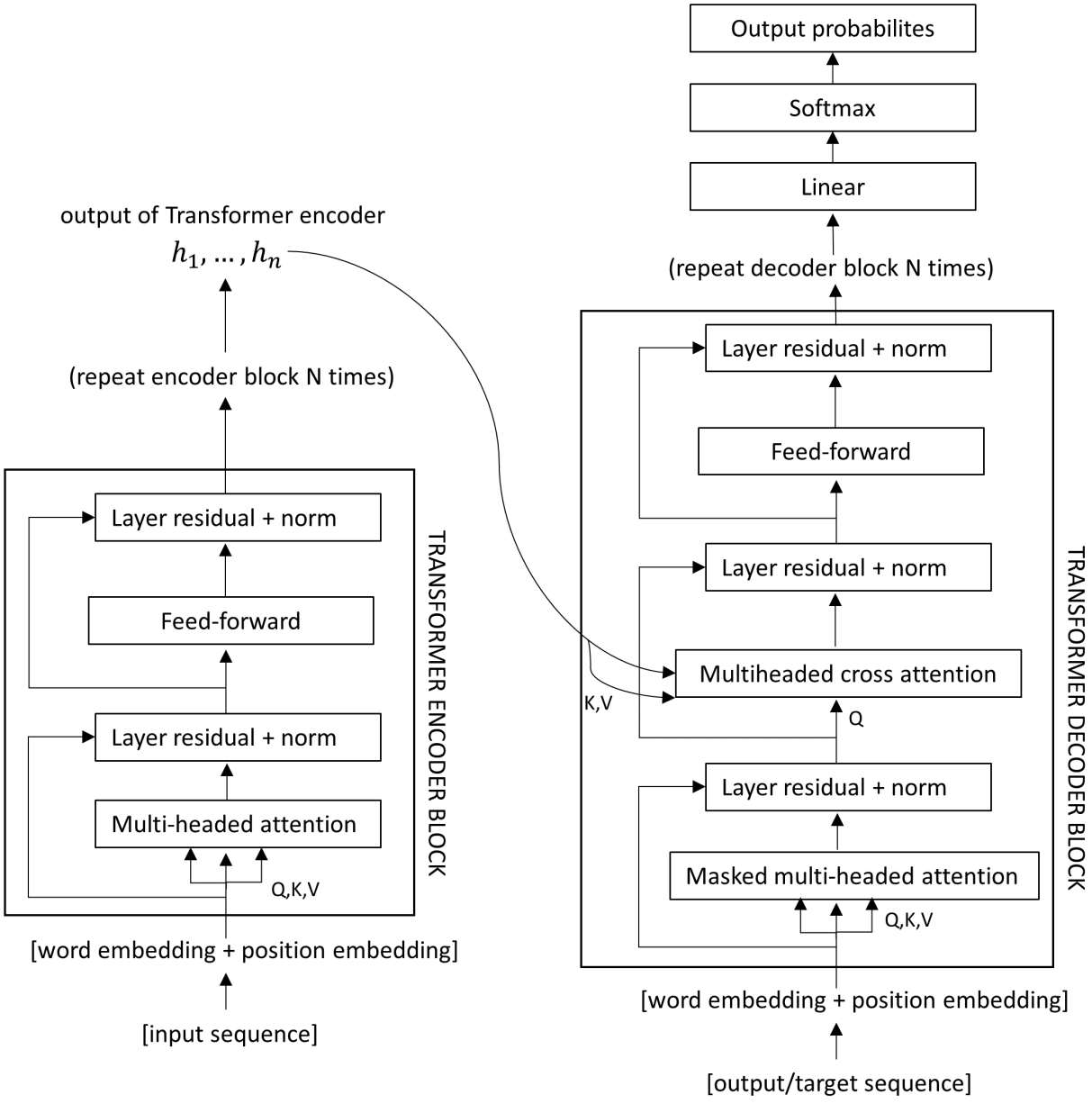

Figure 21: The Transformer architecture

8 Unsupervised Learning

Let $\mathbb{P}(X, Y)$ represent the joint probability distribution of the data. In supervised learning, we are concerned with the conditional probability distribution of Y given a specific value of $X = x$. That is, we want to learn about $\mathbb{P}(Y|X = x)$ with a labelled dataset. In the supervised setting, we can learn about $\hat{y}$ since the true y is known in the training dataset.

The goal of unsupervised learning, however, is different. Here, there is no knowledge of the "true values." Instead, we wish to learn about the properties of $\mathbb{P}(X, Y)$ directly. Under this framework, we often do not have a "target" variable as in supervised learning, so we simplicity we can simply denote $\mathbb{P}(X)$ with X representing all of the data.

For example, in dimensionality reduction techniques (PCA, self-organizing maps, etc...), the goal is to learn lower dimensional manifolds that represent the data. In clustering analysis, the goal is to identify regions that contain the modes of $\mathbb{P}(X)$. Gaussian Mixture Models, for example, answer the question if $\mathbb{P}(X)$ can be described by a mixture of simpler Gaussian densities. Association Rule Mining, given a large and sparse binary data, finds simple descriptions that describe areas of high data density.

8.1 Association Rules

In association rule mining, the goal is to find values of $X \in \mathbb{R}^p$ that jointly have a relatively large probability with respect to the density $\mathbb{P}(X)$. However, searching for specific values of X is intractable, so we focus on finding regions s_j for each X_j with relatively high probability. That is, supposing that X contains p variables, we are looking for $s_1, ..., s_p$ where the following is relatively large

$$\mathbb{P}\left[\bigcap_{j=1}^{p}(X_j \in s_j)\right]$$

In Market Basket Analysis, this can be simplified further, considering only binary data (that is after one-hot-encoding the dataset, for example). Suppose we have K binary variables, represented by $Z_1, ..., Z_K$. This is particularly useful for finding patterns in very large commercial datasets. For example, consider a tabular dataset where the columns are dummy variables representing a grocery store's catalog, and the rows represent the purchases of different customers and we want to know what items are frequently purchased together (hence the name, market basket analysis). In this situation, we can further simplify the above expression considering an "item set", which we denote $\mathcal{K}$. $\mathcal{K}$ is a set of integers representing which dummy variables are in our item set, $\mathcal{K} \subset \{1, ..., K\}$. Then we are searching for item sets where the following probability is relatively large

$$\mathbb{P}\left[\bigcap_{k\in\mathcal{K}}(Z_k = 1)\right] = \mathbb{P}\left[\prod_{k\in\mathcal{K}} Z_k = 1\right]$$

We can estimate this probability using the fraction of observations in the data that contains the item set $\mathcal{K}$. Let z_{ik} denote the outcome of the dummy variables Z_k for the ith data observation. We have

$$\hat{\mathbb{P}}\left[\prod_{k\in\mathcal{K}} Z_k = 1\right] = \frac{1}{n}\sum_{i=1}^{n}\prod_{k\in\mathcal{K}} z_{ik}$$

The above quantity is called the support of the item set $\mathcal{K}$, which we can denote as $T(\mathcal{K})$. The support represents the prevalence of a particular item set in the data. In association rule mining, we are searching for all item sets that have a support greater than some threshold t. That is, we are searching for the set

$$\{\mathcal{K}_\ell | T(\mathcal{K}_\ell > t\}$$

This problem can efficiently be solved using the Apriori Algorithm, with only a few passes of the data needed (with sufficiently sparse data). In the first pass, we consider item sets of size 1, and discard all sets that do not satisfy the support threshold. In the second pass, we consider item sets of size 2, using only only the survivors of the first pass, and so on for the subsequent passes. Then, each high support item sets resulting from the Apriori algorithm is broken into two disjoint subsets $A \cup B = \mathcal{K}$ where we can write the association rule $A \rightarrow B$. A is called the "antecedent" and B is called the "consequent."

The "confidence" of the association rule, which we denote as $C(A \rightarrow B)$ can be written as

$$C(A \rightarrow B) = \frac{T(A \rightarrow B)}{T(A)} = \frac{T(\mathcal{K})}{T(A)}$$

This is an estimate of the conditional probability measure $\mathbb{P}(B|A)$.

Finally, we can define the "lift" of the association rule, which is an estimate of $\mathbb{P}(A \cap B)/\mathbb{P}(A)\mathbb{P}(B)$. Lift is defined as

$$L(A \rightarrow B) = \frac{C(A \rightarrow B)}{T(B)}$$

The overall goal of this analysis is to find all association rules with both high support and high confidence. That is, after finding high support item sets using the Apriori Algorithm, we can to return

$$\{A \rightarrow B | C(A \rightarrow B) > c\}$$

for some confidence threshold c.

8.2 K-means Clustering

In cluster analysis, datapoints are assigned to clusters such that the points inside the cluster are more similar than the points in the other clusters. In k-means clustering, the datapoints are assigned to clusters with the nearest mean, and seeks to minimize the within-cluster (unexplained) variance.
Given k different clusters/sets $S = \{S_1, ..., S_k\}$, the goal is to solve for S such that the within-cluster sum of squares is minimized

$$\underset{S}{\operatorname{argmin}} \sum_{i=1}^{k} \sum_{x \in S_i} ||x - \mu_i||_2^2$$

where $\mu_i = \frac{1}{|S_i|} \sum_{x \ \in S_i} x$ are the cluster means.

This problem is equivalent to maximizing the between cluster sum of squares since total variance is constant for the dataset (recall sum of square decomposition from linear regression).

Unfortunately, finding the optimal solution to the problem as stated above is computationally difficult. Commonly, heuristic algorithms are used to find a local solution. For example, in Lloyd's algorithm the following process is repeated until the cluster assignments have converged. We begin with randomly assigned cluster assignments are repeat

1. Calculate means (centers) for each assigned cluster
2. Update cluster assignments by re-assigning points to the closest mean

8.3 Mixture Models and the EM Algorithm

Mixture models are useful when data is randomly drawn from several different distributions. For example, we can write the data generating process for x as a mixture of K different clusters/distributions

$$x \sim p_\theta(x) = \sum_{j=1}^{K} p_\theta(Z=j)p_\theta(x|Z=j) = \sum_{j=1}^{K} p_\theta(x, Z=j)$$

where Z represents the cluster assignment and θ represents the model parameters. $p_\theta(Z=j)$ are called the mixture proportions. In many problems, we face a situation where the cluster assignments and mixture proportions are unknown.

We have that the maximum likelihood estimator (MLE) for θ over n data observations is

$$\hat{\theta} = \underset{\theta}{\text{argmax}} \sum_{i=1}^{n} log p_\theta(x_i) = \underset{\theta}{\text{argmin}} \sum_{i=1}^{n} log \sum_{j=1}^{K} p_\theta(x_i, Z_i = j)$$

which is a difficult non-convex optimization problem. We turn to the EM algorithm, which is a non-convex optimization strategy that is designed to solve for the MLE in problems with missing or unobserved data (such as unknown cluster assignments). We can rewrite the log-likelihood by introducing a new pmf denoted by q_i

$$\begin{aligned}
\ell(\theta) &= \sum_{i=1}^{n} log \sum_{j=1}^{K} p_\theta(x_i, Z_i = j) \\
&= \sum_{i=1}^{n} log \sum_{j=1}^{K} q_i(Z_i = j) \frac{p_\theta(x_i, Z_i = j)}{q_i(Z_i = j)} \\
&= \sum_{i=1}^{n} log\ \mathbb{E}_{q_i} \frac{p_\theta(x_i, Z_i)}{q_i(Z_i)}
\end{aligned}$$

By Jensen's Inequality, that is $\mathbb{E}f(X) \geq f(\mathbb{E}X)$ for convex f, we have that

$$\begin{aligned}
\ell(\theta) &\geq \sum_{i=1}^{n} \mathbb{E}_{q_i} log \frac{p_\theta(x_i, Z_i)}{q_i(Z_i)} \\
&= \sum_{i=1}^{n} \sum_{j=1}^{K} q_i(Z_i = j) \frac{p_\theta(x_i, Z_i = j)}{q_i(Z_i = j)} \\
&= \sum_{i=1}^{n} \sum_{j=1}^{K} q_i(Z_i = j) log\ p_\theta(x_i, Z_i = j) - q_i(Z_i = j) log\ q_i(Z_i = j)
\end{aligned}$$

We define the first term as

$$Q(\theta) := \sum_{i=1}^{n} \sum_{j=1}^{K} q_i(Z_i = j) log\ p_\theta(x_i, Z_i = j)$$

noting that the second term does not depend on θ and is not needed for the optimization problem.

The question remains on how to chose $q_i(Z_i)$ so that the bound of the inequality is as tight as possible. If we choose

$$\begin{aligned}
q_i(Z_i = k) &= \frac{p_{\theta^{(t)}}(x_i, Z_i = k)}{\sum_{j=1}^{K} p_{\theta^{(t)}}(x_i, Z_i = j)} = \frac{p_{\theta^{(t)}}(x_i | Z_i = k) p_{\theta^{(t)}}(Z_i = k)}{\sum_{j=1}^{K} p_{\theta^{(t)}}(x_i | Z_i = j) p_{\theta^{(t)}}(Z_i = j)} \\
&= p_{\theta^{(t)}}(Z_i = j | x_i)
\end{aligned}$$

where $\theta^{(t)}$ represents the t^{th} iteration for θ of the algorithm, then the bound from Jensen's Inequality becomes

an equality when $\theta^{(t)} = \theta$ since the quantity

$$\begin{aligned}\frac{p_\theta(x_i, Z_i)}{q_i(Z_i)} &= \frac{p_\theta(x_i, Z_i)}{p_\theta(x_i, Z_i)/\sum_{j=1}^K p_\theta(x_i, Z_i = j)} \\ &= \sum_{j=1}^K p_\theta(x_i, Z_i = j)\end{aligned}$$

is deterministic.

The EM algorithm consists of iterating the following steps

1. E-step (Expectation): Solve for $p_{\theta^{(t)}}(Z_i = j|x_i)$ for all $j = 1, ..., K$ using the current parameters $\theta^{(t)}$
2. M-step (Maximization): Update the parameters by solving maximizing $Q(\theta)$

$$\theta^{(t+1)} = \underset{\theta}{\text{argmax}} \sum_{i=1}^n \sum_{j=1}^K p_{\theta^{(t)}}(Z_i = j|x_i) log\ p_\theta(x_i, Z_i = j)$$

Note that the M-step often has an analytical solution and is easy to compute.

8.3.1 Gaussian Mixture Model

Consider the following data generating process

$$x \sim p_\theta(x) = \sum_{j=1}^K \pi_j \mathcal{N}(x; \mu_j, \Sigma_j)$$

where $\pi_j = p_\theta(Z = j)$ and the unknown parameters are

$$\theta = (\pi_1, ..., \pi_K, \mu_1, ..., \mu_K, \Sigma_1, ..., \Sigma_K)$$

We now apply the EM Algorithm. In the E-step, we have by Bayes's Rule that

$$p_{\theta^{(t)}}(Z_i = j|x_i) = \frac{\mathcal{N}(x_i; \mu_j^{(t)}, \Sigma_j^{(t)})\pi_j^{(t)}}{\sum_{j'=1}^K \mathcal{N}(x_i; \mu_{j'}^{(t)}, \Sigma_{j'}^{(t)})\pi_{j'}^{(t)}}$$

and that

$$Q^{(t)}(\theta) = \sum_{i=1}^n \sum_{j=1}^K p_{\theta^{(t)}}(Z_i = j|x_i) log\left(\mathcal{N}(x_i; \mu_j, \Sigma_j)\pi_j\right)$$

In the M-step, we solve for

$$\theta^{(t+1)} = \underset{\theta}{\text{argmax}} Q(\theta)$$

and obtain the following updates

$$\begin{aligned}\pi_j^{(t+1)} &= \frac{1}{n}\sum_{i=1}^n p_{\theta^{(t)}}(Z_i = j|x_i) \\ \mu_j^{(t+1)} &= \frac{\sum_{i=1}^n p_{\theta^{(t)}}(Z_i = j|x_i)x_i}{\sum_{i=1}^n p_{\theta^{(t)}}(Z_i = j|x_i)} \\ \Sigma_j^{(t+1)} &= \frac{\sum_{i=1}^n p_{\theta^{(t)}}(Z_i = j|x_i)(x_i - \mu_j^{(t+1)})(x_i - \mu_j^{(t+1)})^T}{\sum_{i=1}^n p_{\theta^{(t)}}(Z_i = j|x_i)}\end{aligned}$$

In practice, the Gaussian Mixture Model can be used as a soft clustering algorithm. That is, clusters can be assigned for each datapoint x_i by comparing the conditional probabilities $p_{\theta^{(t)}}(Z_i = j|x_i)$ for $j = 1, .., K$.

By doing so, we also have the benefit of built-in uncertainty estimates.

Additionally, clustering using the Gaussian Mixture Model has a more flexible structure compared to K-means clustering due to the flexibility in the covariance parameters Σ_j and allows for ellipsoid clusters. In contrast, K-means clustering only models spherical clusters.

8.4 Principal Component Analysis (PCA)

Consider a random vector $x \in \mathbb{R}^p$ with mean μ and covariance matrix Σ. If x is high-dimensional (p is large), we may be interested in representing x in $k < p$ components with the goal of explaining as much variance in x as possible using these k components. Principal Components Analysis (PCA) gives us the best possible *linear* approximation of x using k uncorrelated components. For the special case when x is multivariate Gaussian, i.e. $x \sim \mathcal{N}(\mu, \Sigma)$, PCA gives us the best possible approximation of x using k independent components (recall that uncorrelated Gaussian random variables are independent).

Assume without loss of generality that x has been scaled to have zero mean. We begin by finding a *linear* combination of the elements of x that has maximal variance, expressed as the vector dot product $v^T x$ where $v \in \mathbb{R}^p$. Recall that the variance of $v^T x$ is $v^T \Sigma v$. So among solutions where $||v||_2 = 1$, we want to solve

$$\underset{v,\ ||v||=1}{\text{argmax}}\ v^T \Sigma v$$

Writing the Lagrangian, we have

$$\mathcal{L}(v, \lambda) = v^T \Sigma v - \lambda(v^T v - 1)$$

where $\lambda \in \mathbb{R}$ is the Lagrange multiplier. Then taking the gradient of $\mathcal{L}(v, \lambda)$ and setting it to zero we arrive at the solution

$$\begin{aligned} \nabla_v \mathcal{L}(v, \lambda) &= 2\Sigma v - 2\lambda v = 0 \\ \Sigma v &= \lambda v \end{aligned}$$

This can be recognized as an eigenvector equation for the matrix Σ. Note that we have variance

$$v^T \Sigma v = v^T(\lambda v) = \lambda v^T v = \lambda$$

Thus, to maximize the variance quantity, we want to choose $\lambda = \lambda_1$ as the largest eigenvalue of Σ and $v = e_1$, the first eigenvector of Σ. The product $z_1 = e_1^T x$ is known as the *first principal component*, and we just showed that the variance of the first principal component is λ_1

To find the second principal component (and so on) we want to search for v so that it is orthogonal to e_1, that is, so that the first and second principal components are uncorrelated. Solving for

$$\underset{v,\ ||v||=1,\ v^T e_1 = 0}{\text{argmax}}\ v^T \Sigma v$$

we find that $v = e_2$, the second eigenvector of Σ and that the variance of the second principal component is λ_2. Similarly, each subsequent $i \in \{1, ..., p\}$ principal component has variance λ_i and direction e_i.

In general, we can express all p principal components as $z \in \mathbb{R}^p$

$$z = Q^T(x - \mu)$$

where $Q \in \mathbb{R}^{p,p}$ is a matrix with the eigenvectors of Σ in its columns. Recall that since Σ is symmetric and square, it will have p orthogonal eigenvectors.

We can interpret PCA as an orthogonal change of basis so that the data in the new coordinate system is uncorrelated and in order of decreasing variance (for example, we can write $Ix = Qz$ where I and Q contain

in their columns the old and new basis vectors, respectively, to represent the transformation $z = Q^T x$). With properly chosen eigenvectors (they are only uniquely defined up to sign), this transformation is a rotation of the original canonical basis.

We can see that the new data $z = Q^T x$ is uncorrelated (again assuming that x has been scaled to zero mean) since

$$\begin{aligned}\Sigma_z &= \mathbb{E}zz^T = \mathbb{E}(Q^T x)(Q^T x)^T \\ &= Q^T \mathbb{E}xx^T Q = Q^T \Sigma_x Q \\ &= Q^T (Q \Lambda Q^T) Q = \Lambda\end{aligned}$$

where $\Lambda = diag(\lambda_1, ..., \lambda_p)$ has zero off-diagonal values.

To understand dimensionality reduction, we represent the total variance of x as

$$Tr(\Sigma) = Tr(Q\Lambda Q^T) = Tr(\Lambda Q Q^T) = Tr(\Lambda) = \sum_{i=1}^{p} \lambda_i$$

and the proportion of variance explained by selecting $k < p$ principal components is

$$\sum_{i=1}^{k} \lambda_i / \sum_{i=1}^{p} \lambda_i$$

In practice, we conduct PCA on the sample covariance matrix instead of the population Σ.