1 Model Types and Algorithms

1.1 Models Types and Accuracy

Supervised vs. Unsupervised Learning

Supervised learning is often referred to as *learning with training.* Given data X, y, we want to fit the mapping between X and y with training data and make predictions with the new y given new X in the test dataset. The ultimate goal is to minimize the distance between $\hat{y}$, the predicted value, and y.

Unsupervised learning is learning without training. This time, there is no response y to be predicted. Instead, we *explore* the pattern of data using methods such as clustering or dimension reduction.

Regression vs. Classification

In supervised machine learning models, regression is often used with numerical data and classification is used with categorical data (consists of multiple classes or levels).

Parametric vs. Non-parametric models

Parametric machine learning models assume that the function can be modeled in a functional form and follows a procedure to *fit* and *train* the model. In linear regression, the functional form is a linear combination of unknown parameters and our predictors.

Non-parametric machine learning algorithms, in contrast, does not explicitly assume a functional form is possible and limits the need for finding parameters. Instead, it uses patterns and trends in existing training data to predict results in the test data. The K-nearest neighbors algorithm is just one example. Test points are compared to a set amount of training data points that are closest, or similar, to it.

Measuring accuracy in machine learning models

The standard measurement of accuracy is computing the mean squared error (MSE). If y_i and x_i are observations in training data and $\hat{f}$ is the *function* described by the model, then

$$\text{Training MSE} = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - \hat{f}(x_i)\right)^2.$$

This is the function to be minimized. Note that we write $y_i = \hat{f}(x_i) + \varepsilon_i$, where ε_i is the error/noise caused by x_i. The MSE for the test data can be expressed as

$$\text{Test MSE} = E[\varepsilon^2] + \underbrace{\left(f(x_0) - E_{x_i,\varepsilon}[\hat{f}(x_0)]\right)^2}_{\text{Bias}(\hat{f}(x_0))} + \underbrace{E_{x_i,\varepsilon}\left[\hat{f}(x_0) - E_{x_i,\varepsilon}[\hat{f}(x_0)]\right]^2}_{\text{Var}(\hat{f}(x_0))}.$$

Here we use properties of expectations from probability theory. The above equation is

merely showing that the test MSE is a sum of the bias, variance, and random error (often negligible).

- **Bias** is the error introduced by making assumptions to simplify the learning process. For example, using a linear model to a dataset with a nonlinear relationship is will have high bias, for it is trying to simplify but does not fit the model. This is *underfitting*.
- **Variance** is the error caused by small fluctuations in the model's test data. A model with high variance will not only capture the patterns in the training data, but the noise as well. This is *overfitting*. High variance models work well with training data but poorly on new test data.

Combining these two factors is what we know as the **Bias-Variance Tradeoff**. Increasing the bias of a linear model lowers the variance, and vice versa. Generally, we want to strike a balance between the two factors that will minimize the test MSE.

Flexible vs. Inflexible Models

Flexible models are often used to capture more complex relationships, such as decision trees or high degree polynomial regression. They can readily adapt to changes in data (i.e. new training/test data). Flexible models are more prone to overfitting and consequently having high variance.

Inflexible models follow a rigid, functional form that cannot capture complex patterns in data and assume a rigid, predefined relationship between inputs and outputs. Flexible models tend to have more bias and succumb to underfitting.

1.2 Basic Algorithms

K-Nearest Neighbors (KNN) Algorithm

The K-Nearest Neighbors method is a supervised learning algorithm that makes predictions based on how *similar* new data points are to existing data. We compute the Euclidean distance between the test data point and all of the observed data, then choosing k of the existing points that are closest in distance.

- In a classification setting, the majority class among the k neighbors determines the predicted class.
- In a regression setting, the predicted value is the average of the values of the k neighbors.

k is what we call a *hyperparamter*, or tuning parameter; it does not depending on the training process. As we increase k, the flexibility of the algorithm increases. If we choose too many neighbors, we may observe overfitting and misleading predictions.

Normal Equation for Linear Regression

Recall that a linear model with p predictors is approximated by

$$\hat{f}(x) = c_0 + c_1 x_1 + c_2 x^2 + \cdots c_p x^p.$$

Where $c_0, \cdots, c_p$ are unknown parameters. We want to find these parameters in a way that will minimize the error

$$\min_{\overline{w}} \sum_{i=1}^{n} y_i - \hat{f}(x_i).$$

$\overline{w}$ is just the set of parameters we want to solve for in the above problem. Recall that with higher degree polynomial regression (degrees of freedom > 2), it is nonlinear with respect to x but it is linear with respect to the parameters. Hence, we can apply methods from linear algebra to solve the optimization problem. The most common approach is through *gradient descent*, for which we take partial derivatives of each parameter and iterate until we find the minimum. The solution is given by the **normal equation**:

$$\widehat{W} = \left(M^T M\right)^{-1} M^T \overrightarrow{Y}.$$

If n is the number of observations in our training data, then M is a $n \times (p+1)$ matrix containing the intercept terms (column of 1s) and the predictors in our training data. $\overrightarrow{Y}$ is a $n \times 1$ vector consisting of the response values in our training data.

2 Classification and Generative Models

Previously, we assumed that we can fit a model $y = f(x) + \varepsilon$, where ε does not depend on x. Now, suppose we let the error also depend on x. Then, $y = f(x) + \varepsilon(x)$. We say that $f(x)$ is deterministic and $\varepsilon(x)$ is a random variable. The main objective is to find

$$p(y, x) \approx P(Y = y \mid X = x)$$

or, that approximation of a model relative to the probability that it predicts y given x. For the models in this section, we assume that the Y is a discrete random variable to avoid complications.

2.1 MLE and MAP estimations

Maximum Likelihood Estimation

We proceed by using a standard logistic regression model. Assume that we are given data $(\overrightarrow{x_i}, y_i)$, where Y can take on two classes: A or B. Our goal is to have our model $\hat{p}$ roughly estimate the probability that each outcome occurs. A standard linear regression model would argue that we can approximate in the form $Y = \beta_0 + \beta_1 X$. However, this can quickly violate our goal! Recall that our objective function is a probability function, so the range of values must be $0 < P(Y = y \mid X = x) < 1$. This is where we introduce logistic regression: a model that takes on the form

$$\hat{p}(Y \in A \mid X = x) = \frac{e^{\beta_0 + \beta_1 x}}{1 + e^{\beta_0 + \beta_1 x}}.$$

This so-called *standardized* form alleviates the negativity issue and forces all outputs to be within 0 and 1. Now that we have developed a probability function, we hit another wall. How can we ensure that we achieve, or get close to, the observations $(\overrightarrow{x_i}, y_i)$ under our new model? The idea is to *maximize* the probability that we get said observations. This is where we introduce the **likelihood function**:

$$\mathcal{L}\left(\beta_0, \overrightarrow{\beta}\right) = \prod_{i=1}^{n} P(Y = y_i \mid X = x_i) = \begin{cases} \hat{p}(x) & \text{when } y \in A \\ 1 - \hat{p}(x) & \text{when } y \in B \end{cases}$$

where n observations are given and assumed that each observation is independent and $\beta_0, \overrightarrow{\beta}$ contains the coefficients $\beta_1, \cdots, \beta_n$. We can rewrite this as the product of probabilities that an observation falls into each class:

$$\mathcal{L}\left(\beta_0, \overrightarrow{\beta}\right) = \prod_{i : y_i \in A} \hat{p}(x_i) \prod_{i : y_i \in B} (1 - \hat{p}(x_i)).$$

Finding such $\beta_0, \overrightarrow{\beta}$ that maximizes $\mathcal{L}$ is the **Maximum Likelihood Estimator (MLE)**.

Example 2.1. Suppose we are given a fair die and we roll it four times with the observed outcomes 5, 2, 3, 6. Let θ be the probability of rolling a 3. What is the Maximum Likelihood Estimation for θ?
Let $P(3) = \theta$ and $P(\text{Not } 3) = 1 - \theta$. Then,

$$f(\theta) = \theta(1-\theta)^3.$$

We take the natural log of f, differentiate, and set equal to zero to find the corresponding θ:

$$\ln(f(\theta)) = \ln(\theta) + 3\ln(1-\theta) \implies \frac{d}{d\theta}\ln(f(\theta)) = \frac{1}{\theta} + \frac{3}{1-\theta} = 0 \iff \theta = \frac{1}{4}.$$

One could argue that this is indeed the maximum through the second derivative test.

Maximum A Posteriori (MAP) Estimation

Recall Bayes' Formula from probability theory. If A and B are both events, each assigned their own probability, then the conditional probability

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

Here $P(A|B)$ is the *posterior probability*, or the probability obtained after we take our data into consideration. $P(A)$ is the *prior*, or the probability before we take our data into consideration. If we are given prior knowledge or data, then we can try to *maximize* Bayes' Theorem, or the posterior probability. If θ is the parameter of interest and X is our observed data, then

$$\underset{\theta}{\text{argmax}}\ P(X|\theta)P(\theta).$$

This is the **Maximum A Posteriori Estimation**. Since we are maximizing with respect to θ, we treat $P(X)$ as a constant and can ignore it.

Example 2.2. Use the same setup from the previous example. Given the priors 0.8, 0.45, 0.1, 0.04 for $\theta = 0.5, 0.3, 0.7, 0.6$ respectively, compute the Maximum A Posteriori Estimation for θ. Compute the posteriori for each prior θ (where $P(X|\theta) = f(\theta)$ from past example) and determine which is the largest:
$\theta = 0.5 \implies (0.5)^4(0.8) \approx 0.08$
$\theta = 0.3 \implies 0.3(0.7)^3(0.45) \approx 0.0463$
$\theta = 0.7 \implies 0.7(0.3)^3(0.1) \approx 0.00189$
$\theta = 0.6 \implies 0.6(0.4)^3(0.04) \approx 0.001536$
The Maximum A Posteriori is therefore 0.08 for when $\theta = 0.05$.

Similar to MLE, we can compute MAP by taking the natural log and differentiating to easily obtain the maximum.

For logistic models, we choose an objective MLE (not MAP!) function, optimize, and apply new inputs to predict new samples $\hat{p}\left(Y|\overrightarrow{X} = X_0\right)$ directly.

2.2 Generative Models

Naive Bayes, LDA, QDA

Compared to logistic regression, generative models are used to learn how the data itself is distributed through training the data and then testing the model on new data. Hence, they aim to learn the joint probability distribution $P(X, Y)$ (supervised case) or $P(X)$ (unsupervised case). For now, we focus on three generative models: Naive Bayes and Linear/Quadratic Discriminant Analysis.

As given by its name, the Naive Bayes' algorithm is derived from Bayes' formula. For Naive Bayes, the underlying assumption is that the observations in *each class* is independent of each other. Thus, we apply the basic law of probability $P(A \cap B) = P(A)P(B)$. Let $\overrightarrow{X}$ be the vector containing our features and let $\left(X^j|Y = k\right)$ be the j-th observation in the k-th class. Then, the Naive Bayes' formula is given as

$$\hat{p}\left(\overrightarrow{X}|Y = k\right) = \prod_j \hat{p}\left(X^j \mid Y = k\right).$$

The expression itself says that the probability that a sample falls into a class is computed as the product of probabilities for each feature within that class. We also say that this conditional probability is directly proportional to the prior of the class. Generally, Naive Bayes is put to quick use for simple classification problems, such as spam/fraud detection, or recommendation systems.

Now, we consider two other algorithms that rely on a different distribution than Naive Bayes. To motivate this idea, we once again consider Bayes' Theorem. Suppose we are given data $\{(x_i, y_i)\}_{i=\{1,\cdots,N\}}$ and we want to classify x_0 into k classes. Then we compute

$$P[Y = k|X = x] = \frac{P[Y = k]P[X = x|Y = k]}{P[X = x]}.$$

Suppose we cannot assume that the observations in each class are independent.

- We know how to compute $P[Y = k]$ within a dataset.
- How do we find $P[X = x|Y = k]$?

While there is no definitive answer, the easiest (and most used) approach is to make an assumption about the type of distribution that it follows. Let $P[X = x|Y = y] = f_k(x)$ and say that $f_k(x)$ follows a normal Gaussian distribution with mean μ_k (mean parameter for

k-th class) and variance σ_k^2 (variance for k-th class). Then,

$$f_k(x) = \frac{1}{\sqrt{2\pi}\sigma_k} e^{\frac{(x-\mu_k)^2}{2\sigma_k^2}}.$$

We can summarize the behavior of this distribution by writing $f_k(x) \sim \mathcal{N}\left(\mu_k, \sigma_k^2\right)$.

As for $P[X = x]$, we multiply the probability that an observation lies in the l-th class (for $1 \leq i \leq k$) by the probability that X takes on a certain outcome x given that it is in the l-th class. Then, you sum this across all k classes. Mathematically, this is expressed as

$$P[X = x] = \sum_{l=1}^{k} P[Y = l]P[X = x|Y = l] \quad \text{(weighted average)}.$$

Before proceeding, let us clean up with some more notation. Let $p_k(x) = P[Y = k|X = x]$ and $\pi_k = P[Y = k]$ (prior). Notice how we can rewrite $P[X = x]$ using normal distributions as well:

$$P[X = x] = \sum_{l=1}^{k} \pi_l \cdot \frac{1}{\sqrt{2\pi}\sigma_l} e^{\frac{(x-\mu_l)^2}{2\sigma_l^2}}.$$

This enables us to obtain the final form of $p_k(x)$:

$$p_k(x) = \frac{\pi_k \cdot \frac{1}{\sqrt{2\pi}\sigma_k} e^{\frac{(x-\mu_k)^2}{2\sigma_k^2}}}{\sum_{l=1}^{k} \pi_l \cdot \frac{1}{\sqrt{2\pi}\sigma_l} e^{\frac{(x-\mu_l)^2}{2\sigma_l^2}}}.$$

Now that we have a closed-form expression, we want to optimize it. That is to say, we want to maximize the posterior probability, or probability that we put a sample in the k-th class given the features in X. Once again, logarithms come to the rescue! Define $\delta_k(x) = \ln(p_k(x))$. Then,

$$\delta_k(x) = -\frac{1}{2\sigma_k}\left(x - \mu_k\right)^2 + \ln(\pi_k) + \ln\left(\frac{1}{\sqrt{2\pi}\sigma_k}\right).$$

We seek to find the k that solves the optimization problem

$$\underset{k\in\{1,\cdots,K\}}{\operatorname{argmax}}\ \delta_k(x).$$

This algorithm is called **Quadratic Discriminant Analysis (QDA)**, namely because $\delta_k(x)$ is quadratic in x. QDA uses the assumption that the variance for each class is different. If

we assume a constant variance across all K classes, then we can eliminate some terms. So, under the assumption $\sigma_1^2 = \sigma_2^2 = \cdots = \sigma_k^2 = \sigma^2$:

$$\underset{k\in\{1,\cdots,K\}}{\text{argmax}} \ -\frac{1}{2\sigma^2}(x-\mu_k)^2 + \ln(\pi_k) = -\frac{x^2}{\sigma^2} + \frac{2x\mu_k}{\sigma^2} - \frac{\mu_k^2}{\sigma^2} + \ln(\pi_k).$$

Note that the first term does not depend on k, so we treat it as a constant and drop it. Therefore, we have

$$\underset{k\in\{1,\cdots,K\}}{\text{argmax}} \ \frac{2x\mu_k}{\sigma^2} - \frac{\mu_k^2}{\sigma^2} + \ln(\pi_k).$$

This form is called **Linear Discriminant Analysis (LDA)** because it is linear in x. Therefore, LDA is a reduced form of QDA if we want to make the simplifying assumption of constant variance.

We can shift perspectives into rewriting both forms using matrices. For simplicity, we omit the derivation and write its closed-form expression. Let μ_k be the sample mean vector in the k-th class, X be the feature vector in class k, and Σ_k be the covariance matrix of the k-th class. Then,

$$\hat{p}\left(X = \overrightarrow{x} | Y = k\right) = \frac{1}{\sqrt{2\pi}|\hat{\Sigma}_k|^{0.5}} e^{-\frac{1}{2}(x-\hat{\mu}_k)^T \hat{\Sigma}_k^{-1}(x-\hat{\mu}_k)}.$$

The closed form optimization function is (at $\overrightarrow{x} = \overrightarrow{x}_0$):

$$\underset{k}{\text{argmax}} \ \ln(\hat{\pi}_k) - \frac{1}{2}\ln\left|\hat{\Sigma}_k\right| + \frac{1}{2}\overrightarrow{x}_0^T \hat{\Sigma}_k^{-1}\overrightarrow{x}_0 + \overrightarrow{x}_0^T \hat{\Sigma}_k^{-1}\overrightarrow{\mu}_k - \frac{1}{2}\overrightarrow{\mu}_k^T \hat{\Sigma}_k^{-1}\overrightarrow{\mu}_k.$$

Recall that $\overrightarrow{x}_0^T \hat{\Sigma}_k^{-1}\overrightarrow{x}_0$ is a quadratic form, which is therefore the QDA in vector/matrix form. In LDA, the second and third terms will cancel since we assume that the covariance matrix Σ is uniform across all classes and so those terms will no longer depend on k. Here we think of the covariance matrix as how strong one feature is correlated to another (non-diagonal elements) and the spread of each feature (variance). For QDA we assume that the correlation between features between classes and for LDA we assume that they are the same between classes.

Some closing thoughts about LDA vs. QDA:

- If we are concerned about the number of parameters used, then LDA is better, for QDA requires roughly k times the number of parameters compared to LDA.
- QDA is more flexible than LDA. If we think about the assumptions, LDA requires us to assume constant variance whereas QDA does not. As always, flexible models are more likely to overfit test data, so we once again need to determine the true pattern of the data.

- QDA is better than LDA if we have a large sample size and relatively small number of predictors. The approximation for each covariance matrix will smooth out and generally be more precise (lower variance). LDA will have more trouble here because it wants to find a uniform covariance matrix across all classes, which will be harder to estimate with the large sample size. If the true covariance matrices of each class differ significantly, then there will be large bias for LDA's uniform covariance matrix assumption. LDA ultimately forces to treat every class as having the same spread and relationship relative to each other. In low dimensional data, LDA becomes too restrained.

Brief Aside: using Bayes' Theorem and notation π_k, f_k, p_k, the Gaussian Naive Bayes is written as

$$p_k(x) = \frac{\pi_k f_{k_1}(x_1) \cdots f_{k_p}(x_p)}{\sum_{l=1}^{k} \pi_l f_{l_1}(x_1) \cdots f_{l_p}(x_p)}.$$

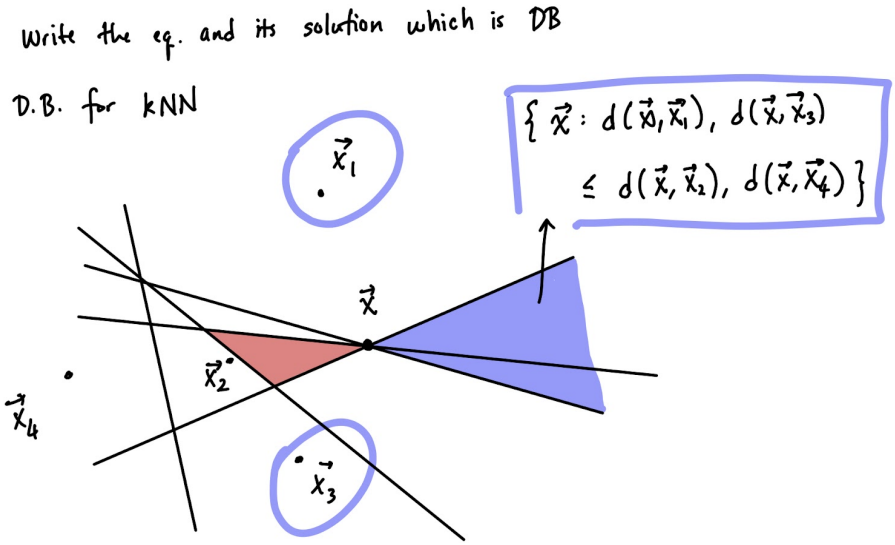
For example, if we wanted to determine the number of parameters required for Naive Bayes, we would need k priors plus $2pk$ since f_{k_i} has p parameters and k classes.

2.3 Decision Boundaries

Simply put, decision boundaries is what really defines classification models; it is how we separate classes from each other. In general, describing decision boundaries for LDA/QDA are straightforward:

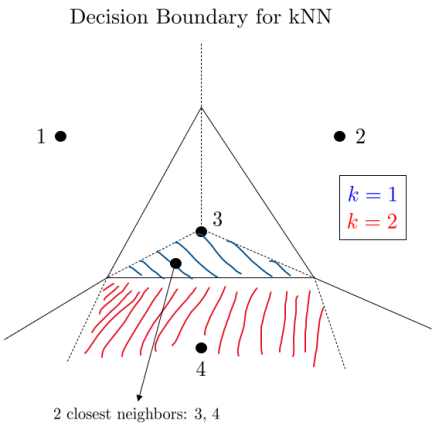
- The decision boundaries in LDA are linear. That is to say, the boundaries separating each class are straight lines.
- The decision boundaries in QDA are quadratic. That is to say, the boundaries separating each class are parabolic.
- The decision boundaries in Naive Bayes are, in most cases, moderately non-linear.
- The decision boundaries in logistic regression are the same as LDA, but they are computed in different ways.

If the decision boundary is complex (in which it cannot be generated by the above algorithms), then we resort to a non-parametric method. We will look at kNN in particular:



Suppose we want to put $\overrightarrow{x}$ into a class using its two nearest neighbors $\overrightarrow{x_1}$ and $\overrightarrow{x_3}$. We want to find the region that is closest to both neighbors, which in this case is the region shaded in blue. Therefore, we would put $\overrightarrow{x}$ into the blue class over the red class.

The regions are generated by *splitting* the plane in half between pair of observations. While the above example doesn't fully live up to that, the intuition is hopefully there. Another example exhibits the difference of predicted region depending on number of neighbors used (1 or 2).



Example 2.3. Suppose we want to perform Linear Discriminant Analysis with two predictors and one output which falls into two classes $y = 0$ and $y = 1$. The training data consists of 6 points (x_1, x_2, y) given by

$$\{(-2,-2,0),(-1,2,0),(-3,0,0),(2,-2,1),(1,2,1),(3,0,1)\}$$

For example, the input $(1,2)$ corresponds to 1. What is the decision boundary in the x-plane?

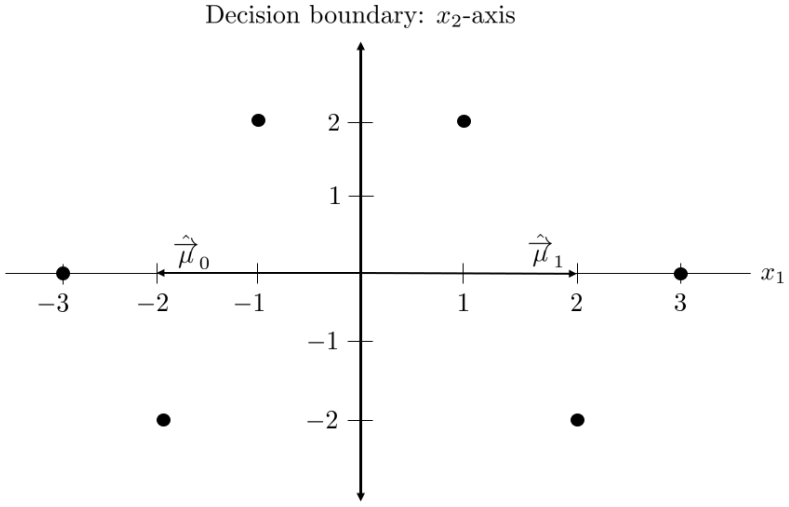
The decision boundary is the center line x between the mean vector $\overrightarrow{\mu}$ of each class, or more precisely:

$$||x - \hat{\mu}_0|| = ||x - \hat{\mu}_1||$$

We have that

$$\hat{\overrightarrow{\mu}}_0 = \begin{pmatrix} -2 \\ 0 \end{pmatrix} \quad \text{and} \quad \hat{\overrightarrow{\mu}}_1 = \begin{pmatrix} 2 \\ 0 \end{pmatrix}$$

The center line is the x_2-axis; any point to the left yields 0 and to the right yields 1.



This holds if the covariance matrix is diagonal, or $\sigma_{12} = \sigma_{21} = 0$.

$$\sigma_{12} = \sigma_{21} = \frac{1}{n-1}\sum_{i=1}^{n}(x_{1i} - \mu_1)(x_{2i} - \mu_2) = 0$$

$$= \frac{1}{n-1}\sum_{i=1}^{n}(x_{1i} - 0)(x_{2i} - 0) = 0$$

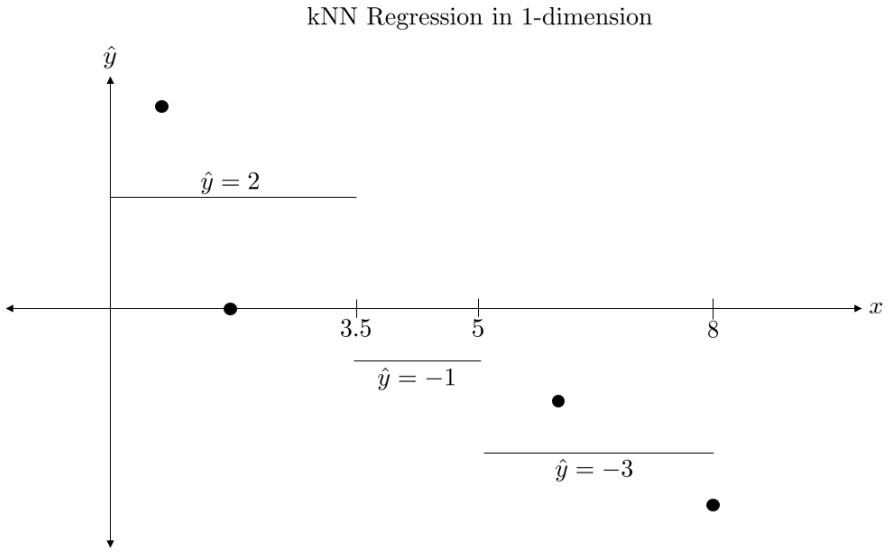
Where μ_1 and μ_2 are the average of all x_1, x_2 values in the dataset.

Example 2.4. Imagine we want to perform KNN regression with $K = 2$ using a training dataset with just four observations and just one predictor. The training data consists of the four points

$$\{(1, 4), (2, 0), (6, -2), (8, -4)\}$$

(i.e. the input 1 corresponds to the output 4). Sketch a graph of what the corresponding prediction function will look like. Do not worry about boundary points (when an input is equally far from two different points)

- When $x < 3.5$, the two closest neighbors are $x = 1$ and $x = 2$. Therefore, we take the average of their y's, which is 2.
- When $3.5 < x < 5$, the two closest neighbors are $x = 2$ and $x = 6$. The average of their y's is $\hat{y} = -1$.
- When $x > 5$, the two closest neighbors are $x = 6$ and $x = 8$. The average of their y's is $\hat{y} = -3$



3 Resampling Methods

The process we have streamlined thus far has been to build a model, train it on existing data (choose objective function and optimize), and then test it on new data. So far, we have assumed that one set of training and test data is sufficient. What if we want, given a fixed amount of data, many sets of train and test data? We refer to this as *resampling*, or generating different samples of training data to get a better understanding of how our models hold up. For example, if we have 100 observations, we could generate 20 sets of (90 training data, 10 test data) models, and see how they compare to each other. Recall from earlier that, given a model $f(x)$ with predicted value and observed value, $f(x_0), y$,

$$\text{Test Error } = E_{\varepsilon,\,\text{train}}\left[y - \hat{f}(x_0)\right]$$

We applied this to a single set of test data. Partitioning our training just once is wasteful; we are not using the available data to its maximum capacity! This is the motivation for resampling methods.

There are two general approaches to resampling:

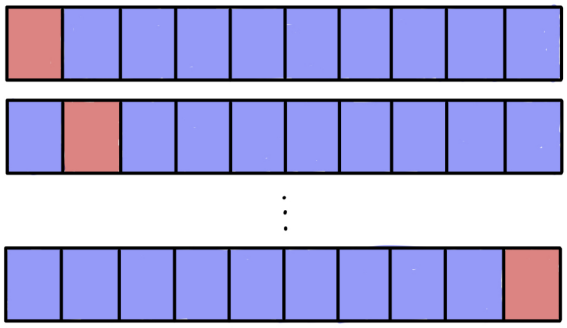
- Pretend there is no separated test data. Instead, choose different training and test sets, out of the entire data, in each sample.
- Ignore the initial set of test. Within the training data, partition into new subsets of training and test data. Then, validate on the initial set of test data

Validating our data really means that we want to *test* the performance of our model, generated by the subsets of training data, on the subsets of test data before using the "real" test data. This is a way of using observations as a preliminary test before generalizing a model to unseen data.

In the following sections, we will discuss three methods of resampling, which approach they fall into, and dive into their cost-benefit trade-off.

3.1 Leave One Out Cross Validation

Suppose we are given a dataset with N observations. The Leave One Out Cross Validation (LOOCV) method constructs N *folds*, or samples, each containing $N - 1$ training data. The remaining observation is the test data. The accuracy is measured by averaging the performance of the N models.

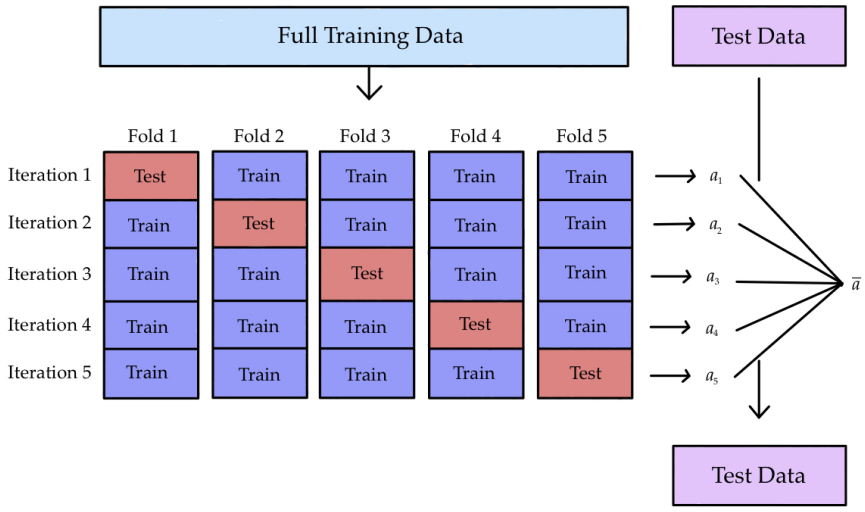


The blue rectangles indicate the set of data we are training, and the red rectangle is the singular test data.

With the large amount (N) of models generated by LOOCV, the accuracy will be more promising compared to other methods because we are using all N observations in each fold, utilizing the data to its full potential. The model works well in practice; however, the computational cost grows significantly as the number of observations increases. Therefore, this method is typically avoided with large datasets and used for when N is small.

3.2 K-Fold Cross Validation

K-Fold validation relaxes the restrictions imposed on partitioning data. Instead of generating N models, we partition the training data into $k < N$ groups. For each group, or fold, we train on $k - 1$ groups and test on the remaining group. This is performed k times to ensure each group is the test data one time.



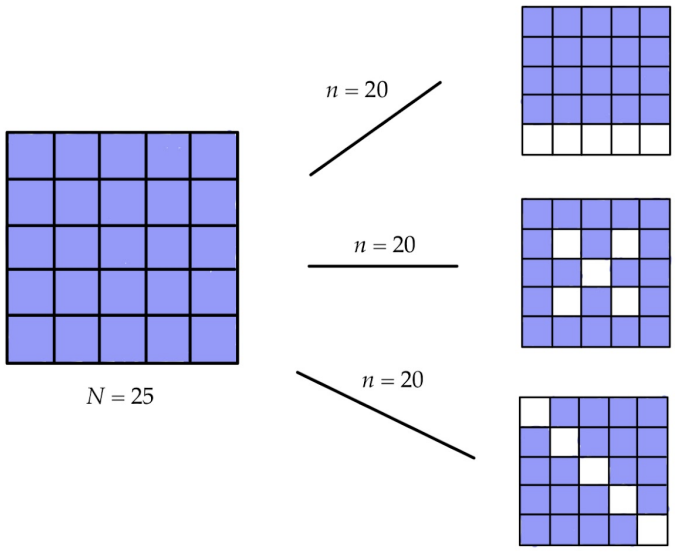
The figure performs a 5-fold cross validation ($k = 5$). First, we partition our entire dataset into training and test data, which will be kept aside for now. For the time being, we operate on the training data by splitting it into 5 equal folds. In the first iteration, we let folds 2–5 be the training data and then test the new-found model on fold 1. We iterate four more times until each fold has been used as test data. Then, we compute the average performance across the 5 folds ($\overline{a}$). Then, we generate a model on the entire training set (combining all 5 folds) and then test on the data we set aside earlier.

Typically, to lessen computational cost, $k = 5$ or $k = 10$ are reasonable choices.

K-Fold Cross Validation is a great resampling method; it efficiently takes advantage of training data before using any test data. Ultimately, we will get an idea of our model will perform before applying it on new, unseen data.

3.3 Bootstrapping

Compared to the other two approaches, bootstrapping takes on a more general philosophy. We generate k subsets of our observation data with replacement, each containing n data points, and apply it on unseen data.



Suppose our data has 25 observations. The bootstrapping method applied here takes 3 subsets of observation data, each containing 20 data points. We would then obtain 3 unique models.

Bootstrapping is incredibly powerful in repeated sampling. Of course, we will obtain a higher accuracy with more samples. 3 is way too small; generally 500-1000 of samples are

more reasonable with larger data. This, in turn, will increase accuracy but also increase the computational cost. Overall, bootstrapping gives us some intuition on the *variance* between splits and the uncertainty of accuracy.

4 Neural Networks, Deep Learning

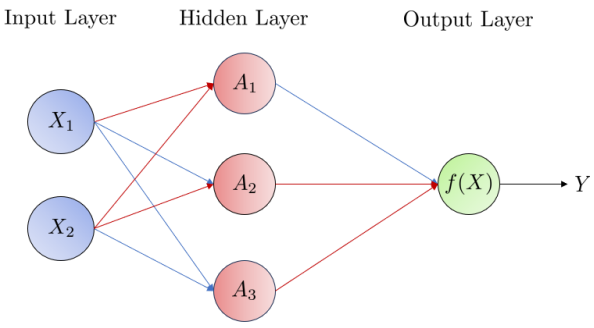
If you have ever studied the brain, we describe its functions as one, complex, entangled web. It is comprised of billions of neurons which pass information to neurotransmitters. These so-called *messengers* "light up" when they are activated by a neuron's incoming signal and pass on the new information to other neurons. This process is repeated until a terminal branch is reached, which then processes the information to different parts of the body. New information then gets sent back to the brain, creating a feedback loop. Ultimately, it is only with a neural network that we can perform simple tasks (i.e. movement, sensations, processing information).

4.1 Overview of Neural Networks

Neural networks in machine learning, fundamentally, are no different from our how brain works. The overarching idea is to initialize a set of neurons and send them through *activation functions*. These functions get "fired" and transmit new information to the next activation function. The process repeats until every set of activation functions has been processed; the final set our neurons is our output, or the terminal branch in this sense. Depending on our objective, the network's output may be a single binary value, as in classification problems, or multiple continuous values.

The input layer consists of the set of P observations $X = (X_1, X_2, \cdots, X_p)$, each assigned to a neuron. Additionally, each neuron is given a weight, which describes how strong of a *connection* it has between other neurons. After the input layer, neurons then get passed into *hidden layers*, each containing *activation functions*. These functions are *non-linear* transformations of our input neurons and generally are not fixed in advance, but instead learned as we train the network. Hidden layers can be thought of "working behind the scenes," or as the intermediate steps of the neural network. Finally, the output layer uses the most recent activation of neurons as its input, resulting in a function $f(X)$. The diagram shown below summarizes the structure of a neural network, containing the input layer, one hidden layer, and output layer.

This is known as a **single layer neuron network**, namely because it only has one hidden layer. We take in two inputs X_1, X_2. The hidden layer computes activations (more on this later) on 3 neurons $A_k = h_k(x)$. The arrows indicate the feeding of the input neurons into the K neurons in the hidden layer, also known as the *weights*.



Then, the neurons from the hidden layer produce the output $f(X)$. Additionally, the

weights are colored red and blue to indicate correlation (negative and positive) between neurons. This will be further discussed in the **gradient descent** and **backpropagation** sections. We can model this the neural network by the function

$$f(X) = \beta_0 + \sum_{k=1}^{K} \beta_k h_k(X) = \beta_0 + \sum_{k=1}^{K} \beta_k g\left(w_{k0} + \sum_{j=1}^{p} w_{kj} X_j\right)$$

$f(X)$ is written here as a culmination of two steps. The K activations for $k = 1, \cdots, K$ are computed as functions of the input neurons:

$$A_k = h_k(X) = g\left(w_{k0} + \sum_{j=1}^{p} w_{kj} X_j\right)$$

$g(z)$ is a non-linear activation function and operates on the weights w_{kj} and inputs X_j. The next section dives into common activation functions. β_k is known as a *bias*, which allows the neuron to adjust its output without relying just on the weights, shifting the activation function and increasing flexibility.

Example 4.1. Let X_1, X_2 be two input neurons. The hidden layer also has two neurons A_1, A_2 corresponding to $h_1(X), h_2(X)$, and $g(z) = z^2$. The weights and biases are given by

$$\beta_0 = 0, \quad \beta_1 = \frac{1}{4}, \quad \beta_2 = -\frac{1}{4}$$

$$w_{10} = 0, \quad w_{11} = 1, \quad w_{12} = 1$$

$$w_{20} = 0, \quad w_{21} = 1, \quad w_{22} = -1$$

What is $f(X)$ with respect to X_1, X_2?

Using the expressions for the activation functions,

$$A_1 = h_1(X) = (0 + X_1 + X_2)^2, \quad A_2 = h_2(X) = (0 + X_1 - X_2)^2$$

Then, plug into $f(X)$:

$$f(X) = 0 + \frac{1}{4}(X_1 + X_2)^2 - \frac{1}{4}(X_1 - X_2)^2 = X_1 X_2$$

This is known as an interaction term, often analyzed in regression models. In some cases, we would not choose $g(z)$ to be quadratic, mainly because there is a higher change to overfit and because $g'(z)$ **is unbounded**, which is a problem when we talk about gradient descent. The next section aims to introduce common activation functions in neural networks.

4.2 Activation Functions

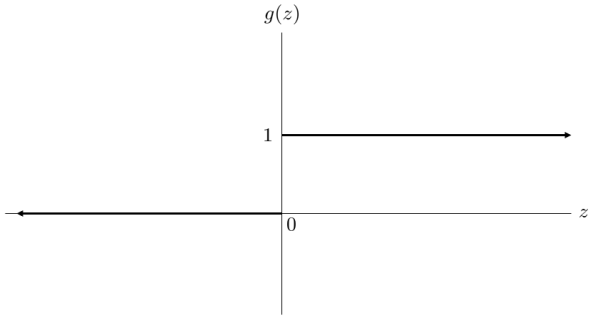
The core of neural networks are the transformations applied to each neuron. Without activation functions, we have cannot reliably train a model with flexibility and expect high accuracy on test data. Below is an overview of common activation functions; we will provide a more detailed description for the starred ones.

- Identity or linear: $g(z) = z$ or $g(z) = az + b, \quad a, b \in \mathbb{R}$.
- Polynomial: $g(z) = a_n z^n + a_{n-1} z^{n-1} + \cdots + a_0, \quad a_i \in \mathbb{R}$.
- Maxpool: $g(z) = \max(z_1, \cdots, z_n), \quad z \in \mathbb{R}^n$.
- **Linear Threshold (Heaviside)
- **ReLU (Rectified Linear Unit)
- **Sigmoid
- **Softmax

Heaviside Function

The heaviside function passes an input into a simple *if-else* statement and outputs a piecewise function.

$$g(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{otherwise} \end{cases}$$



We also compute the gradient

$$\frac{\partial g}{\partial z} = 0.$$

The gradients will be useful to have for gradient descent and backpropagation. The heaviside function can be great for a simple neural network if we want a binary output, implying that the neuron either lights up or it doesn't. However, we will run into a problem when we talk about backpropagation. Namely, we want a nonzero and smooth gradient and, obviously, this is not satisfied by the heaviside function.

We say a neuron is *active*, or firing, when the activation function returns a nonzero value and *inactive* when the activation function returns zero.

ReLU

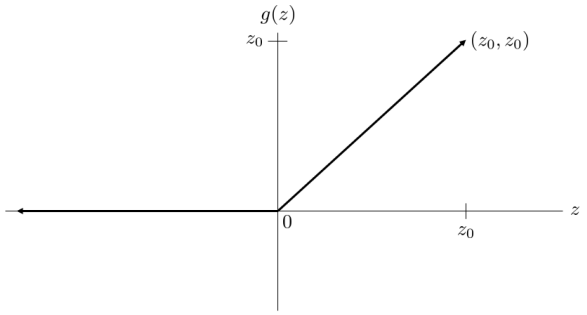
The ReLU takes on a slight modification from the heaviside function if $z \geq 0$. ReLU is known for its low computational cost and its sparsity−the proportion of times where the neurons output zero. However, if too many neurons output zero, then learning is prohibited and is formerly known as the *dying ReLU problem*.

ReLU is great as a hidden layer activation but not an output layer activation because it

only produces positive values.

$$g(z) = \begin{cases} z & \text{if } z \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

$$\frac{\partial g}{\partial z} = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

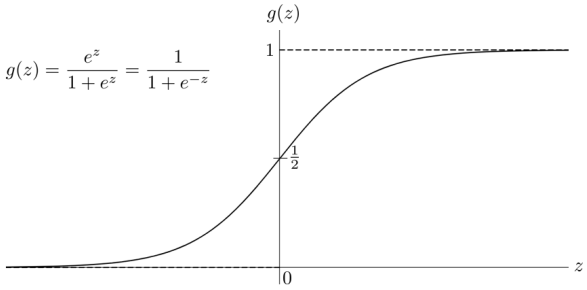


Sigmoid Function

In Chapter 2, we introduced *logistic regression* models to measure probability. The sigmoid function is actually just the same model!

$$g(z) = \frac{1}{1 + e^{-z}}$$

$$\frac{\partial g}{\partial z} = g(z)(1 - g(z))$$



Sigmoid functions tend to be the desired output layer activation for binary classification models because they range from (0, 1). They can be great in hidden layer activations with small changes in inputs because the gradient is smooth and continuous for optimization. However, that's where the benefits end. If z grows too large in magnitude, we run into the *vanishing gradient problem*, where the gradient $\frac{\partial g}{\partial z} \approx 0$, significantly slowing down the learning process.

Aside: Recall that a function is pointwise continuous for a sequence of functions on $\{g_n(z)\}_{n=1}^{\infty}$ if $\lim_{n\to\infty} g_n(z) = g(z)$ for every given $z \in Z$. Suppose we constructed a sequence of sigmoid functions

$$g_n(z) = \frac{1}{1 + e^{-nz}}$$

and took the limit $\lim_{n\to\infty} g_n(z)$? We must consider two cases: $z < 0$ and $z > 0$ since they will determine the sign of $-nz$ and behavior of g_n.

$$\lim_{n\to\infty} g_n(z) = \begin{cases} 0 & \text{if } z < 0 \\ \frac{1}{2} & \text{if } z = 0 \\ 1 & \text{if } z > 0 \end{cases} \implies g_n(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z > 0 \end{cases}$$

This is the heaviside function (with the exception of $z = 0$)!

Note, however, that the convergence for $g_n(z)$ is not uniform. This is to say that there is no "overall speed of convergence."

Softmax Function

The softmax function is simply an extension of the sigmoid function to multiple dimensions. Given a vector of neuron inputs z,

$$g(z) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}.$$

The output represents the probability values for each class. Akin to the sigmoid function, the softmax function is also used in the output layer in a multi-class classification problem, following a multinomial distribution.

Example 4.2. Suppose we want to determine whether a neural network is most likely to classify an image as either a dog, cat, or a bird. The given input $z = [z_1, z_2, z_3]$ corresponds to the raw scores of a dog (z_1), cat (z_2), and bird (z_3). If $z = [2.3, 2.1, 0.6]$, what will the neural network output return?

The output vector (probabilities) $[g(z_1), g(z_2), g(z_3)]$ is generated by inputting each z_i into the softmax function:

$$g(z_1) = \frac{e^{2.3}}{e^{2.3} + e^{2.1} + e^{0.6}} \approx 0.4996, g(z_2) = \frac{e^{2.1}}{e^{2.3} + e^{2.1} + e^{0.6}} \approx 0.4091$$

$$g(z_3) = \frac{e^{0.6}}{e^{2.3} + e^{2.1} + e^{0.6}} \approx 0.0913$$

The output of the neural network is $[0.4996, 0.4091, 0.0913]$ and it will return "dog" because it has the largest probability.

Applications of Activation Functions

Before proceeding, we summarize some important applications of neural networks for the ReLU, sigmoid, and softmax functions.

- ReLU: Image processing in convolutional neural networks. They find local features in an image, such as edges and small shapes. This allows the the network to learn complex patterns in images (for instance, a zebra's stripes).
- Sigmoid: Binary classification problems such as processing emails (spam vs. not spam), medical diagnoses (positive or negative), and fraud detection (real or illegitimate transaction).

- Softmax: Multi-class classification problems such as image classification (see previous example), text categorization and sequential word/vocabulary probability (ChatGPT does this), and handwriting recognition (the MNIST dataset is a very common example).

These activations *break* linearity, namely applying non-linear transformations to each neuron and producing a nonlinear decision boundary in the end.

4.3 Boolean Neural Networks

This section is designed to apply neural networks in simple logic problems. Before we begin, we let z_k be the linear combination of weights plus the bias in the k-th neuron, as described in Section 4.1:

$$z_k = \beta_k + \sum_{j=1}^{p} w_{kj} X_j$$

AND Boolean

Basic True/False boolean problems are often summarized in a truth table. The AND(X_1, X_2) boolean returns true if two predictors return true, and false otherwise. The truth table is therefore

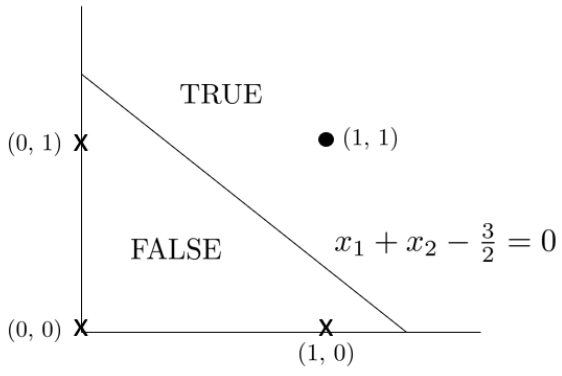
X_1	X_2	AND(X_1, X_2)
T	T	T
T	F	F
F	T	F
F	F	F

Let 0 = False and 1 = True. Suppose we wanted to construct a decision boundary for the set of points

$$\{(0,0)^T, (0,1)^T, (1,0)^T, (1,1)^T\}$$

using a neural network, containing one hidden layer.

Here we are tasked to design a neural network with a set of weights and biases in the hidden layer. We can actually construct a linear decision boundary, which means we do not need any special activation functions in the hidden layer! We can apply a simple linear transformation. Let $\overrightarrow{w}$ be the set of weights $[w_1, w_2]$ given by $\overrightarrow{w} = [1, 1]$. Choose the bias $\beta = -\frac{3}{2}$.



Then, $z = \overrightarrow{w} \cdot \overrightarrow{X} + \beta = X_1 + X_2 - \frac{3}{2}$. The output layer can contain the heaviside function as its activation $g(z)$. Now, we verify that each of the four points is correctly classified:

$$(0,0)^T : z = -\frac{3}{2}, g(z) = 0 \quad (0,1)^T : z = -\frac{1}{2}, g(z) = 0$$

$$(1,0)^T : z = -\frac{1}{2}, g(z) = 0 \quad (1,1)^T : z = \frac{1}{2}, g(z) = 1$$

OR Boolean

This time, the OR(X_1, X_2) boolean returns true if at least one of X_1, X_2 is true. The truth table is now:

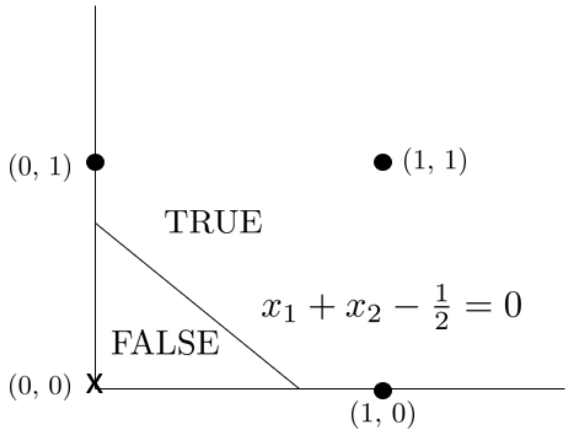
X_1	X_2	OR(X_1, X_2)
T	T	T
T	F	T
F	T	T
F	F	F

Suppose we wanted to construct a decision boundary for the set of points

$$\{(0,0)^T, (0,1)^T, (1,0)^T, (1,1)^T\}$$

using a neural network, containing one hidden layer.

Similarly, we need only a linear decision boundary to classify these points. We can keep $\overrightarrow{w} = [1,1]$ from the previous example, but this time, let $\beta = -\frac{1}{2}$. The output layer will use the heaviside function as its activation. One can verify that the four points are appropriately assigned to their output.



XOR Boolean

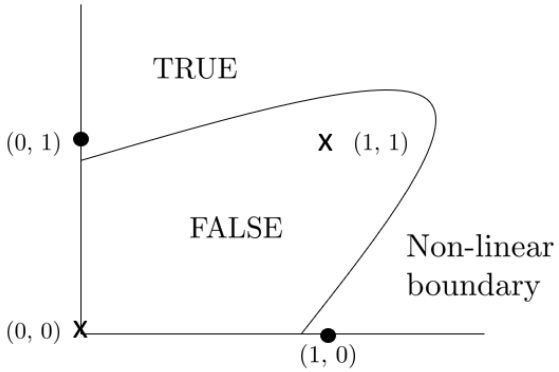
The XOR(X_1, X_2) booelan returns true if only one of X_1, X_2 is true. The truth table is now:

X_1	X_2	XOR(X_1, X_2)
T	T	F
T	F	T
F	T	T
F	F	F

Once again, let us design a neural network that successfully classifies the four points

$$\{(0,0)^T, (0,1)^T, (1,0)^T, (1,1)^T\}.$$

This time, we observe that a linear decision boundary is impossible. A line cannot divide the plane without generating some sort of false positive. This is to say that generating a region that contains both $(0, 1)$ and $(1, 0)$ will also contain $(0, 0)$ or $(1, 1)$.



We consider a neural network with an input layer (2 neurons), hidden layer using ReLU (2 neurons), and output layer using the sigmoid function (1 neuron).
Let $A^{(1)} = g_1(z) = \max(0, W_1X + \beta_1)$, where W_1 and β_1 are the weights and biases given by

$$W_1 = \begin{pmatrix} -1 & 1 \\ 1 & -1 \end{pmatrix}, \quad \beta_1 = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

Let $y = g_2(W_2A^{(1)} + \beta_2)$, where W_2 and β_2 are the weights and bias given by

$$W_2 = \begin{pmatrix} 1 & 1 \end{pmatrix} \quad \beta_2 = -\frac{1}{2}$$

and $g_2(z)$ is the sigmoid function. Then, for each of the following four points:

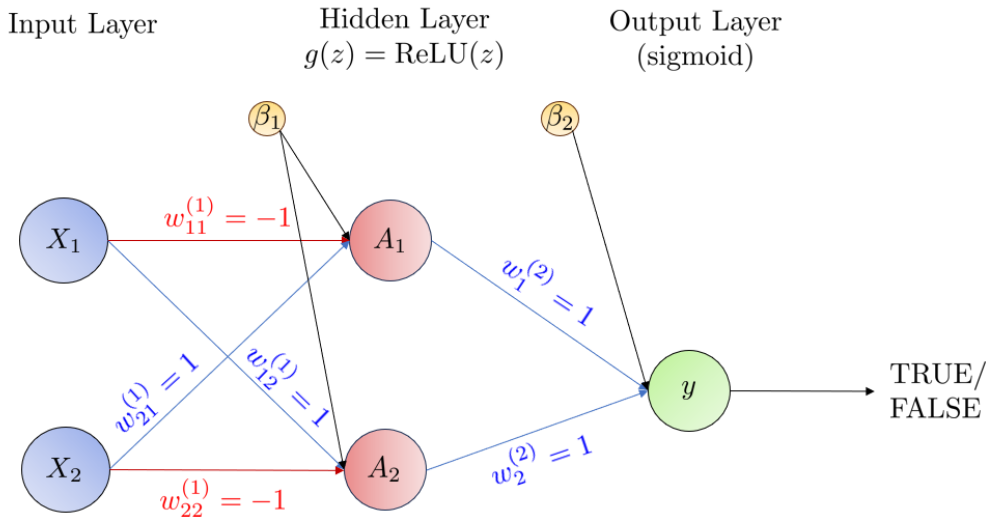
$$(0,0)^T \implies A^{(1)} = g_1(z) = (0 \quad 0)^T \implies y = g_2\left(-\frac{1}{2}\right) \approx 0.38 \rightarrow \text{ False.}$$

$$(0,1)^T \implies A^{(1)} = g_1(z) = (0 \quad 1)^T \implies y = g_2\left(\frac{1}{2}\right) \approx 0.62 \rightarrow \text{ True.}$$

$$(1,0)^T \implies A^{(1)} = g_1(z) = (1 \quad 0)^T \implies y = g_2\left(\frac{1}{2}\right) \approx 0.62 \rightarrow \text{ True.}$$

$$(1,1)^T \implies A^{(1)} = g_1(z) = (0 \quad 0)^T \implies y = g_2\left(-\frac{1}{2}\right) \approx 0.38 \rightarrow \text{ False.}$$

Here $A^{(1)}$ contains two hidden layer activations (neurons)–$A_1^{(1)}$ and $A_2^{(1)}$–generated by W_1, β_1. The following diagram summarizes the neural network we just created!



The superscripts (1), (2) correspond to which layer we are sending the weights to. The weights $w_{11}^{(1)}, w_{12}^{(1)}$ are the weights for X_1 and $w_{21}^{(1)}, w_{22}^{(2)}$ are the weights for X_2 in the first hidden layer. These are the elements stored in W_1. Likewise, $w_1^{(2)}, w_2^{(2)}$ are the weights for A_1, A_2 in the output layer, stored in the vector W_2. There is one bias (or vector of biases) that connect to each activation.

For this neural network, there are [(2 weights)(2 input neurons) + 2 biases] + [(2 hidden layer neurons)(1 weight) + 1 bias] = 9 total parameters.

4.4 Gradient Descent and Backpropagation

The previous section gave us an idea of how to design small neural networks with 100% accuracy. However, fitting large neural networks is somewhat complex. We provide a brief overview of the steps:

- Run your model with an initial set of weights, biases, and activation functions. This is what we did in the previous section, and is known as *forward feeding*−information flows from the input layer and eventually into the output layer.
- Compute the aggregate error between predicted and actual responses using a *loss function*.
- Run back through the neural network, starting from the output layer, adjusting weights that will minimize the squared error. This is *backpropagation*.

We will apply *gradient descent*−an iterative method−to minimize our loss function.

Forward Feeding

For simplicity, we will use a single layer neural network. Forward feeding a regression model $f(x)$ is simply the output

$$f_{\overrightarrow{w}}(\overrightarrow{x_i}) = \beta_0 + \sum_{k=1}^{K} \beta_k g\left(w_{k0} + \sum_{j=1}^{p} w_{kj} x_{ij}\right)$$

where x_i is the i-th observation and K is the number of neurons in the hidden layer. For a classification model, we can use the softmax function

$$f_i(x) = P(y = i \mid X = x) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

as our forward feeding model, where z is the linear combination of weights and biases in each class.

Loss Functions

As mentioned earlier, loss functions measure the total error between our output and what the actual output is. We will denote the loss function as a function of our weights $E(\overrightarrow{w})$.

For regression models, the loss function is often the mean squared error (MSE):

$$E(\overrightarrow{w}) = \frac{1}{n} \sum_{i=1}^{n} \left(f_{\overrightarrow{w}}(\overrightarrow{x_i}) - y_i\right)^2$$

The optimization problem to be solved is one that finds a $\overrightarrow{w}$ minimizing E.

For classification problems, the choice for E is one that maximizes $P(\text{all } y_i \mid \text{all } X_i)$. Mathematically, the optimization problem can be expressed as solving

$$\begin{aligned} \underset{\overrightarrow{w}}{\operatorname{argmax}} \prod_{j=1} \prod_{y_i=j} P(j|X_0) = \prod_{i=1}^{n} P(y_i|X_i) &= \underset{\overrightarrow{w}}{\operatorname{argmax}} \log\left(\prod_{j=1} \prod_{y_i=j} P(j|X_0)\right) \\ &= \underset{\overrightarrow{w}}{\operatorname{argmax}} \sum_{j=1} \sum_{i:y_i=j} \log f_j(x_j) \end{aligned}$$

This can be translated to the minimum by putting a negative sign:

$$= E\left(\overrightarrow{w}\right) = \underset{\overrightarrow{w}}{\operatorname{argmin}} - \sum_{j=1} \sum_{i:y_i=j} \log f_j(x_j)$$

The loss function for a classification problem is known as the *cross-entropy loss* function. Recall that $f_j(x_j)$ is the softmax function.

Gradient Descent

Recall that the derivative of E along a direction $\overrightarrow{V}$ is denoted by $\nabla_{\overrightarrow{V}} E$, or the *gradient*. Let $\eta > 0$ be the learning rate such that the change in E is modeled by

$$E\left(\overrightarrow{w} + \eta\overrightarrow{V}\right) - E\left(\overrightarrow{w}\right) \approx \left(\nabla E(\overrightarrow{w}) \cdot \overrightarrow{V}\right)\eta$$

Here $\overrightarrow{w} + \eta\overrightarrow{V}$ is the small change in $\overrightarrow{w}$ while moving in the direction of $\overrightarrow{V}$, and so the change in E also moves in the direction of $\overrightarrow{V}$. The equation is derived from the definition of a *directional derivative*

$$\lim_{\eta \to 0} \frac{E\left(\overrightarrow{w} + \eta\overrightarrow{V}\right) - E\left(\overrightarrow{w}\right)}{\eta} = \nabla E(\overrightarrow{w}) \cdot \overrightarrow{V}$$

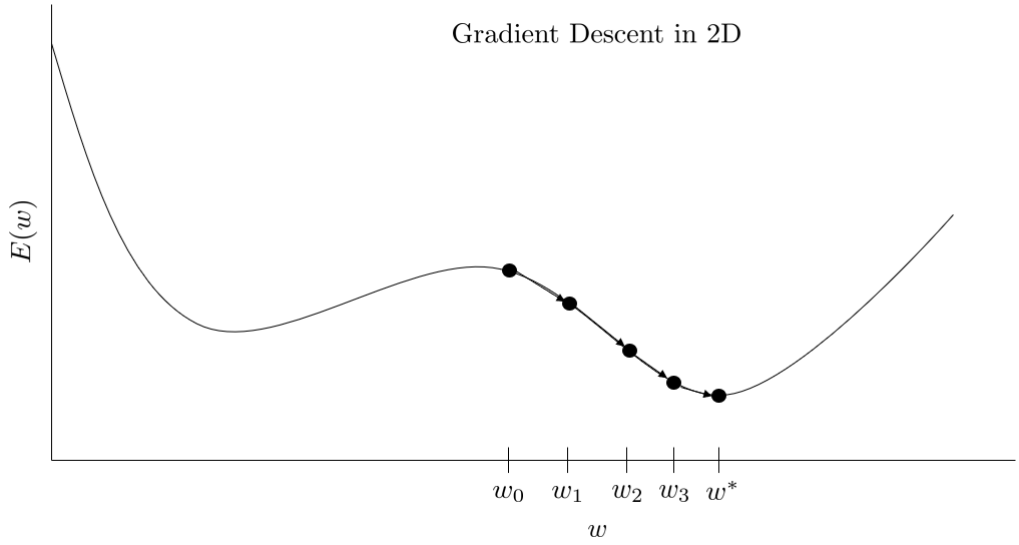
Therefore, the best direction to minimize E is the negative gradient direction:

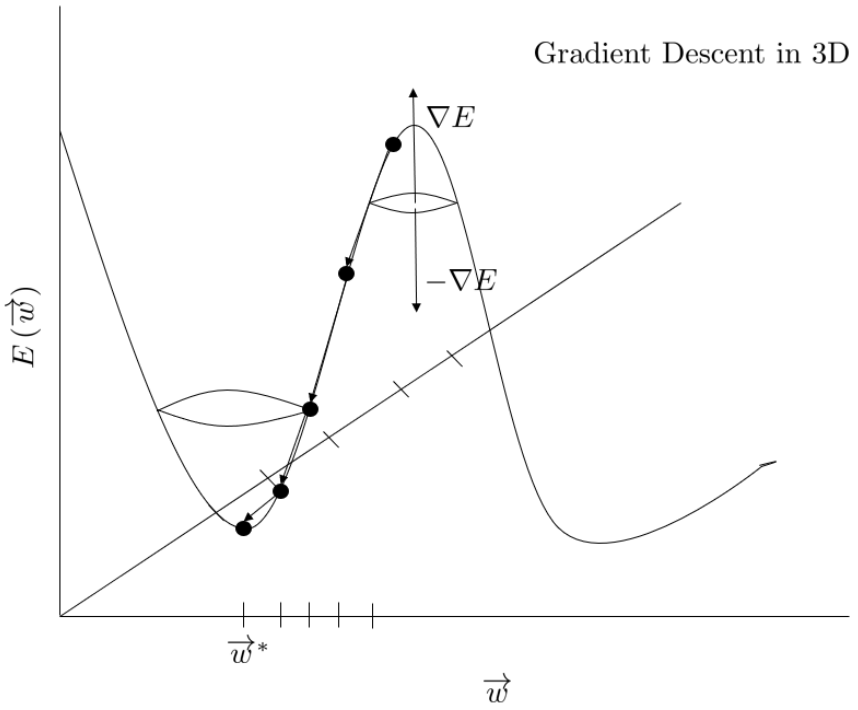
$$\overrightarrow{V} = -\frac{\nabla E}{|\nabla E|}.$$

The procedure for gradient descent, in practice, is pretty straightforward:

1. Initialize a reasonable, random set of weights $\overrightarrow{w}$.
2. Compute $\nabla E(\overrightarrow{w})$.
3. Set $\overrightarrow{w} = \overrightarrow{w} - \eta\nabla E(\overrightarrow{w})$.
4. Repeat steps 1-3 until the change in loss function goes below a certain threshold−until you cannot go down any further−or when you go over the number of iterations you have set.

Generally, we set η, the hyperparameter, to a small number (i.e. 10^{-2}) and the threshold to be a difference of less than 10^{-4} or 10^{-5}. Shown below are examples of gradient descent in 2D and 3D cases.





Some remarks about gradient descent:

- We compute ∇E through backpropagation.
- If we make η too large, then we may overshoot and have divergence. If η is too small, then the computational cost grows larger.

- We cannot say that the minimum we find is indeed the global minimum. It will be a local minimum at worst, but is there a way to fall into the global minimum?

The answer to the last item uses *stochastic gradient descent*, where we try to "force" our weights out of a local min and into another, seeing if it is further down than the one we previously found. We add randomness/diffusion to our gradient descent model:

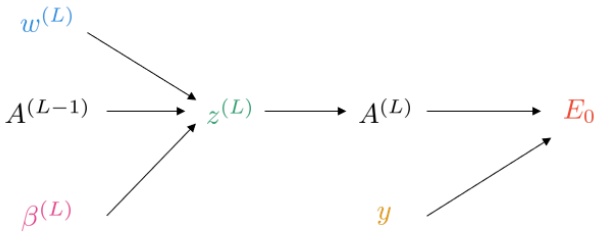
$$\vec{w} = \vec{w} - \underbrace{\eta \nabla E(\vec{w})}_{\text{drift}} + \underbrace{\mathcal{G}}_{\text{diffusion}}, \quad \mathcal{G} \sim \text{Gauss}(0, \sigma)$$

Here an initial choice of $\vec{w}$ is not too important; however, we should randomize it to some extent so it is not uniform.

Backpropagation

So far, we have discussed forward feeding, our loss functions, and gradient descent. The final step in finalizing our neural network is *backpropagating* through our network, adjusting the weights that truly minimize our loss function E.

The computation for backpropagation is quite heavy, so we will stick to backpropagating a regression neural network which uses MSE as the loss function. Also, we will assume a single neuron in each layer.



Think about how E_0 is computed. We combine the activation from the previous layer and the weights and bias in the current layer to obtain $z^{(L)}$. Then, plug in $z^{(L)}$ into the activation $A^{(L)}$. We use the difference between $A^{(L)}$ and y to get the loss E_0! Obviously, $A^{(L-1)}$ is inspired by the previous activation, weights $w^{(L-1)}$, bias $\beta^{(L-1)}$, and so forth, but let us only consider the elements in the diagram.

The idea to determine the sensitivity of E_0 with small perturbations in $w^{(L)}$. Or, more precisely,

$$\frac{\partial E_0}{\partial w^{(L)}}$$

We note that a small perturbation in $w^{(L)}$ will cause some change in $z^{(L)}$, or $\partial z^{(L)}$, which in turn, causes a change in $A^{(L)}$ ($\partial A^{(L)}$), and ultimately influencing E_0 (∂E_0)! The change in loss can ultimately be represented as a product of ratios as we go down the sequence

$$\frac{\partial E_0}{\partial w^{(L)}} = \frac{\partial z^{(L)}}{\partial w^{(L)}} \frac{\partial A^{(L)}}{\partial z^{(L)}} \frac{\partial E_0}{\partial A^{(L)}}$$

This is the chain rule! Now, we compute the relevant derivatives. As we mentioned earlier, we are using the loss function for a regression problem which is the mean squared error. So, $E_0 = \left(A^{(L)} - y\right)^2$ implies that

$$\frac{\partial E_0}{\partial A^{(L)}} = 2\left(A^{(L)} - y\right)$$

We express $A^{(L)}$ as $A^{(L)} = g\left(z^{(L)}\right)$, so

$$\frac{\partial A^{(L)}}{\partial z^{(L)}} = g'\left(z^{(L)}\right)$$

Lastly, $\partial z^{(L)} = \partial w^{(L)} A^{(L-1)} + \beta^{(L)}$, so

$$\frac{\partial z^{(L)}}{\partial w^{(L)}} = A^{(L-1)}$$

Therefore, the change in the loss with the weights is the product

$$\frac{\partial E_0}{\partial w^{(L)}} = 2A^{(L-1)} g'\left(z^{(L)}\right)\left(A^{(L)} - y\right).$$

As given by the diagram, this is just one iteration of backpropagation. We compute the change in loss function ∂E_0 over each training example. If there are n observations, then the aggregate change in loss is

$$\frac{\partial E}{\partial w^{(L)}} = \frac{1}{n} \sum_{k=0}^{n-1} \frac{\partial E_k}{\partial w^{(L)}}.$$

What we derived here is one half of $\nabla \partial E_0$. Recall that

$$\nabla \partial E_0 = \begin{pmatrix} \frac{\partial E}{\partial w^{(1)}} \\ \frac{\partial E}{\partial \beta^{(1)}} \\ \vdots \\ \frac{\partial E}{\partial w^{(L)}} \\ \frac{\partial E}{\partial \beta^{(L)}} \end{pmatrix}$$

We also needed to find the change in loss with respect to the biases. Just apply the chain rule again:

$$\frac{\partial E_0}{\partial \beta^{(L)}} = \frac{\partial z^{(L)}}{\partial \beta^{(L)}} \frac{\partial A^{(L)}}{\partial z^{(L)}} \frac{\partial E_0}{\partial A^{(L)}}$$

Since $\frac{\partial z^{(L)}}{\partial \beta^{(L)}} = 1$,

$$\frac{\partial E_0}{\partial \beta^{(L)}} = 2g'\left(z^{(L)}\right)\left(A^{(L)} - y\right).$$

Similarly, we can find ∂E_0 with respect to the previous activation function.
We would have the expression

$$\frac{\partial E_0}{\partial A^{(L-1)}} = \frac{\partial z^{(L)}}{\partial A^{(L-1)}} \frac{\partial A^{(L)}}{\partial z^{(L)}} \frac{\partial E_0}{\partial A^{(L)}}$$

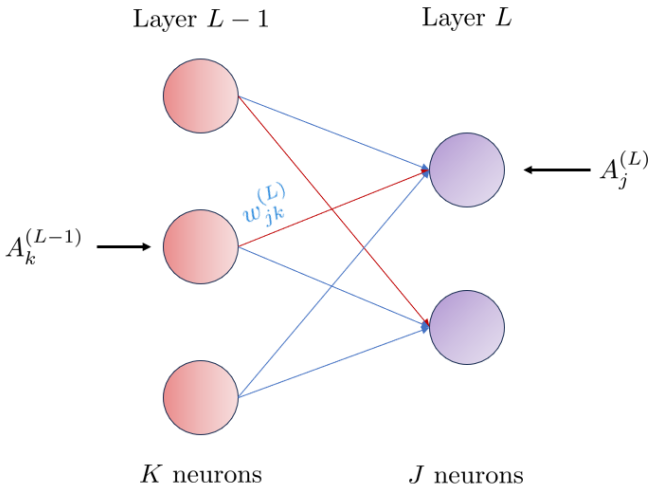
The derivative $\frac{\partial z^{(L)}}{\partial A^{(L-1)}} = w^{(L)}$, which gives us

$$\frac{\partial E_0}{\partial A^{(L-1)}} = 2w^{(L)} g'\left(z^{(L)}\right)\left(A^{(L)} - y\right).$$

Now, we can just iterate the same chain rule idea backwards to see how sensitive the loss function is to previous weights and biases. It is one long-winded expression of the chain rule, multiplying a lot of ratios!

$$\frac{\partial E_0}{\partial w^{(1)}} = \frac{\partial z^{(L)}}{\partial w^{(L)}} \frac{\partial A^{(L)}}{\partial z^{(L)}} \frac{\partial E_0}{\partial A^{(L)}} \cdots \frac{\partial z^{(1)}}{\partial w^{(1)}} \frac{\partial A^{(1)}}{\partial z^{(1)}} \frac{\partial E_0}{\partial A^{(1)}}$$

In fact, we can easily generalize this to layers with additional neurons! We need only add subscripts k and j to represent the j-th neuron in layer L and k-th neuron in the previous layer $L-1$.



The loss function for layer L is basically the same, now adding subscripts to account for each neuron.

$$E_0 = \sum_{j=0}^{n_L-1} \left(A_j^{(L)} - y_j\right)^2$$

We also add subscripts to represent the weights connecting the k-th neuron to the j-th neuron as $w_{jk}^{(L)}$. This gives us a new expression for $z_j^{(L)}$ and $A_j^{(L)}$:

$$z_j^{(L)} = w_{j0}^{(L)} A_0^{(L-1)} + w_{j1}^{(L)} A_1^{(L-1)} + w_{j2}^{(L)} A_2^{(L-1)} + \beta_j^{(L)}$$

$$A_j^{(L)} = g\left(z_j^{(L)}\right)$$

More generally, if layer $L-1$ has K neurons

$$z_j^{(L)} = \beta_j^{(L)} + \sum_{k=1}^{K} w_{jk}^{(L)} A_k^{(L-1)}$$

The chain rule representations for $\frac{\partial E_0}{\partial w_{jk}^{(L)}}, \frac{\partial E_0}{\partial \beta_j^{(L)}}$ follow immediately:

$$\frac{\partial E_0}{\partial w_{jk}^{(L)}} = \frac{\partial z_j^{(L)}}{\partial w_{jk}^{(L)}} \frac{\partial A_j^{(L)}}{\partial z_j^{(L)}} \frac{\partial E_0}{\partial A_j^{(L)}}, \qquad \frac{\partial E_0}{\partial \beta_j^{(L)}} = \frac{\partial z_j^{(L)}}{\partial \beta_j^{(L)}} \frac{\partial A_j^{(L)}}{\partial z_j^{(L)}} \frac{\partial E_0}{\partial A_j^{(L)}}$$

We do need to make a slight modification for $\frac{\partial E_0}{\partial A_k^{(L-1)}}$. Since we sum over $n_L - 1$ for the E_0 and $A_k^{(L-1)}$ contributes to E_0, we must sum over layer L for the derivative, or more precisely:

$$\frac{\partial E_0}{\partial A_k^{(L-1)}} = \sum_{j=0}^{n_L-1} \frac{\partial z_j^{(L)}}{\partial A_k^{(L-1)}} \frac{\partial A_j^{(L)}}{\partial z_j^{(L)}} \frac{\partial E_0}{\partial A_j^{(L)}}$$

Remember that we use j for layer L and k for $L-1$; it is easy to get the notation confused.

Once we know how sensitive the loss function is with respect to the activations in $L-1$, we can repeat this process for all of the weights and biases feeding into that layer.

That's backpropagation–the powerhouse that drives deep learning!

For each iteration of backpropagation, since each weight and bias needs an update, we compute as many partial derivatives of the loss function as there are parameters in our neural network.

Successful neural networks always follow the procedure laid out in this chapter. The math and notation itself is quite tedious, but putting in the hard work leads to a truly remarkable ending. Before we conclude neural networks, we provide one example which follows the steps we outlined throughout the chapter.

Example 4.3. Adapt the following description of the backpropagation algorithm.

- Each training example is a pair of the form $\langle \vec{x}, \vec{t} \rangle$, where $\vec{x}$ is the vector of network input values, and $\vec{t}$ is the target vector values.
- η is the learning rate.
- n_{in} is the number of network inputs, n_{hidden} the number of units in the hidden layer, and n_{out} the number of output values.
- The input from unit i into unit j is x_{ji} and the weight from unit i to unit j is w_{ji}.
- Assume one hidden layer, use ReLU as the activation function in the hidden layer, do not use an activation function for the output layer, and use squared-error as the loss function.

Initialize all network weights to small random numbers (e.g., between -0.05 and 0.05). Set η to be small, say $\eta = 10^{-2}$. Repeat the following steps until the termination condition is met: For each $\langle \vec{x}, \vec{t} \rangle$ in training-examples,

(a) Input the instance $\vec{x}$ to the hidden layer:

$$a_j = \text{ReLU}\left(\sum_i w_{ji} x_i\right) = \max\left(0, \sum_i w_{ji} x_i\right)$$

(b) Since there is no activation function for the output layer, we just have

$$o_k = \sum_j w_{kj} a_j.$$

This completes forward feeding.

(c) To begin backpropagation, compute the error for the output later. Let $\delta_k = \frac{\partial E}{\partial o_k}$. Then,

$$E = \sum_k (o_k - t_k)^2 \implies \delta_k \longleftarrow \frac{\partial E}{\partial o_k} = 2(o_k - t_k)$$

(d) The derivative of the ReLU function is

$$\frac{\partial}{\partial a_j}\text{ReLU}(a_j) = \begin{cases} 1 & \text{if } a_j > 0 \\ 0 & \text{if } a_j \geq 0 \end{cases}$$

The error term for the hidden layer neuron is

$$\delta_j = \frac{\partial E}{\partial a_j} = \frac{\partial E}{\partial o_k}\frac{\partial o_k}{\partial a_j}\frac{\partial a_j}{\partial \text{ReLU}(a_j)} = \sum_k \delta_k w_{kj}$$

Therefore,

$$\delta_j \longleftarrow \begin{cases} \sum_k w_{kj}\delta_k & \text{if } a_j > 0 \\ 0 & \text{otherwise} \end{cases}$$

(e) Update the output layer weights

$$w_{kj} \longleftarrow w_{kj} - \eta \delta_k a_j$$

(f) Update the hidden layer weights

$$w_{ji} \longleftarrow w_{ji} - \eta \delta_j x_i$$

This completes backpropagation.

5 Decision Trees and Ensemble Methods

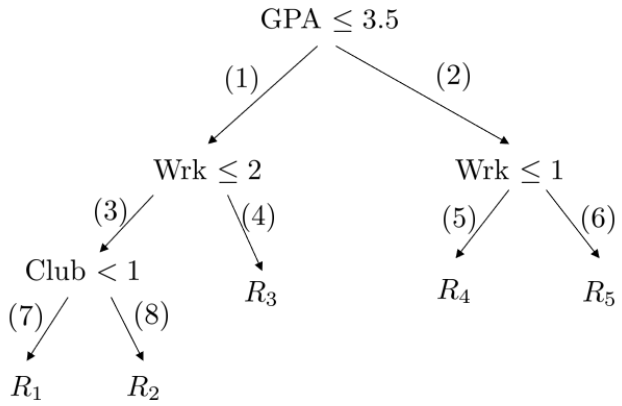
We have looked at various supervised learning models−kNN, linear/logistic models, Naive Bayes, LDA/QDA, Neural Networks−each with their own spin on classification and regression problems. Now, we consider an approach that has a similar concept to kNN. Consider the usual setup: given a set of data $\{(x_i, y_i)\}$, can we predict where y falls on new data?

5.1 Decision Trees

This time, we will predict y based on nested *if-else* statements or *claims*. Or, more plainly, an observation will follow a trajectory based on whether it is lower than or above certain numerical quantities. For instance, let's say we want to predict whether a student gets admitted into grad school. We can classify criteria accordingly:

- Primary claim: Move left if the student's undergraduate GPA is ≤ 3.5 (1), and right if it is > 3.5 (2).
- Secondary claim: Given (1) is true, move left is the student has at most 2 year of work experience (3), and right if they have more than 2 year (4). Given (2) is true, move left if the student has at most 1 year of work experience (5), and right if they have more than 1 year (6).
- Tertiary claim: For students with an undergraduate GPA ≤ 3.5 and at most 2 years of work experience (3), move left if they participated in no clubs (7) and move right if they participated in at least one club (8).

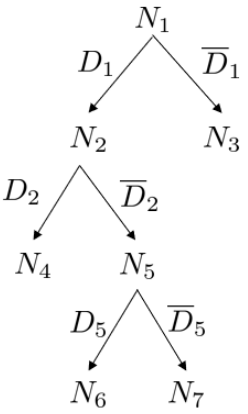
We can structure our findings as such:



The resulting diagram is what we call a *decision tree*. It uses a sequence of decisions to partition the x-space into M regions−$R_1, R_2, \cdots, R_M$. Each nodes N_i are associated with a

region R_i in said x-space alongside a decision D_i, except ones at the end. We define these nodes *leaves*, or *terminal nodes*, as those without descendants or further decisions/partitions.

Taking the decision tree to the right, N_3, N_4, N_6, N_7 are leaves. D_i are the decisions corresponding to the nodes N_i. Therefore, there are three decision nodes: D_1, D_2, and D_5 for the nodes N_1, N_2, N_5. If an observation fails to satisfy a claim, then we denote it as $\overline{D}_i$. Say an observation satisfies the claims D_1 and D_5 but does not satisfy D_2. Then the region R_2 (node N_6) is the set $\{x : D_1, \overline{D}_2, D_5\}$.



The grad school decision tree models a *classification* problem, where the response is a binary outcome (accepted to grad school = 1, not accepted = 0). The terminal nodes in the model R_1, R_2, R_3, R_4, R_5 are the five different regions, or categories of students.

- R_1: Student has GPA ≤ 3.5 and has more than 2 years of work experience.
- R_2: Student has GPA > 3.5 and has at most one year of work experience.
- R_3: Student has GPA > 3.5 and has more than one year of work experience.
- R_4: Student has GPA ≤ 3.5, has at most 2 years of work experience, and has not participated in clubs.
- R_5: Student has GPA ≤ 3.5, has at most 2 years of work experience, and has participated in at least one club.

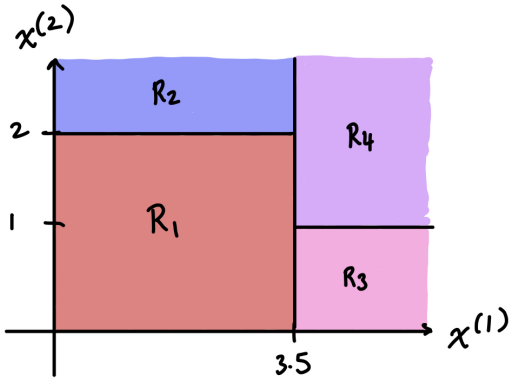
Such a classification problem could be a (over)simplified heuristic for an admissions board in accepting/denying students, saying that students who fall in the regions R_1, R_3, and R_5 should get admitted and denying R_2, R_4.

To further simplify the problem, suppose we no longer want to use club participation as a predictor and only use GPA and work experience in our model. How can we draw each region in the x-plane?

Suppose $x^{(1)}$ measures GPA and $x^{(2)}$ measures work experience. Then we are concerned with the claims $x^{(1)} \leq 3.5$ and $x^{(2)} \leq 2$ if the first claim is true and $x^{(2)}$ if false.

(See next page)

Seeing how the plane is partitioned, an admissions board would be more likely to accept students in region 2 and region 4. Obviously, we cannot guarantee high accuracy, especially if we are only using 2 predictors. If we decided that club participation is significant enough, we would have a 3-dimensional space and $x^{(3)}$ is the predictor associated with clubs.

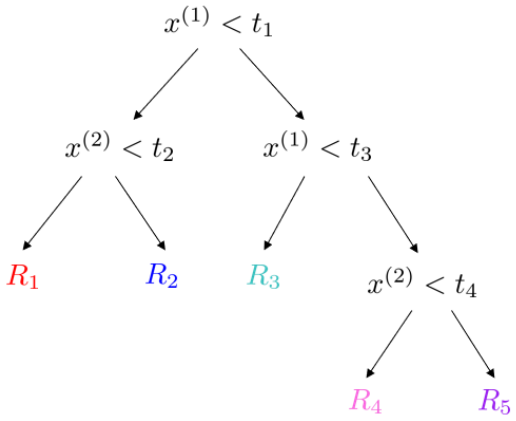


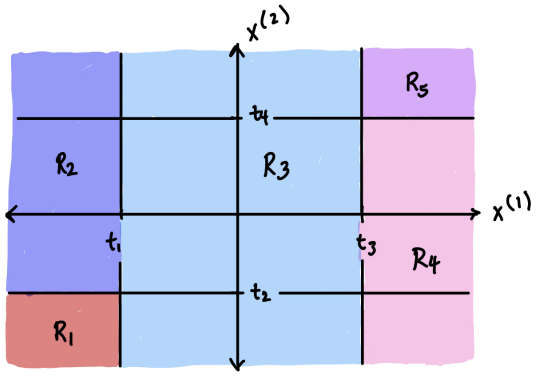
We now look at examples of different decision tree boundaries.

Example 5.1. Suppose $x^{(1)}$ and $x^{(2)}$ are two predictors and t_1, t_2, t_3, t_4 are parameters. Let us define regions $R_1, \cdots, R_5$ by:

- $R_1 : x^{(1)} < t_1, x^{(2)} < t_2$
- $R_2 : x^{(1)} < t_1, x^{(2)} \geq t_2$
- $R_3 : x^{(1)} \geq t_1, x^{(2)} < t_3$
- $R_4 : x^{(1)} \geq t_1, x^{(1)} \geq t_3, x^{(2)} < t_4$
- $R_5 : x^{(1)} \geq t_1, x^{(1)} \geq t_3, x^{(2)} \geq t_4$

Draw the corresponding decision tree and regions in the x-plane.

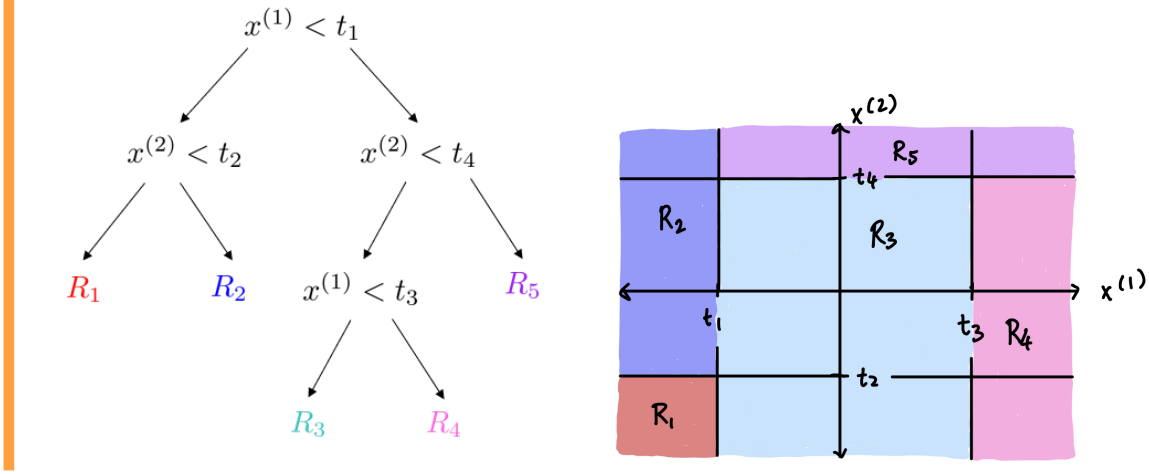




Example 5.2. This time, change R_3, R_4, R_5 such that

- $R_3 : x^{(1)} \geq t_1, x^{(2)} < t_4, x^{(1)} < t_3$
- $R_4 : x^{(1)} \geq t_1, x^{(2)} < t_4, x^{(1)} \geq t_3$
- $R_5 : x^{(1)} \geq t_1, x^{(2)} \geq t_4$

What is the new decision tree and region distribution in the x-plane?



In theory, regions can have any shapes, but we use rectangles for simplicity and ease of interpretation.

The general procedure for generating decision trees and boundaries, in a **regression** setting, follow:

1. Divide the predictor space into J distinct and nonoverlapping regions $R_1, ..., R_J$.
2. For every observation that falls into the same region, we make the same prediction–average the training observations in that region
3. Find $R_1, \cdots, R_J$ that minimize the residual sum of squared error (RSS)

$$\text{RSS} = \sum_{j=1}^{J} \sum_{i \in R_j} \left(y_i - \overline{y}_{R_j} \right)^2$$

where $\overline{y}_{R_j}$ is the mean response for the region R_j.

In more complex cases, it may not be feasible to check all possible combinations of regions. So, we may just optimize for one split. At the splitting step, pick a random predictor X_j and a cutoff point S, and we split into two groups

$$R_1^{(j,s)} = \{x | x_j < s\}, \quad R_i^{(j,s)} = \{x | x_j \geq s\}$$

Then, we minimize

$$\sum_{i: x_i \in R_1^{(j,s)}} (y_i - \hat{y} R_i)^2 + \sum_{i: x_i \in R_2^{(j,s)}} (y_i - \hat{y} R_2)^2$$

This is known as the *greedy algorithm* since it makes the best local decision without considering future splits.

Remark: For **classification** models, we take the majority vote in each region.

A decision tree function for regression models is computed by

$$f(x) = \sum_{m=1}^{J} c_m \mathbb{1}_{x \in R_m}$$

Here $\mathbb{1}$ is the *indicator function*, given by

$$\mathbb{1}_{x \in R_m} = \begin{cases} 1 & \text{if } x \in R_m \\ 0 & \text{otherwise} \end{cases}$$

which really just tells us to return 1 if x belongs to a particular region. c_m is the average response in the corresponding region.

Example 5.3. Suppose we wanted to train a decision tree regression model based on 5 data points

$$\left(\begin{pmatrix}2\\1\end{pmatrix}, 5\right), \left(\begin{pmatrix}3\\2\end{pmatrix}, 7\right), \left(\begin{pmatrix}5\\-2\end{pmatrix}, -1\right), \left(\begin{pmatrix}6\\3\end{pmatrix}, 2\right), \left(\begin{pmatrix}10\\-5\end{pmatrix}, -8\right)$$

with the decision boundaries $D_1 : x_1 < 4, D_2 : x_1 < 6$. Construct a decision tree model $f(x)$ that predicts an outcome y for new data. Additionally, compute the RSS.

Let R_1 be the region left of D_1, R_2 be the region between D_2 and D_3, and R_3 be the region to the right of D_2. Label $N_1 \to N_5$ be the given nodes from left to right. Then, $N_1, N_2 \in R_1, N_3, N_4 \in R_2$, and $N_5 \in R_3$. We compute c_m as the average of the responses for each region.

$$c_1 = \frac{5+7}{2} = 6, \quad c_2 = \frac{2-1}{2} = \frac{1}{2}, \quad c_3 = \frac{-8}{1} = -8.$$

Then, the function can be modeled accordingly:

$$f(x) = 6 \cdot \mathbb{1}_{x \in R_1} + \frac{1}{2} \cdot \mathbb{1}_{x \in R_2} - 8 \cdot \mathbb{1}_{x \in R_3}.$$

The RSS for each region is computed as the sum of differences between the observed value and average of observed values, squared.
For R_1:

$$\text{RSS}_1 = (5-6)^2 + (7-6)^2 = 2$$

For R_2:

$$\text{RSS}_2 = \left(-1 - \frac{1}{2}\right)^2 + \left(2 - \frac{1}{2}\right)^2 = 2.25 + 2.25 = 4.5$$

For R_3:

$$\text{RSS}_3 = (-8+8)^2 = 0$$

The total RSS is therefore $\text{RSS}_1 + \text{RSS}_2 + \text{RSS}_3 = 2 + 4.5 + 0 = 6.5$.

Intuitively, if similar outputs are in the same region, we obtain a better cut.

Decision trees are more likely to overfit our training data, especially if we choose a greater depth (more decisions and nodes). In summary, they are sensitive to training data which leads to high variance and low bias. In a practical setting, decision boundaries for decision trees are nonlinear, which poses an advantage compared to a standard linear regression model. Decision trees are **non-parametric** and have horizontal or vertical decision boundaries.

5.2 Random Forests

The goal of this section is to produce a model that tries to solve the overfitting dilemma that decision trees are prone to.

Decision trees overfit test data due to their large variance. Consider a new idea: what if we construct multiple trees and take the average? Let there be n independent observations $z_1, \cdots, z_n$ following a normal distribution of mean 0 and variance σ^2. Then,

$$\overline{z} = \frac{1}{n}\sum_{i=1}^{n} z_i, \quad \text{Var}(\overline{z}) = \text{Var}\left(\frac{1}{n}\sum z_i\right) = \frac{1}{n^2}\sum_{i=1}^{n} \text{Var} z_i$$

$$= \frac{1}{n^2}\sum_{i=1}^{n} \sigma^2 = \frac{\sigma^2}{n}$$

Hence, if we average the predictions then the end results will have low variance (i.e. constructing multiple trees using different training set). We construct $\hat{f}^{(1)}, \hat{f}^{(2)}, \cdots, \hat{f}^{(B)}$ trees. Then,

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

This approach is called *bagging*.

While this helps fix overfitting to some extent, we can make some more restrictions. Instead of randomly choosing one of p features to split, choose one out of m features such that $m < p$. A fresh sample of m predictors is taken at each split. This is a *random forest*, which notably has less variance than bagging and individual decision trees. For regression problems, we fit many trees and take an average of their predictions, and for classification problems we take the majority vote.

Remark: A common choice for m is $\sqrt{p}$.

In summary, decision trees work great on training data, but tend to capture the noise on test data and overfit as a result. Random forests solve this problem by combining

bootstrapping, bagging, and randomness to reduce variance and capture less noise on test data.

Aside: While random forests are often sufficient enough in modeling data with high accuracy, we impose a new method to construct multiple trees sequentially. Here is an overview of the algorithm:

1. Set $\hat{f}(x) = 0, r_i = y_i \quad \forall i$ in training set.
2. For $b = 1, \cdots, B$:
   (a) Fit a tree with d splits to that training data (X, r_i)
   (b) Update $\hat{f}$: $\hat{f}(x) = \hat{f}(x) + \lambda \hat{f}^{(k)}(x)$
   (c) Update the residual $r_i = r_i - \lambda \hat{f}^{(k)}(x_i)$
3. Return
$$\hat{f}(x) = \sum_{i=1}^{B} \lambda \hat{f}^{(b)}(x_i).$$

This is known as *boosting*. It aims to help weaker models with more misclassified samples (larger error/residual) by giving them higher weights, making the next model focus on them. This process is repeated until we have B models. The final prediction is given as the *weighted combination* of the B models. This generally lowers bias because it focuses on predictions that are harder to classify, but can also fall to overfitting if there are too many "weak" models.

While a random forest trains all B models in parallel, boosting emphasizes the improvement of models one-by-one to strengthen the accuracy of the final model.

6 Unsupervised Learning

The first five sections introduced methods of supervised learning, where we train a model and use it to predict a label y.

Recall from Section 1 that in unsupervised learning, there is no label to be predicted. All we have is a dataset $X \in \mathbb{R}^{n \times p}$ and the general task is to explore the *pattern* of data. There are two types of methodologies that both aim to find these trends−dimension reduction and clustering.

Before proceeding, we assume that **all features** of X has mean zero. For arbitrary X, we can pre-process it by subtracting the mean of each variable for the corresponding column.

6.1 Principal Component Analysis (PCA)

Principal Component Analysis is one of many **dimension reduction** problems. Given a high-dimensional dataset $X \in \mathbb{R}^{n \times p}$ (or n data in $\mathbb{R}^p$), we want to find a k-dimensional subspace that "preserves" the high-dimensional information. This is obtained through means of an orthogonal projection function.

- A naive solution is to randomly pick k components of X, but we are going to lose a lot of information this way.
- A more reasonable assumption is that our projection function is a linear transformation, where the linear coefficients depends on the "structure" of the dataset. This is to say that the "new coordinates" are a linear mapping/combination of "old coordinates."
- Reducing dimensions can help remove features that may contain a lot of redundant information and improve the efficiency and performance of machine learning models.

If you have ever heard of the phrase, "describe yourself in two words," this is the essence of PCA and dimension reduction! Wanting to say the most about yourself using as fewest words as possible, in the realm of machine learning, translates to: how much can we maximize the information in our dataset using the simplest model possible?

We motivate two ways to derive this. The first of which is a more naive approach.

Lagrange Multipliers

Following the assumptions about X (namely the fact that it is centered), we construct a covariance matrix C which describes the relationship between the features

$$C = \frac{1}{n} X^T X \in \mathbb{R}^{p \times p} = E\left[X^T X\right]$$

The goal is to describe the most but say the least about C. This is equivalent to finding a $\overrightarrow{v}$ such that the projection of the covariances onto $\overrightarrow{v}$ is the smallest. Mathematically, we

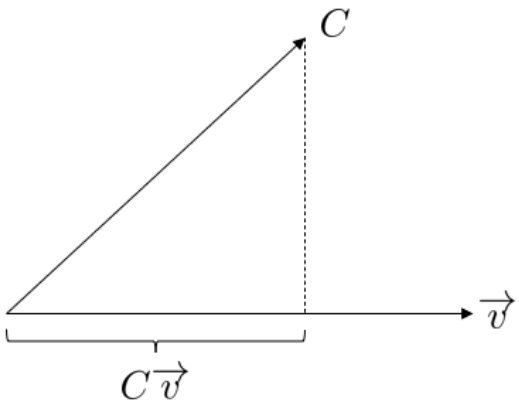
describe this as

$$v \in \mathbb{R}^p, ||v|| = 1 \text{ such that}$$

$$\max_{||\vec{v}||=1} \vec{v}^T C \vec{v}$$

We can find $\vec{v}$ through a constraint optimization problem:

$$\max \underbrace{\vec{v}^T C \vec{v}}_{f(v)} \text{ given } ||\vec{v}|| = 1 = \underbrace{\vec{v}^T \vec{v}}_{g(v)}$$



This is where Lagrange Multipliers come in. The optimization problem satisfies

$$\nabla f = \lambda \nabla g \implies \mathcal{L}(v) = \nabla f - \lambda \nabla g = 0$$

We can cheat a little and use the quadratic form of $f(v)$ to say it is technically equal to $\left(\vec{v}\right)^2 C$.

$$\nabla f = 2\vec{v}^T C, \quad \nabla g = 2\vec{v}^T$$
$$\implies 2\vec{v}^T C = \lambda 2\vec{v}^T \implies \vec{v}^T C = \lambda \vec{v}^T$$

Since C is symmetric, the adjoint (or transpose) of C follows $C^T = C$.

$$C^T \vec{v} = \lambda \vec{v} \implies C\vec{v} = \lambda \vec{v}$$

This is to say, the **eigenvectors** of C optimize our covariances, known as the **principal components**! Note that C is a positive semi-definite matrix, satisfying $\vec{v}^T C \vec{v} \geq 0$, so the eigenvalues will be strictly positive (under the assumption $n > p$, which is almost always the case anyway). We will dive into the interpretation of principal components later, but we introduce a second process of obtaining the same results.

Singular Value Decomposition

Singular Value Decomposition (SVD) yields a more interpretable conclusion compared to Lagrange multipliers. Recall that for a linear operator $T \in \mathbb{R}^{n \times p} : \mathbb{R}^p \to \mathbb{R}^n$, it can be decomposed into

$$T = U \Sigma V^T$$

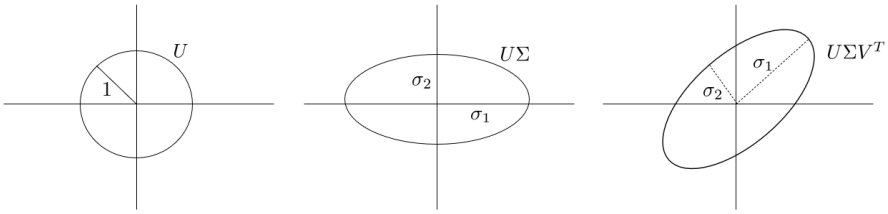
where U and V are $n \times n$ and $p \times p$ unitary matrices, and Σ is a $p \times p$ diagonal matrix containing the **singular values** (σ) of T. A unique property about unitary matrices shows that they contain orthonormal columns, generating an orthogonal basis of $\mathbb{R}^p \left[v_1, \cdots, v_p\right]$, which will be the basis of the principal components (the important directions). When applying T to an arbitrary vector $\vec{x}$, then we have

$$T\vec{x} = U \Sigma V^T \vec{x}$$

We can apply the general idea to our dataset $X \in \mathbb{R}^{n \times p}$, where

$$X = U\Sigma V^T$$

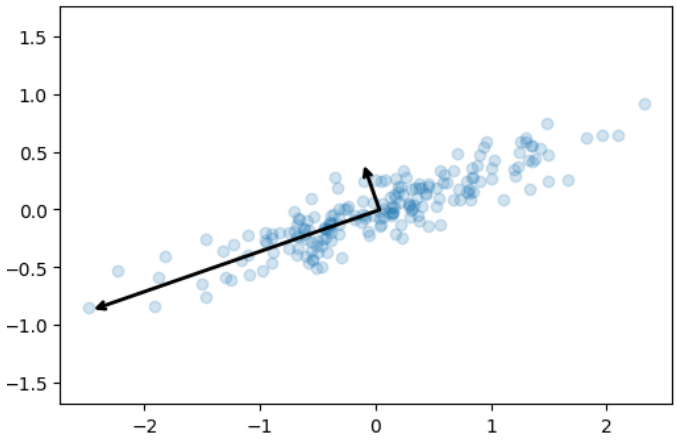
Suppose a simple example where $n = 2, p = 2$. What type of transformations are Σ and V^T?



We start with the unit circle, where the norm of any vector lying on the circle is equal to 1. U is a rotation matrix; however, the ball will not change form regardless of what U is. Applying Σ *squishes* or *stretches* the ball to an ellipse with a major axis of length σ_1 (larger singular value) and minor axis of length σ_2. Once again, V^T is a rotation matrix, rotating the "basis" of the orthogonal singular values (in this illustration, by roughly 45°).

In the context of linear algebra, $U\Sigma$ multiplies different columns in U by the singular values σ_i's, and $U\Sigma V^T$ constructs a linear combination of rows in V^T by coefficients in the corresponding coordinate in $U\Sigma$, producing a new orthonormal frame. The matrix product $U\Sigma$ produces the **scores** of a principal component, or namely the coordinates or positions of the data in the "new" space. For instance, σ_1 contains the most information, and this is the typical convention because he singular values in Σ are always ordered in descending order: $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_p$.

Interpretation



In summary, principal components emphasize the **direction** of the direction and its score explains the **spread** of data, or **explained variance** in that direction. **Scores** are transformed coordinates of the data points into the new principal component subspace. With how singular values are obtained, we say the most explained variance is observed in the first principal component and descending with each successive principal component. In other words, the **explained variance ratio** should decrease with each principal component. Using the plot from the previous page (with $n = 100$, k components $= 2$)

- The arrow pointing in the direction of the "major axis" has the highest explained variance ratio (as given by its magnitude) and thus explains the most information in our data.
- The arrow pointing in the direction of the minor axis has the lowest score/explained variance and thus explains the least information in our data.

We could suggest that the data can be further reduced to one dimension−that the data is generated along one line (major axis) and the short line (minor axis) merely corresponds to "noise." For instance, we would be compelled to believe this if the 1st explained variance ratio was 0.98 and the 2nd was 0.02.

Remark 1: Additionally, because V is unitary, $V^T V = I$ and so

$$XV = U\Sigma$$

implying that the projection of the data onto the principal component space spanned by V, which rotates the data that has the largest variance. This is equivalent to the score!

Remark 2: In an unrealistic scenario, if the explained variance ratio is 1.00 for the first principal component, then our data is one-dimensional (line pointing in the direction corresponding to the first vector in the orthogonal basis). Since principal components are generated from an orthogonal basis, each principal component is orthogonal to each other.

Example 6.1. *Motivation*: Nowadays, facial recognition is widely used for ID verification, known as Face ID. Apple's innovative discovery in 2017 introduced this technology. Before setting up your "ID," you are required to take photos of yourself at different angles. For subsequent uses of your phone, accessing it and transactions are often locked behind Face ID as a safety measure.

Problem: Suppose we have a 64×64 greyscale image (thought of a point in $\mathbb{R}^{4096}$), which is the "digital expression" of the person's face, and a dataset of 16 similar pictures (the different angles in this case), explain how we could use PCA to reduce this into a 200-dimensional subspace. Interpret the 200-dimensional subspace in context of the problem and describe a quantitative method that explains why 200 dimensions may be appropriate.

Idea: We would project the 4096-dimensional vector onto the top 200 principal components, obtaining a 200-dimensional representation of each face, capturing the most significant facial variations while removing noise and redundancy.

The 200 principal components span a subspace that contribute to parts of the facial structure (eyes, eyebrows, glasses, lighting), ensuring efficient storage and fast computation. To summarize, lowering the dimension maintains key characteristics of a person's face while removing unnecessary features.

If 200 dimensions gives us at least 95% **explained variance**, then this ensures that the reduced subspace still retains most of the information. If we lower our metric for the explained variance too much (i.e. 70%), we lose more critical identity information, generating more false positives and making it easier for unauthorized users to access anyone's phone.

Conclusion: This example emphasizes the importance of choosing a reasonable dimension to reduce to and identifying a cutoff for explained variance (balancing security vs. efficiency). With a lower explained variance ratio, we require fewer principal components, blurring the line between individual differences. Therefore, it recognizes different faces as "similar" and ultimately leads to misclassification.

6.2 K-Means Clustering

In addition to dimension reduction methods, another typical task in unsupervised learning model is **clustering**: assigning data samples into several groups, or clusters, based on their similarities. Similar samples should fall in the same cluster and dissimilar samples should be in different clusters.

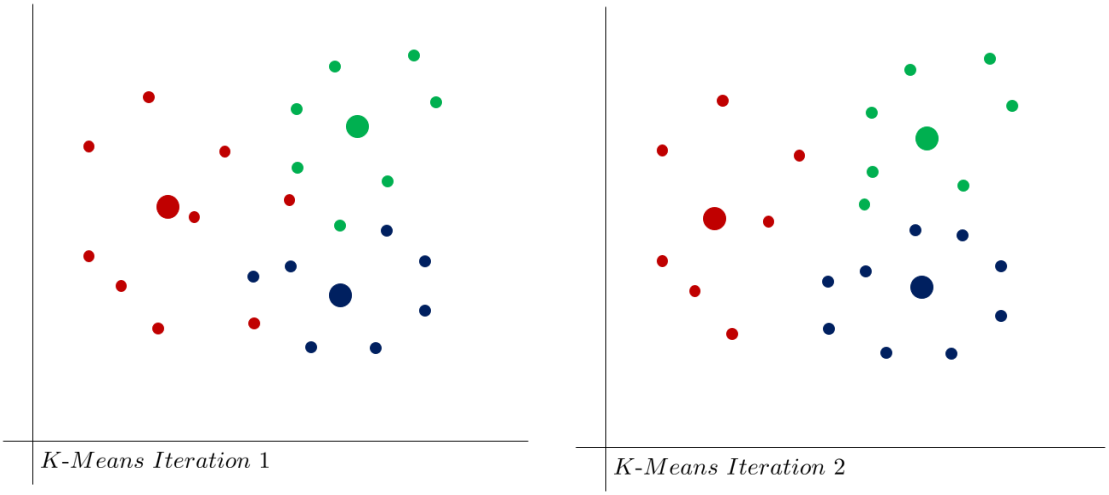
Mathematically, when given a set of observations $X \in \mathbb{R}^{(n\times p)}$, k-means clustering aims to partition the n samples into K sets such that $S = S_1, S_2, \cdots, S_K$, $K \leq n$, so as to minimize the within-cluster sum of squares (variance). Or more formally, we wish to find the best partition of groups that *minimize the loss function* of S:

$$\min_S \sum_{i=1}^{K} \sum_{x\in S_i} ||x - \mu_i||^2$$

where μ_i is the **cluster centroid** or mean of all points in each S_i. The task itself is very difficult; the common approach is in applying Lloyd's algorithm for finding evenly spaced points in Euclidean space. However, the steps are easily streamlined:

- Given the current cluster centroid, update the cluster of data according to its nearest cluster centroid (assignment)

- Given the cluster assignment, update the cluster centroid by calculating the means within each cluster (update). These steps are realized in the illustration below.



The cluster centroids for each class, or color, are given by the large circles, with the observations being the smaller ones. The color for each observation should match the cluster centroid it is closest to. In the first iteration, most observations agree with their closest center. When applying the update step, corrections are made to the previously mismatched observations. By changing color, each class loses and gains new observations, which in turn changes the location of each centroid. Seeing as all observations are now assigned their correct color, we can stop here.

Some general conclusions about K-Means clustering:

- We also need to determine the number of classes K to stratify the data into. In practice, this is a difficult task that is separable from the algorithm itself. It uses a similar approach of explained variance in PCA, computing how much explained variance is added for each cluster. Once the change is negligible, we count the total number of clusters.
- This is not KNN! K-means makes changes within our existing data, whereas KNN is applied on new data and based on its closest points, not by clusters of classes.
- As with loss functions in other machine learning models, we are not guaranteed a global minimum. In K-means clustering, a general approach is to randomly run different initializations of the algorithm in parallel and testing which set of partitions yields the smallest loss.

7 References

All diagrams were self-curated. However, some of them were inspired by 3Blue1Brown's playlist on Neural Networks, which is attached below! The notes themselves were also influenced by the textbook: *An Introduction to Statistical Learning with Python.*

https://www.youtube.com/watch?v=aircAruvnKk&list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi