1 Model Types and Algorithms

1.1 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.

1.2 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).

1.3 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.

1.4 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.

1.5 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.

1.6 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.

1.7 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.

Degrees of freedom: The number of independent terms used to fit the data. A polynomial of degree p has degree of freedom = $p + 1$ (remember to include the intercept term). It is simply another measurement of flexibility to fit the data.

Gradient descent problem

$$\min_{\overrightarrow{c}} \phi(\overrightarrow{c}) = \min_{\overrightarrow{c}} ||\overrightarrow{y} - X\overrightarrow{c}||^2.$$

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 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.

2.2 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.

2.3 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 $\sigma_1^2 = \sigma_2^2 = \cdots = \sigma_k^2 = \sigma^2$:

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

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

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

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 $\overrightarrow{x} = \overrightarrow{x}_0$):

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

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

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

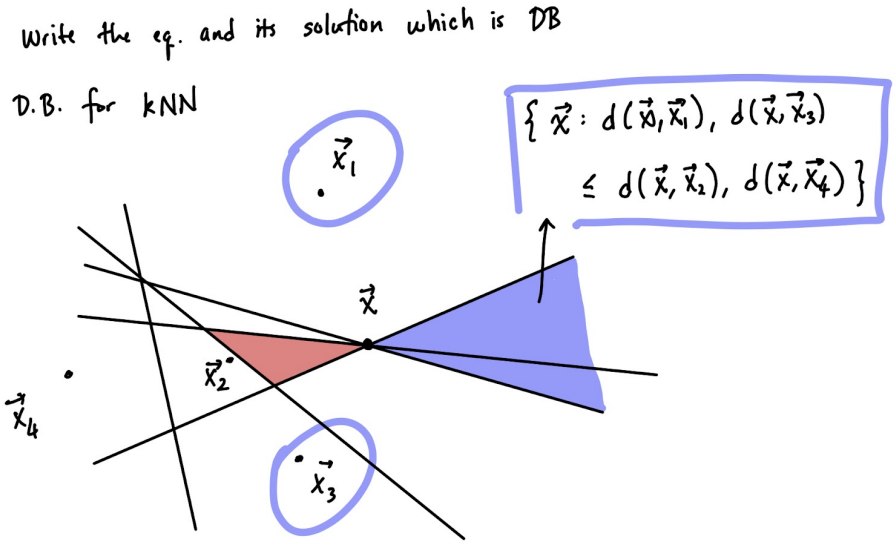
For example, if we wanted to determine the number of parameters required for Naive Bayes, we would need k priors plus $2pk$ since f_{k_i} has p parameters and k classes.

2.4 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:

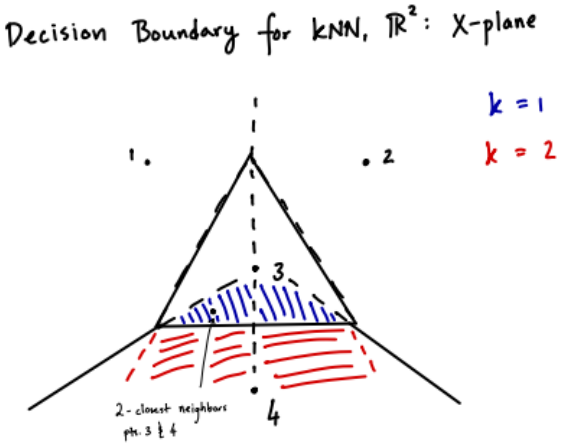
- The decision boundaries in LDA are linear. That is to say, the boundaries separating each class are straight lines.
- The decision boundaries in QDA are quadratic. That is to say, the boundaries separating each class are parabolic.
- The decision boundaries in Naive Bayes are, in most cases, moderately 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. (To be computed: Decision boundary for LDA)

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:

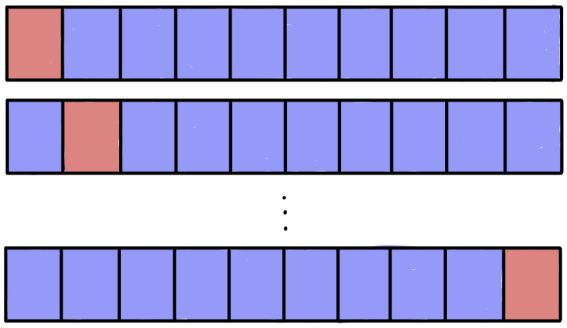
- Pretend there is no separated test data. Instead, choose different training and test sets, out of the entire data, in each sample.
- Ignore the initial set of test. Within the training data, partition into new subsets of training and test data. Then, validate on the initial set of test data

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.

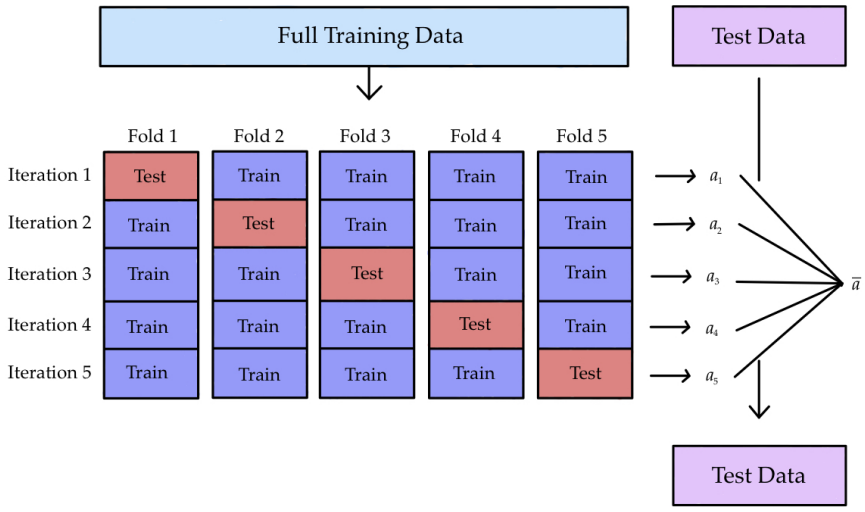


The blue rectangles indicate the set of data we are training, and the red rectangle is the singular test data.

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.



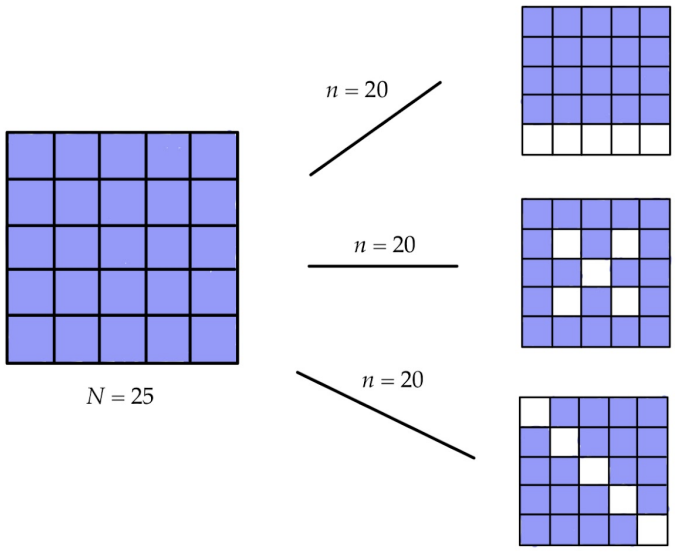
The figure performs a 5-fold cross validation ($k = 5$). First, we partition our entire dataset into training and test data, which will be kept aside for now. For the time being, we operate on the training data by splitting it into 5 equal folds. In the first iteration, we let folds 2–5 be the training 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

1. Overview of Neural Networks, Motivation
2. Common Transfer/Activations
3. Examples of Boolean NN
4. Gradient Descent and Backpropagation

5 Decision Trees and Ensemble Methods