CS 109A Final Notesheet

Author: Gianfranco Randazzo

Missingness

Missing data can arise from various places in data: You decide to start collecting a new variable partway through the data collection of a study. Somethings can't be measured properly. Here are 3 types of missingness:

- **Missing Completely at Random (MCAR)** - the probability of the missingness in a variable is the same for all units. Randomly poking holes in a data set.
- **Missing at Random (MAR)** - the probability of missingness in a variable depends only on available information (other predictors).
- **Missing Not at Random (MNAR)** - the probability of missingness depends on information that has not been recorded and this information also predicts the missing values.

Missing Completely at Random (MCAR)

This is the best-case scenario and the easiest to handle:

- **Examples**: A coin is flipped to determine whether an entry is removed. Or when values were just randomly missed when being entered in the computer.
- **Effect if you ignore**: there is not effect on inferences.
- **How to handle**: lots of options, but best to impute

Missing at Random (MAR)

This is a case that can be handled.

- **Examples**: Men and women respond at different rates to the question, "have you ever felt harassed at work".
- **Effect if you ignore**: inferences are biased, and predictions usually suffer.
- **How to handle**: use the information in the other predictors to build a model and **impute** a value for the missing entry.

Missing Not at Random (MNAR)

This is the worst-case scenario and impossible to handle properly:

- **Examples**: patients drop out of a study because they experienced some bad side effect that was not measured.
- **Effect if you ignore**: there are major effects on inferences or predictions
- **How to handle**: you can improve things by dealing with it like it is MAR, but you likely may never fix the bias. Simply incorporating a **missingness indicator variable** may be the best approach.

Identifying Missingness

Use `pd.isna(df)` to return a boolean matrix with True in each index that had either NaN or None, and False everywhere else. Another way of determining is using `msno.matrix(df.sample(100))` and `msno.heatmap(df.sample(500))` and `msno.dendrogram(df.sample(500))`. Dendrogram clusters leaves linked together at a distance of zero fully predict one another's presence, some variables are 'glued' because they are present in every record.

When using SKLearn, you will need to resolve missingness before you can fit an SKLearn model line LinearRegression on your data.

Dropping

Naively handing missingness, you can just throw away or drop the observations that have missing values. You can use `df.dropna(axis = 0)`, and it will drop any rows with any missing values.

Good idea to drop a predictor column if most of its values are missing across the data. Simiarly drop a row if many of its values are missing. But dropping rows can have **negative consequences**.

- If observations with missing values differ systematically we would be biasing the dataset but dropping such rows.
- If certain values are not missing at randon, but for some underlying reason, then we are getting into trouble.

Imputation

Rather that simply dropping missing values we could try to fill them in with well-chosen values. This filling-in is called **imputation**.

- **Quantitative**: Impute the **mean** or **median** value.
- **Categorical**: Impute the **mode**, the most common class.
- **Hot deck imputation**: for each missing entry, randomly select an observed entry in the variable and plug it in.
- **Model the imputation**: plug in predicted values ($\hat{y}$) from a model based on other observed predictors (for example kNN).
- **Model the imputation with uncertainty**: plug in predicted values plus randomness ($\hat{y} + \epsilon$) from a model based on the other observed predictors.

Modeling with uncertainty:

- Fit a model to predict the predictor variable with missingness from all the other predictors.
- Predict the missing values from a model in the previous part
- Add in a measure of uncertainty by randomly sampling from $\mathcal{N}(0, \sigma^2)$ distribution, where $\sigma^2 = \text{MSE}$ of the model (or bootstrap from observed residuals).

Example with linear regression: Mean imputation leads to biased inference. Mean imputation + missingness indicator performs better.

If want to use a classification model to predict the variable, then all you need to do is flip a 'biased coin' with the probabilities of each class equal to the predicted probabilities of the model.

Imputation across multiple variables. If one variable has missing entries it is easy. If all variable have missingness, then it is an iterative process, impute X_1 based on $X_2, \ldots, X_p$ then impute X_2 based on $X_3, \ldots, X_p$ and continue. Might be an issue that you did not impute all of them the first time, repeat until you do.

Solution for issues about a biased complete data set, you can run multiple imputations and get the average of the estimates.

Missingness Indicator Variable

Since the group of people with a missing entry may be systematically different than those with the variable measured, you want to distinguish them using an indicator variable.

One simple way to handle missingness in variable X_j:

- Impute a value (0 or $\overline{X_j}$).
- Create a new variable $X_{j,miss}$, to indicate a missing value.
- Include both $X_{j,miss}$ and X_j as predictors n any model.

Classification

Why not Linear Regression? If we encode with just values, you are implying an order, and thus going to get errors. If a categorical variable is ordinal, then a linear regression would make some sense but not ideal.

Another choice could be to run a linear regression on the probability, however what if the probability is less than 0 or greater than 1. Thus, still an issue.

Logistic Regression

We will use the Sigmoid function:

$$\log\left(\frac{p}{1-p}\right) = \beta_0 + \beta_1 X \implies P(Y=1) = \frac{1}{1+e^{-(\beta_0+\beta_1 X)}}$$

since we multiply top and bottom by $\exp(-\beta_0 - \beta_1 X)$.

Example:

$$\log\left(\frac{\hat{P}(Y=1)}{1-\hat{P}(Y=1)}\right) = 6.325 - 0.0434(\text{MaxHR})$$

This tell us that as MaxHR increases, the log-odds decrease, thus probability of heart disease is decreasing.

Likelihood

We have the following:

$$L(p \mid Y) = \prod_i P(Y_i = y_i) = \prod_i p_i^{y_i}(1-p_i)^{1-y_i}$$

$$\arg\max_\beta(L(p \mid Y)) = \arg\max_\beta(\ell(p \mid Y))$$

So we then have binary cross entropy, which we want to minimize:

$$L_{BCE} = -\left(\sum_i y_i \log(p_i) + (1-y_i)\log(1-p_i)\right)$$

we have that $\beta_* = \arg\min_\beta L_{BCE}$.

Multiple Logistic Regression

You could use:

$$\log\left(\frac{p}{1-p}\right) = \beta_0 + \sum_{i=1}^{p} \beta_i X_i$$

To fit the multiple logistic regression you take the likelihood approach and you minimize the negative log-likelihood across all the parameters using a method like Gradient Descent. We note that the change of a variable is practically the same, but **multicollinearity** plays a huge role.

Classification Decision Boundaries

Typical classification:

$$C = \begin{cases} 1, & \hat{P}(Y=1) \geq 1 \\ 0, & \hat{P}(Y=1) < 0.5 \end{cases}$$

The decision boundary is when $\mathbf{X}\vec{\beta} = 0$. We note that if the function is linear with the parameters, thn the boundary is linear. If you have a function like $\beta_1 X_1 + \beta_2 X_2 + \beta_3 X_1 X_2$, then we'd expect something nonlinear. For a **circle**, we would want:

$$\mathbf{X} = \{X_1, X_2, X_1^2, X_2^2, X_1 X_2\}$$

Regularization

We can just add the same Ridge Regression penalty term and have:

$$\beta^* = \arg\min_{\vec{\beta}} - \Big(\sum y_i \log(p_i) + (1-y_i)\log(1-p_i)\Big) + \lambda \sum_{j=1}^{p} \beta_j^2$$

You use cross validation to tune the parameter λ.

Multinomial Logistic Regression

The idea is that if you have K classes, you pick the K^{th} class to be the *reference* class. Thus, you will now model with respect to the reference class:

$$\log\left(\frac{P(Y=k)}{P(Y=K)}\right) = \beta_0 + \sum_{j=1}^{p} \beta_{j,k} X_j, \quad \forall k \in \{1,2,\ldots,K-1\}$$

Note that we are training $K-1$ models, when you look at the Sklearn output we note that it has K set of coefficients because it can calculate the probability of the last class, since the softmax of the probabilities needs to sum to 1. We let:

$$\hat{y} = \arg\max_k \hat{P}(Y=k)$$

One vs. Rest

The idea is that you run logistic regressions, by creating K models and letting $y_{\text{OvR }1} = 1$ if $y = k$, and $y = 0$ else. Thus, we are comparing against all the others:

$$\log\left(\frac{P(Y_{\text{OvR }1}=1)}{1-P(Y_{\text{OvR }1}=1)}\right) \beta_0 + \sum_{j=1}^{p} \beta_{j,k} X_j, \quad \forall k \in \{1,2,\ldots,K\}$$

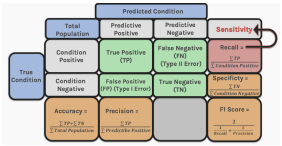
Then Sklearn normalizes by the sum of all the probabilities. And we have the following **cross-entropy or multinomial logistic loss**:

$$\ell = -\frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{K} I(y_i=k)\log(P(y_i=k)) + I(y_i \neq k)\log(1-P(y_i=k))$$

We are trying to minimize the negative log-likelihood which is above (it is already negative).

ROC

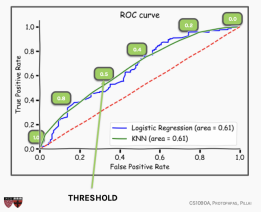
Diagnostic Testing metrics for classification:



We define the ROC curve to plot $x =$ **False Positive Rate** and $y =$ **True Positive Rate**, where:

$$TPR = \frac{TP}{TP+FN}, \quad FPR = \frac{FP}{FP+TN}$$

And we note that this is a parametric curve, where t is the threshold, i.e. if $P(Y=1) \geq t$, then classify $\hat{y} = 1$. The interpretation is in the following figure:



Casual Inference

There is potential for **confounding factors** to be the driving force for the observed association. There are 2 main approaches:

- Model all possible confounders by including them into the model (multiple regression, for **casual methods** to account for the confounder.
  - **Advantages**: cheap
  - **Disadvantages**: not all confounders may be measured
- A **randomized experiment** can be performed where the scientist manipulates the levels of predictor the levels of the predictor (**treatment**) to see how this leads to changes in the response.
  - **Advantages**: confounder will be **balanced**, on average, across treatment groups
  - **Disadvantages**: expensive, can be an artifical environment

What is Casual Inference? Inferring the effects of any treatment/policy/intervention/etc.
Simpson's Paradox: The idea is that there is a confounding variable that does not allow us to see that indeed one treatment is better than another always, but since the dataset is imbalanced, it is not true.
Association vs. Correlation: Association is a statistical dependence and correlation a linear statistical dependence.

Fundamental Problem of Casual Inference

Say you want to see how a person's response changes as we change the treatment, in other words, **casual effect**, we define it:

$$Y_i(1) - Y_i(0)$$

However, the issue is we either treat person i with treatment 0 or treatment 1, so we can never calculate that value. To fix this, we take averages and we can define the **Average Treatment Effect (ATE)**:

$$\tau = E[Y_i(1) - Y_i(0)] = E[Y(1)] - E[Y(0)] \neq E[Y \mid T=1] - E[Y \mid T=0]$$

In order to make a jump we need to make an assumption:
Stable Unit Treatment Values Assumption (SUTVA): the response/outcome of a particular subject depends only on the treatment to which they were assigned, not the treatments of others around them.
Violations:

- If there are any **spillover effects** between subjects (sometimes called **interference**): if one subject's treatment effects another subjet's response
- if the treatment is not consistent (sometimes called **hidden variations of treatments**): if a subject's treatment level may be different the second time around.

For example, we can conduct **Randomized Control Trials (RCT)**, thus we have that there is no confounding association from treatment to response. Thus for this we have:

$$E[Y(1)] - E[Y(0)] = E[Y \mid T=1] - E[Y \mid T=0]$$

Propensity Scores

To adjust for confounding with treatment assignment that may exist in an observational study, the **propensity score** can be estimated. We define the **Propensity Score** to be the probability that a subject is assigned a particular treatment, T_i, given their set of observed predictors, X_i.

$$e(X_i) = P(T_i = 1 \mid X_i)$$

Using Propensity Scores for Balancing

There are generally 3 approaches:

- Covariate Adjustment
  - The simplest way to use propensity score is to incorporate them into the main response model as **an additional (and only) predictor**:
  - This can result in a poorly adjusted model as the **estimand** is poorly defined.
- Weighting
  - Call it inverse probability weighting. The second simplest way to use propensity score is to incorporate them as sampling weights in to the main response model.
  - This can result in a poorly estimated causal effect as the sampling weights can be huge (if $\hat{e}(X_i)$ is close to 0 or 1)
- Matching
  - The best way to use propensity score is match on them: for every treated observation find the most similar control observation. This creates a synthetic counterfactual for every observed treated observation.

Decision Trees

Motivation is that the best case scenario for logistic regression is when the data has a nice geometric decision boundary, however that is not always the case, what if the **classes** are circles of squares. Note that technically we *could* have decision boundaries like:

$$x_1^2 + x_2^2 - 0.25 = 0$$

However, this is not interpretable very well. Want:

- Allow for complex decision boundaries
- Are also easy to interpret
- Are straightforward and efficient to compute

Simple Decision Trees

Splitting Criteria

Common guidelines we should go for:

- The regions in the feature space should grow progressively **purer** with the number of splits
- Should see that region 'specializes' towards a single class
- We should end up with **no empty regions** -every region should contain training points

Classification Error: We define the error as follows on a region R_r:

$$\begin{aligned}\text{ClassError}(R_r) &= \frac{\text{Number of minority class data points}}{\text{Total number of data points}} \\ &= 1 - \frac{\text{Number of majority data points}}{\text{Total number of data points}} \\ &= 1 - \max_k \Psi(k|R_r)\end{aligned}$$

$$\text{Error}(R_r \mid p, t_p) = 1 - \max_k \Psi(k \mid R_r)$$

We define $\Psi(k|R)$ to be the proportion of data points which are labeled as class k. We denote that we are splitting on a predictor x_p at threshold t_p.

We need to take a weighted average over both the region and the number of points in each region, thus we want to pick the split such that:

$$\arg\min_{p,t_p} \left(\frac{N_1}{N} \cdot \text{Error}(R_1 \mid p, t_p) + \frac{N_2}{N} \cdot \text{Error}(R_2 \mid p, t_p) \right)$$

Gini Index: We define the following Gini error function

$$\text{Gini}(R_r \mid p, t_p) = 1 - \sum_k \Psi(k \mid R_r)^2$$

The effect of the squaring of the proportions of each class is that it **accentuates the contrast between pure regions and mixed ones** enabling the index to better differentiate between quality splits. Thus, we can define the splitting criteria:

$$\arg\min_{p,t_p} \left(\frac{N_1}{N} \cdot \text{Gini}(R_1 \mid p, t_p) + \frac{N_2}{N} \cdot \text{Gini}(R_2 \mid p, t_p) \right)$$

Now taking a information theory approach we define the **Entropy error**:

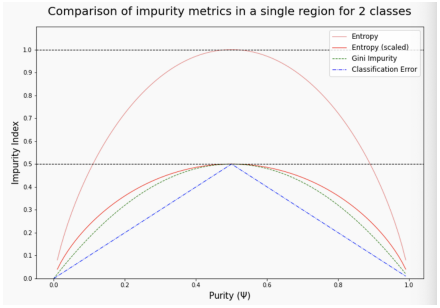
$$\text{Entropy}(R_r \mid p, t_p) = -\sum_k \Psi(k \mid R_r) \log_2 \Psi(k \mid R_r)$$

We can then look at the weighted spitting criteria:

$$\min_{p,t_p} \left(\frac{N_1}{N} \cdot \text{Entropy}(R_1 \mid p, t_p) + \frac{N_2}{N} \cdot \text{Entropy}(R_2 \mid p, t_p) \right)$$

Comparison of Criteria

We have that Entropy penalizes the most from looking at the graph:



Stopping Conditions

Maximum Depth: Most common stopping condition is to limit the maximum depth of the tree `max_depth`.
Purity: Don't split a region if all instances in the region belong to the same class.
Minimum Sample Leafs: Don't split a region if the number of instances in any of the sub-regions will fall below pre-defined threshold (`min_samples_leaf`).
Maximum Leaf Nodes: Don't split a region if the total number of leaves in the tree will exceed a predefined threshold (`max_leaf_nodes`).

These are types of growth on a tree seen from these stopping conditions:

- **Level-Order**: Nodes are split in a level-wise manner, starting from the root. At each step, all nodes at the current depth are evaluated and split simultaneously (if they meet the splitting criteria). After completing one level, the algorithm proceeds to the next level.
- **Best-First (Breadth-first)**: At each step, the algorithm selects the "best" node to split based on a criterion (e.g., information gain, Gini reduction, impurity decrease). Nodes are prioritized, and the one that maximizes the splitting criterion is split next, regardless of its depth in the tree.

Minimum Impurity Decrease: Compute gain in impurity of splitting a region R into R_1 and R_2. Do not split if the gain is less than some pre-defined threshold: M,

$$\text{Gain}(R) = \Delta(R) = m(R) - \frac{N_1}{N} m(R_1) - \frac{N_2}{N} m(R_2)$$

Split if $\text{Gain}(R) > M$, note that $m(R)$ can either be Gini, Entropy, Classification Error.

Variance vs. Bias:

- High Bias: Trees of low depth are not a good fit for the training data - it's unable to capture the nonlinear boundary separating the two classes.
- Low Variance: Trees of low depth are robust to slight perturbations in the training data - the square carved out by the model is stable if you move the boundary points a bit.
- Low Bias: With a high depth, we can obtain a model that correctly classifies all points on the boundary (by zig-zagging around each point).
- High Variance: Trees of high depth are sensitive to perturbations in the training data, especially to changes in the boundary points.

Regression using Trees

For Regression using trees we will now use the splitting criteria by using the MSE as an error function:

$$\arg\min_{p,t_p} \left(\frac{N_1}{N} \cdot \text{MSE}(R_1) + \frac{N_2}{N} \cdot \text{MSE}(R_2) \right)$$

Most of the stopping conditions can be reused from classification decision trees, but instead of impurity gain we can compute accuracy gain:

$$\text{Gain}(R) = \Delta(R) = \text{MSE}(R) - \left(\frac{N_1}{N} \cdot \text{MSE}(R_1) + \frac{N_2}{N} \cdot \text{MSE}(R_2) \right)$$

For the prediction we say for any point x_i:

1. Traverse the tree until we reach a leaf node
2. Predict $\hat{y}_i$ to be the averaged value of the response variable y's in the leaf (from the training set).

For categorical variables we can choose to do a ordinal encoding, but ideally we should do a **One Hot Encoding**.

Pruning

The major issue with pre-specifying a stopping condition is that you may stop too early or stop too late. We can fix this issue by **choosing several stopping criteria** and cross-validate to decide which one is the best.

Motivation for Pruning We can obtain a simple tree by **pruning** a **complex** one. There are many methods of pruning. A common one is the **cost complexity pruning**:

$$C(T) = \text{Error}(T) + \alpha|T|$$

We let T be a decision Tree, the error can be Classification Error, and the $\alpha \cdot |T|$ is the regularization term. Thus, we want to prune a tree at some node, (could by high up node) such that:

$$\arg\max_{T^*} \left(C(T) - C(T^*) \right) \equiv \arg\min_{T^*} \left(\frac{\text{Error}(T) - \text{Error}(T^*)}{\alpha|T| - \alpha|T^*|} \right)$$

The idea is that you run the following algorithm:

- Let $T = T^{(0)}$, $k = 0$
- While $|T^{(k)}| > 1$:
  - $T^{(k+1)} = \arg\min_{T^*} \left(\frac{\text{Error}(T^{(k)}) - \text{Error}(T^*)}{\alpha|T^{(k)}| - \alpha|T^*|} \right)$
  - $k \to k + 1$
- Choose best tree, $T^{(i)}$ using cross-validation.

Now we cross-validate also on α to get the ideal α.

Bagging

Motivation: Decision tree models often under perform when compared to other classification or regression methods in situations of complex decision boundaries.

Ensemble Learning: The idea of ensemble learning is to build a single model by training and aggregating multiple models. You have a lot of collective decisions and you aggregate them to form a final verdict.

Bagging = Bootstrap + Aggregating:

1. **Bootstrap**: We generate multiple samples of training data, via bootstrapping. We train a deep decision tree on each sample of data.
2. **Aggregating**: For a given input, we output the averaged outputs of all the models for that input.

Advantages of Bagging: Bagging enjoys the benefits of:

1. **High expressiveness** - by using deeper trees each model is able to approximate complex functions and decision boundaries.
2. **Low variance** - averaging the prediction of all the models reduces the variance in the final prediction, assuming that we choose a sufficiently large number of trees.

Disadvantages of Bagging:

1. **Interpretability**: A major drawback of bagging and other ensemble methods is that the averaged model is no longer easily interpretable, i.e. one can no longer trace the logic of an output through a series of decisions based on predictor values.

For bagging, the training error does not reach 0. However, the validation error is less than the decision tree error. We see these changes and you increment the max depth of the tree. Note that bagging can still **underfit** if the depth of the trees are too late to capture true relationships.

Out-of-Bag Error

The idea is that **cross-validation** is too expensive computationally, and it uses a larger dataset. So instead we look at Out of Bag error for bagging.

Calculating OOB Error:

- Generate bootstrap samples, $\mathcal{A}^1, \ldots, \mathcal{A}^n$ for your N trees.
- Let $\mathcal{B} = (X, y)$, the set of all training data observations, denote $U^i = \mathcal{B} \cap \mathcal{A}^i$ be the unused data points in the each sample.
- Create decision trees models for each.
- For every $y_{i,j} \in U^i$ calculate
  - For classification: $e_{i,j} = I(\hat{y}_{i,j} \neq y_{i,j})$
  - For regression: $e_{i,j} = (y_{i,j} - \hat{y}_{i,j})^2$.
- Then average the point-wise out-of-bag errors on full training set:
  - For classification: $\text{Error}_{\text{OOB}} = \frac{1}{N} \sum_i I(\hat{y}_{i,j} \neq y_{i,j})$:
  - For regression: $\text{Error}_{\text{OOB}} = \frac{1}{N} \sum_i (y_{i,j} - \hat{y}_{i,j})^2$:

OOB Error prevents leakage and yields a better model with lower variance or less overfitting. There is also lesser computational cost for OOB as compared to CV for bagging.

In practice, the trees in Bagging tend to be **highly correlated**. If there exists some strong predictor x_j each tree would probably split of x_j. However, we would have hoped that each tree is **independently and identically distributed**.

Data Imbalance

Accuracy is a great measure but only when you have **balanced datasets** (false negatives & false positives counts are close). For imbalance datasets, we say F1-Score is a better metric:

$$F_1 = \frac{1}{\frac{1}{\text{Recall}} + \frac{1}{\text{Precision}}} = \frac{2 \cdot \text{Recall} \cdot \text{Precision}}{\text{Recall} + \text{Precision}}$$

Looking at the ROC curve, if the cost of false negatives and false positives are different, the ROC curve allows us to find the classification threshold that gives the best trade-off between FP rate and TP rate which we need in this case.

Dealing with Imbalanced Classes: There are three main ways of dealing with imbalanced class: undersampling, oversampling, and class weighting.

Undersampling

We can reduce the number of samples in the majority class to match the number of samples in the minority class. This can be done in two ways:

1. **Random Sampling**: Randomly sample from majority class with or without replacement (issue may be that you are choosing points that are not informative).
2. **Near Miss**: Select data points by using simple heuristics like finding samples from which the average distance to some data points of minority class is smallest (idea is that you are picking points that are more informative).

Oversampling

We fight imbalanced data by generating new samples for minority class can be done in two ways:

1. **Random Sampling**: Randomly sample from minority class with replacement.
2. **SMOTE**: SMOTE is an improved alternative for oversampling. SMOTE works by finding points that are closer in feature space. Drawing a line between these points and generating new data points along this line.

Class Weighting

A simple way to address the class imbalance is to provide a weight for each class which places more emphasis on the minority classes. In sklearn we can provide the class weight as a dictionary or use `class_weight = balanced`. Then it automatically adjust weights inversely proportional to class frequencies in the input data as:

$$W_k = \frac{N}{K \cdot N_k}$$

where N is the total number of samples, and N_k is the number of samples in class k and K is the number of classes.

Random Forest

Motivation: All the trees in the Bagging method are too correlated, thus all the trees basically look alike. Thus, we want to de-correlate the trees such that the trees are close to independent.

Random Forest works in the following way:

1. Create B bootstrapped datasets with all J predictors (same as bagging)
2. Initialize a random forest with B decision trees.
3. For each tree, at each split, we randomly select a subset, J' from the full set of predictors, ($J' \subset J$)
4. Amongst the J' predictors, we select the optimal predictor and the optimal threshold for the corresponding split.

Tuning Random Forests: The random forest model have multiple hyper-parameters to tune:

1. The number of predictors to randomly select at each split. Standard (default) values that are recommended: $\sqrt{N_j}$ predictors for classification, and $N_j/3$ for regression where N_j is the number of predictors.
2. The total number of trees in the ensemble. Standard (default) values that are recommended: once the OOB error stabilizes you do not need to add more trees.
3. The stopping criteria - maximum depth, minimum leaf node size, etc.
4. The splitting criterium - gini, entropy

Variable Importance

Mean Decrease in Impurity (MDI): Decision trees make splits that maximize the decrease in impurity. By calculating the **mean decrease in impurity for each feature** across all trees, we arrive at the variable importance for a bagging or random forest model.

Algorithm for MDI:

1. Calculate the mean decrease in impurity for each node q in the decision tree:

$$\Delta I_q = \frac{n}{N} \left(\text{Gini}_q - \frac{m_L}{n} \text{Gini}_{m_L} - \frac{m_R}{n} \text{Gini}_{m_R} \right)$$

2. Calculate the importance of the feature by summing the impurity decrease calculated in step 1 for each node in which that feature occurs (for some predictor q:)

$$\text{Feat. Importance}_p = \sum_{q \in \text{nodes split on } q} \Delta I_q$$

3. Normalize this value between 0 and 1 by dividing by the sum of all the feature importances:

$$\text{N. Feat. Importance}_p = \frac{\text{Feat. Importance}_p}{\sum_{j \in \text{Predictors}} \text{Feat. Importance}_j}$$

4. To calculate the feature importance at the Random Forest of Bagging level, we average the normalized feature importance of the given predictor over all the trees:

$$\text{RF N. Feat. Importance}_p = \frac{\sum_{t \in \text{Trees}} \text{N. Feat. Importance}_{p,t}}{\text{Total number of trees}}$$

The biggest advantage of the MDI is the **speed of computation**. All needed values are computed during the RF training. The drawbacks of the method is its tendency to prefer numerical features and categorical features with **high cardinality**, since it can split multiply times instead of a categorical variable with 2 categories.

Permutation Importance: For each feature j in the dataset:

1. Record the validation/OOB (unpermutated) accuracy of RF model: s.
2. For each repetition in $k \in \{1, \ldots, K\}$: **Randomly permute** the data for the column j. Record the validation/OOB accuracy $s_{k,j}$ of the RF model on the modified dataset.
3. Compute the average accuracy from all the permuted datasets:

$$s_j = \frac{1}{K} \sum_{k=1}^{K} s_{k,j}$$

4. Calculate the difference between the unpermutated and average permuted accuracy to get the importance of the feature in the random forest:

$$\text{RF Feat. Importance}_j = s - s_j$$

The features with the largest feature importance are denoted to be more important. We note that variable importance for **RF is smoother** than that for **bagging** due to randomness introduced by selecting a subset of predictors to choose from.

Final Thoughts on Random Forests

When the **number of predictors is large**, but the **number of relevant predictors is small**, random forests can perform **poorly**, since the probability that it will select a relevant predict is low and the trees in the ensemble will be weak.

Increasing the number of trees in the ensemble generally does not increase the risk of overfitting. Also, RF and Bagging can return probabilities by taking the proportion of trees in the forest that predict a certain class.

If you have missing data, we can do **surrogate splits**. Let's say we have some missing values for height, then we can use weight to split instead. The idea is that there might be predictors that are correlated thus when splitting on a predictor that we do not have data on, we rely on the surrogate.

Boosting

Motivation: Shallow trees suffer from high bias and do not train well. Deep trees have low bias but suffer from high variance leading to very low generalization error.
Issues with RF:

- Although variance reduction is better in RF than bagging, the generalization error is still high.
- Large number of trees can make the algorithm very slow and ineffective for real-time predictions.

Rather than performing **variance reduction** on complex trees, can we decrease the bias of simple trees - make the more expressive (learn from our mistakes).

Boosting methods are general algorithms which combine several **weak learners** to product a strong rule.

Gradient Boosting

The key intuition behind boosting is that one can take sum of simple models. **Gradient boosting** is a method for iteratively building a complex model T by adding simple models. Each new simple model added to the ensemble compensates for the weakness of the current model.

$$T = \sum_h \lambda_h T_h$$

The following is the algorithm:

1. Fit a simple model $T^{(0)}$ on the training data $\{(x_1, y_1), \ldots, (x_N, y_N)\}$, and set $T \leftarrow T^{(0)}$.
2. Compute the residuals $\{r_1, \ldots, r_N\}$ for T.
3. For $i = 1, \ldots$ until stopping condition is met:
    - Fit a simple mode, $T^{(i)}$, to the current **residuals**, that is train $T^{(i)}$ using $\{(x_1, r_1), \ldots, (x_N, r_N)\}$
    - Set the current model $T \leftarrow T + \lambda T^{(i)}$.
    - Compute residuals at step i, set $r_n - \lambda T^{(i)}(x_n)$, $n = 1, \ldots N$, where λ is the learning rate.

Choosing λ: if λ is constant, then it should be tuned through cross validation. For better results use a variable λ so we will let it be a function of the gradient:

$$\lambda = h(||\nabla f(x)||)$$

Termination: We can terminate gradient descent:

- We can limit the number of iterations in the descent. But for an arbitrary choice of maximum iterations, we cannot guarantee that we are sufficiently close to the optimum in the end.
- If the descent is stopped when the updates are sufficiently small (e.g. the residuals of T are small), we encounter a new problem, the algorithm may never terminate!

AdaBoost

The two main ideas in AdaBoost:

1. Iteratively build a complex model T by combining several weak learners. For trees, a weak learner is a tree with 1 node with 2 leaves. We call this a stump.
2. Each new stump added to the ensemble model T learns from the mistakes of the current model. This is done by assigning **higher weights** to the model that is **incorrectly classifies**.

We conduct the following algorithm:

- Given training data $\{(x_1, y_1), \ldots (x_N, y_N)\}$, choose an initial distribution $w_n^{(0)} = 1/N$. For $i = 0, \ldots$ until stopping condition is met:
    1. Train a weak learner $S^{(i)}$ a stump, using weights, $w_n^{(i)}$.
    2. Calculate the total error of the weak learner using:
    $$\epsilon^{(i)} = \sum_{n=1}^{N} w_n^{(i)} I(y_n \neq S^{(i)}(x_n)$$
    3. Calculate the importance of each model $\lambda^{(i)}$.
    4. Construct the ensemble model using:
    $$T^{(i)} \leftarrow \begin{cases} \lambda^{(i)} S^{(i)}, & i = 0 \\ T^{(i-1)} + \lambda^{(i)} S^{(i)} & i = 1, 2, \ldots \end{cases}$$
    5. Adjust the weights assigned to each data point to ensure the next stump focuses on the points misclassified by the previous stump:
    $$w_n^{(i+1)} \leftarrow \frac{w_n^{(i)} e^{-\lambda^{(i)} y_n T^{(i)}(x_n)}}{Z}$$
    where $T^{(i)} = \text{sign}(T^{(i-1)}(x) + \lambda^{(i)} S^{(i)}(x))$
    6. **Final Model**:
    $$T^{(L)}(x) = \text{sign}\left(\sum_{i=1}^{L} \lambda^{(i)} S^{(i)}(x)\right)$$

To use the normalized weights to make the stump there are two options:

- Create the new dataset of same size of the original dataset with **repetition** based on the newly updated sample weight.
- Use a weighted version of the Gini index

In AdaBoost, we minimize a different function, we call **exponential loss**:

$$\text{Exp Loss} = \frac{1}{N} \sum_{n=1}^{N} e^{-y_n \hat{y}_n}, \quad y_n \in \{-1, 1\}$$

Unlike the case of gradient boosting for regression, we can analytically solve fro the optimal learning rate for AdaBoost by optimizing:

$$\arg\min_{\lambda} \frac{1}{N} \sum_{n=1}^{N} e^{-y_n (T + \lambda^{(i)} S^{(i)}(x_n))}$$

If we do that we get the following:

$$\lambda^{(i)} = \frac{1}{2} \ln\left(\frac{1-\epsilon}{\epsilon}\right), \quad \epsilon^{(i)} = \sum_{n=1}^{N} w_n^{(i)} I(y_n \neq S^{(i)}(x_n)$$

For **Gradient Boost** the update step is:

$$\hat{y}_n \leftarrow \hat{y}_n + \lambda \hat{r}_n$$

For **AdaBoost** the update step is:

$$\hat{y}_n \leftarrow \hat{y}_n + \lambda w_n y_n, \quad w_n = e^{-y_n \hat{y}_n}$$

Final Thoughts on Boosting

Stopping Condition: Same as gradient boosting: maximum iteration (number of stumps), minimum improvement in loss, number of iterations without improvement in the loss.
Overfitting: Unlike other ensemble methods like bagging and RF, boosting methods like AdaBoost will overfit if run for many iterations. Some libraries implement regularization method swhich is out of the scope of this course.
Hyper-parameters: All parameters associated with the stump and the stopping conditions above.

Other implementations:

- **XGBoost**: An efficient Gradient Boosting Decision.
- **LGBM**: Light Gradient Boosted Machines. It is a library for training GBMs developed by Microsoft, and it competes with XGBoost.
- **CatBoost**: A new library for Gradient Boosting Decision Trees, offering appropriate handling of categorical features.

Blending

Blending trains **heterogeneuous learners**, (different models) on the dataset in parallel, and it further trains a meta-model on the base models' outputs for making predictions. The following is the algorithm:

1. Split the dataset into:
    - Training Set
    - Validation Set
    - Holdout Set
    - Test Set
2. Fit base models on Training Set and validate with Validation Set.
3. Generate base Validation Model Predictions on Validation Set.
4. Concatenate predictions and the original data into a new dataset, i.e. X from Validation Set and Validation Model Predictions = X.
5. Combined features from Validation Set, X, will be used for Training Meta Model.
6. Generate Holdout Model Predictions on Holdout Set.
7. Concatenate predictions and the original data into a new dataset, i.e. X from Holdout Set and Holdout Model Predictions = X.
8. Combined features from Holdout Set, X will be used for Validating Meta Model.
9. Train Metal Model using X, y and validate using X, y.
10. For inference use Test Set, generate X and see final predictions of Meta Model, and compare to y.

Compared with Bagging and Boosting, Blending provided flexibility:

- Allows combining models of different types
- Meta model learns how to best combine diverse model outputs

Blending models can be **more interpretable** if meta-model is **simple (e.g. linear regression)** where contribution of each base model can be explicitly visible. However, we are dividing the dataset into more pieces for training blending models, the amount of data for training meta model is only the size of the validation set (usually 10%-25% of original data).

Stacking

Like Blending, Stacking also trains **heterogeneous learners** on the dataset, and a **meta model** is fit with base models' predictions. The difference is how we construct data for meta model.

1. Similar to blending, we first split the dataset into:
   - Training Set
   - Holdout Set
   - Test Set
2. In cross validation, we need to split the
3. Training Set into folds. For simplicity here, let's assume we want $k = 3$ folds: (X_1, y_1), (X_2, y_2), (X_3, y_3)
4. Fit base models with CV. While we do validation with each fold, collect these predictions to build our training set for meta model. For each fold, pick it to be validation fold, and the other two to be the train folds.
5. Let (X_1, y_1), $(X_2, y_2) \rightarrow (X_1, y_1)$, (X_2, y_2) be the train folds, and $(X_3, y_3) \rightarrow (X_3, y_3)$ be the validation fold.
6. Train on models using train folds.
7. Validate models using validation fold, keep predictions as $\hat{y}_3$.
8. Get predictions from those models that were just trained, $\hat{y}_{(3)}$ (the 3 is for the 3rd fold being excluded).
9. Repeat for excluding every fold.
10. Use $\mathbf{X} = (X_1, \hat{y}_1, X_2, \hat{y}_2, X_3, \hat{y}_3)$ as the Training Set for Meta Model (prediction on y).
11. Use $\mathbf{X} = (\hat{y}_{(1)}, \hat{y}_{(2)}, \hat{y}_{(3)}, X)$ as the Validation Set for Meta Model (prediction on y).
12. Retrain all the base models, with the entire training set: Training Set and Holdout Set.
13. For inference pass X through the base models, calculate $\hat{y}$
14. Pass $\mathbf{X} = (X, \hat{y})$ into Meta Model
15. Comparing final predictions with y.