1 Chapter 9: Classifiers

1.1 Playing with fire

On an August afternoon in 2007, a fire broke out on the 17th floor of the then-vacant Deutsche Bank Building, a skyscraper situated across from the former World Trade Center in New York City. The building, seriously damaged after the 9/11 attacks, had been undergoing hazard abatement and controlled demolition, leading to changes to the building floor plans and safety apparatus. When the New York City Fire Department (FDNY) responded to the scene, it was clear the fire was a serious one, quickly escalating to a seven-alarm fire incident requiring 87 units and 475 firefighters. [1] As standpipes had been disabled and floor plans altered, FDNY units found it difficult to navigate the skyscraper and put water on the fire, resorting to unconventional methods of supplying water to crews. Eventually, the fire was put out seven hours after it started, not before two firefighters lost their lives, succumbing to cardiac arrest from heavy smoke inhalation. [2] In response to the tragedy, a mayoral investigation found that the deaths could have been prevented had city agencies established information sharing protocols and leveraged a risk-based strategy to mitigate and avoid hazards. [3] While an ideal end state would be to end all structural fires, the recommendations focused on reducing death and injury by ensuring that FDNY had the most up-to-date ground intelligence.

Risk mitigation strategy was indeed due for improvements. Since the 1950's, FDNY building inspections were managed using a manual index card system where inspection schedules were based on tiers of perceived risk, where the riskiest buildings needed to be inspected once a year and the least risky buildings were inspected once every seven years. While it was a longstanding process, it left a shortfall in inspection coverage. Of the one million buildings in New York City, only one-third are inspection-qualified. Of those 300-thousand buildings, FDNY had historically been only able to inspect at most 10% of the buildings in a given year due to other operational priorities. This meant that even on a seven-year schedule, not all buildings would be reached and the fixed timeline meant that both perfectly safe and guaranteed fire traps had equal chance of being inspected. This could be easily changed. By incorporating the latest information about where fires did and did not occur and associating it with building characteristics, a new data-driven strategy could direct how buildings are prioritized for inspection.

In 2013, the New York City Fire Department (FDNY) set out to address the risk management problem by melding data and technology with their field operations. On the surface, the idea of using data and technology to reduce the risk of fire is quite alluring. However, under the hood, there were notable obstacles. On the operational side, buy-in was required. Anyone who has observed firefighters on scene will notice that it is a well-choreographed operation – every person knows their part and abides by the established protocols as directed by leadership. For data to drive value, it needed to be integrated and accepted into the culture of a 10,000+ person fire fighting organization. On the technical side, decades worth of index cards needed to be digitized and a scheduling platform needed to be developed. Perhaps most importantly, the system had to work. Scheduling just any inspection is simple. But scheduling inspections to buildings with observable fire risks is far more challenging as such as system would need to be able to distinguish between fire-prone and fire-proof buildings. Without effective targeting, the entire effort would be for naught.

The Commissioner and First Deputy Commissioner at the time both believed that technology had a role to play at FDNY. Aligned with Mayor Michael Bloomberg's vision of smart, data-driven government, they saw an opportunity to set an example for the nation's fire services. They relied on the the Assistant Commissioner for Management Initiatives to lead a change management process with fire chiefs and fire officers, information technology (IT) managers, among others to change the flow of operations so that data served as a pillar on which FDNY could rely. Alliances were forged with leading fire personnel such as the Deputy Chief of Fire Operations and Battalion Chiefs to formalize the role of data in the culture of the fire house, amending standard operating procedures (SOPs) to use a digital inspection system. On the IT front, a lead software engineer and project manager meticulously gathered specifications that were then used to construct a scheduling platform. Recognizing that the proof was in the pudding, a Director of Analytics was hired to lead

[1] https://cityroom.blogs.nytimes.com/2007/08/18/2-firefighters-are-dead-in-deutsche-bank-fire/
[2] http://www.nydailynews.com/news/firefighters-dead-7-alarm-deutsche-bank-blaze-article-1.238838
[3] http://www1.nyc.gov/assets/doi/downloads/pdf/pr_db_61909_final.pdf

the overhaul of a prediction algorithm to rank buildings based on their risk and convincing stakeholders that a statistical representation of fire ignition was indeed trustworthy. The result was the Risk-Based Inspection System (RBIS), a firefighter-facing data platform that scheduled inspections at buildings with the greatest risk of fire. Three times a week for three hours per session, fire officers logged onto RBIS to obtain a list of buildings for scheduled inspection. Buildings were selected using FireCast, a statistical algorithm developed in-house to predict fires at the building level. Through FireCast, buildings no longer used assumed a static risk classification as in the index card system, but rather a dynamic risk score took into account the latest information.

Prediction often relies on accuracy measures to determine how well algorithms perform in the field; FireCast was no different. The algorithm was able to identify buildings with fires over 80% of the time – a degree of accuracy that superceded prior attempts at the problem. Upon implementing the new system, impacts were observed in leading operational indicators. In the first month, the number of safety violations issued grew by +19% relative to the trend under the index card system, but fell to +10% in the second month. This indicated that the riskiest buildings did indeed have more observable risks than less risky buildings, but the amount of observable risk fell as building inspection teams progressed down the risk list.

From a statistical perspective, the prediction should have yielded far more violations, but efficacy of the prediction program was limited by (1) a fire unit's time budget to conduct inspections; (2) a policy requiring that time had to be set aside for weekly inspections, which at times led to inspecting buildings that were not observably risky after all truly risky buildings were exhausted; (3) the rule of law giving residents the right to refuse inspection. To measure efficacy, FDNY developed an indicator known as the Pre-Arrival Coverage Rate (PACR), which measures the proportion of buildings that experienced a fire that were inspected within some period (90 days) before the fire occurred – essentially measuring if fire companies had the opportunity to evaluate risks of priority buildings. Under FireCast, FDNY had achieved a PACR of 16.5%, which was an eightfold improvement over the old strategy that yielded 1.5%. [4] [5]

Since Firecast was launched in 2014, other fire prediction efforts have emerged around the United States such as the Firebird open source system for Atlanta in 2016[6] and a spatio-temporal fire prediction approach for Pittsburgh in 2018[7].

1.2 What's a classifier?

The RBIS/FireCast is an example of a *classification* problem – a task in which a model determines which group or *class* does an observation belongs based on its attributes, doing so based by learning from known examples. Examples must include the factual (what happened) and counter-factual (what did not happen) in order to distinguish between potential fires from non-fires.

Classification is nothing new in everyday life as we use our own mental classification models to contextualize the world around us. For example, marketers and advertisers are always looking to get product offerings in front of prospective customers and will often purchase lists of people and apply models based on past customer behavior in order to identify those who are most likely to be interested.[8] The criminal justice system has incorporated risk classifcation models to determine if those involved in alleged crime pose a flight risk if bail is posted.[9] On a more futuristic front, the technology behind self-driving cars uses a complex array of sensors and cameras that are processed by classifiers in order to distinguish between cars, people, motorbikers and cyclists.

The same is true with fires.

[4] http://www.nfpa.org/news-and-research/publications/nfpa-journal/2014/november-december-2014/features/in-pursuit-of-smart

[5] https://www.nist.gov/publications/research-roadmap-smart-fire-fighting

[6] http://firebird.gatech.edu/

[7] http://www.kdd.org/kdd2018/accepted-papers/view/a-dynamic-pipeline-for-spatio-temporal-fire-risk-prediction

[8] ref needed

[9] ref needed from Vera Institute

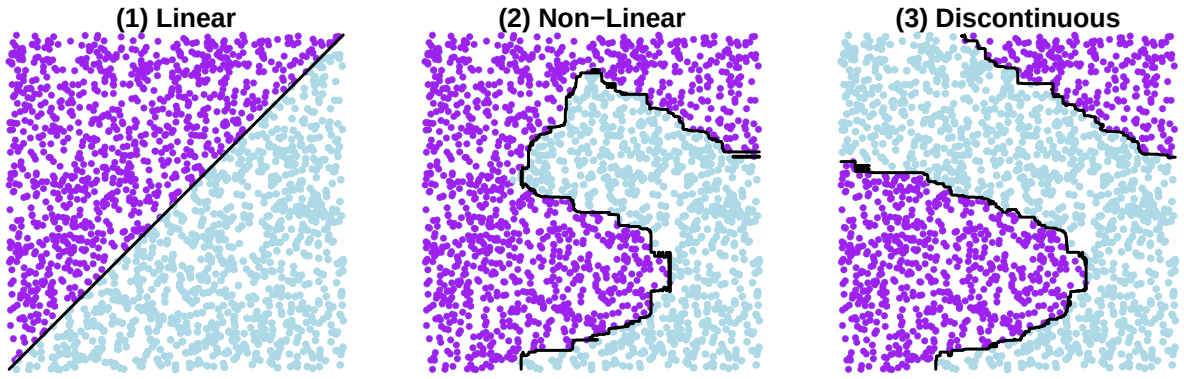


Figure 1: Types of classification problems

By examining buildings that have and have not caught fire in the past, we are able to learn what pattern characteristics are associated with greater risk of fire. We can *score* or apply the learned pattern to new records to obtaining the probability of fire so that fire fighters could have a sneak peak of what may happen to other buildings as their conditions change. There may be thousands of variables that can play a role in predicting fires.

If fires are truly predictable, we can employ supervised learning to map how input variables can distinguish buildings that had fires from those that did not. Otherwise stated, which fire status class does a building likely belong? Given a binary outcome $Y(Fire)$ we can determine class membership as a function of the building's characteristics:

$$Y(Fire) = f(\text{Building characteristics, Location, Complaints, ...})$$

It sounds simple enough, but as it turns out, there are many different ways that algorithms can associate building characteristics to fires. We can illustrate the complexities of this task under three hypothetical scenarios, plotting two input variables on the X and Y axes and color-coding two classes in purple and light blue. The solid black lines represent the true *decision boundary*, or the threshold at which an observation is classified in a specified class.

At first, the idea of classifying records may seem straight forward, but can become a rabbit hole. In policy, we tend to start from a normative theory of how a phenomenon functions – perhaps a simple explanation backed by linear logic. For example, a building that is bigger and older may be a reasonable working hypothesis for identifying high risk buildings. But as we investigate more and deeper, we may add exceptions to the rule as we discover cases that do not conform. Those exceptions may improve the number of correctly classified records, but detract from the simple narrative.

The function that determines the decision boundary can take on many forms and each balances interpretability and accuracy. A simple linear boundary can be represented using a *logistic regression* – the workhorse of the natural and social sciences. While the method provides an analytically convenient answer, the resulting decision boundary may miss finer, more disjointed patterns. Non-parametric methods offer a far more flexible solution to prediction, but at the cost of interpretability. A simple technique known as *k-nearest neighbors* is useful when k-number of similar, comparable observations can serve as a reference for informing the prediction of a given point. *Decision tree learning* along with its many variants such as *Random Forest* learn patterns by partitioning a sample into finer more homogeneous sub-samples. When faced with infinite distributions of data, each method responds to the circumstances differently.

Each technique has its own set of assumptions that in turn make it better suited for certain policy problems and certain types of data. But there are basic considerations that underlie our use of any classifier, namely (1) the idea of separability, (2) balancing interpretability with prediction, and (3) the different definitions of accuracy.

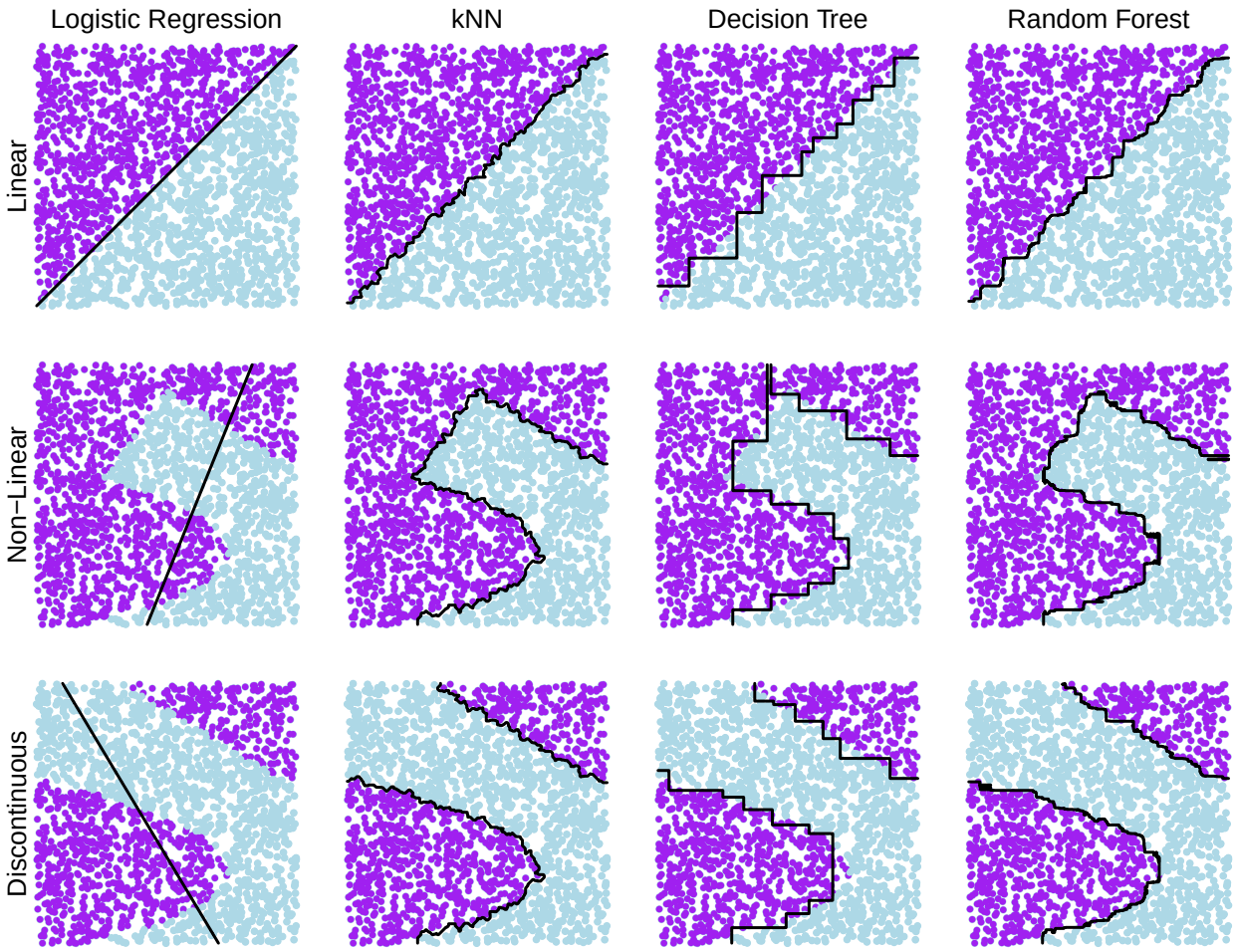


Figure 2: Linear, Non-Linear and Discontinuous Classification Problems.

1.2.1 Separability

A fire prediction algorithm needs to actually predict where fires and non-fires will be. A bail algorithm needs to be able to distinguish between those who are flight risks and those who are not. Simply training a classifier will not do. The success of a classifier fundamentally lies in if the classes are *separable* based on its input variables – basically the idea that the algorithm can learn from the underlying data and distinguish one class from another. Successfully distinguishing between classes can be measured through various forms of accuracy, but fundamentally is dependent on finding as many *True Positives* (TP) and *True Negatives* (TN) as possible.

Separability manifests itself differently depending on the data type. When the inputs are discrete variables, the goal is to find a set of variables that can best describe the target. In the example below, we analyze how building violations are related to the target fires. The ideal situation is to place the mass of sample along the diagonal with TNs (no fire - no violation) and TPs (fire - violation). In this case, whether a building had a building violation can accurately distinguish 80% of observations.

Table 1: Separability of two classes given a separable discrete variable.

	No Violation	Violation - T
Fire	10.9	22.6
No Fire	59.7	6.8

Alternatively, a low separable case may will have a large proportion of observations along the other diagonal, indicating many *False Positive* (Type I error) and *False Negatives* (Type II error). We can see that the majority of cases have been incorrectly classified. Classification problems are a balancing act: it is unlikely that a model will yield 100% accuracy everytime and often times, we will need to weigh accuracy against classification error. As we will see later in the chapter, more True Positives may also lead to more False Positives and how we balance these are very much a policy decision.

Table 2: Separability of two classes given an non-separable discrete variable.

	Elevator Apartment - F	Elevator Apartment- T
Fire	2.3	31.2
No Fire	6.8	59.8

When inputs are continuous variables, separability can be assessed in terms of means and dispersion of the target classes. For example, a low separability scenario (left) would be one where the input variable's distribution for each class substantially overlap, which suggests an absence of any distinguishing information. High separability (middle), in contrast, would have means that are significantly different from one another and the distributions themselves overlap minimally. But perhaps the neatest thing is perfect separability (right). In this case, we basically can produce a perfect definition of the outcome in question, in which case we do not need a classifier to model the relationship – just a threshold to serve as the definition.

The bottom line about separability is that a good data scientist will check their assumptions. In policy, there will be prevailing theories and conventional wisdom that dictate how certain factors influence a phenomenon. But what may sound good may in actuality have little separability and in turn offer little predictive power. In those cases, it is worth revisiting and revising the assumptions. After all, the goal of a classifier is to classify.

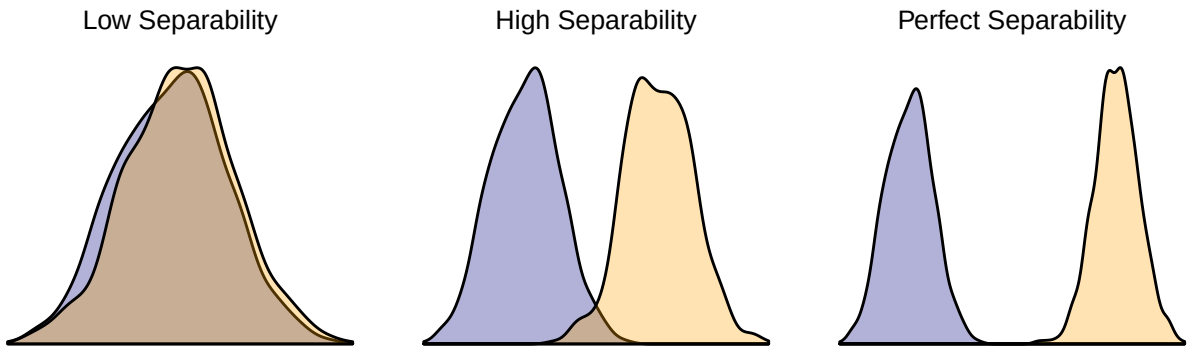


Figure 3: Separability of two classes given a continuous variable.

1.2.2 Measures of accuracy

Accuracy is central to evaluating if a classifier's results are useful. This requires understanding of a minimum of two ideas: what does probability have to do with accuracy and what exactly is the subtance of accuracy.

Predicted Probability. Most classifiers output a form of conditional probability. Accuracies are derived from probabilities produced by the classifier, indicating whether a given observation is predicted to below to a given class (e.g. fire vs. no fire, Yankee fan vs. Red Sox fan, Sith vs. Jedi, Republican v. Democrat). In order to convert probabilities into a prediction of a class, we need to set a threshold. So, what is the classification threshold?

Short answer: It depends on the sample.

For balanced samples in which the outcome variable's classes are represented in approximately equal proportion, the threshold is $pr(Y = 1) \geq 0.5$ as the objective probability of occurrence is close to 0.5. In reality, this may not be all that common scenario due to *class imbalance* and *rare events*. Class imbalance is the case in which the *minority class* – the class with fewer observations – is proportionally less prevalent in the sample than other classes. For example, political campaigns often deal with voter data in districts that are overwhelming in favor of one party over another. Rare events are infrequent events in which there is a super minority of observations. Fires and disasters tend to fall into this category. In some cases, fewer than $n = 200$ observations may lead to biased estimates.[10]. We will revisit class imbalance and rare events later in this chapter, but for now, let's assume that the classification threshold is $Pr(Y = 1) \geq 0.5$.

Accuracy. Classifier accuracy measures rely on metrics derived from the *confusion matrix*, or a $n \times n$ table where the rows represent actual classes and columns represent predicted classes. For a two class problem, the confusion matrix is a 2×2 table.

Table 3: Structure of confusion matrix

	Predicted (-)	Predicted (+)
Actual (-)	True Negative (TN)	False Positive (FP)
Actual (+)	False Negative (FN)	True Positive (TP)

Each cell of the confusion matrix is a building block required to calculate accuracy:

- The True Positive (TP) is the number of cases in which the actual positive observations ($Y = 1$) are correctly predicted (e.g. model predicts a fire and a fire actually occurs). Note that P is used to denote the total number of positive records.
- The True Negative (TN) is the number of cases where the actual negative observation ($Y = 0$) are correctly predicted. Note that N is used to denote the total number of negative records.
- The False Positive (FP) is number of cases where the actual label was $Y = 0$, but the model classified a record as $\hat{Y} = 1$. This is also known as *Type I error*.
- The False Negative (FN) is number of cases where the actual label was $Y = 1$, but the model classified a record as $\hat{Y} = 0$. This is also known as *Type II error*.

In a perfectly balanced sample of $n = 100$, we would expect the $P = 50$ and $N = 50$. Likewise, a perfect predictions should yield $TP = P$ and $TN = N$ along the diagonal. This is a rare case, but it stands that the goal is to check if the download diagonal captures the majority of observations.

In contrast, a model with little predictive power will have the majority of observations in the upward diagonal. Below, the matrix suggests the trained model has little predictive power as it classified the vast majority of records as negative even when some should have been positive.

Table 5: Confusion matrix for poor predictions.

	Predicted (-)	Predicted (+)
Actual (-)	40	10
Actual (+)	45	5

Despite having well-defined building blocks of accuracy, the tricky thing is that there is not just one measure of accuracy or error. In fact, there are many measures of accuracy that emphasize very different predictive qualities.

^^ADD EXAMPLE HERE

Measure	Formula	What It Answers
Individual Measures		
True Positive Rate (TPR), Sensitivity, or Recall	$TPR = \frac{TP}{TP+FN}$	What proportion positive cases were correctly identified?
True Negative Rate (TNR) or Specificity	$TPR = \frac{TN}{TN+FP}$	What proportion negative cases were correctly identified?
False Positive Rate (FPR)	$FPR = \frac{FP}{FP+TN}$	What proportion of negative cases were incorrectly predicted as positive? This is also known as the false detection rate or Type I error.
False Negative Rate (FNR)	$FNR = \frac{FN}{TP+FN}$	Proportion of positive cases that were incorrectly predicted negative. This is also known as the false alarm rate or Type II error.
Predicted Positive Value (PPV) or Precision	$PPV = \frac{TP}{TP+FP}$	Proportion of positive cases that were incorrectly predicted negative.
Overall Measures		
Accuracy (ACC)	$ACC = \frac{TP+TN}{n}$	What proportion of records were correctly classified?
F1-Score (F1)	$F1 = \frac{2}{\frac{1}{TPR} \times \frac{1}{FPR}}$	Alternative method of calculating accuracy using a harmonic mean
Area Under the Curve (AUC) or Concordance statistic	$AUC = \int_0^1 ROC(u)du$	The AUC is the area under the Receiving-Operating Characteristic (ROC) Curve. The ROC is derived by plotting values of TPR (Sensitivity) and FPR for varying probability cutoffs c.

1.2.3 Interpretability versus prediction

When selecting models for classification tasks, data scientists often times balance interpretability and predictive accuracy. Some situations require a narrative to communicate the insights. In the social and natural sciences, classification methods like logistic regression are favored for their use in *parameter estimation* to infer

relationships between variables captured in coefficients. These coefficients are directly interpretable, showing how a variable can contribute or detract from the probability of an outcome holding all else constant. For example, a building built X decades ago is Y-times more likely to catch fire than a building built this decade holding all else constant. This simple factoid can facilitate narratives that communicate insight. But there is a tradeoff. While interpretable methods may extract the gist of the relationships, they may at times miss the finer variations in the data necessary for a reliable prediction.

Alternatively, some audiences are far less interested in the story, but just want an accurate number. In other words, predictive accuracy is prized in which case it may be worth considering a body of exciting methods that have arisen from statistics and computer science that are optimized for the task. These other methods may be more versatile, adapt to scenarios where there are more variables than observations, find interactions between variables and nonlinear patterns, and optimize for robustness. In the tech sector, for example, data science pursuits are often a matter of how well classifiers can scale to service their customers and drive sales. Understanding which variables definitively drive the predictions can be useful for teams focused on communicating insights, but not as much for the technology side of the house.

The bottom line is that one's choice of classifier depends on how much one values interpretability versus accuracy.

1.2.4 Fairness and transparency

In the area of criminal justice, algorithms such as COMPAS are actively being used to predict the chance that someone accused of a crime will recidivate within two years. While classifiers may offer a degree of efficiency when sifting through a multitude of data and are relied upon by judges in passing sentences, there are significant concerns regarding their ethical use. In 2016, ProPublica conducted research that examines the fairness of risk ratings from COMPAS, finding that while the algorithm is able to correctly predict recidivism 61% of the time, black defendents are 1.9-times more likely than white counterparts to be labeled high risk but not actually re-offend.[11] In other words, if two defendents with the same criminal history were scored by COMPAS and the only difference is their race, the two defendents would be treated differently.

It is unsurprising that *fairness* and *transparency* are emerging areas of focus in the use of classifiers. Fairness can take on many definitions, but for the purpose of this text, we define it as whether two or more subpopulations receive equal treatment. Transparency means that the provenance of the underlying information and its method of producing predictions are available for review – not that it fits a neat normative, but that it can be traced and scrutized.

1.2.5 Common techniques

There are hundreds of classification algorithms that are easily accessible using modern data science software, but six algorithms cover much of the use cases that are relevant for public policy. Each method has its strengths and weaknesses, and each is more appropriate in certain contexts than others.

Table 7: Overview of classifiers

Method	Common Uses	Example Data
k-Nearest Neighbors	Imputation of missing data; Relating data that can be treated like coordinates.	Hurricane response
Logistic Regression	Interpretation and narrative building; Hypothesis testing.	Health care coverage
LASSO logistic	Sift through large number of variables to surface likely covariates; Estimate logistic regression when there are more variables than observations.	Health care coverage

[11] https://www.propublica.org/article/machine-bias-risk-assessments-in-criminal-sentencing

Method	Common Uses	Example Data
Decision Tree	Produce predictions that can interact or cross variables together to see joint effects; Resulting nodes can be used as a "profiles".	Activity trackers
Random Forest	Produce predictions that can interact or cross variables together to see joint effects; Reasonable for cases when straight forward explanations are foregone.	Activity trackers
Neural Networks	Suitable for imagery and sound data to learn latent patterns locked within the data.	Faces of presidents

1.3 K-Nearest Neighbors (KNN)

As hurricanes become more intense and leave a trail of destruction, city governments will need to be able to more efficiently triage requests for help. Let's take the example of Hurricane Sandy and its effect on NYC. One of the main services offered by cities is the management and care of its trees. A downed tree can cause property damage, bodily harm and traffic disruptions. Due to the high wind and lush foliage during Sandy, many trees fell.

In NYC, the Department of Parks and Recreation is responsible for tree removal. When a resident makes a call to the city's services hotline 311, a work order is created and a tree removal team is dispatched. This may be a transactional process: one call for tree removal, one tree is then removed. As it takes time for crews to move and set up, a first-in/first-out queuing process can be inefficient. Imagine if 20 of 100 blocks in a neighborhood were flagged for tree removal. It would make sense to use that call data to identify other blocks that may also have downed trees.

We would expect that downed trees are more likely to occur in *pockets* and proximity is the best indicator of activity. As the city knows where residents call for tree- and non-tree-related issues, we can use the location of the calls to triangulate on likely problem areas as well as anticipate pockets of yet-to-be-reported downed trees, or at least serve that is a reasonable working theory.

For this task of predicting based on proximity, k-Nearest Neighbors (KNN) can help.

1.3.1 Under The Hood

K-nearest neighbors (KNN) is a non-parametric, instance-based algorithm that is based on a simple idea: *observations that are closer together are more likely to be similar*. The method is non-parametric as it does not directly use its inputs x to determine the value of y, but rather assumes that inputs serve as ways to quantify how close points are. It is instance-based as each prediction is determine on a case-by-case basis using its surrounding neighbors.

The technique is simple. For each case y_i:

1. *Distance.* First, we calculate the distance d to all other records with known outcomes. Distance most commonly takes the form of Euclidean distance, which is appropriate with continuous values. For cases where the underlying data are boolean or binary, Manhattan distance may be more appropriate. In effect, the input variables X serve as sets of coordinates to triagulate which points are closer to a given example. Note that treating variables like coordinate sets implies two things. First, all variables have equal importance – no single variable should weigh in on distance more than any other. Second, the scale of the observations would need to be the same.

$$d = \text{distance} = \sqrt{\sum_{i=1}^{n}(x_i - x_0)^2}$$

2. *Voting.* For the k nearest observations to a given observation, calculate the proportion of observations in each class j in Y. This procedure yields a conditional probability for each observation, which is converted into a predicted class through *majority voting* – assign an observation to the class that is most represented in the neighborhood. There are various flavors of the voting calculation that account for distance from a given observation.

Voting Type	Formula	Interpretation
Rectangular	$Pr(Y=j) = \frac{1}{k}\sum_{i=1}^{k} I(y^i = j)$	Calculate the proportion of j based on k nearest neighbors.
Inverse	$Pr(Y=j) = \sum_{i=1}^{k} w(d)(y^i = j)$ where $w(d) = \frac{1}{d_i \sum_{i=1}^{k}(\frac{1}{d_i})}$	Calculate the weighted proportion of j based on the inverse distance to k nearest neighbors.

3. *Tuning.* The method is sensitive to the value of k, requiring tuning – or testing different values of k. When $k = 10$, the conditional probability for y_i reflects the 10-nearest neighbors. When $k = n$, the conditional probability is the sample mean.

The above process yields the results for just one value of k. *Is that value of k the right one?* Like many other algorithms, KNNs are an iterative procedure, requiring tuning of *hyperparameters* – or values that are starting and guiding assumptions of a model. In the case of KNNs, k is a hyperparameter and we do not precisely know the best value of k. Tuning hyperparameters involves a grid search in which a range of possible hyperparameters at equal intervals are tested and compared to find the optimal.

To illustrate how tuning works, we have extracted a raster data from the US Department of Agriculture (USDA) CropScape data layer derived from satellite imagery.[12] In particular, we focus on a small area in Kansas that grows corn (yellow) and soybeans (green). More often than not, data will only contain a sample of the full picture. In the case of spatial data, KNNs are quite useful for filling in the full picture. Suppose we only have a 10% sample of the farmland – how important is k? We can see that as the value of k grows, the corn fields become overly represented as it is the majority class in the sample. The TPR is highest when the TNR is highest, demonstrating that there is a balancing act when choosing the value of k.

1.3.2 Tips of the trade

Tuning. As the accuracy of a KNN model is dependent on finding the optimal value of k, it makes the process far simpler to think about the tuning process as a grid search. A systematic way of testing for k is try test all values from $k = 1$ to $k = \sqrt{n}$ in multiples of one's choosing. The idea is to get a ballpark sense of what works, then hone in on the best value of k. Log the values of k that are tested and compare the accuracies to find the optimum. For example, searching for values of k that provide the best prediction in the cropland example suggests that TPR is near its max value at $k = 1$ and the FPR is minimized at the same value, thus choosing the minimum value of k would likely be adequate.

There are a number of other factors that influence the performance of the algorithm:

- *Scale.* A trained KNN is influenced by the scale of its inputs. If one input has a scale from 0 to 10,000 and another ranges from 0.1 to 0.3, a KNN will lean more heavily on the latter variable. To ensure equal weights, it is necessary to transform variables into a standardized scale:

$$scaled = \frac{x_i - \mu}{\sigma}$$

where the result transformed variable is mean centered and standardized. The new unit is standard deviations.

[12] https://nassgeodata.gmu.edu/CropScape/

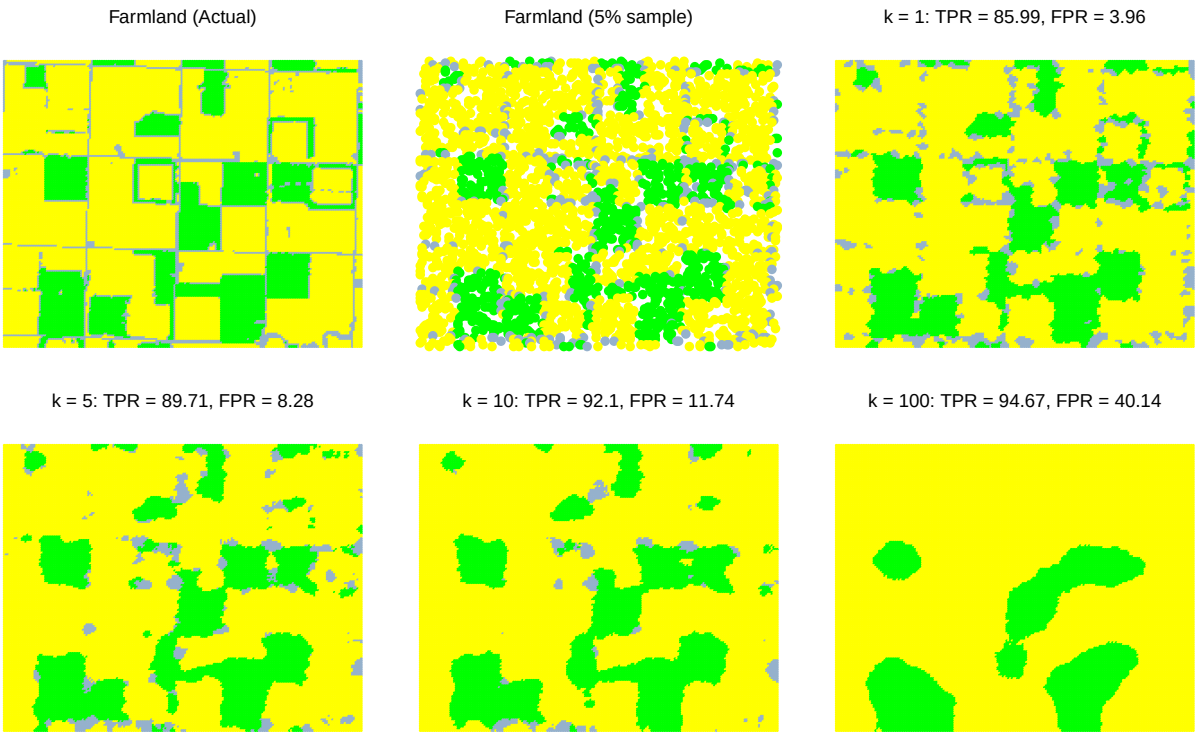


Figure 4: Comparison of prediction accuracies for various values of k.

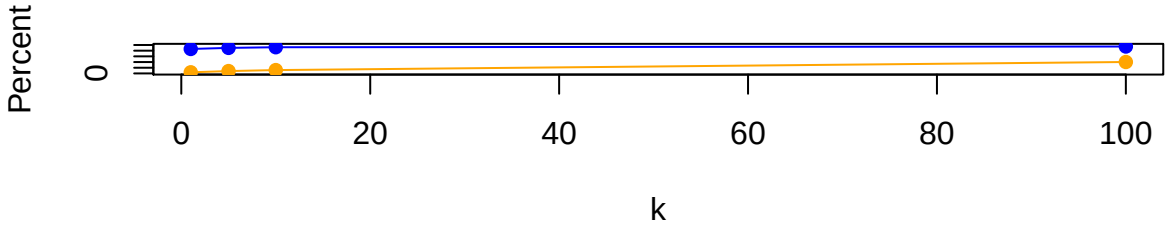


Figure 5: TPR (blue) and FPR (orange) for values of k.

- *Grids.* Similar to the scale issue, KNNs are particularly effective in data that are distributed on a grid – measurements along a continuous scale at equal incremenets, but may be a poor choice when the data are mixed data formats such as integers and binary.
- *Symmetry.* It's key to remember that neighbors around each point will not likely be uniformly distributed. While kNN does not have any probabilistic assumptions, the position and distance of neighboring points may have a skewing effect.

Usage. KNNs are great in some cases; Not so much in others.

KNNs are commonly associated with imputation of missing values and scenarios where proximity of observations has some bearing on the predictive accuracy. But setting up the KNNs requires some care.

As scale matters, data sets with mixed data types (discrete, continuous) need to be transformed into the same units. Discrete variables can be converted into a dummy variable matrix. Continuous variables can be binned into discrete levels, then converted into a dummy variable matrix as well. This effectively means that all variables are in terms of 0/1 and a Minkowski distance may be more appropriate to relate distances than Euclidean.

KNNs are best used when data sets are relatively smaller with fewer variables as each distance calculation is computationally taxing. Furthermore, as more variables are added, the importance of any one variable is diluted – it may be worth trying another algorithm to sift through the data.

Lastly, KNNs are not interpretable as it is a nonparametric approach. It should be instead be viewed as a processing method to fill in the gaps.

Table 9: The good and ugly of KNNs.

Useful Properties	Challenges
Efficient and timely when there are relatively few variables.	Mixed data types require convert all data into dummy matrices
Effective in capturing patterns in cases were proximity matters.	Does not offer an interpretation.
Common choice for imputing missing values.	

1.3.3 DIY: Anticipating the extent of damage from a storm

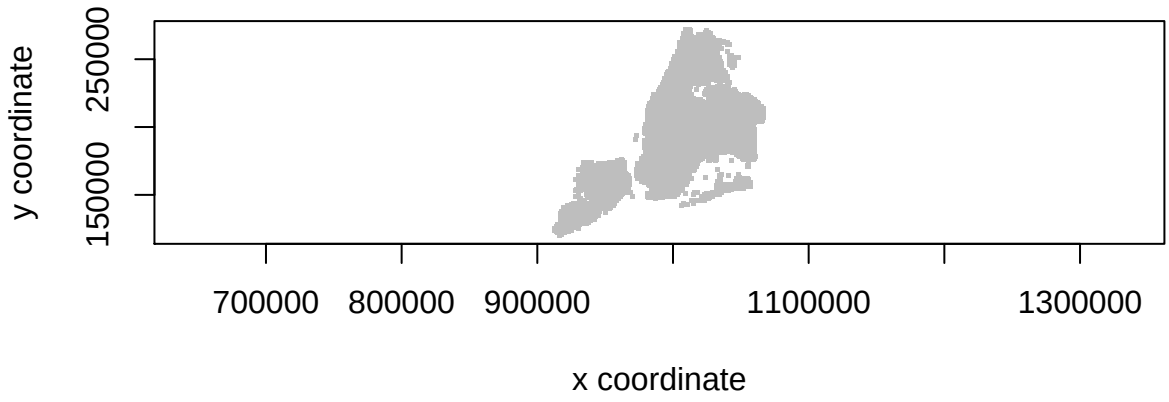
We now turn back to our motivating example. Suppose the location of all calls for non-emergency help from the day of Hurricane Sandy are captured in NYC's 311 system, yet there are still neighborhoods that likely have downed trees but have not reported it. *From what we know, how can we guess the disposition of other parts of the city to have a fuller picture?*

Prepare the data. Some calls for help are associated with downed trees while others may concern non-emergency issues. By this logic, we could assume that a downed tree would be called in if a call were made at all.

Our data set contains $n = 7513$ observations, each of which is a 1000 foot by 1000 foot area. When plotted, the data set captures the outline of NYC.

```
#Load data (need persistent link)
  nyc <- read.csv("data/sandy_trees.csv")

#Plot
  plot(x = nyc$xcoord, y = nyc$ycoord,
       xlab = "x coordinate", ylab = "y coordinate",
       cex = 0.3,  pch = 15, asp = 1, col = "grey")
```

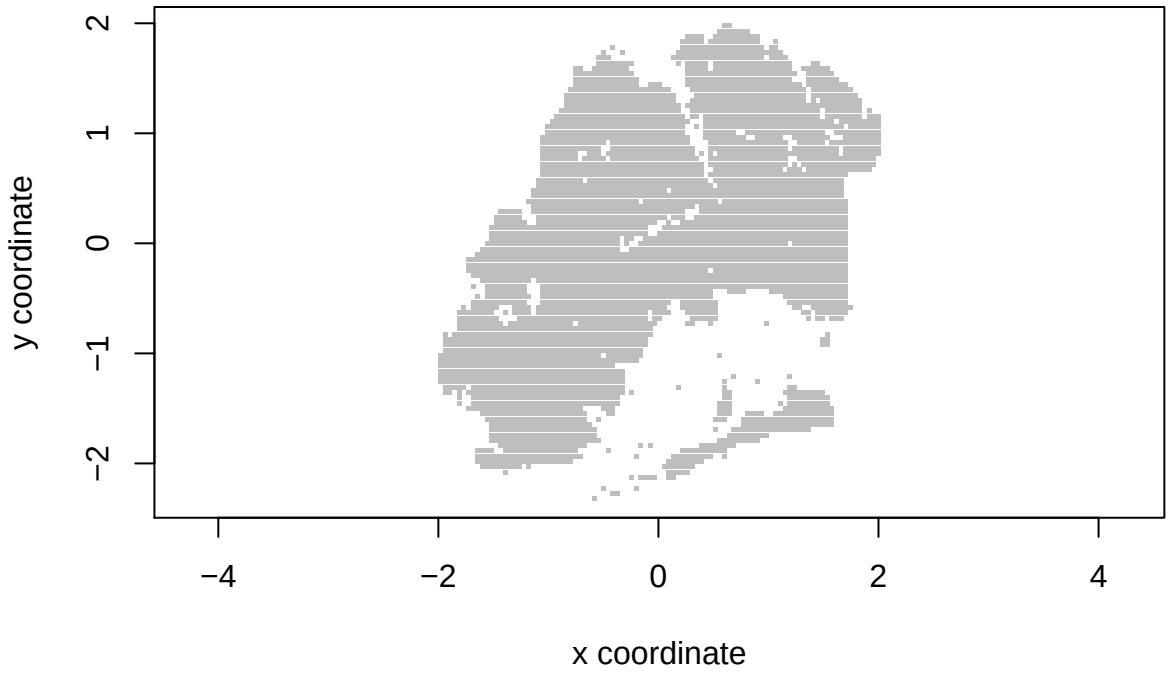


For simplicity, we will focus on the two largest and geographically connected boroughs, Brooklyn and Queens. The remaining boroughs of the city are separated by rivers, which may be better represented using separate models. We subset our data using the `boro` variable to focus, scale the geographic coordinates using the `scale` function, and map the resulting subset.

```
#Subset
  nyc <- subset(nyc, boro %in% c("QN", "BK"))

#Standardize input variables
  nyc$xcoord <- scale(nyc$xcoord)
  nyc$ycoord <- scale(nyc$ycoord)

#Plot
  plot(x = nyc$xcoord, y = nyc$ycoord,
       xlab = "x coordinate", ylab = "y coordinate",
       cex = 0.3, pch = 15, asp = 1, col = "grey")
```



Train. While this is a retrospective analysis, we simulate the process of producing the complete map as if we had partial information. The `nyc` data frame is split into a `train` set, keeping only locations where the target variable `tree.sandy` are available. A quick tabulation shows that the $n = 1550$ of the $n = 1946$ training set

observations have a downed tree reported.

```
#Subset training sample
  train <- subset(nyc, !is.na(tree.sandy),
                  select = c("ycoord", "xcoord", "tree.sandy"))

#Split out
  table(train$tree.sandy)
```

```
##
##    0    1
##  396 1550
```

The test set is the entirety of Brooklyn and Queens. The `tree.next7` variable flags any location that had a report of a downed tree over the seven days after the hurricane.

```
  test <- subset(nyc,
                 select = c("ycoord", "xcoord", "tree.next7"))
```

With the data in the right shape, we load the `kknn` library:

```
#Call "class" library
  library(kknn)
```

The KNN algorithm needs to be calibrated for the best k using the training set, then applied to a test set. To do this, we will use the `kknn` library. The training portion uses the `train.kknn()` function to conduct k-folds cross validation, then the scoring uses `kknn()`. While both functions can be fairly easily written from scratch (and we encourage new analysts to write their own to intimately understand the assumptions), we will plow forth with using the library.

In order to find the optimal value of k, we will execute the `train.kknn()` function, which accepts the following arguments:

`train.kknn(formula, data, kmax, kernel, distance, kcv)`

- `formula` is a formula object (e.g. "`coverage ~ .`").
- `data` is a matrix or data frame of training data.
- `kmax` is the maximum number of neighbors to be tested
- `kernel` is a string vector indicating the type of distance weighting (e.g. "rectangular" is unweighted, "biweight" places more weight towards closer observations, "gaussian" imposes a normal distribution on distance, "inv" is inverse distance).
- `distance` is a numerical value indicating the type of Minkowski distance. (e.g. 2 = euclidean, 1 = binary).
- `kcv` is the number of partitions to be used for cross validation.

The flexibility of `train.kknn()` allows for test exhaustively and find the best parameters. Below, we conduct 20-folds cross validation testing between $k = 1$ and $k = 100$ neighbors using two kernels (rectangular and inverse) that impact the voting step. This simple command does much of the hard work by running the KNN algorithm 2000 times (20 cross-validation models for each k and *kernel* combination), then surfaces the best parameters. We store the results in `fit.cv`.

```
#Set seed to ensure cross validation is replicable
  set.seed(100)

#Run with 20-folds cross validation
  fit.cv <- train.kknn(tree.sandy ~ ycoord + xcoord ,
                       data = train,
                       kcv = 20,
                       distance = 1, kmax = 100,
```

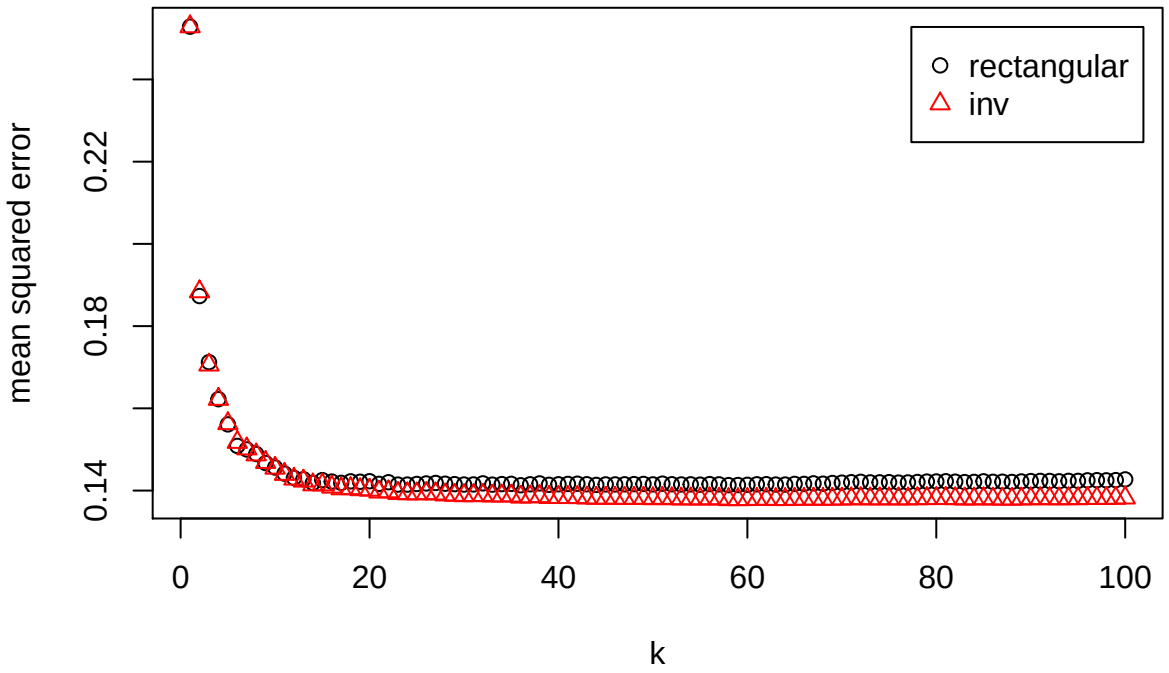


Figure 6: 20-fold cross validated errors for k = 1 to k = 100

```
                    kernel = c("rectangular", "inv"))
```

Within `fit.cv` is a `best.parameters` element that KNNs perform the best when $k = $ 59 using an inverse distance kernel.

```
#Plot Cross Validation
   plot(fit.cv)
```

```
#Retrieve best parameters
   best <- fit.cv$best.parameters
```

With the KNN algorithm tuned, we can now proceed to scoring the test set using the `kknn()` function. The function syntax is as follows:

```
kknn(formula, train, test, k, kernel, distance)
```

- `formula` is a formula object (e.g. "`coverage ~ .`").
- `train` is a matrix or data frame of training data.
- `test` is a matrix or data frame of test data.
- `k` is the number of neighbors.
- `kernel` is the type of weighting of distance (e.g. "rectangular" is unweighted, "biweight" places more weight towards closer observations).
- `distance` is a numerical value indicating the type of Minkowski distance. (e.g. 1 = binary, 2 = euclidean,).

Notice that in the following code block, we train the KNN and apply it to the test sample all in one step as the KNN itself does not learn patterns, but just applies a simple calculation following a pre-specified routine. This is a marked difference compared with other algorithms covered in this chapter.

```
#Apply tune KNN parameters
   fit <- kknn(tree.sandy ~ ycoord + xcoord,
               train = train,
```

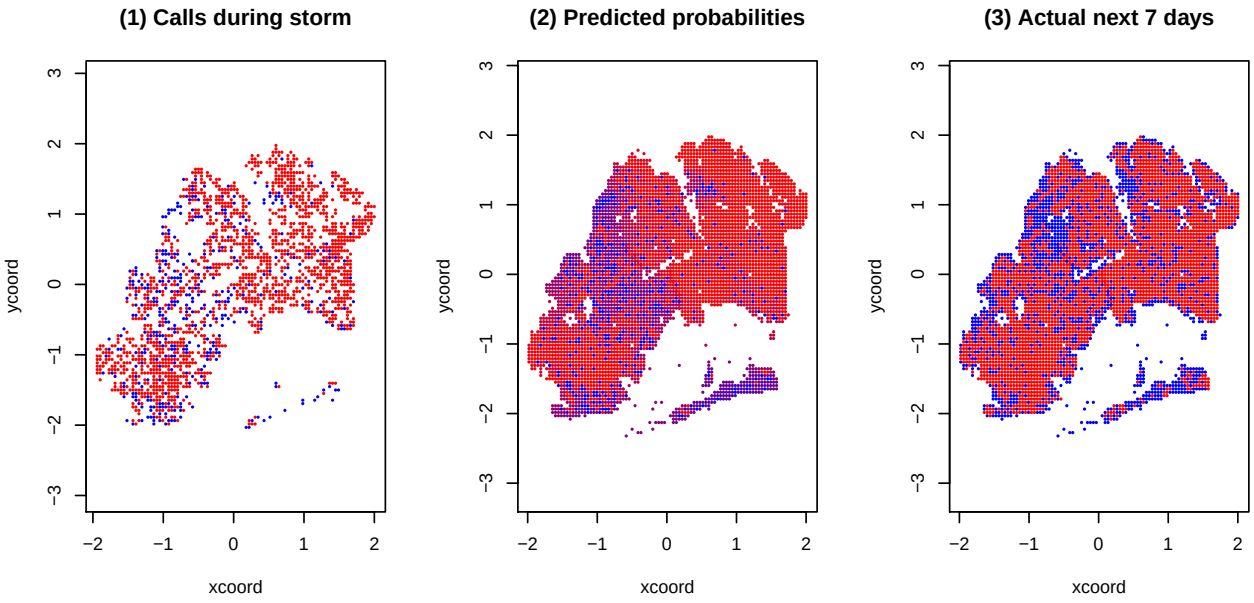


Figure 7: Graphical comparison of actual and predicted areas with reported downed trees. Red indicates at least one tree was reported in a given 0.359 square-mile area

```
                test = test,
                k = best$k,
                kernel = best$kernel)

#Produce
    test$prob <- fit$fitted.values
    test$tree.next7[is.na(test$tree.next7)] <-0
```

Evaluate. With all the right pieces computed, we can examine how closely the predictions based on tree downing patterns on the day of Hurricane Sandy compare with where trees were reported to have fallen over the 7 days that followed. During the storm, approximately 43% percent of the focus area made a call, of which 80% reported a downed tree. This appears as a cloud of points capturing the gist of the downed tree pattern.

```
par(mfrow = c(1,3))

 plot(train[,2:1], main = "(1) Calls during storm",
      col = rgb(train$tree.sandy , 0, 1- train$tree.sandy, 1),
      cex = 0.4, pch = 16, asp = 1)

 plot(test[,2:1], main =  "(2) Predicted probabilities",
      col = rgb(test$prob, 0, 1 - test$prob, 1),
      cex = 0.4, pch = 16, asp = 1)

 plot(test[,2:1], main =  "(3) Actual next 7 days",
      col = rgb(test$tree.next7, 0, 1 - test$tree.next7, 1),
      cex = 0.4, pch = 16, asp = 1)
```

Using the predicted probabilities for the test sample, we calculate the TPR and FPR using both a naive cutoff threshold ($p = 0.5$), finding a high TPR but a FPR of almost similar magnitude. Using such prediction

would not provide any insight to field crews.

```
tab <- table(test$prob >= 0.5, test$tree.next7)
tpr <- tab[2,2]/sum(test$tree.next7)
fpr <- tab[2,1]/sum(test$tree.next7 == 0)
print(paste0("TPR = ", tpr, ", FPR = ", fpr))
```

```
## [1] "TPR = 0.881461969048403, FPR = 0.777083333333333"
```

However, a more informed cutoff based on the first day's probability of a downed tree (p = 0.8) yields slightly more balanced results – sacrifiing some true positives for far fewer false positives.

```
tab <- table(test$prob >= 0.8, test$tree.next7)
tpr <- tab[2,2]/sum(test$tree.next7)
fpr <- tab[2,1]/sum(test$tree.next7 == 0)
print(paste0("TPR = ", tpr, ", FPR = ", fpr))
```

```
## [1] "TPR = 0.71188673032598, FPR = 0.390277777777778"
```

The test model accuracy can also be calculated by taking the Area Under the Curve (AUC) of the Receiving-Operating Characteristic. The ROC calculates the TPR and FPR at many thresholds, that produces a curve that indicates the general robustness of a model. The AUC is literally the area under that curve, which is a measure between 0.5 and 1 where the former indicates no predictive power and 1.0 indicates a perfect model.

In order to visualize the ROC, we will rely on the `plotROC` library, which is an extension of `ggplot2`. We will create a new data frame `input` that is comprised of the labels for the test set `ytest` and the predicted probabilities `test.prob`.

```
#Load libraries
  library(ggplot2)
  library(plotROC)

#Set up test data frame
  input <- data.frame(ytest = test$tree.next7,
                      prob = test$prob)
```

We then will first create a ggplot object named `base` that will contain the labels (`d =`) and probabilities (`m =`), then create the ROC plot using `geom_roc()` and `style_roc()`. A ROC curve for a well-performing model should sit well-above the the 45 degree diagonal line, which is the reference for an AUC of 0.5 (the minimum expected for a positive predictor). However, as the curve is below the 45 degree line, we may have a seriously deficient model.

```
#Base object
  roc <- ggplot(input, aes(d = ytest, m = prob)) +
         geom_roc() + style_roc()

#Show result
  roc
```

As estimated using `calc_auc()`, the out-of-sample AUC is 0.721, which is not a bad start. While we are able to fill impute the status of downed trees in other parts of the city, it is helpful to remember that the output of the KNN needs to match the intended use. If a limited number of field crews are deployed, then it may more sense to use the probabilities to prioritize neighborhoods. Otherwise, if additional resources could be hired, then knowing the total number of likely affected areas could inform how much to budget for the downed trees effort.

```
  calc_auc(roc)$AUC
```

```
## [1] 0.7205748
```

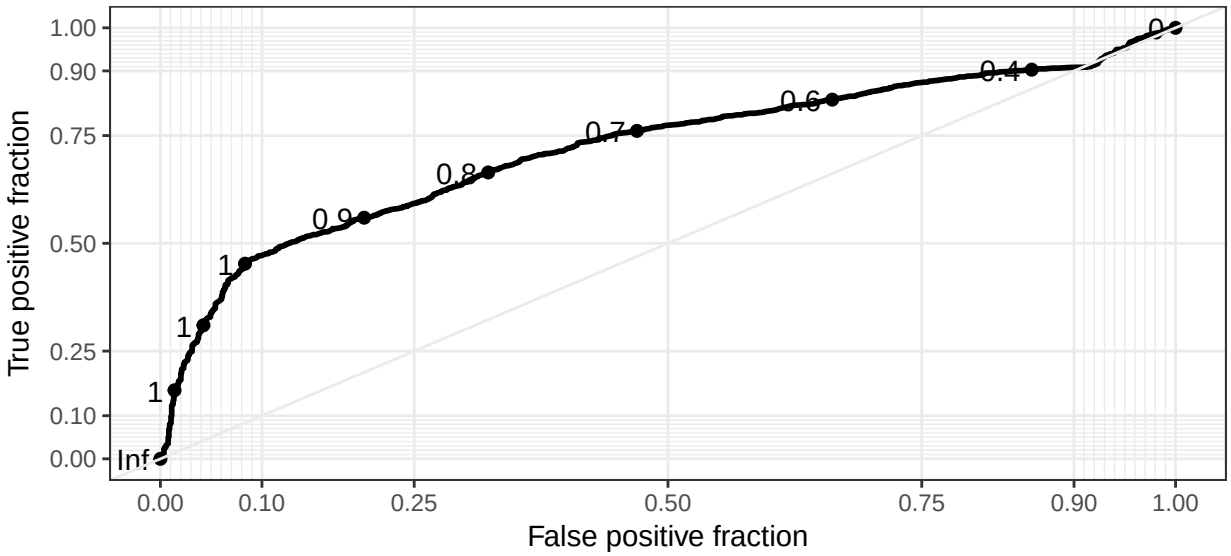


Figure 8: ROC curve out of sample

Despite the promising result, we should be cognizant that KNNs generally are not the algorithm of choice of modelers unless there is relatively little data. We should thus ask: *Is there a better classifier?*

1.3.4 Practice Exercises

The US Census Bureau's American Community Survey provides an in-depth view of life in America. One of the many features that are captured in the survey is healthcare coverage. Apply the above methods to predict healthcare coverage in the US State of Georgia in the year 2009.

The data can be obtained here [link to go here]:

1. Randomly split the sample into a 50% training and 50% test set.
2. Predict healthcare `coverage` using continuous variables such as age (`agep`) and `wage`.
3. Calculate the performance on the test sample.

1.4 Logistic Regression

NEED A NEW BEGINNING For much of the natural and social sciences, the goal of classification is inference. Inference of how much specific factors are associated with an observed phenomenon – not just prediction. The association typically are furnished with probabilistic qualities that allow an analyst to gauge how certain the pattern is. The output, in turn, lend themselves to building narratives.

The statistically-driven narrative are part of our daily lives. Nowadays, it would not surprise one to hear that a smoker has X-times higher chance of developing cancer than a non-smoker.[13] ^^MORE STATS EXAMPLES NEEDED HERE^^

These short empirical tid bits are rooted in a method known as *logistic regression*. Like ordinary least squares, logistic regressions are the workhorse of the social and natural sciences for inferring the marginal effects of input factors holding all else constant.

[13] https://www.cdc.gov/cancer/lung/basic_info/risk_factors.htm

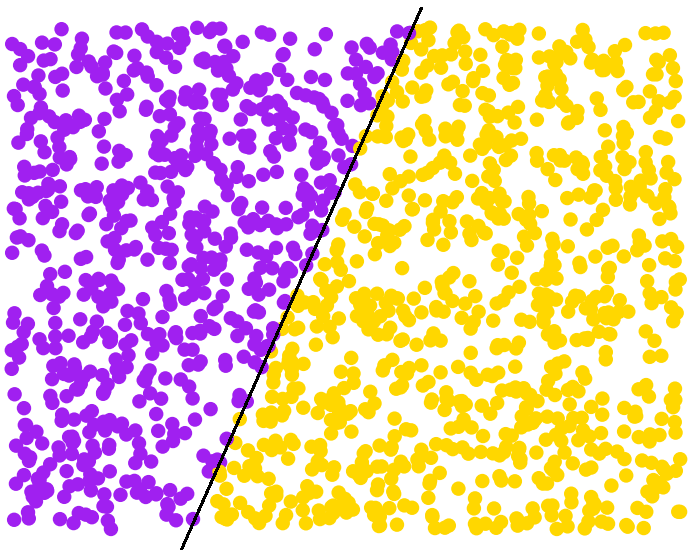


Figure 9: A linearly separable problem.

1.4.1 Under The Hood

Before we dive into the particulars, we will cut straight to the chase: logistic regression is is best suited for cases where we believe that the decision plane between two or more classes is a straight line. It imposes strong linear assumptions on a problem, meaning that the method may gloss over any pattern that is non-linear or discontinuous. That aside, logistic regression has the ability to contribute to statistically-driven narratives and has a long tradition of probabilistic inference backing it.

Let's assume that the classes of a target variable y can be dinstinguished using some linear combination of input variables $x1$ and $x2$. Upon graphing the features and color coding using the labels, you see that the points are clustered such that purple points represent to $z = 1$ and gold points represent $z = 0$.

We can express the relationship between z, x_1, and x_2 as a linear model similar to OLS:

$$z = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \epsilon$$

where z is a binary outcome and, like OLS, β_k are coefficients that are learned using a *Maximum Likelihood Estimation* or MLE. The simple idea of MLE is that coefficients can be adjusted so the coefficients can work together to maximize the liklihood of mimicking the target. While the innards of MLE are beyond the scope of this text, for more in-depth treatment on the topic, refer to *Elements of Statistical Learning* [14] or *Introduction to Statistical Learning.*[15]

The challenge with binary outcomes in a linear framework is that we run the risk that $\hat{y}$ would exceed the binary bounds of 0 and 1 and would thus make little sense. Imagine if the predicted $\hat{y}$ were -103 or +4 – *What would that mean in the case of a binary variable? How would we interpret a positive value versus a negative?* This is an obvious shortcoming of a linear model. Statisticians have cleverly solved the bounding problem by inserting the predicted output into a logistic function:

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

[14] Ref needed
[15] Ref needed

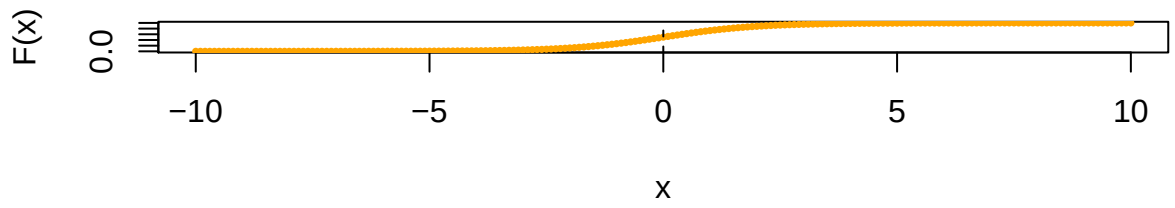


Figure 10: A sigmoid.

For a random variable x that takes on values from -10 to +10, the logit transformation converges to +1 where $x > 0$ and to 0 where $x < 0$. This S-shaped curve, or *sigmoid*, bounds $\hat{y}$ to a 0/1 range.

By substituting the linear model output z into the logistic function, we bound the output between 0 and 1 and interpret the result as a conditional probability:

$$p = Pr(Y = 1|X) = F(z) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x_1 + \beta_2 x_2)}}$$

This may seem to be a convoluted set of formulas, but it serves a convenient purpose. Unlike many of the techniques in this book, logistic regression is directly interpretable so as long as we believe that the decision boundary is linear. To be able to state that, for example, smokers have a 15 to 30-times higher chance of lung cancer than non-smokers, we use coefficients as a means to contextualize relative effects of input variables. This requires some basic math. The odds of an event are defined as the following:

$$odds = \frac{p}{1-p} = \frac{F(z)}{1 - F(z)} = e^z$$

In its purest form, probability p can be calculated without a model, but to hold all covariates constant, we can fit the output of a logistic regression into this framework where $F(z)$ is a probability of some event $z = 1$and $1 - F(z)$ is the probability of $z = 0$. The odds can be re-arranged as:

$$pr(success) = \frac{e^{(\beta_0 + \beta_1 x_1 + \beta_2 x_2)}}{1 + e^{(\beta_0 + \beta_1 x_1 + \beta_2 x_2)}}$$
$$pr(failure) = \frac{1}{1 + e^{(\beta_0 + \beta_1 x_1 + \beta_2 x_2)}}$$

Typically, we deal with *odds* in terms of *log odds* as the exponentiation may be challenging to work with:

$$log(odds) = log(\frac{p}{1-p}) = \beta_0 + \beta_1 x_1 + \beta_2 x_2$$

where *log* is a natural logarithm transformation. This relationship is particularly important as it allows for conversion of probabilities into odds and vice versa.

The underlying coefficients of the logistic regression can be interpretted using *Odds Ratios* or *OR*. Odds ratios essential express a marginal unit comparison. Since $odds = e^z = e^{\beta_0 + \beta_1 x_1 + \beta_2 x_2}$, then we can express an odds ratio as a marginal 1 unit increase in x_1 comparing $odds(x + 1)$ over $odd(x + 0)$:

$$OR = \frac{e^{\beta_0 + \beta_1 (x_1 + 1) + \beta_2 x_2}}{e^{\beta_0 + \beta_1 (x_1 + 0) + \beta_2 x_2}} = e^{\beta_1}$$

After some arithmetic, it turns out the OR is simply equal to e^{β_1}, which can be interpreted as a multiplicative effect or a percentage effect $(100 \times (1 - e^{\beta_1})\%)$. More simply, this means that one can obtain the ballpark

effect of a regression coefficient by exponentiating it. For example, if a logistic regression were trained to relate wages and citizenship to whether people have health care insurance, the result may look as follows:

$$y(\text{no coverage}) = 0.468 - 0.048 \times wage + 0.372 \times \text{non-citizen}$$

The coefficients provide little information other than the fact that a positive cofficient is associated with an increase in the chance of y. However, by exponentiating the cofficients, the odds of coverage are as follows for each variable:

- $OR_{wage} = e^{-0.048} = 0.953$ translates to -4.68% lower chance of not having health coverage for every $1000 increase in wages. Otherwise stated, more that one earns, better the chance of having health coverage.
- $OR_{non-citizen} = e^{0.372} = 0.451$ translates to 45% higher chance of not having health coverage among non-citizens.

1.4.2 Tips of the trade

Training a logistic regression is easy to do using a statistical analysis software, but is also easily used incorrectly. Tuning is a matter of finding the optimal combination of variables and generally starts from a hypothesis of how variables are related to a target. As tuning process unfolds, it invariably becomes a series of trial and error tests. While this forces data scientists to start from a theory and hypothesis, it also highlights some of the core challenges of modeling.

Suppose an analyst is tasked with understanding what drives health care insurance coverage in the United States and settles on a four-variable specification that lends itself to a compelling narrative. On the surface, the results sounds promising. However, it may seem less so when considering that the four variables were selected from among 100 possible variables – the results reflect only one of 3.9 million possible four variable specifications. *How does one know if the specification is the best?*

The possibilities of tuning an accurate model are be seemingly endless, but are far more manageable when put in the context of the policy objective. As logistic regression is one of the few truly interpretable statistical learning algorithms, the goals can be broadly divided between *interpretability* (focus on $\hat{\beta}$) and *prediction* (focus on $\hat{y}$).

Table 10: The good and ugly of logistic regression.

Useful Properties	Challenges
Interpretable in terms of odds ratios.	Collinear input variables may cause coefficients to return odd, inexplicable results.
Probabilistic properties allow for hypothesis testing that help build statistically-rooted arguments.	Unable to handle cases where k > n
Problems where the distinction between classes is linear	When k is large, finding optimal specification is challenging.
	Class imbalance makes prediction a non-trivial task.

Interpretability. As we have seen previously, the coefficients lend themselves to articulating how $X \rightarrow Y$ as the ultimate goal is causal inference. Causality is a challenging thing to prove, thus the process of producing reliable and trustworthy parameter estimates takes into account the basic assumptions of logistic regression. Failing to account for the basic assumptions of the technique will lead to misleading results. There are many assumptions, but we focus on a few key considerations:

In virtually all introductory texts that cover logistic regression, *multicollinearity* is flagged and rightfully so.

As a refresher, the condition is one in which two or more input variables are not only correlated with the target variables, but amongst each other as well. The consequence is that coefficients will behave oddly. The magnitude of the coefficients may seem unreasonably large. At times, the direction of the relationship is not what one may expect. The answer lies within what the coefficients represent: they are the average effect of x on y, partially isolated holding all else constant. Thus, if two or more variables have identical or very similar information, the algorithm is not able to precisely distill each variable's effects when coefficients are learned during maximum likelihood estimation process. This explains the odd behavior in coefficients and makes coefficients invalid for interpretation. Interestingly, the predictions $\hat{y}$ will still be usable. The best option is to conduct variable selection in advance to minimize double counting of signal. Note that while multicollinearity may interfere with the coefficients, the prediction $\hat{y}$ is still valid.

In information-rich environments, the *ill-posed problem* becomes more common: there may be more variables than observations. Like ordinary least squares, logistic regression cannot be solved when $k > n$. In the social sciences, subject matter intuition is the guiding force in choosing a parsimonious specification. More recently, regularization methods such as Least Absolute Shrinkage and Selection Operator (LASSO) and Ridge Regression have be relied upon to efficiently surface correlated variables. These methods are not without their problems, however. LASSO, for example, can force coefficient of noisy variables to zero, but coefficients do not have standard errors – thus for causal inference is a challenge.

For more in-depth treatment on the subject, consider reviewing [Need recommendations here].

Prediction. Unlike interpretation, prediction is purely focused on accuracy. Automated techniques are often employed to conduct variable selection, but there is one core modeling consideration that matters significantly for logistic regression: *class imbalance*. The idea of class imbalance is simple – when the classes in target y are not represented in relatively even proportions.[16] We can imagine that many problems are imbalanced:

- fire-related fatalities (only 3,400 fire deaths in 2017[17] vs. 2.7 million total deaths in the US in 2016[18])
- homelessness (approximately 0.17-percent of the US population[19])
- health care coverage (8.7% of US population is uninsured in 2017[20])

The smaller, less represented class is referred to as the *minority class*. While a logistic regression can be estimated, the size of the predicted values $\hat{y}$ will typically hover around average probability that an event will occur. Thus, if a model is trained to predict homelessness, a representative sample of the US will yield $\hat{y} = 0.17$ as the intercept in the linear model will have shrunken.[21] This, in turn, makes evaluating accuracy more challenging to evaluate as the classification threshold is no longer $c = 0.5$. There is not one solution to addressing the class imbalance problem, but rather many possible avenue. Here are a few common strategies:

- *AUC*: If the goal is to rank observations (e.g. prioritization of risky buildings), consider using the AUC derived from the ROC curve as it summarizes the overall accuracy across multiple thresholds.
- *Adjusted cutoff*: If the goal is to definitively flag observations for closer inspection, then it is worth considering to adjust the classification threshold c to a value that optimizes desired accuracy. For example, setting the threshold to the average probability can better balance the TPR and FPR. In these cases, it also becomes necessary to assign value to how much more or less a false positive is worth compared with one additional true positive prediction.
- *Over-sampling/Under-sampling*: If the goal is to produce a balanced prediction, there is a whole host of methods for adjusting the composition of the sample. The simplest is over-sampling in which the minority class observations are replicated until it is of equal proportion as the majority class. The inverse is also done: under-sampling the majority class so that it is of equal proportion to the minority. In either case, the threshold c can remain at 0.5, but there is a higher chance of overfitting.[22] Alternatively, Synthetic Minority Over-Sampling Technique (SMOTE) produces new observations that look like the

[16] ref needed
[17] https://www.usfa.fema.gov/data/statistics/
[18] https://www.cdc.gov/nchs/data/hus/2017/019.pdf
[19] https://www.bbc.com/news/world-us-canada-42248999
[20] https://www.census.gov/data/tables/time-series/demo/health-insurance/historical-series/hic.html
[21] Ref from Gary King from Harvard?
[22] ref required

minority class that add fresh signal to the sample while minimizing overfitting.[23]

1.4.3 DIY: Expanding Health Care Coverage

Background. Universal healthcare has become a basic human right in many countries. In the United States, this is not currently a guarantee, shrouded in heated political debate and controvery whether its a matter of human rights or a matter in which an individual may choose his or her fate. Regardless of the politics, there is data on healthcare coverage.

According to the American Community Survey ACS, an annual survey of approximately 3.5% of the US population as conducted by the US Census Bureau, over 22.4% of residents of the U.S. state of Georgia were without healthcare coverage in 2009. That is a fairly sizable proportion of the population – for every ten people, between two to three did not have coverage. If you read the news in 2010, a new law to provide affordable healthcare came into effect to help the uninsured.

Imagine that you are hypothetically tasked with getting the word out and drive recruitment in the state of Georgia. There is a hiccup, however. While commercial registries exist with people's demographic and personal contact information, most statistics on coverage are based on surveys, thus we do not precisely know *who* does not have insurance. A brute force approach could be to reach out to everyone under the sun though we can easily infer a wasted effort as 776 of every 1000 people are already covered. *How do we get to the 224 people who are not already insured?* For marketers, this is a classic targeting problem.

Data needs to enable the prediction and classification of a population into two classes: covered and not covered. By correctly classifying people as covered and not covered, decision makers and outreach staff can mobilize targeted outreach. From a data science perspective, the real objective is to be able to identify and replicate re-occurring patterns in the training data, then generalize the insights onto a sample or population that is not contained in the sample.

Given the label $y(Coverage)$, we can use logistic regression to not only infer what is associated with coverage, but also train a model to prioritize who should be contacted about receiving coverage:

$$y(Coverage) = f(\text{Sex, Age, Education, Marital Status, Race, Citizenship})$$

Data. Based on the Census American Community Survey (ACS), we will illustrate how to construct a logistic regression. First, we will load the ACS sample, which has been balanced – meaning that both covered and non-covered survey respondents are represented in equal proportions in the sample.

```
# Load ACS health care data
  load("data/acs_health.Rda")
```

Next, we will simulate a train and test to determine how well our models can do when applied in the wild.

```
# Randomly assign
  set.seed(100)
  rand <- runif(nrow(health)) > 0.5

# Create train test sets
  train <- health[rand == T, ]
  test <- health[rand == F, ]
```

Function. Training a logistic regression can be easily performed using the `glm()` function, which is a flexible algorithm class known as Generalized Linear Models. Using this one method, multiple types of linear models can be estimated including ordinary least squares for continuous outcomes, logistic regression for binary outcomes and Poisson regression for count outcomes.

At a minimum, three parameters are required:

[23] https://jair.org/index.php/jair/article/view/10302

`glm(formula, data, family)`

where:

- `formula` is a formula object. This can take on a number of forms such as a symbolic description (e.g. $y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \epsilon$ is represented as `y ~ x1 + x2`).
- `data` is a data frame containing the target and inputs.
- `family` indicates the probability distribution used in the model. Distributions typically used for GLMs are *binomial* (binary outcomes), *poisson* (count outcomes), *gaussian* (continuous outcomes - same as OLS), among others.

The `family` refers to the type of probability distributions that underlie each estimation method. In the case of logistic regression, the probability family is *binomial*.

Focusing on interpretation. In the social sciences and in public policy, the focus of regression modeling is typically placed on identifying an effect or an associated relationship that describes the process being studied. To tease out the effects, logistic regression analyses tend to center around what is colloqially known as a *build up*. Blocks of conceptually-related variables are tested in order to infer their important and effect on the target outcome. As we are focused on insurance coverage, we specify four models:

- *Personal*: $coverage = f(log(age) + race + sex)$
- *Economic*: $coverage = f(wage + employment)$
- *Social*: $coverage = f(citizenship + marital + schooling)$
- *Combined*: $coverage = f(log(age) + race + sex + wage + employment + citizenship + marital + schooling)$

It is worth highlighting that the inclusion of certain characteristics, such as race and sex, can improve the accuracy of a model. However, when used for prediction – particularly for targeting – one needs to consider the ethics and fariness of including these variables in application.

```
# Estimated GLM models
  glm_pers <- glm("coverage ~ log(age) + race + sex",
                      data = train, family = binomial)

  glm_econ <- glm("coverage ~ wage + esr",
                   data = train, family = binomial)

  glm_soc <- glm("coverage ~ cit + mar + schl",
                  data = train, family = binomial)

  glm_all <- glm("coverage ~ log(age) + wage + schl + esr + cit + mar + race + sex",
                  data = train, family = binomial)
```

The regression outputs chronicles every pertinent aspect of each model, such as the direction of the relationships (e.g. positive or negative weights), their statistical significance (e.g. p-value or t-statistics), and the relative fit of the model (e.g. the lowest Akaike Information Criterion or AIC provides *relative* model fit comparison). For example, an analyst may point out that education has an effect on coverage by interpreting the coefficient point estimates. In the combined model, education attainment coefficients are are estimated relative to people who hold a graduate degree, thus indicating that people who :

- did not finish high school have a *6.58-times* higher chance of not having health coverage ($ e^{w = 1.884} = 6.58$)
- hold a high school degree have a *4.91-times* higher chance of not having health coverage ($ e^{w = 1.592} = 4.91$)
- hold a college degree are relatively better off than the previous two groups with a *1.79-times* higher chance of not having health coverage ($ e^{w = 0.584} = 1.79$)

All coefficients are statistically significant. While it is valid to evaluate models on this basis, it is necessary to remember that this is not the same as evaluating a model for predictive use cases as predictive accuracy is not assessed on the basis of coefficients.

% Table created by stargazer v.5.2 by Marek Hlavac, Harvard University. E-mail: hlavac at fas.harvard.edu
% Date and time: Wed, Oct 24, 2018 - 23:23:21

Focus on prediction. Prediction, in contrast, emphasizes absolute generalizable accuracy and requires model validation techniques like k-cross validation. We can rely on the `boot` library to generate cross-validated accuracy estimates through the `cv.glm()` function:

```
cv.glm(data, glmfit, cost, K)
```

where:

- `data` is a data frame or matrix.
- `fit` is a glm model object.
- `cost` specifies the cost function for cross validation.
- `K` is the number of cross validation partitions.

Note that the cost function needs to take two vectors. The first is the observed responses and the second is the predicted responses. For example, the cost function could be the overall accuracy rate:

$$\frac{FP + FN}{TP + FP + TN + FN}$$

as translated into R code below:

```
# Classification Rate
  costAccuracy <- function(y, yhat){
    a <- sum((y == 1 ) & (yhat >= mean(y)))
    b <- sum((y == 0 ) & (yhat < mean(y)))
    c <- ((a + b) / length(y))
    return(c)
  }
```

So, that we can compare the cross validation accuracy with KNN, we will specify the `cost` using the misclassification rate for each of the three candidate models and set $k = 10$. Whereas KNN was able to achieve a 74% accuracy rate, the best GLM model was able to reach 72%, suggesting that some of the underlying variability in coverage rate is not captured in linear relationships. Also note that the input features for the KNN model were in a dummy matrix, thus the comparison is not perfect.

Table 12: Comparison of CV accuracy (k = 10 folds)

specification	accuracy
Personal	0.6646384
Economic	0.4857574
Social	0.6680039
All	0.7204243

Scoring. In order to obtain the score the test set, we use `predict()`:

```
predict(object, newdata, response)
```

where:

- `object` is a GLM model object.
- `newdata` is a data frame. This can be the training data set or the test set with the same format and features as the training set.
- `response` indicates the type of value to be returned, whether it is the untransformed "link" or the probability "response".

Table 11:

	Coefficient table of four alternative logistic regression specifications.			
			NA	
	Personal	Economic	Social	All
	(1)	(2)	(3)	(4)
log(age)	−1.223***			−0.697***
	(0.032)			(0.039)
raceAmer. Ind.	10.186			10.446
	(80.842)			(82.538)
raceAsian	9.791			9.495
	(80.842)			(82.537)
raceBlack	9.918			10.209
	(80.842)			(82.537)
raceNat. Hawaiian/Pac. Isl.	9.212			9.323
	(80.844)			(82.540)
raceOther	11.531			10.846
	(80.842)			(82.537)
raceTribes Spec.	10.440			10.350
	(80.843)			(82.538)
raceTwo or More	9.582			9.874
	(80.842)			(82.538)
raceWhite	9.400			9.867
	(80.842)			(82.537)
sex	−0.320***			−0.277***
	(0.028)			(0.029)
wage		−0.001***		−0.001***
		(0.0001)		(0.0001)
esr		0.021***		−0.011*
		(0.006)		(0.007)
cit			0.387***	0.346***
			(0.011)	(0.013)
mar			0.326***	0.203***
			(0.008)	(0.010)
schl			−0.100***	−0.106***
			(0.004)	(0.004)
Constant	−6.158	−1.527***	−1.248***	−7.691
	(80.842)	(0.022)	(0.070)	(82.538)
Observations	37,142	37,142	37,142	37,142
Log Likelihood	−16,304.320	−17,632.960	−15,759.410	−15,209.060
Akaike Inf. Crit.	32,630.640	35,271.920	31,526.820	30,450.120

Note: *p<0.1; **p<0.05; ***p<0.01

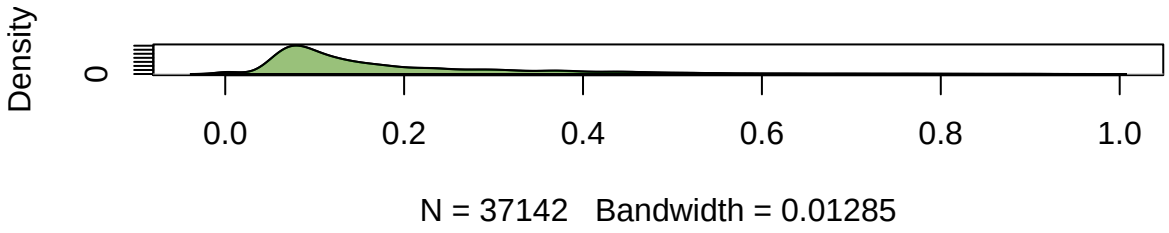


Figure 11: Predicted probabilities

We will now apply `predict()` to score the responses for each `train` and `test` samples.

```
  pred.glm.train <- predict(glm_all, train, type = "response")
  pred.glm.test <- predict(glm_all, test, type = "response")
```

A quick review of the predicted probabilities indicates confirms that we have the right response values (probabilities), bound by 0 and 1.

Lastly, to calculate the prediction accuracy, we will once again rely on the combination of `ggplot2` and 'plotROC libraries for the AUC. Interestingly, the test set AUC is greater than that of the train set. This occurs occassionally and is often times due to the luck of the draw.

```
#plotROC
  library(plotROC)
  library(ggplot2)

#Set up ROC inputs
  input.glm <- rbind(data.frame(model = "train", d = train$coverage, m = pred.glm.train),
                     data.frame(model = "test", d = test$coverage,  m = pred.glm.test))

#Graph all three ROCs
  roc.glm <- ggplot(input.glm, aes(d = d, model = model, m = m, colour = model)) +
              geom_roc(show.legend = TRUE) + style_roc()  + ggtitle("ROC: GLM")

#AUC
  calc_auc(roc.glm)[,2:3]
```

```
##   group       AUC
## 1     1 0.7697812
## 2     2 0.7722329
```

1.4.4 Practice Exercises

1. Can logistic regression be applied to the downed tree problem from the KNN section? Apply the method to the downed trees data. How do the accuracies compare and why?

1.5 LASSO regression

1.6 Decision Tree Learning

```
# Load ACS health care data
  library(digIt)
  health <- digIt("acs_health")
```

```r
# Convert characters into discrete factors
  factor_vars <- c("coverage", "mar", "cit", "esr", "schl")
  for(var in factor_vars){
    health[,var] <- as.factor(health[,var])
  }

# Randomly assign
  set.seed(100)
  rand <- runif(nrow(health)) > 0.5

# Create train test sets
  train <- health[rand == T, ]
  test <- health[rand == F, ]
```

In everyday policy setting and operations, decision trees are a common tool used for communicating complex processes, whether for how an actor moves through intricate and convoluted bureaucracy or how a sub-population can be described based on a set of criteria. While the garden variety decision tree can be laid out qualitatively, supervised learning allows decision trees to be created in an empirical fashion that not only have the power to aesthetically communicate patterns, but also predict how a non-linear system behaves.

The structure of a decision tree can be likened to branches of a tree: moving from the base of the tree upwards, the tree trunk splits into two or more large branches, which then in turn split into even smaller branches, eventually reaching even small twigs with leaves. Given a labeled set of data that contains input features, the branches of a decision tree is grown by subsetting a population into smaller, more homogeneous units. In other words, moving from the root of the tree to the terminating branches, each subsequent set of branches should contain records that are more similar, more homogeneous or purer.

As was demonstrated at the beginning of this chapter, decision trees use a form of recursive partitioning to learn patterns, doing so using central concepts of *information theory*. There are a number of decision tree algorithms that were invented largely in the 1980s and 1990s, including the ID3 algorithm, C4.5 algorithm, and Classification And Regression Trees for Machine Learning (CART). All these algorithms follow the same framework that includes the following elements: (1) nodes and edges, (2) attribute tests, and (3) termination criteria.

1.6.0.1 Under The Hood

Nodes + Edges. Recalling the healthcare insurance decision tree, the tree can be characterized by nodes and edges.

- Nodes (circles) contain records.
- Edges (lines) show dependency between nodes and is the product of a split decision. Nodes are split based on an attribute test – a technique to identify the optimal criterion to subset records into more homogeneous groups of the target variable.
- The node at the top of the tree is known as the *root*and represents the full population.
- Each time a node is split, the result is two nodes – each of which is referred to as a child node. A node without any child nodes is known as a leaf.

The goal is to grow a tree from the root node into as many smaller, more homogeneous child nodes with respect to the target variable.

Attribute tests. To understand attribute tests means to have a thorough understanding of separability. Let's suppose we have a list of residents of a town. The list contains both users and non-users of a given healthcare service. For each person, the inventory captures whether a given person is employed, has income over $20k, and lives on the west side or east side of town. Each of the features are plotted in the pie chart below. 50% of town residents use the health service, but which of the features is best at separating users from non-users?

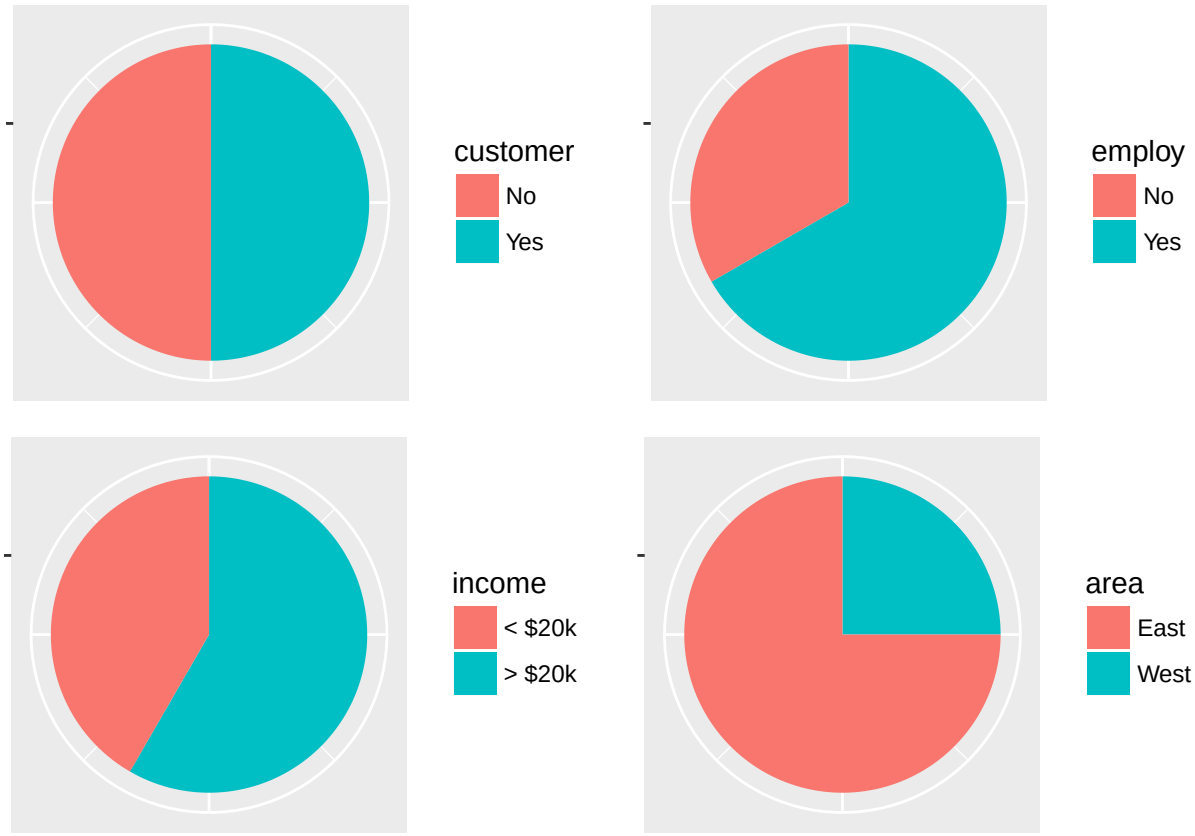


Figure 12: Summary characteristics of town residents.

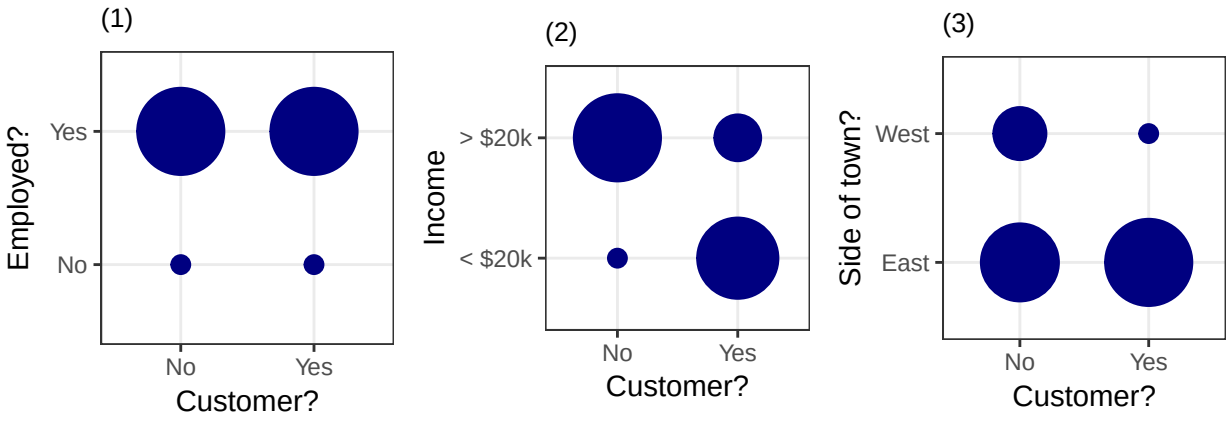


Figure 13: A visual comparison of low separability (1 and 3) and high separability (2).

To answer that question, we can rely on a visual cross-tabulation where the size of the circles is scaled proportional to the number of records. The objective is to identify the matrix where the circles are the largest along any diagonal – this would indicate that given usership, a feature is able to serve as a criterion that separates users from non-users. Of the three graphs below, graph #2 is able to separate a relatively large proportion of users from non-users. For a relatively low-dimensional dataset (fewer attributes), a visual analysis is accomplishable. However, on scale, undertaking this process manually may be onerous and prone to error.

Enter attribute tests.

Decision trees are grown by splitting a data set into many smaller samples. Attribute tests are the mode of finding the split criterion, following an empirical process to systematically test all input features to find the feature with the greatest separability. The process starts from the root node where the algorithm examines each input feature to find the one that maximizes separability at that node:

```
Let Sample = S, Target = Y, Input Features = X
    For each X:
        Calculate the attribute test statistic comparing X and Y
        Store statistic
    Compare and identify Xi that yields the greatest separability
    Split S using input feature that maximizes separability
    Iterate process on child node
```

Upon finding the optimal feature for a given node, the decision tree algorithm splits the node into two child nodes based on the optimal feature, then moves onto the next node (often times a child node) and runs the same process to find the next split. There are a number of attribute tests, of which we will cover two: *Information Gain* and *Gini Impurity*.

Information gain is a form of *Entropy*, which is a measure of purity of information. Based on these distinct states of activity, entropy is defined as:

$$\text{Entropy} = \sum -p_i log_2(p_i)$$

where i is an index of states, p is the proportion of observations that are in state i, and $log_2(p_i)$ is the Base 2 logarithm of the proportion for state i. Information Gain (IG) is variant of entropy, which is the entropy of the root node *less* the average entropies of the child nodes.

$$\text{IG} = \text{Entropy}_{\text{root}} - \text{Avg Child Entropy}$$

How does this work in practice? Starting from the root node, we need to calculate the root entropy, where the classes are based on the classes of the target `usership`.

$$\begin{aligned}\text{Entropy}_{\text{usership}} &= (-p_{user} log_2(p_{\text{user}})) - (-p_{\text{non-user}} log_2(p_{\text{non-user}})) \\ &= (-\tfrac{6}{12} log_2(\tfrac{6}{12})) + (-\tfrac{6}{12} log_2(\tfrac{6}{12})) \\ &= 1.0\end{aligned}$$

Then, the attribute test is applied to the root node by calculating the weighted entropy for each proposed child node. Using the `income` feature, the calculation is as follows:

- Split the root node into two child nodes using the `income` class. This yields the following subsamples as shown in the table below:

	$<$ \$20k	$>$ \$20k
No	0	6
Yes	5	1
Total	5	7

- For each child node (the columns in the table), calculate entropy:

$$\begin{aligned}\text{Entropy}_{\text{income} < 20\text{k}} &= (-p_{user} log_2(p_{\text{user}})) - (-p_{\text{non-user}} log_2(p_{\text{non-user}})) \\ &= -\tfrac{5}{5} log_2(\tfrac{5}{5}) = 0 \\ \text{Entropy}_{\text{income} > 20\text{k}} &= (-p_{user} log_2(p_{\text{user}})) - (-p_{\text{non-user}} log_2(p_{\text{non-user}})) \\ &= -\tfrac{6}{7} log_2(\tfrac{6}{7}) + -\tfrac{1}{7} log_2(\tfrac{1}{7}) = 0.5916728\end{aligned}$$

- Calculate the weighted average entropy of children:

$$\text{Entropy}_{\text{income split}} = \tfrac{5}{12}(0) + \tfrac{7}{12}(0.5916728) = 0.3451425$$

- Then calculate the information gain:

$$\begin{aligned}\text{IG}_{\text{income}} &= \text{Entropy}_{\text{root}} - \text{Entropy}_{\text{income split}} \\ &= 1 - 0.3451425 = 0.6548575\end{aligned}$$

- We then can perform the same calculation on all other features (e.g. employment, part of town) and compare results. The goal is to *maximize* the IG statistic at each decision point. In this case, we see that income is the best attribute to use for splitting. This split is easily interpretable: "The majority of users of health services can be predicted to earn less than \$20,000."

Measure	IG
Employment	0.00
Income	0.6548575
Area of Town	0.027119

Gini Impurity is closely related to the entropy with a slight modification:

$$\text{Gini Impurity} = \sum p_i(1 - p_i) = 1 - \sum p_i^2$$

Using Gini Impurity as an attribute test is also similar to Information Gain:

$$\text{Gini Gain} = \text{Gini}_{\text{root}} - \text{Weighted Gini}_{\text{child}}$$

1.6.0.2 (3) Stopping Criteria + Tree Pruning

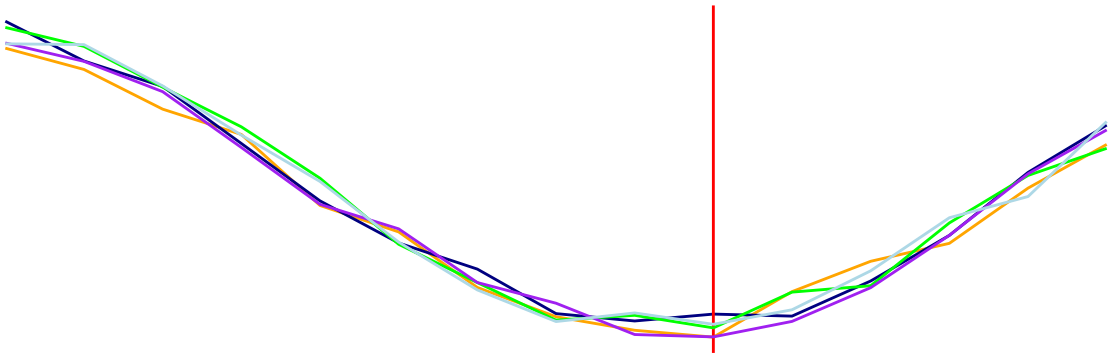
Both Gini Gain and Information Gain attribute tests can be recursively applied until there are no longer input features available to split the data. This is also known as a "fully grown tree" or an "unpruned tree". While the terminal leafs may yield a high degree of accuracy in training, trees may grow to epic and complex proportions that have leaf sizes are often times too small to provide accurate and generalizable results. While fully grown trees are considered to have low bias, their out-of-sample performance may be high in variance. There [theoretically] exists some optimal balancing point where trees are complex enough to capture statistical patterns, but are not too complex to yield misleading results.

Fortunately, the methodologists who invented decision tree learning have designed two approaches to balance accuracy and generalizability: stopping criteria and pruning.

Recall that a leaf is defined as a node with no child nodes. Otherwise stated, a leaf is a terminal node in which no additional attribute testing is conducted – it's placed out of commission. Stopping criteria are employed to determine if a node should be labeled a leaf during the growing process, thereby stopping tree growth at a given node. These criteria are specified before growing the tree and take on a number of different forms including:

- A node has fewer records than a pre-specific threshold;
- The purity or information gain falls below a pre-specified level or is equal to zero;
- The tree is grown to n-number of levels (e.g. Number of levels of child nodes relative to the root exceeds a certain threshold).

While stopping criteria are useful, the results in some studies indicate their performance may be sub-optimal. The alternative approach involves growing a tree to its fullest, then comparing the prediction performance given tree complexity (e.g. number of nodes in the tree) using cross-validation. In the example graph below, model accuracy degrades beyond a certain number of nodes. Thus, optimal number of nodes is defined as when cross-validation samples (e.g. train/test, k-folds) reaches a minimum across samples. Upon finding the optimal number of nodes, the tree is *pruned* to only that number of nodes.



1.6.0.3 Issues

Like any technique, decision trees have strengths and weaknesses:

Strengths	Weakness
- Rules (e.g. all the criteria that form the path from root to leaf) can be directly interpreted. - Method is well-suited to capture interactions and non-linearities in data. - Technique can accept both continuous and continuous variables without prior transformation. - Feature selection is conducted automatically	- Data sets with large number of features will have overly complex trees that, if left unpruned, may be too voluminous to interpret. - Trees tend to overfitted at the terminal leafs when samples are too small.

1.6.0.4 In Practice: Decision Trees

To put decision trees into practice, we will use the same `train` and `test` data frames introduced in the GLM section. There are a number of R implementations of decision trees, the most popular of which is the `rpart` library:

```
library(rpart)
```

The main function within the library comes with flexible capabilities to grow decision trees:

```
rpart(formula, method, data, cp, minbucket, minsplit)
```

where:

- `formula` is a formula object. This can take on a number of forms such as a symbolic description (e.g. $y = f(x_1, x_2, ...)$ is represented as "`y ~ x1 + x2`"").
- `method` indicates the type of tree, which are commonly either a classification tree "class" or regression tree "anova". Split criteria can also be custom written.
- `data` is the data set in data frame format.
- `cp` is a numeric indicates the complexity of the tree. $cp = 1$ is a tree without branches, whereas $cp = 0$ is the fully grown, unpruned tree. If cp is not specified, `rpart()` defaults to a value of 0.01.
- `minbucket` is a stopping criteria that specifies the minimum number of observations in any terminal leaf.
- `minsplit` is a stopping criteria that specifies the number of observation in a node to qualify for an attribute test.

As a first pass, we'll run `rpart()` with the default assumptions. Note that in `rpart()` automatically conducts k-folds cross-validation for each level of tree growth. If one were to use `summary()` or `str()` to check the structure of the output object named `fit`, the inner workings would likely be found to be quite exhaustive and rather complex. Fortunately, the `printcp()` method can be used to obtain a summary of the overall model accuracy for tree at different stages of growth. Key features of the `printcp()` output include:

- A listing of the variables actually used in construction (note that `cit`)
- In the table, `CP` indicates the tree complexity, `nsplit` is the number of splits, `rel error` is the prediction error in the training data, `xerror` is the cross-validation error, and `xstd` is the standard error.

To choose the best tree, a *rule of thumb* is to first find the tree with the lowest cross-validation `xerror`, then find the tree that has the lowest number of splits that is still within one standard deviation `xstd` of the best tree[24]. The idea behinds this rule of thumb takes advantage of uncertainty: the true value lies somewhere within a confidence interval, thus any value within a tight confidence interval of the best value is approximately the same. In this first model, the best tree has nsplit = 7 and xerror = 0.542760247714538. By applying the rule, the upper bound of acceptable error is xerror = 0.54276 + 0.00764 = 0.550399764766261. As it turns out, the tree with nsplit = 6 is within one standard deviation and is thus the best model.

```
#Fit decision tree under default assumptions
  fit <- rpart(coverage ~ agep + wage + cit + mar + schl + esr,
               method = "class", data = train)

#Tools to review outpu
  printcp(fit)
```

The model's learned rules contained in `fit` can be plotted with `plot()`, but it takes a bit of work to get the plot into a presentable format. The substitute is using the `rpart.plot` library, which auto-formats the tree and color codes nodes based on the concentration of the target variable.

```
#Plot
  library(rpart.plot)
  rpart.plot(fit, shadow.col="gray", nn=TRUE)
```

[24] Hastie et. al (2001)

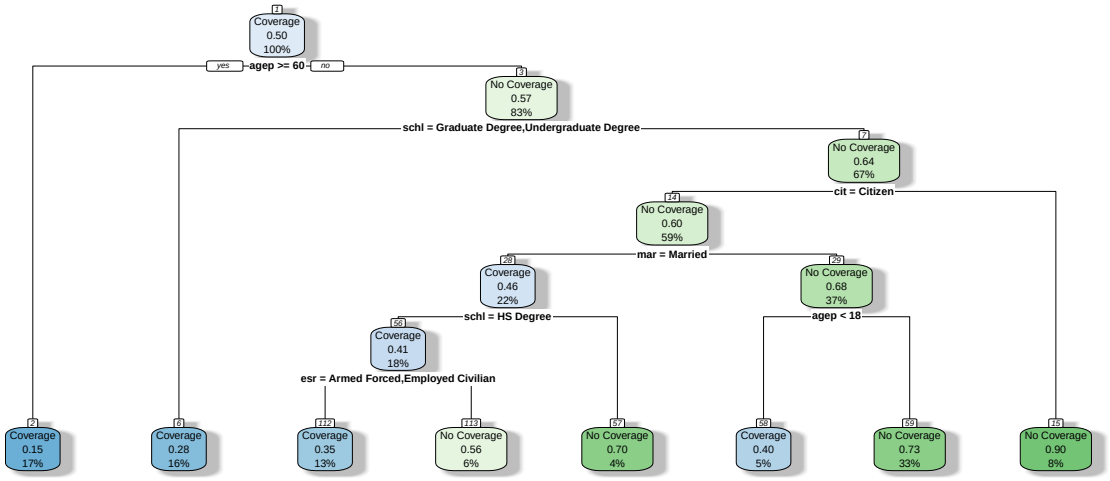


Figure 14: Decision tree using default parameters.

While this answer is valid, it should be noted that the CP lower threshold is 0.01, which is the default value. For robustness, we should run the model once more, this time specifying $cp = 0$ to obtain the full, unpruned tree (see below). Applying the error minimization rule once more, the minimum xerror = 0.495429, which corresponds to nsplit = 40. The maximum *xerror* within one standard deviation is xerror = 0.495429 + 0.007416 = 0.502845, which corresponds to nsplit = 21 with xerror = 0.502802 and cp = 0.000737

```
#cp = 0
  fit.0 <- rpart(coverage ~ agep + wage + cit + mar + schl + esr ,
                 method = "class", data = train, cp = 0)
  printcp(fit.0)
```

```
##
## Classification tree:
## rpart(formula = coverage ~ agep + wage + cit + mar + schl + esr,
##     data = train, method = "class", cp = 0)
##
## Variables actually used in tree construction:
## [1] agep cit  esr  mar  schl wage
##
## Root node error: 6782/13596 = 0.49882
##
## n= 13596
##
##            CP nsplit rel error  xerror      xstd
## 1  2.3061e-01      0   1.00000 1.00664 0.0085963
## 2  1.4258e-01      1   0.76939 0.81598 0.0084465
## 3  1.6883e-02      2   0.62681 0.63934 0.0080128
## 4  1.2976e-02      6   0.54364 0.54527 0.0076506
## 5  9.5842e-03      7   0.53067 0.53657 0.0076119
## 6  5.9717e-03      8   0.52109 0.52389 0.0075538
## 7  4.8658e-03     10   0.50914 0.51533 0.0075135
## 8  2.9490e-03     11   0.50428 0.50944 0.0074852
## 9  1.9168e-03     12   0.50133 0.50723 0.0074744
## 10 1.8185e-03     13   0.49941 0.50560 0.0074665
## 11 1.1059e-03     16   0.49395 0.50413 0.0074593
## 12 9.3384e-04     18   0.49174 0.50472 0.0074622
```

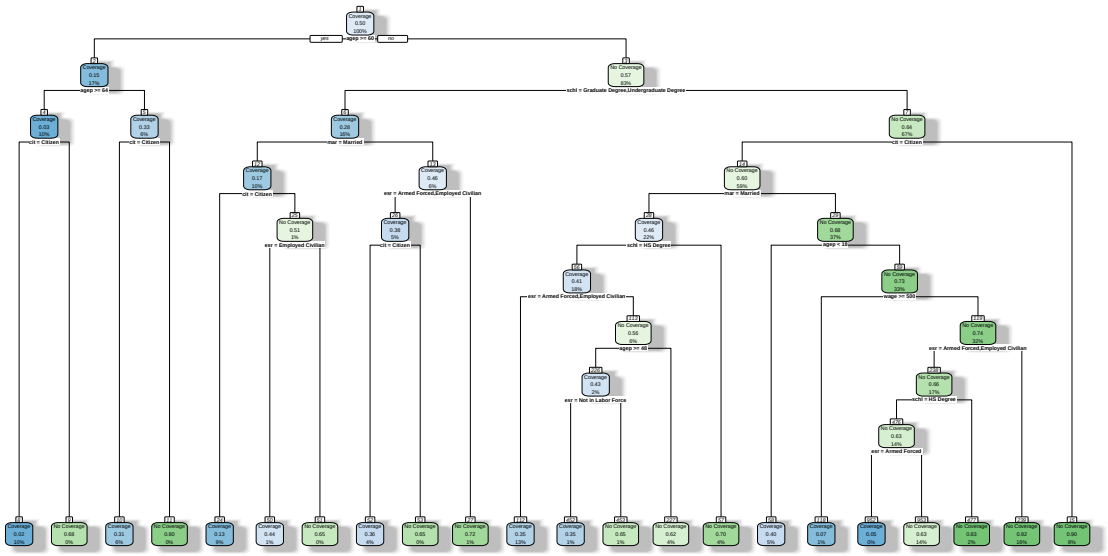


Figure 15: Decision tree for optimized complexity.

```
## 13 7.3725e-04     21   0.48894 0.50280 0.0074527
## 14 6.8810e-04     25   0.48599 0.50088 0.0074433
## 15 5.8980e-04     28   0.48393 0.50015 0.0074396
## 16 4.4235e-04     31   0.48172 0.49808 0.0074293
## 17 3.4405e-04     34   0.48039 0.49646 0.0074212
## 18 3.3176e-04     40   0.47832 0.49543 0.0074160
## 19 2.9490e-04     44   0.47700 0.49720 0.0074249
## 20 2.4575e-04     54   0.47405 0.49912 0.0074345
## 21 2.2117e-04     58   0.47302 0.49882 0.0074330
## 22 1.4745e-04     68   0.47066 0.50265 0.0074520
## 23 1.2287e-04     94   0.46682 0.50973 0.0074866
## 24 1.1796e-04    100   0.46609 0.51239 0.0074994
## 25 9.8299e-05    105   0.46550 0.51357 0.0075050
## 26 8.8469e-05    108   0.46520 0.51489 0.0075114
## 27 7.3725e-05    118   0.46432 0.51858 0.0075289
## 28 5.8980e-05    122   0.46402 0.51932 0.0075324
## 29 4.9150e-05    132   0.46343 0.52197 0.0075449
## 30 3.6862e-05    141   0.46299 0.52197 0.0075449
## 31 2.9490e-05    149   0.46270 0.52256 0.0075476
## 32 2.1064e-05    154   0.46255 0.52374 0.0075531
## 33 0.0000e+00    161   0.46240 0.52433 0.0075559
```

At this point, we'll re-run the decision tree once more with the updated *cp* value, assign the decision tree object to `fit.opt`, and plot the resulting decision tree. Notice how the rendered tree is significantly more complex relative to the default and interpretation may be more challenging with a plethora of criteria.

```
fit.opt <- rpart(coverage ~ agep + wage + cit + mar + schl + esr,
                 method = "class", data = train, cp = opt.select)
rpart.plot(fit.opt, shadow.col="gray", nn=TRUE)
```

In lieu of a thorough review of the learned rules, we may rely on a measure of variable importance, that is defined as follows:

$$\text{Variable Importance}_k = \sum \text{Goodness of Fit}_{\text{split, k}} + (\text{Goodness of Fit}_{\text{split,k}} \times \text{Adj. Agreement}_{\text{split}})$$

Where *Variable Importance* for variable k is the sum of *Goodness of Fit* (e.g. Gini Gain or Information Gain) at a given split involving variable k. In otherwords, a variable's importance is the sum of all the contributions variable k makes towards predicting the target. Below, we can see that the measure can be extracted from the `fit.opt` object. As it turns out, `age` is the most important factor.

```
#Extract variable importance list from fit object
  fit.opt$variable.importance
```

```
##      agep      schl       mar       cit       esr      wage
## 928.68854 559.14016 375.75373 271.38939 192.99726  68.11775
```

Using the `plotROC` package once again, we calculate the AUC score for each model to assess predictive performance on both the training and test set. One particularly striking difference is the switch in position of the *optimal* and $cp = 0$ curves: $cp = 0$ is higher in the training set, but are at the approximate safe height in test. This indicates that $cp = 0$ notably overfits, likely to the extra low bias of unpruned leafs.

```
#plotROC
  library(plotROC)
  library(gridExtra)

#Predict values for train set
  pred.opt.train <- predict(fit.opt, train, type='prob')[,2]
  pred.0.train <- predict(fit.0, train, type='prob')[,2]
  pred.default.train <- predict(fit, train, type='prob')[,2]

#Predict values for test set
  pred.opt.test <- predict(fit.opt, test, type='prob')[,2]
  pred.0.test <- predict(fit.0, test, type='prob')[,2]
  pred.default.test <- predict(fit, test, type='prob')[,2]

#Set up ROC inputs
  input.test <- rbind(data.frame(model = "optimal", d = test$coverage, m = pred.opt.test),
                  data.frame(model = "CP = 0", d = test$coverage,  m = pred.0.test),
                  data.frame(model = "default", d = test$coverage,  m = pred.default.test))
  input.train <- rbind(data.frame(model = "optimal", d = train$coverage,  m = pred.opt.train),
                  data.frame(model = "CP = 0", d = train$coverage,  m = pred.0.train),
                  data.frame(model = "default", d =  train$coverage,  m = pred.default.train))


#Graph all three ROCs
  roc.test <- ggplot(input.test, aes(d = d, model = model, m = m, colour = model)) +
              geom_roc(show.legend = TRUE) + style_roc()  + ggtitle("Test")
  roc.train <- ggplot(input.train, aes(d = d, model = model, m = m, colour = model)) +
              geom_roc(show.legend = TRUE) + style_roc()  +ggtitle("Train")

#Plot
  grid.arrange(roc.train, roc.test, ncol = 2)
```

Lastly, we can extract the AUC statistics using `calc_auc()`. As multiple AUCs were calculated, we will need to extract the labels for the AUCs from the `input` file in order to produce a a 'prettified' table using `xtable`. The resulting table below presents the results of the three models that were trained. For all models, we should expect that the training AUC will be greater than the test AUC. This is generally true, but occassionally the test AUC may be greater and is largely a matter of how the data was sampled.

Starting from the top of the table:

- *Full grown.* The unpruned tree is the most complex model, which means the model has a higher chance

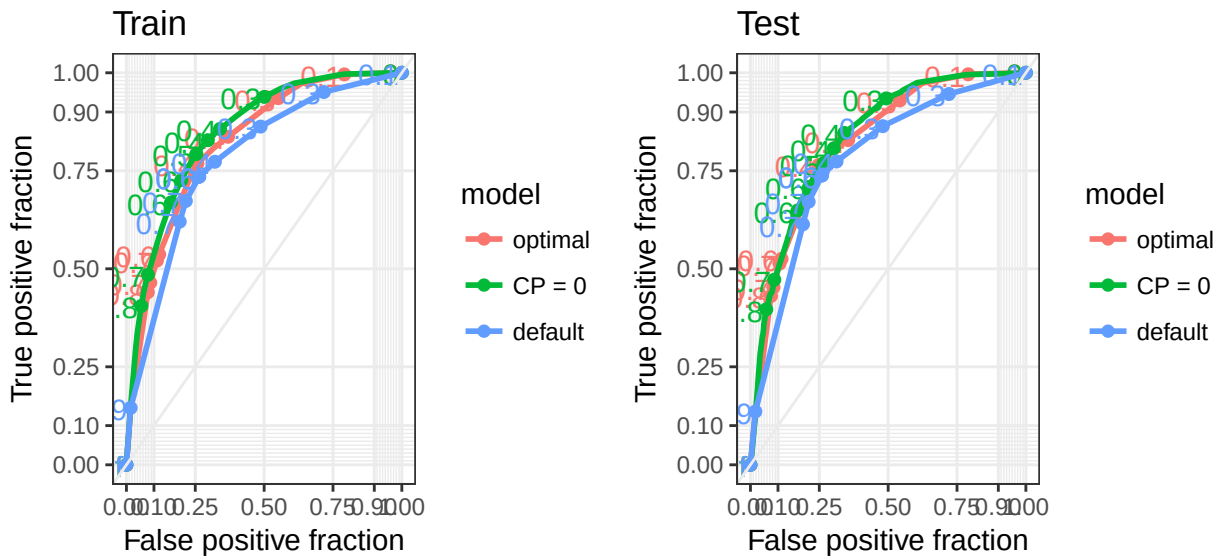


Figure 16: ROC curves for train and test sets.

of overfitting. This is characterized by an artificially inflated training AUC and a large drop in test AUC. As seen, the AUC drops from 0.88 to 0.826 in the test sample. The unreliable results of an unpruned tree are likely due to the algorithm's sensitivity to irregular noise at leafs.

- *Optimal*. The optimal tree achieves a consistent $AUC = 0.83$ with minimal loss of accuracy as an appropriate level of complexity was precisely tuned.
- *Default*. An underfit model will have consistently low performance in both training and testing. As we can see, these patterns are played out in the table below containing AUCs for each the default decision tree, the optimal model complexity and the fully grown tree.

As the result of tuning towards an optimal model, we can see that the decision tree yields a marked improvement over the kNN model's $AUC = 0.44$. For a social science problem, this is considered to be a decent result.

```
#Assemble a well-formatted table
  tab <- data.frame(model = unique(input.test$model),
                    train = round(calc_auc(roc.train)$AUC,3),
                    test = round(calc_auc(roc.test)$AUC,3))
```

model	train	test
optimal	0.824	0.824
CP = 0	0.847	0.835
default	0.781	0.780

1.6.1 Random Forests

```
# Load ACS health care data
  library(digIt)
  health <- digIt("acs_health")

# Convert characters into discrete factors
  factor_vars <- c("coverage", "mar", "cit", "esr", "schl")
  for(var in factor_vars){
    health[,var] <- as.factor(health[,var])
  }

# Randomly assign
  set.seed(100)
  rand <- runif(nrow(health)) > 0.5
```

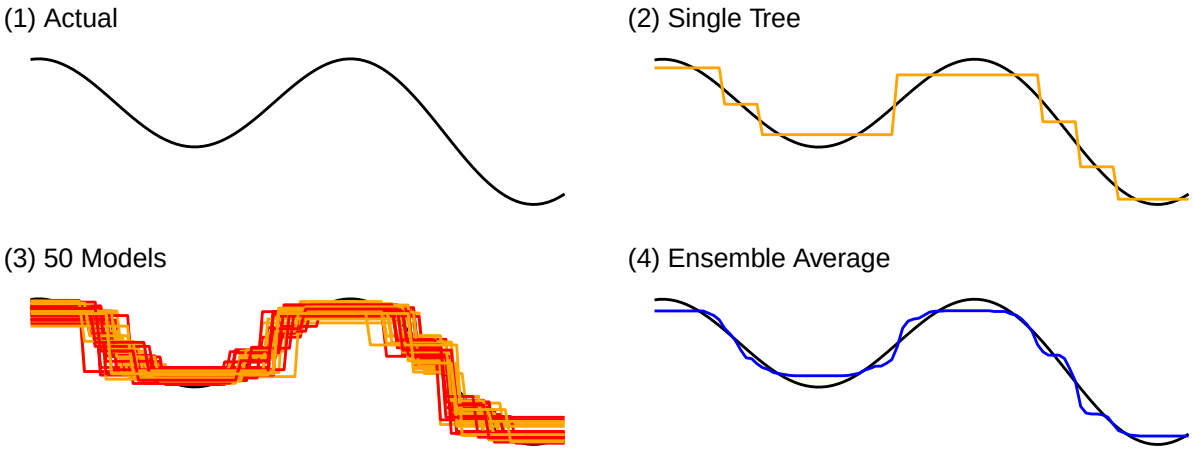


Figure 17: Comparison of results of applying a single model to fit a curve versus an ensemble of models.

jagged fashion (#2). That one tree may impose biases on the data, perhaps through how the tree is pruned or the assumption that the jagged approximation is appropriate, which may then translate into greater variance in predictions. One could imagine that the structure of that one tree may have happened by chance, and under different situations, the fit could be better.

Bootstrapping can help. Recall from elementary statistics that bootstrapping is defined as any statistical process that involves sampling records with replacement. By bootstrapping a sample, we treat a sample like a population, we can expose and characterize the qualities of an estimator under various scenarios already available in the data, which in turn produces an empirical probability distribution for predictions using the estimator. We can bootstrap the decision tree by (1) sampling the data with replacement up to the full size of the sample, then (2) run the decision tree. The result of repeating the process 50 times is (graph #3) produces a result that appears to be more organic and more accurate. This process of *bootstrapping* and *aggregating* the results is referred to as *bagging*.

Applying bagging to decision trees may not necessarily be enough to develop a well-balance prediction. In the social sciences and public policy, it is generally assumed that a model's specification is a choice left to the analyst; However, it may also be a source of methodological bias.

Random forests can help. The technique, as crystallized in Breiman (2001), is an extension of decision trees using a modified form of bootstrapping and ensemble methods to mitigate overfitting and bias issues. Not only are individual records bootstrapped, but input features are bootstrapped such that if K variables are in the training set, then k variables are randomly selected to be considered in a model such that $k < K$. Each bootstrap sample is exhaustively grown using decision tree learning and is left as an unpruned tree. The resulting predictions of hundreds of trees are ensembled. The logic is described below.

Pseudo-code

```
Let S = training sample, K = number of input features
  1. Randomly sample S cases with replacement from the original data.
  2. Given K features, select k features at random where k < K.
  3. With a sample of s and k features, grow the tree to its fullest complexity.
  4. Predict the outcome for all records.
  5. Out-Of-Bag (OOB). Set aside the predictions for records not in the s cases.
Repeat steps 1 through 5 for a large number of times saving the result after each tree.
Vote and average the results of the tree to obtain predictions.
Calculate OOB error using the stored OOB predictions.
```

The *Out-Of-Bag* (OOB) sample is a natural artifact of bootstrapping: approximately one-third of observations are naturally left un-selected, which can be used as the basis of calculating each tree's error and the overall

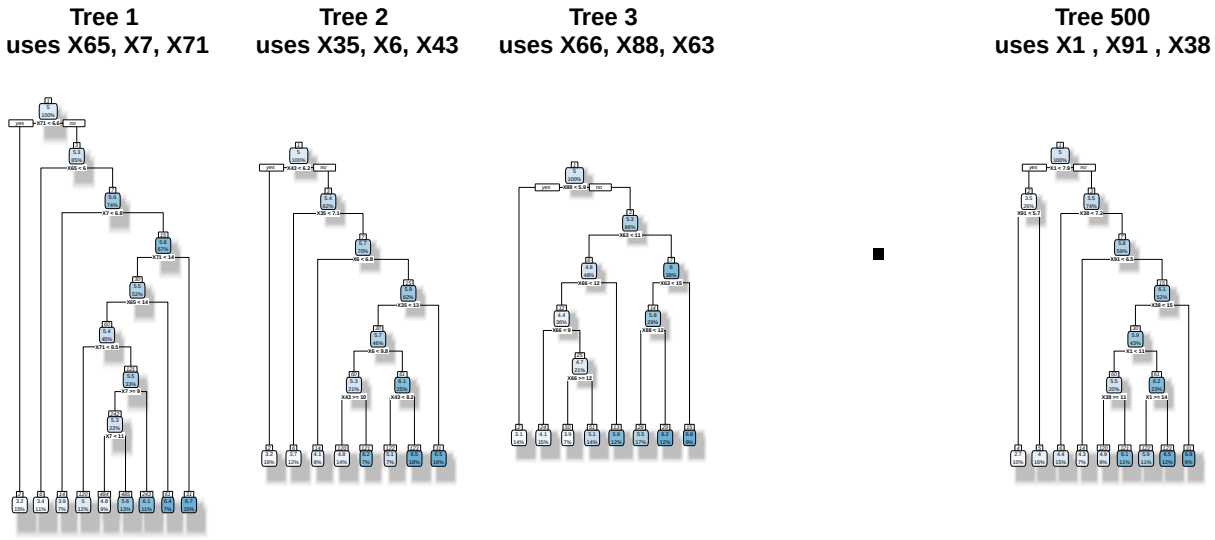


Figure 18: Random Forests construct hundreds of trees sampling from both observations and features, then combine the trees into one prediction through voting.

model error. Think of it as a convenient built in test sample.

How about interpretation? Unlike decision trees, it is not a simple task to deduce rules or criteria that describe the target variable. Instead, random forests use *variable importance*, which, like for a decision tree, measures the contribution of a feature to the homogeneity of a classifier. Unlike decision trees, variable importance for a Random Forest is calculated as the mean decrease in the Gini coefficient of a split relative to the Gini coefficient of the root node. Gini coefficients measures homogeneity on a scale of 0 to 1, where 0 is perfect homogeneity and 1 is perfect heterogeneity. The Gini changes are summed for each variable and normalized.

1.6.1.1 Tuning

Whereas methods like regression have a closed form solution, Random Forest require tuning as optimal models need to be searched for under different conditions. The principal tuning parameters include: Number of features and number of trees.

- *Number of input features.* As k number of parameters need to be selected in each sampling round, the value of k needs to minimize the error on the OOB predictions.
- *Number of trees* influences the stability the Variable Importance metric that is commonly used to infer variable influence in decision tree learning. More trees help to stabilize the Variable Importance estimate. To determine the number of trees, keep adding trees to a sample until the OOB error for a randomly select set of trees is approximately equal to that of the ensemble.

1.6.1.2 Random Forests in Practice

Like decision trees, much of Random Forests rely on easy to use methods made available through the `randomForest` library. There are a couple of ways to run the algorithm, including:

```
randomForest(formula, data, method, mtry, ntree)
```

where: - `formula` is an object containing the specification to be estimated. Note that - `data` is a data frame. - `mtry` is the number of variables to be randomly sampled per iteration. Default is $\sqrt{k}$ for classification trees. - `ntree` is the number of trees. Default is 500.

Using the same formula as the `rpart()` function, we can train a naive Random Forest and check the OOB error. Approximately 75.6% of observations in the OOB sample were correctly classified using 2 randomly

selected variables in each of the 500 trees.

```
#Load randomForest library
  library(randomForest)

#Only complete obs
  train <- na.omit(train)

#Run Random Forest
  spec <- as.formula("coverage ~  agep + wage + cit + mar + schl + esr")
  fit.rf <- randomForest(spec, data = train, mtry = 2, ntree = 500)

#Check OOB error
  fit.rf
```

```
##
## Call:
##  randomForest(formula = spec, data = train, mtry = 2, ntree = 500)
##                Type of random forest: classification
##                      Number of trees: 500
## No. of variables tried at each split: 2
##
##         OOB estimate of  error rate: 24.43%
## Confusion matrix:
##             Coverage No Coverage class.error
## Coverage        4958        1856   0.2723804
## No Coverage     1466        5316   0.2161604
```

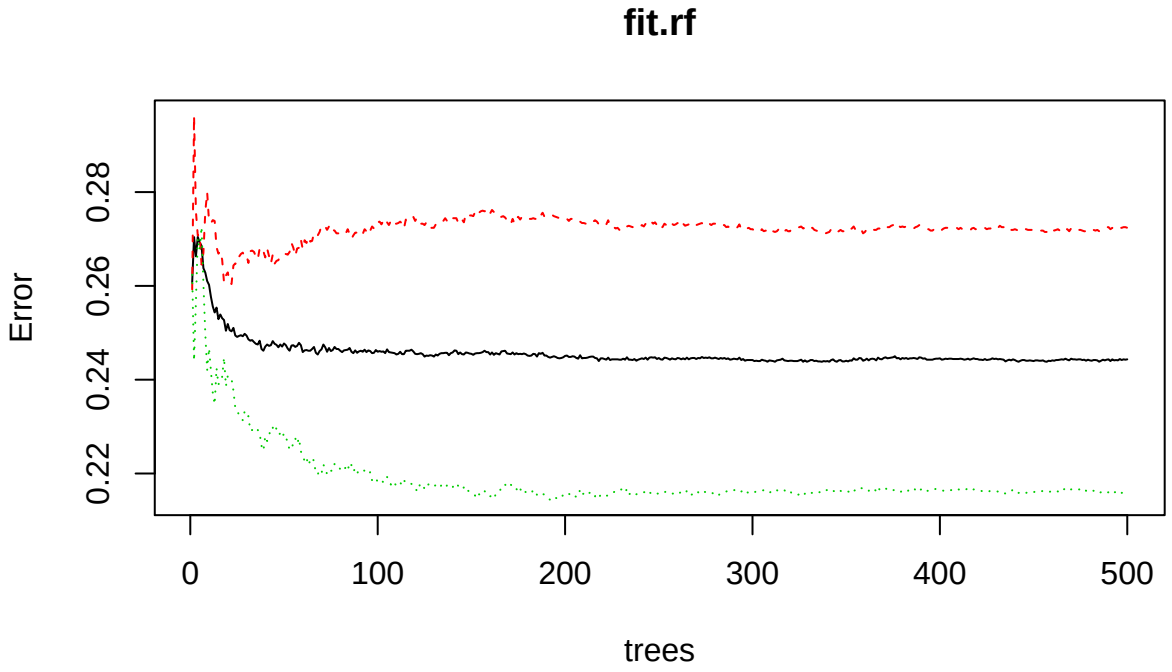
Using the `importance()` method, we can see the Mean Decrease Gini, which calculates the mean of Gini coefficients. `agep` has the largest value of 801.3155193, indicating that age is the best predictor of coverage; However, the values themselves do not have any meaning outside of a comparison with other Gini measures.

```
importance(fit.rf)
```

```
##      MeanDecreaseGini
## agep         801.31552
## wage          77.79932
## cit          321.67357
## mar          446.41256
## schl         444.07750
## esr          234.16484
```

By default, the `randomForests` library sets the number of trees to equal 500. By plotting the fit object, we can see how OOB error and the confidence interval converges asymptotically as more trees are added to the ensemble. Otherwise stated, more trees will help up to a certain point and the default is likely more than enough.

```
  plot(fit.rf)
```



As we know that $n = 500$ trees is more than enough, we will now need to tune the tree for the number of variables. To tune the algorithm, we will use the `tuneRF()` method. The method searches for the optimal number of variables per split by incrementally adding variables. While it's a useful function, it is relatively verbose. In addition to the target and input features, a number of other parameters need to be specified:

```
tuneRF(x, y, ntreeTry,  mtryStart, stepFactor, improve, trace, plot)
```

where: - `x` is a data frame or matrix of input features. - `ntreeTry` is the number of trees used in each iteration of tuning. - `mtryStart` is the number of variables to start. - `stepFactor` is the number of additional variables tested per iteration. - `improve` is the minimum relative improvement in OOB error for the search to go on. - `trace` is a boolean that indicates where to print the search progress. - `plot` is a boolean that indicates whether to plot the search results.

Below, we conduct a search from `mtryStart = 1` with a `stepFactor = 2`. The search result indicates that *2* variables per split are optimal.

```
#Search for most optimal number of input features
  fit.tune <- tuneRF(x = train[,3:ncol(train)], y =  train[,2], ntreeTry = 500,
                     mtryStart = 1, stepFactor = 2,
                     improve = 0.001, plot = TRUE)
```

```
## mtry = 1  OOB error = 26.53%
## Searching left ...
## Searching right ...
## mtry = 2     OOB error = 24.37%
## 0.08123094 0.001
## mtry = 4     OOB error = 25%
## -0.02564876 0.001
```

```
#Extract best parameter
  tune.param <- fit.tune[fit.tune[, 2] == min(fit.tune[, 2]), 1]
```

Using the optimal result, we can plug back into the `randomForest()` method and re-run. However, as the default model already has the same parameters as the optimal model, we can proceed to calculating the model accuracy. Comparing the training and test models for the Random Forest algorith, we see a large drop in the AUC between train and test, indicating quite a bit of overfitting.

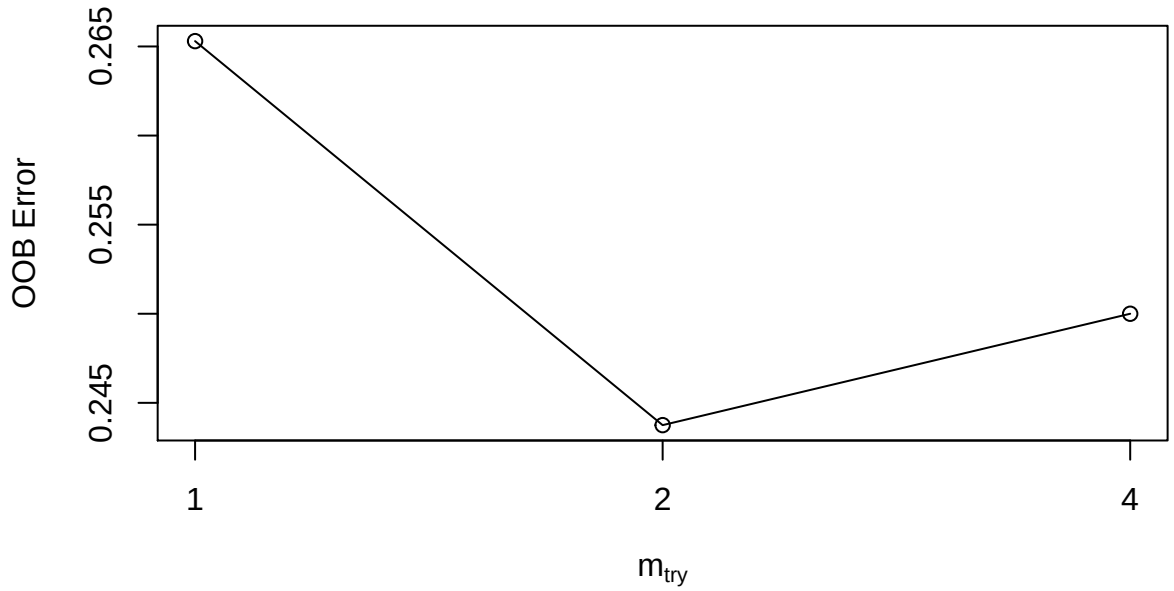


Figure 19: Random Forest tuning result (m = number of features, OOB Error = out of sample error).

```
#plotROC
  library(plotROC)

#Predict values for train set
  pred.rf.train <- predict(fit.rf, train, type='prob')[,2]

#Predict values for test set
  pred.rf.test <- predict(fit.rf, test, type='prob')[,2]


#Set up ROC inputs
  input.rf <- rbind(data.frame(model = "train", d = train$coverage, m = pred.rf.train),
                    data.frame(model = "test", d = test$coverage,  m = pred.rf.test))

#Graph all three ROCs
  roc.rf <- ggplot(input.rf, aes(d = d, model = model, m = m, colour = model)) +
              geom_roc(show.legend = TRUE) + style_roc()  +ggtitle("Train")

#AUC
  calc_auc(roc.rf)
```

```
##   PANEL group       AUC
## 1     1     1 0.8374157
## 2     1     2 0.8316592
```