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 look 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

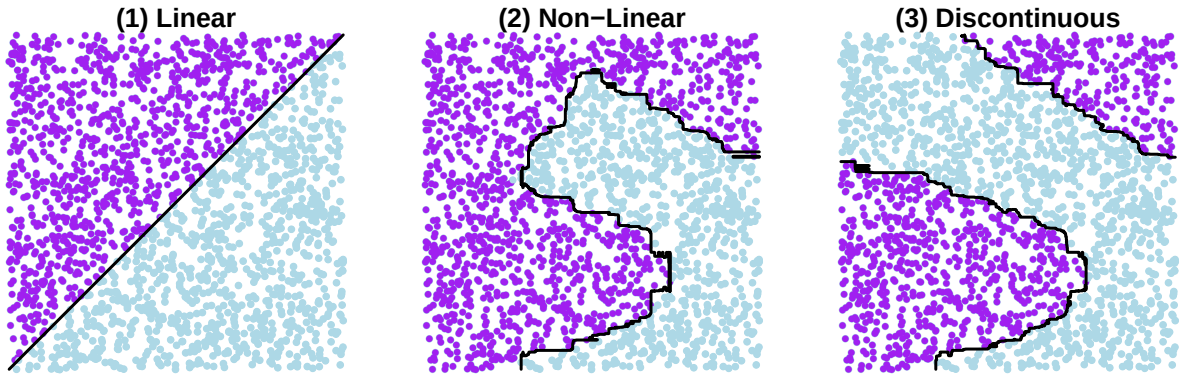


Figure 1: Linear, Non-Linear and Discontinuous Classification Problems.

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. In policy, we tend to start from a normative theory of how a phenomenon functions – perhaps a simple linear explanation. 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.

```
## Warning: glm.fit: algorithm did not converge
```

```
## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
```

```
## randomForest 4.6-12
```

```
## Type rfNews() to see new features/changes/bug fixes.
```

As it turns out, the function for drawing the decision boundary can take on many forms and each balances interpretability and accuracy. A simple linear pattern can be produced 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 isolated patterns. Non-parametric methods offer a far more flexible solution to prediction 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*

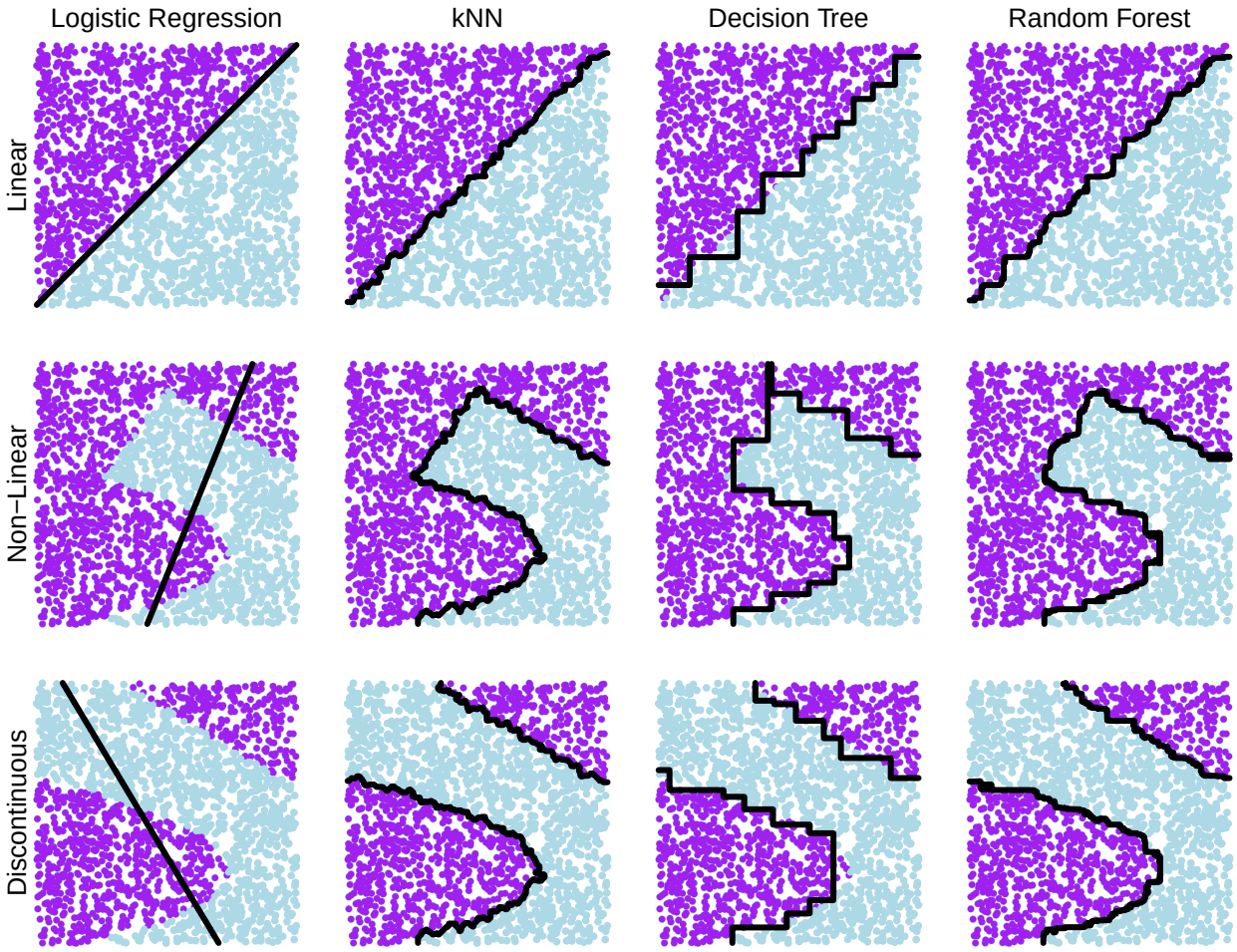


Figure 2: Linear, Non-Linear and Discontinuous Classification Problems.

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.

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.

The idea of separability manifests itself different in discrete and continuous values. With discrete variables, separability is a matter of placing as many *True Positives* and *True Negatives* as represented in a $2x2$ table below. This translates into placing the vast majority of observations into the upper left cell and bottom left cell. In the first example below, we have a separable case in which nearly 80% of observations are distributed along the diagonal.

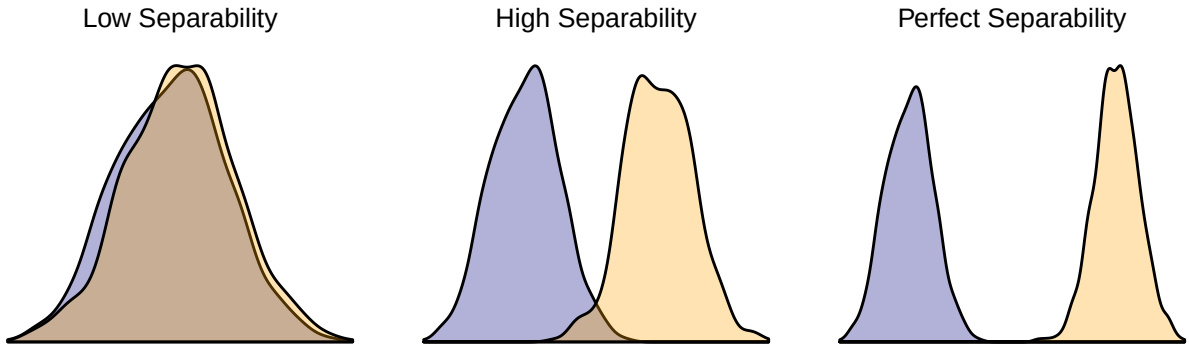


Figure 3: Separability of two classes given a continuous variable.

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). In the case below, we see that the majority of cases have been incorrectly classified.

Table 2: Separability of two classes given a unseparable discrete variable.

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

When working with continuous variables, separability is in terms of differences in means and distributions 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.

Measures of accuracy

As alluded to when describing separability, 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.

Table 4: Confusion matrix for perfectly accurate predictions

	Predicted (-)	Predicted (+)
Actual (-)	50	0
Actual (+)	0	50

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.

[10] https://gking.harvard.edu/files/gking/files/0s.pdf

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 that support different analytical use cases and help accentuate different policy priorities.

- Take the reader through an example of how to interpret different

While its simplicity makes for a great summary measure, it can be misleading in samples with class imbalance. For example, if 90% of a sample is composed of the negative class ($Y = 0$) and the predictions correctly classify all negatives and none of the positives, the accuracy score would still be 90%.

Measure	Formula	What It Answers
Individual Measures		
True Positive Rate (TPR)	$TPR = \frac{TP}{TP+FN}$	What proportion positive 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 I error.
Predicted Positive Value (PPV) or Precision	$PPV = \frac{TP}{TP+FP}$	Proportion of positive cases that were incorrectly predicted negative. This is also known as the false alarm rate or Type I error.
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
Receiving-Operating Characteristic (ROC) Curve	sd	asd

Interpretability versus prediction

When selecting models for classification tasks, data scientists often times balance interpretability and predictive accuracy. It is similar to the idea of buying clothing of a generic size (e.g. small, medium, large) or having it custom tailored. The former captures the gist of one's body dimensions and it is easy to explain: *I bought a medium*. A custom tailored piece of clothing requires many adjustments beyond the size, allowing a master tailor to mold the clothing to features of the body and to one's comfort.

The same idea applies to classifiers.

In the social and natural sciences, classification methods like logistic regression are favored for their use in *parameter estimation* to infer relationships between variables. The model coefficients are directly interpretable,

showing how a variable can contribute or detract from the probability of an outcome holding all else constant. Inferring relationships from estimated parameters in turn facilitate narratives for communicating insight. But there is a tradeoff. While linear methods may extract the gist of the relationships, they may miss the finer movements necessary for a reliable prediction.

If the goal is to produce a highly accurate prediction, it may be worth considering a body of exciting methods that have arisen from statistics and computer science that are optimized for predictive accuracy. 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 prizes interpretability versus accuracy.

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.

1.3 Six common classifiers

Six algorithms cover much of the common use cases that are relevant for public policy and other applied fields. Each method has its strengths and weaknesses, and each is more appropriate in certain contexts than others. Thus, each method set is illustrated using data that are well-suited for its purposes.

Method	Common Uses	Key Ass
k-Nearest Neighbors	Imputation of missing data; Relating data that is collected along an equally-spaced grid.	All varia
Logistic Regression	General use method; Coefficients allow for narrative building.	The bou
LASSO logistic	Cases where there are a large number of variables	
Decision Tree	Well-suited for teasing out interactions; Leafs in the tree can be used as discrete profiles.	Tree will
Random Forest	Well-suited for teasing out interactions and surfacing variables that hold any separability.	Tree will
Neural Networks	Data distributed along an equal interval (e.g. imagery, sound);	

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

1.3.1 K-Nearest Neighbors (KNN)

As hurricanes become more intense and leave a trail of destruction, city services 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.

1.3.1.1 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 to determine the value of y. It is instance-based as each prediction is determine on a case-by-case basis using a pre-defined procedure.

The procedure is simple. For each case y_i:

1. *Distance*. First, we calculate the *distance* to all other records with known outcomes. Distance most commonly takes the form of Euclidean distance (below), 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 case.

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

2. *Voting*. For the k nearest observations, take the average of y (below). Basically, for each class j in Y, we take the average of k observations that are closest based on X. In effect, this procedure yields a local conditional probability for each observation, which is converted into a predicted class through *majority voting*.

$$Pr(Y = j|X) = \frac{1}{k}\sum_{i \in A} I(y^i = j)$$

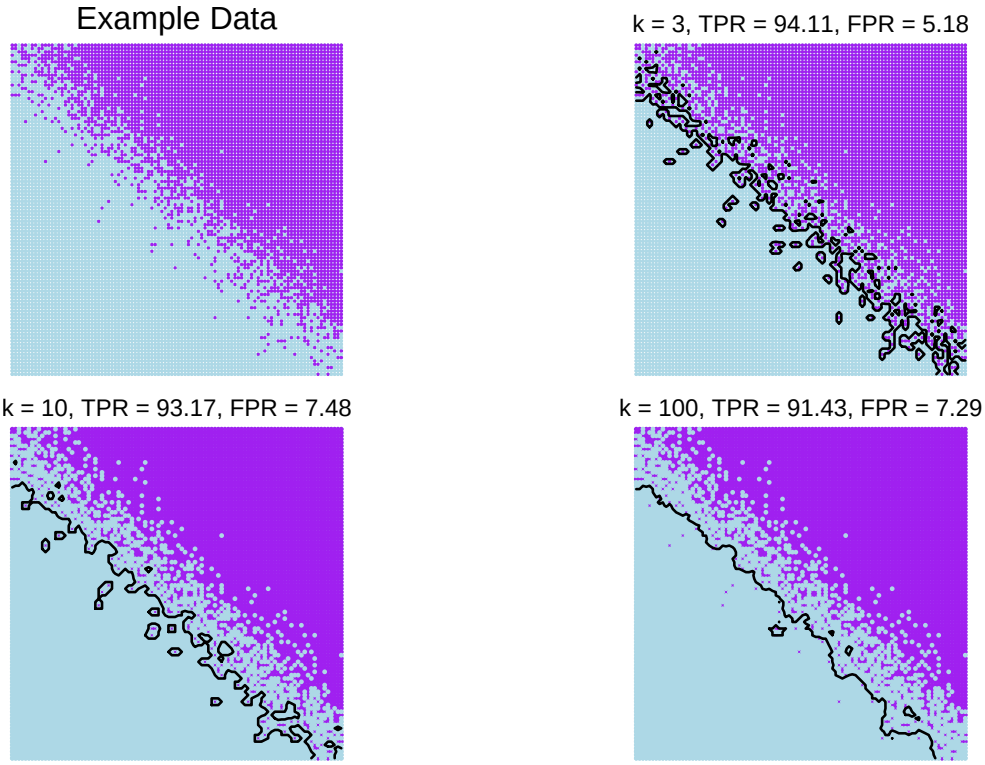
3. *Tuning*. The method is sensitive to the value of k, requiring tuning – or testing different values of k. When $k = 10$, the conditional probability for y_i reflects the 10-nearest neighbors. When $k = n$, the conditional probability is the sample mean.

The procedure described above yields the results for just one value of k. However, kNNs, like many other algorithms, 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. Often times, tuning of hyperparameters involves a *grid search*, a process whereby a range

of possible hyperparameters is determined and the algorithm is tested at equal intervals from the minimum to maximum of that tuning range.

To illustrate this, a two-dimensional dataset with a target y that takes of values 0 and 1 has been plotted below. Graph (1) plots the points, color-coded by their labels. Graph (2), (3), and (4) show the results of a grid search along intervals of a log_{10} scale, where the background is color-coded as the predicted label for the corresponding value of k. In addition to k, two measures are provided above each graph to help contextualize predictions: the True Positive Rate or TPR and the True Negative Rate or TNR.

The TPR is defined as $\#TPR = \frac{\text{Number of values that were correctly predicted}}{\text{Number of actual cases values}}\#$. The TNR is similarly defined as $\#TNR = \frac{\text{Number negative values that were correctly predicted}}{\text{Number of actual negative values}}\#$. Both are measures bound between 0 and 1, where higher values indicate a higher degree of accuracy. A high TPR and low TNR indicates that the algorithm is ineffective in distinguishing between positive and negative cases. The same is true with a low TPR and high TNR. This is exactly the case in Graph (4) where all points are classified as $Y = 1$, which is empirically characterized by $TNR = 0.02$ and $TPR = 1$.



1.3.1.2 Which K is the right K?

The accuracy of a KNN model is dependent on finding the right value of k. Thus, to optimize for accuracy, try multiple values of k and compare the resulting accuracy values. It is helpful to first see that when $k = n$, kNNs are simply the sample statistic (e.g. mean or mode) for the whole dataset. Below, the True Positive Rate (TPR, blue) and True Negative Rate (TNR, green) have been plotted for values of k from 1 to n. The objective is to ensure that there is a balance between TPR and TNR such that predictions are accurate. Where $k > 20$, the TPR is near perfect. For values of $k < 10$, TPR and TNR are more balanced, thereby yielding more reliable and accurate results.

There are other factors that influence the selection of k:

- *Scale.* kNNs are strongly influenced by the scale and unit of values of x as ranks are dependent on straight Euclidean distances. For example, if a dataset contained measurements of age in years and wealth in dollars, the units will over emphasize income as the range varies from 0 to billions whereas

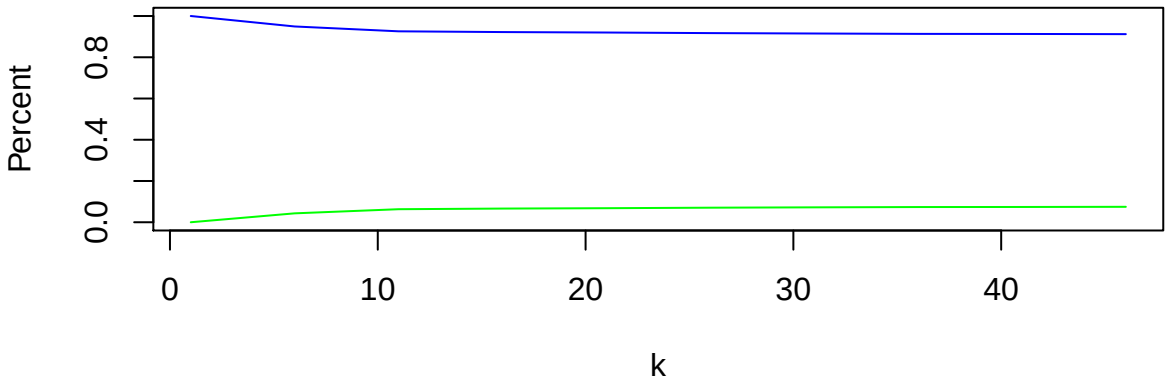


Figure 4: True Positive Rate (TPR = blue) and True Negative Rate (FPR = green) performance for varying values of k

age is on a range of 0 to 100+. To ensure equal weights, it is common to transform variables into standardized scales such as:

- Range scaled or

$$\frac{x - \min(x)}{\max(x) - \min(x)}$$

  yields scaled units between 0 and 1, where 1 is the maximum value
- Mean-centered or

$$\frac{x - \mu}{\sigma}$$

  yield units that are in terms of standard deviations

- *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.

1.3.1.3 Tips

KNNs are efficient and effective under certain conditions:

- KNNs can handle target values that are either discrete or continuous, making the approach relatively flexible. However, best performance is achieved when the input features should are in the same scale (e.g. color values in a grid).
- They are best used when there are relatively few features as distances to neighbors need to be calculated for each and every record and need to be optimized by searching for the value of k that optimizes for accuracy. In cases where data is randomly or uniformly distributed in fewer dimensions, a trained KNN is an effective solution to filling gaps in data, especially in spatial data.
- KNNs are not interpretable as it is a nonparametric approach – it does not produce results that have a causal relationship or illustrate. Furthermore, kNNs are not well-equipped to handle missing values.

1.3.1.4 Use Case: Anticipating the extent of damage from a storm

As it common and proper, 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 the `kknn()`. While both functions can be fairly easily written from scratch (and we encourage new users to write their own as was demonstarted in the previous chapter), we will plow forth with using the library.

To start, we will load the `kknn` library:

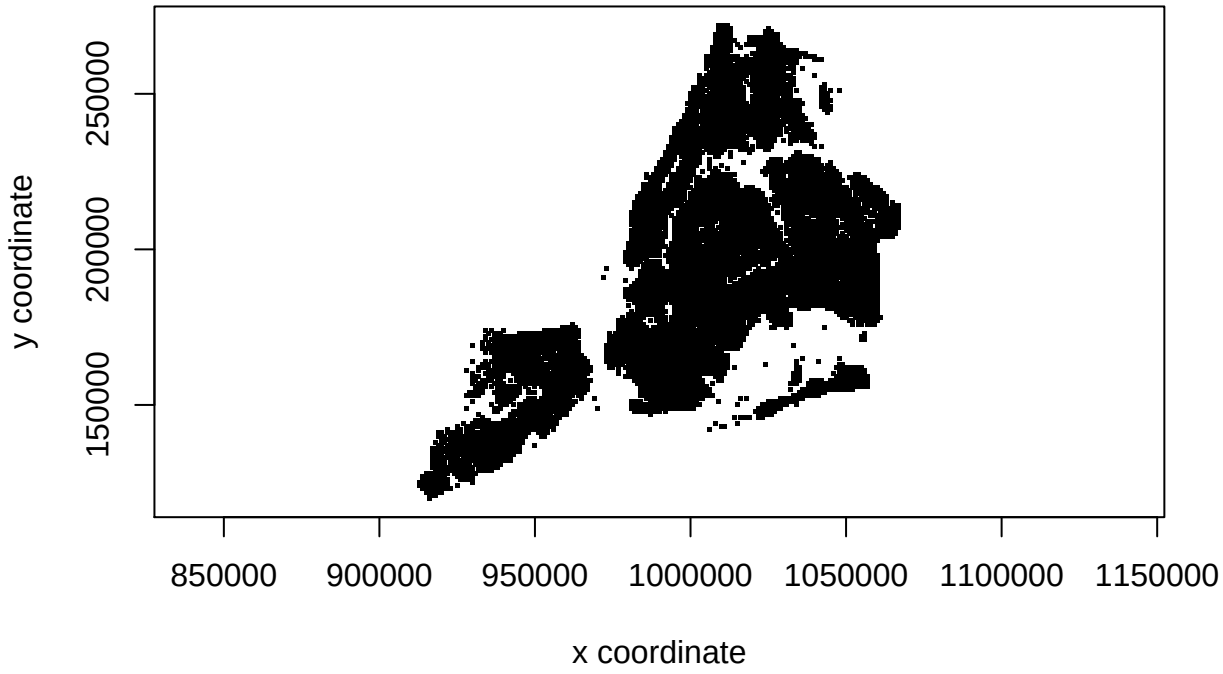
```
#Call "class" library
  library(kknn)
```

Next, we'll load in our data set

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

and take a look at what the geographic distribution of the data.

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

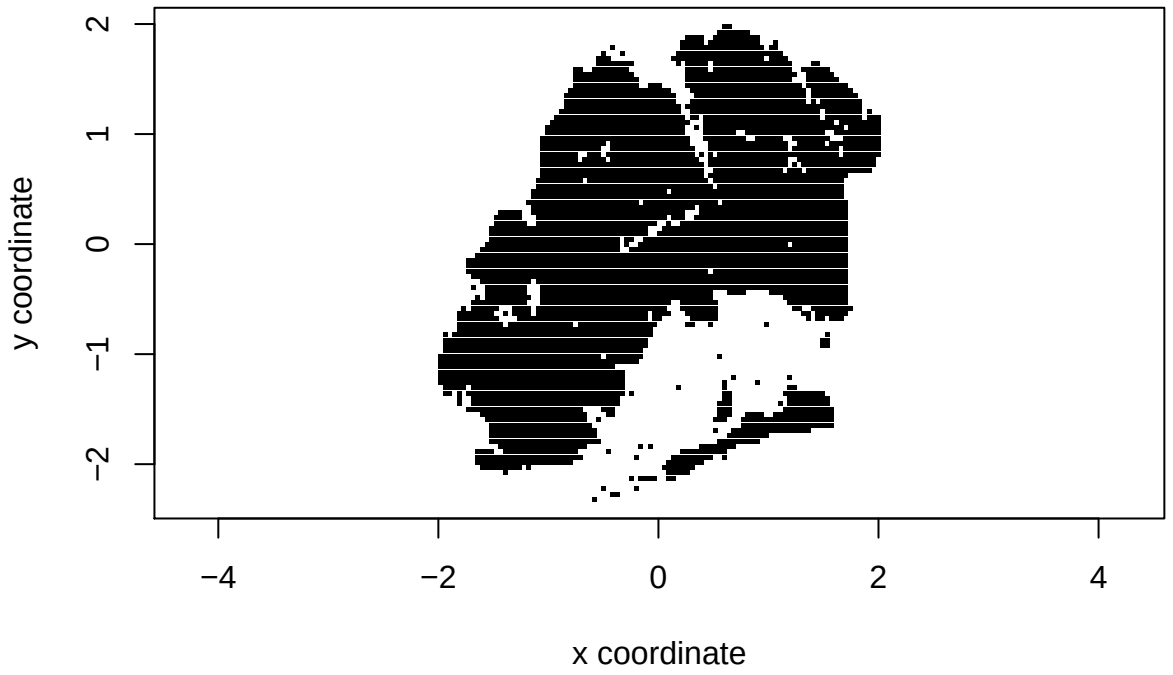


In New York City, only the two largest boroughs, Brooklyn and Queens, share the same land mass. The rest are separated by rivers. For a more consistent data set, we subset our data using the `boro` variable to focus.

```
#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)
```




```
#Set up data
  train <- subset(nyc, !is.na(tree.sandy),
                  select = c("ycoord", "xcoord", "tree.sandy"))
  test <- subset(nyc,
                 select = c("ycoord", "xcoord", "tree.next7"))
```

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 10-folds cross validation up to $k = 100$ for three kernel (rectangular, biweight and inverse) assuming L1-distances. While the command is simple, it runs the kNN algorithm for 2000 times (20 cross-validation models for each k - kernel combination).

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

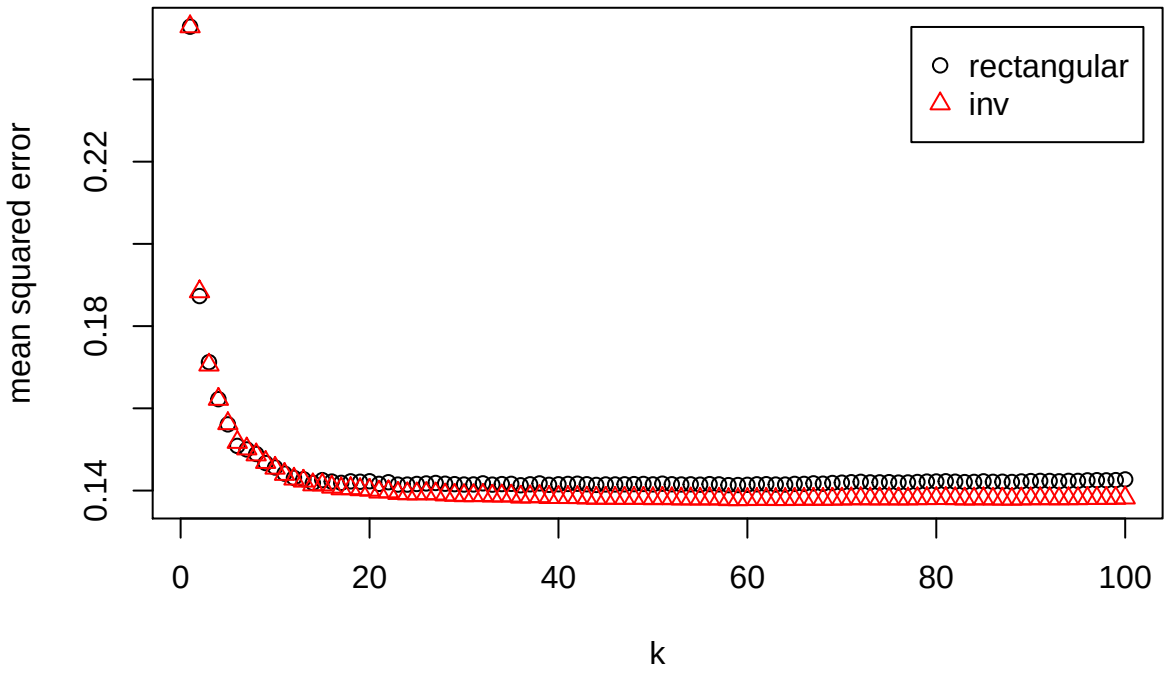


Figure 5: 20-fold cross validated errors for k = 1 to k = 100

The resulting model object contains the cross-validation error log in the `MISCLASS` attribute, which has been plotted below, as well as `best.parameters` that indicates that $k = $ 59 using an inverse distance kernel yields the lowest error.

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

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

With the KNN algorithm tuned, we can now use the `kknn()` function to score the test set. 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. 2 = euclidean, 1 = binary).

Notice that in the following code block, we train the kNN and apply it to the test sample all in one step.

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

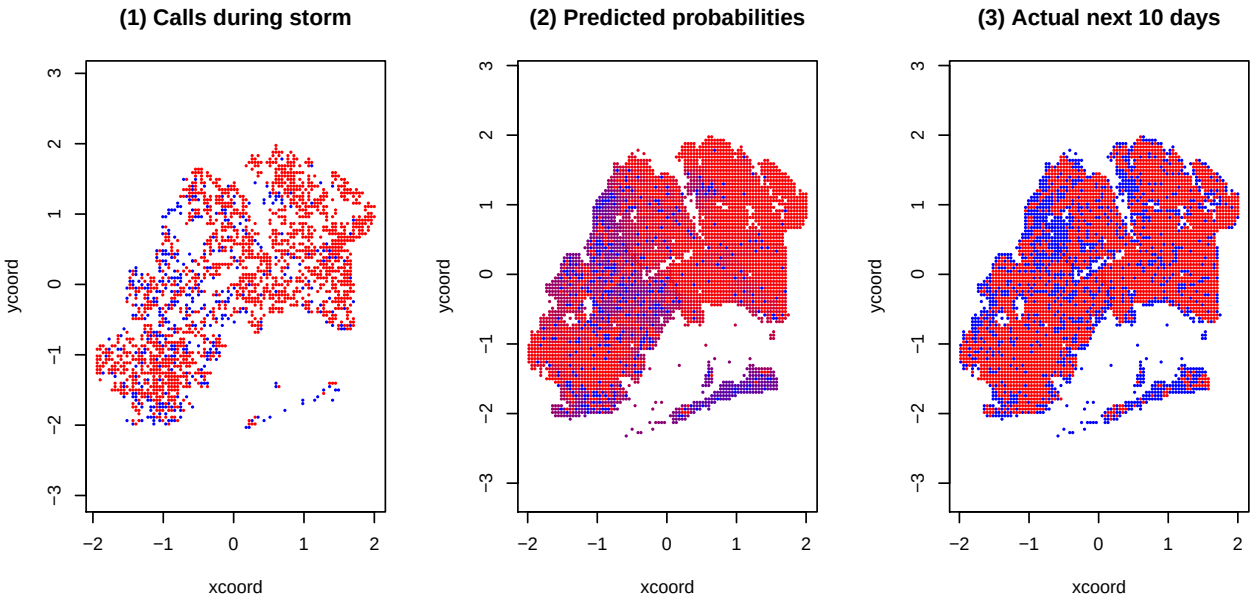


Figure 6: Graphical comparison of actual and predicted areas with reported downed trees. Red indicates at least one tree was reported in a given 0.359 square-mile area

```
              kernel = "inv")

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

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 10 days that followed. In the first panel, we see that a cloud of points capture the gist of the downed tree pattern. About 25.7% of all cells in Brooklyn and Queens made at least one call to 311, of which 78% made a request to deal with a tree issue.

```
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 10 days",
      col = rgb(test$tree.next7, 0, 1 - test$tree.next7, 1),
      cex = 0.4, pch = 16, asp = 1)
```

Using the extracted probabilities, we now can calculate the accuracy using the True Positive Rate (TPR) using a probability cutoff of 0.5. Typically, one would expect a 2×2 matrix given a binary label where the accuracy rate can be calculated based on the diagonals. In this case, prediction accuracy was `r`, indicating that the model performs reasonably well.

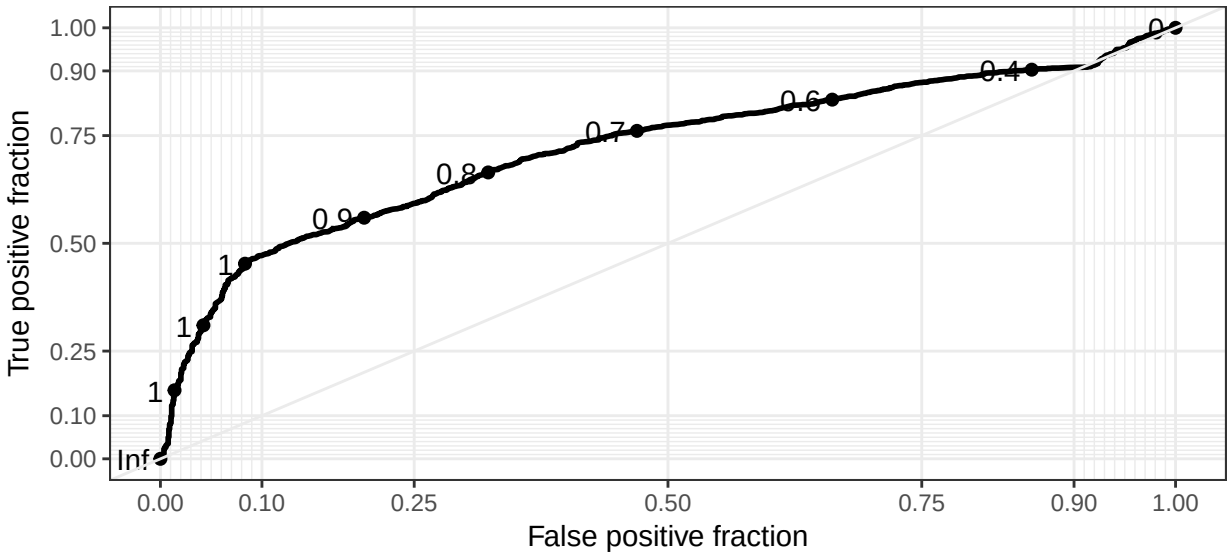


Figure 7: ROC curve out of sample

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 generally a decent level of accuracy for this type of problem.

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

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

Despite the promising result, we are reminded by there are a few one should ask the following question: *Is there a better classifier?*

1.3.1.5 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:

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.

1.3.2 Logistic Regression

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.[12] ^^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.

1.3.2.1 Under The Hood

Before we dive into the particulars, we will cut straight to the chase: logistic regression is the workhorse of many fields, but it is truly well-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, which may not afford the flexibility of KNNs.

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 $z = 0$.

As it turns out, we can express the relationship between y, x_1, and x_2 as a linear model similar to OLS:

$$y = w_0 + w_1 x_1 + w_2 x_2 + \epsilon$$

where y 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 weights can be interatively adjusted so that we maximize the chance that the all the coefficients can jointly maximize the chance of accurately mimicking the target. While the innards of MLE are beyond the scope of this text, refer to *Elements of Statistical Learning* [13] or the more introductory version *Introduction to Statistical Learning.*[14]

[12] https://www.cdc.gov/cancer/lung/basic_info/risk_factors.htm
[13] Ref needed
[14] Ref needed

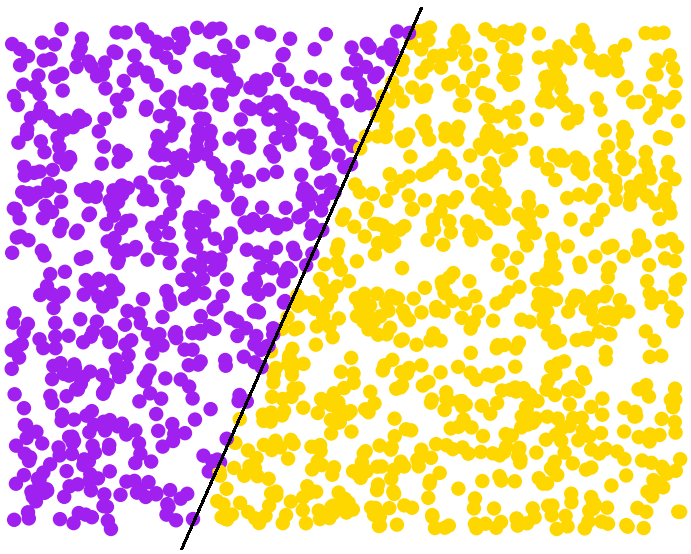
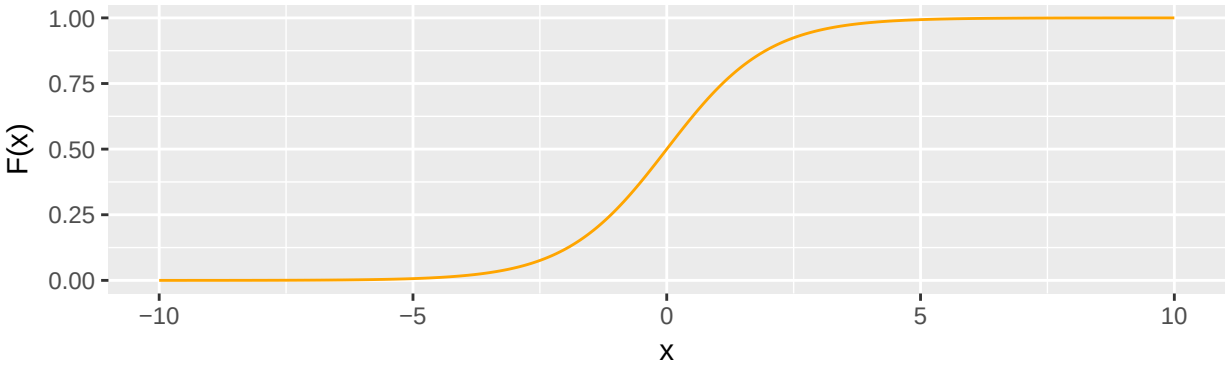


Figure 8: A linearly separable problem.

If treated as a typical linear model with a continuous outcome variable, we run the risk that $\hat{y}$ would exceed the binary bounds of 0 and 1 and would thus make little sense. Imagine if $\hat{y}$, the predicted value of y were -103 or +4 – *what would that mean in the case of a binary variable?* This is a logical shortcoming of a linear model for binary outcomes. Statisticians have cleverly solved the bounding problem by inserting the predicted output into a logistic function:

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

For a variable x that ranges for -10 to +10, the logit transformation converges to +1 where $x > 0$ and to 0 where $x < 0$. This S-shaped curve is known as a *sigmoid* and 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:

$$p = Pr(Y = 1|X) = F(z) = \frac{1}{1 + e^{-(w_0 + w_1 x_1 + w_2 x_2)}}$$

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 can interpret coefficients as an odds ratio, implying that two quantities are compared. 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-formulated as:

$$pr(success) = \frac{e^{(w_0+w_1x_1+w_2x_2)}}{1+e^{(w_0+w_1x_1+w_2x_2)}}$$
$$pr(failure) = \frac{1}{1+e^{(w_0+w_1x_1+w_2x_2)}}$$

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

$$log(odds) = log(\frac{p}{1-p}) = w_0 + w_1x_1 + w_2x_2$$

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 $odds = e^z = e^{w_0+w_1x_1+w_2x_2}$, then we can express an odds ratio as a marginal 1 unit increase in x_1 comparing $odds(x+1)$ over $odd(x+0)$:

$$OR = \frac{e^{w_0+w_1(x_1+1)+w_2x_2}}{e^{w_0+w_1(x_1+0)+w_2x_2}} = e^{w_1}$$

After a little arithmetic, it turns out the OR is simply equal to e^{w_1}, which can be interpreted as a multiplicative effect or a percentage effect ($100 \times (1 - e^{w_1})\%$). 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 binary variable (x) to whether people have health care insurance, the result may look as follows:

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

As it, the coefficients provide little information other than the fact that a positive cofficient indicates that an increase in an input x increases the chances of y. By exponentiating the cofficients 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.

1.3.2.2 Tips

Training a logistic regression is a fairly straight forward process as will be demonstrated in later sections. However, there are a number of key issues issues to keep in mind.

Tuning a logistic regression is a matter of selecting combinations of features (variables): finding the right combination of features will maximize classification accuracy. The process generally starts from a hypothesis of how variables are related to a target, but ultimately, the process unfolds as a series of trial and error tests. Suppose an analyst finds that a four-variable specification is the best model. If the underlying data set has 100 variables, then we can infer that the chosen specification is 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 can be seemingly endless, but can be bound if the goals are set in advance. As logistic regression is one of the few truly interpretable machine learning algorithms, the goals can be broadly divided between *interpretability* (focus on $\hat{\beta}$) and *prediction* ($\hat{y}$).

Interpretability. Logistic regression is well-suited for socializing an empirical problem by crafting narratives around the coefficients. In order to tease out the associated effect of a variable, it is customary to estimate a series of models to understand how robust the observed effect size is.

- **Multicollinearity**. In virtually all introductory texts that cover logistic regression, collinearity is consistently flagged and rightfully so. As a refresher, multi-collinearity is a condition in which two or more input variables are not only correlated with the target variables, but amongst each other as well. The consequence is odd behavior among the coefficients – a factor that should be negative signals positive, the size of effects may be extraordinarily large, etc. 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 model's learning process guided by MLE will be challenged in distilling the effects for each variable. 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.
- **"Ill-Posed Problems"**. Like ordinary least squares, logistic regression are not well suited for scenarios where the number of observations n is out-numbered variables k – $k > n$. A more recent solution has been to rely on regularization methods (e.g. LASSO and Ridge) that are designed to efficiently perform in these scenarios.

Sample imbalance. ^^Sample imbalance will directly impact prediction accuracy^^

1.3.2.3 Use Case: Health Care Coverage

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 predict who is likely to need coverage:

$y(Coverage) = f$(Sex, Age, Education, Marital Status, Race, Citizenship)

Based on the Census American Community Survey, we will illustrate how to construct a logistic regression. The sample 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
  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

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

Training a logistic regression can be easily done 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:

```
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. $y = w_0 + w_1 x_1 + w_2 x_2 + \epsilon$ 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 probability distribution family that underlies the specific estimation method. In the case of logistic regression, the probability family is *binomial*.

To start, we will specify three models:

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

then assign each to a formula object and estimate each formula.

```
# Formula objects
  econ <- as.formula("coverage ~ log(agep) + wage + esr")
  soc <- as.formula("coverage ~ cit + mar + schl")
  all <- as.formula("coverage ~ log(agep) + wage + schl + esr + cit + mar")

# Estimated GLM models
  glm_econ <- glm(econ, data = train, family = binomial)
```

```
glm_soc <- glm(soc, data = train, family = binomial)
glm_all <- glm(all, data = train, family = binomial)
```

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. Often times, coefficient tables are examined, in particular 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: Sat, Oct 20, 2018 - 10:43:30

Like the KNN example, the absolute accuracy of a model needs to be obtained through model validation techniques like cross validation. The `boot` library can be used 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:

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

or the true positive rate (TPR):

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

Both are written as functions below:

```
# Misclassification Rate
  costAccuracy <- function(y, y.hat){
    a <- sum((y == 1 ) & (y.hat >= 0.5))
    b <- sum((y == 0 ) & (y.hat < 0.5))
    c <- ((a + b) / length(y))
    return(c)
  }
```

Table 8:

	Dependent variable:		
	coverage		
	(1)	(2)	(3)
log(agep)	-1.415^{***}		-0.785^{***}
	(0.045)		(0.062)
wage	-0.0005^{***}		-0.001^{***}
	(0.0001)		(0.0001)
esrEmployed Civilian	3.842^{***}		3.676^{***}
	(0.720)		(0.723)
esrNot in Labor Force	4.011^{***}		3.630^{***}
	(0.721)		(0.724)
esrUnemployed	5.261^{***}		4.937^{***}
	(0.723)		(0.726)
citNon-citizen		2.132^{***}	2.016^{***}
		(0.083)	(0.084)
marMarried		-0.993^{***}	-1.038^{***}
		(0.059)	(0.061)
marNever Married		0.237^{***}	-0.237^{***}
		(0.060)	(0.071)
marSeparated		0.279^{**}	0.207
		(0.125)	(0.130)
marWidowed		-1.837^{***}	-1.459^{***}
		(0.113)	(0.118)
schlHS Degree		1.712^{***}	1.616^{***}
		(0.102)	(0.103)
schlLess than HS		1.994^{***}	1.937^{***}
		(0.107)	(0.110)
schlUndergraduate Degree		0.648^{***}	0.580^{***}
		(0.114)	(0.115)
Constant	1.166	-1.291^{***}	-1.908^{**}
	(0.734)	(0.109)	(0.765)
Observations	13,596	13,596	13,596
Log Likelihood	$-8,497.670$	$-7,927.152$	$-7,529.608$
Akaike Inf. Crit.	17,007.340	15,872.300	15,087.220

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

```
# True Positive Rate
  costTPR <- function(y, y.hat){
    a <- sum((y == 1 ) & (y.hat >= 0.5))
    b <- sum((y == 1 ) & (y.hat < 0.5))
    return((a) / (a + b))
  }
```

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.

specification	accuracy
Economic	0.6482789
Social	0.6946161
All	0.7217564

In order to obtain the predicted values of *coverage*, 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".

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.

```
  summary(pred.glm.train)
```

```
##       Min.    1st Qu.     Median       Mean    3rd Qu.       Max.
## 0.0000006 0.3015294 0.5117123 0.4988232 0.6880606 0.9892398
```

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 0.7898052
## 2     2 0.7958594
```

1.3.2.4 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?
2. ^^Another question goes here^^

1.3.3 Decision Tree Learning

```
# 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

# 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

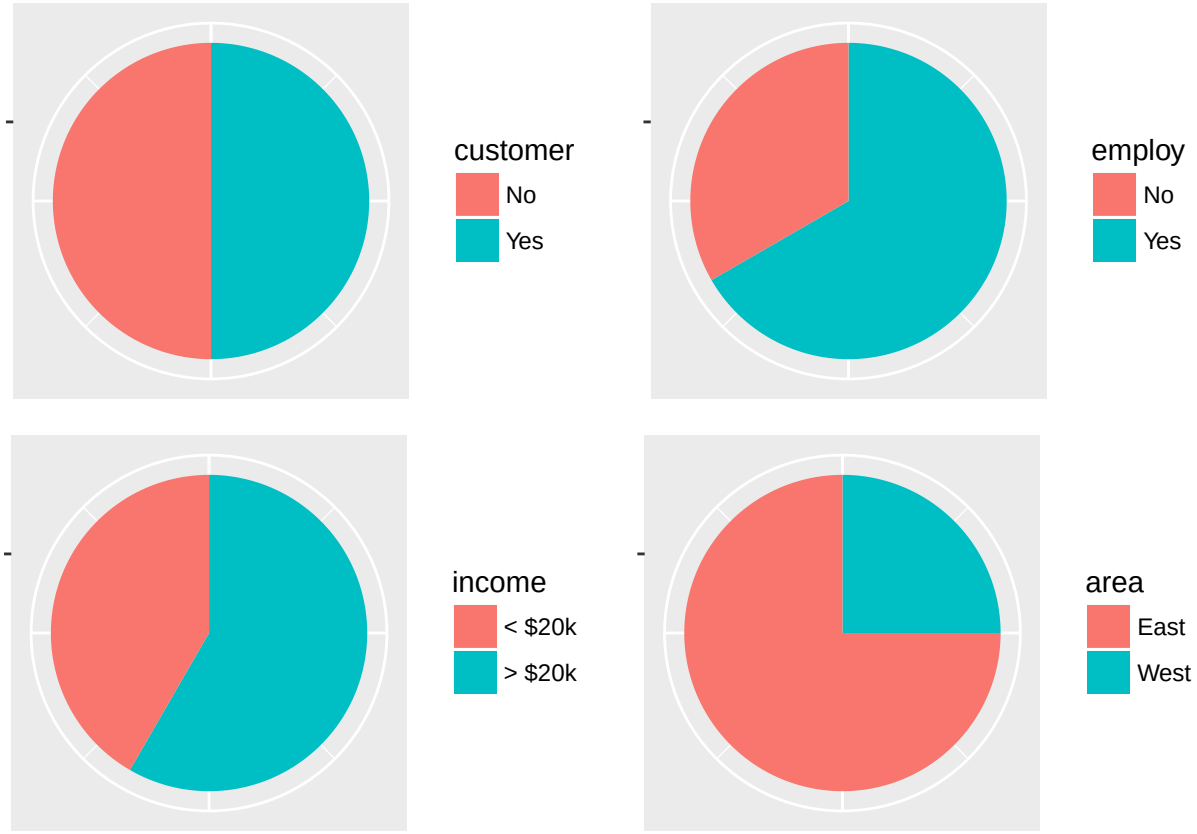


Figure 9: Summary characteristics of town residents.

framework that includes the following elements: (1) nodes and edges, (2) attribute tests, and (3) termination criteria.

1.3.3.1 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?

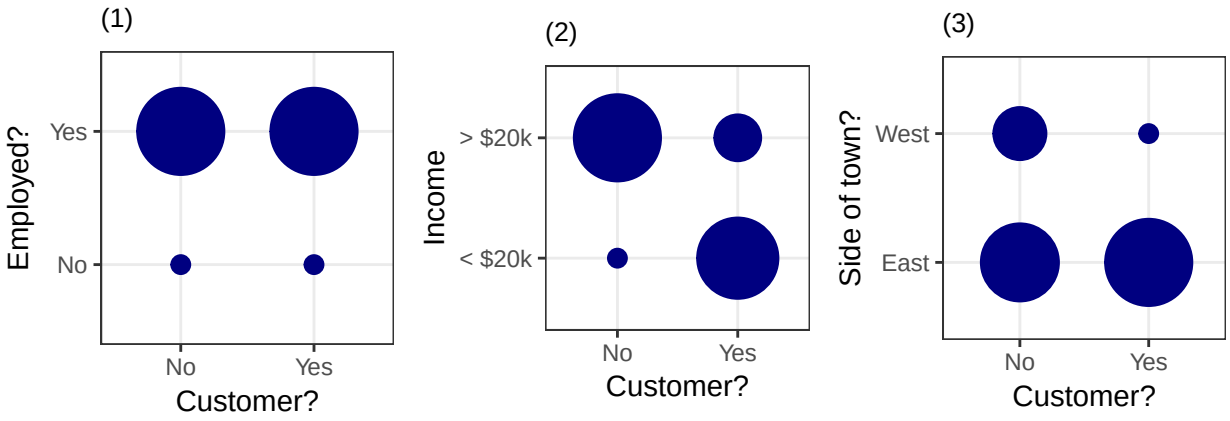


Figure 10: A visual comparison of low separability (1 and 3) and high separability (2).

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:

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

1.3.3.2 (3) Stopping Criteria + Tree Pruning

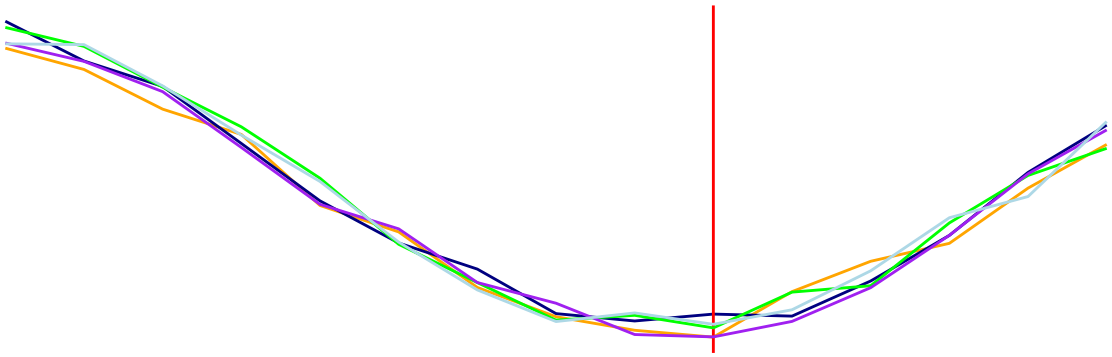
Both Gini Gain and Information Gain attribute tests can be recursively applied until there are no longer input features available to split the data. This is also known as a "fully grown tree" or an "unpruned tree". While the terminal leafs may yield a high degree of accuracy in training, trees may grow 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.



1.3.3.3 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.

1.3.3.4 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[15]. 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)
```

[15] Hastie et. al (2001)

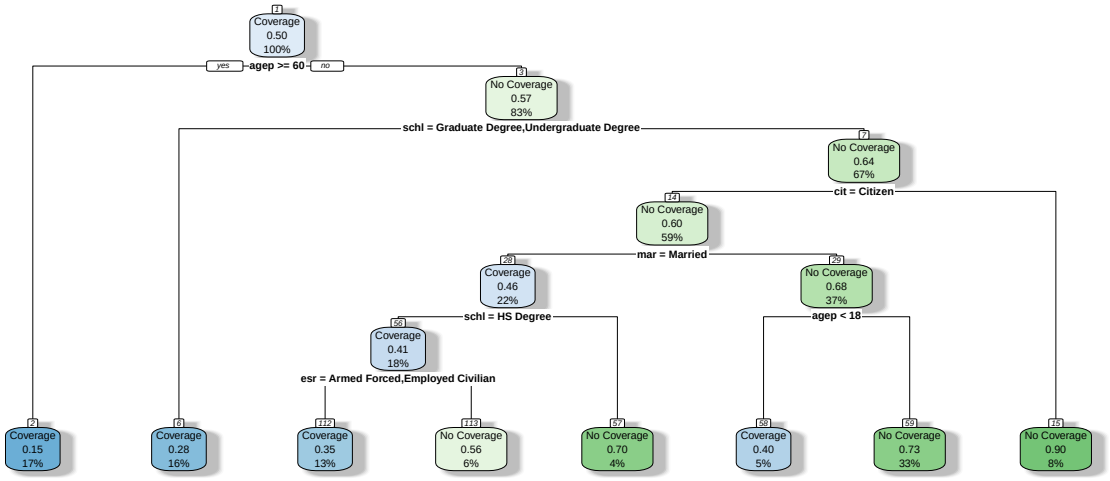


Figure 11: Decision tree using default parameters.

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
```

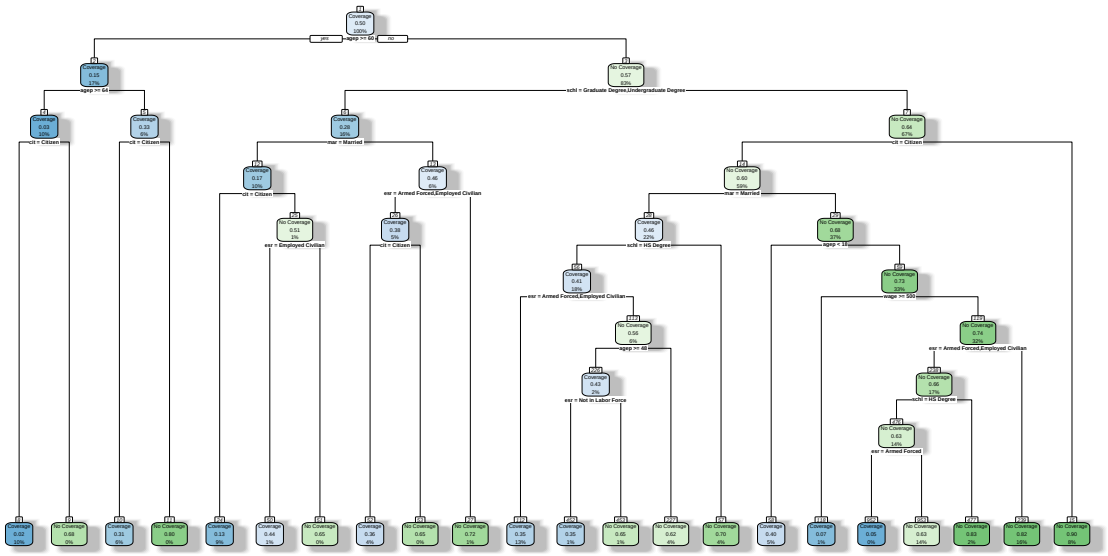


Figure 12: Decision tree for optimized complexity.

```
## 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

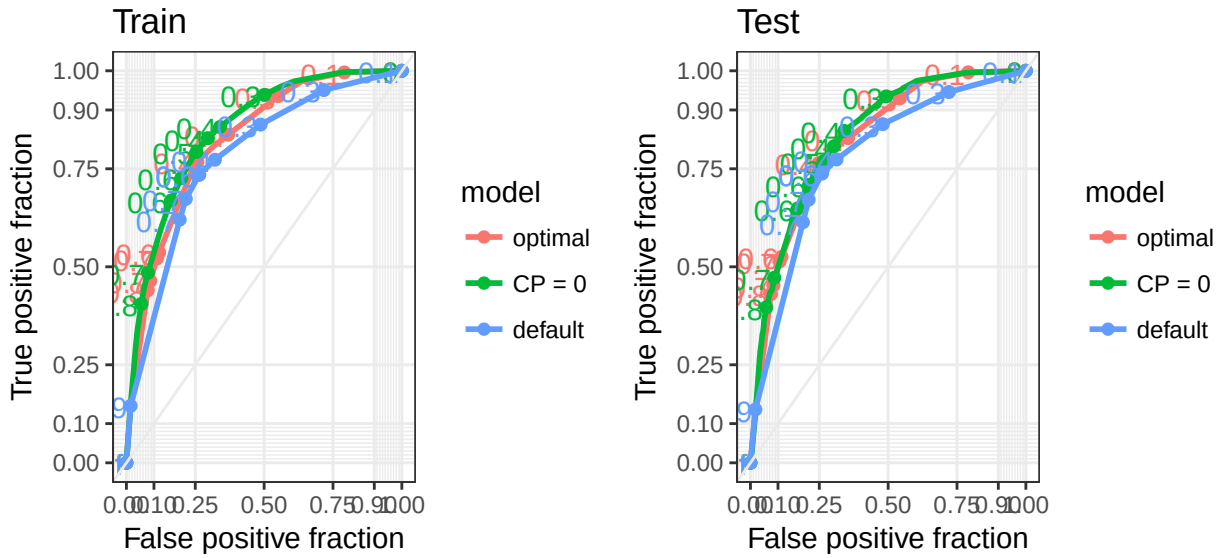


Figure 13: ROC curves for train and test sets.

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

1.3.4 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
```

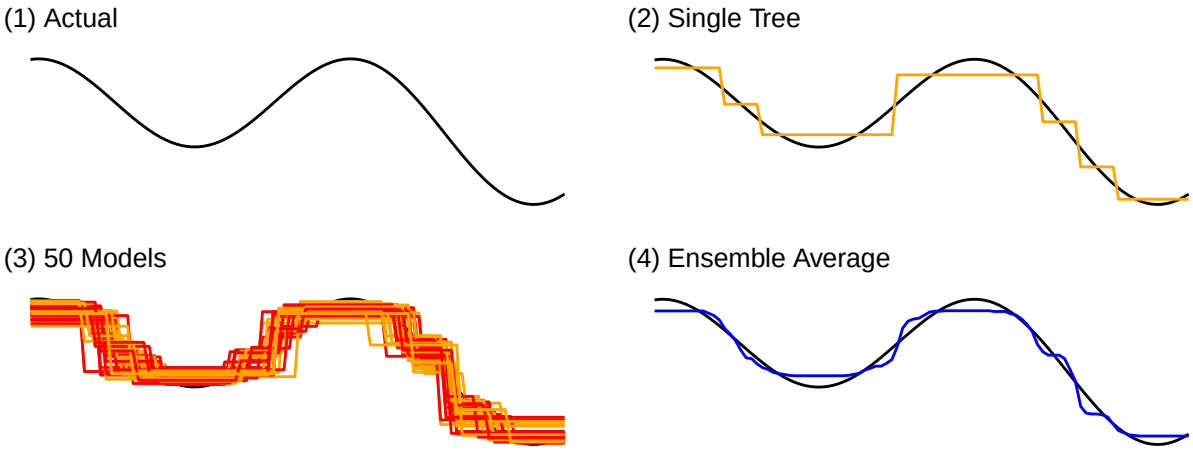


Figure 14: Comparison of results of applying a single model to fit a curve versus an ensemble of models.

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

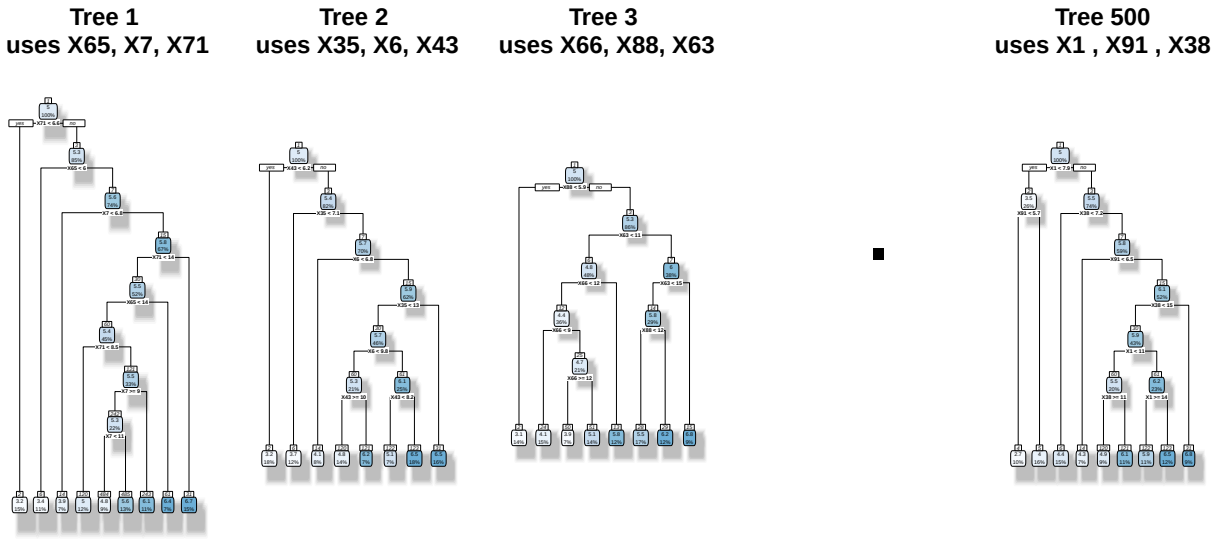


Figure 15: Random Forests construct hundreds of trees sampling from both observations and features, then combine the trees into one prediction through voting.

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.

1.3.4.1 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.

1.3.4.2 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
```

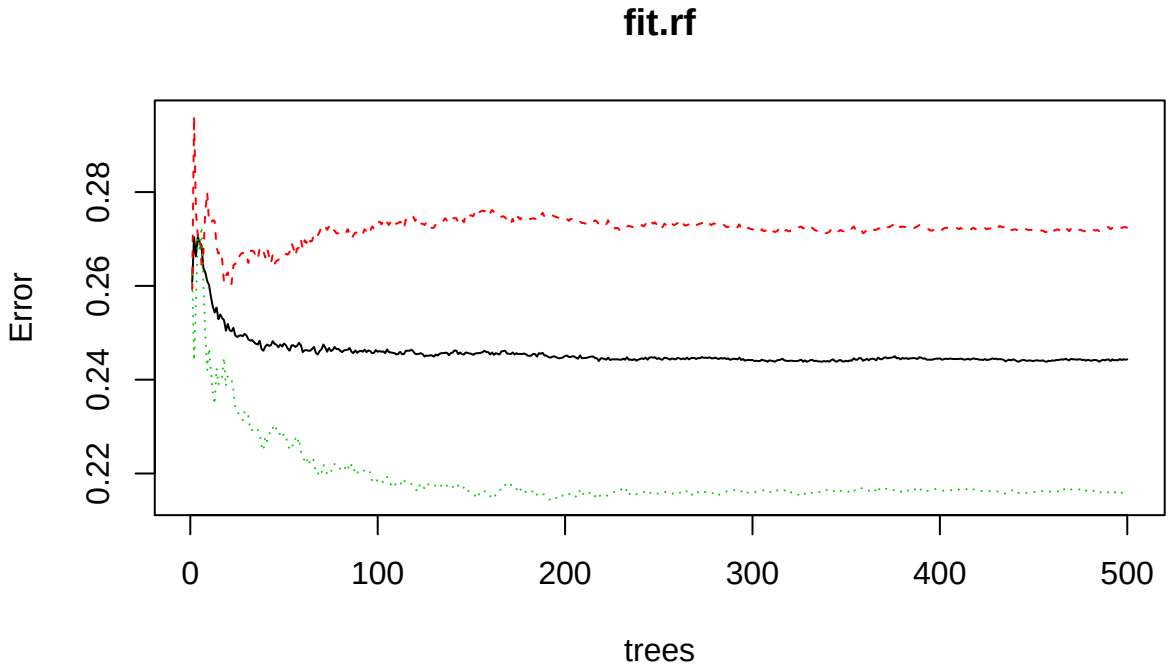
Using the `importance()` method, we can see the Mean Decrease Gini, which calculates the mean of Gini coefficients. `agep` has the largest value of 801.3155193, indicating that age is the best predictor of coverage; However, the values themselves do not have any meaning outside of a comparison with other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

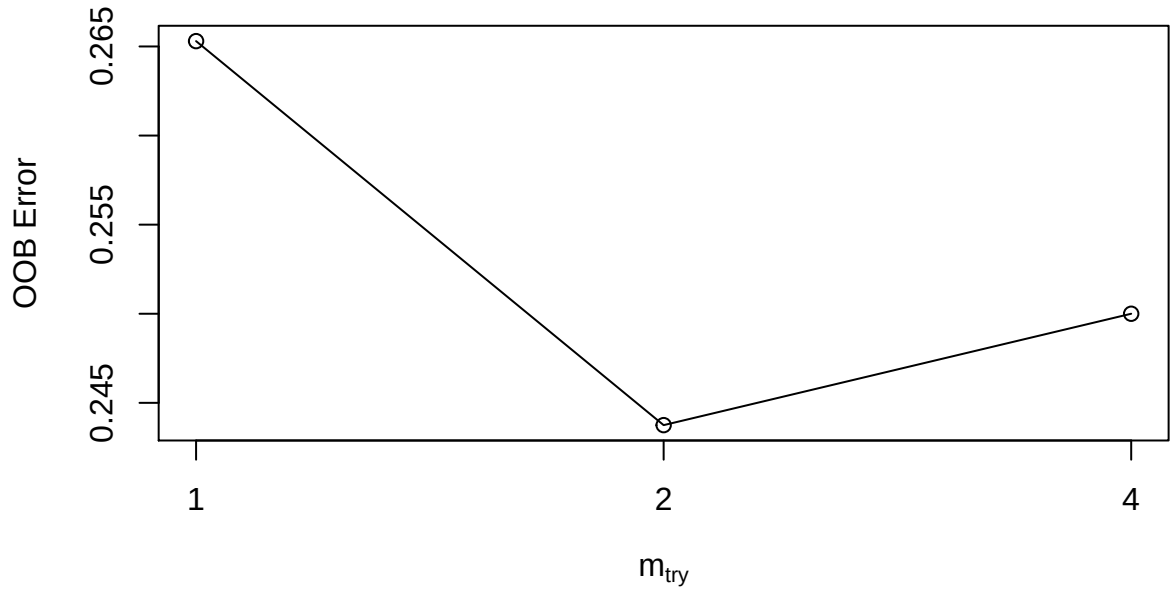


Figure 16: Random Forest tuning result (m = number of features, OOB Error = out of sample error).

```
#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
```