Chapter 1

Prediction Strategies

1.1 Map Data as Guestimates

On an early September day in 2013, the staff of Fairbanks International Airport had an unusual encounter: TSA agents intercepted a motorist headed for the airport. This was not just any routine traffic stop — it was on the tarmac of an airfield. A motorist unwittingly drove passed marked signs and lights, venturing down a taxiway and eventually across an active runway on the way to the airport car park (Cole (2013), Koosner (2013)). Later that same month, yet another motorist made the same mistake. What is the connection? Both incidents were the result of faulty driving directions from an iPhone app. Needless to say, the airport blocked the entrance to the tarmac and filed complaints with Apple.

Incidents like these are not uncommon. In fact, competing apps like Waze have led drivers into danger, whether its impassable snow covered roads (Kraus (2019)) or into the path of a wildfire (Wagstaff (2017)). Part of the problem lies in the maintaining an up-to-date record of the built environment. Imagine that the address of every building, condition and speed limits of every road, status of traffic, among other real-time conditions need to be available for any user to use at a moment's. It is a massive challenge that has led companies to acquire other companies to improve data quality (Trenholm (2013)). Sometimes, companies will even use sensitive user data to fill data gaps (Panzarino (2018)). Keeping up-to-date data is a problem of scale – it is simply too hard for humans to manually curate and keep information up to date for literally all of existence.

Machine learning can help.

Everyday, we already rely on high frequency and high resolution data to make even the smallest of decisions, yet we do so often without realizing that much of the data are filled with approximations produced by predictive models. This is the new paradigm of artificial intelligence that has taken the tech sector by storm and its predictive power has been increasingly seen in the service of social and public good. In recent memory, Microsoft illustrated that machine learning algorithms can perform a mapping task that would normally require years for a team of humans to perform. Computer scientists trained a pair of algorithms to identify building footprints. One set of neural network algorithms were trained on five million satellite imagery tiles to identify pixels that belong to buildings, then an additional filter converted pixels into building polygons. (Microsoft (2018)) In other words, one model examines images that contain buildings and non-buildings and predicts which pixels are likely be part of a building. Since there may be some rogue pixels making the building footprint jagged, a second algorithm converts pixels into polygons that resemble realistic building footprints. The methodology achieved a precision of 99.3% and recall of 93.5% — it is quite accurate and scalable. The algorithm was then set loose on satellite imagery for the entirety of the United States, depositing its findings into the first comprehensive national building database containing 125,192,184 building footprints (Wallace, Watkins, and Schwartz (2018)). A complete inventory of the state of the built environment has never been available at this level of resolution and coverage, and could one day support real-time decision

making. When faced with a hurricane, emergency services could have access to a more up-to-date inventory of all structures in the path of destruction, enabling more accurate damage estimates and more informed mitigation strategies. Local governments can make better zoning decisions so cities can evolve intelligently. And perhaps such a database could support address canvasing for the US census.

What if algorithm can provide help distill a mass of data into a simple, more useful form. The economy, for example, is comprised of an extraordinarily large number of variables. Some are related, others are not. When faced with thousands of economic variables, a human analyst may be biased towards what they are familiar or their world perspective. A machine learning algorithm, in contrast, will seek out variables that have the greatest signal for predicting a measure of interest. This is what the U.S. Bureau of Economic Analysis (BEA) has experimented with in recent memory. The agency, which is responsible for estimating the Gross Domestic Product (GDP), is faced with a constant scheduling tango with their data sources – some data are available in time for their advance estimate of GDP and others are not. When data are not available in time, economically-motivated projections carry the estimates forward until when there is an opportunity to incorporate the "gold copy" data. The risk of projection is the chance that it does not reflect the gold copy when it is available, leading to revisions in economic estimates – a source of anxiety for economists and financial analysts. To reduce revisions to service sector estimates, BEA has developed an experimental approach that relies on *ensembles* of machine learning algorithms, such as *Random Forests* and *Regularized Regression*, to sift through thousands of alternative economic variables and predict economic growth before data is available. This strategy has been shown that predictions can reduce revisions to economic estimates by billions of dollars (Chen et al. (Forthcoming)), which in turn can mitigate unnecessary market responses to revisions. Similar prediction strategies have been applied by tech companies to optimize their resources. Uber, for example, trained a Long-Short Term Memory (LSTM) algorithm to forecast ridership when faced with extreme events such as sports events and holidays. By improving their short-range forecasts across their platform, they can better optimize resource allocation to meet customer demand and manage budgets more efficiently (Laptev, Smyl, and Santhosh Shanmugam (Forthcoming)).

The pursuit of prediction has driven computer scientists and statisticians to constantly develop new strategies that maximize predictive accuracy at scale. [Machine learning algorithms sit within a data science pipeline]

[Image]

In fact, each type of machine learning algorithm excels under different conditions. In Figure 1.1, we plot three distinct scenarios for a two-class classification problem: simple linear boundary, non-linear, and discontinuous. A *logistic regression* is a natural fit for a linear boundary given its root in linear regression. While it is the champion of parameter estimation, logistic regression is ill-fit for more complex relationships. Non-parametric techniques are far more flexible and can mold the decision boundary to the counters of the data. This gain in accuracy comes at the cost of interpretability. For example, *K-nearest neighbors* (kNN) performs classification by looking at neighborhoods of observations. Given a training sample, each record in the test sample is predicted using the most common label for the k closest known records. In essence, kNN is driven by a majority or plurality vote from neighboring records. Alternatively, *decision tree learning* such as *Classification and Regression Trees* (CART) and *Random Forest* algorithms assorb the patterns that it has learned by encoding as binary rules that split a sample into finer, more homogeneous partitions. Each algorithm has its strengths and weaknesses rooted in how it deals with and assimilates information.

You will no doubt have noticed that the algorithms mentioned sound quite exotic when compared with the plain vanilla regressions in Chapters 8 and 9. There are hundreds of machine learning algorithms that are commonplace in the modern data science workbench, many of which are well-suited for both classification and regression problems. In this chapter, we explore a number of these methods, illustrating their basic properties and their potential role in helping public and social missions.

1.2 K-Nearest Neighbors (KNN)

K-nearest neighbors (KNN) is a non-parametric algorithm built on a simple idea: *observations that are closer together are likely to be similar*. Promixity of points matters as the disposition or label of each y_i is

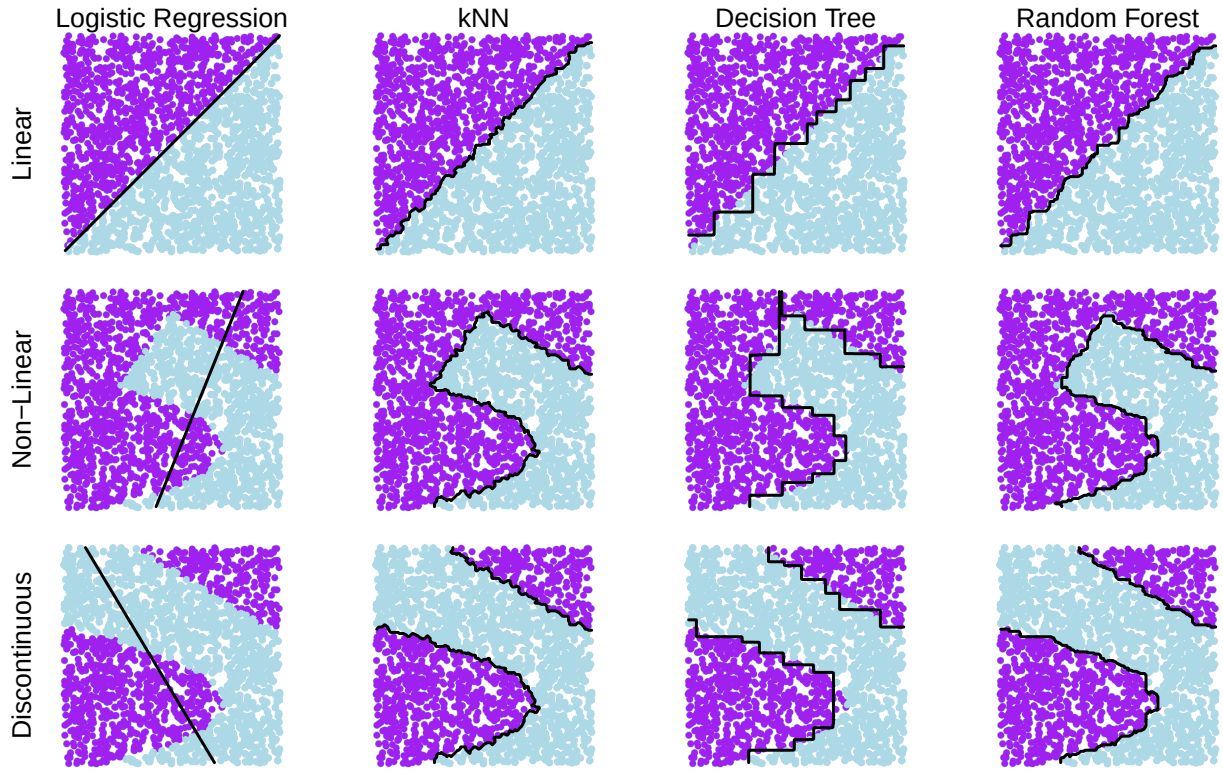


Figure 1.1: Linear, Non-Linear and Discontinuous Classification Problems.

inferred from its surrounding neighbors. These neighbors are identified by treating each input variables x_k as coordinate, which in turn allows distance to be calculated between all points in the data set. KNN is like the spackle of data – it can fill holes while retaining the shape of the surrounding surface to which it is applied. It is this maliable quality that makes KNN an ideal technique for impute missing values in a realistic, organic fashion.

1.2.1 Under the hood

KNNs are very much like being on a road trip in the pre-GPS and pre-smartphone era, then stopping in a town for lunch. To find out where you are, you would typically ask the first k-number of people you see (usually just $k = 1$) what town you current are in. When KNN produces predictions, it first locates where a new point falls within the coordinate space of the training data, then constructs a prediction from k-nearest neighboring points in the training data. Each prediction $\hat{y}_i$ relies on training targets and inputs $\{(x_n, y_n)\}_{n=1}^{N}$ that serve as landmarks to triangulate where new points should fall – like geolocation on a map. If the base map is reliable and densely populated, then there is enough information to make a precise guess for new points.

As illustrated in Figure 1.2, the KNN algorithm is quite simple:

1. Set k. Prior to starting the algorithm, start by setting a value of k that determines the number of surrounding points around a record i that will be used to produce a prediction for y_i. The true value of k is not known – this will require some systematic trial and error.

For each record i in the test sample:

2. Calculate distance. The collection of input variables X serve as coordinates. To identify the neighborhood around each i in the test sample, calculate the distance d_{ij} from i to each point j in the training sample.

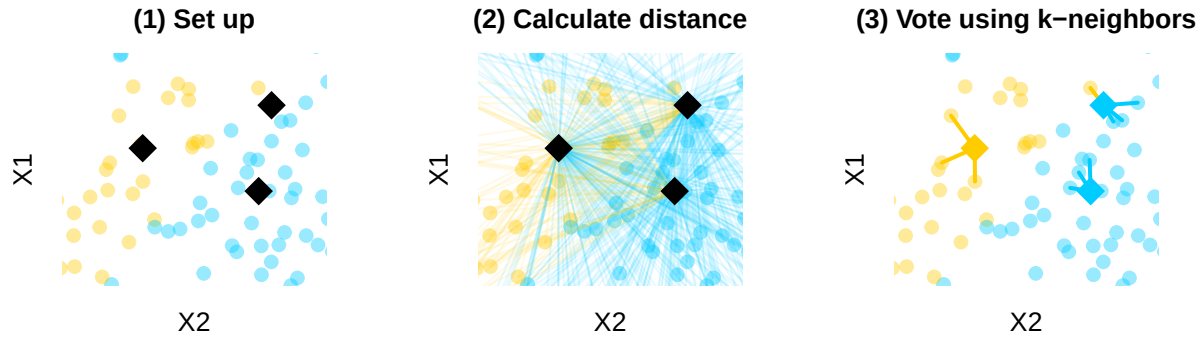


Figure 1.2: KNN algorithm process.

Distance most commonly takes the form of Euclidean distance ($\sqrt{\sum_{i=1}^{n}(x_i - x_0)^2}$), which is appropriate with continuous values. For cases where the underlying data are discrete or binary, Manhattan distance ($\sqrt{\sum_{i=1}^{n}|x_i - x_0|}$) is more appropriate.

3. Vote!. The prediction $\hat{y}_i$ is calculated from a vote of the k nearest points to point i. In classification problems, the proportion p_c of points that are in each class c in Y. The proportion p_c is converted into a predicted class through *majority voting* – assign an observation to the class that is most represented in the neighborhood. In regression contexts, $\hat{y}_i$ is simply the average of y_i for the k-nearest neighors.

There are various flavors of voting as shown in Table 1.1. The procedure illustrated give all k neighbors equal weight, but if its believed that closer points should be given more weight, then a variety of *kernels* such as inverse distance and biweights discount the importance of points that are farther away even if they are part of k.

Table 1.1: Types of kernels used in voting.

Kernel	Formula	Interpretation
Biweight	For $\|d\| < m$, kernel weight is $d(1 - \frac{d^2}{m^2})^2$. Otherwise, 0 when $\|d\| > m$. The value $m = d$ for the k^{th} neighbor.	
Rectangular	$Pr(Y = j) = \frac{1}{k}\sum_{i=1}^{k} I(y_i = c)$	Calculate the proportion of j based on k nearest neighbors. This is the same of simple arithmetic mean.
Inverse	$Pr(Y = j) = \sum_{i=1}^{k} w(d)(y_i = j)$ where $w(d) = \frac{1}{d_i \sum_{i=1}^{k}(\frac{1}{d_i})}$	Calculate the weighted proportion of j based on the inverse distance to k nearest neighbors.
Gaussian		

4. Tune k. When $k = 10$ with a rectangular kernel, the conditional probability for y_i reflects the 10-nearest neighbors. When $k = n$, the probability is equivalent to the sample mean. These two extremes illustrate that KNN is sensitive to the value of k and the kernel, yet we do not truly know which is right. Tuning is a necessity to test different scenarios in order to optimize accuracy.

1.2.2 In Practice

Tuning. Like many other algorithms, KNNs require tuning of *hyperparameters* in order to maximize predictive accuracy. We do not know what is the true value of k or the absolute best kernel to use, thus tuning of the hyperparameters is typically a grid search. The idea is to develop a ballpark sense of what works, then hone

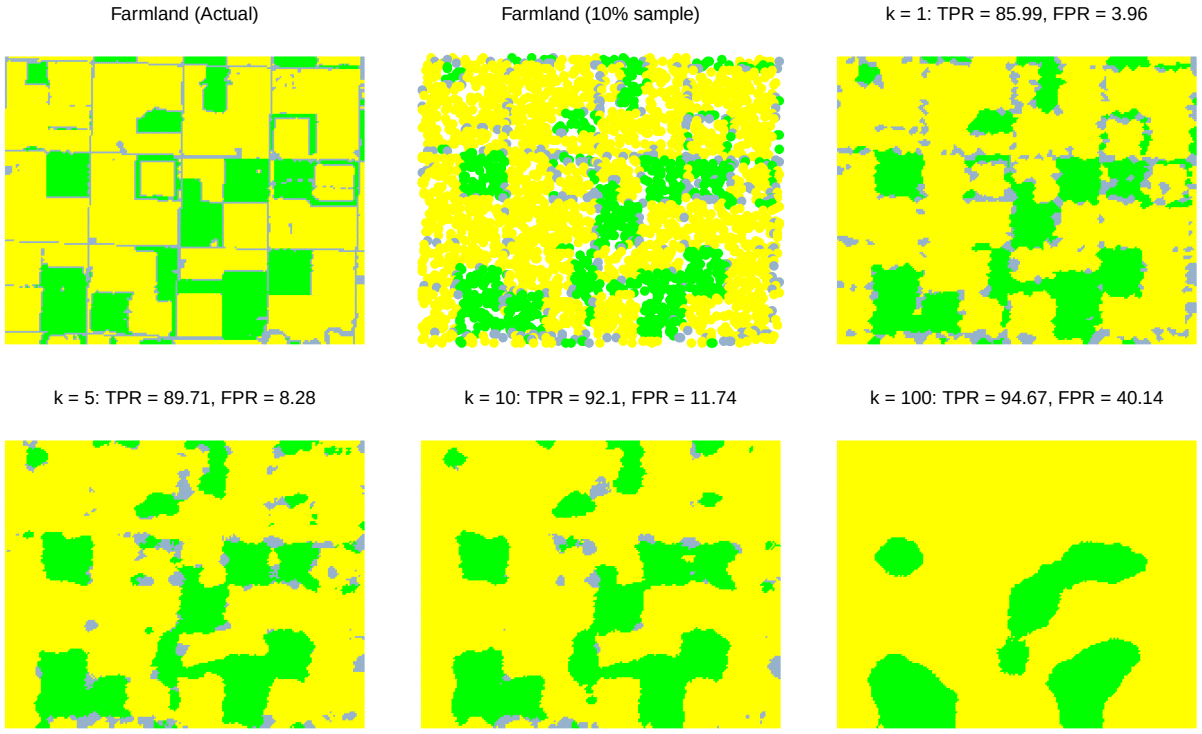


Figure 1.3: Comparison of prediction accuracies for various values of k.

in on the best value of k. One search strategy involves testing all values from $k = 1$ to $k = \sqrt{n}$ in multiples of one's choosing, keeping track of how each k performs in terms of a loss function (e.g. TPR, FPR, F1-statistic).

To illustrate this process in Figure 1.3, we have assembled a sample of USDA CropScape landcover data (W. Han (n.d.)) for farmland that grows corn (yellow) and soybeans (green). More often than not, data will having missing values and capture only a fraction of the full picture. In our example below, we simulate this missingness by retaining a random sample of 10% of the land cover data. We naturally would like to know what is in the remaining 90%. KNNs are an excellent choice for imputing data with a spatial component, but they depend heavily on the choice of k. In the imputations below, we compare values of k at 1, 5, 10, and 100. It is apparent that as the value of k increases, the cornfields increasingly creep into areas where soy would be expected. Not only is this loss in accuracy reflected visually, but statistically as well.

Normalization. Treating variables as coordinates implies that they all should have equal importance – no single variable should weigh on the distance calculation more than any other. Of course, there are flavors of KNN where variables are weighted, but in the plain vanilla case, the scale of each variable should be normalized, at least for continuous variables. For example, if an input variable x_1 has a scale from 1 to 10,000 and x_2 ranges from 0.1 to 0.3, a KNN will rely more heavily on the latter variable. Normalization can be as simple as calculating the z-score for each x_k:

$$scaled = \frac{x_k - \mu_{x_k}}{\sigma_{x_k}}$$

where the transformed variable is mean centered with unit variance.

There are, of course, cases when certain variables are more naturally suited for KNNs. They are particular useful for gridded data – information recorded on equal intervals, including time series, sound, and imagery. But when the data contain mixed formats like categorical variables and integers, data should be transformed into the comparable units. Discrete variables can be converted into a dummy variable matrix while continuous

variables can be binned into an ordinal scale. Manhattan distance would be a more suitable measure to relate records.

Speed and Importance. KNNs are best used when data sets are smaller with fewer variables as each distance calculation is computationally taxing. We can see that for a pair of training and test data sets, distance will need to be calculated $n_{train} \times n_{test}$ times, which can become more costly as the number of variables increases. Thus, KNN is a practical solution when the data are smaller. Furthermore, as more variables are added, the importance of any one variable is diluted. In situations where only a few variables contain meaningful information, the equal weighting assumption can introduce more noise than signal, relying on neighborhoods that might actually be far away.

Nonetheless, the simplicity of this approach makes it a popular choice for fill gaps in data.

Table 1.2: The pros and cons of KNN.

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

1.2.3 DIY: Anticipating the extent of damage from a storm

Hurricane Sandy, for example, had a tremendous effect on New York City. Perhaps one of the most visible impact was the number of downed 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.

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 help, we know where downed trees have and have not been observed based on the nature of the 311 call. As only a fraction of neighborhoods may call on a given day, we can use the incoming data stream to update an imputed picture of the state of downed trees. With more complete information, city operations can efficiently and systematically triage the need for help.

In this DIY, we examine the case of downed trees, using available 311 calls from the day after Hurricane Sandy to predict whether downed trees are reported in the proceeding seven days. The predicted probabilities can inform emergency response triage, guiding how areas in need are prioritized to help.

Prepare the data. The data are comprised of a pair of coordinates recorded in *state plane feet*, a grid cell `id`, a borough label `boro`, and a set of binary variables indicating if a downed tree was reported on the day of Hurricane Sandy (`tree.sandy`) or in the following seven days (`tree.next7`).

The two binary variables were crafted under the assumptions that residents of a neighborhood would call to report downed trees if they are observed. If a call were made, then we could flag a neighborhood as one that experienced tree troubles. Alternatively, if complaints from a neighborhood are devoid of tree-related issues, we can assume that the area was unaffected. These assumptions were applied to NYC 311 call data, defining sub-neighborhoods as $1000ft \times 1000ft$ grid cells – $n = 7513$ cells in all.

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

For simplicity, we focus on the two largest and populous boroughs (`boro`), Brooklyn (BK) and Queens (QN), that share the same land mass and have similar geographic and geological characteristics. During the period of analysis, service requests were logged in a total of $n = 4477$ grid cells (59.6% of NYC) in Brooklyn and

Queens. Of these, we only have information for 43.5% of grid cells at the passing of the hurricane, thus over half its not clear what has happened in the remaining 56.5% of the two boroughs. On the one hand, we can wait and see what has happened, or otherwise deploy a predictive strategy to paint a likely picture of the state of affairs.

```
#Extract Queens and Brooklyn
  pacman::p_load(dplyr)
  nyc <- filter(nyc, boro %in% c("BK", "QN"))
```

Train. Our predictive approach is built on the assumption that data immediately available after the storm (the target variable `tree.sandy`) contain enough information to make an informed guess about the status of the entire study area. The training sample (`train`) is constructed for where the variable `tree.sandy` is complete. A quick tabulation shows that 79.7% ($n = 1550$) of grid cells in the training set have at least one downed tree reported – evidence that the storm had widespread impacts. The test set (`test`) is a collection of every grid cells in all of Brooklyn and Queens. This is simply a short-hand way of asking the kNN algorithm to produce a complete picture of what it can gleen from the data.

```
#Extract the training and test samples
  train <- subset(nyc, !is.na(tree.sandy),
                  select = c("ycoord", "xcoord", "tree.sandy"))
  test <- subset(nyc,
                 select = c("ycoord", "xcoord", "tree.next7"))

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

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

With the data ready, we apply KNN to predict downed trees. While it is generally good practice to write algorithms from scratch at least once to fully understand its underpinnings, it is hard to ignore the efficiency gains made possible through the `kknn` library. The `class` library, which is the typical R library used for KNNs, is fit for purpose, but is limited to only classification problems and cross validation of the value of k. The `kknn` library provides additional flexibility to tune the type of kernel and consider not only classification problems, but continuous and ordinal targets as well. The library is furnished with a pair of functions to help tune a KNN and score new data sets.

```
  pacman::p_load(kknn)
```

The `train.kknn` function implements k-folds cross validation to find the optimal set of hyperparameters. The function expects a few key inputs:

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

- `formula` is a formula object (e.g. "`no.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.

For a first pass, we conduct 20-folds cross validation, searching between $k = 1$ and $k = 100$ neighbors in combination with three kernels (rectangular, inverse, and biweight). This simple command does much of the hard work, running the KNN algorithm 6000 times (20 cross-validation models for each k and *kernel*

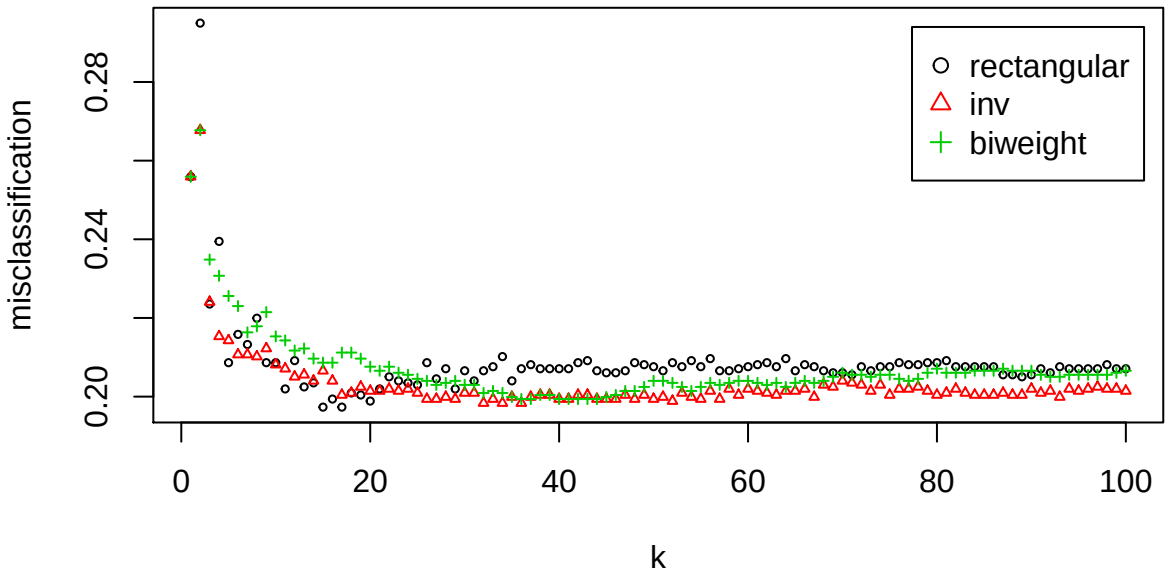


Figure 1.4: 20-fold cross validated errors for k = 1 to k = 100

combination), then identifies the parameters that minimize classification error. The results are stored in the `fit_cv` object.

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

#Run with 20-folds cross validation
  fit_cv <- train.kknn(factor(tree.sandy) ~ ycoord + xcoord ,
                       data = train,
                       kcv = 20,
                       distance = 1, kmax = 100,
                       kernel = c("rectangular", "inv", "biweight"))
```

What does the optimization process look like? The `fit_cv` object contains diagnostics that are pertinent in identifying the best KNN model. By plotting the results (see Figure 1.4), it is apparent that all kernels have comparable performance with error converging to a steady level beyond $k = 20$. The optimal conditions, as stored in the `best.parameters` element, show that predictions are optimized when $k = $ 15 using a rectangular kernel achieving a misclassification rate of under 20%.

```
   plot(fit_cv, cex = 0.5, cex.lab = 1)
```

By feeding the optimal conditions back to the `kknn` function, the algorithm can score and piece together a fuller picture of the extent of tree damage. The syntax is quite similar to the training function with some modifications:

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

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

Notice that both train and test are required by `kknn`. This is a consequence of KNN's instance-based learning – it does not store relationships but rather applies a set of algorithmic logic to a test sample.

```
#Retrieve best parameters
   best <- fit_cv$best.parameters

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

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

Evaluate performance. *Is a 20% misclassification error in the training sample a good result? Are KNNs effective at anticipating where tree-related calls will originate for the seven days after the hurricane?* The habitual thing to do is to calculate some loss and accuracy statistic to summarize out-of-sample prediction performance, such as the Area Under the Curve (AUC) of the ROC Curve for assessing the robustness of a classification predictions. Rather than writing a new script to calculate the statistic, we can re-use the `costAccuracy` function constructed in the previous chapter. With an out-of-sample `AUC` =0.66, this first attempt with KNNs would have had some operational value, offering greater clarity during a chaotic time. However, a single AUC value does not reveal how useful the predictions would have been for triaging downed tree requests.

```
#Load libraries
  pacman::p_load(ROCR)

#Calculate ROC
  costAccuracy(test$tree.next7, test$prob)
```

A closer visual inspection of the predictions in Figure 1.5 shows that the KNNs do in fact mold to the surface that it is trained to predict. Below, we compare the KNNs predictions produced using information that was available right as the storm passed versus the status of all grid cells collected over the seven days following the storm. The algorithm appears to be relatively effective in detecting both the unaffected (blue) and affected (red). Predictions are distributed across a gradient when the evidence is less clear, indicating that there is value in using the rankings for prioritizing aid.

When applied in a prioritization context, visits to higher probability areas will likely yield higher hit rates (e.g. finding downed trees when an inspection is conducted) – nearly 80% at its peak as seen in 1.6. Approximately 40% of impacted grid cells have predicted probabilities north of 90%, suggesting that some areas were quite obviously impacted and the data contained enough signal to flag these areas.

Despite the promising result, we should be cognizant that KNNs are among the simplest of prediction algorithms. They are great with relatively little data. We naturally should ask: *Is there a better classifier?*

1.3 Decision Tree Learning

Decision tree learning is not exactly what a typical social scientist would want in an estimation method (e.g. well-defined parametric qualities, *ceteris paribus* interpretation), but they should not be dismissed. They offer the predictive flexibility of non-parametric approaches while enable interpretation of variable importance that is useful for communicating insights. Furthermore, decision tree learning is the basis of among the most widely used and influential machine learning models such as Random Forests and Gradient Boosting.

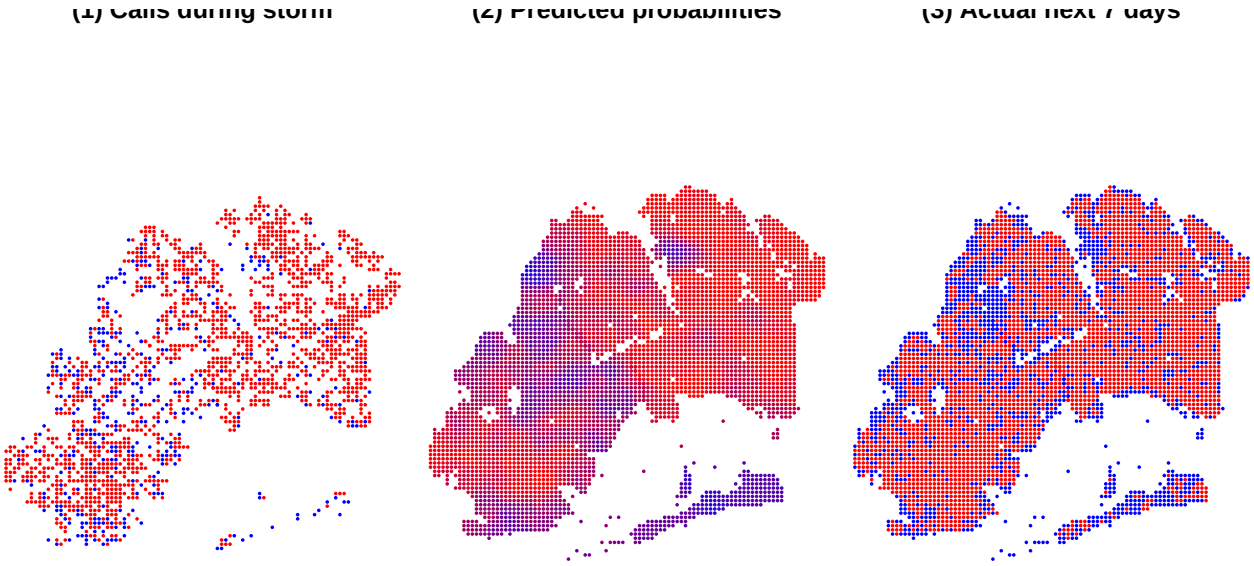


Figure 1.5: Comparison of actual and predicted areas with reported downed trees. Red indicates at least one tree was reported in a given 1000 x 1000 square-mile area

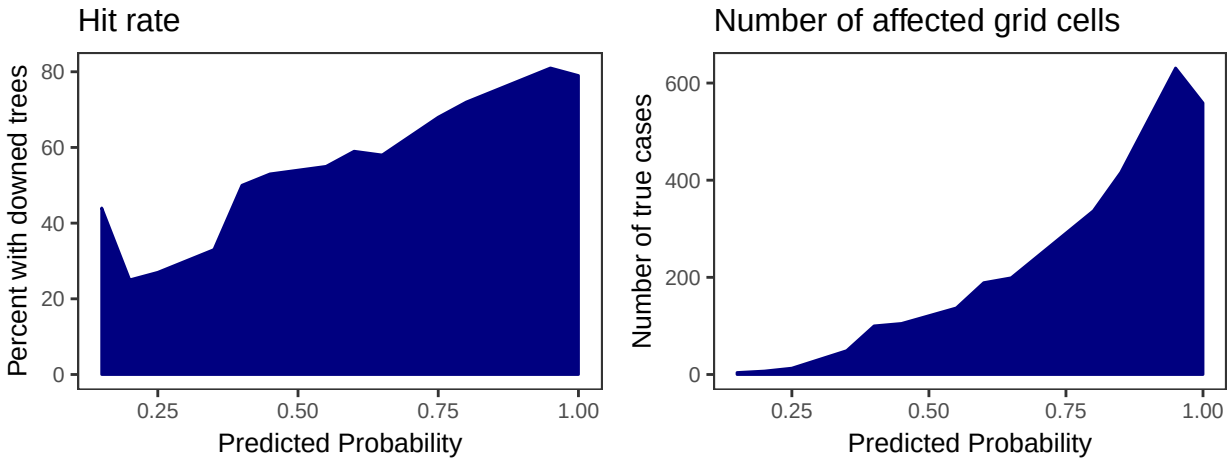


Figure 1.6: Hit rates by predictive probability.

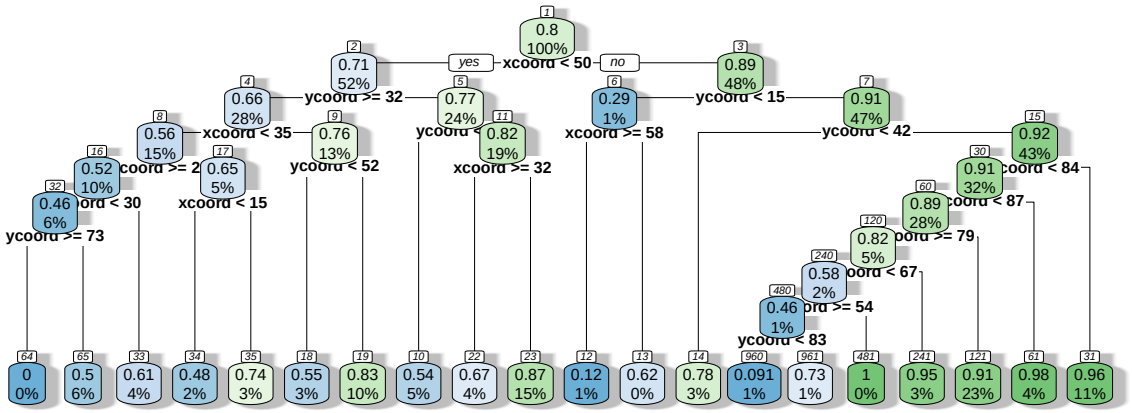


Figure 1.7: A grown decision tree.

How does tree learning work? Recall the hurricane example. The objective is to distinguish and predict which areas have downed trees and which do not. Affected areas tend to be located in pockets of activity. Tree learning brings structure to finding affected areas by analyzing all available input variables to find thresholds at which the sample is partitioned. Each split yields two smaller, more homogeneous partitions so that affected areas are approximately separated from unaffected areas. By splitting each resulting partition into even smaller subsamples, we can more cleanly separate affected areas from unaffected areas. In effect, the procedure draws rectagular and polygonal boundaries that classify patterns on the ground.

Similar to Figure 1.7, the model's structure and learned patterns can be depicted as an inverted tree. Each decision tree is comprised of nodes and edges. Nodes contain records. Edges show dependency between nodes and is the result of an *attribute test* – or an optimal criterion that finds that best split to subset records into more homogeneous partitions. The node at the top of the tree is known as the *root* and represents the full population. In our example, 80% of the root contains downed trees. Each time a node is split, the result is two nodes – each of which is referred to as a *child node* that contains a subsample. In our hurricane example, the first split is identified at the threshold ($xcoord < 50$) that can best separate affected from unaffected areas, doing so imperfectly but yielding an improvement over the root node: 71% of the child node below $x < 50$ is comprised of downed trees compared with 89% in the other child node. Each successive split concentrates downed tree areas into some nodes and unaffected areas into others. As the tree continues to partition into smaller child nodes, eventually nodes cannot be split any further. Each node that is without child nodes is referred to a *leaf* – it is the terminal node.

Because the tree is built on a series of binary criteria, each leaf is defined as a list of if-statements. The leaf at the bottom right of Figure 1.7, for example, is identified as any area (or record) that is located $ycoord \geq 42$ and $xcoord \geq 84$. During training, this leaf was determined based on the patterns in 11% of the sample. As 96% of records in the leaf contained downed trees, any other area that meets the same criteria is assumed to have 96% probability of having a downed tree as it is observably no different than training examples with the characteristics.

How does a decision tree arrive at this conclusion? Among the varieties of decision tree learning methods that have been developed over the years, the most common of which is Classification and Regression Trees (CART) as described in L. Breiman et al. (1984). In the next section, we explore how decision trees are grown.

1.3.1 Under the hood

Growing trees. Each split is a part of an incremental strategy to distill the data into homogeneous partitions. At each split, there are seemingly infinite number of potential variable-threshold combinations as all input variables are considered. But not all variables hold useful signal. *How does tree learning decide on which*

variables to use? Drawing from *information theory*, we can apply *information gain* to evaluate all candidate splits and select the threshold that gives the *most* information. The search for the best threshold is known as an *attribute test*. Tree learning can be described as a three step process:

1. *Base Cases*. The process starts with checking for "base cases" at the root node, the idea being that it might not be worth exerting effort to grow the tree if the data do not support it. The algorithm will first check to see if (a) all values of the target are of one class, and (b) none of the input variables offer any useful information. There are other base cases to consider depending on the algorithm, but any true base case will result in stopping the algorithm and returning only the root node.

2. *Recursive Partitioning*. If none of the base cases are true, the algorithm proceeds with searching for a threshold along an input variable that splits the data into two smaller samples. With respect to the target, each partition should be more homogeneous in the composition of discrete classes or lower variance in the case of a continuous target. *How is the threshold determined?* Given a large number of inputs, there is likewise a large number of candidate splits. The optimal split is selected by an *attribute test* to identify the split that maximizes *information gain*, or a split that gives the most information, increased homogeneity or reduced variance. Attribute tests are applied to each resulting partition, thus the sample is *recursively* diced into smaller concentrated subsamples.

3. *Stopping Criteria versus Pruning*. At some point, the algorithm will need to stop. The question is *when*? One approach allows the tree to grow until each leaf node contains $n = 1$, in which case we run the risk of an overgrown, overfit tree. Furthermore, like a real tree, a fully grown tree may take some time to grow when the data are large. An alternative approach employs *stopping criteria*. The tree partitioning is terminated if a leaf has fewer records than a pre-specified threshold (e.g. do not split if $n < 5$), the information gain can no longer improve beyond a pre-specified level, among other criteria. And while these stopping criteria may sound reasonable, premature termination of the tree growing can prevent the tree from reaching its full potential – it can bias the predictions. The happy medium involves grows a tree to its fullest, employing cross-validation to identify the optimal degree of tree complexity, then *pruning* the tree to the optimal level. The pruning process will be demonstrated in the DIY example.

Attribute tests. The engine of the CART algorithm is information gain – it is the statistical heart that enable tree learning to make meaningful predictions. Simply put, information Gain (IG) is a comparative measure that estimates the information content laerned (referred to as Impurity I) from splitting a parent node (D_{parent}) into a pair of child nodes D_j, taking the weighted contribution of each child node j.

$$IG = I(D_{\text{parent}}) - \sum_{j=1}^{J} \frac{n_j}{N} I(D_j)$$

The impurity measure is critical – it gauges the amount of information content contained in a pair of partitions. It can also determine if one candidate split is more meaningful than an alternative. Impurity is typically measured using Gini Impurity, Entropy or Variance Reduction (see Table 1.3). The first two measures are utilized for classification problems, calculated on the proportion of records that belong to each of the J target classes. Both have similar performance, though Gini Impurity is slightly easier to implement. Variance is a natural choice for continuous targets, designed for prioritizing splits that minimize within-partition variance.

Table 1.3: Common measures used for decision tree learning attribute tests.

Measure	Formula	Description
Gini Impurity	Gini Impurity $= \sum_{i=1}^{k} p_i(1-p_i) = 1 - \sum_{i=1}^{k} p_i^2$	Commonly use in CART. The Gini Impurity ranges from 0 to $1 - \frac{1}{k}$.
Entropy	Entropy $= \sum_{i=1}^{k} -p_i log_2(p_i)$	Relied on in other decision tree learning algorithms. Values range from 0 to 1

Measure	Formula	Description
Variance	$Variance = \frac{\sum_{i=1}^{n}(y_i - \bar{y})^2}{n}$	Focus on reducing the spread in the target within partitions.

To illustrate the power of impurity metrics, consider the case of predicting which neighborhoods have downed trees, which emergency crews can use to prioritize resources. To simplify the problem, let's suppose that a single latitudinal boundary line can adequately separate affected and unaffected areas. *Which degree of latitude do we choose?*

Table 1.4 presents two candidate boundaries. Split A is somewhere north and Split B is farther north (North+). As quantitative thinkers, we naturally will attempt to make sense of the numbers to contextualize and weigh the pros and cons. Perhaps the distinguishing factor between the two options is the positive rate or maybe there is a ratio that can be calculated that elegantly illustrates the tradeoffs. In a tree learning setting, metrics like Gini Impurity are an effective comparison strategy.

Table 1.4: Two sets of candidate splits.

Label	A: North	A: Remainder	B: North+	B: Remainder
Downed Trees	10	3	9	4
Untouched	5	21	0	26

To evaluate which split is better, we calculate the *Gini Gain*, which is information gain (IG) when applying Gini Impurity (I). The IG value is calculated for each proposed split. Both calculations require the impurity of the root node ($I(D_{\text{region}})$) to capture the baseline information content in the data, then the weighted impurity of the pair of candidate child nodes are subtracted to obtain the Gini Gain. Ideally, the child nodes would have values close to zero that indicate the presence of an overwhelming amount of signal, which in turn would mean a larger IG value is desirable.

$$IG_A = I(D_{\text{region}}) - \frac{n_{\text{A,north}}}{N} I(D_{\text{north}}) - \frac{n_{\text{A,remainder}}}{N} I(D_{\text{A,remainder}})$$

$$IG_B = I(D_{\text{region}}) - \frac{n_{\text{B,north+}}}{N} I(D_{\text{B,north+}}) - \frac{n_{\text{B,remainder}}}{N} I(D_{\text{B,remainder}})$$

As we step through each element of the calculation in Table 1.5, it becomes apparent that Split B can more effectively isolate areas with downed trees, achieving an 100% hit rate in the *North+* partition. Granted, the estimated hit rate is on a small sample, nonetheless is a marked improvement over Split A's 66% hit rate in the *North* partition. Indeed, the IG calculation confirms that Split B would yield twice as much information as Split A.

It is unlikely that a single split would do the job. For each partition (*North+* and *Remainder*), the attribute test needs to be applied recursively, searching for the optimal split as the tree grows deeper giving way to finer, more homogeneous partitions.

Table 1.5: Calculating information gain for each Split A and B.

Element	Split A		Split B	
Regional	$I(D_{\text{regional}})$	$= 1 - (p^2_{\text{trees}} + p^2_{\text{untouched}})$	$I(D_{\text{regional}})$	$= 1 - (p^2_{\text{trees}} + p^2_{\text{untouched}})$
		$= 1 - ((\frac{13}{39})^2 + (\frac{26}{39})^2$		$= 1 - ((\frac{13}{39})^2 + (\frac{26}{39})^2$
		$= 0.4\bar{4}$		$= 0.4\bar{4}$

Element	Split A	Split B
	$I(D_{\text{A,north}}) = 1 - (p^2_{\text{trees}} + p^2_{\text{untouched}})$	$I(D_{\text{B,north+}}) = 1 - (p^2_{\text{trees}} + p^2_{\text{untouched}})$
Northern	$= 1 - ((\frac{10}{10+5})^2 + \frac{5}{10+5})^2$	$= 1 - ((\frac{9}{9+0})^2 + \frac{0}{9})^2$
	$= 0.4\bar{4}$	$= 0$
	$I(D_{\text{A,remainder}}) = 1 - (p^2_{\text{trees}} + p^2_{\text{untouched}})$	$I(D_{\text{B,remainder}}) = 1 - (p^2_{\text{trees}} + p^2_{\text{untouched}})$
Remainder	$= 1 - ((\frac{3}{3+21})^2 + \frac{21}{21+3})^2$	$= 1 - ((\frac{4}{4+26})^2 + \frac{26}{26+4})^2$
	$= 0.21875$	$= 0.23\bar{1}$
Information Gain	$IG_A = 0.4\bar{4} - \frac{15}{39}0.4\bar{4} - \frac{24}{39}0.21875$	$IG_A = 0.4\bar{4} - \frac{9}{39}0 - \frac{30}{39}0.23\bar{1}$
	$= 0.13\bar{8}$	$= 0.2\bar{6}$

Variable Importance. Unlike a regression framework, input variables on which a CART model is trained are only putforth for the algorithm's consideration – simply including a variable does not mean it is used. The machine uses its attribute tests to determine which should be used to achieve its predictive goal, thus users may feel somewhat uncomfortable about relinquishing some control. However, we can take comfort that the algorithm's pursuit of pure predictive accuracy can reduce arbitrary human decisions in the modeling process.

How does one determine the variables that a CART model relies on? Measures of *variable importance*, sometimes referred to as *feature importance*, can inform a narrative that describes the inner workings of a model. There is a variety of importance measures that can be applied to any algorithm, each of which emphasizes different qualities of "importance".

Permutation importance is a common approach to determining how much of an effect a specific variable has on the accuracy of the model. The idea is to compare the accuracy of a fully specified model to one that omits a variable in question. As one can imagine, machine learning problems often are highly dimensional, implying that a large number of comparison models would need to be trained. An alternative approximation involves training only one fully specified model, then scoring k-number of test sets. Each test set is designed to estimate the importance of a specific variable by randomly shuffling its values, which has the effect of removing a variable from the data in the form of random noise. If the accuracy rate of a scored set is far lower than any other set, then the randomized variable has the gretest contribution to predictive accuracy and is thus the most important variable ((**???**)).

A far less computationally intensive strategy makes use of the *goodness of split* measures (e.g. Gini Impurity) produced from each split. How the measure is calculated is implemented differently in each sofware package and programming language, but all make use of information gain metrics. In the `rpart` package (a popualar CART package in R), variance importance of the k^{th} variable (VI_k) is the sum of information gain from all splits that a variable plays a *primary* or *surrogate* role:

$$VI_k = \sum_{j=1}^{J}(IG_{primary,j}) + \alpha \sum_{m=1}^{M}(IG_{surrogate,m})$$

The distinction between primary and surrogate depends on the completeness of data. A primary variable is used to splits in a CART algorithm. However, missing values in the training or test set present problems for splitting – the algorithm will not know how to handle missing information. In fact, this is a common problem in most modeling approaches, including regression. A clever work around introduced in L. Breiman et al. (1984) is the surrogate variable that serves as a backup splitting variable that mimics the primary variable's split behavior as close as possible. Even if the data is complete, it might be worth considering a surrogate variable to avoid incomplete predictions in new data streams of varying degrees of completeness. VI_k takes

into account the information gained in each of J-number of splits that a variable is the primary as well as the M-number of splits when the variable is a surrogate. As the surrogate variable is an approximation, its importance is tempered by how often it agrees with the primary (α). The resulting VI_k can either be presented in raw information gains or normalized to the largest VI($VI_{normalized} = 100 \times \frac{VI_k}{max(VI)}$).

Which one should be used? It helps to think about the value of variable importance in the context of implementation. A cursory look of the goodness of split measures can help articulate which variables contain more information than others. But from an operational perspective, permutation importance can be relied upon to directly estimate the usefulness of a variable. As more private data sources become privately available, the accuracy of forecasts and predictions may increasingly rely on specific variables, like the use of alternative data (e.g. email receipts, satellites) in economics and finance. The costs and benefits of each variable can be weighed on its contribution to model performance, whether small or large, which in turn can information data acquisition priorities. For models that are used for tactical, person-level decisions, it is worth understanding the risks of relying too much on specific variables. For example, the hurricane data heavily relies slightly more on latitude than longitude to make predictions. If the geographic accuracy of latitudinal coordinates falls, what risk does that pose?

1.3.2 In Practice

As we have shown, decision tree learning is a remarkable resource for prediction problems. Their versatility allows them to conduct automated variable selection on any type of data type, even for ill-posed problems when the number of input variables exceed the number of records. Furthermore, CART can detect patterns in data that would otherwise require substantial effort to detect through regression modeling. When tree models achieve visibly higher predictive accuracy than a regression approach, this may be an indication of the presence of discontinuities, interactions, and non-linearities in the data. For example, regressions would have been a poor choice for detecting downed trees as affected areas are not more likely as latitude or longitude increase, but rather on a gradient but rather in discontinuous pockets. Using CART as a diagnostic can in turn help improve the quality of regression-based approaches.

When the optimal, pruned tree is not exceedingly complex, each leaf node's records can be visualized and characterized as a "profile". Areas in Northern Queens that were spared by the Hurricane Sandy, for example, may be easily defined by a polygon whose vertices are coordinates learned from a CART algorithm. Often times, however, trees need to be somewhat complex in order to learn the quirks and eccentricies of the data, making the profile-based communication strategy less tractable. Instead, consider using variable importance to articulate how much more information one input variable contributes relative to others. While variable importance does not lend itself to crafting a story of how X causes Y, it can provide audiences a look under the hood and some re-assurances about transparency. In variable importance values for each latitude and longitude were 36 and 39.5, respectively, indicating that both factors had comparable contributions to the prediction problem. However, if the variable importance for longitude were 240, then storm damage may have been more heavily distributed along a East-West gradient.

There are shortcomings of CART models, particularly their reputation for overfitting the data. The problem lies in splitting data: as the tree grows to its fullest extent, partitions are forced split even if the information gained is negligible and eventually reaching a point when an individual leaf node is of size $n = 1$. It is for this reason that trees must be pruned.

Table 1.6: The pros and cons decision trees.

Useful Properties	Challenges
Rules (e.g. all the criteria that form the path from root to leaf) can be directly interpreted.	Data sets with large number of features will have overly complex trees that, if left unpruned, may be too voluminous to interpret.
Method is well-suited to capture interactions and non-linearities in data.	Trees tend to overfitted at the terminal leafs when samples are too small.

Useful Properties	Challenges
Technique can accept both continuous and continuous variables without prior transformation.	Feature selection is conducted automatically.

1.3.3 DIY: Predicting monetary and non-monetary relief

Everyday, consumers of financial products submit complaints to the Consumer Financial Protection Bureau (CFPB). Below, is an anonymized complaint written by a real consumer of a non-federal student loan:

> *Navient has been unresponsive after I've asked to speak with several managers. I have called them consistently since summertime as my automatic payments were not being taken out. They say it came back as NSF, but my bank says it was never billed. Then Navient admitted to having I.T. issues. Late fees, interest, you name it. After calling to resolve many times, they offered me payment options and promised to repair my credit damage of their reporting me. After making each payment, they would change the terms and tell me I had to pay additional money or pay this or that, and then they would rescind the negative info. I finally scraped up the money and paid what it said I was past due on after their lies - and the online account still shows I owe money as of {xx/xx/xxxx} and its {xx/xx/xxxx} - so my payment made on {xx/xx/xxxx} obviously didn't post although it was taken from my bank account. I called again today and requested that a manager call me back - nothing yet. Supposedly they record all calls - well, someone should listen to all the lies they've been telling me over the months and changing the agreements. It is unethical and immoral. They are ruining students lives! Check the recordings and the call logs - I'm the one who calls them! And if they ever seldom call me, they never leave a voicemail but the recordings will prove they lie and say they did leave a voicemails, and I dispute that and tell them they are lying directly. listen to the recordings!*

For transparency, CFPB publishes complaint narratives as an open, anonymized database furnished with the details of individual cases and their disposition. In the narrative example, we can identify a mix of facts that are particular to a case. In addition, the choice of words signals the sentiment of the user. When we examine the patterns that emerge throughout the *corpus* (collection) of consumer complaints, we may find clues about how some financial instruments are more likely to receive monetary relief while others are not.

The key to anticipating the outcome of a complaint lies in the text. If the current state of consumer protection can be nowcasted, policy makers and the watchdog apparatus can make pre-emptive interventions rather than reactive responses. However, to use the textual information requires work. Prior to machine learning, a large team of analysts might be tasked to read through and assign tags to each piece of text to generate usable covariates. This, however, is a long arduous task requiring a significant amount of labor hours. In this DIY, we explore how CART can capitalize upon the intricate patterns in textual data to make accurate predictions. In particular, we rely on the algorithm's properties to conduct implicit variable selection and find iteractions among words that predict monetary relief.

Data. To start, let's load a sampling of CFPB complaints that cover a 49-month period from March 2015 to March 2019. To make the problem tractable for the DIY, we only focus on complaints that have resulted in either monetary and non-monetary relief.

```
#Load data
  load("data/cfpb_dtm.Rda")
```

Each row in the data is a real consumer complaint (see the `consumer.complaint.narrative` field). Similar to parsing the State of the Union in Chapter 4, each narrative has been *tokenized*. First, text is standardized through stemming word endings (e.g. tell, tells, and telling all become "tell"). The stemmed words are processed into *n-grams* of individual words and short word sequences. For example, "*the online account still shows I owe money*" contains eight unigrams (e.g. the, online, account, still, shows, I, owe, money), seven bigrams (e.g. the online, online account, account still, still shows, shows I, I owe, owe money), and six trigrams (e.g. the online account, online account still, account still shows, still shows I, shows I owe, I owe

Table 1.7: Sample words frequencies.

High		Medium		Low	
Variable	Frequency	Variable	Frequency	Variable	Frequency
account	18843	mail.address	18	local.polic	2
credit	12972	major.credit	18	loos	2
call	9597	manipul	18	lower.rate	2
payment	8801	master.card	18	luckili	2
bank	8476	medic.bill	18	lure	2
card	7914	merchant.servic	18	mailbox	2
report	7395	monitor.servic	18	major.credit.report	2
charg	6878	overcharg	18	maker	2
receiv	6103	pacif	18	malici	2
told	4924	paycheck	18	manag.call	2

money). In total, the sentence yields a *bag of words* of 21 variables. Of course, some of these words are filler (e.g. "the", "I") and can be treated as *stop words* or words that can be removed as they do not likely contain signal. Combinations of these n-grams are the key to understanding what is associated with monetary relief – it is a game of interactions that is well-suited for CART algorithm. The processed data set contains 469,046 unique n-grams, but was reduced to tokens with three or more characters and appear ten or more times in 2015 ($k = 4657$).

The data have been partitioned into a training set containing complaints from 2015 (n = 9519) and a test set of the remaining years (n = 49547). Within the data, the target is the `target.series` variable that includes labels for both *Closed with non-monetary relief* and *Closed with monetary relief*. A tabulation of the target finds a balanced sample with approximately equal representation of the two classes.

```
#Quick summary
  table(train$target.series)
```

```
##
##     Monetary Non-monetary
##         4894         4625
```

A cursory review of the n-grams finds an abundance of details. The word frequencies in Table 1.7 give hints of what is described in the narratives. The most frequent words help set the stage, focusing on the problems with certain financial instruments (e.g. credit, account) or action (e.g. payment). Less frequent words are likely the finer details of the problem, specific to each consumer's case. By combining words through interactions, a CART model can map out the problem space, finding the patterns that underlie monetary relief.

Training. As always, we advocate for writing estimators and algorithms at least once to understand the mechanics. However, for applied problems, training a CART is made easy through open source libraries. The `rpart` library is among the most popular implementations in R:

```
  pacman::p_load(rpart)
```

Once loading the function, we find the library comes with flexible options to guide how a decision tree is induced. At a minimum, `rpart` requires three parameters:

```
rpart(formula, method, data)
```

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.

These three inputs are just enough to train a model, but training just any model does not guarantee predictive accuracy. In fact, the complexity of the tree defaults to a fairly simple model – it may simply gloss over the finer details that make a model predictive. To tune the tree's complexity, we adjust the complexity parameter `cp` value along with the `minbucket` and `minsplit`:

- `cp` is a numeric indicates the complexity of the tree. $cp = 1$ is a tree without branches, whereas $cp = 0$ is a fully grown tree. If cp is not specified, `rpart()` defaults to a value of 0.01. The tree is
- `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.

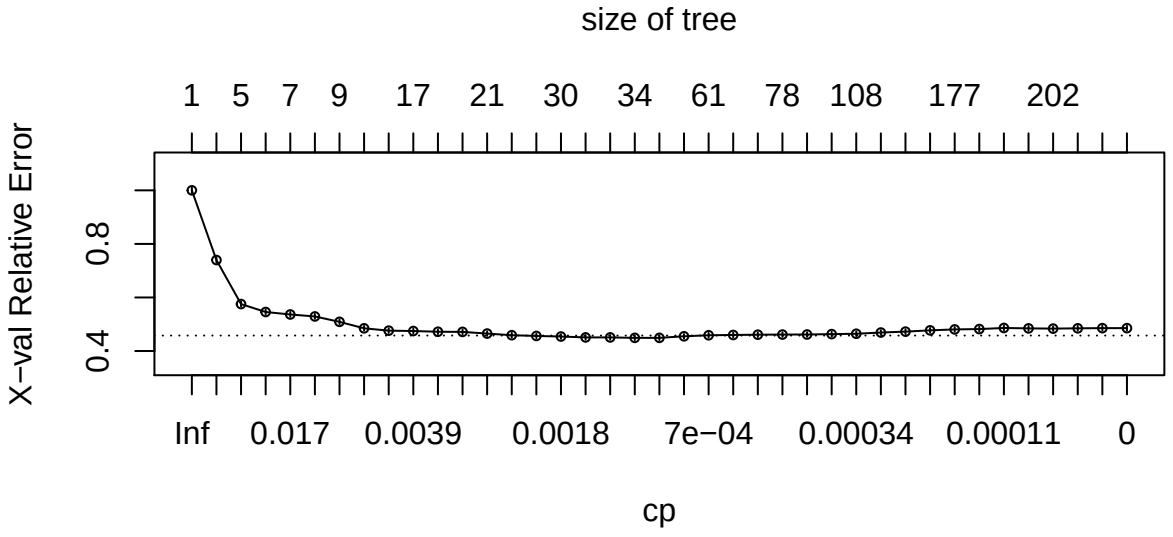
As a first pass, we recommend setting `cp = 0` to allow the tree to its full glory. It may take the CART algorithm a few minutes to learn the patterns in the words.

```
fit <- rpart(target.series ~ .,
             method = "class",
             data = train[, -c(1:5)],
             cp = 0)
```

The `fit` object captures all of the inner workings of the decision tree. For example, just plotting the fit object will show the full depth of the tree. More importantly is the cross validation results collected at each level of additional complexity. Using the `printcp()` function, we can extract the *CP table*, which contains various accuracy measures associated with each value of the tree complexity value `cp`, including:

- the number of splits `nsplit`,
- the prediction error in the training data `rel error`,
- the cross-validation error `xerror`, and
- the standard error `xstd`.

```
plotcp(fit)
```



Tuning. *How do we find the optimal tree depth?* First, find 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[1]. The idea behinds this rule of thumb takes advantage of uncertainty: the true value lies somewhere within a

[1] (???)

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 = 33, nsplit = 38 and xerror = 0.449081081081081. By applying the rule, the upper bound of acceptable error is xerror = 0.449081 + 0.008713 = 0.457793839665076, xerror = 0.449081 + 0.008713 = 0.457793839665076. As it turns out, the tree with nsplit = 27 is within one standard deviation and is thus the best model.

In other words, the following function can extract the optimal `cp` value.

```
bestCP <- function(fit_obj){
  ## Returns best CP val within 1 SD of lowest xerror
  #
  ## Args:
  ##   fit_obj: decision tree object
  #

  #Pull cross-validated error
    xerror <- fit$cptable[, 4]

  #Find lowest error and associated xstd
    best_error <- min(xerror)
    best_sd <- fit$cptable[, 5][which(xerror == best_error)]

  #Pull CP closest to lower bound
    lower_bound <- best_error + best_sd
    opt_select <- fit$cptable[,1][which(xerror <= lower_bound)][1]

   return(opt_select)
}
```

Now, we can prune the tree using the optimal `cp` value, then score both the test set. As a comparison point, we will also apply the unpruned model as well.

```
#Get best CP
  best_value <- bestCP(fit)

#Prune tree
  fit.opt <- prune.rpart(fit, cp = best_value)

#Score, returning probabilities
  pred.full <- predict(fit, test, type = 'class')
  pred.opt <- predict(fit.opt, test, type = 'class')
```

What works. One of the fascinating aspect of CART is its interpretability. Each terminal node is a set of binary criteria, making it possible to articulate under what conditions can the target occur. This is a reasonable mode of interpretable when trees are relatively simple. Alternatively, CARTs can be reviewed through variable importance that builds upon the *impurity* measures used to construct the trees.

Below, we can see that the measure can be extracted from the `fit.opt` object. As may be expected, `accel` is not the main contributor to predictions, but rather measures of the maximum, mean and variability of acceleration. This also implies that the model could be further tuned by trying different windows for producing the engineered variables – perhaps shorter or longer windows could be even more important.

```
  fit.opt$variable.importance
```

When applied to the function to the predictions (`pred.opt` and `pred.full`), we find that the mean F1-statistics reached `meanF1(test$activity, pred.opt)` and `meanF1(test$activity, pred.full)` – not bad for a first cut, but certainly can benefit from extra attention.

Table 1.8: Words with high, medium and low importance for predicting monetary relief.

High		Medium		Low	
Variable	Impurity	Variable	Impurity	Variable	Impurity
report	585.111	close.cost	1.986	NA	NA
charg	455.559	close.disclosur	1.986	NA	NA
credit.report	268.365	creditor	1.882	NA	NA
bank	208.821	violat	1.882	NA	NA
card	104.090	alleg	1.838	NA	NA
close	69.498	citigold.check	1.832	NA	NA
experian	55.761	manag	1.758	NA	NA
check	51.517	direct.deposit	1.686	NA	NA
monei	45.922	aadvantag.mile	1.684	NA	NA
late.fee	45.250	citigold.account	1.684	NA	NA

1.4 Random Forests

How do we know anything for sure? Virtually every aspect of life has some uncertainty tied in. When a hurricane approaches the US Eastern Seaboard, forecasters often map the *cone of uncertainty* that provides the possible range of motion of a storm based on the results of many forecasted simulations. In presidential elections, often times the most polling results are ones that ensemble or average the results of many other similarly conducted polls. The reliance on predictions from a group of models with the same aims may well improve prediction accuracy. In statistical learning, average the results of multiple models is known as *ensemble learning* or *ensembling* for short.

Single models may imposes biases on data and may be well-suited in specific situations. Ensemble methods combine the results of many models to obtain more stable results. For example, the curve in graph #1 can be approximated using a decision tree algorithm. The result of a single tree only loosely fits the curve in a 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 is an extension of decision trees using a modified form of bootstrapping and ensemble methods to mitigate overfitting and bias issues.[2] 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.

[2]Leo Breiman (2001)

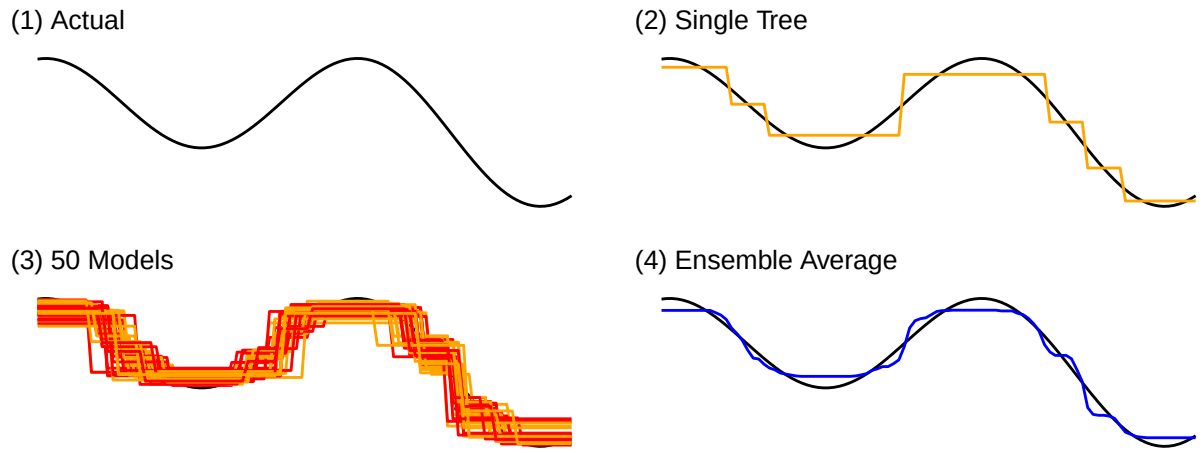


Figure 1.8: Comparison of results of applying a single model to fit a curve versus an ensemble of models.

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 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.

1.4.1 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.

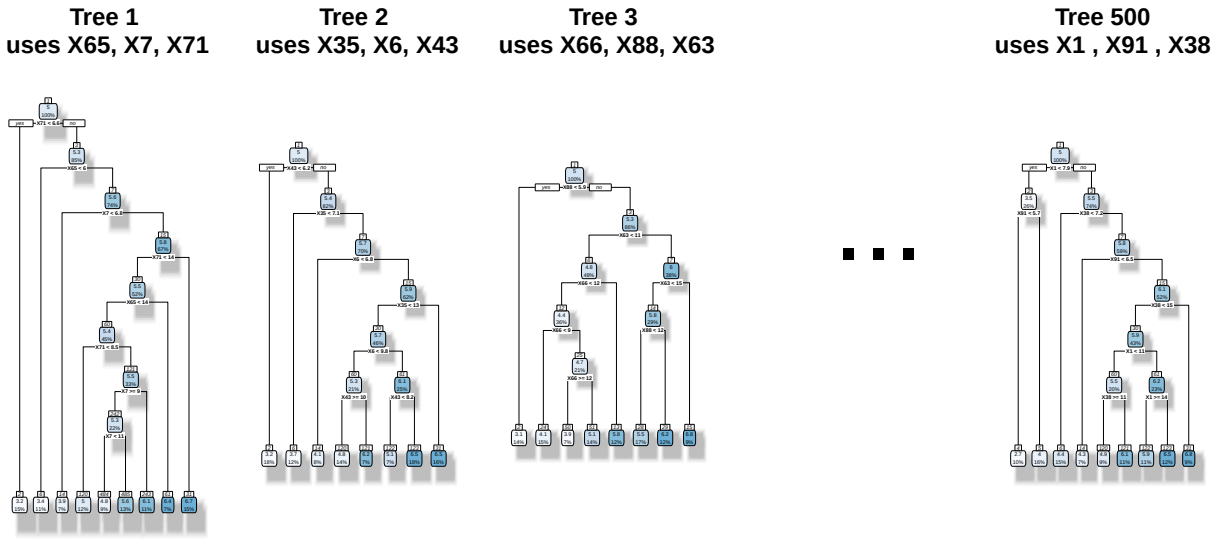


Figure 1.9: Random Forests construct hundreds of trees sampling from both observations and features, then combine the trees into one prediction through voting.

1.4.2 DIY: Revisiting monetary relief

There are a number of R libraries that implement the Random Forest algorithm. The more commonly used version is `randomForest` as it automates most of the procedure, but is less scalable and efficient than its younger sibling `ranger`. As we will revisit the bag-of-words data set in this DIY, the `ranger` library will vastly reduce the time required to train the Random Forest. The `ranger` function expects at least a formula and a data frame,

```
ranger(formula, data, mtry, numtree)
```

where: - `formula` is an expression of the model to be train. The target variable should be in factor format. - `data` is a data frame. - `mtry` (optional) is the number of variables to be randomly sampled per iteration. Default is $\sqrt{k}$ for classification trees. Default set to the square root of the number of variables. - `ntree` (optional) is the number of trees. Default is 500. - `importance` (optional) needs to be specified as "impurity" in order to retrieve variable importance measures. - `num.threads` (optional) is a speed enhancing option that allows the many repetitive steps of a Random Forest to be parallelized across multiple cores or CPUs. By default, `ranger` uses the number of CPUs available.

Using the same formula as the `rpart()` function, we can train a Random Forest with default settings and check the OOB error.

```
#Load randomForest library
  pacman::p_load(ranger)

#Run Random Forest
  fit.rf <- ranger(target.series ~ .,
                   data = train[, -c(1:5)],
                   num.trees = 500,
                   importance = "impurity")
```

```
## Growing trees.. Progress: 19%. Estimated remaining time: 2 minutes, 12 seconds.
## Growing trees.. Progress: 36%. Estimated remaining time: 1 minute, 48 seconds.
## Growing trees.. Progress: 55%. Estimated remaining time: 1 minute, 17 seconds.
## Growing trees.. Progress: 75%. Estimated remaining time: 41 seconds.
## Growing trees.. Progress: 95%. Estimated remaining time: 8 seconds.
```

Table 1.9: Sample words frequencies.

High		Medium		Low	
Variable	Frequency	Variable	Frequency	Variable	Frequency
charg	202.843	final.payment	0.162	compani.polici	0.003
report	176.480	revok	0.162	pai.servic	0.003
bank	119.372	stoneg.mortgag	0.162	nt.worri	0.003
card	87.293	fargo.credit	0.162	origin.account	0.003
credit.report	71.066	integr	0.161	inform.request	0.003
debt	58.299	atm.withdraw	0.161	miss.call	0.003
check	49.193	truck	0.161	excess.amount	0.003
monei	43.099	town	0.161	nt.appli	0.003
transact	36.940	circl	0.161	sale.price	0.003
collect	35.603	renov	0.161	receiv.bill	0.003

Approximately 75.6% of observations in the OOB sample were correctly classified using randomly selected variables in each of the 500 trees.

The `fit.rf` records a number of model outputs such as variable importance calculated as the Mean Decrease Gini. However, the values themselves do not have any meaning outside of a comparison with other Gini measures.

```
fit.rf$variable.importance
```

By default, the `ranger` library sets the number of trees to equal 500. But what if we would like to find the model that optimizes predictive accuracy? Random Forest algorithms can be tuned by the number of underlying trees in the forest (`num.trees`), the number of variables sub-sampled for any given tree (`mtry`), the minimum node size of terminal nodes (`min.node.size`) among others.

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 tree are optimal.

```
library(caret)
data(iris)

grid_params <-  expand.grid(mtry = c(1,4),
                            min.node.size = 1,
                            splitrule = "gini")

fitControl <- trainControl(method = "CV",
                           number = 5,
                           verboseIter = FALSE)
```

```
fit = train(
  x = iris[ , names(iris) != 'Species'],
  y = iris[ , names(iris) == 'Species'],
  method = 'ranger',
  num.trees = 200,
  tuneGrid = grid_params,
  trControl = fitControl
)
```

Normally, we can plug the tuned parameter back into the `randomForest()` method and re-train the algorithm, but it unnecessary in this case as the default model already uses the same parameters. When applied to the test set, we see that the mean F1-statistic is much improved – or a whole 10-percentage point increase.

```
#Predict classes in test
  yhat <- predict(fit.rf, test)
```

This result does not mean that Random Forests will always turn better results, but rather multiple techniques should be tested when tackling prediction problems. Also, remember the policy goal: is the objective to predict or to explain? If a little of both, then it is worth understanding the value of increased accuracy at the cost of interpretability.

References

Breiman, L., J. Friedman, C.J. Stone, and R.A. Olshen. 1984. *Classification and Regression Trees*. The Wadsworth and Brooks-Cole Statistics-Probability Series. Taylor & Francis. https://books.google.com/books?id=JwQx-WOmSyQC.

Breiman, Leo. 2001. "Random Forests." *Machine Learning* 56 (1): 5–32.

Chen, Jeffrey, Abe Dunn, Kyle Hood, Alexander Driessen, and Andrea Batch. Forthcoming. "Off to the Races: A Comparison of Machine Learning and Alternative Data for Predicting Economic Indicators." *National Bureau of Economic Research (NBER): Big Data for 21st Century Economic Statistics*. http://papers.nber.org/conf_papers/f109801.pdf.

Cole, Dermot. 2013. "Phone Map App Directs Fairbanks Drivers onto Airport Taxiway." *Anchorage Daily News*, September. https://www.adn.com/aviation/article/iphone-map-app-directions-fairbanks-drivers-airport-taxiway/2013/09/24/.

Koosner, Amanda. 2013. "Apple Maps Leads Drivers onto Alaska Airport Taxiway." *CNET*, September. https://www.cnet.com/news/apple-maps-leads-drivers-onto-alaska-airport-taxiway/.

Kraus, Rachel. 2019. "Out of Traffic, into a Ditch: Why Waze on Snowy Mountain Roads Could Be a Bad Idea." *Mashable*, February. https://mashable.com/article/is-waze-apple-maps-google-safe-in-the-snow/.

Laptev, Nikolay, Slawek Smyl, and and Santhosh Shanmugam. Forthcoming. "Off to the Races: A Comparison of Machine Learning and Alternative Data for Predicting Economic Indicators." *National Bureau of Economic Research (NBER): Big Data for 21st Century Economic Statistics*. http://papers.nber.org/conf_papers/f109801.pdf.

Microsoft. 2018. "Computer Generated Building Footprints for the United States." *Github Repository*. https://github.com/microsoft/USBuildingFootprints.

Panzarino, Matthew. 2018. "Apple Is Rebuilding Maps from the Ground up." *CNET*, June. https://techcrunch.com/2018/06/29/apple-is-rebuilding-maps-from-the-ground-up/.

Trenholm, Richard. 2013. "Apple Buys Location Data Company to Sort Out Maps App." *CNET*, July.

https://www.cnet.com/news/apple-buys-location-data-company-to-sort-out-maps-app/.

W. Han, L. Di, Z. Yang. n.d. "CropScape - Cropland Data Layer." *National Agricultural Statistics Service*. https://nassgeodata.gmu.edu/CropScape/.

Wagstaff, Keith. 2017. "LAPD Warns That Navigation Apps Are Steering People to Neighborhoods on Fire." *Mashable*, December. https://mashable.com/2017/12/07/lapd-warns-that-navigation-apps-are-steering-people-to-neighborhoo

Wallace, Tim, Derek Watkins, and John Schwartz. 2018. "A Map of Every Building in America." *The New York Times*, October. https://www.nytimes.com/interactive/2018/10/12/us/map-of-every-building-in-the-united-states.html.