Chapter 10: Different Flavors of Classifiers

1 Classifiers come in different flavors

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

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

2 K-Nearest Neighbors (KNN)

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

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

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

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

2.1 Under The Hood

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

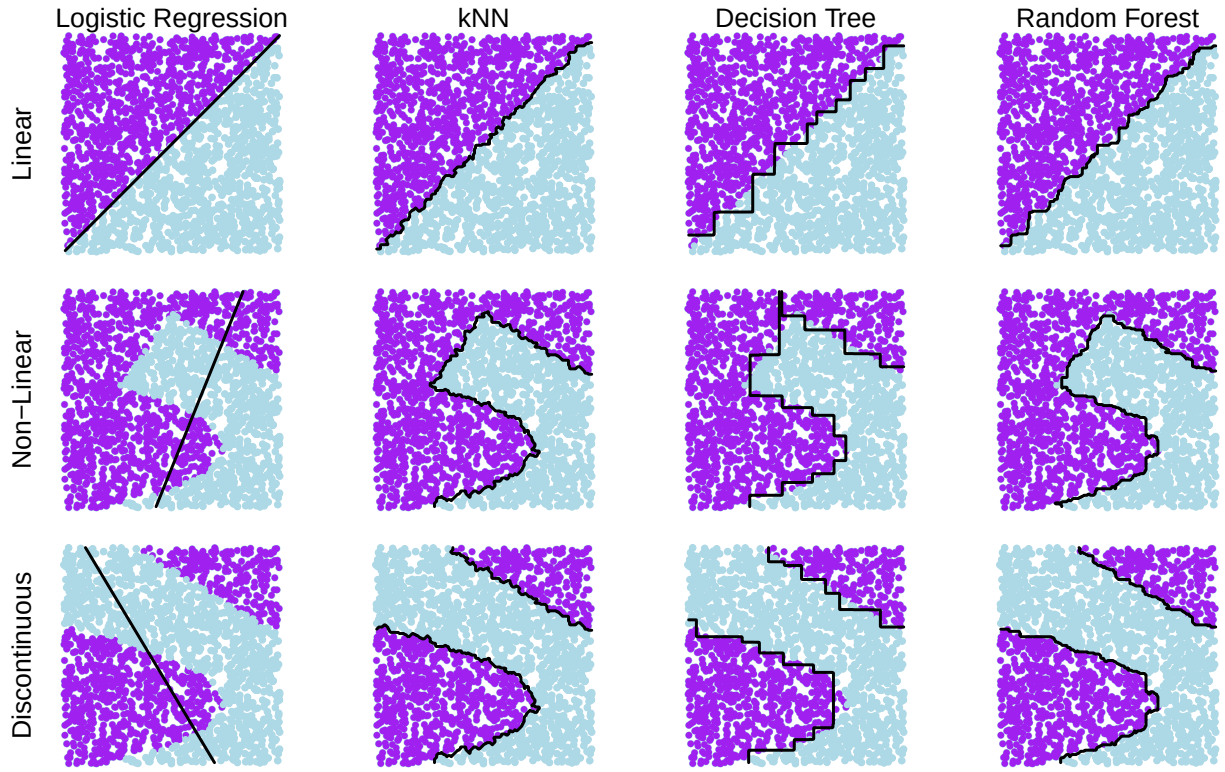


Figure 1: Linear, Non-Linear and Discontinuous Classification Problems.

quantify how close points are. It is instance-based as each prediction is determine on a case-by-case basis using its surrounding neighbors.

The technique is simple. For each case y_i:

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

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

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

Voting Type	Formula	Interpretation
Rectangular	$Pr(Y = j) = \frac{1}{k}\sum_{i=1}^{k} I(y^i = j)$	Calculate the proportion of j based on k nearest neighbors.

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

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

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

To illustrate how tuning works, we have extracted a raster data from the US Department of Agriculture (USDA) CropScape data layer derived from satellite imagery.[1] In particular, we focus on a small area in Kansas that grows corn (yellow) and soybeans (green). More often than not, data will only contain a sample of the full picture. In the case of spatial data, KNNs are quite useful for filling in the full picture. Suppose we only have a 10% sample of the farmland – how important is k? We can see that as the value of k grows, the corn fields become overly represented as it is the majority class in the sample. The TPR is highest when the TNR is highest, demonstrating that there is a balancing act when choosing the value of k.

2.2 Tips of the trade

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

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

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

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

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

- *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] https://nassgeodata.gmu.edu/CropScape/

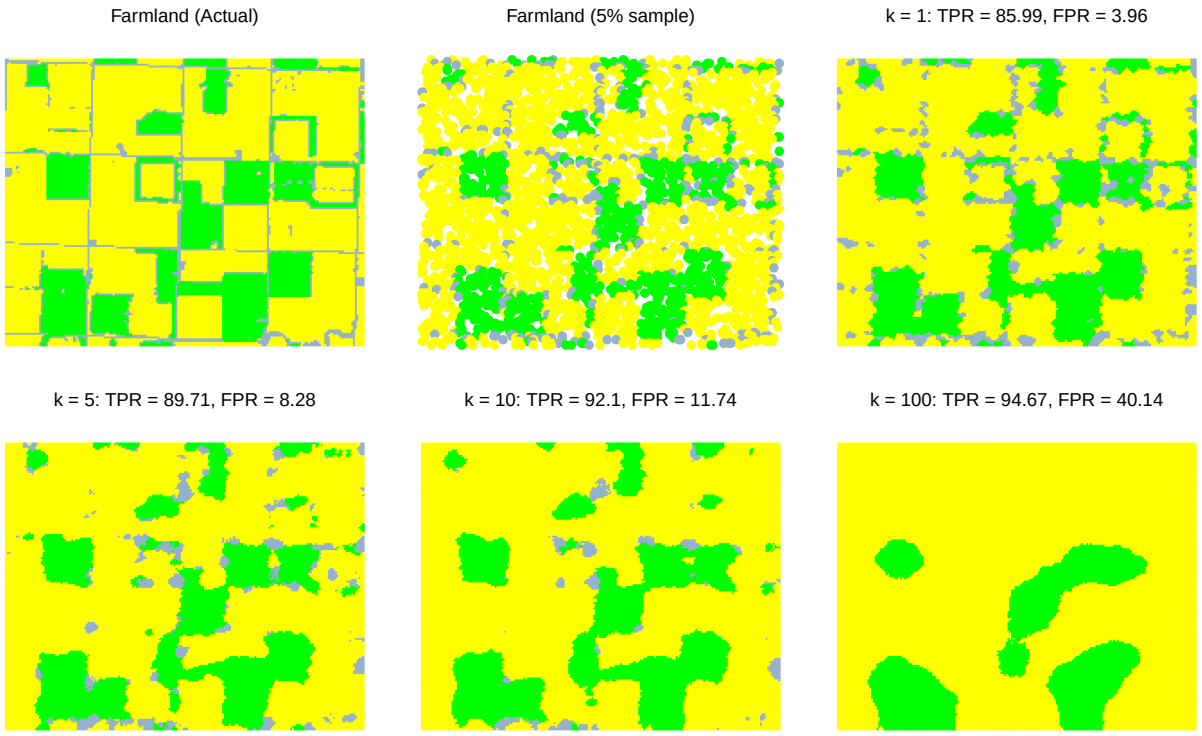


Figure 2: Comparison of prediction accuracies for various values of k.

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

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

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

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

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

Table 2: The good and ugly of KNNs.

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

2.3 DIY: Anticipating the extent of damage from a storm

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

Prepare the data. Some calls for help are associated with downed trees while others may concern non-

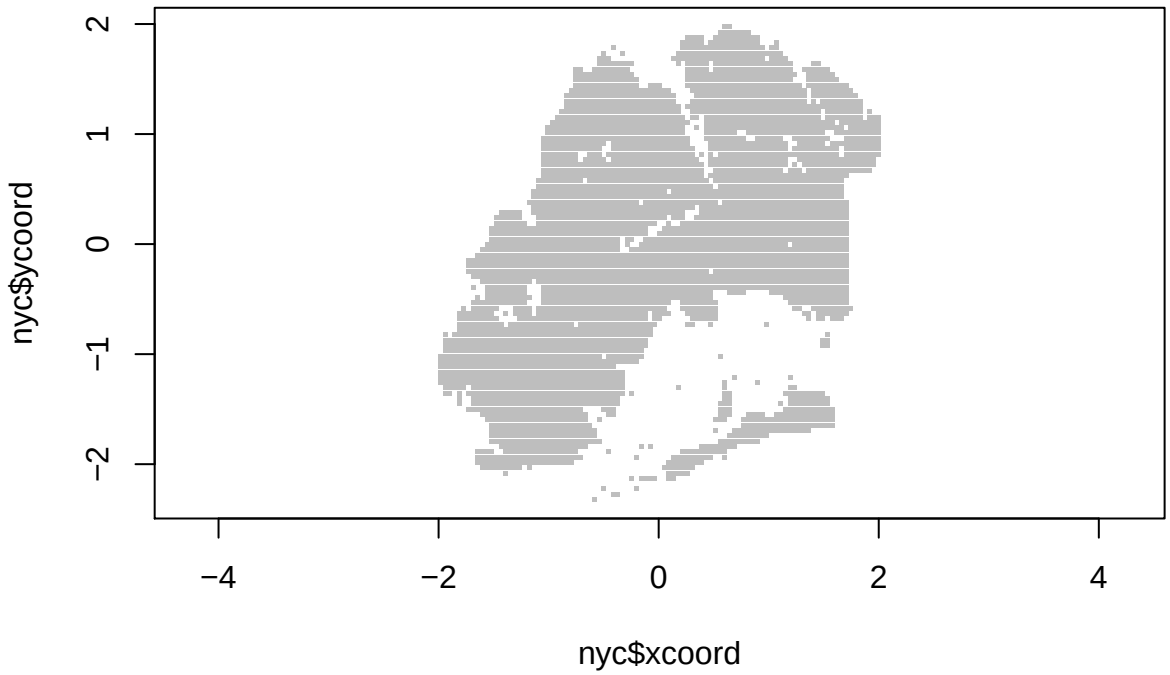


Figure 3: Plot of X-Y coordinates for Brooklyn and Queens.

```
  nyc$xcoord <- scale(nyc$xcoord)
  nyc$ycoord <- scale(nyc$ycoord)

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

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

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

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

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

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

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

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

```
  library(kknn)
```

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

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

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

- `formula` is a formula object (e.g. "`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.

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

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

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

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

```
   plot(fit.cv)
```

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

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

- `formula` is a formula object (e.g. "`no.coverage ~ .`").
- `train` is a matrix or data frame of training data.
- `test` is a matrix or data frame of test data.

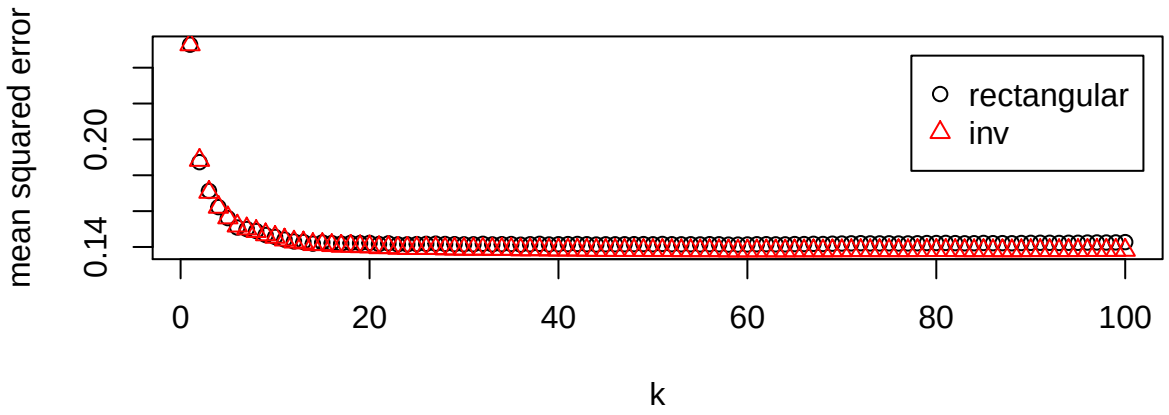


Figure 4: 20-fold cross validated errors for k = 1 to k = 100

- `k` is the number of neighbors.
- `kernel` is the type of weighting of distance (e.g. "rectangular" is unweighted, "biweight" places more weight towards closer observations).
- `distance` is a numerical value indicating the type of Minkowski distance. (e.g. 1 = binary, 2 = euclidean,).

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

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

Using the predicted probabilities for the test sample, we calculate the TPR and FPR using both a naive cutoff threshold ($p = 0.5$), finding a high TPR but a FPR of almost similar magnitude. Using such prediction would not provide any insight to field crews.

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

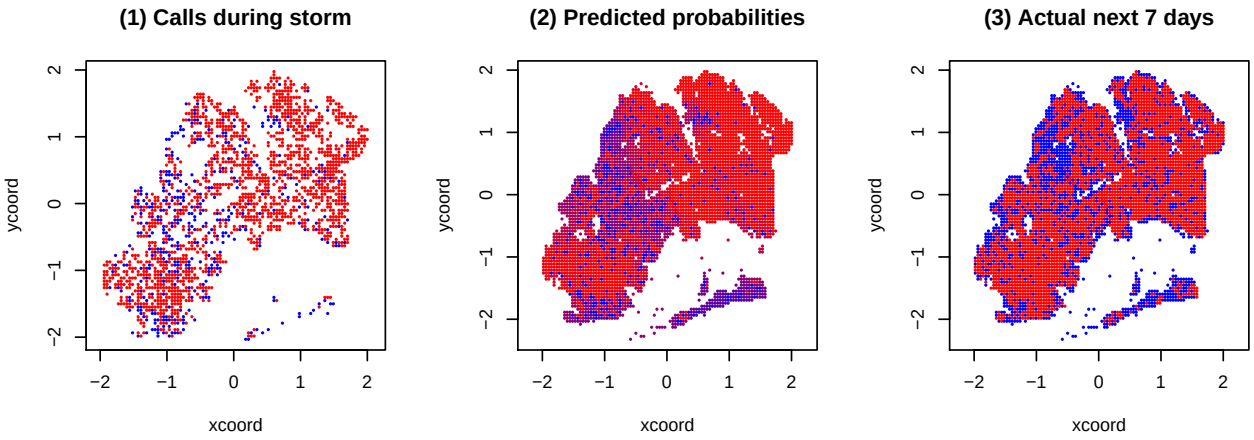


Figure 5: Comparison of actual and predicted areas with reported downed trees. Red indicates at least one tree was reported in a given 0.359 square-mile area

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

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

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

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

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

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

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

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

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

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

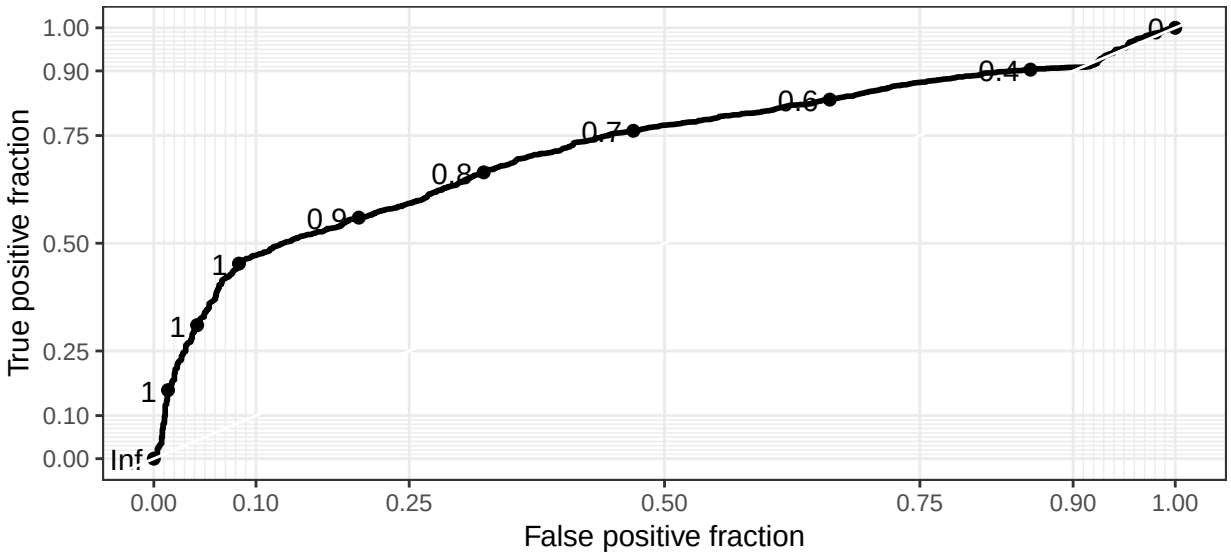


Figure 6: ROC curve out of sample

```
#Show result
  roc
```

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

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

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

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

2.4 Practice Exercises

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

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

3 Decision Tree Learning

Mobile technologies have lowered the bar to using lightweight sensors that measure the physical world and have opened new applications of data in daily life. From a smart phone's accelerometer, it's possible to

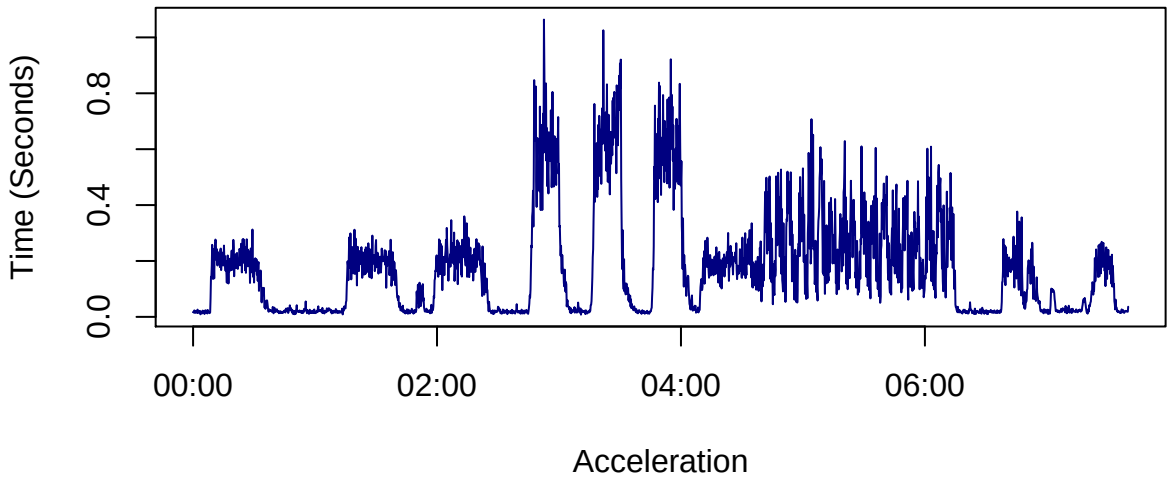


Figure 7: Accelerometer data collected from an iPhone.

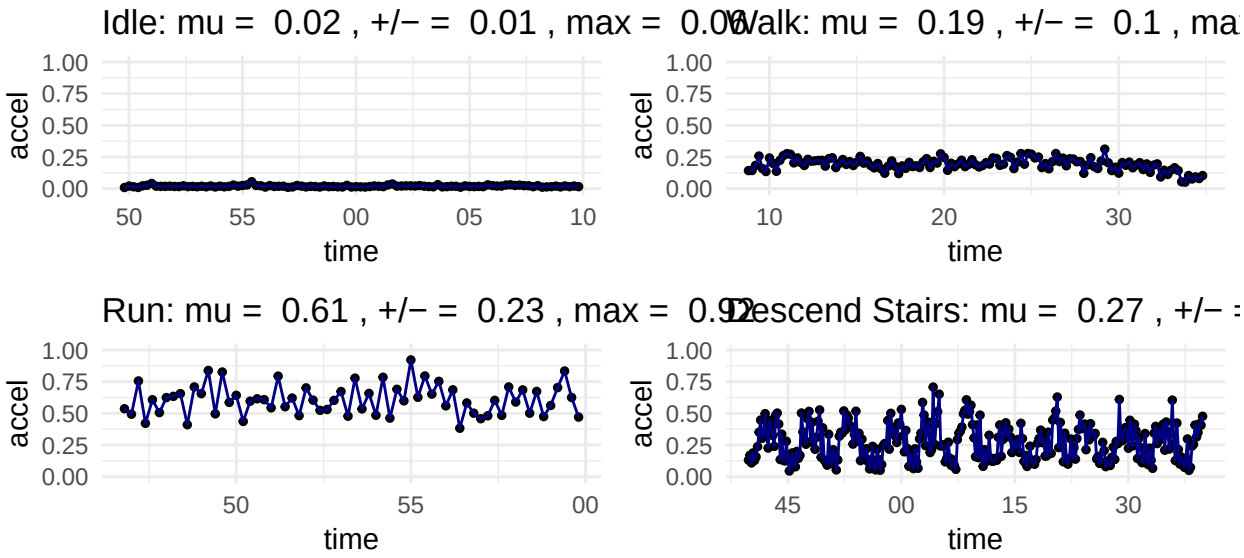


Figure 8: Acceleration by type of movement.

track distinct patterns in one's activity based on the fluctuations in acceleration ($\frac{m}{s^2}$). In fact, many of these technologies have become commonly available, enabling physical fitness activity monitoring to characterizing transportation quality. Below is a set of exercise measurements from an smartphone accelerometer that lasted approximately 6.5 minutes and graphed at a frequency of 5 hertz (five readings per second).

Can you visually identify distinct patterns? What makes those patterns distinct?

It becomes immediately apparent that the methods covered thus far are not suitable for the task at hand. If we manually extract samples from these periods, we can quantify the patterns in terms of their central tendencies. Idle periods have near zero acceleration, walking periods have acceleration around 0.2 with tight dispersion, running periods hover around 0.6 +/- 0.2, and descending stairs vary widely. Using simply the level of acceleration may not accurate as at least two types of motion have overlapping distributions.

Decision tree learning can help bring clarity. Trees are designed to look at inputs and partition the sample into smaller more homogeneous cells with respect to the target. This recursive partitioning allows a tree to resemble an inverted tree: moving away from the base of the tree, 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

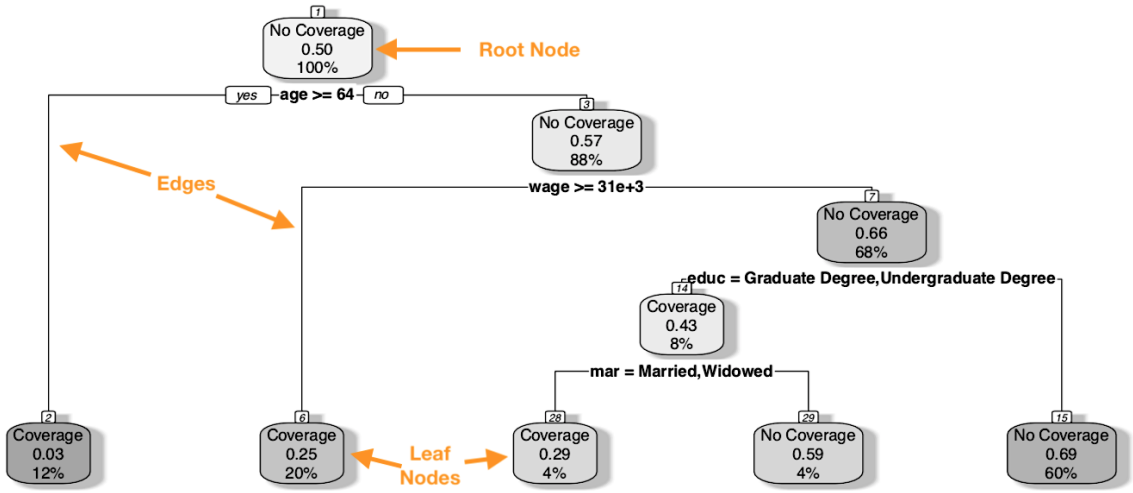


Figure 9: Anatomy of a Decision Tree.

leaves.

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

3.1 Under the hood

Anatomy of a decision tree. The tree is comprised of nodes and edges. Nodes (circles) contain records. Edges (lines) show dependency between nodes and is the result of an *attribute test* – or a process that finds 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 node is labeled using majority voting based on whichever class is most represented. The goal is to grow a tree from the root node into as many smaller child nodes that contain more of one class than another.

Decision trees split nodes based on finding thresholds along the input variables. There can be seemingly infinite number of potential variable-threshold combinations – which is best? Drawing from *information theory*, we can apply an *information gain* formula to evaluate all candidate splits and find one that provides the most information. This optimal split yields to more homogeneous child nodes, which in turn can be split even further. The search for the best threshold is known as an *attribute test*.

As we can see in the example decision tree for health care insurance, each node is connected to at least one other node. Starting at the root node, we can see that overall, the population is labeled "no coverage" based on the decimal percentage 0.5. The 100% indicates the proportion of the sample that is contained at the node. Below is `age >= 64`, which is the most informative attribute test that is used to split. To the left, the edge leads to another node at the bottom left corner of the diagram, which contains people who are age 64 or older. While the leaf node only contains 12% of the entire sample, it is almost exclusively people who have health care coverage. To the right, the remaining 88% of the sample, which is further split by wage and other variables. Each leaf node is defined as the intersection of multiple binary criteria, giving way to profiles of users that can be easily segmented.

Growing trees. 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). The process is fairly clear cut and iterative:

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 to attribute testing using either *Information Gain* or *Gini Impurity* as will be covered in the following section. At the root node, if there are 10 variables with each 30 possible thresholds, attribute tests are applied 300 times choosing the candidate threshold that yields the most homogeneous child nodes. Upon splitting, attribute tests are applied to each child node, making this a recursive partitioning procedure.
3. **Stopping Criteria versus Pruning**. At some point, the algorithm needs to stop. The question is *when*? One way is to grow the tree until some *stopping criteria* are met, such as if a leaf has fewer records than a pre-specific threshold, the purity or information gain falls below a pre-specified level, or if the tree has grown to n-number of levels (e.g. number of rows of splits). While stopping criteria are useful, the results in some studies indicate their performance cap the tree from reaching its full predictive potential.[2] 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.

Attribute Tests. Information gain is a form of *entropy* that measures the consistency of information. Based on these distinct states of activity, entropy is defined as:

$$\text{Entropy} = \sum_{i=1}^{k} -p_i log_2(p_i)$$

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:

[2]

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

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

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

- 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.

$$\text{Gini Impurity} = \sum_{i=1}^{k} p_i(1 - p_i) = 1 - \sum_{i=1}^{k} p_i^2$$

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

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

3.2 Tips of the trade

Like any technique, decision trees have strengths and weaknesses. Unlike logistic regression and KNN, decision trees can conduct automated variable selection on any type of data type. In addition, the recursive partitioning produce a tangible definition for each subpopulation represented by a node. The splitting mechanism make it possible to capture interactions and non-linearities that are otherwise not easily accounted for in the previous methods. The implications are that two or more input variables can be blended together to find cells of activities that are otherwise overlooked.

There are detractors, however. Trees can be grown so deeply that there are too many subpopulations to articulate. If left unpruned, terminal leafs may give a false impression of accuracy and precision – the small

samples may give a false impression. Nontheless, decision tree learning is an important contribution to classification problems and form the basis of many other algorithms.

Table 5: The good and ugly of 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.
Technique can accept both continuous and continuous variables without prior transformation.	Feature selection is conducted automatically.

3.3 DIY: Predicting activity from phone accelerometers

Overview. Accelerometer data may become more common place in the future. In many respects, one's regular movements like walking and running can be viewed as a finger print. Thus, accelerometer data has many uses from measuring and improving health outcomes to multi-factor authentication[3] and roadway. quality[4]

It is quite easy to record and use the data. For this DIY, we use data collected on an iPhone at 20 hertz, or 20 recordings per second and will attempt to build our own activity classification algorithm that are calibrated to a specific person's motions. There are two pre-partitioned data frames – one for training (n = 311174) that was captured under controlled conditions and an extended test sample taken in non-controlled conditions (n = 35486).

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

Our primary target is the `activity` variable that includes labels for *idle*, *run*, *stairs*, and *walk*. For a real-world application, the data should be augmented with other activities as algorithms only know what they have previously seen.

```
#Quick summary
  table(train$activity)
```

```
##
##   idle    run stairs   walk
## 193408  21063   7733  88970
```

Smartphone accelerometers captures data along three axes. As the position of the phone can shift, data from any one axis may not be useful without considering measures on orientation. While rotation is captured by the sensors, we simply acceleration into total acceleration $accel = \sqrt{\text{accel x}^2 + \text{accel y}^2 + \text{accel z}^2}$. But if we recall the acceleration chart, we could deduce that a single value of acceleration does not likely have enough context to make for an informed prediction. We augment this data by engineering a few additional variables, namely the average, maximum, minimum, and standard deviation of total acceleration for the last 2.5 seconds (or 50 measurements).

Training. There are a number of R implementations of decision trees, the most popular of which is the `rpart` library:

[3] https://ieeexplore.ieee.org/document/8090407
[4] https://www.wired.com/insights/2014/03/potholes-big-data-crowdsourcing-way-better-government/

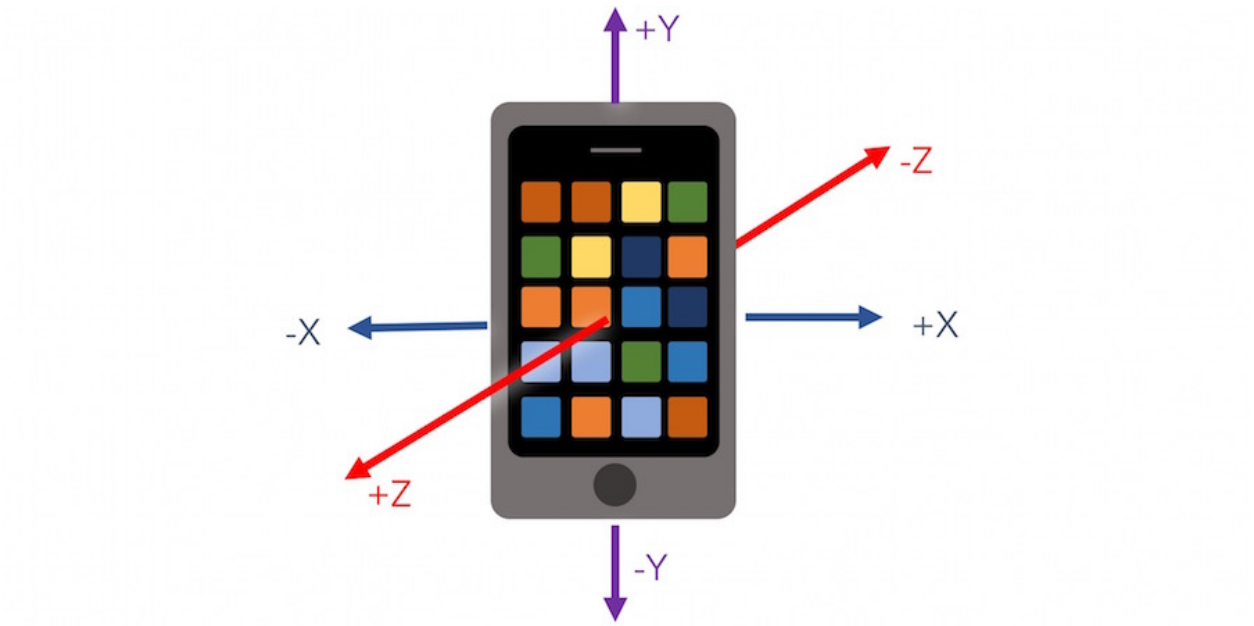


Figure 10: Accelerometer and its three axes.

```
library(rpart)
```

The main function within the library comes with flexible capabilities to induct 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()` setting `cp = 0`, meaning that the tree will be fully grown without any stopping criteria applied.

```
fit <- rpart(factor(activity) ~ accel + avg50 + max50 + min50 + sd50,
             method = "class", data = train,
             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`.

```
printcp(fit)
```

```
##             CP nsplit rel error    xerror         xstd
## 1 0.6944788819      0 1.0000000 1.0000000 0.0022973417
## 2 0.1768252297      1 0.3055211 0.3056655 0.0015150175
## 3 0.0021058710      2 0.1286959 0.1289082 0.0010203974
## 4 0.0011803067      4 0.1244841 0.1250955 0.0010059564
## 5 0.0009170728      6 0.1221235 0.1238303 0.0010011078
## 6 0.0005745858      7 0.1212065 0.1221830 0.0009947517
```

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[5]. 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 = 1023, nsplit = 1028 and xerror = 0.0295416334086239. By applying the rule, the upper bound of acceptable error is xerror = 0.029542 + 0.000498 = 0.0300396752824418, xerror = 0.029542 + 0.000498 = 0.0300396752824418. As it turns out, the tree with nsplit = 982 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
  #

  #Find lowest error and associated xstd
   best.error <- min(fit$cptable[,4])
   best.sd <- fit$cptable[,5][which(fit$cptable[,4]==best.error)]

  #Pull CP closest to lower bound
   lower.bound <- best.error + best.sd
   opt.select <- fit$cptable[,1][which(fit$cptable[,4] <= 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')
```

[5]Hastie et. al (2001)

Evaluating performance. One of the fascinating aspect of decision trees is that it is interpretable in a very different way than logistic regression. In lieu of a thorough review of the learned rules, we may rely on a measure of variable importance, that is defined as follows:

REVISIT AT SOME POINT

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

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 other words, 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 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.

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

```
##    max50    avg50     sd50    min50    accel
## 144679.0 143317.8 139400.9 119687.1 117792.5
```

Unlike previous examples, there are four classes to predict, meaning that the AUC measure may be less effective in producing a singular summary. We consider the F1-statistic defined as:

$$F_1 = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}}$$

where precision $= \frac{TP}{TP+FP}$ and recall $= \frac{TP}{TP+FN} = \frac{TP}{Actual(+)}$. The score is out of 1, where a value of 1 is a very well-balanced prediction. To accommodate three or more classes, the Mean F1-Score is the weighted by each activity class k (e.g. run, walk):

$$\text{Mean}F_1 = \sum_{i=1}^{k} (\frac{n_k}{n} F_{1k})$$

These formulae effectively boil down to the following function:

```
meanF1 <- function(actual, predicted){
  # Mean F1 score function
  #
  # Args:
  #   actual = a vector of actual labels
  #   predicted = predicted labels
  #

  #Loop through each class and calculate components
  classes <- unique(actual)
  results <- data.frame()
  for(k in classes){
    results <- rbind(results,
                     data.frame(weight = sum(actual == k, na.rm = T)/length(actual),
                                precision = sum(predicted == k & actual == k)/sum(predicted == k, na.rm
                                recall = sum(predicted == k & actual == k, na.rm = T)/sum(actual == k,
  }
```

```
  results$score <- results$weight * 2 * (results$precision * results$recall) / (results$precision + resu
  return(sum(results$score, na.rm=T))
}
```

When applied to the function to the predictions (`pred.opt` and `pred.full`), we find that the mean F1-statistics reached 0.8530794 and 0.8527223 – not bad for a first cut, but certainly can benefit from extra attention.

3.4 Practice Exercise

Tune the decision tree to increase the mean F1-statistic by testing different windows for the average, maximum and standard deviation of `accel`.

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.[6] 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.

[6] Breiman (2001)

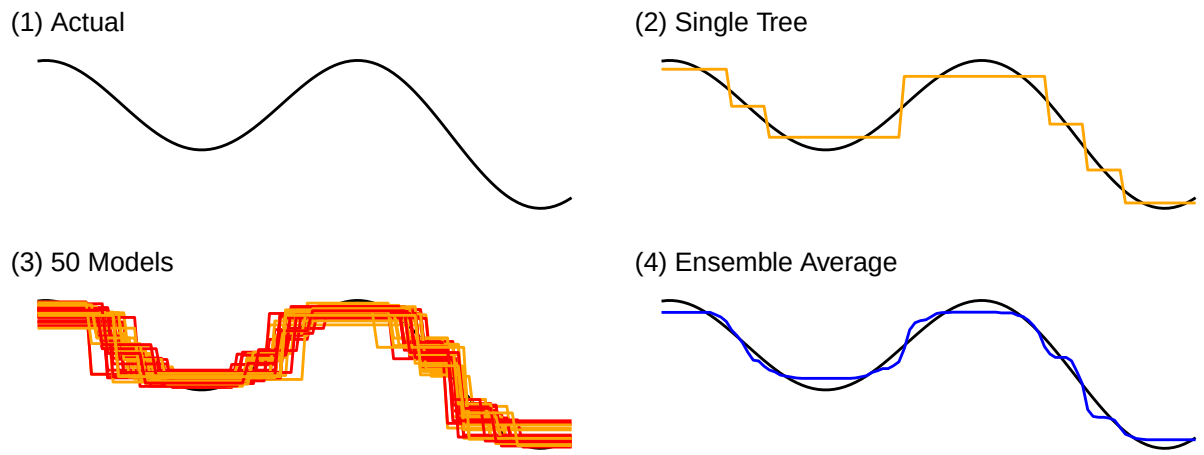


Figure 11: Comparison of results of applying a single model to fit a curve versus an ensemble of models.

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.

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

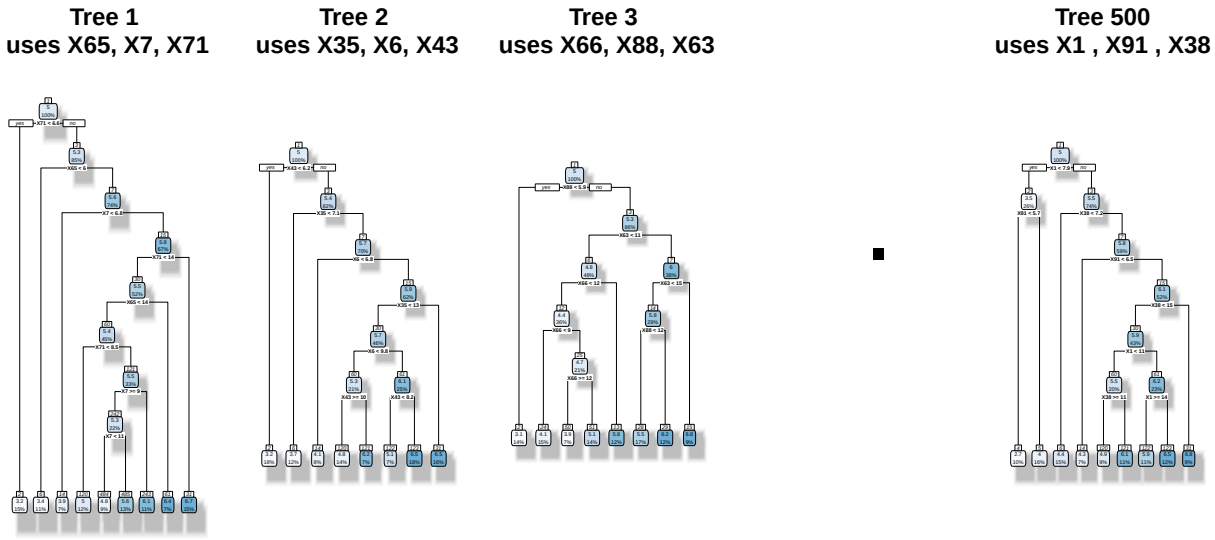


Figure 12: Random Forests construct hundreds of trees sampling from both observations and features, then combine the trees into one prediction through voting.

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.

4.2 DIY: Revisiting activity classification

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, mtry, ntree)`

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` 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
  pacman::p_load(randomForest)

#Run Random Forest
  spec <- as.formula("factor(activity) ~ accel + avg50 + max50 + min50 + sd50")
  fit.rf <- randomForest(spec, data = train, mtry = 2,
                         ntree = 200)
```

Using the `importance()` function, we can see the Mean Decrease Gini, which calculates the mean of Gini coefficients. However, the values themselves do not have any meaning outside of a comparison with other Gini measures.

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

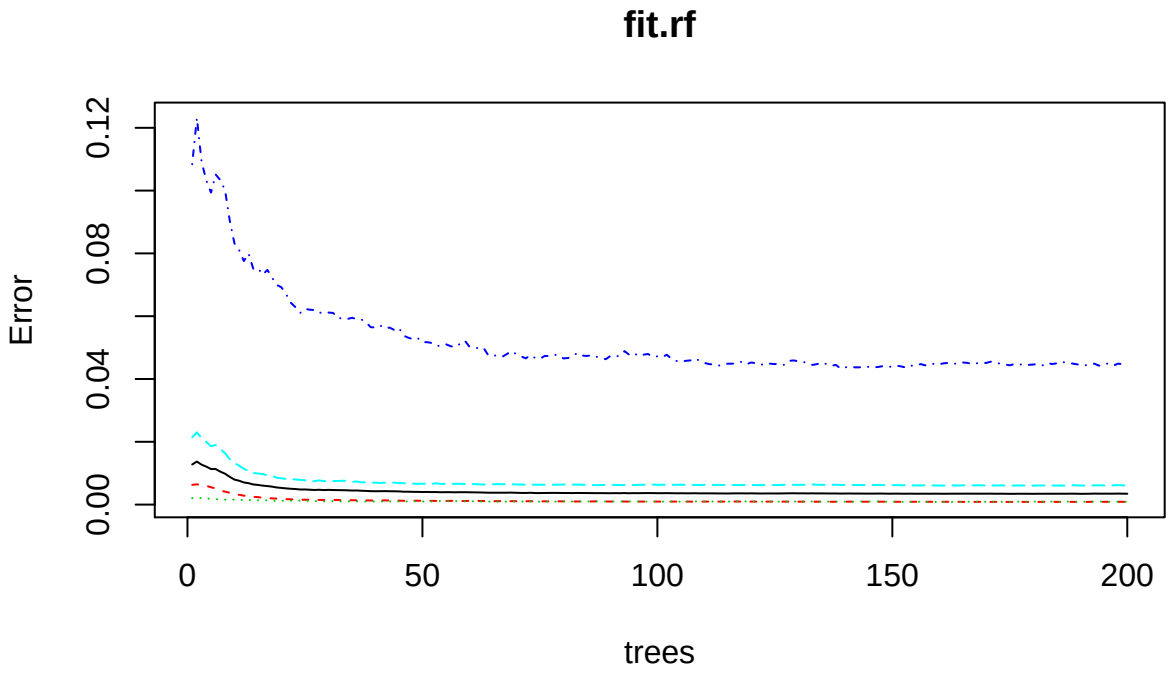
```
## Warning in kable_styling(., latex_options = c("hold_position")): Please
## specify format in kable. kableExtra can customize either HTML or LaTeX
## outputs. See https://haozhu233.github.io/kableExtra/ for details.
```

Table 6: Variable importance.

Variable	Increase Node Purity
avg50	59147.96
max50	40706.41
sd50	37940.55
min50	22554.48
accel	3588.40

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 tree are optimal.

```
#Search for optimal number of input features
  fit.tune <- tuneRF(x = train[,c("avg50", "max50", "min50", "sd50")],
                     y =  factor(train$activity),
                     ntreeTry = 500,  mtryStart = 1,
                     stepFactor = 2,  improve = 0.001,
                     plot = FALSE, trace = FALSE)
```

```
## 0.3352601 0.001
## -0.1507246 0.001
```

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

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)

#Calculate mean F1
  meanF1(test$activity, yhat)
```

```
## [1] 0.8509137
```

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.