Lecture 8: Classifiers

Intro to Data Science for Public Policy, Spring 2016

by Jeff Chen & Dan Hammer, Georgetown University McCourt School of Public Policy

Contents

Building on Lecture 7, this section lightly introduces three more classification algorithms: Logistic Regression, Support Vector Machines, and Artificial Neural Networks. Logistic regression is a probabilistic cousin of Ordinary Least Squares known in a class of techniques known as Generalized Linear Models. In most of the social sciences, this is the de facto technique for modeling discrete outcomes as the weights can be loosely interpretted as point estimates of relationships. Support Vector Machines (SVMs) is a geometry-inspired technique that fits a hyperplane (an n-dimensional plane). The method does not have an interpretation, but is often times hyper accurate. Typically SVMs are used for highdimensional with non-linear relationships and is a light weight method for conducting computer vision tasks. Artificial Neural Networks (ANNs) are the foundational technique for what is now known as 'deep learning' and 'artificial intelligence'. ANNs also do not lend themselves to direct interpretation and are computationally costly to build well.

Before diving into the mechanics of these methodologies, we will load in the health data from Lecture 7 and partition the data into train-test samples.

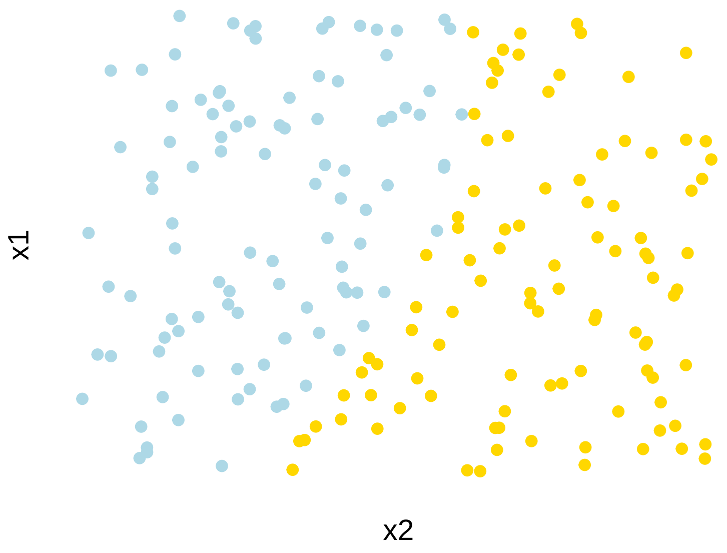
```r
  health <- read.csv("data/lecture8.csv")

#Create index of randomized booleans of the same length as the health data set
  set.seed(100)
  rand <- runif(nrow(health))
  rand <- rand > 0.5

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

Logistic Regression

Let's assume that you've been provided with a three feature dataset: a target label z and two input features ($x1$ and $x2$). Upon graphing the features and color coding using the labels, you see that the points are clustered such that light blue points represent to $z = 1$ and gold points represent $z = 0$. We could, of course, use decision trees and random forests to determine some threshold to classify the two groups; but surely, there is a way to write an elegant statistical formula that would separate one group from the other?



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

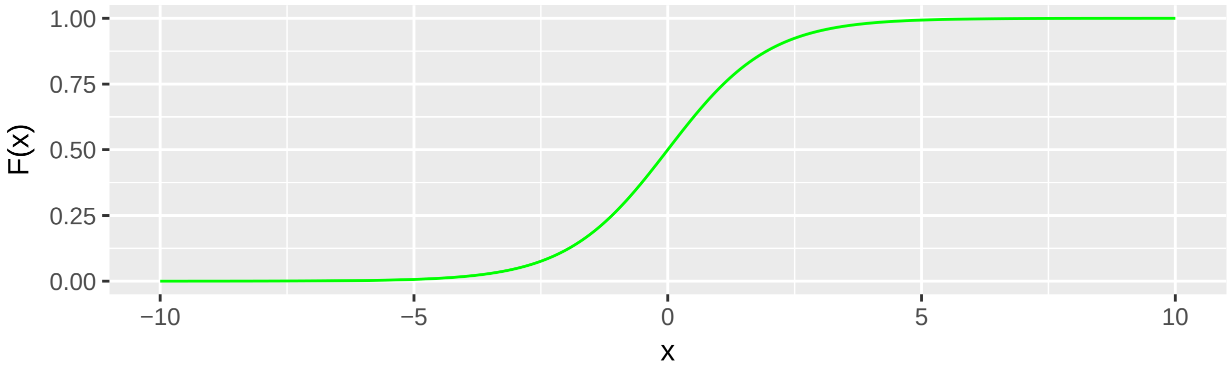
$$z = w_0 + w_1 x_1 + w_2 x_2 + \epsilon$$

where z is a binary outcome and, like OLS, w_k are weights that are learned using some optimization process. If treated as a typical linear model with a continuous outcome variable, we run the risk that $\hat{z}$ would exceed the binary bounds of 0 and 1 and would thus make little sense. Imagine if $\hat{z}$, the predicted value of z were -103 or +4: what would that mean in the case of a binary variable? This could easily be the shortcoming of a linear model approach.

Statistical methodologists have, however, cleverly solved the bounding problem by inserting the predicted output into a logistic function:

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

For a feature x that ranges for -10 to +10, the logit transformation converges to +1 where $x > 0$ and to 0 where $x < 0$. This S-shaped curve is known as a *sigmoid* and is a well-used distribution for bounding variables to a 0/1 range.



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

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

To interpret the coefficients, we need to start by defining what *odds* are:

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

where $F(z)$ is a probability of some event $z = 1$and $1 - F(z)$ is the probability of $z = 0$. The odds can be re-formulated as:

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

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

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

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

The underlying weights of the logistic regression can be interpretted using *Odds Ratios* or *OR*. Odds ratios can be expressed as marginal unit comparison. Since $odds = e^z = e^{w_0+w_1x_1+w_2x_2}$, then we can express an odds ratio as a marginal 1 unit increase in x_1 comparing $odds(x+1)$ over $odd(x+0)$:

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

After a little exponential arithmetic, the OR is simply equal to e^{w_1}, which can be interpreted as a multiplicative effect or a percentage effect if transformed as $100 \times (1 - e^{w_1})\%$. In practice, this means simply exponentiating the regression weights to interpret the point relationship. For example, if the following regression were estimated for healthcare non-coverage where *wage* is a continuous variable and $non-citizen$ is a discrete binary:

$$z(\text{non-coverage}) = 0.1878 - 0.000001845 \times wage + 1.69 \times \text{non-citizen}$$

Then, the odds of coverage are as follows for each variable:

- $OR_{wage} = e^{0.000001845} = 0.9999816$ translates to -0.00000184% lower chance of not being covered.
- $OR_{non-citizen} = e^{1.690} = 5.419481$ translates to 441% higher chance of not being covered.

Optimization

As in the case of all machine learning methods, the formulae need to be optimized. In order to estimate each weight w_k, we will rely on *maximum likelihood estimation* (MLE) as a framework, starting with a probability function for one record that is inspired by a Bernoulli random variable:

$$p(z = z_i|x) = [F(x)]^{z_i}[1 - F(x)]^{1-z_i}$$

If $z_i = 1$, then the function is equal to the $[F(x)]^{z_i}$. Otherwise, if $z_i = 0$, then the function is equal to $[1 - F(x)]^{1-z_i}$. For all records, we can define a likelihood function as the product of the above:

$$L = \Pi_{i=1}^N[F(x)]^{z_i}[1 - F(x)]^{1-z_i}$$

Mathematically, it is easier to handle this formula by taking the natural logarithm, which is also known as the *log-likelihood*:

$$log(L) = z_i log(F(x)) + (1 - z_i) log(1 - F(x))$$

The goal here is to maximize $log(L)$, driven by a search for w_k by taking partial derivatives of L with respect to each w_k and setting them to zero:

$$\frac{\partial L}{\partial w_k} = 0$$

This process can be driven using optimization algorithms such as gradient descent, the Newton-Raphson algorithm, among other commonly used techniques.

Practicals

After all the derivation is done, keep the following points in mind when applying logistic regression:

- Tuning is largely a matter of feature selection: it all depends on the variables that are available.
- Logistic regression have strong probabilistic assumptions and assume that a linear combination of features is sufficient to describe a phenomenon.
- The technique is well-suited for socializing an empirical problem, but often is outperformed in accuracy by more flexible techniques such as Random Forests. This trade off between narrative and accuracy is a common example of the bias-variance trade off.

Applying Logistic Regression

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

At a minimum, four parameters are required:

- input features: `<x>`
- target feature: `<y>`
- data: `<data>`
- family: `<probability func>`

The family refers to the probability distribution family that underlies the specific estimation method. In the case of logistic regression, the probability family is *Binomial*.

```r
glm(<y> ~ <x>, data = <data>, family = <probability func>)
```

To start, we will calibrate a logistic regression where `coverage` is trained on all other features in the dataset. Like `lm()`, by specifying `.` as the input feature, all features other than the target are included in the model. We store the regression output in the objection `glm.fit`, then take a look under the hood using `summary()`.

```r
glm.fit <- glm(coverage ~ ., data = train, family = binomial())
summary(glm.fit)
```

```
##
## Call:
## glm(formula = coverage ~ ., family = binomial(), data = train)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -2.8971  -0.8858   0.2242   0.9230   3.5087
##
```

```
## Coefficients:
##                               Estimate Std. Error z value Pr(>|z|)    
## (Intercept)                  1.330e+00  4.806e-01   2.768 0.005648 ** 
## age                         -3.472e-02  1.634e-03 -21.255  < 2e-16 ***
## wage                        -1.307e-05  1.000e-06 -13.071  < 2e-16 ***
## citNon-citizen               1.598e+00  9.895e-02  16.150  < 2e-16 ***
## marMarried                  -8.360e-01  6.799e-02 -12.297  < 2e-16 ***
## marNever Married            -4.423e-01  7.684e-02  -5.756 8.59e-09 ***
## marSeparated                 1.648e-01  1.429e-01   1.154 0.248559    
## marWidowed                  -1.076e+00  1.228e-01  -8.766  < 2e-16 ***
## educHS Degree                1.302e+00  1.127e-01  11.556  < 2e-16 ***
## educLess than HS             1.669e+00  1.211e-01  13.781  < 2e-16 ***
## educUndergraduate Degree     4.855e-01  1.263e-01   3.844 0.000121 ***
## raceAsian                   -1.113e+00  4.735e-01  -2.350 0.018752 *  
## raceBlack                   -2.030e-01  4.584e-01  -0.443 0.657888    
## raceNat. Hawaiian/Pac. Isl. -2.407e+00  1.243e+00  -1.937 0.052714 .  
## raceOther                    6.174e-01  4.910e-01   1.257 0.208627    
## raceTribes Spec.            -3.012e-01  8.756e-01  -0.344 0.730886    
## raceTwo or More             -2.176e-01  4.866e-01  -0.447 0.654685    
## raceWhite                   -4.159e-01  4.573e-01  -0.909 0.363113    
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
## 
## (Dispersion parameter for binomial family taken to be 1)
## 
##     Null deviance: 15364  on 11082  degrees of freedom
## Residual deviance: 12267  on 11065  degrees of freedom
## AIC: 12303
## 
## Number of Fisher Scoring iterations: 5
```

The coefficients can be extracted from the glm.fit object and exponentiated for ease of interpretation. For example, people who have not completed a high school degree are 5.3-times more likely to not have health care coverage compared with the reference group (graduate degree holders).

```r
exp(glm.fit$coefficients)
```

```
##                 (Intercept)                         age 
##                  3.78085375                  0.96587328 
##                        wage              citNon-citizen 
##                  0.99998692                  4.94306794 
##                  marMarried            marNever Married 
##                  0.43343657                  0.64253368 
##                marSeparated                  marWidowed 
##                  1.17921369                  0.34082790 
##               educHS Degree            educLess than HS 
##                  3.67791007                  5.30496830 
##    educUndergraduate Degree                   raceAsian 
##                  1.62500206                  0.32858152 
##                   raceBlack raceNat. Hawaiian/Pac. Isl. 
##                  0.81629072                  0.09004779 
##                   raceOther            raceTribes Spec. 
##                  1.85407609                  0.73994899 
##             raceTwo or More                   raceWhite 
##                  0.80441460                  0.65973652
```

In order to obtain the predicted values of *coverage*, we use `predict()`. Note that three types of values can be obtained using `predict()` via the `type =` argument:

- `terms`: For each record i, `terms` returns the product of each w_k with the corresponding input feature x_k.
- `link`: For each record, the sum of `terms` is returned.
- `response`: For each record, the `link` output is transformed using the logit link.

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

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

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

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

```
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
## 0.0003694 0.2944000 0.5222000 0.5027000 0.7136000 0.9855000
```

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

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

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

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

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

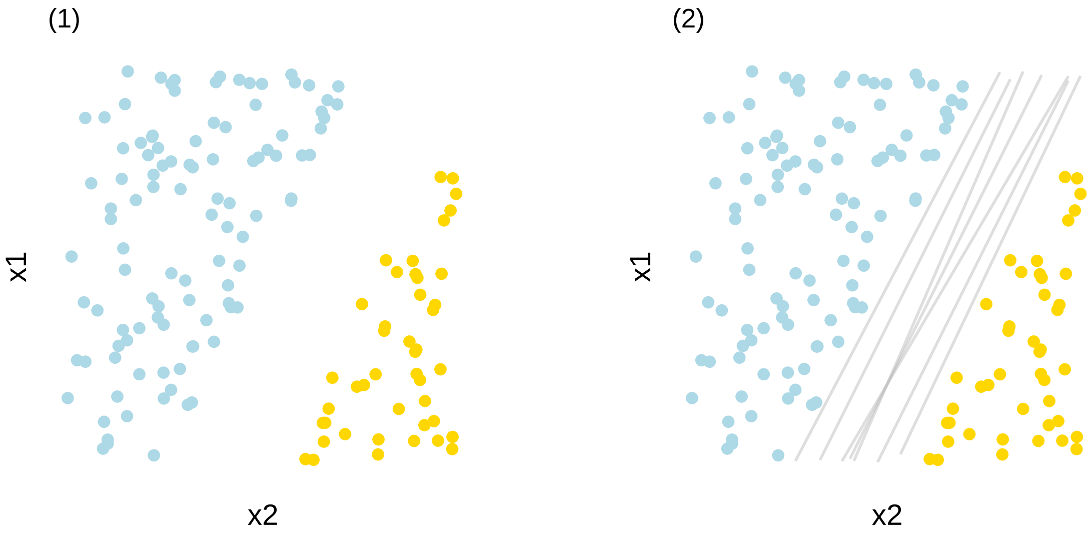
```
##   group       AUC
## 1     1 0.7875094
## 2     2 0.7926148
```

Support Vector Machines

Logistic regression is a probabilistic approach. The linear formulation allows for ease of interpretation and is thus a technique of choice in many fields for general applications. But the predictive accuracy may be a whole magnitude lower relative to other methods. Support Vector Machines, on the other hand, take a purely geometric approach to classification. The technique often yields relatively higher accuracy, at the expense of interpretation. For technical tasks that involve organic relationships such as computer vision or genetic research, SVMs are particularly adept at pattern recognition and classification. It should be noted that it is due to the highly mathematical nature of SVMs in addition to the computational requirements that the technique is typically used for tasks where social interpretation is not required.

Building upon the same three feature dataset once more, let’s assume this time when data are plotted, there is a clear gap between groups such that a straight line can partition one group from the other (Figure 1). A line as simple as $wx + b = y$ may do the trick in two dimensional space, but can also be described as a plane in n-dimensional space $w^T + b = y$. That line may then serve as a boundary between the two groups where $w^T + b > y$ may describe the group above the boundary and $w^T + b < y$ describes the group below.

Given the space, however, you realize that multiple lines could do the job: there are almost infinite lines (Figure 2) that could serve as the boundary between the groups. But which is the best? There should, in theory, be one line that optimally describes the separation between the groups.



Classification

If we are to assume a straight line is appropriate, we can find a line that maximizes the distance between the groups. To intuit distance requires defining points of reference. Let's then assume that there exists two parallel planes: each sits at the edge of each respective group and the space, labeling the top plane as $y = +1$ and bottom plane as $y = -1$. As seen in Figure 3, the dashed grey lines and the solid purple lines are *hyperplanes*, but are simply lines in two dimensional space. H1 ($y = +1$) and H2 ($y = -1$) are hyperplanes that are defined by a set of "support vectors" – points that serve as control or reference points for the location of the hyperplane (see Figure 4). The elegance of this method is that not all points in a dataset are used to define H1 and H2: only select points on or near the hyperplanes are required to define the plane. These planes are defined using simple linear equations shown in dot-product form:

$$w^T x - b = +1$$

and

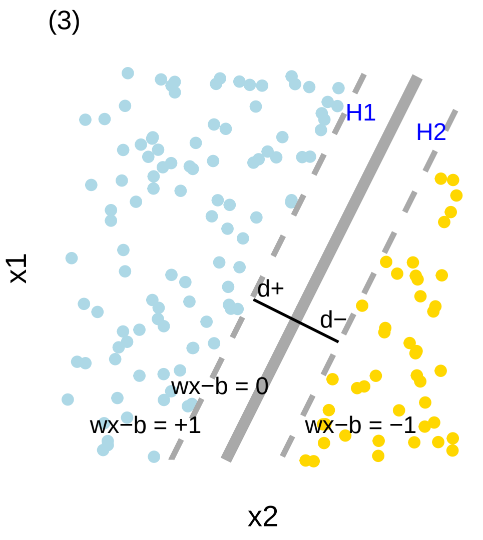
$$w^T x - b = -1$$

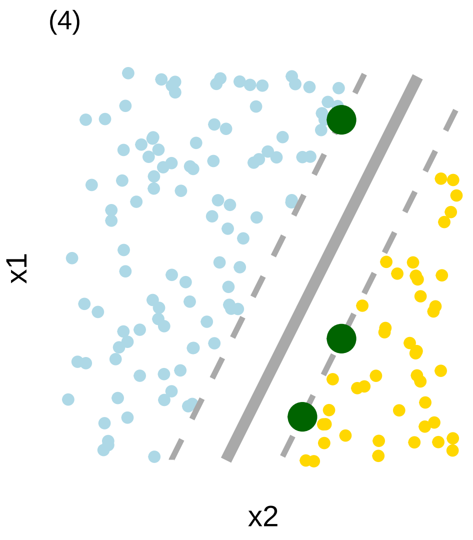
for H2, where w is a weight that needs to be calibrated. H1 and H2 primarily serve as the boundaries of what is known as the *margin*, or the space that maximally separates the two classes that are linearly separable The optimal hyperplane or *decision boundary* is defined as

$$w^T x - b = 0$$

and sits at a distance of $d+$ from H1 and $d-$ from H2.

When H1, H2, and the decision boundary are determined through training, scoring essentially maps where a new record falls in the decision space. A point to the left of H1 is scored as +1 and to the right of H2 is −1. Note that thus far, a point that falls in between H1 and H2 is not considered.





Learning Function

To tune a SVM, we want to find the maximum distance between H1 and H2. This can be done by finding the distance of the line that is perpendicular to H1 and H2 since they are parallel. The following equations are the points at which the perpendicular line intersects at two points:

$$w_1^T + b = 1$$

and

$$w_2^T + b = -1$$

By subtract the two equations, we obtain $w^T(x_1 - x_2) = 2$, which then can be manipulated by dividing the normalized w vector $||w||$ of the weights. This yields a distance formula for the *margin*:

$$\text{margin} = x_1 - x_2 = \frac{2}{||w||}$$

To maximize the margin in its current form may be challenging and is typically reformulated as a minimization problem that can be solved using quadratic programming:

$$min\frac{1}{2}||w||^2$$

subject to $y_i(w^T x_i + b) \geq 1$ for all records in the sample. Like gradient descent and Newton Raphson, these are problems that have standard implementations that are pre-packaged in R in the `e1071` library.

For the sake of exposure, the learning function for w is maximized using the following formulation:

$$w(\alpha) = \sum_i \alpha_i - \frac{1}{2}\sum_i \alpha_1 \alpha_0 y_1 y_0 x_1^T x_0$$

subject to $\alpha_i \geq 0$ (non-negatives), $\sum_i \alpha_i y_i = 0$ (sum of alpha and y are equal to zero). Otherwise stated: the equation is the sum of all points i minus the product of alphas, labels, values. α are parameters that are being tuned in order to maximize w. An interesting observation of this formula is that since the hyperplanes H1 and H2 sit on the edge of their respective groups, the hyperplane will only intersect with only a few records or "vectors". Mathematically, many of the α values will be zero. Intuitively, that means that the optimization equation will retain only a fraction of the total vectors to support the calculation on the plane. This is the origin of the name of the method: only vectors that support the planar calculation are retained.

Upon maximizing w, a vector of w containing the weights associated with each feature can be extracted

$$w = \sum_{i}^{N} \alpha_i y_i x_i + b$$

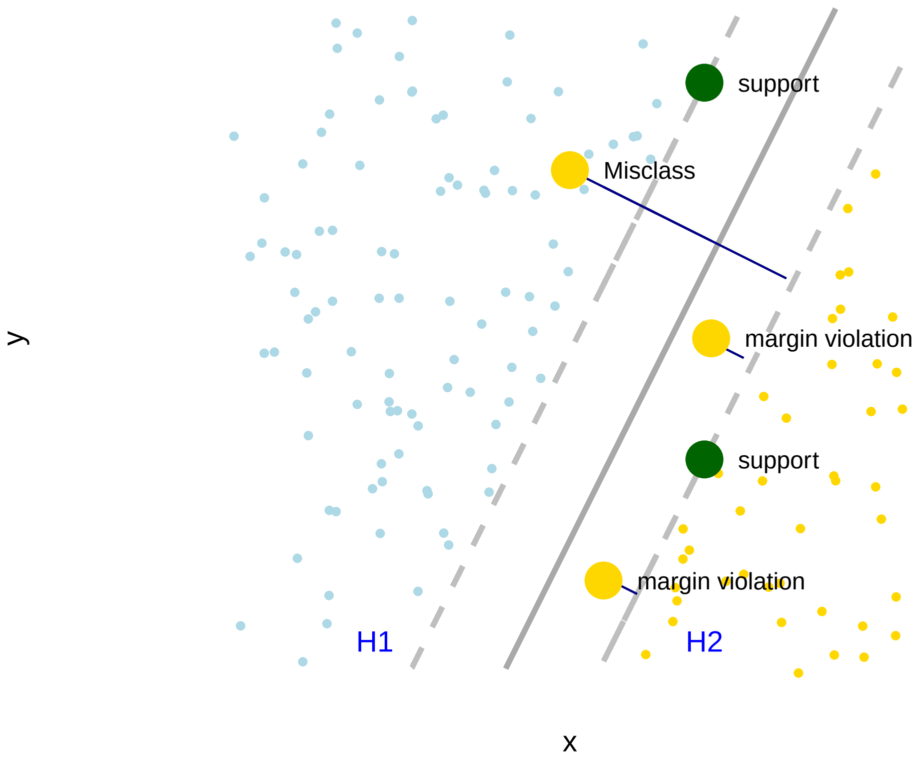
which in turn can be used to solve a planar equation to find the corresponding value of b to define the plane. While there are weights in this method, they are not directly interpretable in the way as logistic regression, but the magnitude of the underlying weights correspond to the importance of each feature.

In Actuality

The first example provided is what is know as a *hard margin*, where classes are linearly separable. In actuality, most classification problems do not have a clear margin between classes, meaning that there may be points that are misclassified or lie in the margin. A *soft margin* formulation is more commonly used to handle cases where there is some fuzziness in the separation: the margin must be determined allowing for misclassification of points. We can characterize the position of challenging-to-classify points using *slack variables*, or a variable ξ that represents distance from the margin to a point.

Figure 6 illustrates a number of commonly observed scenarios:

- The green points are the support vectors that sit on H1 and H2, which are $\xi = 0$.
- The distance from each H2 and H1 to the decision boundary is $\frac{1}{||w||}$.
- The large gold points sit between H0 and H2 such that $0 \leq \xi \leq \frac{1}{||w||}$. While they still are correctly classified (correct side of the decision boundary), the points sit within the margin. These points are referred to as *margin violations*.
- The large blue point is a misclassified record as it is to the left of H1, but should be to the right of H2. In terms of slack distance, $\xi > \frac{2}{||w||}$ as its distance from the correct hyperplane is greater than the width of the margin.



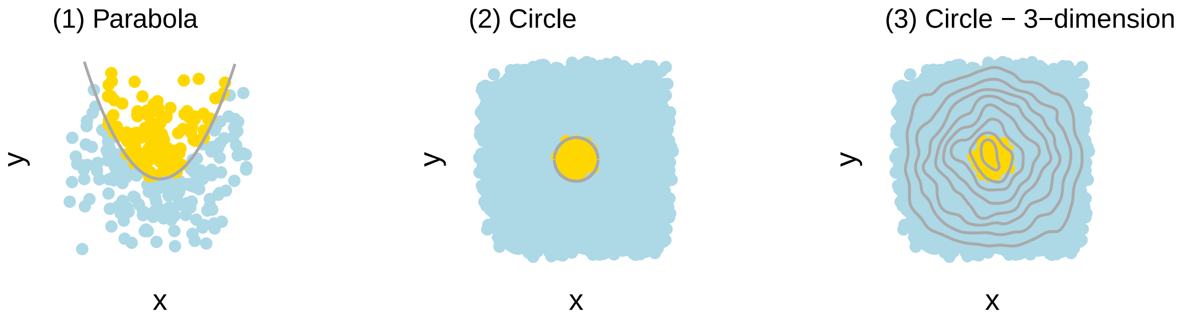
What does this mean for optimizing the margin? The slack variables need to be accounted for in the optimization of $||w||$:

$$min\frac{1}{2}||w||^2 + C\sum_{i}^{N}\xi_i$$

subject to $y_i(w^Tx_i + b) \geq 1 - \xi_i$ for all records in the sample. The first half of the formula is the same as the hard margin formula. The second half adds a constraint where the new variable C is known as a regularization variable or the *Cost*. If C is small, the slack variables are ignored and thus allows for larger margins. If C is large, then the slack variables reduce the size of the margin. It is worth noting that C is one of two tuning parameters that data scientists will need to calibrate when running SVMs.

Extending the hypothesis space

So far, the examples have focused on linear problems with hard and soft margins in two dimensional space. What if classes are clearly separated in a parabolic (1) or circular pattern (2)? A parabolic separation between classes can be described in terms of polynomials (e.g. $y = x^2$). A circular pattern may actually be separable if points are projected into higher dimensional space. Moving from two-dimensions (2) to three-dimensions (3), the contour lines demonstrate that there may be some threshold of the third feature at which a hyperplane can separate the two classes. The projection of records into higher dimensional space to improve separability is known as the *kernel trick*.



In a paper by Boser et al. (1992) modified the maximization function:

$$w(\alpha) = \sum_i \alpha_i - \frac{1}{2}\sum_i \alpha_1\alpha_0 y_1 y_0 x_1^T x_0$$

such that the dot products $x_1^T x_0$ are replaced with non-linearkernel functions. Of particular significance are two common kernels: the Gaussian Radial Basis Function (RBF) and Polynomial kernels.

RBP is defined as:

$$RBF = exp(-\gamma||x_1 - x_0||^2)$$

where $\gamma = \frac{1}{2\sigma^2}$ and σ >0. The value of γ determines the tightness of the kernel, where larger values of γ yield a compact, tight kernel whereas smaller values of γ are associated with wider-spread kernels. In `R`, the value of γ is one of the tuning parameters that a data scientist would need to specify as RBFs are the default kernel. Note that one needs to use a grid search to find the appropriate value of γ as it cannot be mathematically optimized, but rather analyzed.

The polynomial kernel is defined as:

$$Polynomial = (1 + x_1^T x_0)^d$$

where the value of $d > 0$, indicates the polynomial degree, and assumes that all polynomials from 1 to d are included.

Practical Details

After all the derivation is done, keep the following points in mind when applying SVMs:

- Tuning is centered on two variables: C to manage the extent to which the margin is hard or soft, and γ for when a RBF is applied. Note that the quantities of each are tuned using cross-validation in the form of a grid search (e.g. test multiple values at equal intervals).
- Non-linear SVMs are computationally expensive. Very high dimensional data sets will likely take a long time to compute.
- SVMs are particularly well-suited for a pattern recognition, computer vision among other computationally challenging problems. While they may yield more accurate results than many other classifiers, the ability for data scientists to give social policy decision makers control over the story is limited.
- ROC and AUC may at times be challenging to calculate for SVM results. An alternative is to utilize the F1 statistic defined as:

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

Applying SVMs

SVMs are neatly packaged into an interface library called `e1071`. The library contains a suite of machine learning tools in addition to SVMs.

```
library(e1071)
```

Syntax is fairly simple and requires a minimum, six parameters are required:

- input features: `<x>`
- target feature: `<y>`
- data: `<data>`
- cost: `<C>` regularization parameter – needs to be grid searched. Default = 1.
- gamma: `<g>` determines the "tightness" of a RBF. Default =
- kernel: `<kernel>` may include one of four kerels including *linear*, *polynomial*, *radial*, and *sigmoid*. Default = radial basis.

```
  svm(<y> ~ <x>, data = <data>, cost = <C>, gamma = <g>, kernel = <kernel>)
```

To start, we will fit an SVM using a *radial* kernel assuming $cost = 1$ and $gamma = \frac{1}{\text{dim}}$, where *dim* is the number of effective variables in our data (e.g. continuous variables, expanded dummies, and intercept). This effectively is 18 in the health data.

```
  svm.rbf.fit <- svm(coverage ~ ., data=train, kernel = "radial", cost = 1, gamma = 0.05555)
```

Typically, it is a good idea to test various values of `cost` and `gamma`, though noting that this process for SVMs is computationally expensive (takes a long time), especially for RBF kernels. The `e1071` library provides a method `tune.svm()` to find the best `cost` and `gamma` (see below). In this example, we will manually tune to develop a sense of how calibration works in practice.

```
#Tune SVM
  tune <- tune.svm(coverage ~. ,
                   data = train,
                   kernel = "linear",
                   cost=10^(-1:2), gamma=c(.5,1,2))
```

To determine search for the best parameters, we will conduct a grid search: a combination of four values of `cost` and four values of `gamma` will be tested for a total of 16 models. We choose equally spaced values on on an exponential scale (e.g. 0.01, 1, 10) to emphasize differences in model fit. To evaluate accuracy, we will rely on the F1-scores

```r
#F1 score
 meanf1 <- function(actual, predicted){
    #Mean F1 score function
    #actual = a vector of actual labels
    #predicted = predicted labels

    classes <- unique(actual)
    results <- data.frame()
    for(k in classes){
      results <- rbind(results,
                       data.frame(class.name = k,
                                  weight = sum(actual == k)/length(actual),
                                  precision = sum(predicted == k & actual == k)/sum(predicted == k),
                                  recall = sum(predicted == k & actual == k)/sum(actual == k)))
    }
    results$score <- results$weight * 2 * (results$precision * results$recall) / (results$precision + re
    return(sum(results$score))
 }

#Prep grid search
  cost.vec <- 10^(-1:2)
  gamma.vec <- 2^(seq(-5,2,2))
  combo <- expand.grid(cost = cost.vec, gamma = gamma.vec)
  combo$f1 <- NA

#Search
  for(i in 1:nrow(combo)){
    fit <- svm(coverage ~ ., data=train, kernel = "radial", cost = combo[i, 1], gamma = combo[i, 2])
    pred <- predict(fit, train)
    combo$f1[i] <- meanf1(train$coverage, pred)
  }

#View table
  print(combo)
```

```
##     cost   gamma        f1
## 1    0.1 0.03125 0.7385791
## 2    1.0 0.03125 0.7440702
## 3   10.0 0.03125 0.7471208
## 4  100.0 0.03125 0.7568490
## 5    0.1 0.12500 0.7434940
## 6    1.0 0.12500 0.7548671
## 7   10.0 0.12500 0.7624759
## 8  100.0 0.12500 0.7738608
## 9    0.1 0.50000 0.7533451
## 10   1.0 0.50000 0.7713171
## 11  10.0 0.50000 0.7807074
## 12 100.0 0.50000 0.7923816
## 13   0.1 2.00000 0.7683156
## 14   1.0 2.00000 0.7864243
## 15  10.0 2.00000 0.8007152
## 16 100.0 2.00000 0.8117518
```

Based on the grid search, we find that the best model has a $C = 100$ and $gamma = 2$. We then train a model with those parameters, then predict the classes of the test set to find that a $F1 = 0.75$.

```
#Predict labels
  pred.test <- svm(coverage ~ ., data = train, kernel = "radial", cost = 1, gamma = 8)
  pred.rbf <- predict(pred.test, test)

#examine result
  table(pred.rbf)
```

```
## pred.rbf
##    Coverage No Coverage
##        5477        5794
```

```
##RBF
  meanf1(test$coverage, pred.rbf)
```

```
## [1] 0.7456706
```

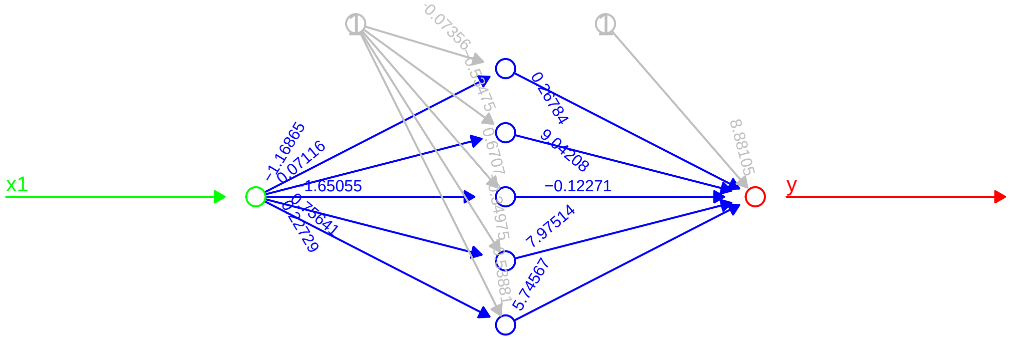
Artificial Neural Networks

ANNs have gained much notoriety over the years, especially with the advent of "artificial intelligence". With origins in pattern detection methodologies in the 1940s and later implemented in the form of algorithms such as the perceptron in the 1950's, ANNs fluctuated into and out of popularity over the better part of the 80 years. While this is a hot topic, we will only cover it from a high level as the technical requirements are notably demanding.

The technique is inspired by biological neurons that are located in the brain. In biology, neurons are part of a vast network of brain cells that store and transmit information. Information may be passed from one neuron to another in the form of an electrical and chemical impulses, and neurons are connected with many other neurons. Given a neuron that needs to transmit information to another neuron, the eletrical impulse needs to be sufficiently large to overcome inhibitory effects of the gap junction between neurons.

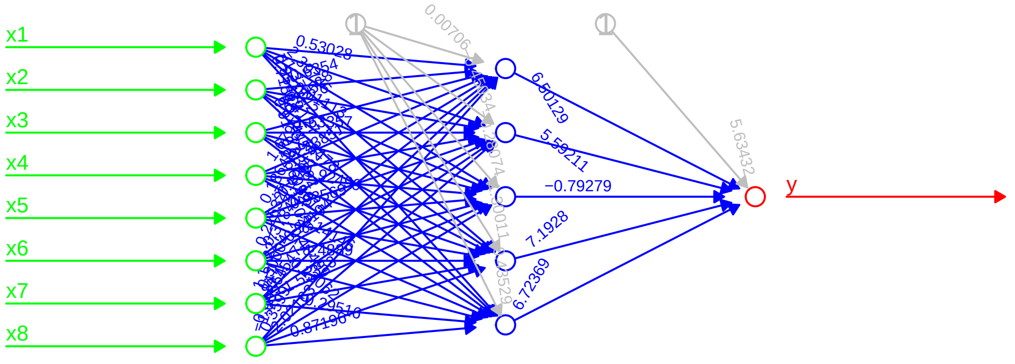
Similarly, ANNs are comprised of *nodes* that are organized into *layers* such that all nodes in each layer feed into all other neurons. The diagram below lays out a simple one layer neural network that follows the formulation: $Y = f(x_1)$:

- *Input*. Starting from the left, the green node is the input node (synapse). This is simply X_1.
- *Hidden layers*. The input node is linked to the blue circles, which are five nodes that comprise a hidden layer. The grey node above the hidden layer contains the intercepts for equations leading into that layer. Each hidden node is associated with an *activation function* that is based on a weighted value of X_1. The activation function is analogous to what determines if an electrical impulse fires strongly enough to transmit a recognizable signal, except in ANNs, some value is always transmitted. Like other machine learning algorithms, the weight is learned from examples of y. For example, the equation of the top blue node is $H1 = -0.073556019136 - 1.168651424424 \times x_1$. As each of the equations feeds into another, we find that hidden layers are used to handle cases where classes are not linearly acceptable as hidden layers are essentially a non-linear series of equations.
- *Output*. The red node y accepts the results of each of the formulae produced in the hidden layer.



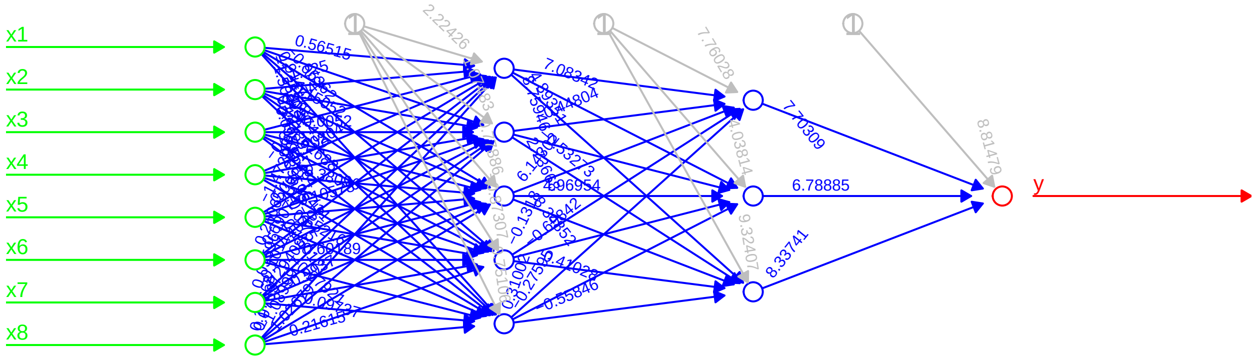
Error: 52.247128   Steps: 103

As more variables are added, more input nodes are added to the left. The case below, for example, illustrates an ANN where $Y = f(x_1, ..., x_8)$.



Error: 52.247128   Steps: 85

A common strategy to help improve capture of non-linear patterns is to add an additional hidden layer. However, the additional hidden layers may not necessarily yield significant accuracy gains. Notice that the number of activation functions that are captured in this ANN with 8 hidden nodes. In the past, if an ANN required hundreds of inputs and hundreds of hidden nodes, the computation required was unwieldy and technique was less favored due to the required computation. However, advances in computing technology now allow for faster training of sophisticated and complex ANNs.



Error: 52.247128   Steps: 107

For further reading, please take a look at this text that is focused on Neural Network Design.