CS M148 –

Data Science Fundamentals

Lecture #11: SVM & Classification Trees

Baharan Mirzasoleiman

UCLA Computer Science

Announcements

HW 2

- Is posted, due Wed Feb 16 at 2pm

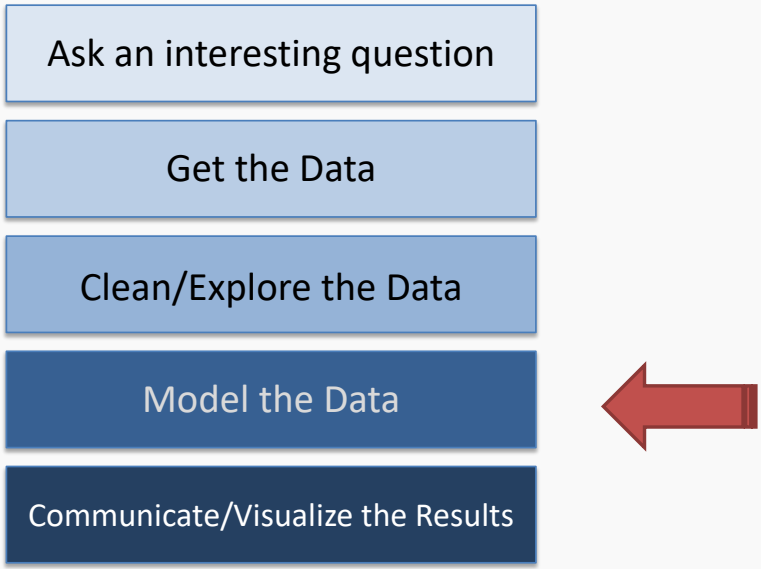
Lecture and discussions

- Online this week
- **Back to in person from Feb 14**

Let’s quickly review what we saw last time

Still here!

The Data Science Process



Classification

Outline

- Support Vector Machines (SVMs)
- Classification Trees

Support Vector Machines (SVMs)

Outline

- Classifying Linear Separable Data
- Classifying Linear Non-Separable Data
- Kernel Trick

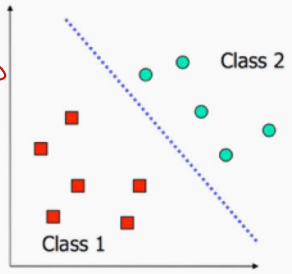
Decision Boundaries Revisited

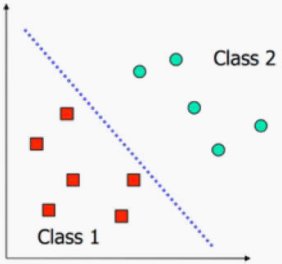
In logistic regression, we learn a ***decision boundary*** that separates the training classes in the feature space.

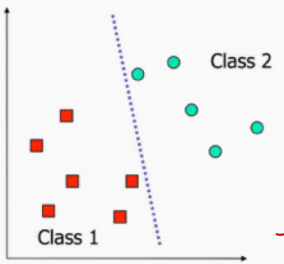
When the data can be perfectly separated by a linear boundary, we call the data ***linearly separable***.

In this case, multiple decision boundaries can fit the data. How do we choose the best?

large margin from boundary is good







may not generalize as well

Question: What happens to our logistic regression model when training on linearly separable datasets?

Decision Boundaries Revisited (cont.)

Constraints on the decision boundary:

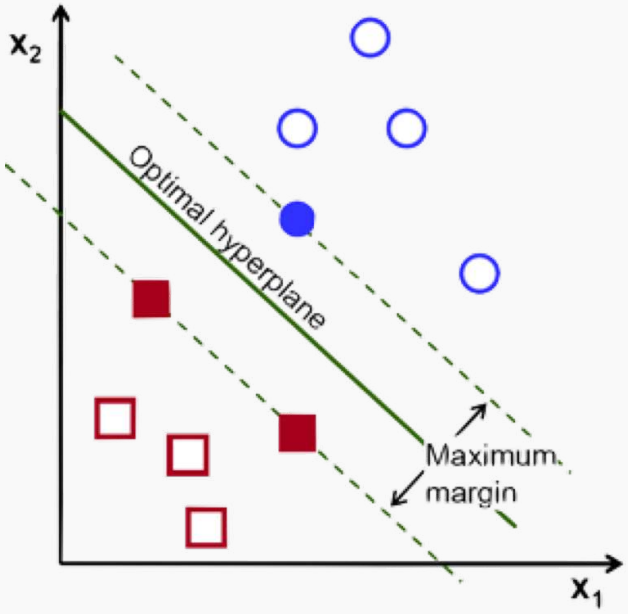
- In logistic regression, we typically learn an $\ell 1$ or $\ell 2$ regularized model.
- So, when the data is linearly separable, we choose a model with the 'smallest coefficients' that still separate the classes.
- The purpose of regularization is to prevent overfitting.

Decision Boundaries Revisited (cont.)

Constraints on the decision boundary:

- We can consider alternative constraints that prevent overfitting.
- For example, we may prefer a decision boundary that does not ‘favor’ any class (esp. when the classes are roughly equally populous).
- Geometrically, this means choosing a boundary that maximizes the distance or ***margin*** between the boundary and both classes.

Illustration of an SVM

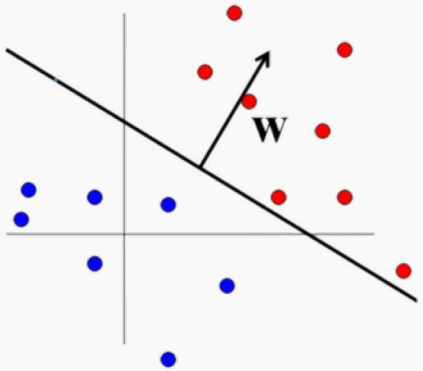


Geometry of Decision Boundaries

Recall that the decision boundary is defined by some equation in terms of the predictors. A linear boundary is defined by:

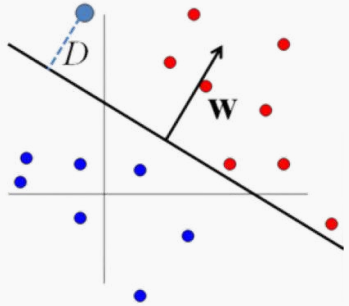
$$w^\top x + b = 0 \text{ (General equation of a hyperplane)}$$

Recall that the non-constant coefficients, w, represent a ***normal vector***, pointing orthogonally away from the plane



Geometry of Decision Boundaries (cont.)

Now, using some geometry, we can compute the distance between any point to the decision boundary using *w* and *b*.



The signed distance from a point $x \in \mathbb{R}^n$ to the decision boundary is

$$D(x) = \frac{w^\top x + b}{\|w\|}$$ (Euclidean Distance Formula)

Maximizing Margins

Now we can formulate our goal - find a decision boundary that maximizes the distance to both classes - as an optimization problem:

$$\begin{cases} \max\limits_{w,b} M \\ \text{such that } |D(x_n)| = \frac{y_i(w^\top x_n + b)}{\|w\|} \geq M,\ n = 1, \ldots, N \end{cases}$$

where M is a real number representing the width of the 'margin' and $y_i = \pm 1$. The inequalities $|D(x_n)| \geq M$ are called ***constraints***.

The constrained optimization problem as present here looks tricky. Let's simplify it with a little geometric intuition.

Maximizing Margins (cont.)

Notice that maximizing the distance of ***all points*** to the decision boundary, is exactly the same as maximizing the distance to the ***closest points***.

The points closest to the decision boundary are called ***support vectors***.

For any plane, we can always scale the equation:

$$w^\top x + b = 0$$

so that the support vectors lie on the planes:

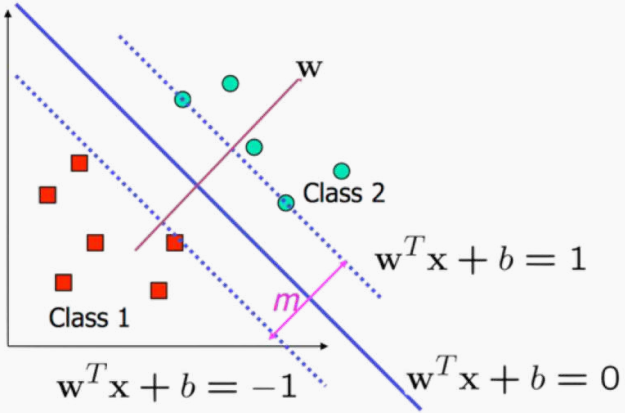
$$w^\top x + b = \pm 1,$$

depending on their classes.

Maximizing Margins Illustration

For points on planes $w^T x + b = \pm 1$, their distance to the decision boundary is $\pm 1/\|w\|$.

So we can define the ***margin*** of a decision boundary as the distance to its support vectors, $m = 2/\|w\|$.



Support Vector Classifier: Hard Margin

Finally, we can reformulate our optimization problem - find a decision boundary that maximizes the distance to both classes - as the maximization of the margin, *m*, ***while maintaining zero misclassifications***,

$$\begin{cases} \max\limits_{w,b} \dfrac{2}{\|w\|} \\ \text{such that } y_n(w^\top x_n + b) \geq 1,\ n = 1, \ldots, N \end{cases}$$

The classifier learned by solving this problem is called ***hard margin support vector classification***.

Often SVC is presented as a minimization problem:

$$\begin{cases} \min\limits_{w,b} \|w\|^2 \\ \text{such that } y_n(w^\top x_n + b) \geq 1,\ n = 1, \ldots, N \end{cases}$$

SVC and Convex Optimization

As a convex optimization problem SVC has been extensively studied and can be solved by a variety of algorithms:

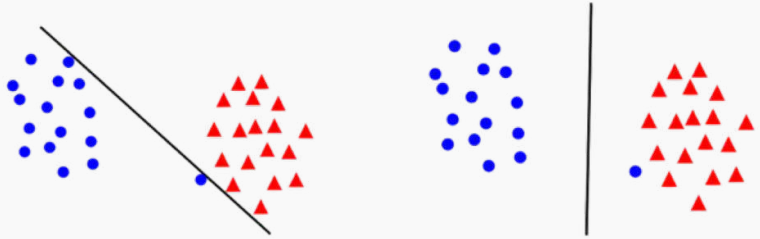
- **(Stochastic)** libLinear
  Fast convergence, moderate computational cost
- **(Greedy)** libSVM
  Fast convergence, moderate computational cost
- **(Stochastic)** Stochastic Gradient Descent Slow convergence, low computational cost per iteration
- **(Greedy)** Quasi-Newton Method
  Very fast convergence, high computational cost

Classifying Linear Non-Separable Data

Geometry of Data

Maximizing the margin is a good idea as long as we assume that the underlying classes are linear separable and that the data is noise free.

If data is noisy, we might be sacrificing generalizability in order to minimize classification error with a very narrow margin:



With every decision boundary, there is a trade-off between maximizing margin and minimizing the error.

Support Vector Classifier: Soft Margin

Since we want to balance maximizing the margin and minimizing the error, we want to use an objective function that takes both into account:

$$\begin{cases} \min\limits_{w,b} \|w\|^2 + \lambda \text{Error}(w,b) \\ \text{such that } y_n(w^\top x_n + b) \geq 1,\ n = 1, \ldots, N \end{cases}$$

where λ is an intensity parameter.

So just how should we compute the error for a given decision boundary?

Support Vector Classifier: Soft Margin (cont.)

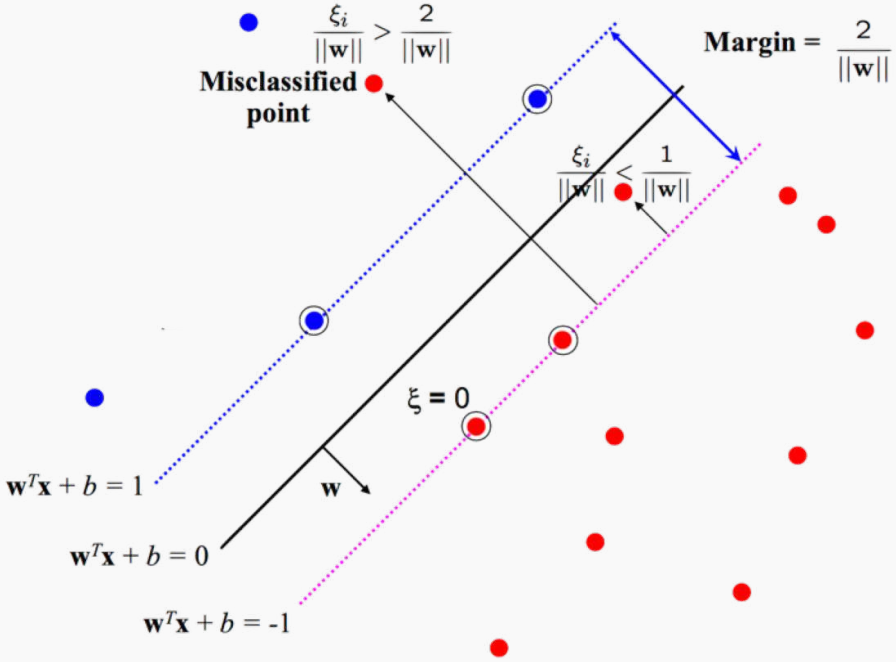
We want to express the error as a function of distance to the decision boundary.

Recall that the support vectors have distance $1/\|w\|$ to the decision boundary. We want to penalize two types of 'errors'

- **(margin violation)** points that are on the correct side of the boundary but are inside the margin. They have distance $\xi/\|w\|$, where $0 < \xi < 1$.
- **(misclassification)** points that are on the wrong side of the boundary. They have distance $\xi/\|w\|$, where $\xi > 1$.

Specifying a nonnegative quantity for ξ_n is equivalent to quantifying the error on the point x_n.

Support Vector Classifier: Soft Margin Illustration



Support Vector Classifier: Soft Margin (cont.)

Formally, we incorporate error terms ξ_n 's into our optimization problem by:

$$\begin{cases} \min\limits_{\xi_n \in \mathbb{R}^+, w, b} \|w\|^2 + \lambda \sum\limits_{n=1}^{N} \xi_n \\ \text{such that } y_n(w^\top x_n + b) \geq 1 - \xi_n, \ n = 1, \ldots, N \end{cases}$$

The solution to this problem is called ***soft margin support vector classification*** or simply ***support vector classification***.

Tuning SVC

Choosing different values for λ in

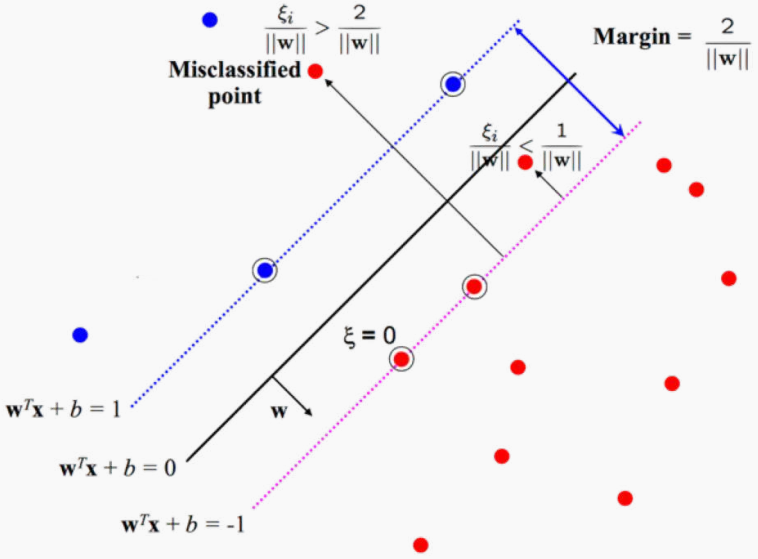
$$\begin{cases} \min\limits_{\xi_n \in \mathbb{R}^+, w, b} \|w\|^2 + \lambda \sum\limits_{n=1}^{N} \xi_n \\ \text{such that } y_n(w^\top x_n + b) \geq 1 - \xi_n, \ n = 1, \ldots, N \end{cases}$$

will give us different classifiers. In general,

- small λ penalizes errors less and hence the classifier will have a large margin
- large λ penalizes errors more and hence the classifier will accept narrow margins to improve classification
- setting $\lambda = \infty$ produces the hard margin solution

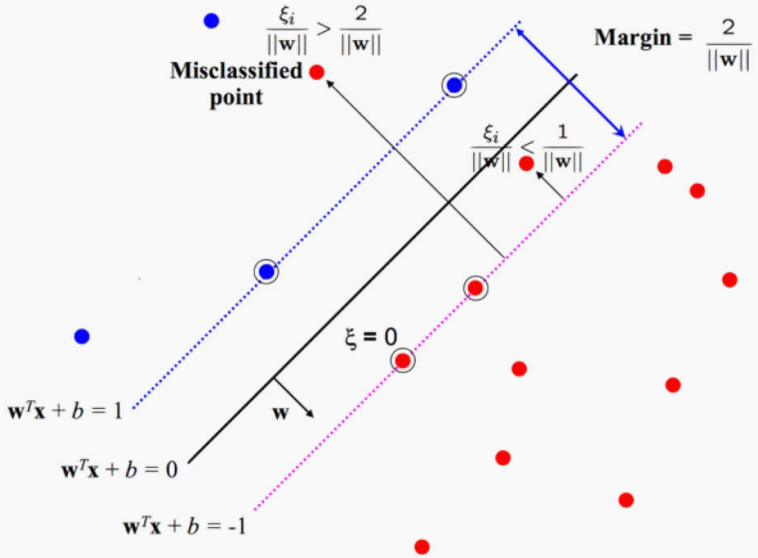
Decision Boundaries and Support Vectors

Recall how the error terms ξ_n's were defined: the points where $\xi_n = 0$ are precisely the support vectors



Decision Boundaries and Support Vectors

Thus to re-construct the decision boundary, ***only the support vectors are needed!***



Decision Boundaries and Support Vectors

The decision boundary of an SVC is given by

$$\hat{w}^\top x + \hat{b} = \sum_{x_n \text{ is a support vector}} \hat{\alpha}_n y_n (x_n^\top x\ ) + b$$

where $\hat{\alpha}_n$ and the set of support vectors are found by solving the optimization problem.

- To classify a test point x_{test}, we predict

$$\hat{y}_{test} = \text{sign}\left(\hat{w}^\top x + \hat{b}\right)$$

SVC as Optimization

With the help of geometry, we translated our wish list into an optimization problem

primal

$$\begin{cases} \min\limits_{\xi_n \in \mathbb{R}^+, w, b} \|w\|^2 + \lambda \sum\limits_{n=1}^{N} \xi_n \\ \text{such that } y_n(w^\top x_n + b) \geq 1 - \xi_n, \ n = 1, \ldots, N \end{cases}$$

where ξ_n quantifies the error at x_n.

The SVC optimization problem is often solved in an alternate form (the dual form):

$$\max_{\alpha_n \geq 0, \sum_n \alpha_n y_n = 0} \sum_n \alpha_n - \frac{1}{2} \sum_{n,m=1}^{N} y_n y_m \alpha_n \alpha_m x_n^\top x_m$$

Later we'll see that this alternate form allows us to use SVC with non-linear boundaries.

Extension to Non-linear Boundaries

Polynomial Regression: Two Perspectives

Given a training set:

$$\{(x_1, y_1), \ldots, (x_N, y_N)\}$$

with a single real-valued predictor, we can view fitting a 2nd degree polynomial model:

$$w_0 + w_1 x + w_2 x^2$$

on the data as the process of finding the best quadratic curve that fits the data. But in practice, we first expand the feature dimension of the training set

$$x_n \mapsto (x_n^0, x_n^1, x_n^2)$$

and train a ***linear model*** on the expanded data

$$\{(x_n^0, x_n^1, x_N^2, y_1), \ldots, (x_N^0, x_N^1, x_N^2, y_N)\}$$

Transforming the Data

The key observation is that training a polynomial model is just training a linear model on data with transformed predictors.

In our previous example, transforming the data to fit a 2nd degree polynomial model requires a map:

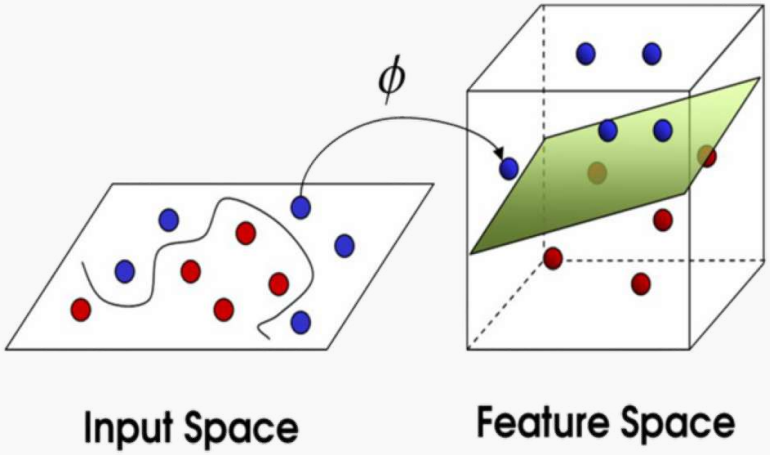
$$\phi : \mathbb{R} \to \mathbb{R}^3$$
$$\phi(x) = (x^0, x^1, x^2)$$

where $\mathbb{R}$ called the ***input space***, $\mathbb{R}^3$ is called the ***feature space***.

While the response may not have a linear correlation in the input space $\mathbb{R}$, it may have one in the feature space $\mathbb{R}^3$.

SVC with Non-Linear Decision Boundaries

The same insight applies to classification: while the response may not be linear separable in the input space, it may be in a feature space after a fancy transformation:



SVC with Non-Linear Decision Boundaries (cont.)

The motto: instead of tweaking the definition of SVC to accommodate non-linear decision boundaries, we map the data into a feature space in which the classes are linearly separable (or nearly separable):

- Apply transform ϕ: $\mathbb{R}^J \rightarrow \mathbb{R}^{J'}$ on training data

  $$x_n \rightarrow \phi(x_n)$$

  where typically *J*′ is much larger than *J*.
- Train an SVC on the transformed data

  $$\{(\phi(x_1), y_1), (\phi(x_2), y_2), \ldots, (\phi(x_N), y_N)\}$$

Inner Products

Since the feature space $\mathbb{R}^{J'}$ is potentially extremely high dimensional, computing ϕ explicitly can be costly.

Instead, we note that computing ϕ is unnecessary. Recall that training an SVC involves solving the optimization problem:

$$\max_{\alpha_n \geq 0,\, \sum_n \alpha_n y_n = 0} \sum_n \alpha_n - \frac{1}{2} \sum_{n,m=1}^{N} y_n y_m \alpha_n \alpha_m \phi(x_n)^\top \phi(x_m)$$

In the above, ***we are only interested in computing inner products*** $\phi(x_n)^T \phi(x_m)$***in the feature space*** and not the quantities $\phi(x_n)$ themselves.

The Kernel Trick

The ***inner product*** between two vectors is a measure of the similarity of the two vectors.

Definition

Given a transformation $\phi : \mathbb{R}^J \to \mathbb{R}^{J'}$, from input space $\mathbb{R}^J$ to feature space $\mathbb{R}^{J'}$, the function $K : \mathbb{R}^J \times \mathbb{R}^J \to \mathbb{R}$ defined by

$$K(x_n, x_m) = \phi(x_n)^\top \phi(x_m), \quad x_n, x_m \in \mathbb{R}^J$$

is called the ***kernel function*** of ϕ.

Generally, ***kernel function*** may refer to any function $K : \mathbb{R}^J \times \mathbb{R}^J \to \mathbb{R}$ that measure the similarity of vectors in $\mathbb{R}^J$, without explicitly defining a transform ϕ.

The Kernel Trick (cont.)

For a choice of kernel *K*,

$$K(x_n, x_m) = \phi(x_n)^\top \phi(x_m)$$

we train an SVC by solving

$$\max_{\alpha_n \geq 0,\ \sum_n \alpha_n y_n = 0} \sum_n \alpha_n - \frac{1}{2} \sum_{n,m=1}^{N} y_n y_m \alpha_n \alpha_m K(x_n, x_m)$$

Computing $K(x_n, x_m)$ can be done without computing the mappings $\phi(x_n)$, $\phi(x_m)$.

This way of training a SVC in feature space without explicitly working with the mapping ϕ is called ***the kernel trick***.

Transforming Data: An Example

Example

Let's define $\phi : \mathbb{R}^2 \to \mathbb{R}^6$ by

$$\phi([x_1, x_2]) = (1, \sqrt{2}x_1, \sqrt{2}x_2, x_1^2, x_2^2, \sqrt{2}x_1x_2)$$

The inner product in the feature space is

$$\phi([x_{11}, x_{12}])^\top \phi([x_{21}, x_{22}]) = (1 + x_{11}x_{21} + x_{12}x_{22})^2$$

Thus, we can directly define a kernel function $K : \mathbb{R}^2 \times \mathbb{R}^2 \to \mathbb{R}$ by

$$K(x_1, x_2) = (1 + x_{11}x_{21} + x_{12}x_{22})^2.$$

Notice that we need not compute $\phi([x_{11}, x_{12}])$, $\phi([x_{21}, x_{22}])$ to compute $K(x_1, x_2)$.

Kernel Functions

Common kernel functions include:

- **Polynomial Kernel** (kernel='poly')

$$K(x_1, x_2) = (x_1^\top x_2 + 1)^d$$

  where *d* is a hyperparameter.

- **Radial Basis Function Kernel** (kernel='rbf')

$$K(x_1, x_2) = \exp\left\{-\frac{\|x_1 - x_2\|^2}{2\sigma^2}\right\}$$

  where σ is a hyperparameter.

- **Sigmoid Kernel** (kernel='sigmoid')

$$K(x_1, x_2) = \tanh(\kappa x_1^\top x_2 + \theta)$$

  where κ and θ are hyperparameters.

If data is linearly separable, we don't need kernel

otherwise, we need nonlinear boundary

Classification Trees

Outline

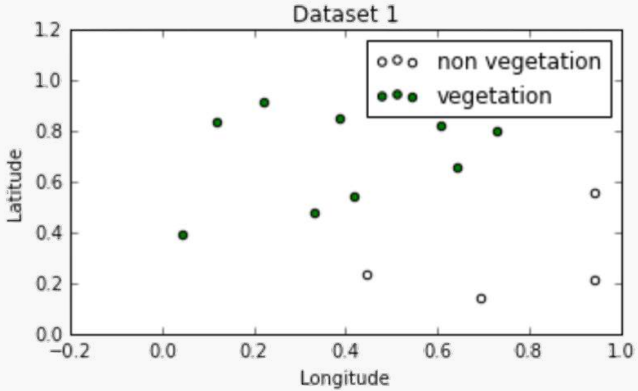
- Motivation
- Decision Trees
- Classification Trees
- Splitting Criteria
- Stopping Conditions & Pruning
- Regression Trees

Geometry of Data

Recall:

logistic regression for building classification boundaries works best when:

- the classes are well-separated in the feature space
- have a nice geometry to the classification boundary)



Geometry of Data

Recall:

the decision boundary is defined where the probability of being in class 1 and class 0 are equal, i.e.

$$P(Y = 1) = 1 - P(Y = 1) \quad \Rightarrow. \quad P(Y = 1) = 0.5,$$

Which is equivalent to when the `log-odds=0`:

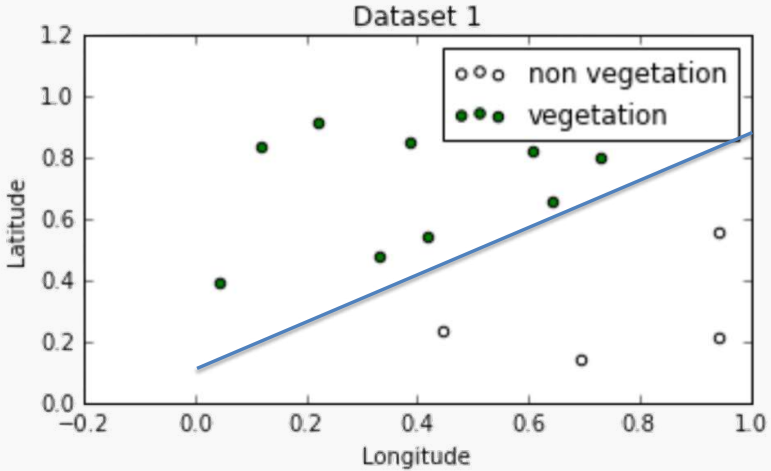
$$\boldsymbol{x}\beta = 0,$$

this equation defines a line or a hyperplane. It can be *generalized* with higher order polynomial terms.

Geometry of Data

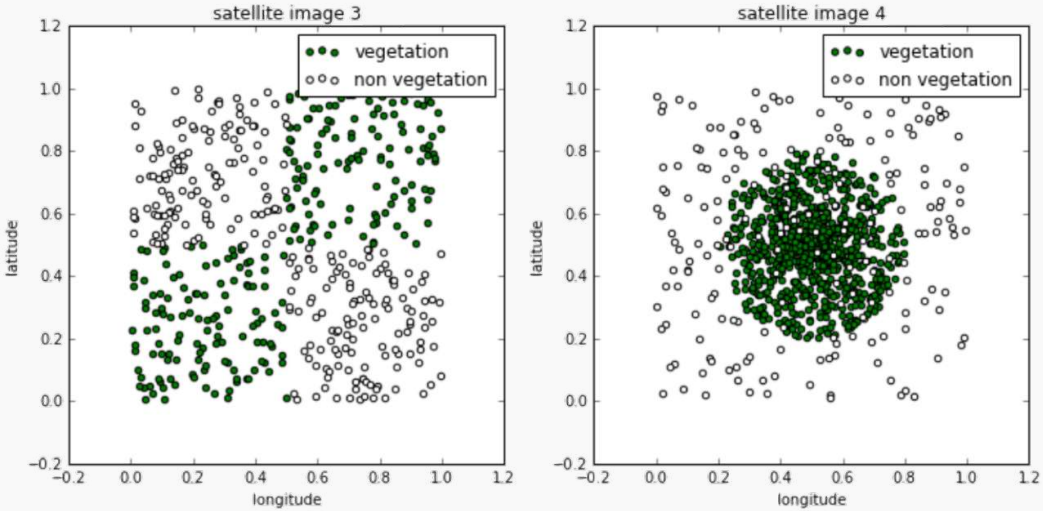
Question: Can you guess the equation that defines the decision boundary below?

$$-0.8x_1 + x_2 = 0 \implies x_2 = 0.8x_1 \Rightarrow \ Latitude = 0.8\ Lon$$



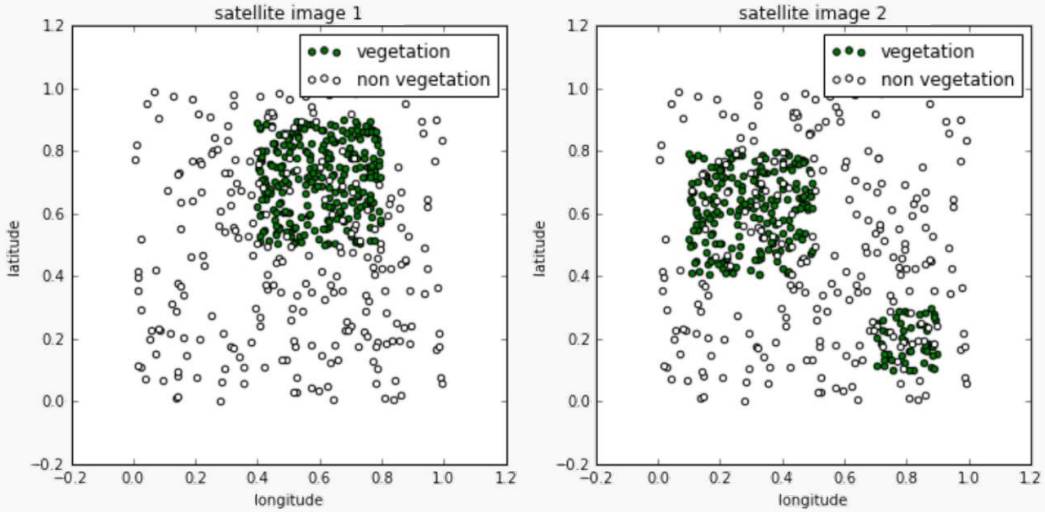
Geometry of Data

Question: How about these?



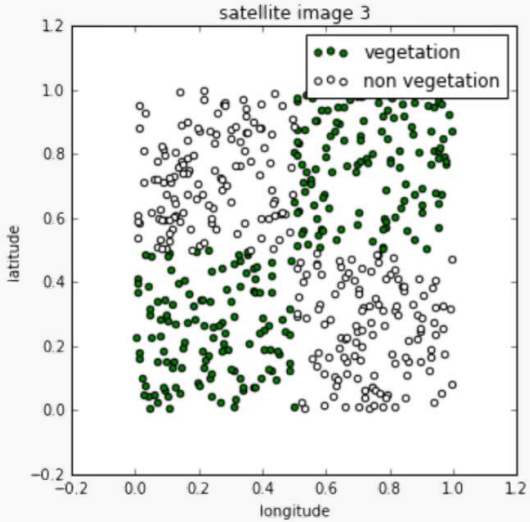
Geometry of Data

Question: Or these?



Geometry of Data

Notice that in all of the datasets the classes are still well-separated in the feature space, but ***the decision boundaries cannot easily be described by single equations***:



Geometry of Data

While logistic regression models with linear boundaries are intuitive to interpret by examining the impact of each predictor on the log-odds of a positive classification, it is less straightforward to interpret nonlinear decision boundaries in context:

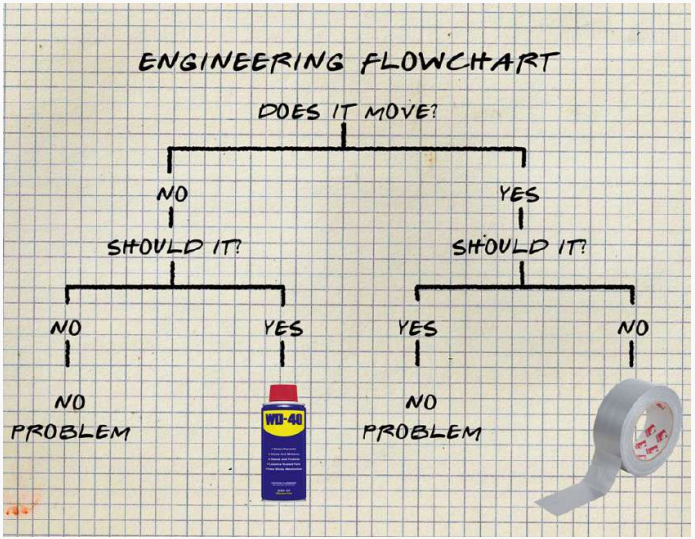
$$(x_3+2x_2) - x_1^2 + 10 = 0$$

It would be desirable to build models that:

1. allow for ***complex decision boundaries***.
2. are also ***easy to interpret***.

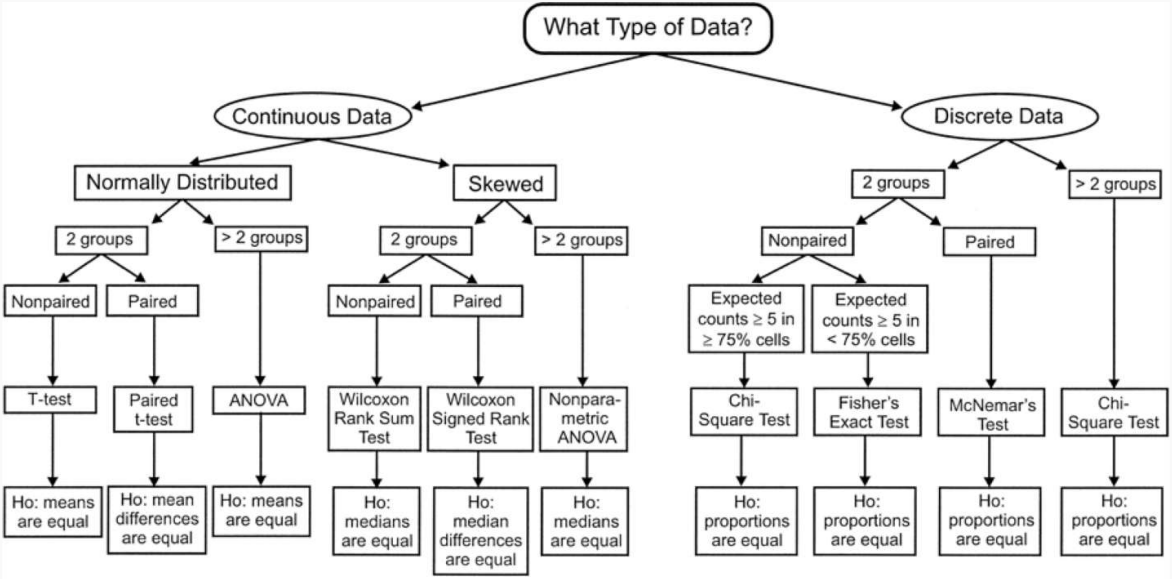
Interpretable Models

People in every walk of life have long been using interpretable models for differentiating between classes of objects and phenomena:



Interpretable Models (cont.)

Or in the [inferential] data analysis world:



Source: Waning B, Montagne M: *Pharmacoepidemiology: Principles and Practice*: http://www.accesspharmacy.com

Decision Trees

It turns out that the simple flow charts in our examples can be formulated as mathematical models for classification and these models have the properties we desire; they are:

1. interpretable by humans
2. have sufficiently complex decision boundaries
3. the decision boundaries are locally linear, each component of the decision boundary is simple to describe mathematically.

Decision Trees

The Geometry of Flow Charts

Flow charts whose graph is a tree (connected and no cycles) represents a model called a ***decision tree***.

Formally, a ***decision tree model*** is one in which the final outcome of the model is based on a series of comparisons of the values of predictors against threshold values.

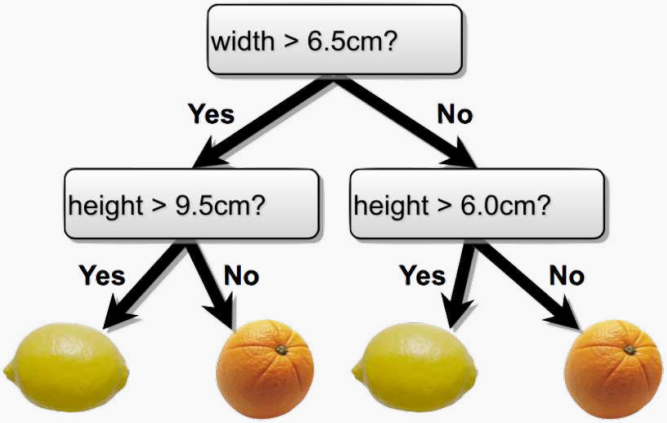
In a graphical representation (flow chart),

- the internal nodes of the tree represent attribute testing.
- branching in the next level is determined by attribute value (yes/no).
- terminal leaf nodes represent class assignments.

The Geometry of Flow Charts

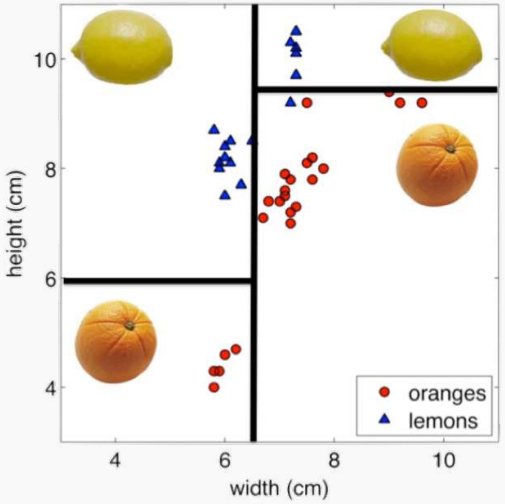
Flow charts whose graph is a tree (connected and no cycles) represents a model called a ***decision tree***.

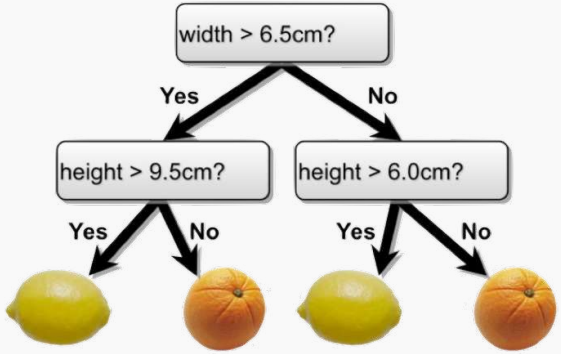
Formally, a ***decision tree model*** is one in which the final outcome of the model is based on a series of comparisons of the values of predictors against threshold values.



The Geometry of Flow Charts

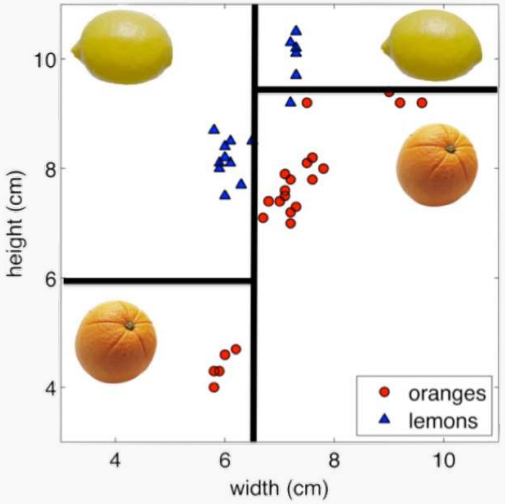
Every flow chart tree corresponds to a partition of the feature space by **axis aligned lines or (hyper) planes**. Conversely, every such partition can be written as a flow chart tree.

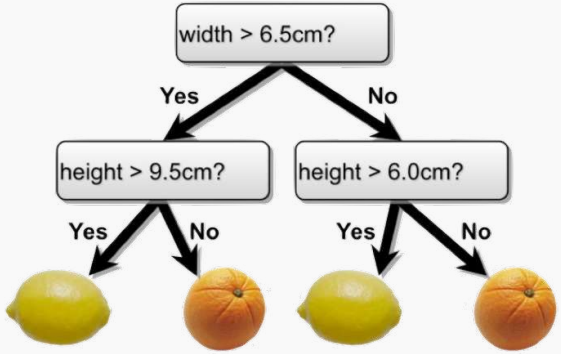




The Geometry of Flow Charts

Each comparison and branching represents splitting a region in the feature space on a single feature. Typically, at each iteration, we split once along one dimension (one predictor). Why?





Learning the Model

Given a training set, *learning* a decision tree model for binary classification means:

- producing an ***optimal*** partition of the feature space with axis-aligned linear boundaries (very interpretable!),
- each region is predicted to have a class label based on the **largest class** of the training points in that region (Bayes' classifier) when performing prediction.

Learning the Model

Learning the smallest ‘optimal’ decision tree for any given set of data is NP complete for numerous simple definitions of ‘optimal’. Instead, we will seek a reasonably model using a greedy algorithm.

1. Start with an empty decision tree (undivided feature space)
2. Choose the ‘optimal’ predictor on which to split and choose the ‘optimal’ threshold value for splitting.
3. Recurse on each new node until ***stopping condition*** is met

Now, we need only define our splitting criterion and stopping condition.

Numerical vs Categorical Attributes

Note that the ‘compare and branch’ method by which we defined classification tree works well for numerical features.

However, if a feature is categorical (with more than two possible values), comparisons like *`feature < threshold`* does not make sense.

How can we handle this?

A simple solution is to encode the values of a categorical feature using numbers and treat this feature like a numerical variable. This is indeed what some computational libraries (e.g. `sklearn`) do, however, this method has drawbacks.

Numerical vs Categorical Attributes

Example

Supposed the feature we want to split on is **color**, and the values are: Red, Blue and Yellow. If we encode the categories numerically as:

Red = 0, Blue = 1, Yellow = 2

Then the possible non-trivial splits on **color** are

{{Red}, {Blue, Yellow}} {{Red, Blue},{Yellow}}

But if we encode the categories numerically as:

Red = 2, Blue = 0, Yellow = 1

The possible splits are

{{Blue}, {Yellow, Red}} {{Blue,Yellow}, {Red}}

Depending on the encoding, the splits we can optimize over can be different!

Numerical vs Categorical Attributes

In practice, the effect of our choice of naive encoding of categorical variables are often negligible - models resulting from different choices of encoding will perform comparably.

In cases where you might worry about encoding, there is a more sophisticated way to numerically encode the values of categorical variables so that one can optimize over all possible partitions of the values of the variable.

This more principled encoding scheme is computationally more expensive but is implemented in a number of computational libraries (e.g. `R's randomForest`).

Splitting Criteria

Optimality of Splitting

While there is no ‘correct’ way to define an optimal split, there are some common sensical guidelines for every splitting criterion:

- the regions in the feature space should grow progressively more pure with the number of splits. That is, we should see each region ‘specialize’ towards a single class.
- the fitness metric of a split should take a differentiable form (making optimization possible).
- we shouldn’t end up with empty regions - regions containing no training points.

Classification Error

Suppose we have J number of predictors and K classes.

Suppose we select the j^{th} predictor and split a region containing N number of training points along the threshold $t_j \in \mathbb{R}$.

We can assess the quality of this split by measuring the **classification error** made by each newly created region, R_1, R_2:

$$\text{Error}(i|j, t_j) = 1 - \max_k p(k|R_i)$$

where $p(k|R_i)$ is the proportion of training points in R_i that are labeled class k.

Classification Error

Example

	Class 1	Class 2	Error$(i\|j,t_j)$
R_1	0	6	$1-\max\{6/6,0/6\}=0$
R_2	5	8	$1-\max\{5/13,8/13\}=5/13$

We can now try to find the predictor j and the threshold t_j that minimizes the average classification error over the two regions, weighted by the population of the regions:

$$\min_{j,t_j}\left\{\frac{N_1}{N}\text{Error}(1|j,t_j)+\frac{N_2}{N}\text{Error}(2|j,t_j)\right\}$$

issue: max is not differentiable

where N_i is the number of training points inside region R_i.

Gini Index

Suppose we have J number of predictors, N number of training points and K classes.

Suppose we select the j^{th} predictor and split a region containing N number of training points along the threshold $t_j \in \mathbb{R}$.

We can assess the quality of this split by measuring the purity of each newly created region, R_1, R_2. This metric is called the **Gini Index**:

$$\text{Gini}(i|j, t_j) = 1 - \sum_k p(k|R_i)^2$$

Question: What is the effect of squaring the proportions of each class? What is the effect of summing the squared proportions of classes within each region?

Gini Index

Example			
	Class 1	Class 2	$\text{Gini}(i\|j,t_j)$
R_1	0	6	$1-(6/6^2+0/6^2)=0$
R_2	5	8	$1-[(5/13)^2+(8/13)^2]=80/169$

We can now try to find the predictor j and the threshold t_j that minimizes the average Gini Index over the two regions, weighted by the population of the regions:

penalizes minority misclassification

$$\min_{j,t_j}\left\{\frac{N_1}{N}\text{Gini}(1|j,t_j)+\frac{N_2}{N}\text{Gini}(2|j,t_j)\right\}$$

where N_i is the number of training points inside region R_i.

Information Theory

entropy

The last metric for evaluating the quality of a split is motivated by metrics of uncertainty in information theory.

Ideally, our decision tree should split the feature space into regions such that each region represents a single class. In practice, the training points in each region is distributed over multiple classes, e.g.:

	Class 1	Class 2
R_1	1	6
R_2	5	6

However, though both imperfect, R_1 is clearly sending a stronger 'signal' for a single class (Class 2) than R_2.

Information Theory

One way to quantify the strength of a signal in a particular region is to analyze the distribution of classes within the region. We compute the **entropy** of this distribution.

For a random variable with a discrete distribution, the entropy is computed by:

$$H(X) = -\sum_{x \in X} p(x) \log_2 p(x)$$

Higher entropy means the distribution is uniform-like (flat histogram) and thus values sampled from it are ‘less predictable’ (all possible values are equally probable).

Lower entropy means the distribution has more defined peaks and valleys and thus values sampled from it are ‘more predictable’ (values around the peaks are more probable).

Entropy

Suppose we have J number of predictors, N number of training points and K classes.

Suppose we select the j^{th} predictor and split a region containing N number of training points along the threshold $t_j \in \mathbb{R}$.

We can assess the quality of this split by measuring the entropy of the class distribution in each newly created region, R_1, R_2:

$$\min_{j,t_j} \left\{ \frac{N_1}{N} \text{Entropy}(1|j,t_j) + \frac{N_2}{N} \text{Entropy}(2|j,t_j) \right\}$$

Note: we are actually computing the conditional entropy of the distribution of training points amongst the K classes given that the point is in region i.

Entropy

Example

	Class 1	Class 2	$\text{Entropy}(i\|j, t_j)$
R_1	0	6	$-(\frac{6}{6}\log_2\frac{6}{6} + \frac{0}{6}\log_2\frac{0}{6}) = 0$
R_2	5	8	$-(\frac{5}{13}\log_2\frac{5}{13} + \frac{8}{13}\log_2\frac{8}{13}) \approx 1.38$

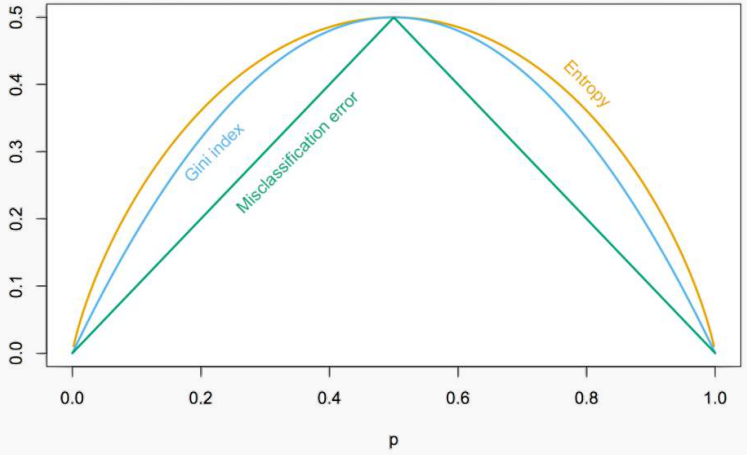
We can now try to find the predictor *j* and the threshold t_j that minimizes the average entropy over the two regions, weighted by the population of the regions:

$$\min_{j,t_j}\left\{\frac{N_1}{N}\text{Entropy}(1|j,t_j) + \frac{N_2}{N}\text{Entropy}(2|j,t_j)\right\}$$

Comparison of Criteria

Recall our intuitive guidelines for splitting criteria, which of the three criteria fits our guideline the best?

We have the following comparison of the value of the three criteria at different levels of purity (from 0 to 1) in a single region (for binary outcomes).



use gini & entropy to figure out when to split

Comparison of Criteria

Recall our intuitive guidelines for splitting criteria, which of the three criteria fits our guideline the best?

To note that entropy penalizes impurity the most is not to say that it is the best splitting criteria. For one, a model with purer leaf nodes on a training set may not perform better on the testing test.

Another factor to consider is the size of the tree (i.e. model complexity) each criteria tends to promote.

To compare different decision tree models, we need to first discuss ***stopping conditions***.

Stopping Conditions & Pruning

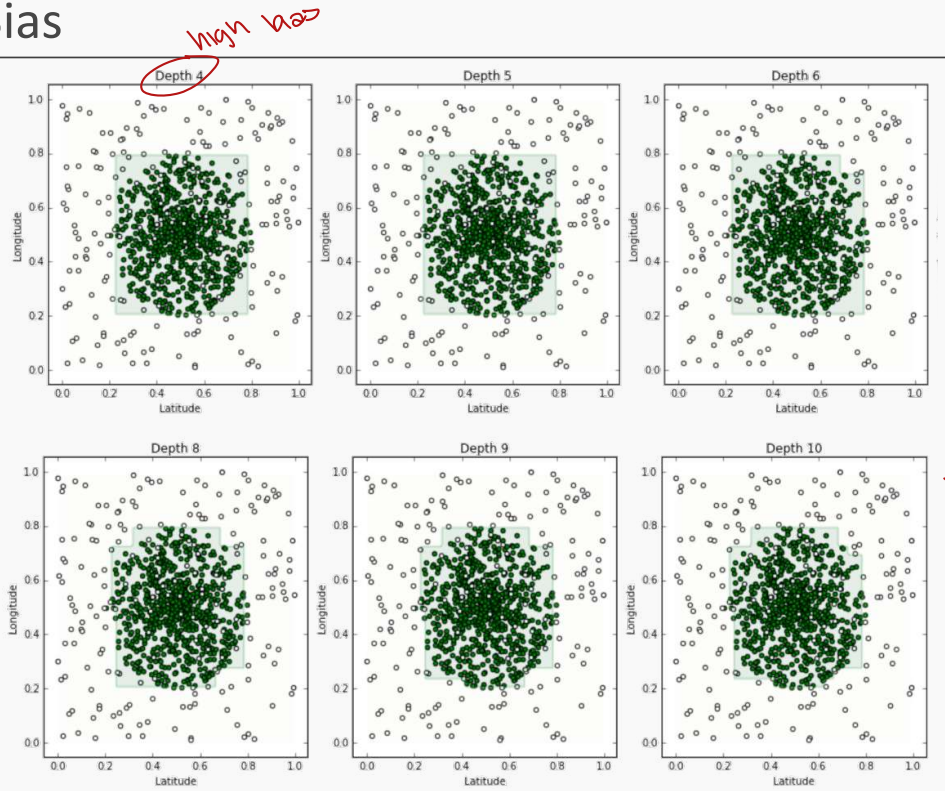
Variance vs Bias

If we don't terminate the decision tree learning algorithm manually, the tree will continue to grow until each region defined by the model possibly contains exactly one training point (and the model attains 100% training accuracy).

To prevent this from happening, we can simply stop the algorithm at a particular depth.

But how do we determine the appropriate depth?

Variance vs Bias



Variance vs Bias

We make some observations about our models:

- **(High Bias)** A tree of depth 4 is not a good fit for the training data - it's unable to capture the nonlinear boundary separating the two classes.
- **(Low Bias)** With an extremely high depth, we can obtain a model that correctly classifies all points on the boundary (by zig-zagging around each point).
- **(Low Variance)** The tree of depth 4 is robust to slight perturbations in the training data - the square carved out by the model is stable if you move the boundary points a bit.
- **(High Variance)** Trees of high depth are sensitive to perturbations in the training data, especially to changes in the boundary points.

Not surprisingly, complex trees have low bias (able to capture more complex geometry in the data) but high variance (can overfit). Complex trees are also harder to interpret and more computationally expensive to train.

Stopping Conditions

Common simple stopping conditions:

- Don’t split a region if all instances in the region belong to the same class.
- Don’t split a region if the number of instances in the sub-region will fall below pre-defined threshold (**min_samples_leaf**).
- Don’t split a region if the total number of leaves in the tree will exceed pre-defined threshold.

The appropriate thresholds can be determined by evaluating the model on a held-out data set or, better yet, via cross-validation.

Stopping Conditions

More restrictive stopping conditions:

- Don't split a region if the class distribution of the training points inside the region are independent of the predictors.
- Compute the gain in purity, information or reduction in entropy of splitting a region *R* into R_1 and R_2*:*

$$Gain(R) = \Delta(R) = m(R) - \frac{N_1}{N} m(R_1) - \frac{N_2}{N} m(R_2)$$

where *m* is a metric like the Gini Index or entropy. Don't split if the gain is less than some pre-defined threshold (**min_impurity_decrease**).

Alternative to Using Stopping Conditions

What is the major issue with pre-specifying a stopping condition?

- you may stop too early or stop too late.

How can we fix this issue?

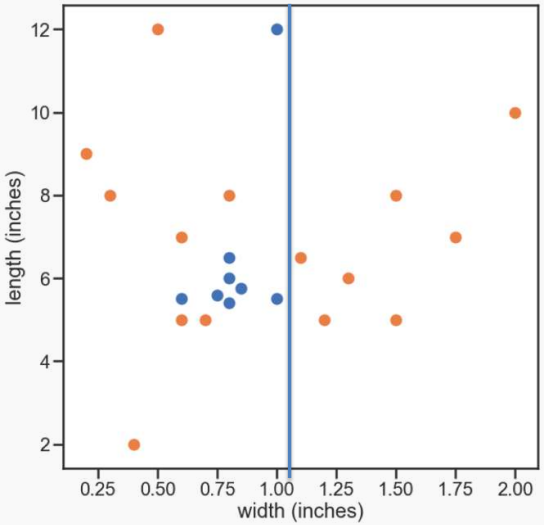
- choose several stopping criterion (set minimal Gain(R) at various levels) and cross-validate which is the best.

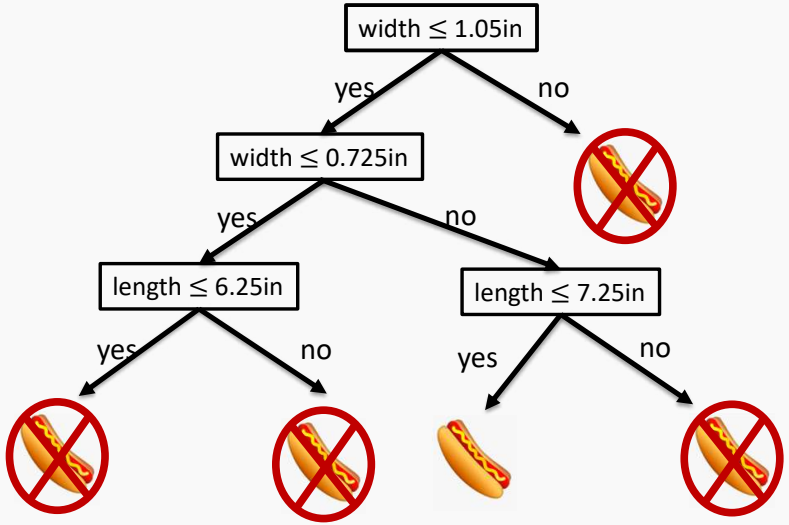
What is an alternative approach to this issue?

- Don't stop. Instead prune back!

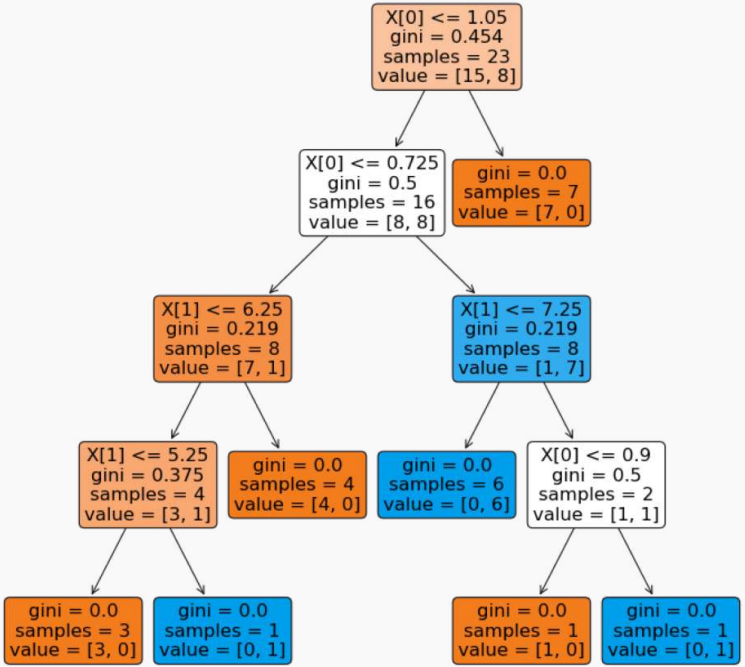
To Hot Dog or Not Hot Dog…

Hot Dog or Not

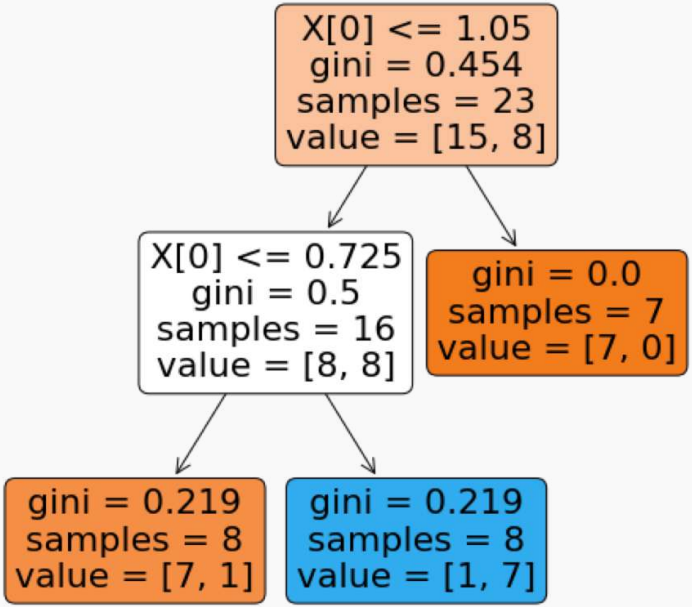




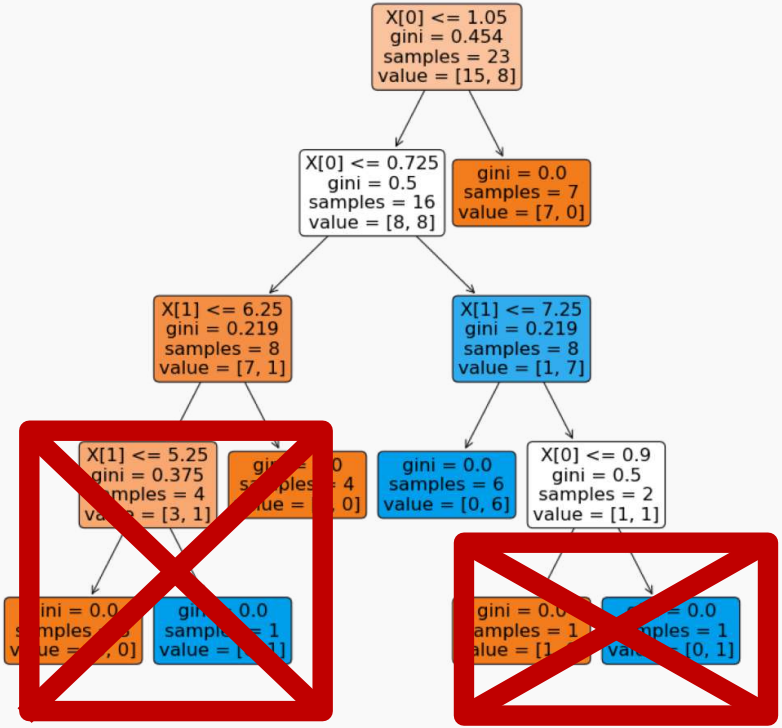
Motivation for Pruning



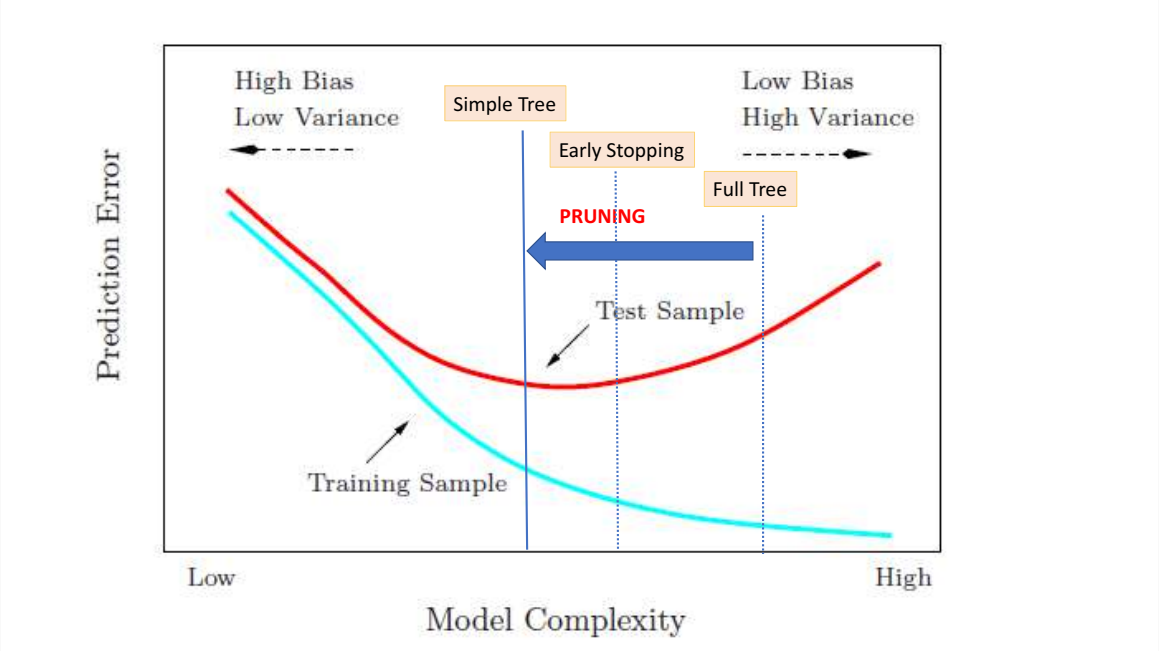
Motivation for Pruning



Motivation for Pruning



Motivation for Pruning



Pruning

Rather than preventing a complex tree from growing, we can obtain a simpler tree by ‘pruning’ a complex one.

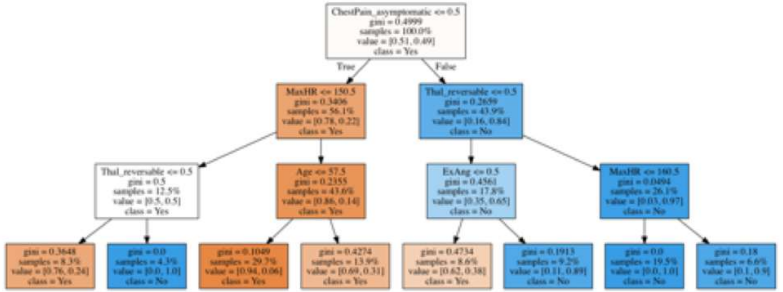
There are many method of pruning, a common one is ***cost complexity pruning***, where by we select from a array of smaller subtrees of the full model that optimizes a balance of performance and efficiency.

That is, we measure

$$C(T) = Error(T) + \alpha|T|$$

where *T* is a decision (sub) tree, $|T|$ is the number of leaves in the tree and α is the parameter for penalizing model complexity.

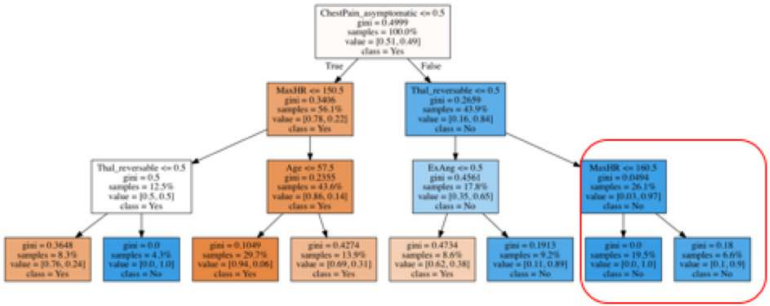
Pruning



$\alpha = 0.2$

Tree	Error	Num Leaves	Total

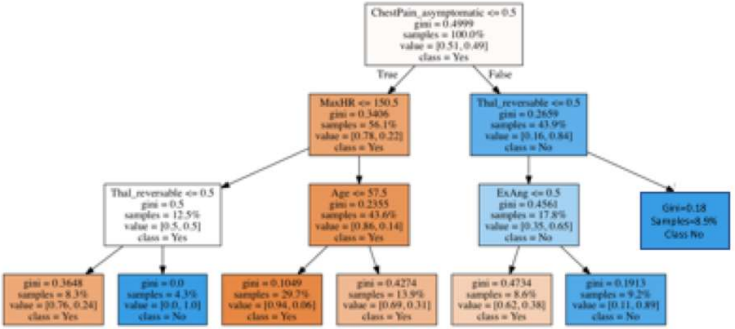
Pruning



$\alpha = 0.2$

Tree	Error	Num Leaves	Total
T	0.32	8	1.92

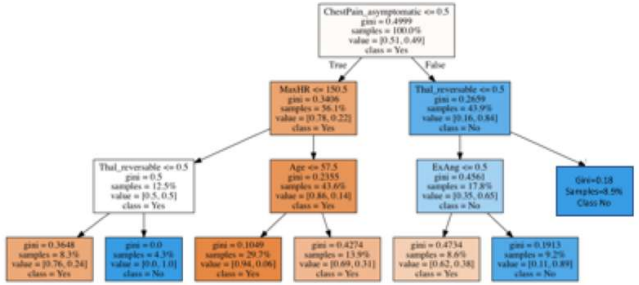
Pruning



$\alpha = 0.2$

Tree	Error	Num Leaves	Total
T	0.32	8	1.92
Tsmall	0.33	7	1.73

Pruning



$\alpha = 0.2$

Tree	Error	Num Leaves	Total
T	0.32	8	1.92
Tsmall	0.33	7	1.73

Smaller tree has larger error but less cost complexity score

Pruning

$$C(T) = Error(T) + \alpha|T|$$

1. Fix α.

2. Find best tree for a given α and based on cost complexity *C*.

3. Find best α using CV (what should be the error measure?)

Pruning

The pruning algorithm:

1. Start with a full tree T_0 (each leaf node is pure)
2. Replace a subtree in T_0 with a leaf node to obtain a pruned tree T_1. This subtree should be selected to minimize

$$\frac{Error(T_0) - Error(T_1)}{|T_0| - |T_1|}$$

3. Iterate this pruning process to obtain $T_0, T_1, \ldots, T_L$ where T_L is the tree containing just the root of T_0
4. Select the optimal tree T_i by cross validation.

Note: you might wonder where we are computing the cost-complexity $C(T_l)$. One can prove that this process is equivalent to explicitly optimizing *C* at each step.