Supervised Learning Algorithms

Michelle Chung

Last updated Mar. 27, 2018

Purpose: This document was created for the students of Dr. Kapelner's 390.4 Intro to Data Science and Machine Learning class (`github.com/kapelner/QC_Math_390.4_Spring_2018`) to supplement their notes and coursework. It details machine learning algorithms covered in the Spring 2018 semester. Some mathematical notation may be specific to the class.

Please submit errors to `mchung.92@gmail.com`.

Introduction

"All models are wrong."

It is certainly true that all models are wrong. Consider the levels of abstraction in building a model.

If $t(\vec{z})$ is the true causal relationship; the reality we wish to capture, then
$f(\vec{x})$ is the numerical function that most closely captures $t(\vec{z})$,
$h^*(\vec{x})$ is the best approximation of $f(\vec{x})$, and
$\hat{f}(\vec{x})$ is our best guess of $h^*(\vec{x})$.

At each level of abstracting from the "truth", there is room for error to be introduced. Some error is a result of making poor modeling decisions and can be identified and reduced. There is, however, a portion of error that we refer to as *irreducible error*, or error due to ignorance. This type of error exists in all models and cannot be completely eliminated, which implies that all models are inherently imperfect.

"But some are useful."

If all models are wrong, why do we build them?

We build numerical models of natural phenomenon for the purposes of *inference*, learning about the phenomenon, and/or *prediction*, guessing at what might happen based on what we've already seen. While models may not exactly represent how things happen in the world, with a little dose of humility, we can use them to better inform our understanding of the world.

Notation

To estimate f, we need to acquire data, choose our candidate function set, and choose an algorithm that will output our $\hat{f}$.

$$\hat{f} = \mathcal{A}(\mathcal{D}, \mathcal{H})$$

Where $\mathcal{D}$ is the training data containing n observations of inputs $\langle \overrightarrow{X_i}, y_i \rangle$ where $i = 1 \dots n$ and $\overrightarrow{X_i} = \langle x_{i,1}, x_{i,2}, \dots, x_{i,p} \rangle$. A visual representation of $\mathcal{D}$ is illustrated below.

$$\mathcal{D} = \begin{bmatrix} \langle x_{1,1}, x_{1,2}, \dots, x_{1,p} \rangle \\ \langle x_{2,1}, x_{2,2}, \dots, x_{2,p} \rangle \\ \vdots \\ \langle x_{n,1}, x_{n,2}, \dots, x_{n,p} \rangle \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

Because there are infinitely many functions in $\mathbb{R}$ to consider, we narrow the scope of what imagine our f might look like. These are our candidate functions, $\mathcal{H}$. Finally, we choose an algorithm $\mathcal{A}$ that will output the best candidate function, noted as h^*, given $\mathcal{D}$ and $\mathcal{H}$. This document describes some of these algorithms.

1 Single-Layer Perceptron

1.1 Purpose

The purpose of the perceptron algorithm is to produce a binary classifier of the form:

$$\hat{f}(x) = \begin{cases} 1 & \text{if } \vec{w} \cdot \vec{x} + b \ > 0 \\ 0 & \text{otherwise} \end{cases}$$

Where $\vec{w}$ is our weight vector (intuitively, the slope of our line) and b is our bias (intuitively, the y-intercept).

Thm 1.1. The perceptron will converge under the condition that the data is linearly separable.

1.2 Algorithm

1. Initialize the weight vector and bias to 0 or small random values. We will use 0 in this example.*
2. For each vector $\langle \overrightarrow{X_i}, y_i \rangle$ in $\mathcal{D}$, perform the following actions:

*In neural networks with more than one input layer, we prefer to initialize $\vec{w}$ with small random values instead of 0 for the purpose of optimizing learning.

(a) Calculate $\texttt{sum}(w \cdot x_i) = \hat{y}_i$.

(b) Calculate your error, e, by taking the difference of the true y, y_i, and your calculated y, $\hat{y}_i$ such that:

$$e = \hat{y}_i - y_i$$

(c) Update the weights in the weight vector by multiplying $\vec{X}_i \cdot e$ and adding it to $\vec{w}$ such that:

$$w^{t=n} = w^{t=n-1} + e \cdot \vec{X}_i$$

Where t represents the number iteration of the algorithm. The magnitude of e represents the size of your error. If the error is large, we want to change our weights more dramatically. If the error is small, our current weights are performing well, and we don't want to change them too much.

3. Repeat step 2 for all $i = 1 \dots n$.

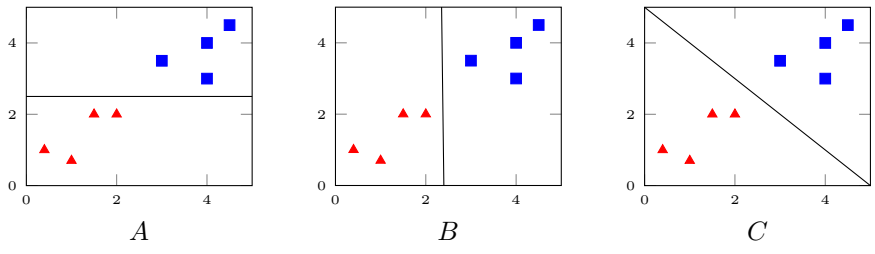
4. Steps 1 - 3 can be repeated for a user-defined number of iterations or until the average sum error is less than a user-specified threshold error.

At first glance, Step 4 seems redundant. According to Thm 1.1, if the data is linearly separable, the perception *must* converge to a solution. If the first iteration of the algorithm produces a solution, why should we want to go through the trouble of producing additional solutions?

The answer has to do with the nature of linear models. If data is linearly separable, more than one line will be able to bisect the feature space. In fact, there will be infinitely many such lines.

We want to build several models because there are infinitely many "correct" models–but some models are more "correct" than others.

Consider the following models:



Each of these classifiers is technically correct based on the data, but intuitively, it seems that classifier C most accurately bisects the feature space and will give us the best predictions.

Fortunately, using a different algorithm, we can capture something close to C every time.

2 Support Vector Machines

2.1 Purpose

Like the perceptron, the support vector machine (SVM) outputs binary classifiers. Unlike the perceptron, the SVM optimizes the model and is able to perform when the data is not linearly separable–an area in which the perceptron fails. These characteristics make the SVM a better choice in most cases of producing a binary classifier.

2.2 Algorithm

The general idea of the SVM is to find the largest "wedge" between the response spaces y_1 and y_2 and return the line that rests in the middle.

The boundaries of this wedge are defined as parallel hyperplanes (the *support vectors*), which touch tangentially on $y_1, y_2 \in Y$. The distance from the line $\vec{w} \cdot \vec{x}$ to its support vectors is called the *margin*, or M.

In the case of perfect linear separability, the SVM is said to have a "hard margin." In the case that the data is not linearly separable, the SVM is said to have a "soft margin."

Hard Margin

Consider the line

$$\vec{w} \cdot \vec{x} - b = 0$$

that bisects the response space $Y = \{-1, 1\}$. Its support vectors can be described by the following equations*:

$$\vec{w} \cdot \vec{x} - b = 1$$

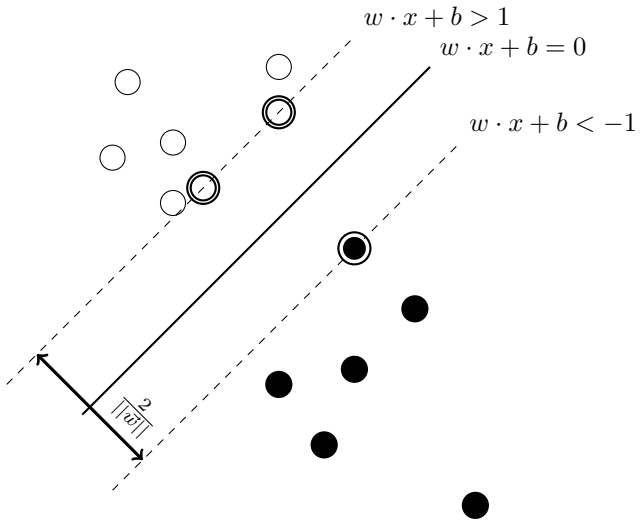
Anything on or above this boundary is of one class, with label 1.

$$\vec{w} \cdot \vec{x} - b = -1$$

Anything on or below this boundary is of one class, with label -1.

The SVM is visualized below, where the solid line represents our linear model and the dashed lines represent our support vectors which touch tangentially on the binary response space.

*Note that we arbitrarily coerce the value 1 to represent the distance, δ, from our linear model to each support vector. We could just as easily choose any number in $\mathbb{R}$ to represent our distance.



The slope of the line is determined by the values in $\vec{w}$, which is orthogonal to $\vec{x}$, while the size of the margin is determined by the length of $\vec{w}$. Note that minimizing $\vec{w}$ is equivalent to maximizing the margin, and that the perpendicular distance of the i^{th} observation to the hyperplane is given by $y_i(\vec{w} \cdot \vec{x}_i)$.

The goal of the SVM is to maximize the margin while under the constraint that no misclassification errors should be made–that is, each point is on the correct side of the hyperplane. Thus, the SVM solves the numerical optimization problem:

$$\texttt{min}||\vec{w}|| \text{ such that } y_i(\vec{w} \cdot \vec{x}_i + b) \geq M \text{ for } \forall i \text{ in } i = 1 \dots n$$

Where y_i is the i^{th} true output and $(\vec{w} \cdot \vec{x}_i + b)$ is the i^{th} predicted output. Note that outputs can either be $-1, 1$, while the margin M, because it is a distance, is always a positive number. Should the predicted and true output differ, $y_i(\vec{w} \cdot \vec{x}_i + b)$ would result in a negative number, violating the constraint that $y_i(\vec{w} \cdot \vec{x}_i + b) \geq M$.

Soft Margin

In the case that the data is not linearly separable, our model could not converge to a solution under the above constraint. We could, however, choose a model that produces the least amount of error.

To do this, we use loss functions. The purpose of a loss function is to penalize misclassification errors, thereby giving us a metric to measure the performance of our model and to choose the best one.

The hinge-loss function is one such function. For the SVM defined in our example, the hinge-loss function is defined as:

$$\texttt{max}(0, 1 - y_i(\vec{w} \cdot \vec{x}_i + b)$$

Again, $y_i(\vec{w} \cdot \vec{x}_i + b)$ would only produce a negative number in the case that the predicted output and the true output do not belong to the same class. In addition to recording the number of misclassification errors, the hinge-loss function is accounts by measuring the distance from the erroneous output to the line.

Therefore, in the soft margin model, we are trying to minimize:

$$\left[\frac{1}{n}\sum_{i=1}^{n}\max\left(0, 1 - y_i(\vec{w} \cdot \vec{x}_i + b)\right)\right] + \lambda\|\vec{w}\|^2$$

Where the addition of the parameter λ allows us to determine the trade-off between the margin size and the number of errors in the model.

3 K-Nearest Neighbors

3.1 Purpose

The K-Nearest Neighbors (KNN) algorithm uses a non-linear approach to classification. The algorithm receives an input vector $\vec{x}$, finds its k-closest neighbors in the feature space, and assigns a classification y to $\vec{x}$ based on the majority classification of its k neighbors. Think of it as a majority vote in a district of k people.

3.2 Algorithm

1. For $x_i \in \vec{X}_{input}$ where $i = 1 \ldots p$, and
2. For $\vec{X}_{i,j} \in \mathcal{D}$ where $i = 1 \ldots p$ and $j = 1 \ldots n$,
3. return $\hat{y} = \texttt{argmin}\left\{\ d(\vec{X}_{input}, \vec{X}_{i,j})\ \right\}$

Where d is our distance function.

The KNN algorithm is based on the notion of distance. Two common distance functions are described below.

Manhattan Distance

The sum of absolute distance.

$$d(\vec{x}, \vec{y}) = \sum_{i=1}^{n} |x_i - y_i|$$

Euclidean Distance

The root sum of squared distance.

$$d(\vec{x}, \vec{y}) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$