Instructors: Parth Shah, Riju Pahwa

Lecture 1 Notes

Outline

1. Machine Learning
    - What is it?
    - Classification vs. Regression
    - Error
    - Training Error vs. Test Error
2. Linear Classifiers
    - Goals and Motivations
    - Perceptron Algorithm
    - Passive Aggressive
    - SVM, Pegasos
3. Basic Algorithms: Clustering
    - Goals and Motivations
    - Nearest Neighbors

Machine Learning

What is it?

There has been a lot of buzz around machine learning which has led to a lot of ambiguous definitions and mistaken notions.

Fundamentally, machine learning is about predicting things we have not seen by using what we have seen. For example, Facebook autotagging looks at existing tagged photos, to predict who is who when you post your newest photo album on Facebook. In this class, you will have access to a large amount of urban data and might end up working on a project that tries to predict where car collisions may occur in the next hour in Boston.

Machine learning is therefore an approach to predict unseen data, referred to as **test data**, using seen data, referred to as **training data**. The more training data that is available the better we would hope our machine learning algorithms will perform on predicting the test data. I have used prediction loosely here. What exactly does predicting mean?

Classification vs. Regression

Prediction comes in two flavors: **classification** and **regression**. Classification is the process of applying labels to data. For example, I may have a bunch of movies for which I want to determine, based on the previous ratings I have given movies, whether I will like or dislike the movie. This is a classification problem because the goal is to tag the test data with either the label "LIKE" or the label "DISLIKE". Regression on the other hand is assigning a value to the data. It does not explicitly bucket the data. If we reformulate this movie problem as a regression problem it would be predicting the numerical quality score I give it out of 100. Note while classification is cares only about getting the label right, regression is more about trying to predict a value that is not far off from the real value.

Error

This jumps right into the idea of error. It seems odd at first, but error is arguably one of the most important parts of machine learning. Beyond allowing us to evaluate performance, it turns out to be the tool we use to **train** our machine learning algorithms. Training an algorithm means using the training data to learn some prediction scheme that can then be used on the unseen test data. Therefore error is how the actual" learning" in machine learning happens.

Before we get into the fascinating ways in which different algorithms learn from the training data, we will explain how we measure error.

Classification Error: The following definitions are how we calculate error when we are using a classification algorithm. Note in classification the concept of seen data means we know the label of the data. So the training data not only includes data points but also their associated labels. The goal is then to use this training data to label unlabeled test data points.

Let the training data be $\{(x^{(1)}, y^{(1)}); (x^{(2)}, y^{(2)}), \cdots (x^{(n)}, y^{(n)})\}$ where each $x^{(i)}$ is a specific training data point and $y^{(i)}$. We will think of our machine learning algorithm as some function f that takes some input data point x and maps it to a label y.

Training error is then defined as follows:

$$\epsilon_{\text{train}} = \frac{1}{n} \sum_{i=1}^{n} [[f(x^{(i)}) \neq y^{(i)}]] \tag{1}$$

$[[f(x^{(i)}) \neq y^{(i)}]]$ is a function that equals 1 when the statement inside the brackets is true and 0 otherwise. In many languages this is equivalent to applying an int cast to a boolean (which we will see in python).

Test error is then defined as, for a given test data set $\{\xi^{(1)}, \xi^{(2)}, \cdots \xi^{(m)}\}$:

$$\epsilon_{\text{test}} = \frac{1}{m} \sum_{i=1}^{m} [[f(\xi^{(i)}) \neq \text{real label of } \xi^{(i)}]] \tag{2}$$

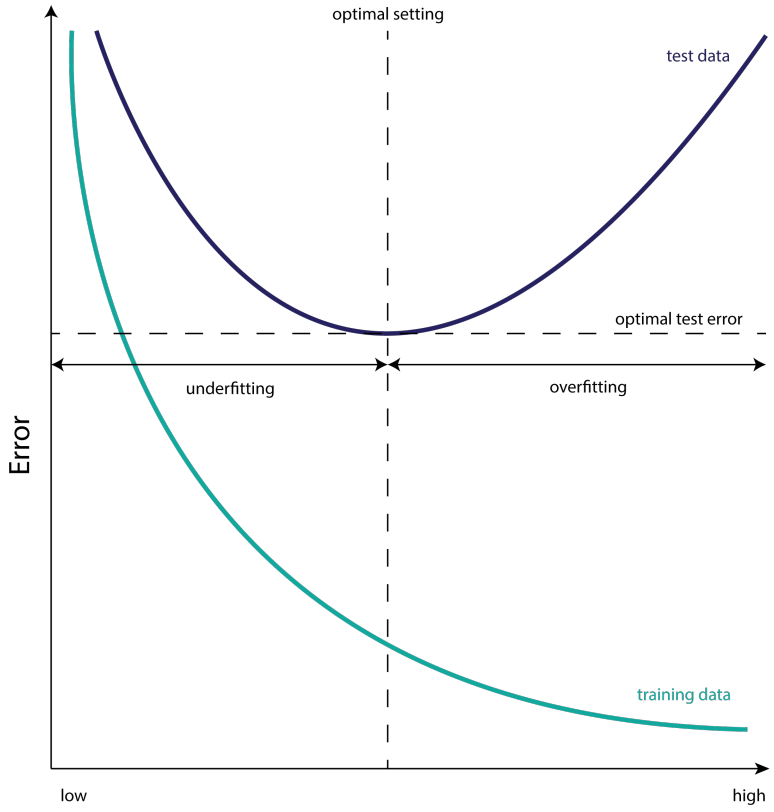
Regression Error: I use this term loosely, because there is not exactly a single regression error. Often a mean squared error scheme is used to calculate error for a regression algorithm, but that is not necessarily the only way to calculate the error.

Loss Functions: Both classification error and regression error calculate performance, but the algorithm itself does not necessarily optimize for the classification error or regression error. Why? Because an algorithm only knows the training data. Optimizing for the training data does not necessarily optimize for the test data, which is what we care about.

The machine learning algorithms themselves define error as **loss** and different algorithms optimize for different **loss functions**. We will explain loss functions more as we encounter specific machine learning algorithms.

Training Error vs Test Error

This is arguably the biggest take away to make from this introduction section. I will start by providing the graph that shows the relationship and then explaining what it implies.



The relationship between training error and test error is best understood from the standpoint of using the information given by the data. If the algorithm does not use enough of the information contained in the training then it is clearly not going to perform as well as another algorithm that uses more of the information. This is called **underfitting**. Meanwhile, the opposite issue is when the algorithm becomes top specific to the training data. An extreme example is an algorithm that on seen data reads back the label, but otherwise just guesses randomly. In this case the performance on the training data is flawless. It makes 0 mistakes.

But on the test data this algorithm would perform poorly. In general, when the algorithm overadapts to the training data this leads to poor performance on the test data and is called **overfitting**.

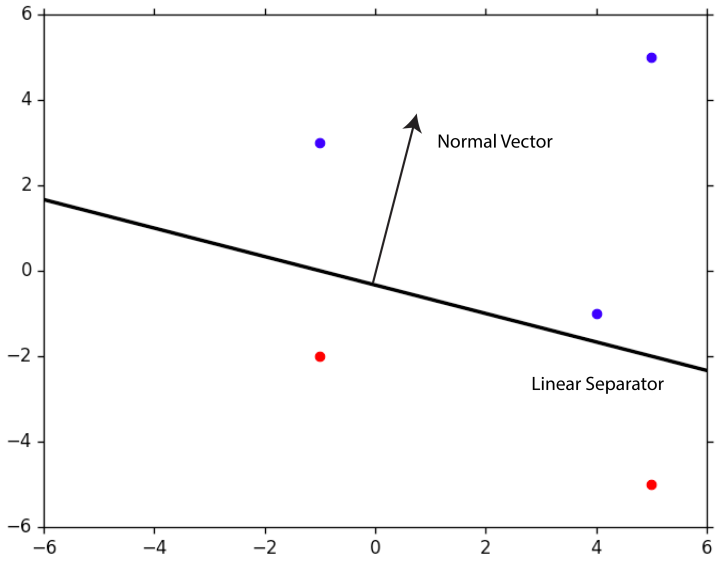
The key is to find the sweetspot which is the middle ground between underfitting and overfitting. A couple final observations before we move into some actual algorithm. Note the x-axis is unlabeled. This is because this graph actually applies in a lot of situations. Sometimes the x-axis could be a specific parameter of a given algorithm. Or it could be a comparison of the complexity of various algorithms. The examples are endless and of course its a generalization, but the point is you want to extract as much information from the training data as you can without including the noise of the data itself. There is a lot of interesting theory work on the relationship between training error and test error and if you are interested a good starting point would be **probably approximate correct (PAC)** learning and **VC dimension/shattering**.

Linear Classifiers

Goals and Motivation

A linear classifier is simply a classifier that uses a line to label the data points (duh!). Lets say in 2 dimensions we take a line and all points above are positive and all points below are negative. This is a linear classifier.

Interestingly enough, all linear classifiers can be defined in a such manner with a slight modification. Instead of saying above and below (which is ambiguous) we will instead define a line by its normal vector. The normal vector tells us the direction of the line. Why not just use the generic point slope version of a line as definition? Before we answer that question notice that two normal vectors define the same line. The advantage of the normal is that we now say all points on the same side of the line as the normal vector are positively labeled and all points on the other side are negative. This also addresses the fact that one linear separator can lead to two different labelings.



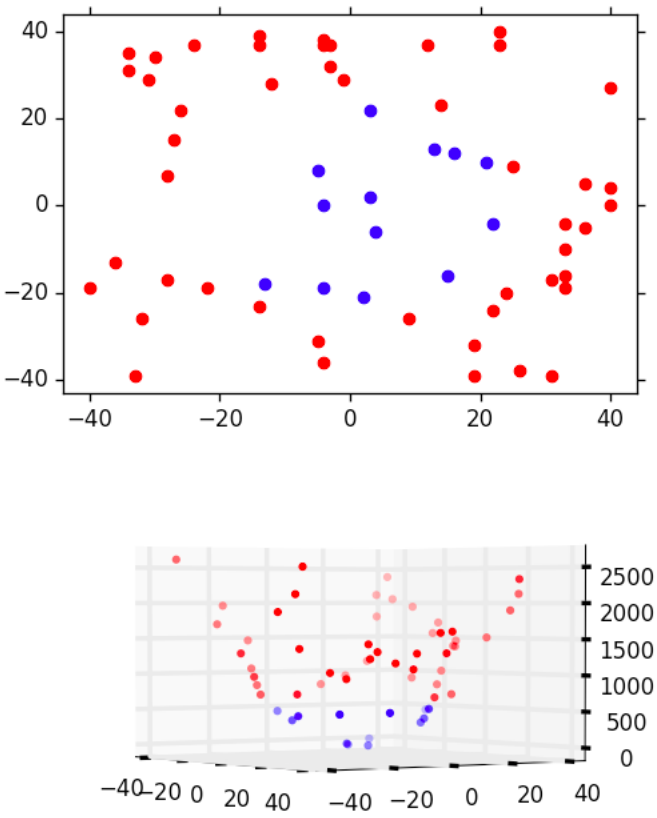
Note that to actually define the line you need the normal vector and one point on the line because the vector only gives the slope. To get the actual location of the line most algorithms use a notion of the offset which relates to how far the line is from the origin.

Why learn about linear classifiers? In any field the "why" questions are always the most important and in the field of machine learning this is a natural one. There are a couple clear issues with linear classifiers.

Issue 1: What if the data is not linearly separable?

This is a pretty big problem. If our machine learning algorithms can only classify using linear relationships then we are quite limited. Sure a lot of data may be linearly separable but a lot of data is not.

Solution: Who said we have to work with the data strictly as it is inputted. What if we transform the data using a set of functions and then try to linearly classify our transformed data. For example, lets say we have a bunch of data as follows (generated by data_transformation.py):



Here we see an original data set that is radially separable and is simply transformed into a new dataset which is linearly separable. Sure not all data is linearly separable, but if we apply a certain transformation to the data is can be. This is where we see the first true limitation of linear classifiers. All though most data can be expected to obey a certain set of relationships, we obviously cannot cover all transformations without understanding the training data. Of course understanding the data is the point of a machine learning algorithm so in cases where relationships of the data are complex we look to neural networks and other more advanced algorithms.

Issue 2: What if we want more labels than just positive and negative?

Earlier we defined classification as effectively bucketing data points. Using two buckets seems quite limiting. Turns out we can devise schemes that can assign multiple labels using *multiple* linear classifiers.

Solution: One way to do this is to use a binary classification scheme. Lets say I have labels A, B, C, D, E.

My first classifier could assign a positive label to data points that are in A or B and negative label to C, D or E. The second classifier could separate our new data-set of points in $A + B$ and decide whether to label them as either A or B.

Another solution could be to create a set of linear classifiers that determine for each individual label if the data point should receive the label or not. So in the A, B, C, D, E case we would have 5 classifiers. Of course this introduces it's own set of issues (what if a point is labeled into two different groups? what if a point is not labeled at all?), but these are reconcilable by making simple augmentations to the classifier.

What this goes to show is that linear classifiers are sufficient and extremely important for this reason. Its hard for algorithms to work with classifier based on complicated functions. It turns out to be easier to just work with linear classifiers and then simply transform the data. Furthermore, if we want more labels, then we simply use multiple linear classifiers.

Of course there are still other more nuanced issues and depending on the data linear classifiers may not be all the helpful. For these reasons there are various other algorithms used in Machine Learning and we will learn many of them. Nonetheless, I hope this has convinced you on why linear classifiers are still important to learn and can be valuable algorithms to have in your knowledge-base.

Perceptron

Perceptron is a beautifully simple algorithm. The general concept is as follows: if you make a mistake adjust the line based on that mistake. More specifically, push the normal vector in the direction of the mistake if it was positively labeled and away if it was negatively labeled.

Note I use notation that follows from a theta vector notation used in 6.036, a class taught at MIT. Other notation involves updating weights for each coordinate. The intuition is the same for both; I find the

In pseudocode it looks as follows:

---

procedure PERCEPTRON($\{\mathbf{x}^{(1)}, \mathbf{x}^{(2)} \cdots \mathbf{x}^{(m)}\}$, $\{y^{(1)}, y^{(2)}, \cdots y^{(m)}\}$)
  $\theta \leftarrow n \times 1$ zero vector ▷ Normal vector defining linear separator
  $\theta_0 \leftarrow 0$ ▷ Offset for linear separator
  while Error on Training Data **do**
    loop Through Training Data
      if $y^{(i)}(\theta \cdot \mathbf{x}^{(i)} + \theta_0) \leq 0$ **then** ▷ Mistake on training data i
        $\theta_0 \leftarrow \theta_0 + y^{(i)}$ ▷ Update offset
        $\theta \leftarrow \theta + y^{(i)}\mathbf{x}^{(i)}$ ▷ Update normal vector
      end if
    end loop
  end while
end procedure

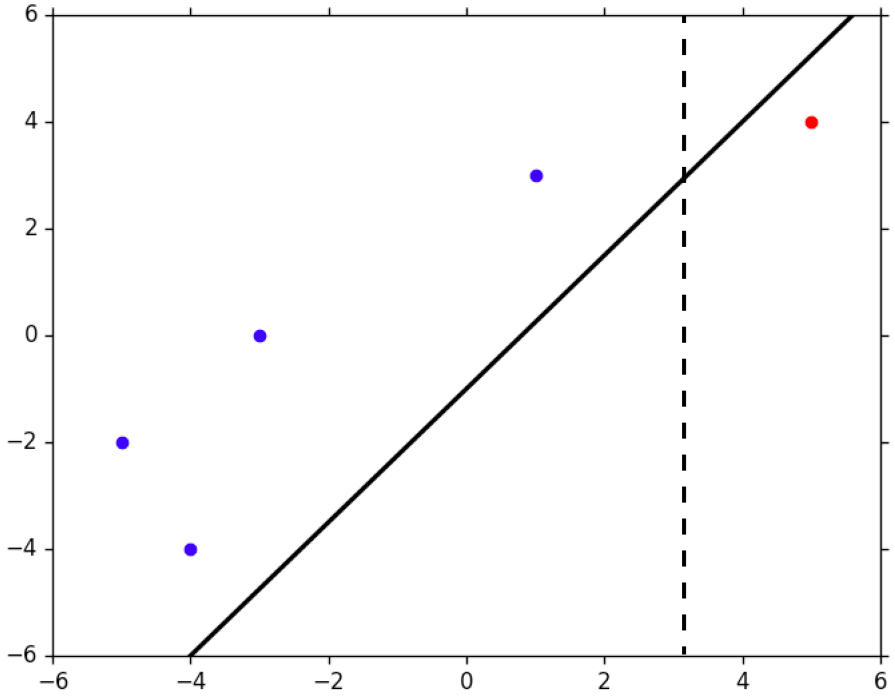
Now with a ridiculously simple algorithm like this there are bound to be questions. 1) Are there guarantees it works? 2) When does this work?

There are guarantees this works if the data is linearly separable. There are notes on the convergence of perceptron in the supplemental notes section.

However, the guarantee simply says the algorithm will converge. Which means it will find **a** linear separator, not necessarily a "good" separator.

The perceptron algorithm is affected by the order the data is processed. Below we have an example where two very different linear separators (one solid, one dashed) are possible given the perceptron algorithm.

Therefore the linear separator is not well-defined (reorderings of the *same* input lead to different outputs) in the perceptron algorithm.



To answer the second question I will point out that this only works if the data is linearly separable. Why is this the case? Well if the data is not linearly separable we will never exit the while loop because there is an error on the training data. It will simply keep adjusting the line based on whatever mistake exists.

As always, we should ask can we do better? Can we solve these issues?

Passive Aggressive

Passive aggressive attempts to solve the issues of the perceptron algorithm. To understand Passive Aggressive we must first return to loss functions.

The loss function used by the perceptron algorithm is called 0-1 loss. 0-1 loss simply means that for each mistaken prediction you incur a penalty of 1 and for each correct prediction incur no penalty.

The problem with this loss function is given a linear classifier its hard to move towards a local optimum. If we plotted the loss function the surface would not be continuous. This makes it difficult to use gradient descent methods.

Stochastic Gradient Descent (SGD) is one of the most powerful and important concepts in Machine Learning. Since in itself stochastic gradient descent is quite an elaborate topic, I will not discuss it in depth here (check the supplemental notes for a full description). Fundamentally SGD optimizes a continuously differentiable function and finds a local optimum given a starting point. One helpful way to visualize this is to imagine a 3-dimensional terrain and putting a ball at some point and letting it roll due to the effects of gravity. It will eventually settle at some local optimum. Stochastic gradient descent basically moves the ball in discrete steps assuming the derivative is constant for that step length. As long as the step size is not too large, SGD will also converge.

Passive-Aggressive is the first method we will learn that uses a loss function that can be optimized with stochastic gradient descent.

First we will define hinge loss, where θ defines the classifier, $\mathbf{x}$ is a data point, and y is a label:

$$\text{Loss}_{hinge}(\theta, \mathbf{x}, y) = \max(1 - y(\theta \cdot \mathbf{x}), 0) \tag{3}$$

Note the hinge loss is continuous and differentiable at all points except when $y(\theta \cdot \mathbf{x}) = 1$. We can fix this by setting the derivative when $y(\theta \cdot \mathbf{x}) = 1$.

So our gradient is therefore as follows:

$$\frac{\partial}{\partial \theta}\text{Loss}_{hinge}(\theta, \mathbf{x}, y) = \begin{cases} -y\mathbf{x} & y(\theta \cdot \mathbf{x}) \leq 1 \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

So the goal of the passive aggressive algorithm is to find the next θ, $\theta^{(k+1)}$ that minimizes:

$$\frac{\lambda}{2}\|\theta^{(k+1)} - \theta^{(k)}\|^2 + \text{Loss}_{hinge}(\theta, \mathbf{x}, y) \tag{5}$$

Now we see the inspiration behind the name passive-aggressive. The name comes from the fact that we want to reduce loss quickly (aggressive) but we also do not want to overcorrect our theta (passive). This is because the first terms is θ^2 and the loss is linear with respect to theta. The θ^2 adds this concept of "strong convexity" which is good for convergence in gradient descent methods.

From the formula for gradient descent we know that our update step looks as follows:

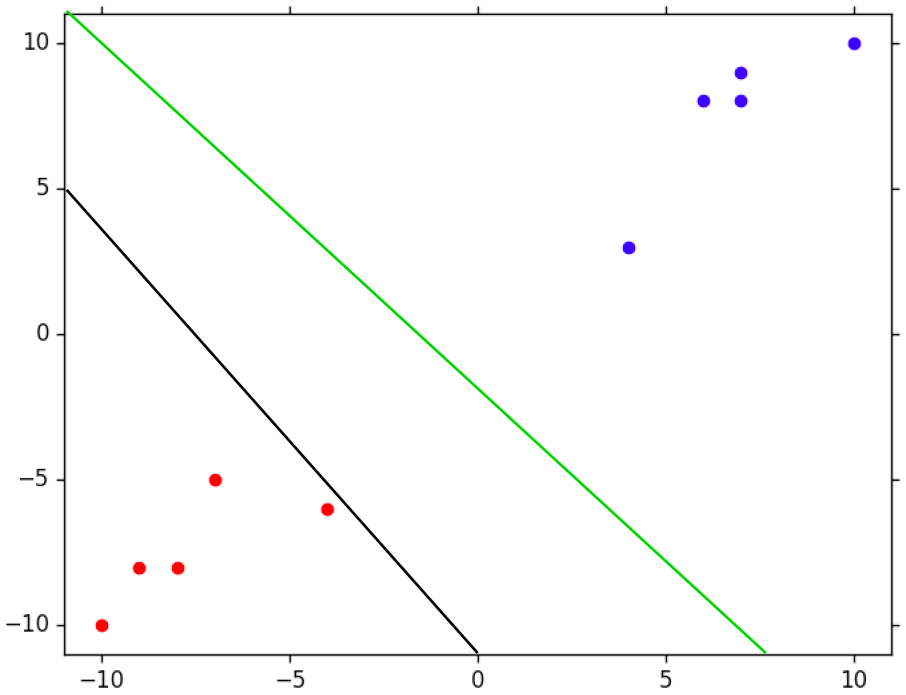
$$\theta^{(k+1)} = \theta^{(k)} - \eta\nabla\text{Loss}_{hinge}(\theta, \mathbf{x}, y) = \theta^{(k)} + \eta y\mathbf{x} \tag{6}$$

---

procedure PassiveAggressive($\{\mathbf{x}^{(1)}, \mathbf{x}^{(2)} \cdots \mathbf{x}^{(m)}\}$, $\{y^{(1)}, y^{(2)}, \cdots y^{(m)}\}$, λ)
  Augment training data by adding a dimension with value 1 ▷ This for offset
  $\theta \leftarrow (n+1) \times 1$ zero vector ▷ Normal vector defining linear separator
  while Gradient is Above Some Threshold **do**
    loop Through Training Data
      if Error on $(\mathbf{x}, y)$ **then**
        $\eta \leftarrow \min(\text{Loss}_{hinge}(\theta, \mathbf{x}, y)/\|\mathbf{x}\|^2, 1/\lambda)$ ▷ Explained in Supplemental Notes
        $\theta \leftarrow \theta + \eta y\mathbf{x}$ ▷ Update normal vector
      end if
    end loop
  end while
end procedure

Note: a derivation can be found in the supplementary Passive Aggressive notes.

Intuition Why is passive aggressive an upgrade from perceptron. One thing is that it converges even if the data is not linearly separable. Another big one is it creates this concept of margins. The hinge loss is zero when all data points also have some amount of separation from the separator. This is important because sometimes just because a line separates most the points well does not mean its a good separator. The figure on the next page illustrates this:



The green line is a better separator because it has a larger margin to both labels while the black line is too close to the red. The advantage of margins is it leads to better generalization. Although these two lines have the same training error, we would expect the test error of the green line to be lower. The final linear classifier we will learn, SVM/Pegasos, achieves this maximum margin separation.

SVM, Pegasos

We will focus on a specific case (soft-case) where we use hinge loss but in general the goal of support vector machine is to minimize the following objective function:

$$J(\theta) = \frac{1}{n}\sum_{i=1}^{n} \text{Loss}_{hinge}(\theta, \mathbf{x}, y) + \frac{\lambda}{2}\|\theta\|^2 \tag{7}$$

Once again we optimize using gradient descent. We will actually cover two examples. One is offline learning. In this case we know how many examples we have. The other is online. Online learning is extremely powerful because it means you can stream through training data and train your algorithm in the process. Its particularly useful when the data sample is large and easier to stream that have stored locally.

The offline solution follows simply:

$$\theta^{(k+1)} = \theta^{(k)} - \eta \nabla J(\theta^{(k)}) \tag{8}$$

To get the online version we have to look at an example at a time. On a given example we have:

procedure SVM($\{\mathbf{x}^{(1)}, \mathbf{x}^{(2)} \cdots \mathbf{x}^{(m)}\}$, $\{y^{(1)}, y^{(2)}, \cdots y^{(m)}\}$, η, λ)
  Augment training data by adding a dimension with value 1 ▷ This for offset
  $\theta \leftarrow (n+1) \times 1$ zero vector ▷ Normal vector defining linear separator
  while Gradient is Above Some Threshold **do**
    loop Through Training Data
      $\theta \leftarrow (1 - \eta\lambda)\theta + \eta \frac{1}{n} \sum_{i \in \text{mistakes}} y^{(i)} \mathbf{x}^{(i)}$ ▷ Update normal vector
    end loop
  end while
end procedure

$$\nabla J(\theta) = \begin{cases} -y\mathbf{x} + \lambda\theta & y(\theta \cdot \mathbf{x}) \leq 1 \\ \lambda\theta & \text{otherwise} \end{cases} \tag{9}$$

$$\theta^{(k+1)} = \theta^{(k)} - \eta \nabla J(\theta) = \begin{cases} (1 - \eta\lambda)\theta^{(k)} + \eta y\mathbf{x} & y(\theta \cdot \mathbf{x}) \leq 1 \\ (1 - \eta\lambda)\theta^{(k)} & \text{otherwise} \end{cases} \tag{10}$$

This gives us the online algorithm Pegasos. Note Pegasos simply rescales the previous updates to at any given time what its seen acts like the full batch in SVM.

procedure PEGASOS(Data Stream, λ)
  $\theta \leftarrow (n+1) \times 1$ zero vector ▷ Normal vector defining linear separator
  $i = 1$
  loop Go Through Stream
    $\eta = 1/i$
    if $y(\theta \cdot \mathbf{x}) \leq 1$ **then**
      $\theta \leftarrow (1 - \eta\lambda)\theta^{(k)} + \eta y\mathbf{x}$ ▷ Update normal vector
    else
      $\theta \leftarrow (1 - \eta\lambda)\theta^{(k)}$
    end if
    $i = i + 1$
  end loop
end procedure

Basic Algorithms: Clustering

Goals and Motivations

Linear classifiers represent a type of supervised learning. We know what we want as an output and use that to direct the algorithm. Clustering represents unsupervised learning. Instead of telling the algorithm what we want we simply ask it to find relationships within the data. Clustering in particular simply tells us what data points are similar.

Why is unsupervised learning useful? Hinton does a good job of explaining this in his course on deep learning. In summary, the major point is that unsupervised learning methods a) represent the data by finding relations in lower dimensions than the dimension of the original data b) highlight the substructure of the data c) can learn a wider range of behavior since undirected by supervision.

We will now cover a simple algorithm and then explore a paradigm that fixes the issues associated with algorithms of the first kind.

K-Means

Nearest neighbors is arguably one of the simplest algorithms. You start with deciding how many clusters you want. Then you pick K random start means. Each data point is assigned to the mean it is closest too. After all data points are assigned, the mean is recomputed based on the data points assigned to it. The process continues until convergence (the means have the same points reassigned to them). To see the K-means algorithm work in a demo check the K-means demo in the resources section.

The main plus of K-means is its simple. But it does have many major issues. In general a random initialization means that the convergence is not unique. Similar to perceptron vs. SVM we would prefer an optimum as opposed to a large set of possible convergences, some of which could be poor. This is important for generalization so the test error is also low. Furthermore, and arguably a more important problem of clustering algorithms like K-means is the K. You have to specify the number of clusters before hand. Now of course you could try different K and come up with some metric that optimizes for a combination of the similarity of the cluster and the number of clusters.

Hierarchical clustering offers a solution to the issues of K-means. Check the supplemental notes on hierarchical clustering.
