[MITx 6.86x Notes Index]

Unit 01 - Linear Classifiers and Generalizations

Course Introduction

There are lots and lots of applications out there, but what’s so interesting about it is that, in terms of algorithms, there is a relatively small toolkit of algorithms you need to learn to understand how these different applications can work so nicely and be integrated in our daily life.

The course covers these 3 topics

Supervised Learning

- Make predictions based on a set of data for which correct examples are given
- E.g. Attach the label “dog” to an image, based on many couple (image/dog label) already provided
- The correct answers of the examples provided to the algorithm are already given
- What to produce is given explicitly as a target
- First 3 units in increasing complexity from simple linear classification methods to more complex neural network algorithms

Unsupervised Learning

- A generalisation of supervised learning
- The target is no longer given
- “Learning” to produce output (still at the end, “make predictions”), but now just observing existing outputs, like images, or molecules
- Algorithms to learn how to generate things
- We’ll talk about probabilistic models and how to generate samples, mix and match (???)

Reinforced Learning

- Suggest how to act in the world given a certain goal, how to achieve an objective optimally
- It involves making predictions, aspects of SL and UL, but also it has a goal
- Learning from the failures, how to do things better
- E.g. a robot arm trying to grasp an object

- Objective is to control a system

Interesting stuff (solving real problems) happens at the mix of these 3 categories.

Lecture 1. Introduction to Machine Learning

1.1. Unit 1 Overview

Unit 1 -> Linear classification

Exercise: predicting whether an Amazon product review, looking at only the text, will be positive or negative.

1.2. Objectives

Introduction to Machine Learning

At the end of this lecture, you will be able to:

- understand the goal of machine learning from a movie recommender example
- understand elements of supervised learning, and the difference between the training set and the test set
- understand the difference of classification and regression - two representative kinds of supervised learning

1.3. What is Machine Learning?

Many examples: Interpretation of web search queries, movie reccomendations, digital assistants, automatic translations, image analysis… playing the go game

Machine learning as a discipline aims to design, understand, and apply computer programs that learn from experience (i.e. data) for the purpose of **modelling**, **prediction**, and **control**.

There are many ways to learn or many reasons to learn. You can try to model, understand how things work. You can try to predict about, say, future outcomes or try to control towards a desired output configuration.

Prediction

We will start with prediction as a core machine learning task. There are many types of predictions that we can make.:

- We can predict outcomes of *events that occur in the future* such as the market, weather tomorrow, the next word a text message user will type, or anticipate pedestrian behavior in self driving vehicles, and so on.
- We can also try to predict *properties that we do not yet know*. For example, properties of materials such as whether a chemical is soluble in water, what the object is in an image, what an English sentence translates to in Hindi, whether a product review carries positive sentiment, and so on.

1.4. Introduction to Supervised Learning

Common to all these “prediction problems" mentioned previously is that they would be very hard to solve in a traditional engineering way, where we specify rules or solutions directly to the problem. It is far easier to provide examples of correct behavior. For example, how would you encode rules for translation, or image classification? It is much easier to provide large numbers of translated sentences, or examples of what the objects are on a large set of images. I don’t have to specify the solution method to be able to illustrate the task implicitly through examples. The ability to learn the solution from examples is what has made machine learning so popular and pervasive.

We will start with supervised learning in this course. In supervised learning, we are given an example (e.g. an image) along with a target (e.g. what object is in the image), and the goal of the machine learning algorithm is to find out how to produce the target from the example.

More specifically, in supervised learning, we hypothesize a collection of functions (or mappings) parametrized by a parameter (actually a large set of parameters), from the examples (e.g. the images) to the targets (e.g. the objects in the images). The machine learning algorithm then automates the process of finding the parameter of the function that fits with the example-target pairs the best.

We will automate this process of finding these parameters, and also specifying what the mappings are.

So this is what the machine learning algorithms do for you. You can apply the same approach in different contexts.

1.5. A Concrete Example of a Supervised Learning Task

Let’s consider a movie recommender problem. I have a set of movies I’ve already seen (the **training set**), and I was to use the experience from those movies to make recommendations of whether I would like to see tens of thousands of other movies (the **test set**).

We will compile a **feature vector** (a binary list of its characteristics… genre, director, year…) for each movie in the training set as well for those I want to test (we will denote these feature vectors as x).

I also give my recommendation (again binary) if I want to see again or not the films in the training set.

Now I have the training set (feature vectors with associated labels), but I really wish to solve the task over the test set. It is this discrepancy that makes the learning task interesting. I wish to generalize what I extract from the training set and apply that information to the test set.

More specifically, I wish to learn the mapping from x to labels (plus minus one) on the basis of the training set, and hope and guarantee that that mapping, if applied now in the same way to the test examples, it would work well.

I want to predict the films in the test set based on their characteristics and my previous recommendation on the training set.

1.6. Introduction to Classifiers: Let's bring in some geometry!

On the basis of the training set, pairs of feature vectors associated with labels, I need to learn to map each each x, each future vector, to a corresponding label, which is a plus or minus 1.

What we need is a **classifier** (here denoted with h) that maps from points to corresponding labels.

So what the classifier does is divide the space into essentially two halves-- one that's labeled 1, and the other one that's labeled minus 1. –> a "linear classifier" is a classifier that perform this division of the full space in the two half linearly

We need to now, somehow, evaluate how good the classifier is in relation to the training examples that we have. To this end, we define something called **training error** (here denoted with ϵ) It has a subscript n that refers to the number of training examples that I have. And I apply it to a particular classifier. I evaluate a particular classifier, in relation to the training examples that I have.

$$\epsilon_n(h) = \sum_{i=1}^{n} \frac{[\![h(x^i) \neq y^i]\!]}{n}$$

(The double brackets denotes a function that takes a comparison expression inside and returns 1 if the expression is true, 0 otherwise. It is basically an indicator function.)

In the second example of classifier given in the lesson, the training error is 0. So, according to just the training examples, this seems like a good classifier.

The fact that the training error is zero however doesn't mean that the classifier is perfect!

Let's now consider a non-linear classifier.

We can build an “extreme” classifier that accept as +1 only a very narrow area around the +1 training set points.

As a result, this classifier would still have training error equal to zero, but it would classify all the points in the test set, no matter how much close to the +1 training set points they are, as -1 !

what is going on here is an issue called **generalization**– how well the classifier that we train on the training set generalizes or applies correctly, similarly to the test examples, as well? This is at the heart of machine-learning problems, the ability to generalize from the training set to the test set.

The problem here is that, in allowing these kind of classifiers that wrap themselves just around the examples (in the sense of allowing any kind of non-linear classifier), we are making the **hypothesis class**, the set of possible classifiers that we are considering, too large. We improve generalization, we generalize well, when we only have a very limited set of classifiers. And, out of those, we can find one that works well on the training set.

The more complex set of classifiers we consider, the less well we are likely to generalize (this is the same concept as overfitting in a statistical context).

We would wish to, in general, solve these problems by finding a small set of possibilities that work well on the training set, so as to generalize well on the test set.

Training data can be graphically depicted on a (hyper)plane. Classifiers are mappings that take feature vectors as input and produce labels as output. A common kind of classifier is the linear classifier, which linearly divides space (the hyperplane where training data lies) into two. Given a point x in the space, the classifier h outputs $h(x) = 1$ or $h(x) = -1$, depending on where the point x exists in among the two linearly divided spaces.

Each classifier represents a possible “hypothesis" about the data; thus, the set of possible classifiers can be seen as the space of possible hypothesis

1.7. Different Kinds of Supervised Learning: classification vs regression

A supervised learning task is one where you have specified the correct behaviour. Examples are then feature vector and what you want it to be associated with. So you give “supervision” of what the correct behaviour is (from which the term).

Types of supervised learning:

- Multi-way classification : $h : \mathrm{X} \rightarrow \text{finite set}$
- Regression: $h : \mathrm{X} \rightarrow R$
- Structured prediction: $h : \mathrm{X} \rightarrow \text{structured object, like a language sentence}$

Types of machine learning:

- Supervised learning
- Unsupervised learning: You can observe, but the task itself is not well-defined. The problem there is to model, to find the irregularities in how those examples vary.
- Semi-supervised learning
- Active learning (the algorithm asks for useful additional examples)
- Transfer learning
- Reinforcement learning

The training set is illustration of the task, the test set is what we wish to really do well on. But those are examples that we don't have available at the time we need to select the mapping.

Classification maps feature vectors to categories. The number of categories need not be two - they can be as many as needed. Regression maps feature vectors to real numbers. There are other kinds of supervised learning as well.

Fully labelled training and test examples corresponds to supervised learning. Limited annotation is semi-supervised learning, and no annotation is unsupervised learning. Using knowledge from one task on another task means you're "transferring" information. Learning how to navigate a robot means learning to act and optimize your actions, or reinforcement learning. Deciding which examples are needed to learn is the definition of active learning.

Lecture 2. Linear Classifier and Perceptron Algorithm

2.1. Objectives

At the end of this lecture, you will be able to

- understand the concepts of Feature vectors and labels, Training set and Test set, Classifier, Training error, Test error, and the Set of classifiers
- derive the mathematical presentation of linear classifiers
- understand the intuitive and formal definition of linear separation
- use the perceptron algorithm with and without offset

2.2. Review of Basic Concepts

A supervised learning task is when you are given the input and the corresponding output that you want, and you're supposed to learn irregularity between the two in order to make predictions for future examples, or inputs, or feature vectors x.

Test error is defined exactly similarly over the test examples. So defined similarly to the training error, but over a disjoint set of examples, those future examples that you actually wish to do well.

We typically drop the n on it, assuming that the test set is relatively large,

Much of machine learning, really, the theory part is in relating how a classifier that might do well on the training set would also do well on the test set. That's the problem called Generalization, as we've already seen. We can effect generalization by limiting the choices that we have at the time of considering minimizing the training error.

So our classifier here belongs to a **set of classifiers** (here denoted with H), that's not the set of all mappings, but it's a limited set of options that we constrain ourselves to.

The trick here is to somehow guide the selection of the classifier based on the training example, such that it would do well on the examples that we have not yet seen.

In order for this to be possible at all, you have to have some relationship between the training samples and the test examples. Typically, it is assumed that both sets are samples from some large collection of examples as a random subset. So you get a random subset as a training set.

What is this lecture going to be about?

This lecture we will consider only linear classifiers, such that we keep the problem well generalised (we will return to the question of generalization more formally later on in this course).

We're going to formally define the set of linear classifiers, the set H restricted set of classifiers. We need to introduce parameters that index classifiers in this set so that we can search over the possible classifiers in the set.

We'll see what's the limitation of using linear classifiers.

We'll next need to define the learning algorithm that takes in the training set and the set of classifiers and tries to find a classifier, in that set, that somehow best fits the training set.

We will consider initially the **perceptron algorithm**, which is a very simple online mistake driven algorithm that is still useful as it can be generalized to high dimensional problems. So perceptron algorithm finds a classifier $\hat{h}$, where hat denotes an estimate from the data. It's an algorithm that takes, as an input, the training set and the set of classifiers and then returns that estimated classifier,

2.3. Linear Classifiers Mathematically Revisited

The dividing line here is also called **decision boundary**. In 2D, that decision boundary is a line. If x was a one-dimensional quantity, the decision boundary would be a point. In 3D, it would be a plane. And in higher dimensions, it's called hyperplane that divides the space into two halves.

So now we need to parameterize the linear classifiers, so that we can effectively search for the right one given the training set.

Through the origin classifiers

Definition of a decision boundary through the origin:

All points x such that $x: g_1 * x_1 + g_2 * x_2 = 0$ or, calling θ the vector of coefficients and x the vector of the points, $x: \theta \cdot x = 0$.

Note that this means that the vector θ is ortoghonal to the positional vectors of the points in the boundary.

The classifier is now parametrised by theta: $h(x;\theta)$ So each choice of theta defines one classifier. You change theta, you get a different classifier. It's oriented differently. But it also goes through origin.

Our classifier become: $h(x;\theta) = sign(\theta \cdot x)$

Note that this association between the classifier and the parameter vector theta is not unique. There are multiple parameter vectors theta that defined exactly the same classifier. the norm of the parameter vector of theta is not relevant in terms of the decision boundary.

General linear classifiers

Decision boundary: $x: \theta \cdot x + \theta_0 = 0$.

Our classifier becomes: $h(x;\theta,\theta_0) = sign(\theta \cdot x + \theta_0)$

2.4. Linear Separation

A training set is said to be **linearly separable** if it exists a linear classifier that correctly classifies all the training examples (i.e. if parameter $\hat{\theta}$ and $\hat{\theta}_0$ exists such that $y^i * (\hat{\theta} \cdot x^i + \hat{\theta}_0) > 0 \;\; \forall i = 1, \ldots, n$).

Given θ and θ_0, a linear classifier $h: X \rightarrow \{-1, 0, +1\}$ is a function that outputs $+1$ if $\theta \cdot x + \theta_0$ is positive, 0 if it is zero, and -1 if it is negative. In other words, $h(x) = sign(\theta \cdot x + \theta_0)$.

2.5. The Perceptron Algorithm

When the output of a classifier is exactly 0, if the example lies exactly on the linear boundary, we count that as an error, since we don't know which way we should really classify that point.

Training error for a linear classifier:

$\epsilon_n(h) = \sum_{i=1}^{n} \frac{[\![h(x^i) \neq y^i]\!]}{n}$

become:

$$\epsilon_n(\theta, \theta_0) = \sum_{i=1}^{n} \frac{[[y^i * (\theta \cdot x^i + \theta_0) \leq 0]]}{n}$$

We can now turn to the problem of actually finding a linear classifier that agrees with the training examples to the extent possible.

Through the origin classifier

- We start with a $\theta = 0$ (vector) parameter
- We check if, with this parameter, the classifier make an error
- If so, we progressively update the classifier

The update function is $\theta^i = \theta^{i-1} + y^i * x^i$.

As we start with $\theta^0 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$, the first attempt is always leading to an error and to a first "update" that will be $\theta^1 = \begin{bmatrix} 0 \\ 0 \end{bmatrix} + y^1 * x^1$.

For example, given the pair $(x^1 = \begin{bmatrix} 2 \\ 4 \end{bmatrix}, y^1 = -1)$, θ_1 becomes

$$\theta^1 = \begin{bmatrix} 0 \\ 0 \end{bmatrix} + -1 * \begin{bmatrix} 2 \\ 4 \end{bmatrix} = \begin{bmatrix} -2 \\ -4 \end{bmatrix}$$

Its error is then $\epsilon_n(\theta^1) = [[y^i * (\theta^1 \cdot x^i)) \leq 0]] = [[((y^i)^2 * ||x^i||^2) \leq 0]] = 0$

Now, what are we going to do here over the whole training set, is we start with the 0 parameter vector and then go over all the training examples. And if the i-th example is a mistake, then we perform that update that we just discussed.

So we'd nudge the parameters in the right direction, based on an individual update. Now, since the different training examples might update the parameters in different directions, it is possible that the later updates actually make the earlier undo some of the earlier updates, and some of the earlier examples are no longer correctly classified. In other words, there may be cases where the perceptron algorithm needs to go over the training set multiple times before a separable solution is found.

So we have to go through the training set here multiple times, here capital T times go through the training set, either in order or select at random, look at whether it's a mistake, and perform a simple update.

- function perceptron $\left(\left\{(x^{(i)}, y^{(i)}), i = 1, \ldots, n\right\}, T\right)$:
    - initialize $\theta = 0$ (vector);
    - for $t = 1, \ldots, T$ do

```
        - for $i = 1, \dots, n$ do
            - if $y^{(i)}(\theta \cdot x^{(i)}) \leq 0$ then
                - update $\theta = \theta + y^{(i)} x^{(i)}$
    - return $\theta$
```

So the perceptron algorithm takes two parameters: the training set of data (pairs feature vectors => label) and the T parameter that tells you how many times you try to go over the training set.

Now, this classifier for a sufficiently large T, if there exists a linear classifier through origin that correctly classifies the training samples, this simple algorithm actually will find a solution to that problem. There are many solutions typically, but this will find one. And note that the one found is not, generally, some "optimal" one, where the points are "best" separated, just one where the points *are* separated.

Generalised linear classifier

we can generalize the perceptron algorithm to run with the general class of linear classifiers with the offset parameter. The only difference here is that now we initialize the parameter back to 0, as well as the scalar to 0, that we consider also θ_0 in the error check, that we update θ_0 as well and that we return it together with θ:

```
- function perceptron $\left(\{(x^{(i)}, y^{(i)}), i = 1, \dots, n\}, T\right)$:
    - initialize $\theta = 0$ (vector), $\theta_0 = 0$ (scalar);
    - for $t = 1, \dots, T$ do
        - for $i = 1, \dots, n$ do
            - if $y^{(i)}(\theta \cdot x^{(i)} + \theta_0) \leq 0$ then
                - update $\theta = \theta + y^{(i)} x^{(i)}$
                - update $\theta_0 = \theta_0 + y^{(i)}$
    - return $\theta, \theta_0$
```

The update of θ_0 can be seen as the update of a linear model trough the origin, where the offset is a further dimension of the data:

The model $\theta \cdot x + \theta_0$ can be then see equivalently as $\begin{bmatrix} \theta \\ \theta_0 \end{bmatrix} \cdot \begin{bmatrix} x \\ 1 \end{bmatrix}$.

The update function $\theta = \theta + x^i * y^i$ becomes then $\begin{bmatrix} \theta \\ \theta_0 \end{bmatrix} = \begin{bmatrix} \theta \\ \theta_0 \end{bmatrix} + y^i * \begin{bmatrix} x \\ 1 \end{bmatrix}$ from which, going back to our original model, we obtain the given update functions.

So now we have a general learning algorithm. The simplest one, but it can be generalized to be quite powerful and therefore, hence it is a useful algorithm to understand.

Lecture 3 Hinge loss, Margin boundaries and Regularization

3.1. Objective

Hinge loss, Margin boundaries, and Regularization

At the end of this lecture, you will be able to

- understand the need for maximizing the margin
- pose linear classification as an optimization problem
- understand hinge loss, margin boundaries and regularization

3.2. Introduction

Today, we will talk about how to turn machine learning problems into optimization problems. That is, we are going to turn the problem of finding a linear classifier on the basis of the training set into an optimization problem that can be solved in many ways. Today's lesson we will talk about what linear large margin classification is and introduce notions such as margin loss and regularization.

Why optimisation ?

The perceptron algorithm choose a correct classifier, but there are many possible correct classifiers. Between them, we would favor the solution that somehow is drawn between the two sets of training examples, leaving lots of space on both sides before hitting the training examples. This is known as a **large margin classifier**.

A large margin linear classifier still correctly classifies all of the test examples, even if these are noisy samples of unknown true values. A large margin classifier in this sense is more robust against noises in the examples. In general, large margin means good generalization performance on test data.

How to find such large margin classifier? --> optimisation problem

We consider the parallel plane to the classifier passing by the closest positive and negative point as respectively positive and negative **margin boundaries**, and we want them to be equidistant from the classifier.

The goal here is now to use these margin boundaries essentially to define a fat decision boundary that we will still try to fit with the training examples. So we will push these margin boundaries apart that will force us to reorient there and move that system boundary into a place that carves out this large empty space between the two sets of examples.

We will minimise an objective function made of two terms, the **regularisation term**, that push to set the boundaries as much apart as possible, but also a **loss function term**, that gives us the penalty from points that now, as the boundaries extend, got trapped inside the margin or even more to the other part (becoming misclassified)

3.3. Margin Boundary

The first thing that we must do is to define what exactly the margin boundaries are and how we can control them, how far they are from the decision boundary. Remember that they are equidistant from the decision boundary.

Given the linear equation $\theta \cdot x + \theta_0 = k$, the decision boundary is the set of points where $k = 0$, the positive margin boundary is the set of points where k is some positive constant and the negative margin boundary is where k is some negative constant.

The equation of the decision boundary can be wrote in normalised version by dividing it by the norm of the θ vector:

$$\frac{\theta}{||\theta||} \cdot x + \frac{\theta_0}{||\theta||} = \frac{k}{||\theta||}$$

3.4. Hinge Loss and Objective Function

Obj is to maximise the distance of the decision boundary from the marginal boundaries, $k/||\theta||$ for the choosen k (plus/minus 1 in the lesson). This is equivalent to minimise $||\theta||$, in turn equivalent to minimize $\frac{||\theta||^2}{2}$

We can define the **Hinge loss** function as a function of the amount of *agreement* (here denoted with z) between the classifier score and the data given by an equation we already saw: $y^i * (\theta \cdot x^i + \theta_0)$.

Previously, in the perceptron algorithm, we used the indicator function of such agreement in the definition of the error ϵ.

Now we use its value, as the loss function is defined as:

$loss(z) = 0$ if $z \geq 1$ (the point is outside the boundary, on the right direction) and $loss(z) = 1 - z$ if $z < 1$ (either the point is well classified, but inside the boundary - $0 \leq z < 1$ - or it is even misclassified - $z \leq 0$).

Note that while for a point on the decision boundary, being exactly on the decision boundary implies a misclassification, being on the (right) margin boundary implies a zero loss.

A bit of terminology

- **Support vectors** are the data points that lie closest to the decision surface (or hyperplane). They are the data points most difficult to classify but they have direct bearing on the optimum location of the decision surface.
- **Support-vector machines** (SVMs, also support-vector networks) are supervised learning models with associated learning algorithms that analyze data used for classification and regression analysis. Given a set of training examples, each marked as belonging to one or the other of two categories, an SVM training algorithm builds a model that assigns new examples to one category or the other, making it a non-probabilistic binary linear classifier (SVMs can perform also non-linear classification using what is called the kernel trick, implicitly mapping their inputs into high-dimensional feature spaces).
- In mathematical optimization and decision theory, a **Loss function** or cost function is a function that maps an event or values of one or more variables onto a real number intuitively representing some "cost" associated with the event. An optimization problem seeks to minimize a loss function. An objective function is either a loss function or its negative (in specific domains, variously called a reward function, a profit function, a utility function, a fitness function, etc.), in which case it is to be maximized.
- The **Hinge loss** is a specific loss function used for training classifiers. The hinge loss is used for "maximum-margin" classification, most notably for support vector machines (SVMs). For an intended output $y = \pm 1$ and a classifier score s, the hinge loss of the prediction s is defined as $\ell(s) = \max(0, 1 - y \cdot s)$. Note that s should be the "raw" output of the classifier's decision function, not the predicted class label. For instance, in linear SVMs, $s = \theta \cdot \mathbf{x} + \theta_0$, where $\mathbf{x}$ is the input variable(s). When y and s have the same sign (meaning s predicts the right class) and $|s| \geq 1$, the hinge loss $\ell(s) = 0$. When they have opposite signs, $\ell(s)$ increases linearly with s, and similarly if $|s| < 1$, even if it has the same sign (correct prediction, but not by enough margin).

We are now ready to define the objective function (to minimise) as:

$$J(\theta, \theta_0) = \frac{1}{n} \sum_{i=1}^{n} \text{Loss}_h(y^{(i)} * (\theta \cdot x^{(i)} + \theta_0)) + \frac{\lambda}{2} \mid\mid \theta \mid\mid^2$$

Now, our objective function how we wish to guide the solution is a balance between the two. And we set the balance by defining a new parameter called **regularization parameter** (in the previous equation denoted by λ) that simply weighs how these two terms should affect our solution. Regularization parameter here is always greater than 0.

Greater λ : we will favor large margin solutions but potentially at a cost of incurring some further loss as the margin boundaries push past the examples.

Optimal value of θ and θ_0 is obtained by minimizing this objective function. So we have turned the learning problem into an optimization problem.

Homework 1

1. Perceptron Mistakes

1. (e) Perceptron Mistake Bounds

Novikoff Theorem (1962): https://arxiv.org/pdf/1305.0208.pdf

Assumptions:

- There exists an optimal θ^* such that $\frac{y^{(i)}(\theta^* x^{(i)})}{\|\theta^*\|} \geq \gamma \forall i = 1, \ldots, n \text{ and some } \gamma > 0$ (the data is separable by a linear classifier through the origin)
- All training data set examples are bounded by being $\|x^{(i)}\| \leq R, i = 1, \cdots, n$

Predicate:

Then the number of k updates of the perceptron algorithm is bounded by $k < \frac{R^2}{\gamma^2}$

Lecture 4. Linear Classification and Generalization

4.1. Objectives

At the end of this lecture, you will be able to:

- understanding optimization view of learning
- apply optimization algorithms such as gradient descent, stochastic gradient descent, and quadratic program

4.2. Review and the Lambda parameter

Last time (lecture 3), we talked about how to formulate maximum margin linear classification as an optimization problem. Today, we're going to try to understand the solutions to that optimization problem and how to find those solutions.

The objective function to minimise is still

$$J(\theta, \theta_0) = \frac{1}{n} \sum_{i=1}^{n} \text{Loss}_h(y^{(i)} * (\theta \cdot x^{(i)} + \theta_0)) + \frac{\lambda}{2} \mid\mid \theta \mid\mid^2$$

where the first term is the average loss and the second one is the regularisation.

The average loss is what we are trying to minimize.

Minimizing the regularization term will try instead to push the margin boundaries further and further apart.

And the balance between these two are controlled by the regularization parameter lambda.

Minimising the regularisation term $\frac{\lambda}{2} \mid\mid \theta \mid\mid^2$ we maximise the distance of the margin boundary $\frac{1}{||\theta||}$.

And as we do that, we start hitting the actual training examples. The boundary needs to orient itself, and we may start incurring losses.

The role of the lambda parameter

As we change the regularization parameter, lambda, we change the balance between these two terms. The larger the value lambda is, the more we try to push the margin boundaries apart; the smaller it is, the more emphasis we put on minimizing the average laws on the training example, reducing the distance of the margin boundaries up to the extreme $\lambda = 0$ where the margin boundaries themselves collapse towards the actual decision boundary.

The further and further the margin boundaries are posed, the more the solution will be guided by the bulk of the points, rather than just the points that are right close to the decision boundary. So the solution changes as we change the regularization parameter.

In other terms:

- $\lambda = 0$: if we consider a minimisation of only the loss function, without the margin boundaries, (admitting a separable problem) we would have infinite solutions within the right classifiers (similarly to the perceptron algorithm);
- $\lambda = \epsilon^+$: With a very small (but positive) λ, the margin boundaries would be immediately pushed to the first point(s) at the minimal distance to the classifier (aka the "support vectors", from which the name of this learning method), and only this point(s) would drive the "optimal" classifier within all the possible ones (we overfit relatively to these boundary points);
- $\lambda = \text{large}^+$: As we increase λ, we are starting to give weight also to the points that are further apart, and the optimal classifier may be changing as a consequence, eventually even misclassifying some of the closest points, if there are enough far points pushing for such classifier.

4.3. Regularization and Generalization

We minimise a new function $J/\lambda = \frac{1}{\lambda} * \frac{1}{n} \sum_{i=1}^{n} \text{Loss}_h(\cdot) + \frac{1}{2} \mid\mid \theta \mid\mid^2$ (as λ is positive minimising the new function is equal to minimising the old one).

We can think the objective function as function of the $1/\lambda$ ratio (that we denote as "c"):

- $1/\lambda$ small -> wide margins, high losses

- $1/\lambda$ big -> narrow margins, small/zero losses

Such function, once minimised, will result in optimal classifiers $h(\theta^*, \theta_0^*, c)$ from which I can compute the average loss. Such average loss would be a convex, monotonically decreasing function of c, although it would not reach zero losses even for very large values of c if the problem is not linearly separable.

If we could measure also the test loss (from the test set) obtained from such optimal classifiers, we would find a typical u-shape where test loss first decreases with c, but then would growth up again. Also, the test loss (or "error") would be always higher than the training loss, as we are fitting from the training set, not the test set.

We call c^* the "optimal" c value that minimise the test error. If we could find it, we would have found the value of λ that best generalise the method.

Now, we cannot do that, but we will talk about how to approximately do that.

With c below c^* I am underfitting, with c above c^* I am overfitting (around the point(s) closest to the decision margin). How to find c^* ? By dividing my training set in a "new" training set, and a (smaller) **validation set**, our new "pretending" test set.

Using this "new" training set we can find the optimal classifier as function of c, and at that point use $h(\theta^*, \theta_0^*, c)$ to evaluate the **validation loss** (or "validation error") to numerically find an approximate value of c^* .

4.4. Gradient Descent

Objective: So far we have seen how to qualitatively understand the type of solutions that we get when we vary the regularization parameter and optimize with respect to theta and theta naught. Now, we are going to talk about, actually, algorithms for finding those solutions.

Gradient descent algorithm to find the minimum of a function $J(\theta)$ with respect to a parameter $\theta \in \mathbb{R}$ (for simplicity).

- I start with a given θ_{start}
- I evaluate the derivative $\partial J/\partial\theta$ at θ_{start}, i.e. the slope of the function in θ_{start}. If this is positive, increasing θ will increase the J function and the opposite if it is negative. In both cases, if I move θ in the opposite direction of the slope, I move toward a lower value of the function.
- My new, updated value of θ is then $\theta = \theta - \eta\frac{\partial J}{\partial\theta}$ where η is known as the **step size** or **learning rate**, and the whole equation as **gradient descent update rule**.
- We stop when the variation in θ goes below a certain threshold.

If η is too small: we may converge very slowly or end up trapped in small local minima;

If η is too small: we may diverge instead of converge to the minimum (going to the opposite direction respect to the minimum).

If the function is strictly convex, then there would be a constant learning rate small enough that I am guaranteed quickly to get to the minimum of that function.

Multi-dimensional case

The multi-dimensional case is similar, we update each parameter with respect to the corresponding partial derivative (i.e. the corresponding entry in the gradient):

$\theta_j = \theta_j - \eta \frac{\partial J}{\partial \theta_j}$ or, in vector form, $\theta = \theta - \eta \nabla J(\theta)$

(where $\nabla J(\theta_\mathbf{j})$ is the gradient of J, i.e. the column vector concatenating the partial derivatives with respect to the various parameters)

4.5. Stochastic Gradient Descent

Let firs note that we can write $\frac{1}{n}\sum_{i=1}^{n}(a_i) + b$ as $\frac{1}{n}\sum_{i=1}^{n}(a_i + b)$.

Our original objective function can hence be written as

$J(\theta, \theta_0) = \frac{1}{n}\sum_{i=1}^{n}\left[\text{Loss}_h(y^{(i)} * (\theta \cdot x^{(i)} + \theta_0)) + \frac{\lambda}{2} \mid\mid \theta \mid\mid^2\right]$

Let's also, for now, simplify the calculation omitting the offset parameter:

$J(\theta) = \frac{1}{n}\sum_{i=1}^{n}\left[\text{Loss}_h(y^{(i)} * (\theta \cdot x^{(i)})) + \frac{\lambda}{2} \mid\mid \theta \mid\mid^2\right]$

We can now see the above objective function as an *expectation* of the function $J(\theta)$: $E[J(\theta)] = \frac{1}{n}\sum_{i=1}^{n} J[\theta]$ where t

We can then use a stochastic approach (SGD, standing for **Stochastic Gradient Descen**), where we sample at random from the training set a single data pair i, we compute the gradient of the objective function with respect to that single parameter $\frac{\partial J_i(\theta)}{\partial \theta}$, we compute an update of theta following the same rule as before ($\theta = \theta - \eta_t \nabla J_i(\theta)$), we sample an other data pair j, we re-compute the gradient with respect to this new pair, we update again ($\theta = \theta - \eta_t \nabla J_j(\theta)$), and so on.

This has the advantage to be more efficient than taking all the data, compute the objective function and the gradients with respect to all the data points (could be milions!), and perform the gradient descent with respect to such function.

By sampling we introduce however some stochasticity and noises, and hence we need to put care on the learning parameter that need to be reduced to avoid to diverge.

The new learning rate η needs to:

- Be lower than the one we would employ for the deterministic gradient descent algorithm in order to reduce the variance from the stochasticity that we're introducing:
    - reduce at each update t with the limit as the steps go to infinite to go to zero $\lim_{t\to\infty} \eta_t = 0$
    - be such that are "square summable", i.e. the sum of the square values must be finite, $\sum_{t=1}^{\infty} \eta_t^2 < \infty$
- But at the same time retain enough weight that we can reach the minimum wherever it is:
    - if we sum at the infinite the learning rates we must be able to reach infinity, i.e. $\sum_{t=1}^{\infty} \eta_t = \infty$

One possible learning rate that satisfies all the above constraints is $\eta_t = \frac{1}{1+t}$.

Which is the gradient of the objective function with respect to data pair i ?

$$\nabla J_i(\theta) = \begin{cases} 0 & \text{when loss=0} \\ -y^i \mathbf{x}^{(i)} & \text{when loss >0} \end{cases} + \lambda\theta$$

The update rule for the i data pair is hence:

$$\theta = \theta - \eta_t * \left(\begin{cases} 0 & \text{when loss=0} \\ -y^i \mathbf{x}^{(i)} & \text{when loss >0} \end{cases} + \lambda\theta \right)$$

There are three differences between the update in the stochastic gradient descent method and the update in our earlier perception algorithm ($\theta = \theta + y^{(i)}\mathbf{x}^{(i)}$):

- First, is that we are actually using a decreasing learning rate due to the stochasticity.
- The second difference is that we are actually performing the update, even if we are correctly classifying the example, because the regularization term will always yield an update, regardless of what the loss is. And the role of that regularization term is to nudge the parameters a little bit backwards, so decrease the norm of the parameter vector at every stamp, which corresponds to trying to maximize the margin.
- To counterbalance that, we will get a non-zero derivative from the last terms, if the loss is non-zero. And that update looks like the perceptor update, but it is actually made even if we correctly classify the example. If the example is within the margin boundaries, you would get a non-zero loss.

An other point of view on SGD compared to a deterministic approach:

The original concern for using SGD is due to computational constraint: it is inefficient to compute the gradients over all the samples and update the model, think about you have more than a million training samples, and you can only make progress after computing all gradients. Thus, instead of compute the exact value of the gradient over the entire training data set, we randomly sample a fraction of them to estimate the true gradient. You see, there is a trade-off controlled by the batch size (number of samples used in each SGD updates), large batch size is more accurate but slower, and it need to be tuned in practice.

Nowadays, researchers start to understand SGD from a different perspective. The noise caused by SGD could make the trained more robust, (high level idea, noisy updates will make you converge to a flatter minima), which results in a better generalization performance. Thus, SGD could also be viewed as a way of regularization, in that sense, we may say it is better compared to gradient descent.

4.6. The Realizable Case - Quadratic program

Let's consider a simple case where (a) the problem is separable and (b) we do not allow any error, i.e. we want the loss part of the objective function to be zero (that's the meaning of "realisable case": we set as constraint that the loss function must return zero).

The problem becomes:

$\min_{(\theta,\theta_0)} \frac{1}{2}||\theta||^2$

Subject to:

$y^{(i)} * (\theta \cdot \mathbf{x}^{(i)} + \theta_0) \geq 1 \ \ \forall i = 1, \ldots, n$

We want hence extend the margins as much as possible while keeping zero losses, that is until the first data pair(s) is reach (we can't go over them otherwise we would start encoring a loss). Note that this is exactly equivalent to the minimisation of our original $J(\theta)$ function when λ is very small but not zero. There the zero loss output (in case of a separable problem) is the result of the minimisation of the loss function, here is it imposed as a constraint.

This is a problem quadratic in the objective and with linear constraints, and it can be solved with so-called quadratic solvers.

Relaxing the loss constraint and incorporate the loss in the objective would still leave the problem as a quadratic one.

Homework 2

Project 1: Automatic Review Analyzer

Recitation 1: Tuning the Regularization Hyperparameter by Cross Validation and a Demonstration

Outline

Supervised learning: Importance to have a model Objective: classification or regression

Cross validation: using the validation set to find the optimal parameters of our learning algorithm.

Supervised Learning

obj: derive a function f_{est} that approximate the real function f that link an input x to an output y.

We started by sampling from X and corresponding y --> training dataset We call the relation between this sampled pairs f_{data}

Our first obj is then to find a f_{est}, parametrised by some parameters (θ, θ_0), that approximate *this* f_{data}.

In order to find (θ, θ_0), we use an optimisation approach that minimise the function $J()$ made of a “loss” term $L(\theta, \theta_0; x, y)$ that describes the difference between our f_{est} and f_{data}.

But $f_{data} \neq f$. Because of that we add to our loss function a regularisation term $R(\theta)$ that constraints f_{est} not to be too similar to f_{data} that then is no longer able to generalise to f.

A **hyperparameter** α then balance these two terms in our function to minimise. And we remain with the task to choose this parameter, as it is not determined by the minimisation of the function as it is for (θ, θ_0). We’ll see that to find α we will use indeed the cross validation using training data only and how choosing alpha will affect the performance of the model.

We’ll go trough cross the method of cross validation which this is actually how we find α in practice using training data only, not the test data.

Support Vector Machine

What are the Loss and the regularisation functions for Support Vector Machines (SVM)

SVm obj: maximise the margins of the decision boundary

Distance from point i to the decision boundary:

$\gamma = \frac{y^i * (\theta \cdot x^i + \theta_0)}{||\theta||}$ (note that this is a distance with positive sign if the point side match the label side, negative otherwise)

The margin d is the minimal distance of any point with the decision boundary:

$d = \min_i \gamma(x^i, y^i, \theta, \theta_0)$

Large margin --> more ability of our model to generalise

Loss term

For SVM the loss term is the hinge loss L_h:

$$L_h = f\left(\frac{\gamma}{\gamma_{ref}}\right) = \begin{cases} 1 - \frac{\gamma}{\gamma_{ref}} & \text{when} \gamma < \gamma_{ref} \\ 0 & \text{otherwise} \end{cases}$$

As a function of γ, the hinge loss function for each point is above 1 for negative gammas, is 1 for $\gamma = 1$, continue to linearly decrease up to reaching 0 when $\gamma = \gamma_{ref}$ and then remains zero, where γ_{ref} is indeed the margin d as previously defined.

Regularisation term

The goal of the regularisation term is to make the model more generalisable.

For that, we want to maximise the margin, hence γ_{ref}. As we have a minimisation problem, that is equivalent to minimise $1/\gamma_{ref}$ or (as did in practice) $1/\gamma_{ref}^2$.

Objective function

$J = \frac{1}{n} \sum_{i=1}^n L_h(\frac{\gamma_i}{\gamma_{ref}}) + \alpha * \frac{1}{\gamma_{ref}^2}$

We now are left to define what γ_{ref} is in terms of (θ, θ_0).

Maximum margin

Note that we can scale θ by any constant (i.e. change it’s magnitude) without changing the decision boundary (obviously also θ_0 will be rescaled).

In particular, we can scale θ such that $y^m(\theta \cdot x^m + \theta_zero) = 1$, where m denotes the point at the minimum distance with the decision boundary.

In such case θ_{ref} simply becomes $\frac{1}{||\theta||}.

The objective function becomes the one we saw in the lecture:

$J = \frac{1}{n} \sum_{i=1}^n L_h(\$y^i(\theta \cdot x^i + \theta_0)) + \alpha * ||\theta||^2$

Note that the regularisation term is a function of θ alone.

Testing and Training Error as Regularization Increases

We will now see how different values of alpha affect the performance of the model.

Our objective function is

$J = L_(\theta, \theta_0, x_{data}, y_{data}) + \alpha R(\theta)$

At $\alpha = 0$, the obj model becomes just the loss model, and our functions should approximate very well f_{data}, the relation between training data and training outputs.

Accuracy is defined as $1/J$, as we are trying to minimise J.

As we increase α, J increases and the accuracy reduces.

What about the J function as computed from the testing dataset? As α increases it first reduce, as the model is gaining generability, but then it start back to increases as the model becomes too much general. The opposite for its accuracy.

There is indeed a value α^* for which indeed the J function under the testing set is minimised (accuracy is maximised).

And what't the α that we want to use. How do we find it with only training set data ? (we can't use testing data, as we don't yet have that data). The method is "cross validation" (next video).

Cross Validation

We want to find α^* . We just split our training data in a "new" training data, and a "validation set" to act as the "test set" and fine tune our α hyperparameter.

We start to discretize our hyperparameter in a number of K values to test (there are many ways to do that… ex-ante using a homogeneous grid or sampling approach, or updating the values to test as we have the results of the first α we tested…)

We then divide our training data in n partitions (where n depends on the data available), and, for each α_k, we choose one of that n partitions to be the "validation set": we minimise the J function using α_k and the remaining $(n-1)$ partitions, and we compute the accuracy $S_n(\alpha_k)$ for the validation set.

We then compute the average accuracy for α_k: $S(\alpha_k) = \frac{1}{n}\sum_{i=1}^{n} S_n(\alpha_k)$.

Finally we "choose" the α^* with the lowest $S(\alpha_k)$.

Tumor Diagnosis Demo

In this demo we found the best value of alpha in a SVM model that links breast cancer characteristics (such as cancer texture or size) with its benign/malign nature.

The example use sikit-learn, a library that allows to train your own machine learning algorithm (linear SVM in our case).

We use the function `linear_model.SGDClassifier(loss='hinge', penalty='l2', alpha=alpha[i])` that set up a SVM model using gradient descendent algorithm on hinge loss, using the L2 norm as regularisation term ("penalty") and the specific α we want to test.

We then use `ms.cross_val_score(model, X, y, cv=5)` to actually train the model on the training set divided in 5 different partitions. The function return already an array of the scores of the remaining validation partition. To compute the score for that particular alpha we now need just to average the score array obtained by that function.

[MITx 6.86x Notes Index]