[MITx 6.86x Notes Index]

Unit 02 - Nonlinear Classification, Linear regression, Collaborative Filtering

Lecture 5. Linear Regression

5.1. Unit 2 Overview

Building up from the previous unit, in this unit we will introduce:

- linear regression (output a number in R)
- non-linear classification methods
- recommender problems (sometime called collaborative filtering problems)

5.2. Objectives

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

- write the training error as least squares criterion for linear regression
- use stochastic gradient descent for fitting linear regression models
- solve closed-form linear regression solution
- identify regularization term and how it changes the solution, generalization

5.3. Introduction

Today we will see Linear Classification In the last unit we saw linear *classification*, where we was trying to land the mapping between the feature vectors of our data ($\boldsymbol{x}^{(t)} \in \mathbb{R}^{d}$) and the corresponding (binary) label ($\boldsymbol{y}^{(t)} \in \{-1, +1\}$).

This relatively simple set up can be already used to answer pretty complex questions, like making recommendations to buy or not some stocks, where the feature vector is given by the stock prices in the last d-days.

We can extend such problem to return, instead of just a binary output (price will increase - buy, vs price will drop - sell) a more informative output on the extent of the expected variation in price.

With regression we want to predict things that are continuous in nature.

The set up is very similar to before with $\boldsymbol{x}^{(t)} \in \mathbb{R}^{d}$. The only differences is that now we consider $\boldsymbol{y}^{(t)} \in \mathbb{R}$).

The goal of our model will be to map-- to learn how to map-- feature vectors into these continuous values.

We will consider for now only linear regression:

$$f(\mathbf{x}; \theta, \theta_0) = \sum_{i=1}^{d} \theta_i x_i + \theta_0 = \theta \cdot \mathbf{x} + \theta_0$$

For compactness of the equations we will consider $\theta_0 = 0$ (we can always think of θ_0 of being just a $d + 1$ dimension of θ).

Are we limiting ourself to just linear relations? No, we can still use linear classifier by doing an appropriate representation of the feature vector, by mapping into some kind of complex space and from that mapping then apply linear regression (note that at this point we will not yet talk about how we can construct this feature vector position. We will assume instead that somebody already gave us an appropriate feature vector).

3 questions to address:

1. Which would be an appropriate objective that can quantify the extent of our mistake?. How do we address the correctness of our output? In classification we had a binary error term, here it must be a range, our prediction could be "almost there" or very far.
2. How do we set up the learning algorithm. We will see two today, a numerical, gradient based ones, and a analytical, closed-form algorithm where we do not need to approximate.
3. How we do regularisation. How we perform a better generalization to be more robust when we don't have enough training data or when the data is noisy

5.4. Empirical Risk

Let's deal with the first question, the objective. We want to measure how much our prediction deviates from the know values of y, how far from y they are. We call our objective **empirical risk** (here denoted with R) and will be a sort of average loss of all our data points, depending from the parameter to estimate.

$$R_n(\theta) = \frac{1}{n}\sum_{t=1}^{n} \text{Loss}_h(y^{(t)} - \theta \cdot \mathbf{x}^{(t)}) = \frac{1}{n}\sum_{t=1}^{n} \frac{(y^{(t)} - \theta \cdot \mathbf{x}^{(t)})^2}{2}$$

Why squaring the deviation? Intuitively, since our training data may be noisy and the values that we record may be noisy, if it is a small deviation between our prediction and the true value, it's OK. However, if the deviation is large, we want really, truly penalize. And this is the behaviour we are getting from the squared function, that the bigger difference would actually result in much higher loss.

Note that the above equation use the squared error as loss function, but other loss functions could also be used, e.g. the hinge loss we already saw in unit 1:

$$\text{Hinge Loss}_h(z) = \begin{cases} 0 & \text{if } z \geq 1 \\ 1 - z & \text{oth.} \end{cases}$$

I will minimise this risk for the known data, but what I really want to do it so get it minimised for the unknown data I don't already see.

2 mistakes are possible:

1. **structural mistakes** Maybe the linear function is not sufficient for you to model your training data. Maybe the mapping between your training vectors and y's is actually highly nonlinear. Instead of just considering linear mappings, you should consider a much broader set of function. This is one class of mistakes.
2. **estimation mistakes** The mapping itself is indeed linear, but we don't have enough training data to estimate the parameters correctly.

There is a trade-off between these two kind of error: on one side, minimising the structural mistakes ask for a broader set of functions with more parameters, but this, at equal training set size, would increase the estimation mistakes. On the other side, minimising the estimation mistakes call for simpler set of functions, with less parameters, where however I become susceptible for structural mistakes.

In this lesson we remain commit to linear regression, and we want to minimise the empirical risk.

5.5. Gradient Based Approach

In this segment we will study the first of the two algorithms to implement the learning phase, the gradient based approach.

The advantage of the Empirical Risk function with the squared error as loss is that it is differentiable everywhere. Its gradient with respect to the parameter is:

$$\nabla_\theta\left(\frac{(y^{(t)}-\theta\cdot\mathbf{x}^{(t)})^2}{2}\right) = -(y^{(t)} - \theta\cdot\mathbf{x}^{(t)}) * \mathbf{x}^{(t)}$$

We will implement its stochastic variant: we start by randomly select one sample in the training set, look at its gradient, and update our parameter in the opposite direction (as we want to minimise the empirical risk).

The algorithm will then be as follow:

1. We initialise the thetas to zero
2. We randomly pick up a data pair
3. We compute the gradient and update $\boldsymbol{\theta}$ as $\boldsymbol{\theta = \theta - \eta * \nabla_\theta = \theta + \eta * (y^{(t)} - \theta\cdot\mathbf{x}^{(t)}) * \mathbf{x}^{(t)}}$ where $\boldsymbol{\eta}$ is the learning rate, influencing the size of the movement of the parameter at each iteration. We can have $\boldsymbol{\eta}$ constant or making it depends from the number of $\boldsymbol{k}$ iteration we already have done, e.g. $\boldsymbol{\eta = 1/(1+k)}$, so to minimise the steps are we get closer to our minimum.

Note that the parameter updates at each step, not only on some "mistake" like in classification (i.e. we treat all deviations as "mistake"), and that the amount depends on the deviation itself (i.e. not of a fixed amount like in classification). Going against the gradient assure that the algorithm self-correct itself, i.e. we obtain parameters that lead to predictions closer and closer to the actual true $\boldsymbol{y}$.

5.6. Closed Form Solution

The second learning algorithm we study is the closed form (analytical) solution. This is quite an exception in the machine learning field, as typically closed form solutions do not typically exist. But here the empirical risk happens to be a convex function that we can solve it exactly.

Let's compute the gradient with respect of $\boldsymbol{\theta}$, but this time of the whole empirical risk, not just the loss function:

$$R_n(\hat{\theta}) = \frac{1}{n}\sum_{t=1}^{n}\frac{(y^{(t)}-\hat{\theta}\cdot\mathbf{x}^{(t)})^2}{2}$$

$$\nabla_\theta R_n(\hat{\theta}) = \frac{1}{n}\sum_{t=1}^{n}\nabla_\theta\left(\frac{(y^{(t)}-\hat{\theta}\cdot\mathbf{x}^{(t)})^2}{2}\right) = -\frac{1}{n}\sum_{t=1}^{n}(y^{(t)} - \hat{\theta}\cdot\mathbf{x}^{(t)}) * \mathbf{x}^{(t)}$$

$$\nabla_\theta R_n(\hat{\theta}) = \frac{1}{n}\sum_{t=1}^{n} y^{(t)} * \mathbf{x}^{(t)} + \frac{1}{n}\sum_{t=1}^{n}\mathbf{x}^{(t)} * (\mathbf{x}^{(t)})^T * \hat{\theta}$$

$$\nabla_\theta R_n(\hat{\theta}) = -b + \mathbf{A}\hat{\theta} = 0$$

With $\boldsymbol{b = \frac{1}{n}\sum_{t=1}^{n} y^{(t)}x^{(t)}}$ is a scalar and $\mathbf{A} = \frac{1}{n}\sum_{t=1}^{n} x^{(t)}(x^{(t)})^T$ is an $\boldsymbol{(dxd)}$ matrix. If $\mathbf{A}$ is invertible we can finally write $\hat{\theta} = \mathbf{A}^{-1}b$.

We can invert \mathbf{A} only if the feature vectors $\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(n)}$ span over R^d, that is if $n >> d$.

Also, we should put attention that inverting $\mathbf{A}$ is an operation of order $O(d^3)$, so we should be carefully when d is very large, like in bag of words approaches used in sentiment analysis where d can be easily be in the tens of thousands magnitude.

5.7. Generalization and Regularization

We now focus the discussion in how do we assure that the algorithm we found, the parameters we estimated, will be good also for the unknown data, will be robust and not

too much negatively impacted by the noise that it is in our training data?

This question is more and more important as we have less data to train the algorithm.

The way to solve this problem is to use a mechanism called regularisation that try to push us away to fit the training data “perfectly” (where we “fit” also the errors, the noises embedded in our data) and try to instead generalise to possibly unknown data.

The idea is to introduce something that push the thetas to zero, so that it would be only worth for us to move our parameters if there is really a very strong pattern that justify the move.

5.8. Regularization

The implementation of the regularisation we see in this lesson is called **ridge regression** and it is the same we used in the classification problem:

The new objective function to minimise becomes:

$$J_{n,\lambda} = R_n(\theta) + \frac{\lambda}{2}||\theta||^2$$

The first term is our empirical risk, and it catches how well we are fitting the data. The second term, being the square norm of thetas, tries to push the thetas to remain zero, to not move unless there is a significant advantage in doing so. And $\boldsymbol{\lambda}$ is the parameter that determine the trade off, the relative contribution, between these two terms. Note that being the norm it doesn’t influence any specific dimension of theta. Its role is is actually to determine how much do I care to fit my training examples versus how much do I care to be staying close to zero. In other words, we don’t want any weak piece of evidence to pull our thetas very strongly. We want to keep them grounded in some area and only pulls them when we have enough evidence that it would really, in substantial way, impact the empirical loss.

In other terms, the effect of regularization is to restrict the parameters of a model to freely take on large values. This will make the model function smoother, leveling the ‘hills’ and filling the ‘vallyes’. It will also make the model more stable, as a small perturbation on $\boldsymbol{x}$ will not change $\boldsymbol{y}$ significantly with smaller $\|\boldsymbol{\theta}\|$.

What’s very nice about using the squared norm as regularisation term is that actually everything that we discussed before, both the gradient and closed form solution, can be very easily adjusted to this new loss function.

Gradient based approach with regularisation

With respect to a single point the gradient of $\boldsymbol{J}$ is :

$$\nabla_\theta\left(\frac{(y^{(t)}-\theta\cdot\mathbf{x}^{(t)})^2}{2} + \frac{\lambda}{2}||\theta||^2\right) = -(y^{(t)} - \theta\cdot\mathbf{x}^{(t)}) * \mathbf{x}^{(t)} + \lambda\theta$$

We can modify the gradient descent algorithm where the update rule becomes:

$$\theta = \theta - \eta * \nabla_\theta = \theta - \eta * (\lambda\theta - (y^{(t)} - \theta\cdot\mathbf{x}^{(t)}) * \mathbf{x}^{(t)}) = (1-\eta\lambda)\theta + \eta * (y^{(t)} - \theta\cdot\mathbf{x}^{(t)}) * \mathbf{x}^{(t)})$$

The difference with the empirical risk without the regularisation term is the $(\mathbf{1}-\boldsymbol{\eta\lambda})$ term that multiply $\boldsymbol{\theta}$ trying to put it down at each update.

The closed form approach with regularisation

In the homework ?

5. 9. Closing Comment

By using regularisation, by requiring much more evidence to push the parameters into the right direction we will increase the mistakes of our prediction within the training set, but

we will reduce the test error when the fitted thetas are used with respect to data that has not been used for the fitting step.

However if we continue to increase λ, if we continue to give weight to the regularisation term, then also the testing error will start to increase as well.

Our objective is to find the "sweet spot" of lambda where the test error is those that is minimised, and we can do that using the validation set to calibrate the value that lambda should have.

We saw regularization in the context of linear regression, but we will see regularization across many different machine learning tasks. This is just one way to implement it. We will see some other mechanism to implement regularisation, for example in neural networks.

Lecture 6. Nonlinear Classification

6.1. Objectives

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

- derive non-linear classifiers from feature maps
- move from coordinate parameterization to weighting examples
- compute kernel functions induced from feature maps
- use kernel perceptron, kernel linear regression
- understand the properties of kernel functions

6.2. Higher Order Feature Vectors

Outline

- Non-linear classification and regression
- Feature maps, their inner products
- Kernel functions induced from feature maps
- Kernel methods, kernel perceptron
- Other non-linear classifiers (e.g. Random Forest)

This lesson deals with non-linear classification, with the basic idea to expand the feature vector, map it to a higher dimensional space, and then feed this new vector to a linear classifier.

The computational disadvantage of using higher dimensional space can be avoided using so-called kernel functions. We will then see linear models applied to these kernel models, and in particular, for simplicity, the perceptron linear model (that when used with kernel function becomes the "kernel perceptron").

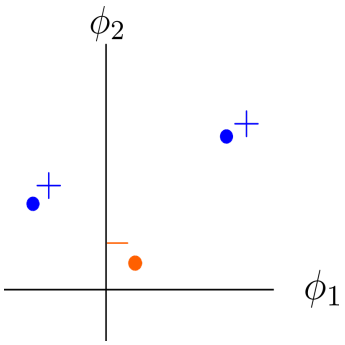
Let's see an example in 1D: we have the real line on which we have our points we want to classify. We can easily see that if we have the set of points {-3: positively labelled, 2: negatively labelled, 5:positively labelled} there is no linear classifier that can correctly classify this data set:

We can remedy the situation by introducing a feature transformation feeding a different type of example to the linear classifier.

We will always include in the new feature vector the original vector itself, so as to be able to retain the power that was available prior to feature transformation. But we will also add to it additional features. Note that, differently from statistics, in Machine Learning we know nothing (assume nothing) about the distribution of our data, so removing the original data to keep only the new added feature would risk to remove information that is not captured in the transformation.

In this case we can add for example x^2. So the mapping is $\mathbf{x} \in \mathbb{R}^2 = \phi(x \in \mathbb{R}) = \begin{bmatrix} x \\ x^2 \end{bmatrix}$. As result also the θ parameter of the classifier became bidimensional.

Our dataset becomes {(-3,9)[+], (2,4)[-], (5,25)[+]} that can be easily classified by a linear classifier in 2D (i.e. a line) $h(x;\theta,\theta_0) = \mathbf{sign}(\theta \cdot \phi(\theta) + \theta_0)$:



Note that the linear classifier in the new feature space we had found, back in the original space becomes a non-linear classifier: $h(x;\theta,\theta_0) = \mathbf{sign}(\theta_1 * x + \theta_2 * x^2 + \theta_0)$.

An other example would be having our original dataset in 2D as {(2,2)[+],(-2,2)[-],(-2,-2)[+],(2,-2)[-]} that is not separable in 2D, but it becomes separable in 3D when I use a feature transformation like $\mathbf{x} \in \mathbb{R}^3 = \phi(\mathbf{x} \in \mathbb{R}^2) = \begin{bmatrix} x_1 \\ x_2 \\ x_1 x_2 \end{bmatrix}$, for example by the plane given by $\left(\theta = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \theta_0 = 0\right)$.

6.3. Introduction to Non-linear Classification

We can get more and more powerful classifiers by adding linearly independent features, x^2, x^3,… This x, x^2,…, as functions are linearly independent, so the original coordinates always provide something above and beyond what were in the previous ones.

Note that when $\mathbf{x}$ is already multidimensional, even just $\phi(\mathbf{x}) = \mathbf{x}^2$ would result in dimensions exploding, e.g.

\mathbf{x} \in \mathbb{R^5} = \phi(\mathbf{x} \in \mathbb{R^2}) = \array{x_1\\x_2\\x_1^2\\x_2^2\\\sqrt{2}x_1x_2}### 6.3. Introduction to Non
We can get more and more powerful classifiers by adding linearly independent features, $x²
, x^3,… This x, x^2,…, as functions are linearly independent, so the original coordinates always provide something above and beyond what were in the previous ones.

Note that when $\mathbf{x}$ is already multidimensional, even just $\phi(\mathbf{x}) = \mathbf{x}^2$ would result in dimensions exploding, e.g. $\mathbf{x} \in \mathbb{R}^5 = \phi(\mathbf{x} \in \mathbb{R}^2) = \begin{bmatrix} x_1 \\ x_2 \\ x_1^2 \\ x_2^2 \\ \sqrt{2}x_1x_2 \end{bmatrix}$ (the meaning of the scalar associated to the cross term will be discussed later).

Once we have the new feature vector we can make non-linear classification or regression in the original data making a linear classification or regression in the new feature space:

- Classification: $h(x;\theta,\theta_0) = \mathbf{sign}(\theta \cdot \phi(\theta) + \theta_0)$
- Regression: $f(x;\theta,\theta_0) = \theta \cdot \phi(\theta) + \theta_0$

More feature we add (e.g. more polinomian grades we add), better we fit the data. The key question now is when is time to stop adding features ? We can use the validation test to test which is the polynomial form that, trained on the training set, respond better in the validation set.

At the extreme, you hold out each of the training example in turn in a procedure called **leave one out cross validation**. So you take a single training sample, you remove it from the training set, retrain the method, and then test how well you would predict that particular holdout example, and do that for each training example in turn. And then you average the results.

While very powerful, this explicit mapping into larger dimensions feature vectors is indeed… that the dimensions could become quickly very high as our original data is already multidimensional

Let's our original $\mathbf{x} \in \mathbb{R}^d$. Then a feature transformation:

- quadratic (order 2 polynomial): would involve $d+ \approx d^2$ dimensions (the original dimensions plus all the cross products)
- cubic (order 3 polynomial): would involve $d+ \approx d^2+ \approx d^3$ dimensions

The exact number of terms of a feature transformation of order p of a vector of d dimensions is $\sum_{i=1}^{p} \binom{d+i-1}{i}$ (the sum of multiset numbers).

So our feature vector becomes very high-dimensional very quickly if we even started from a moderately dimensional vector.

So we would want to have a more efficient way of doing that – operating with high dimensional feature vectors without explicitly having to construct them. And that is what kernel methods provide us. $ (the meaning of the scalar associated to the cross term will be discussed later).

Once we have the new feature vector we can make non-linear classification or regression in the original data making a linear classification or regression in the new feature space:

- Classification: $h(x; \theta, \theta_0) = \mathbf{sign}(\theta \cdot \phi(\theta) + \theta_0)$
- Regression: $f(x; \theta, \theta_0) = \theta \cdot \phi(\theta) + \theta_0$

More feature we add (e.g. more polinomian grades we add), better we fit the data. The key question now is when is time to stop adding features ? We can use the validation test to test which is the polynomial form that, trained on the training set, respond better in the validation set.

At the extreme, you hold out each of the training example in turn in a procedure called **leave one out cross validation**. So you take a single training sample, you remove it from the training set, retrain the method, and then test how well you would predict that particular holdout example, and do that for each training example in turn. And then you average the results.

While very powerful, this explicit mapping into larger dimensions feature vectors is indeed… that the dimensions could become quickly very high as our original data is already multidimensional

Let's our original $\mathbf{x} \in \mathbb{R}^d$. Then a feature transformation:

- quadratic (order 2 polynomial): would involve $d+ \approx d^2$ dimensions (the original dimensions plus all the cross products)
- cubic (order 3 polynomial): would involve $d+ \approx d^2+ \approx d^3$ dimensions

The exact number of terms of a feature transformation of order p of a vector of d dimensions is $\sum_{i=1}^{p} \binom{d+i-1}{i}$ (the sum of multiset numbers).

So our feature vector becomes very high-dimensional very quickly if we even started from a moderately dimensional vector.

So we would want to have a more efficient way of doing that – operating with high dimensional feature vectors without explicitly having to construct them. And that is what kernel methods provide us.

6.4. Motivation for Kernels: Computational Efficiency

The idea is that you can take inner products between high dimensional feature vectors and evaluate that inner product very cheaply. And then, we can turn our algorithms into operating only in terms of these inner products.

We define the kernel function of two feature vectors (two different data pairs) applied to a a given ϕ transformation as the dot product of the transformed feature vectors of the two data:

$$k(x, x'; \phi) \in \mathbb{R}^+ = \phi(x) \cdot \phi(x')$$

We can hence think of the kernel function as a kind of similarity measure, how similar the x example is to the x' one. Note also that being the dot product symmetric and positive, kernel functions are in turn symmetric and positive.

For example let's take x and x' to be two dimensional feature vectors and the feature transformation $\phi(x)$ defined as $\phi(x) = [x_1, x_2, x_1^2, \sqrt{(2)}x_1x_2, x_2^2]$ (so that $\phi(x')$ is $[x'_1, x'_2, {x'_1}^2, \sqrt{(2)}x'_1x'_2, {x'_2}^2]$)

This particular ϕ transformation allows to compute the kernel function very cheaply and having very few dimensions:

$$k(x, x'; \phi) = \phi(x) \cdot \phi(x')$$

$$= x_1x'_1 + x_2x'_2 + x_1^2{x'_1}^2 + 2x_1x'_1x_2x'_2 + x_2^2{x'_2}^2$$

$$= (x_1x'_1 + x_2x'_2) + (x_1x'_1 + x_2x'_2)^2$$

$$= x \cdot x' + (x \cdot x')^2$$

Note that even if the transformed feature vectors have 5 dimensions, the kernel function return a scalar. In general, for this kind of feature transformation function ϕ, the kernel function evaluates as $k(x, x'; \phi) = \phi(x) \cdot \phi(x') = (1 + x \cdot x')^p$, where p is the order of the polynomial transformation ϕ.

However, it is only for *some* ϕ for which the evaluation of the kernel function becomes so nice! As soon we can prove that a particular kernel function can be expressed as the dot product of two particular feature transformations (for those interested the *Mercer's theorem* stated in these notes) the kernel function is *valid* and we don't actually need to construct the transformed feature vector (the output of ϕ).

Now our task will be to turn a linear method that previously operated on $\phi(x)$, like $\text{sign}(\theta \cdot \phi(x) + \theta_0)$ to an inter-classifier that only depends on those inner products, that operates in terms of kernels.

And we'll do that in the context of kernel perception just for simplicity. But it applies to any linear method that we've already learned.

6.5. The Kernel Perceptron Algorithm

Let's show how we can use the kernel function in place of the feature vectors in the perceptron algorithm.

Recall that the perceptron algoritm was (excluding for simplicity θ_0):

```
θ = 0                # initialisation
for t in 1:T
  for i in 1:n
    if yⁱ θ·Φ(xⁱ) ≦ 0   # checking if sign is the same, i.e.
data is on the right side of its label
      θ = θ + yⁱΦ(xⁱ)   # update θ if mistake
```

Which is the final value of the parameter θ resulting from such updates ? We can write it as

$$\theta^* = \sum_{j=1}^{n} \alpha^{(j)} y^{(j)} \phi(x^{(j)})$$

where α is the vector of number of mistakes (and hence updates) underwent for each data pair (so $\alpha^{(j)}$ is the (scalar) number of errors occurred with the *j*-th data pair).

Note that we can interpret α^j in terms of the relative importance of the *j*-th training example to the final predictor. Because we are doing perceptron, the importance is just in terms of the number of mistakes that we make on that particular example.

When we want to make a prediction of a data pair $(x^{(i)}, y^{(i)})$ using the resulting parameter value θ^* (that is the "optimal" parameter the perceptron algorithm can give us), we take an inner product with that:

$$\text{prediction}^{(i)} = \theta^* \cdot \phi(x^{(i)})$$

We can rewrite the above equation as :

$$\theta^* \cdot \phi(x^{(i)}) = [\sum_{j=1}^{n} \alpha^{(j)} y^{(j)} \phi(x^{(j)})] \cdot \phi(x^{(i)})$$

$$= \sum_{j=1}^{n} [\alpha^{(j)} y^{(j)} \phi(x^{(j)}) \cdot \phi(x^{(i)})]$$

$$= \sum_{j=1}^{n} \alpha^{(j)} y^{(j)} k(x^{(j)}, x^{(i)})$$

But this means we can now express success or errors in terms of the α vector and a valid kernel function (typically something cheap to compute) !

An error on the data pair $(x^{(i)}, y^{(i)}) \leq 0$ can then be expressed as $y^{(i)} * \sum_{j=1}^{n} \alpha^{(j)} y^{(j)} k(x^{(j)}, x^{(i)})$. We can then base our perceptron algorithm on this check, where we start with initiating the error vector α to zero, and we run trought the data set checking for errors and, if found, updating the corresponding error term. I practice, our endogenous variable to minimise the errors is no longer directly theta, but became the α vector, that as said implicitly gives the contribution of each data pair to the θ parameter. The perceptron algorithm becomes hence the **kernel perceptron algorithm**:

```
α = 0                # initialisation of the vector
for t in 1:T
  for i in 1:n
    if yⁱ ∑ⱼ[αʲyʲk(xʲ,xⁱ)] ≦ 0   # checking if prediction is
right
      αⁱ += 1  # update αⁱ if mistake
```

When we have run the algorithm and found the optimal α^* we may immediately retrieve the optimal θ^* by the above equation, even if at this point we really do not need θ (or sometimes can't, i.e. when θ has infinite dimensions) to make predictions.

6.6. Kernel Composition Rules

Now instead of directly constructing feature vectors by adding coordinates and then taking it in the product and seeing how it collapses into a kernel, we can construct kernels directly from simpler kernels by made of the following **kernel composition rules**:

1. $K(x, x') = 1$ is a valid kernel whose feature representation is $\phi(x) = 1$;
2. Given a function $f : \mathbb{R}^d \to \mathbb{R}$ and a valid kernel function $K(x, x')$ whose feature representation is $\phi(x)$, then $\tilde{K}(x, x') = f(x)K(x, x')f(x')$ is also a valid kernel whose feature representation is $\tilde{\phi}(x) = f(x)\phi(x)$
3. Given $K_a(x, x')$ and $K_b(x, x')$ being two valid kernels whose feature representations are respectively $\phi_a(x)$ and $\phi_b(x)$, then

$K(x, x') = K_a(x, x') + K_b(x, x')$ is also a valid kernel whose feature representation is $\phi(x) = \begin{bmatrix} \phi_a(x) \\ \phi_b(x) \end{bmatrix}$

4. Given $K_a(x, x')$ and $K_b(x, x')$ being two valid kernels whose feature representations are respectively $\phi_a(x) \in \mathbb{R}^A$ and $\phi_b(x) \in \mathbb{R}^B$, then $K(x, x') = K_a(x, x') * K_b(x, x')$ is also a valid kernel whose feature representation is $\phi(x) = \begin{bmatrix} \phi_{a,1}(x) * \phi_{b,1}(x) \\ \phi_{a,1}(x) * \phi_{b,2}(x) \\ \phi_{a,1}(x) * \phi_{b,...}(x) \\ \phi_{a,1}(x) * \phi_{b,B}(x) \\ \phi_{a,2}(x) * \phi_{b,1}(x) \\ \phi_{a,...}(x) * \phi_{b,...}(x) \\ \phi_{a,A}(x) * \phi_{b,B}(x) \end{bmatrix}$ (see this lecture notes for a proof)

Armed with these rules we can build up pretty complex kernels starting from simpler ones.

For example let's start with the identity function as ϕ, i.e. $\phi_a(x) = x$. Such feature function results in a kernel $K(x, x'; \phi_a) = K_a(x, x') = (x \cdot x')$ (this is known as the **linear kernel**).

We can now add to it a squared term to form a new kernel, that by virtue of rules (3) and (4) above is still a valid kernel:

$$K(x, x') = K_a(x, x') + K_a(x, x') * K_a(x, x') = (x \cdot x') + (x \cdot x')^2$$

6.7. The Radial Basis Kernel

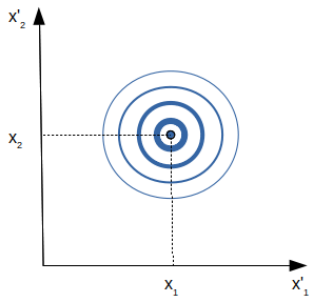
We can use kernel functions, and have them in term of simply, cheap-to-evaluate functions, even when the underlying feature representation would have infinite dimensions and would be hence impossible to explicitly construct.

One example is the so called **radial basis kernel**:

$$K(x, x') = e^{-\frac{1}{2}||x - x'||^2}$$

It can be proved that suck kernel is indeed a valid kernel and its corresponding feature representation $\phi(x) \in \mathbb{R}^\infty$, i.e. involves polynomial features up to an infinite order.

Does the radial basis kernel look like a Gaussian (without the normalisation term) ? Well, because indeed it is:



The above picture shows the contour lines of the radial basis kernel when we keep fixed x (in 2 dimensions) and we let x' to move away from it: the value of the kernel then reduces in a shape that in 3-d would resemble the classical bell shape of the Gaussian curve. We could even parametrise the radial basis kernel replacing the fixed $1/2$ term with a parameter γ that would determine the width of the bell-shaped curve (the larger the value of γ the narrower will be the bell, i.e. small values of γ yield wide bells).

Because the feature has infinite dimensions, the radial basis kernel has infinite expressive power and can correctly classify any training test.

The linear decision boundary in the infinite dimensional space is given by the set $\{x : \sum_{j=1} n\alpha^{(j)} y^{(j)} k(x^{(j)}, x) = 0\}$ and corresponds to a (possibly) non-linear boundary in the original feature vector space.

The more difficult task it is, the more iterations before this kernel perception (with the radial basis kernel) will find the separating solution, but it always will in a finite number of times. This is by contrast with the "normal" perceptron algorithm that when the set is not separable would continue to run at the infinite, changing its parameters unless it is stopped at a certain arbitrary point.

Other non-linear classifiers

We have seen as we can have nonlinear classifiers extending to higher dimensional space and evenutally using kernel methods to collapse the calculations and operate only *implicitly* in those high dimension spaces.

There are certainly other ways to get nonlinear classifiers.

Decision trees make classification operating sequentially on the various dimensions and making first a separation on the first dimension and then, in a subsequent step, on the second dimension and so on. And you can "learn" these trees incrementally.

There is a way to make these decision trees more robust, called **random forest classifiers**, that adds two type of randomness: the first one is in randomly choosing the dimension on which to operate the cut, the second in randomly selecting the single example on which operate from the data set (with replacement) and then just average the predictions obtained from these trees.

So the procedure of a random forest classifier is:

- boostrap the sample
- build a randomized (by dimension) decision tree
- average the predictions (ensemble)

Summary

- We can get non-linear classifiers (or regression) methods by simply mapping our data into new feature vectors that include non-linear components, and applying a linear method on these resulting vectors;
- These feature vectors can be high dimensional, however;
- We can turn linear methods into kernel methods by casting the computation in terms of inner products;
- A kernel function is advantageous when the inner products are faster to evaluate than using explicit feature vectors (e.g. when the vectors would be infinite dimensional!)
- We saw the radial basis kernel that is particularly powerful because it is both (a) cheap to evaluate and (b) has a corresponding infinite dimensional feature vector

Lecture 7. Recommender Systems

7.1. Objectives

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

- understand the problem definition and assumptions of recommender systems
- understand the impact of similarity measures in the K-Nearest Neighbor method
- understand the need to impose the low rank assumption in collaborative filtering
- iteratively find values of U and V (given $X = UV^T$) in collaborative filtering

7.2. Introduction

This lesson deals about recommender systems, when the algorithm try to guess preferences based on choices already made by the user (like film to watch or products to buy).

We'll see:

- the exact problem definition
- the historically used algorithm, the K-Nearest Neighbor algorithm (KNN);
- the algorithm in use today, the matrix factorization or collaborative filtering.

Problem definition

We keep as example across the lecture the recommendation of movies.

We start with a (n, m) matrix Y of preferences for user $a = 1, \ldots, n$ of movie $i = 1, \ldots, m$. While there are many ways to store preferences, we will use a real number.

The goal is to base the prediction on the prior choices of the users, considering that this Y matrix could be very sparse (e.g. out of 18000 films, each individual ranked very few of them!), i.e. we want to fill these "empty spaces" of the matrix.

Why not to use classification/regression based on feature vectors as learned in Lectures 1? For two reasons:

1. Deciding which feature to use or extracting them from data could be hard/infeasible (e.g. where can I get the info if a film has a happy or bad ending ?), or the feature vector could become very very large (things about in general "products" for Amazon, could be everything)
2. Often we have little data about a single users preferences, while to make a recommendation based on its own previous choices we would need lot of data.

The "trick" is then to "borrow" preferences from the other users and trying to measure how much a single user is closer to the other ones in our dataset.

7.3. K-Nearest Neighbor Method

The number K here means, how big should be your advisory pool on how many neighbors you want to look at. And this can be one of the hyperparameters of the algorithm.

We look at the k closest users that did score the element I am interested to, look at their score for it, and average their score.

$$\hat{Y}_{a,i} = \frac{\sum_{b \in KNN(a,i;K)} Y_{b,i}}{K}$$

where $KNN(a, i; K)$ is the set of K users close to user a that have a score for item i

Now, the question is of course how do I define this similarity? We can use any method to define similarity between vectors, like cosine similarity ($\cos\theta = \frac{x_a \cdot x_b}{\|x_a\| \|x_b\|}$) or Euclidean distance ($\|x_a - x_b\|$).

We can make the algorithm a bit more sophisticated by weighting the neighbour scores to the level of similarity rather than just take their unweighted average:

$$\hat{Y}_{a,i} = \frac{\sum_{b \in KNN(a,i;K)} sim(a,b) * Y_{b,i}}{\sum_{b \in KNN(a,i;K)} |sim(a,b)|}$$

where $sim(a, b)$ is some similarity measure between users a and b.

There has been many improvements that has been added to this kind of algorithm, like adjusting for the different "average" score that each user gives to the items (i.e. they compare the deviations from user's averages rather than the raw score itself).

Still they are very far from today's methods. The problem of KNN is that it doesn't enable us to detect the hidden structures that is there in the data, which is that users may be similar to some pool of other users in one dimension, but similar to some other set of users in a different dimension. For example, loving machine learning books, and having there

some “similarity” with other readers of machine learning books on some hidden characteristics (e.g. liking equation-rich books or more discursive ones), and plant books, where the similarity with other plant-reading users would be based on completely different hidden features (e.g. loving photos or having nice tabular descriptions of plants).

Conversely in collaborative filtering, the algorithm would be able to detect these hidden groupings among the users, both in terms of products and in terms of users. So we don’t have to explicitly engineer very sophisticated similarity measure, the algorithm would be able to pick up these very complex dependencies that for us, as humans, would be definitely not tractable to come up with.

7.4. Collaborative Filtering: the Naive Approach

Let’s start with a *naive* approach where we just try to apply the same method we used in regression to this problem, i.e. minimise a function $\boldsymbol{J}$ made of a distance between the observed score in the matrix and the estimated one and a regularisation term.

For now, we treat each individual score independently… and this will be the reason for which (we will see) this method will not work.

So, we have our (sparse) matrix $\boldsymbol{Y}$ and we want to find a dense matrix $\boldsymbol{X}$ that is able to replicate at best the observed points of $\boldsymbol{Y_{a,i}}$ when these are available, and fill the missing ones when $\boldsymbol{Y_{a,i} = missing}$.

Let’s first define as $\boldsymbol{D}$ the set of points for which a score in $\boldsymbol{Y}$ is given:
$\boldsymbol{D = \{(a,i) | Y_{a,i} \neq \text{missing}\}}$.

The J function then takes any possible X matrix and minimise the distance between the points in the $\boldsymbol{D}$ set less a regularisation parameter (we keep the individual scores to zero unless we have strong belief to move them from such state):

$$\boldsymbol{J(X;Y,\lambda) = \frac{\sum_{(a,i)\in D}(Y_{a,i}-X_{a,i})^2}{2} + \frac{\lambda}{2}\sum_{(a,i)} X_{a,i}^2}$$

To find the optimal $\boldsymbol{X^*_{a,i}}$ that minimise the FOC $\boldsymbol{(\partial X_{a,i}/\partial Y_{a,i}) = 0}$ we have to distinguish if $\boldsymbol{(a,i)}$ is in $\boldsymbol{D}$ or not:

- $\boldsymbol{(a,i) \in D}$: $\boldsymbol{X^*_{a,i} = \frac{Y_(a,i)}{1+\lambda}}$
- $\boldsymbol{(a,i) \notin D}$: $\boldsymbol{X^*_{a,i} = 0}$

Clearly this result doesn’t make sense: for data we already know we obtain a bad estimation (as worst as we increase lambda) and for unknown scores we are left with zeros.

7.5. Collaborative Filtering with Matrix Factorization

What we need to do is to actually relate scores together instead of considering them independently.

The idea is then to constrain the matrix X to have a lower rank, as rank captures how much independence is present between the entries of the matrix.

At one extreme, constraining the matrix to be rank 1, would means that we could factorise the matrix $\boldsymbol{X}$ as just the matrix product of two single vectors, one defining a sort of general sentiment about the items for each user ($\boldsymbol{u}$), and the other one ($\boldsymbol{v}$) representing the average sentiment for a given item, i.e. $\boldsymbol{X = uv^T}$.

But representing users and items with just a single number takes us back to the KNN problem of not being able to distinguish the possible multiple groups hidden in each user or in each item.

We could then decide to divide the users and/or the items in respectively $(n, 2)U$ and $(2, m)V^T$ matrices and constrain our X matrix to be a product of these two matrices (hence with rank 2 in this case): $X = UV^T$

The exact numbers K of vectors to use in the user/items factorisation matrices (i.e. the rank of X) is then a hyperparameter that can be selected using the validation set.

Still for simplicity, in this lesson we will see the simplest case of constraining the matrix to be factorisable by a pair of single vectors.

7.6. Alternating Minimization

Using rank 1, we can adapt the J function to take the two vectors u and v instead of the whole X matrix, and our objective becomes to found their elements that minimise such function:

$$J(\mathbf{u}, \mathbf{v}; Y, \lambda) = \frac{\sum_{a,i \in D} (Y_{a,i} - u_a * v_i)^2}{2} + \frac{\lambda}{2} \sum_a^n u_a^2 + \frac{\lambda}{2} \sum_i^m v_i^2$$

How do we minimise J ? We can take an iterative approach where we start by randomly sampling values for one of the vector and minimise for the other vector (by setting the derivatives with respect on its elements equal to zero), then fix this second vector and going minimise for the first one, etc., until the value of the function J doesn't move behind a certain threshold, in an alternating minimisation exercise that will guarantee us to find a local minima (but not a global one!).

Note also that when we minimise for the individual component of one of the two vectors, we obtain derivatives with respect to the individual vector elements that are independent, so the first order condition can be expressed each time in terms of a single variable.

Numerical example

Let's consider a value of λ equal to 1 and the following score dataset:

$$Y = \begin{bmatrix} 5 & ? & 7 \\ 1 & 2 & ? \end{bmatrix}$$

and let start out minimisation algorithm with $v = [2, 7, 8]$

L becomes :

$$J(\mathbf{u}; \mathbf{v}, Y, \lambda) = \frac{(5-2u_1)^2 + (7_8u - 1)^2 + (1-2u_2)^2 + (2-7u_2)^2}{2} + \frac{u_1^2 + u_2^2}{2} + \frac{2^2 + 7^2 + 8^2}{2}$$

From where, setting $\partial L / \partial u_1 = 0$ and $\partial L / \partial u_2 = 0$ we can retrieve the minimising values of (u_1, u_2) as 22/23 and 8/27. We can now compute $J(\mathbf{v}; \mathbf{u}, Y, \lambda)$ with these values of u to retrieve the minimising values of v and so on.

Homework 3

Project 2: Digit recognition (Part 1)

[MITx 6.86x Notes Index]