Machine Learning

Notes on A. Ng lectures

Jérémy Fix
Jeremy.Fix@Supelec.fr

Résumé *Notes on the Machine Learning lectures of A. Ng.*

Table des matières

1 Linear regression with one or multiple variables

2 Logistic Regression

We focus on classification problems for which the output y is binary (bi-class classification problem, later on, we will speak about multiclass classification) : $y \in \{0, 1\}$. We call class-0 the negative class and class-1 the positive class but these are just arbitrary names. For example, one may seek to learn to classify a tumor to be malignant or not depending on its size, or filter mails as being spam or not depending on some measure.

With linear regression, we can define a boundary condition saying : "if $h_\theta(x)$ is larger or equal than 0.5 then consider it as belonging to class-1, otherwise it belongs to class-0".

Note : il utilise le problème de classer des tumeurs, en ajoutant un outlier tends à déplacer la frontière de décision pour donner un mauvais classifieur. "I would not use linear regression for classification problems".

Il introduit la régression logistique pour contenir l'hypothèse $h_\theta(x)$ dans $[0, 1]$ alors qu'avec la régression linéaire peut donner des valeurs bien au dela de ce domaine alors que les étiquettes valent disont 0 et 1... Mais il y a aussi la sensibilité aux outliers(**à vérifier ! !**)

On introduit le modèle de régression logistique :

$$h_\theta(x) \quad = \quad g(\theta^T x) = \frac{1}{1 + e^{-\theta^T x}} \tag{1}$$

On peut voir $h_\theta(x)$ comme la probabilité estimée que $y = 1$ pour l'entrée x. C'est à dire qu'on considère $h_\theta(x)$ comme un modèle paramétré de $P(y = 1|x; \theta)$. Pour trancher si une entrée x appartient à l'une ou l'autre des classes, il faut introduire un seuil sur cette probabilité. Par exemple, si on prédit $y = 1$ quand $h_\theta(x) \geq 0.5$, cela correspond à prédire $y = 1$ quand $\theta^T x \geq 0$. Le plan correspondant à $\theta^T x = 0$ est appelée "decision boundary". Comme le feature vector peut contenir des combinaisons non-linéaires de nos observations, e.g. $x = [1, x_1, x_2, x_1^2, x_2^2]$, la boundary decision peut être non-linéaire quand tracée dans l'espace des observations, e.g. x_1, x_2.

On doit maintenant dériver des algorithmes pour trouver les paramètres θ. Partant d'une base d'apprentissage $(x^{(i)}, y^{(i)})$ avec $\forall i, y^{(i)} \in \{0, 1\}$, on cherche les paramètres θ. Pour la régression linéaire, on a quantifié la classification par une fonction quadratique $J(\theta) = \frac{1}{2m}\sum_i (h_\theta(x^{(i)}) - y^{(i)})^2 = \frac{1}{m}\sum_i \text{cost}(h_\theta(x^{(i)}), y^{(i)})$. La fonction de coût est donc une fonction des coûts de chaque exemple. Pour la régression linéaire, avec un coût quadratique, le coût $J(\theta)$ était convexe et le problème d'optimisation avait donc un seul minimum. Avec une hypothèse h_θ logistique, la fonction de coût n'est plus convexe si on utilise un coût quadratique ce qui a la conséquence fâcheuse d'introduire plusieurs minimums locaux à la fonction de coût (ce qui pose des problèmes pour la minimiser avec, par exemple, une descente de gradient). Pour la régression logisitique, on introduit le coût logarithmique :

$$\text{cost}(h_\theta(x), y) = \begin{cases} -\log(h_\theta(x)) & \text{si } y = 1 \\ -\log(1 - h_\theta(x)) & \text{si } y = 0 \end{cases} \tag{2}$$

On remarque qu'on peut réécrire cette fonction de coût [1] sous la forme $\text{cost}(h_\theta(x), y) = -(1 - y)\log(1 - h_\theta(x)) - y\log(h_\theta(x))$. Cette fonction de coût est convexe et donc sans minimums locaux. On en vient à introduire la fonction de coût $J(\theta)$:

$$J(\theta) \quad = \quad \frac{1}{m}\sum_i \text{cost}(h_\theta(x^{(i)}), y^{(i)}) \tag{3}$$

$$\quad = \quad -\frac{1}{m}\left(\sum_{i=1}^{m} y^{(i)}\log(h_\theta(x^{(i)})) + (1 - y^{(i)})\log(1 - h_\theta(x^{(i)}))\right) \tag{4}$$

Pour trouver les paramètres qui minimise cette fonction de coût, on peut considérer une descente de gradient :

$$\theta_j \leftarrow \theta_j - \alpha\frac{\partial}{\partial\theta_j}J(\theta) \tag{5}$$

En calculant le gradient du coût par rapport aux paramètres, on obtient la règle de mise à jour des paramètres :

$$\theta_j \leftarrow \theta_j - \alpha\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)} \tag{6}$$

1. On aurait pu imaginer une autre formulation de la fonction de coût $\text{cost}(h_\theta(x), y) = -\log((1 - y)(1 - h_\theta(x)) + yh_\theta(x))$ **mais** il y a des raisons théoriques pour préférer la version du texte, elle est convexe et est dérivée de principe de Maximum Likelihood estimation **? ? ? ? ? ?**

Comme pour la régression linéaire, les paramètres sont mis à jour simultanément. On pourra remarquer que cette règle de mise à jour est identique à la règle de mise à jour des paramètres pour la régression linéaire, à ceci près que l'hypothèse h_θ est différente (et la fonction de coût a été particulièrement bien choisie). Comme pour la régression linéaire, on peut ajuster l'échelle des features (feature scaling) pour améliorer la vitesse de convergence de la descente de gradient.

Tracer la fonction de coût quand $y = 1$ pour montrer que si $h_\theta(x)$ tend vers 1, le coût tends vers 0 ; Au contraire si $h_\theta(x)$ tends vers 0 (alors que $y = 1$ puisque que $h_\theta(x) = P(y = 1|x; \theta)$), le coût tends vers l'infini et l'algorithme est donc fortement pénalisé pour cette erreur. On peut mener un raisonnement similaire pour le cas $y = 0$ dans lequel on pénalise fortement lorsque $h_\theta(x) \to 1$.

La descente de gradient est une des manières de minimiser notre fonction de coût. Il en existe d'autres, parfois plus rapide en terme du nombre d'itérations nécessaires pour atteindre le minimum. Une liste d'algorithmes alternatifs est donnée ci-dessous :

- gradient conjugué
- BFGS
- L-BFGS

Ces algorithmes ont un certain nombre d'avantages par rapport à la descente de gradient :

- il n'y a pas besoin de spécifier manuellement un taux d'apprentissage α, celui-ci est ajusté automatiquement
- ils sont en général plus rapide à converger qu'une descente de gradient

Ils sont néanmoins plus compliqués à comprendre et mettre en œuvre. Cela dit, il existe un certain nombre d'implémentations clé-en-main de ces algorithmes.

With Octave, you would make use of the *fminunc* function with specific options setting which algorithm to use.

We now go back to multi-class classification. Therefore, we suppose that the output y can take more than two values. One way to solve this problem is to consider multi-class classification problems as multiple binary classification problems. We then train as many binary classifiers as we have classes. Each of these classifier learns to recognize one class versus all the others. Given all the individual classifiers h_i, we may define label an input x as belonging to the class which maximizes the outputs $h_i(x), \forall i \in [1..k]$.

3 Regularization

3.1 Motivation

Sometimes, when trying to fit an hypothesis to a data set, we may run into a problem called overfitting. Overfitting arises because we have just a partial observation of the true model that generates the data we are trying to fit. We therefore make an hypothesis on the shape of this model. Obviously, when we fit a model on a dataset, we want to interpolate the output for unobserved inputs (inputs that are not in the training set). The hypothesis we consider define how different we can interpolate this output. The simplest example of overfitting can be seen when fitting a polynomial on data set. If we increase the degree of the polynomial, we introduce more flexibility to the model (indeed, with the Lagrange polynomials, we know that we can fit perfectly any data set if we don't constrain the degree of the polynomial). This flexibility can lead to more variability in the model which may impairs the ability of the model to generalize, i.e. to give the right answer on new data unavailable in the training set.

There are two options to adress overfitting. The first one is to reduce the number of features whether by manually selecting the features to keep or to use model selection algorithm. However, throwing away some features may throw away some information contained in the dataset. The second option is called regularization. Regularization is a technique which allows to balance the tendency to overfit the training set by a term which constrains the *complexity* of the model.

Des données à fiter avec un polynôme

Suppose we consider linear regression with some polynomials of the inputs :$h_\theta(x) = \theta_0+\theta_1 x+\theta_2 x^2+\theta_3 x^3+\theta_4 x^4$. If we penalize the two parameters θ_3 and θ_4, the polynomial h_θ will be simpler. To penalize these two parameters we can modify the quadratic cost function by adding terms that increases the cost depending on the amplitude of these parameters, e.g. $J(\theta) = \frac{1}{m}(h_\theta(x^{(i)}) - y^{(i)})^2 + 1000\theta_3^2 + 1000\theta_4^2$. Minimizing this modified cost function will constrain the parameters θ_3 and θ_4 to be small. More generally, as we may not know which parameters influence overfitting, we introduce the following cost function for linear regression :

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 + \lambda\sum_{i=1}^{n}\theta_j^2 \tag{7}$$

By convention, θ_0 is not regularized. The parameter λ balances the trade off between fitting the dataset and keeping the parameters θ_i small. If λ is set too large, minimization of this cost function may unfortunately lead to underfitting, i.e. we even do not fit the training set. Fortunately, one may define algorithms automatically adjusting the penalization coefficient λ.

Question : est ce qu'introduire le terme de régularization ne rends pas la fonction de coût non convexe ? NON d'après une question du quizz, la fonction de coût reste convexe.

Question : pourquoi est ce qu'on s'en fout de ne pas régularizer le terme θ_0.

Question : pourquoi ne pas pénaliser plus les termes de grand degré ?

3.2 Regularized linear regression

We remind the cost function for linear regression with L2 regularization :

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 + \lambda\sum_{i=1}^{n}\theta_j^2 \tag{8}$$

Considering this regularized cost function, one must modify the gradient descent algorithm. The update of the parameters now reads :

TODO, calculer le gradient, d'après Ng : $\theta_0 \leftarrow \theta_0 - \alpha/m\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_0^{(i)}; \forall j \neq 0, \theta_j \leftarrow \theta_j - \alpha(1/m\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)} + \lambda/m\theta_j) = (1 - \alpha\lambda/m)\theta_j - \alpha/m....$

By isolating the terms depending on θ_j, one gets an update rule of the form $\theta_j \leftarrow \beta\theta_j - \alpha/m..$, we see that we shrink the parameter and move in a direction minimizing the cost function (when considered without the regularization term).

We also saw that we can use normal equations for solving linear regression. Without regularization, the optimal parameters using normal equation was :$\theta = (X^TX)^{-1}X^Ty$. We regularization inside, the normal equation now reads $\theta = (X^TX + \lambda R)^{-1}X^Ty$ where R is a diagnoal matrix filled in with 1, except for the first element which is set to 0. For e.g., with $n = 2$, $R = [000; 010; 001]$.

If you have less examples than features, the matrix X^TX is non invertible. If you use regularization, it is possible to prove that the matrix $X^TX + \lambda R$ cannot be singular and that you can always invert it.**AH OUAIS ? ? ?**.

3.3 Regularized logistic regression

For logistic regression, the cost function reads :

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}(y^{(i)}\log(h_\theta(x^{(i)})) + (1 - y^{(i)})\log(1 - h_\theta(x^{(i)}))) \tag{9}$$

We can also add L2 regularization to this cost function to get :

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}(y^{(i)}\log(h_\theta(x^{(i)})) + (1 - y^{(i)})\log(1 - h_\theta(x^{(i)}))) + \lambda/(2m)\sum_j\theta_j^2 \tag{10}$$

The gradient descent update is then modified into $\theta_0 \leftarrow \theta_0 - \alpha/m\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_0^{(i)}; \forall j \neq 0, \theta_j \leftarrow \theta_j - \alpha(1/m\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)} + \lambda/m\theta_j) = (1 - \alpha\lambda/m)\theta_j - \alpha/m....$

The update is very similar to regularized linear regression but remind that the hypothesis h_θ is different for logistic regression.

4 Idées de démos

Pour la régularisation, une appli dans laquelle on fit un polynôme sur des données et on règle dynamiquement la contribution λ de la régularisation pour voir la forme du modèle appris.

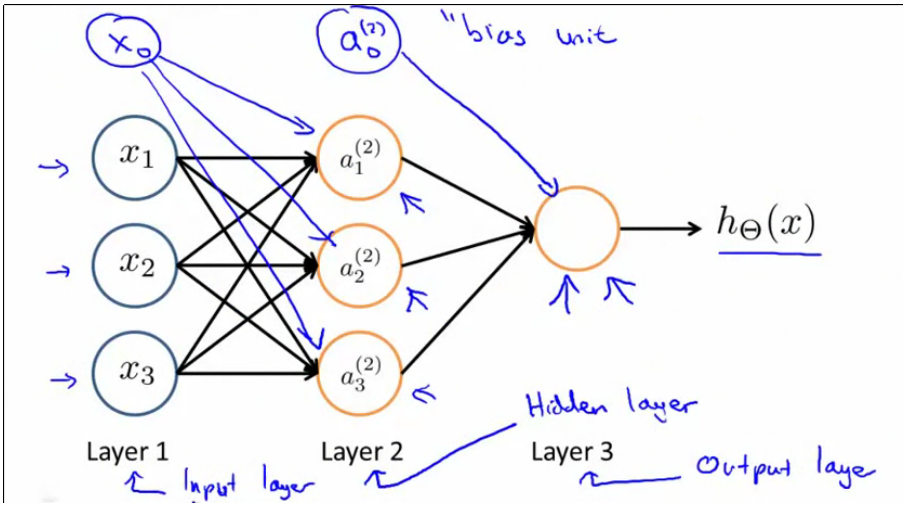
5 Neural Networks : Representation

A vérifier : overfitting if we increase the number of features with a constant number of data ?

Neural networks are the state of the art technique for several machine learning problems. To motivate neural networks, let us consider a non-linear classification problem. As we saw, we can use logistic regression with polynomial features (say $x_1x_2, x_2^3x_3, ..$). However, given the size of our inputs, the size of the feature vector may be a polynomial of it which will certainly be quite big and increase the tendency to overfitting (if we consider too many features relative to the number of the data we have, we can easily encounter overfitting). To give on example, if one considers object recognition from images of size 50×50, one obtains images with $n = 2500$ pixels. If we just consider only quadratic terms x_ix_j, one gets around 3 million features, the number of features is indeed $n^2/2$. One may limit the number of features, but this selection is, there, somehow arbitrary. Neural networks are a much better way to define non-linear hypotheses than logistic regression with polynomial features.

Neural networks, while originally derived from the desire to mimick the brain, is used from an abstract viewpoint in Machine Learning where we consider simple massively interconnected entities. Each entity receives signal from others, process it locally and transmit a signal to other entities. The simple model we consider is a logistic unit, which receives inputs denotes x_i, computes a weighted sum of it and apply a sigmoid transfer function on it. This is nothing more than what we saw in the previous section $y = h_\theta(x) = \frac{1}{1+e^{-\theta^T x}}$. We also add a bias unit to the inputs $x_0 = 1$ in addition to the signals received from the other neurons $x_i, i \geq 1$. In the neural network literature, θ are called the weights. We can build layers of such units and interconnect them. A multilayer neural network is shown on fig.1.

Figure 1 – Representation of a feedforward neural network. Capture from the A. Ng lecture.



We will denote $a_i^{(j)}$ the activation of the i-th unit of j-th layer. We denote $\theta^{(j)}$ the matrix of weights from layer j to layer $j+1$ [2]. The activation is computed according to :

$$\begin{cases} \forall i \geq 1, \forall j > 1, a_i^{(j)} & = g(z_i^{(j)}) \\ \forall i \geq 1, \forall j > 1, z_i^{(j)} & = \sum_k \theta_k^{(j-1)} a_k^{(j-1)} \\ \forall i \geq 1, a_i^{(1)} & = x_i \\ \forall j, a_0^{(i)} & = 1 \end{cases} \tag{11}$$

These equations tell nothing more than that the first neuron is the bias for the next layer, the activations $a_i^{(1)}$ of the first layer equal our inputs and each neuron in a subsequent layer computes its activation $a_i^{(j)}$ as a sigmoid of a weighted sum of the activations of the previous layer $a_k^{(j-1)}$. A vectorized notation for computing the activations of layer $j+1$ is simply $a^{(j+1)} = g(\theta^{(j)}.a^{(j)})$ where $\theta^{(j)}$ is a matrix of size $|L_{j+1}| \times |L_j|$ with $|L_j|$ the number of neurons in layer j (the bias being included). For regression, the last layer contains a single neuron of which the activation, if unfolded, is a function of the input x. Computing the activations in a layer-wise manner from the very first layer to the output is called the forward propagation. If we take a closer look to the 3-layers neural network depicted above, the hidden to output layer is nothing more than a logistic regression with Layer 2 representing the features. Now, if we include the very first layer with the weights projecting the inputs to the hidden layer, we can see it as building the

2. We here consider only feedforward neural networks were the connections always go from layer j to layer $j+1$

features feeding the logistic regression. And the nice thing there is that the features are learned through the weights from Layer 1 to Layer 2. It may not be completely clear that the features we compute that way (the activations of layer 1) can be arbitrary polynomials but indeed one may demonstrate that if we don't restrict ourselves to a single hidden layer, one can approximate any function (feedforward neural networks are universal approximators) so that feedforward neural networks have at least the expressiveness of logistic regression.

Now let us consider simple binary operations like and, or, not, xor. We can easily write a simple neural network with a single input layer and a single output for the and and *or* functions :

- *and* function : $\theta = (-30, 20, 20)$
- *or* function : $\theta = (-10, 20, 20)$
- *not* function : $\theta = (10, -20)$

These are both linearly separable problems. The Xor function is not linearly separable and one needs to consider at least one hidden layer to compute it. To put it simply, we just need to write $x_1 \text{xor} x_2 = \text{or}(\text{and}(\text{not}(x1), x_2), \text{and}(x1, \text{not}(x_2)))$. The functions not(x1) or x2, as well as x1 and not(x2) can be compute with a single layer. These are the input to hidden layer connections. Finally, we just connect the hidden to the output layer for computing the or function.

Before, we were speaking about a simple regression example. An example of multi-class classification is given by the LeNet 5 feedforward neural network developed by Y. LeCun which he trained for recognizing handwritten digits. The trained network is quite unsensitive to scale, orientation, and some forms of noise. For multi-class classification, we consider an output layer of the size of the number of classes. The output $y^{(i)}$ for input $x^{(i)}$ is set to $y_j^{(i)} = \delta(j, l^{(i)})$ where δ is the kronecker symbol and $l^{(i)}$ is the class of the input $x^{(i)}$.

Ref pour l'approximateur universel ?

6 10

In this section, few advices are given so as to know on which parameters one must play in order to improve the performances of the machine learning algorithm.

For example, when we test an hypothesis and find that it performs very badly on new unknown data, there are indeed various options one may be tempted to try, among which :

- get more training examples, it will probably better constrain the optimization of the hypothesis
- try smaller sets of features to avoid overfitting
- try getting additional features
- try adding polynomial features
- try decreasing or increasing the regularization parameter λ

Instead of trying one of these options at random, we can diagnose what is going wrong with our algorithm. One possibility to diagnose how an hypothesis performs is, in low dimension (up to 3), to plot the decision boundary for classification or the hypothesis directly for regression. However, in higher dimensions, this is not possible.

6.1 Training set/Test set and error measures

We would rather split the dataset in (at least) two datasets : a **training set** (70% of the data) and a **test set** (30 % of the data). [3]. Then we optimize the parameters on the training set and measure the generalization error on the test set. For classification problems, one may also measure the misclassification error :

$$\text{err}(h_\theta(x), y) = \begin{cases} 1 & \text{if } h_\theta(x) \geq 0.5, y = 0 \\ & \text{or } h_\theta(x) < 0.5, y = 1 \\ 0 & \text{otherwise} \end{cases} \tag{12}$$

The test classification error is then just the mean over the test set of the misclassification error above which represents the proportion of the test set that is misclassified.

6.2 Model selection and train/validation/test sets

As we know already, the training error alone is not an appropriate measure for evaluating the performances of an hypothesis on unseen data since we might be overfitting. Instead, we should use the test set error. Suppose we wish to select among several polynomial hypothesis one that performs better. To do such a model selection, we will compare the test set errors and select the model with the lowest test error ; However, to report the generalization ability of the model that we selected, we should use additional data as using the test set might overestimate the generalization

3. be sure that these sets are made of random samples of the datasets to avoid training on a specific region of the space while testing on another non-overlapping region

error of the model (the degree of polynomial was indeed selected so that the test set error is decreased). Therefore, we should split the dataset into three pieces : **the training set** (60%), **the cross-validation (cv) or validation set** (20%) and **the test set** (20%) :

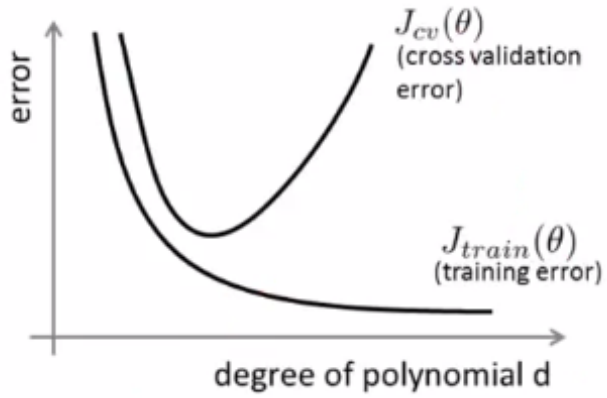
- the training set is used to optimize the parameters of the models
- the test set is used for selecting among the various that we have trained
- the validation set is used to estimate the generalization error of the selected model

6.3 Bias/variance issues

High bias issue is when the hypothesis is not sufficiently complex (underfit) while high variance is when the hypothesis is too complex (overfit). To better appreciate these issues, let us consider the case of polynomial hypothesis and plot both the training and validation errors as function of the degree of the polynomial (fig. 2).

As we increase the degree of the polynomial, we expect the training error to decrease as the model is more and more complex and we can overfit the training set more easily with higher degrees. The cross validation error will usually start as high as the training error, decrease and, then , at some point, increase as we overfit the training set and impair the generalization ability of the hypothesis.

Figure 2 – Typical curves of training and validation errors function of the degree of a polynomial hypothesis



Detecting whether we encounter a high variance or high bias problem is critical as we will not behave the same way in the two conditions. A high bias means a model that is not enough complex such that it cannot even fit the training set properly. A high variance means a model that is too complex so that it certainly fits perfectly the training set but performs poorly on the test set.

		Validation error Low	 High
Training error	Low	Good	High variance
	High	-	High bias

6.4 Regularization and bias variance

Varying the regularization parameter will affect the bias/variance. A very large parameter will penalize high values, leading to small parameters, leading to a high bias (fig. 3). If the regularization parameter is too small, say absent, a model that is too complex will not be penalized and we will probably run into high variance issues. Therefore, the regularization parameter must be correctly adjusted. This parameter must be choosen empirically, by testing different values for this parameter and checking both the training and cross validation errors. We then peak out the parameter giving a the lowest cross validation error and then evaluate the true performances on the test set.

We can also plot the training and cross validation errors as functions of the number of training examples (**the learning curves**). The training error usually increases as you use a larger training set as it is pretty easy to fit perfectly few examples while more complicated to fit more. The cross validation error will usually decrease as you more easily overfit small training set and you generalize better if you use more training examples that better constrain the hypothesis.

- if an algorithm is suffering from **high bias** (e.g. the hypothesis is not complex enough), getting **more training data will not help much**
- if an algorithm is suffering from **high variance**, getting **more training data will help**

Given the previous observations, below is a list of options to consider depending on whether we face a high bias or high variance issue :

FIGURE 3 – Skecthes of how the training and cross validation errors usually evolve as a function of the regularization parameter.

- Getting more training examples : will help fixing high variance problems
- try smaller sets of features : will help fixing high variance problems
- try getting additional features : will usually help fixing high bias
- try adding polynomial features : might help fixing high bias
- try decreasing λ for fixing high gbias, increasing λ for fixing high variance

7 MACHINE LEARNING SYSTEM DESIGN

7.1 PRIORITIZING WHAT TO WORK ON : SPAM CLASSIFICATION EXAMPLE

This section deals with various aspects of designing a machine learning system using the example of designing a spam classifier.

The first step is to decide what to use for the features. For example, we might use 100 words that we consider might be indicative of the spam/non-spam category or to actually pike the words appearing the most frequently in the emails. The feature vector will then be a binary vector indicating whether a word appears or not in the email to classify. Some specific sophisticated features might also be added (e.g. from the header of the email, the body, checking for mispelled words, ...).

7.2 ERROR ANALYSIS

To design a machine learning system, one should

1. start with a simple, quickly to implement, (dirty) algorithm and test it on a cross validation set
2. plot learning curves to decide whether we should focus on more data, more features, ...
3. error analysis : check which examples are misclassified from the cross validation set and why they are misclassified (what is particular about them)

In particular, error analysis is really critical as it will really help in adjusting the features to use. For spam classification, a stemming software (e.g. Porter Stemmer) can be usefull for deciding whether or not some variations of the same word should be considered the same word. In this situation, there is no other solutions than trying different solutions and comparing their performances with respect to the cross-validation and test errors.

7.3 ERROR METRICS FOR SKEWED CLASSES

An important caveat can lead us to misinterpret the test error. Imagine we train a classifier for deciding whether or not a patient has a cancer. If we get a test error of 1% we might say the learning algorithm is doing pretty well. However, if the data are such that only 0.5 % of the patients have a cancer, the 1% is actually pretty high. We speak about **skewed classes** when the positive and negative examples of the data are really unbalanced as it was the case for the cancer data. Indeed, in such a situation, it is pretty easy to design an efficient algorithm with respect to the

test error : just attribute the label of the largest class ! In such a situation, the usual **classification error on the test set is not a sound error metric**. We rather introduce other measures : precision, recall, accuracy, QUATIREME ELEMENT ! ! ! ! Suppose we work on a binary classification task, we then introduce true/false positives and true/false negatives according to the following table :

		Actual class	
		1	0
Predicted class	1	True positive	False positive
	0	false negative	true negative

The four measures are then defined as :

$$\text{precision} = \frac{\text{true positives}}{\#\text{predicted positive}} = \frac{\text{true positives}}{\text{true positives} + \text{false positives}} \tag{13}$$

$$\text{recall} = \frac{\text{true positives}}{\#\text{actual positives}} = \frac{\text{true positives}}{\text{true positives} + \text{false negatives}} \tag{14}$$

The **precision** indicates the fraction of the patients that are positives among the patients classified as positives [4]. The **recall** indicates the fraction of positive examples that were correctly classified as positive. If we consider our example where the algorithm classifies all the patients as not having a cancer, both the recall and precision will be zero.

7.4 Trading off precision and recall and F_1 score

If we continue with our example of cancer classification and check the influence of the decision threshold (say with a hypothesis $0 \leq h_\theta(x) \leq 1$, we predict 1 if $h_\theta(x) \geq T$ and 0 if $h_\theta(x) < T$). If $T = 0.5$ we have a certain precision and recall. If **we increase** T, say $T = 0.9$, we will be much more selective and much more confident about the examples that we classify as positives. Therefore, the fraction of examples that are really positives among the ones that we classify as positives will be higher, then a **higher precision**. However, as much less examples are classified as positives, the **recall will be lower**. If on the opposite, we decrease the threshold, say $T = 0.3$, we will be more conservative as more than necessary patients will be classified as positive, in which case we will have a **lower precision** and **a higher recall**. Usually, there is a trade off between precision and recall. Now, if we compare several algorithms (or the same with different meta-parameters), they will have different precision P and recall R values. To compare them, we introduce the F_1 score. As we saw previously, some simple algorithms (like always assigning the same class) is not a usefull classifier. Whether always saying "class 0" or "class 1", these algorithms will have whether high precision and low recall or low precision and high recall. If we compute the mean of precision and recall, it will be the same in the two situations as well as the same as an algorithm which has equal precision and recall. Therefore, the mean is not discriminative enough for deciding which algorithm performs best. The $F_1 \in [0, 1]$ score is rather defined as :

$$F = 2\frac{PR}{P + R} \tag{15}$$

7.5 Data for machine learning

Banko and Brill 2001 worked on the problem of predicting a word in a set of confusable words. They compared the performance of four algorithms : Perceptron, Winnow, Memory-based, Naïve bayes. They measured the performance of these algorithms varying the training set size. It appears that all the algorithms improved with the training set size but as the dataset get larger and larger, three of them were better than the last (Memory based). Collecting a larger dataset is particularly useful if the features bring enough information to predict the output (e.g. "Can a human expert confidently predict the output ?"). Using a very large training set allow to consider *complex* models (which might lead to high variance with a small dataset) without falling into an overfit issue.

8 Support vector machines

We already saw previously various supervised learning algorithms (namely logistic regression, neural networks). We now turn on to another learning algorithm : the Support Vector Machine (SVM).

8.1 Optimization objective

The objective function for SVM reads :

$$J(\theta) = C\sum_{i=1}^{m} y^{(i)}\text{cost}_1(\theta^T x^{(i)}) + (1 - y^{(i)})\text{cost}_0(\theta^T x^{(i)}) + \frac{1}{2}\sum_{j=1}^{n}\theta_j^2 \tag{16}$$

It is just the cost function of logistic regression where :

1. we dropped the normalization term $1/m$ (which does not change the position of the optimum) ,

4. by convention, when faced a classification problem with skewed classes, the rare cases are the positive ones

2. replaced $-\log(h_\theta(x^i)), -\log(1-h_\theta(x^{(i)}))$ by $\text{cost}_1(\theta^T x^{(i)}), \text{cost}_0(\theta^T x^{(i)})$,
3. and moved the regularization term on the first member in the cost function (again, this does not change the optimization problem drastically, it is just a scaling factor).

PLOTTER cost0 et cost1 ! ! ! ! ! qui peuvent être vues comme des approximations linéaires par morceaux de log(1/(1+exp(-theta.T x)))

The hypothesis of SVM is simply :

$$h_\theta(x) = \begin{cases} 1 & \text{if} \theta^T x > 0 \\ 0 & \text{otherwise} \end{cases} \tag{17}$$

8.2 Large margin intuition

Given the cost function introduced before, we want $\theta^T x \geq 1$ (not just ≥ 0) to predict 1 and we want $\theta^T x \leq -1$ (not just < 0) to predict 0. If all the data are correctly classified, the first term of the cost function vanishes and we only keep the regularization term. Therefore, the optimization might be rewritten as :

$$\text{minimize} \qquad \frac{1}{2}\sum_{j=1}^{n} \theta_j^2 \tag{18}$$

$$\text{undertheconstrainthat} \quad \begin{cases} \theta^T x \geq 1 & \text{if} y = 1 \\ \theta^T x \leq -1 & \text{if} y = 0 \end{cases} \tag{19}$$

The constrain forces that the data are not too close to the decision boundary and it introduces a margin so that the data are not allowed to be in this domain.

8.3 Mathematics behind large margin classification

une intuition sur l'orientation de la frontière de décision en 2D. Lien entre minimiser la norme de theta et maximiser la distance entre la frontière de décision et les exemples. En prenant un exemple en 2D avec 2 classes sans outliers.

8.4 Kernels

When classifying or regressing some data, we have seen it might be interesting to devise new features (e.g. polynomial features). Here, we introduce yet another way to define new features based on a similarity measure between the data and some particular vectors in the input space that we call *landmarks*. This similarity measure, that we call a kernel, can be computed as gaussian, in which case the features f_i are defined from landmarks $l^{(i)}$:

$$\forall i, f_i = \exp(-\frac{||x - l^{(i)}||^2}{2\sigma^2}) \tag{20}$$

We call it a similarity measure in the sense that if x is close to (say equal) the landmark $l^{(i)}$, the associated feature f_i will be approximately equal to 1 and will decrease to 0 (in the limit) when the input is moved far away from the landmark. With such features, even using a linear classifier we can learn non-linear decision boundaries as we combine gaussians centered on different landmarks (like in radial basis functions networks).

The next question is how to choose these landmarks we introduced before. One natural definition of such landmarks is to use the training examples as landmarks. So given the training examples $(x^{(i)}, y^{(i)}), \forall i \in [1..m]$, we introduce landmarks $l^{(i)} = x^{(i)}, \forall i \in [1..m]$. These landmarks allow us to project a sample x on a $m+1$ dimensional feature space $f_i = \text{similarity}(x, l^{(i)}), \forall i \in [1..m], f_0 = 1$. Using this feature vector, the cost function then reads [5] :

$$J(\theta) = C\sum_{i=1}^{m} y^{(i)} \text{cost}_1(\theta^T f^{(i)}) + (1 - y^{(i)}) \text{cost}_0(\theta^T f^{(i)}) + \frac{1}{2}\sum_{j=1}^{m} \theta_j^2 \tag{21}$$

The parameter C (C might be understood as the inverse of the regularization parameter of a logistic regression) will modify the balance between perfectly fitting the training examples or allowing more generalization. Indeed, a large C will decrease the bias but will increase the variance ; the hypothesis will tend to overfit the training set. On the contrary, a small C will increase the bias and potentially lower the variance. By potentially, we mean that we encounter the similar situation as logistic regression where a too high λ might indeed lead to a parameter vector that is equal to 0 and therefore both the bias and variance will be high. Similarly, a small σ^2 will tend to overfit [6] (low bias, high variance), while a large $sigma^2$ will favor generalization with a higher bias but a lower variance.

5. we still don't regularize θ_0. One might read $\theta'^T \theta'$ where
6. we might consider the extreme situation where the variance of the gaussians equals 0 and we have very peaky, dirac like, gaussian functions

8.5 Using a SVM, SVM vs logitstic regression

In order to make use of a SVM classifier, probably the best option is to use of the shelf libraries such as **liblinear** or **libsvm**. We only need to provide the parameters such as C, but also σ^2 in case of gaussian kernels. Several other kernels might be used such polynomial kernels, string kernels, chi-square kernels, intersection kernel, etc... All the kernels must satisfy the Mercer theorem. For multi-class classification, make us of the algorithms provided in the library or do it use one versus all. Between the linear kernel and the gaussian kernel, it is better to use a linear kernel when the number of features is large enough and a gaussian kernel in case the number of features is not sufficiently large.

It is advised to use a SVM or logistic regression depending on the situation. Let's denote n the number of features ($x \in \Re^{n+1}$) and m the number of training examples.

- if n is large relative to m (e.g. $n \geq m, n = 10^4, m = 10..1000$), use a logitstic regression or linear SVM
- if n is small and m intermediate (e.g. $n = 1..1000, m = 10..10^4$), use SVM with gaussian kernel
- if n is small and m is large (e.g. $n = 1..1000, m = 50.10^4+$), create or add more features and then use a logistic regression or a linear SVM.

Neural networks are likely to work better in all these situations but slower to train. Also, don't forget to normalize (center and reduce the data before applying one of these algorithms).

9 Clustering

9.1 Unsupervised learning introduction

In the previous sections, we dealt with supervised learning algorithms where the dataset was made of labeled data. In unsupervised learning, we use unlabeled data. In this situation, we try to identify a structure in the data, to isolate clusters.

9.2 K-means algorithm

The K-means algorithm is a simple algorithm to extract a predefined number K of clusters :

1. generate randomly K so-called centroids $\mu_i, i \in [1..K]$
2. repeat centroid assignement, centroid move

In the centroid assignement step, we just assign the closest centroid to each element of the training set. In the centroid move step, we update the centroids as the mean of the vectors assigned to this cluster. We can repeat these steps until the centroids don't get modified.

The K-means algorithm can be seen as minimizing the distorsion defined as :

$$J(\{c^{(i)}, i \in [1..m]\}, \{\mu_j, j \in [1..K]\}) = \sum_{i=1}^{m} ||x^{(i)} - \mu_{c^{(i)}}||^2 \tag{22}$$

where $c^{(i)}$ is the cluster number of the exemple $x^{(i)}$ and μ_j is the centroid of cluster j. The cluster assignement step minimizes the cost function with respect to the assignements $c^{(i)}$ (i.e. μ_j held fixed) while the centroid move step minimizes the cost function with respect to the centroid centers μ_j (i.e. the assignements held fixed).

When initializing the centroids, it is better to pick randomly K training examples. Indeed, there are some situations where the algorithm would have very bad performances if the clusters are initialized purely randomly. For example, if the clusters are initialized far from the examples, it is possible that one cluster will get all the training data, it will be moved closer to the distribution of the data and might always be the only populated cluster, the others staying far away from the data. This might be avoided when picking K random training samples for initializing the clusters.

Also, the algorithm will converge and maintain its assignements at some point in time but this might correspond to a local minimum of the distorsion. So it is better to run several times the algorithm and keep only the run in which the distorsion is minimum.

To choose the number of clusters, it might be pretty easy in some situations (say when you can actually plot the data, ... but even there it might not be always clear how many clusters should considered). One method to tune the number of clusters is to plot the quality of the clustering function of the number of clusters and select the smallest number of clusters that does not show a significant improvement (the elbow method). The quality might be assessed from the distorsion or from the explained variance ratio (F-test). If clustering is one step of a larger machine learning system, one might also choose the number of clusters that lead to an overall better performance.

10 Dimensionality reduction

There are several reasons why one is interested in reducing the dimensionality of some data. One of them is to compress the information for example for a training algorithm for which the execution time depending on the dimension of the feature vector. Another motivation is for visualizing a dataset in 2D/3D instead of the original, say, 50 dimensional space.

Principal Component Analysis is one way to perform such a dimensionality reduction. In general, it is formulated as finding k vectors such that the projection error of n data is minimized [7]. To compute the PCA :

1. mean normalize the data (optionally feature scale the data as well)
2. compute the covariance matrix of the normalized data : $\Sigma = \sum_i (x^{(i)})^T (x^{(i)}$
3. compute the SVD of the covarance matrix $U.S.V = \text{svd}\Sigma$
4. retain only the K first columns of U, i.e. the K eigen vectors with the largest singular components in S.

By projecting the data on the selected vectors $U_j, j \in [1..K]$, we get the compressed data (the scalar product of a sample on these vectors). We can also reconstruct an approximation of the original data by summing the U_i vectors weighted by their projection value. To select the number of dimensions, one should compute the ratio of the average reconstruction error to the total variance of the data and select K such that this ratio is lower than 0.01 if we wish to retain 99% of the variance, or 0.05 if we wish to retain 95% of the variance.

It might be useful to use PCA to reduce the number of features of a supervised learning algorithm but it is a **bad option** to use it to prevent overfitting (it is much better to use regularization).

11 Anomaly detection

7. it is different from linear regression where we seek to minimize the norm of the residual not of the projection error.