Mathematical Approach of Machine Learning Algorithm III

– A Short Introduction –

Julien Lin

Department of Bioengineering | Imperial College London | 2015-2016

Part I

Introduction to

Supervised Machine Learning

Part I

Chapter 3

Optimization

Optimization

Optimizing Machine Learning Algorithm is important to increase the accuracy of the model. Several techniques are commonly used to overcome issues such as Underfitting/Overfitting and Bias/Variance when fitting the model to a new unseen testing set.

I. Overfitting and Regularization

Regularization enable to overcome a problem called overfitting and thus, to run the machine learning algorithm much faster.

Issue: What is overfitting?

1) Example of overfitting with Linear Regression (housing prices)

The problem with overfitting is that by using a high order polynomial when fitting to the training model, the hypothesis function will be fitted the training set too specifically. This might result in high accuracy when applying the hypothesis to the training set, but because this hypothesis function were built specifically to the training set, this might end up in fitting to the testing set poorly (i.e. lower accuracy).

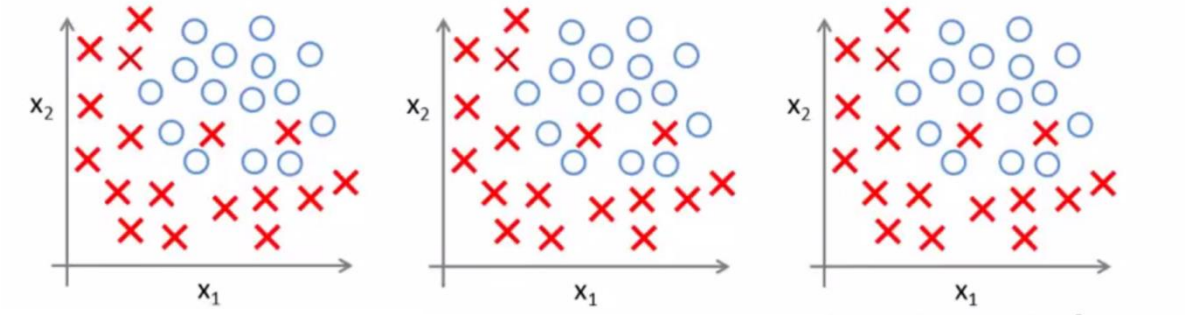
"Overfitting means that the hypothesis fails to generalize on new examples that was not in the training set"

In more details,

Overfitting occurs when due to too many features. The hypothesis may fit the training set very well such as

$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2 \approx 0$, but fail to generalize to new examples (i.e. here, fail to predict prices on new examples). This means that the training error $J_{train}(\theta)$ is likely to be lower than the actual generalization error once parameters $\theta_0, \theta_1, \ldots, \theta_4$ were fit to the training set.

2) Example of overfitting with Logistic Regression



$h_\theta(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2)$

g is a sigmoid function

Poor Fitting of the Training Set.

This problem is call "**Underfitting**", which means that the algorithm has "**High Bias**"

$h_\theta(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1^2 + \theta_4 x_2^2 + \theta_5 x_1 x_2)$

Perfectly fitted

$h_\theta(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_1^2 + \theta_2 x_1^2 x_1 + \theta_3 x_3^3 + \theta_4 x_4^4 + \ldots)$

Seems to fit the Training Set well, but too well.

This problem is call "**Overfitting**", which means that the algorithm has "**High Variance**"

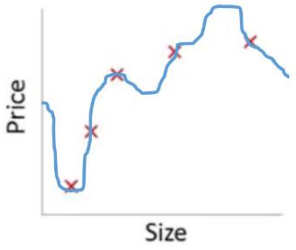
3) Solutions to Underfitting and Overfitting

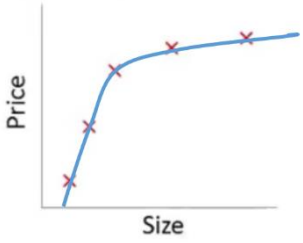
As it is shown in the linear and logistic regression examples, deciding on the degree of polynomial is important to obtain a model that fit well to the data without any underfitting or overfitting issue.

In the case of a learning problems where there are a lot of features ($x_1, x_2, \dots, x_{100}$), it is often hard to plot and visualize data and thus, to choose the right features to construct a model (i.e. deciding on the degree of polynomial). To overcome the issue of Overfitting, the main options are:

- Reduce the number of features by:
  - Select <u>manually</u> the most important features and throw away the one that are useless. This is a long process but enable to know which features to throw away or not.
  - Use a *model selection algorithm*, which is a faster process but the disadvantage is that it might throw away some of important features
- Regularization.
  - Keep all the features, but reduce the values of parameters θ_j.
  - Works well when having a lot of features. Each of the features will contribute a bit to predict y.

4) Regularization and cost function

In order to pass from 

to 

,

$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$ $\quad$ $h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2$

The predictive model need to become simpler, to pass from $h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$ to $h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2$ by setting $\theta_3 x^3 \approx 0 \; and \; \theta_4 x^4 \approx 0$

"Penalization means that the chosen parameters need to be minimise as much as possible (usually near 0) in order to obtain a less complicated hypothesis"

In other term, the predictive model need to be penalized by making θ_3, θ_4 really small, such as:

<u>For example</u>:

If the optimization objective was

$$minimize_\theta \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 + 1000\theta_3^2 + 1000\theta_4^2$$

Where in this case $1000\theta_3^2 \; or \; 1000\theta_4^2$ are huge numbers.

To minimize this cost function, the only way to do it is to setup $\theta_3 \approx 0 \; and \; \theta_4 \approx 0$

This idea of setting small values for parameters $\theta_0, \theta_1, \dots, \theta_n$ to simplify hypothesis model (i.e. reduction of the number of features, here $x^3 \; and \; x^4$ by setting their parameter $\theta_3 \approx 0 \; and \; \theta_4 \approx 0$) that are less prone to overfitting is called **Regularization**.

<u>In more details,</u>

To set parameters $\theta \approx 0$ (here, for instance $\theta_3 \approx 0 \; and \; \theta_4 \approx 0$), a regularisation term $\lambda \sum_{i=1}^{n} \theta_j^2$ is added to the cost function (here, for linear regression) $J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2$ such as:

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

$\lambda \sum_{i=1}^{n} \theta_j^2$ enables the penalization all the parameters $\theta_1, \theta_2, \dots, \theta_n$ is order to simplify more or less the predictive function, allowing a better fitting to the dataset

λ is the regularization parameter that controls a *trade-off* between two different aims. The 1st aim is to fit the predictive model to training data well (i.e. $\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2$) while the 2nd aim (i.e. *regularization objective*) is to keep the parameters $\theta_1, \theta_2, \dots, \theta_n$ as small as possible (i.e. $\lambda \sum_{i=1}^{n} \theta_j^2$). In other term, λ **controls** how well to fit the predictive model to the training set while keeping the parameters $\theta_1, \theta_2, \dots, \theta_n$ small in order to avoid overfitting.

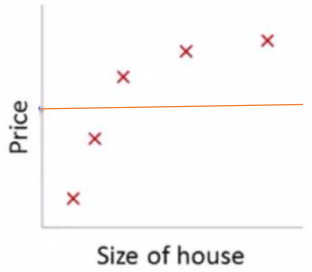
Issue: What is λ is set as a **very large value** ($\lambda = 10^{10}$)?

For $h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$, $\theta_1, \theta_2, \theta_3, \theta_4$ are going to be penalized such as

$$\theta_1 \approx 0, \theta_2 \approx 0, \theta_3 \approx 0, \theta_4 \approx 0$$

$$h_\theta(x) = \theta_0 + \cancel{\theta_1 x_1} + \cancel{\theta_2 x^2} + \cancel{\theta_3 x^3} + \cancel{\theta_4 x^4},$$

$$\text{For } \lambda = 10^{10},\ h_\theta(x) \approx \theta_0$$



This is an "**Underfitting** situation" where the price of all houses (with different sizes) will be predicted to be the same (i.e. equal to θ_0)

5) *Regularized Linear regression*

The Optimization Objective will be

$$minimize_\theta\ regularised\ J(\theta) = \frac{1}{2m}\left[\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2 + \lambda \sum_{i=1}^{n} \theta_j^2\right]$$

To $minimize_\theta\ regularized\ J(\theta)$, **Gradient Descent** could be used such as:

Repeat {

$$\boldsymbol{\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) * x_0{}^{(i)}} \quad \left(\boldsymbol{\frac{\partial}{\partial \theta_0} J(\theta)}\right)$$

The regularization objective does not take in account θ_0 since j∈{1,2,…,n}

$$\boldsymbol{\theta_j := \theta_j - \alpha \left[\frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) * x_j{}^{(i)} + \frac{\lambda}{m}\theta_j\right]}$$

$$\left(\boldsymbol{Regularized \frac{\partial}{\partial \theta_j} J(\theta)}\right)$$

}

Factorization by $\boldsymbol{\theta_j}$**:**

$$\boldsymbol{\theta_j := \theta_j(1 - \alpha \frac{\lambda}{m}) - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) * x_j{}^{(i)}}$$

It is noticed that $\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)}) * x_j{}^{(i)} = \boldsymbol{\frac{\partial}{\partial \theta_j} J(\theta)}$ such as:

$$\theta_j := \theta_j\left(1 - \alpha\frac{\lambda}{m}\right) - \alpha\frac{1}{m}\frac{\partial}{\partial\theta_j}J(\theta)$$

This term is identical to the one in gradient descent that is not regularized

$$When\ \alpha\ is\ small\ and\ m\ is\ large, 1 - \alpha\frac{\lambda}{m} < 1$$

Since $\theta_j\left(1 - \alpha\frac{\lambda}{m}\right)$, $the\ regularization\ parameter\ \lambda$ in $(1 - \alpha\frac{\lambda}{m})$ enables to **shrink** the value θ_j

The other algorithm that can be used instead of gradient descent is the **Normal Equation**:

For *m* **example** $(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)})$; *n* **features,**

$$x^{(i)} = \begin{bmatrix} x_0^{(i)} \\ x_1^{(i)} \\ \dots \\ x_n^{(i)} \end{bmatrix};\ (x^{(i)})^T = [x_0^{(i)}\ \ x_1^{(i)}\ \dots\ x_n^{(i)}];\ X = \begin{bmatrix} -(x^{(1)})^T - \\ -(x^{(2)})^T - \\ \dots \\ -(x^{(m)})^T - \end{bmatrix}; y = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \dots \\ y^{(m)} \end{bmatrix}$$

m * (n+1) dimensional matrix

Finally, the optimal value of θ that $minimize_\theta\ J(\theta)$ can be found by computing

$$\theta = (X^T X)^{-1} X^T y$$

Then, the regularized term will be

$$\theta = \left(X^T X + \lambda \begin{vmatrix} 0 & 0 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & 0 & \dots & 0 \\ 0 & 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 0 & 1 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & 0 & \dots & 1 \end{vmatrix} \right)^{-1} X^T y$$

(n+1) * (n+1) dimensional matrix

For the case of **non-invertibility,**

Suppose *m<n,*

$\theta = (X^T X)^{-1} X^T y$ is described as non-invertible/singular

So long as the regularization parameter $\lambda > 0$,

$$\theta = \left(X^T X + \lambda \begin{vmatrix} 0 & 0 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & 0 & \dots & 0 \\ 0 & 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 0 & 1 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & 0 & \dots & 1 \end{vmatrix} \right)^{-1} X^T y$$ is invertible/not singular. Thus, using regularization solves the problem of non-invertibility

6) Regularized logistic regression

The Optimization Objective will be

$$minimize_\theta \ regularized\ J(\boldsymbol{\theta}) = -\frac{1}{m}[\sum_{i=1}^{m} \mathrm{y}^{(i)} \log\left(h_\theta(x^{(i)})\right) + (1-\mathrm{y}^{(i)}) \log\left(1-h_\theta(x^{(i)})\right)] + \frac{\lambda}{2m}\sum_{i=1}^{n}\theta_j^2$$

Similarly to regularised linear regression,

To $minimize_\theta \ regularized\ J(\theta)$, **Gradient Descent** could be used such as:

Repeat {

$$\boldsymbol{\theta_0} := \boldsymbol{\theta_0} - \boldsymbol{\alpha}\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)}) * {x_0}^{(i)} \qquad \boldsymbol{\frac{\partial}{\partial \theta_0} J(\theta)}$$

The regularization objective does not take in account θ_0 since j∈{1,2,…,n}

$$\boldsymbol{\theta_j} := \boldsymbol{\theta_j} - \boldsymbol{\alpha}\left[\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)}) * {x_j}^{(i)} + \frac{\lambda}{m}\boldsymbol{\theta_j}\right]$$

$$\boldsymbol{Regularized}\ \boldsymbol{\frac{\partial}{\partial \theta_j} J(\theta)}$$

}

Where $h_\theta(x) = \frac{\mathbf{1}}{\mathbf{1}+\boldsymbol{e}^{-\boldsymbol{\theta}^T x}}$

Factorization by $\boldsymbol{\theta_j}$**:**

$$\boldsymbol{\theta_j} := \boldsymbol{\theta_j}(\mathbf{1} - \boldsymbol{\alpha}\frac{\lambda}{m}) - \boldsymbol{\alpha}\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)}) * {x_j}^{(i)}$$

It is noticed that $\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)}) * {x_j}^{(i)} = \boldsymbol{\frac{\partial}{\partial \theta_j} J(\theta)}$ such as:

$$\boldsymbol{\theta_j} := \boldsymbol{\theta_j}(\mathbf{1} - \boldsymbol{\alpha}\frac{\lambda}{m}) - \boldsymbol{\alpha}\frac{1}{m}\boldsymbol{\frac{\partial}{\partial \theta_j} J(\theta)}$$

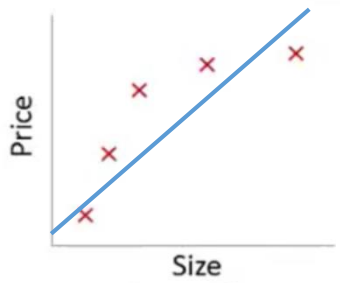
This term is identical to the one in gradient descent that is not regularized

$$When\ \boldsymbol{\alpha}\ is\ small\ and\ \boldsymbol{m}\ is\ large, \mathbf{1} - \boldsymbol{\alpha}\frac{\lambda}{m} < 1$$

Since $\boldsymbol{\theta_j}\left(\mathbf{1} - \boldsymbol{\alpha}\frac{\lambda}{m}\right)$, $the\ regularization\ parameter\ \lambda$ in $(\mathbf{1} - \boldsymbol{\alpha}\frac{\lambda}{m})$ enables to **shrink** the value $\boldsymbol{\theta_j}$

II. Variance and Bias

1) *General Understanding*

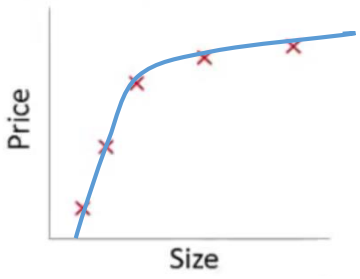


$h_\theta(x) = \theta_0 + \theta_1 x$

Poor Fitting of the Training Set.

Degree of polynomial, d=1

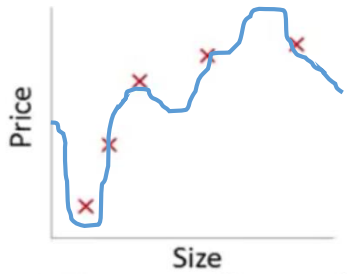
This problem is call "**Underfitting**", which means that the algorithm has "**High Bias**"



$h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2$

Perfectly fitted

Degree of polynomial, d=2



$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4 + \ldots$

Seems to fit the Training Set well, but too well.

Degree of polynomial, d=4

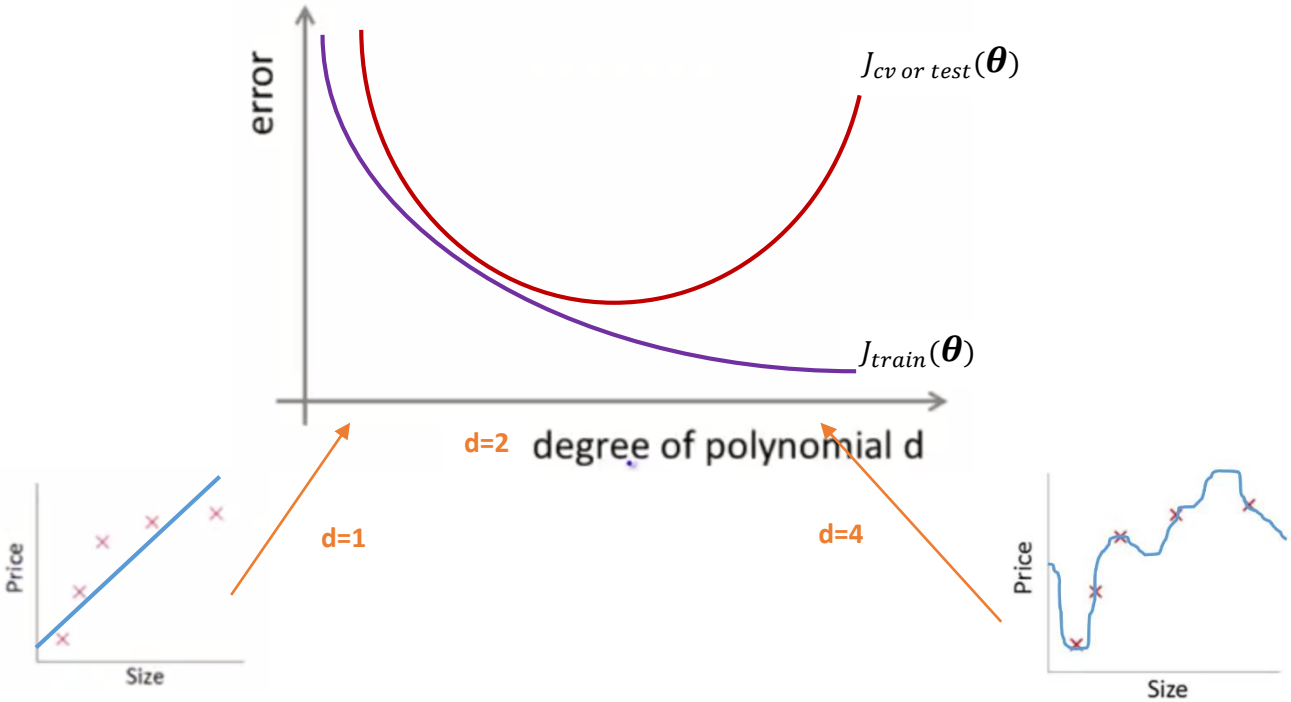
This problem is call "**Overfitting**", which means that the algorithm has "**High Variance**"

Now, suppose

Training error: $J_{train}(\boldsymbol{\theta}) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2$

Cross validation error: $J_{cv}(\boldsymbol{\theta}) = \frac{1}{2m_{cv}}\sum_{i=1}^{m_{cv}}(h_\theta(x_{cv}{}^{(i)}) - y_{cv}{}^{(i)})^2$

Test error: $J_{test}(\boldsymbol{\theta}) = \frac{1}{2m_{test}}\sum_{i=1}^{m_{test}}(h_\theta(x_{test}{}^{(i)}) - y_{test}{}^{(i)})^2$



For the training set

At d=1, the hypothesis $h_\theta(x)$ fits to the Training sets very poorly, and the error $J_{train}(\boldsymbol{\theta})$ is very high.

At d=2, the hypothesis $h_\theta(x)$ fits to the Training sets well, and the error $J_{train}(\boldsymbol{\theta})$ is low.

At d=4, the hypothesis $h_\theta(x)$ fits to the Training sets very well, and the error $J_{train}(\boldsymbol{\theta})$ is very low.

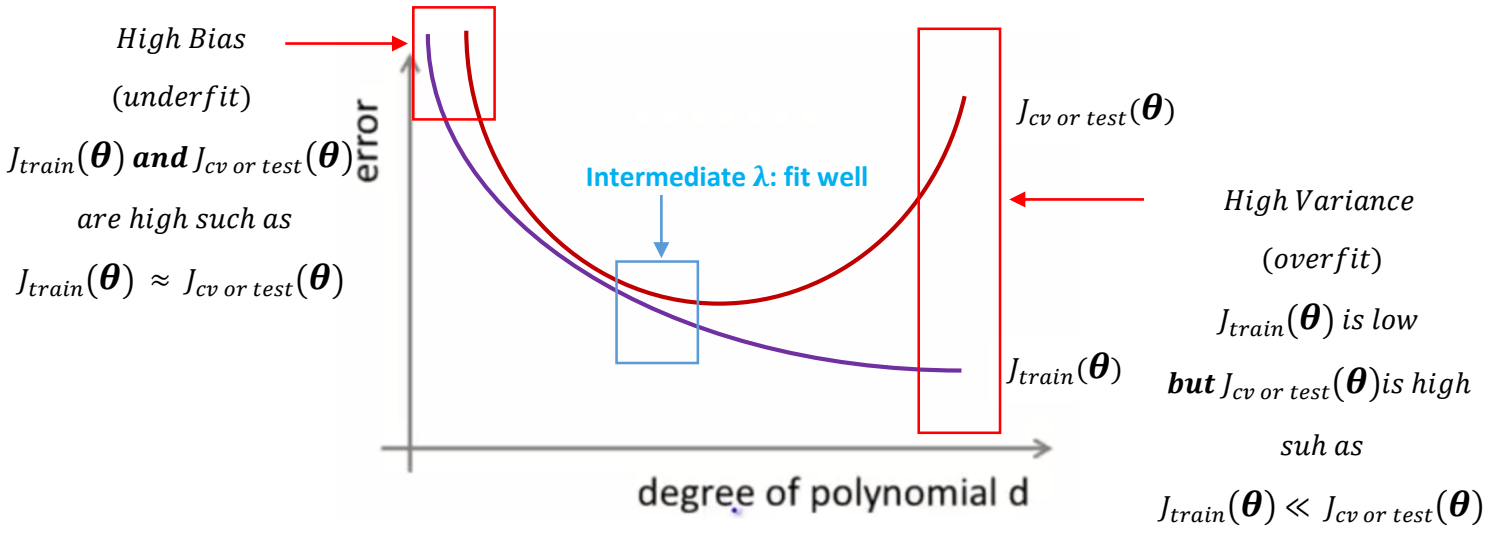
For the cross validation and the test set

At d=1, the hypothesis $h_\theta(x)$ underfits to the cv or test sets very poorly, and the error $J_{cv\ or\ test}(\boldsymbol{\theta})$ is very high.

At d=2, the hypothesis $h_\theta(x)$ fits to the cv or test sets well, and the error $J_{cv\ or\ test}(\boldsymbol{\theta})$ is low.

At d=4, the hypothesis $h_\theta(x)$ overfits to the cv or test sets poorly, and the error $J_{cv\ or\ test}(\boldsymbol{\theta})$ is very high.

Issue: In the case of a learning algorithm that end up with a high $J_{cv\ or\ test}(\boldsymbol{\theta})$ (i.e. performing less efficiently than expected), how is it possible to know that it is a problem of bias rather than variance?



When fitting a very **low order polynomial**, if $J_{train}(\boldsymbol{\theta})$ ***and*** $J_{cv\ or\ test}(\boldsymbol{\theta})$ are high, the algorithm might suffer from **High Bias.**

When fitting a very **high order polynomial**, if $J_{train}(\boldsymbol{\theta})$ *is low* ***but*** $J_{cv\ or\ test}(\boldsymbol{\theta})$*is high*, the algorithm might suffer from **High Variance.**

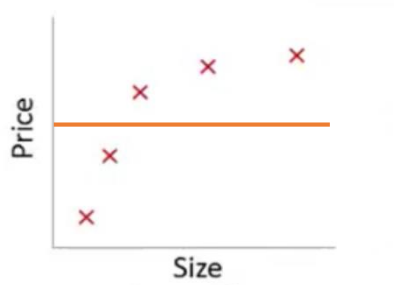
2) Regularization and Bias/Variance

Issue: since regularization help to prevent overfitting, how does it affect the bias and the variances of a learning algorithm?

Suppose

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4 \text{ and}$$

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



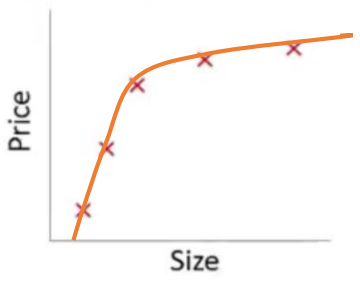
$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$

LARGE λ

High bias (underfitting)

$\boldsymbol{\lambda = 1000}$.

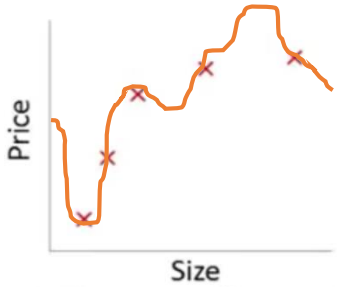
$\theta_1 \approx 0, \theta_2 \approx 0, \theta_3 \approx 0, \theta_4 \approx 0$



$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$

Intermediate λ

Perfectly fitted



$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$

SMALL λ

High variance (overfitting)

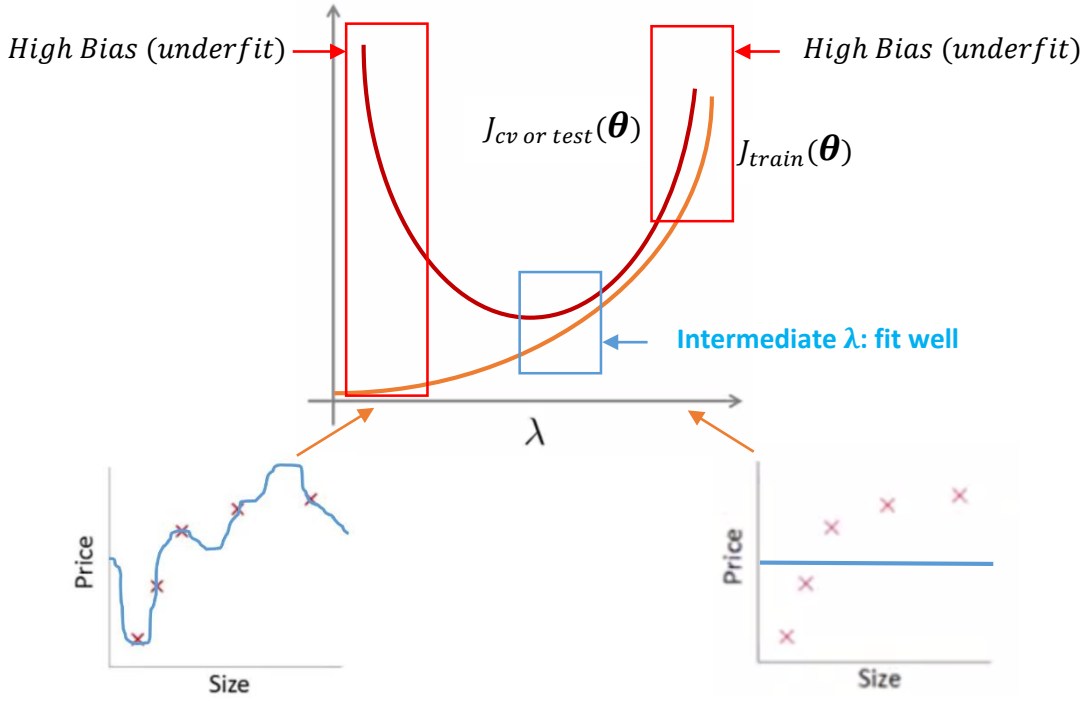
$\boldsymbol{\lambda = 0}$

To choose an adequate value of the regularization parameters **λ,** it is possible to try several values of **λ** such as

1. Try $\lambda = 0 \rightarrow minimize_\theta\ regularized\ J_{train}(\theta) \rightarrow \theta^{(1)} \rightarrow J_{cv}(\theta^{(1)})$
2. Try $\lambda = 0.0 \rightarrow minimize_\theta\ regularized\ J_{train}(\theta) \rightarrow \theta^{(2)} \rightarrow J_{cv}(\theta^{(2)})$
3. Try $\lambda = 0.02 \rightarrow minimize_\theta\ regularized\ J_{train}(\theta) \rightarrow \theta^{(3)} \rightarrow J_{cv}(\theta^{(3)})$
4. Try $\lambda = 0.04 \rightarrow minimize_\theta\ regularized\ J_{train}(\theta) \rightarrow \theta^{(4)} \rightarrow J_{cv}(\theta^{(4)})$
5. Try $\lambda = 0.08 \rightarrow minimize_\theta\ regularized\ J_{train}(\theta) \rightarrow \theta^{(5)} \rightarrow J_{cv}(\theta^{(5)}) \rightarrow J_{test}(\theta^{(5)})$

   …
12. Try $\lambda = 10.24 \rightarrow minimize_\theta\ regularized\ J_{train}(\theta) \rightarrow \theta^{(12)} \rightarrow J_{cv}(\theta^{(12)})$

In order to select the $\theta^{(i)}$ with the lowest error on the cross validation set $J_{cv}(\theta^{(i)})$ and apply it to the test set and see which test error value of $J_{test}(\theta^{(i)})$ might result.



SMALL λ

High variance (overfitting)

$\boldsymbol{\lambda = 0}$

LARGE λ

High bias (underfitting)

$\boldsymbol{\lambda = 1000}$.

$\theta_1 \approx 0, \theta_2 \approx 0, \theta_3 \approx 0, \theta_4 \approx 0$

$J_{train}(\boldsymbol{\theta})$ will normally increase when lambda increases because a large value of **λ** corresponds to high bias whereas a small value of lambda corresponds to high variance if a very high degree polynomial is fitted to the training data.

With large **λ**, the predictive model will underfit the cross validation or test set, resulting in high bias and thus, high $J_{cv\ or\ test}(\boldsymbol{\theta})$. With small **λ**, the predictive model will overfit the cross validation or test set, resulting in high variance and thus, high $J_{cv\ or\ test}(\boldsymbol{\theta})$

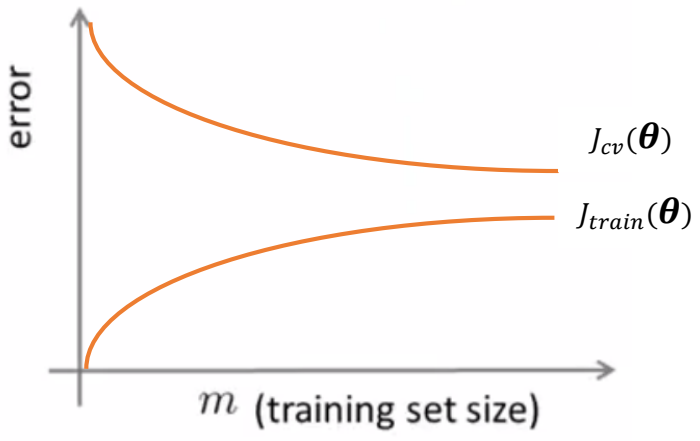
3) Learning curve

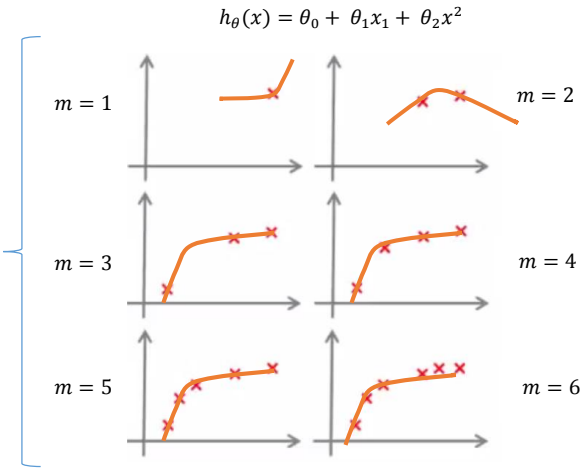
Learning curve is a tool that is used to diagnose if a machine learning algorithm is suffering from a bias or a variance problem in order to provide sufficient guidance on how to improve its performance.

Suppose,

Training error: $J_{train}(\boldsymbol{\theta}) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2$

Cross validation error: $J_{cv}(\boldsymbol{\theta}) = \frac{1}{2m_{cv}}\sum_{i=1}^{m_{cv}}(h_\theta(x_{cv}{}^{(i)}) - y_{cv}{}^{(i)})^2$

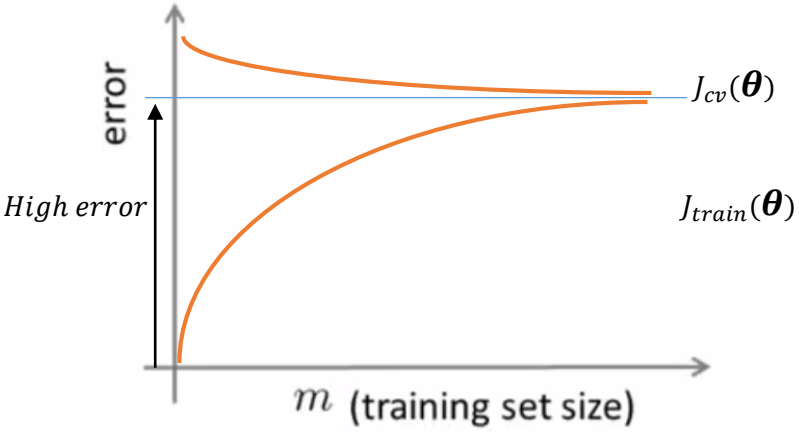


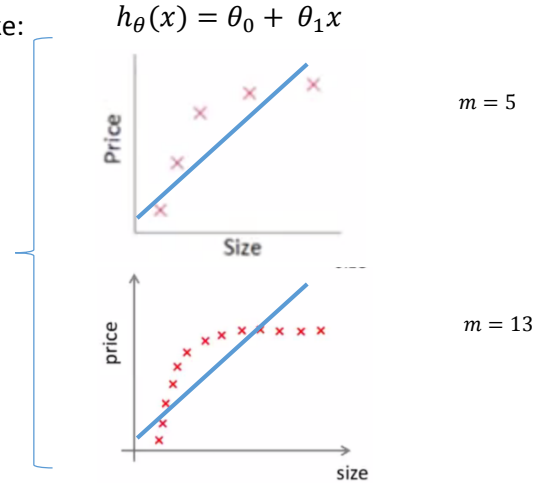


At m *increase*, it becomes harder and harder to fit perfectly to all the training set

To plot a learning curve, only 10-40 training examples are intentionally used when plotting the training error
As m is small, it is easier to fit the data while as m becomes bigger, it becomes harder to fit the training set.
When m is small, the hypothesis generally fails to generalising the cross validation set whereas as the size of the training set increases, the hypothesis may fit better to the cross validation set.

In case of **high bias** (i.e. underfitting), the learning curve will look like:





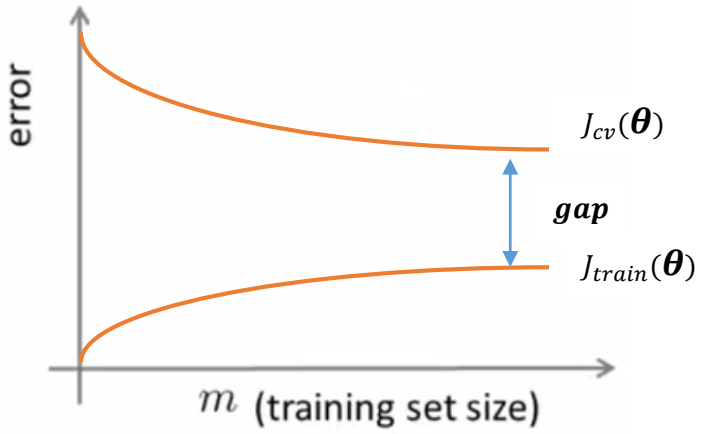
Adding more training examples in this case does not help to fit the model to the training set better

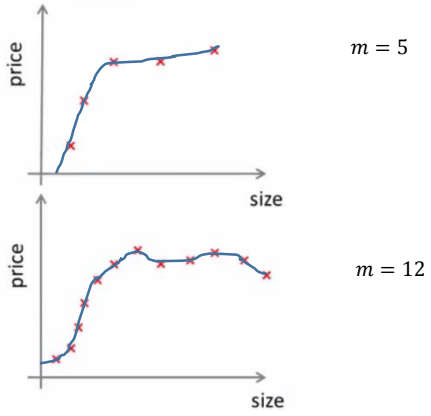
Since adding more training examples after a certain point does not help to fit the model to the training set better, the cross-validation error plateau out whereas the training error will be small when m is small and will end up close to the cross validation error in the high bias case because having very few parameters and very large m, the training and the cross validation set will actually end up to be very similar.

Conclusion: If a learning algorithm is suffering from high bias, adding more training example m will not help to fit better the model to the dataset. This is a useful information because it will prevent you from spending more time collecting more training example.

In case of **high variance** (i.e. overfitting), the learning curve will look like:

$h_\theta(x) = \theta_0 + \theta_1 x + \cdots + \ldots + \theta_{100} x^{100}$ with small $\boldsymbol{\lambda}$

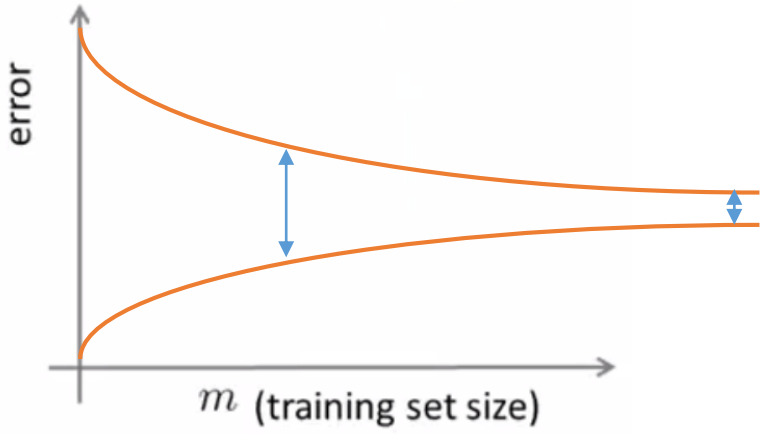




Adding more training examples in this case helps to fit a better model to the training set

Even if increasing when m increase, the training error stay still very low whereas the cross validation set error decrease slightly.

Conclusion: If a learning algorithm is suffering from high variance, adding more training data might help such as:



By adding more training example, $J_{cv}(\boldsymbol{\theta}) \approx J_{train}(\boldsymbol{\theta})$

<u>Issue</u>: Suppose an implemented regularized linear regression to predict housing prices. However, the hypothesis in a new set of houses makes an unacceptably large errors in its prediction. What is it possible to try to solve this issue?

Fix High Variance	Fix High Bias
-Get more training examples (increase m) -Try smaller sets of features (reduce n) -Try to increase $\boldsymbol{\lambda}$	-Try adding more features (increase n) -Try adding polynomial features ($\boldsymbol{x_1^2, x_2^2, x_3^2}$) -Try to decrease $\boldsymbol{\lambda}$

III. Scaling Up

In practice, when deal with very big datasets (e.g. m=1,000,000,000) for an **Online Learning** for example, some computationally efficient method such as **Stochastic Gradient Descent** or **Map Reduce** are used to fix high variance or high bias after plotting the learning curve.

1) *Stochastic Gradient Descent*

Linear Regression is used as an example.

Knowing that for Linear Regression with Gradient Descent,

- Hypothesis Function:

$$h_\theta = \sum_{j=0}^{n} \theta_j x_j$$

- Cost Function:

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2$$

- *"Batch"* Gradient Descent:

Repeat {

$$\theta_j := \theta_j - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)}) * x_j{}^{(i)}$$

(for every *j=0,…,n*)

}

This type of gradient descent that is used in this section is called "*Batch*" Gradient Descent. In "*Batch*" Gradient Descent, all the training examples (e.g. $\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})$) are used at each step of the gradient descent when updating the algorithm. Other type of gradient descent might use only one or a subset of the training examples instead of all the training set at each step when updating the algorithm.

Since "*Batch*" Gradient Descent updates all the training examples at each step, when dealing with a very large dataset (m=1,000,000), the algorithm might at the end be slow.

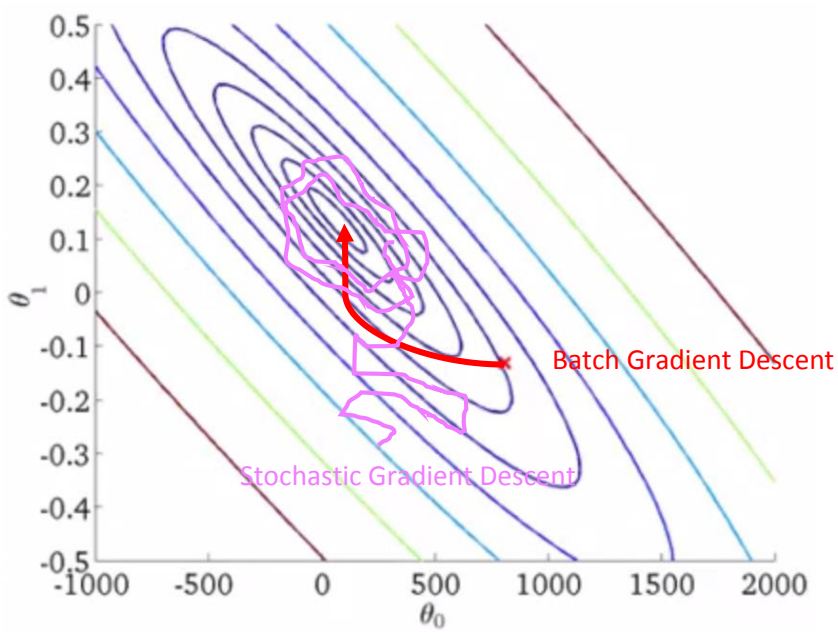
To solve this problem, an alternative would be to be "*Stochastic Gradient Descent*", defined as:

Batch Gradient Descent	Stochastic Gradient Descent
$J_{train}(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2$	$cost\left(\theta, \left(x^{(i)}, y^{(i)}\right)\right) = \frac{1}{2}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2$ $J_{train}(\theta) = \frac{1}{m}\sum_{i=1}^{m} cost\left(\theta, \left(x^{(i)}, y^{(i)}\right)\right)$

Repeat {
$\theta_j := \theta_j - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)}) * x_j{}^{(i)}$
(for every *j=0,…,n*)
}
$\frac{\partial}{\partial\theta_j}J_{train}(\theta)$
	1. Randomly Shuffle Dataset.
	2. Repeat{
	for i=1,…,m{
	$\theta_j := \theta_j - \alpha(h_\theta(x^{(i)}) - y^{(i)}) * x_j{}^{(i)}$
	(for every *j=0,…,n*)
	}
	)
	$\frac{\partial}{\partial\theta_j} cost\left(\theta, \left(x^{(i)}, y^{(i)}\right)\right)$

Thus, Stochastic Gradient Descent differ from Batch Gradient Descent by scanning through the training examples by looking at first to $(x^{(1)}, y^{(1)})$ before taking a basic little gradient descent step with respect to the cost function of just this first training example. In more details, the Stochastic Gradient Descent modify the parameters to try to fit just the 1st training a little bit better. Then, the *for* loop is going to continue to the 2nd training example $(x^{(2)}, y^{(2)})$ before taking again a basic little gradient descent step. Once done, the *for* loop is going to continue to the 3rd, the 4th, etc… until getting the entire training set. Finally, the whole process is going through the *Repeat* loop. Depending on the size of the training set (m=300,000,000), doing the *Repeat* loop <u>just 1-10 times</u> may be enough to obtain a perfectly good hypothesis. The randomly shuffle dataset ensure that the training set will be sorted in a random order.

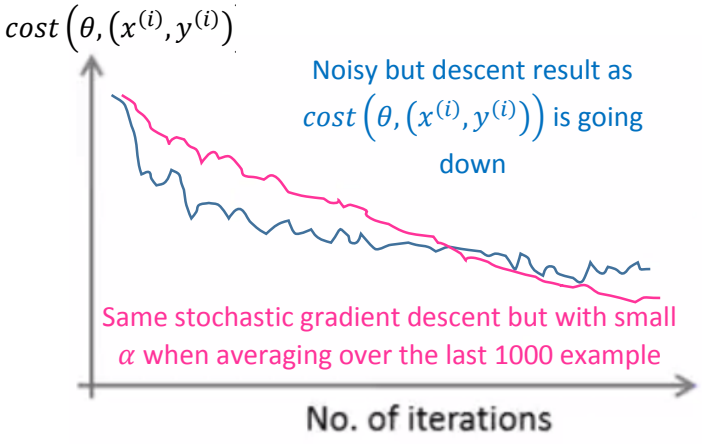
Rather than waiting to sum up the gradient terms over all m training examples (i.e. Batch Gradient Descent), the gradient term using just one single training example is taken to make progress in improving the parameters (i.e. Stochastic Gradient Descent).

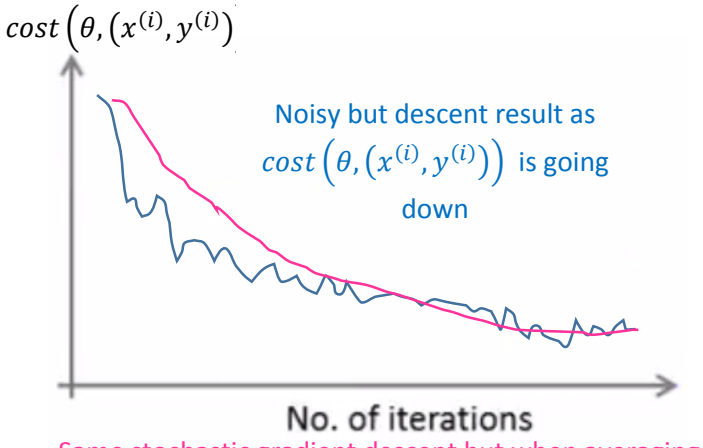


Since all the iterations will be much faster because there is no need to sum up over all the training examples, every iteration will just try to fit a single training example better. However, when running the Stochastic gradient descent, it will generally move the parameters in the direction of the global minimum but not always. Graphically, it will take some more random-looking circuitous path towards the global minimum but does not actually converge as Batch gradient descent does. In more detail, Stochastic gradient descent will ends up wandering around continuously in some regions that are closed to the global minimum without reaching it. However, as parameters end up pretty close to the global minimum, it will be still a pretty good hypothesis.

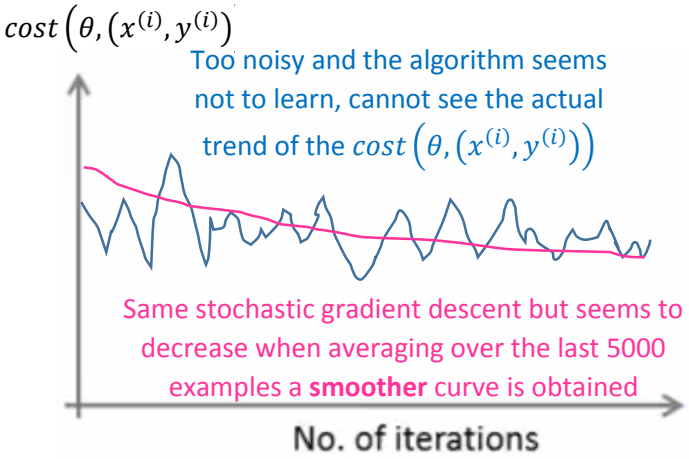
<u>Issue:</u> How to make sure that the hypothesis of Stochastic gradient descent is accurate enough since it does not reach the global minimum? How is the learning rate α chosen?

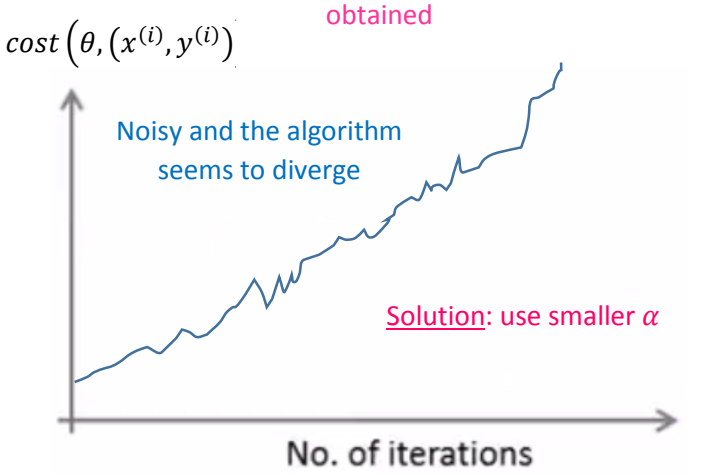
The standard method to check if Batch gradient descent is converging is to plot $J(\theta)$ as a function of the number of iterations of gradient descent. In contrast, while Stochastic gradient descent is learning, it is important to compute $cost\left(\theta, \left(x^{(i)}, y^{(i)}\right)\right)$ before updating θ using $\left(x^{(i)}, y^{(i)}\right)$. Then, to check convergence for 1000 iterations (more time-efficient), plot $cost\left(\theta, \left(x^{(i)}, y^{(i)}\right)\right)$averaged over the last 1000 examples processed by algorithm.



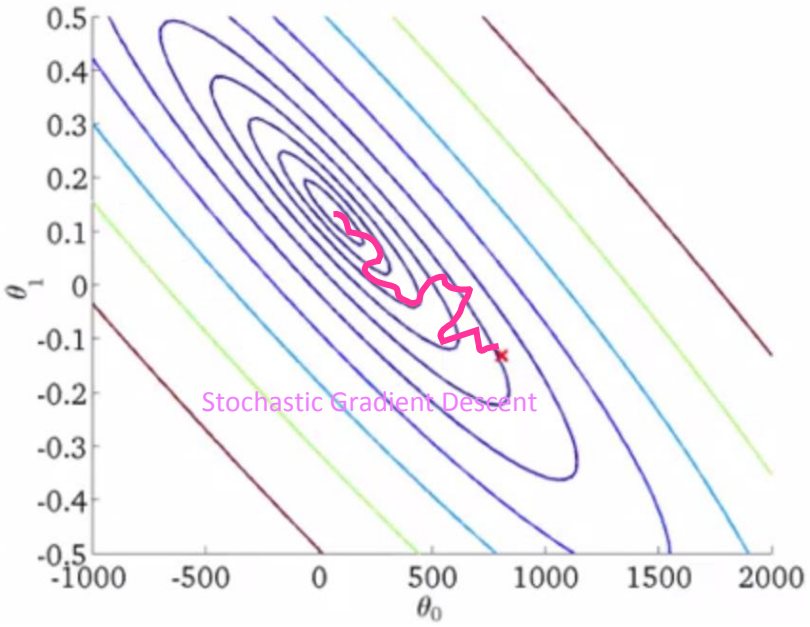


Same stochastic gradient descent but when averaging over the last 5000 examples, a **smoother** curve is obtained





Decreasing slowly α over time instead of keeping it constant over time enable θ to converge by setting as $\alpha = \frac{const\ 1}{iterationNumber + const\ 2}$. As the number of iterations come bigger, α will become smaller and will tend more and more to 0 until converging.



2) *Mini-Batch Gradient Descent*

Mini-Batch Gradient Descent is also an alternative to Batch and Stochastic Gradient Descent.

- Batch gradient descent: Use all *m* examples in each iteration
- Stochastic gradient descent: Use only 1 example in each iteration
- Mini-batch gradient descent Use *b* examples in each iteration
  Where *b* is the mini-batch size, usually *b*=2-100 examples if *m* was very large.

If *b*=10 training examples and *m=1000*, given $(x^{(i)}, y^{(i)})$,…, $(x^{(i+9)}, y^{(i+9)})$,

Repeat {

for *i*=1,11,21,31,…,991{

$$\theta_j := \theta_j - \alpha \frac{1}{10} \sum_{k=i}^{i+9} (h_\theta(x^{(k)}) - y^{(k)}) * x_j{}^{(k)} \qquad \text{(for every } j=0,\ldots,n)$$

}

}

Since all the iterations will be much faster because there is no need to sum up over all the training examples, every iteration will just try to fit a small batch of training example better. In more details, the first 10 examples are takes to make progress before taking the second ten examples and modify the parameters a little bit again and so on.

Issue: why is it better to look at a smaller batch rather than a single training example?

Mini-batch gradient descent is likely to outperform Stochastic gradient descent only if it has a good vectorise implementation. By using appropriate vectorization to compute the sum over 10 examples, it is possible to use partially a good numerical algebra libraries and parallelize the gradient computations over the *b* examples whereas if having only one training examples at the time, it is not possible to parallelize over. However, one disadvantage of Mini-batch gradient descent is that the parameter *b* need to be defined.

3) *Online Learning*

Online Learning is used when dealing with a huge among of continuous streaming of data. The advantage of this type of Learning is that it can adapt to real-time need.

Suppose a shipping service website where user comes, specifies origin and destination. Ship their package are a service offer for some asking price, and users sometimes choose to use the shipping service (y=1) and sometimes not (y=0).

Features x capture properties of user, of origin/destination and asking price. Thus, a logistic regression will be computed to estimate p(y=1|x; θ) to optimize price.

The gradient descent of an online algorithm will be:

Repeat **forever**{

Get (x,y) corresponding to user.

Update θ using (x,y) and not $(x^{(i)} - y^{(i)})${

$$\theta_j := \theta_j - \alpha(h_\theta(x) - y) * x_j \qquad \text{(for every } j=0,\ldots,n)$$

}

}

This algorithm can adapt to user preferences over time.

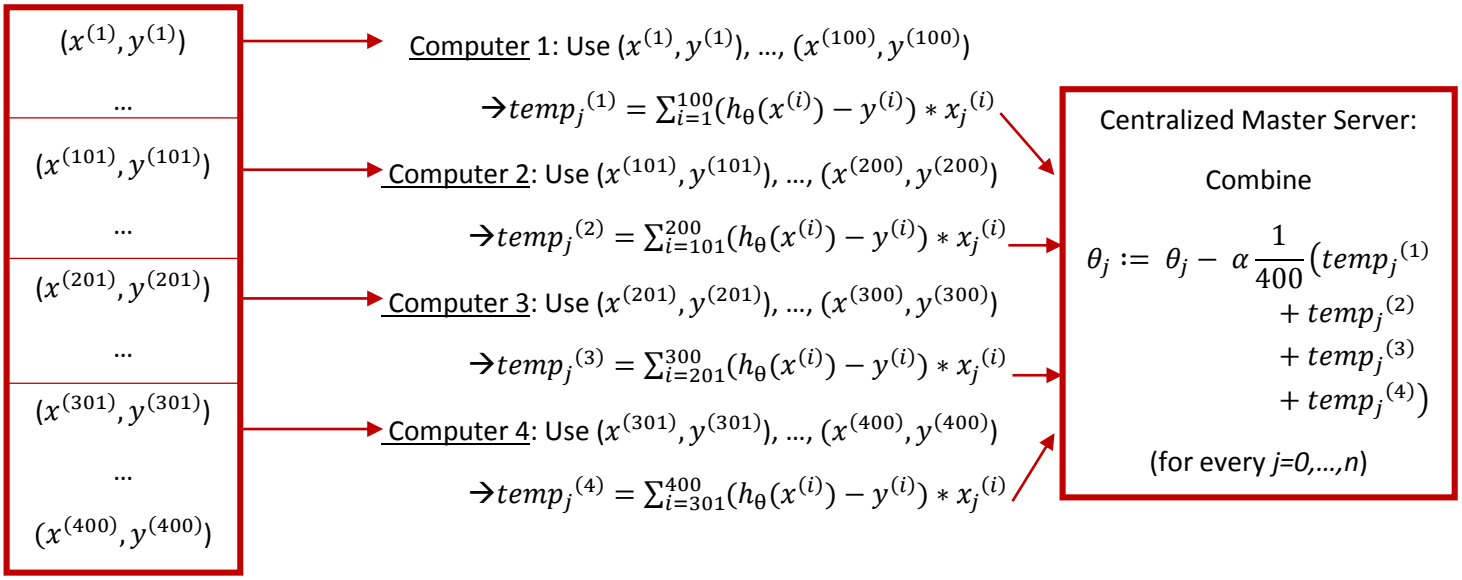
4) Map Reduce and Data Parallelism

Suppose,

Batch gradient descent with m=400:

$$\theta_j := \theta_j - \alpha \frac{1}{400} \sum_{i=1}^{400} (h_\theta(x^{(i)}) - y^{(i)}) * x_j{}^{(i)}$$

The idea of **Map Reduce** and **Data Parallelism** is to divide the total number of training example into equally sized subsets (i.e. mini batches) to run them in parallel using the same number of computers as the number of subsets (i.e. 4 computers for 4 subsets) such as



Since each computer has to sum over only 100 instead of 400 training examples (i.e. has to do only a quarter of the works), the total amount of time spend to compute will be 4 times as fast.

Once all the computers has done the works, all the $temp_j{}^{(i)}$ variables are sent to a centralized master server that will combine the result together by update θ_j such as

$$\theta_j := \theta_j - \alpha \frac{1}{400} (temp_j{}^{(1)} + temp_j{}^{(2)} + temp_j{}^{(3)} + temp_j{}^{(4)})$$

(for every *j=0,...,n*)

$$\theta_j := \theta_j - \alpha \frac{1}{400} (\sum_{i=1}^{100} (h_\theta(x^{(i)}) - y^{(i)}) * x_j{}^{(i)} + \sum_{i=101}^{200} (h_\theta(x^{(i)}) - y^{(i)}) * x_j{}^{(i)} + \sum_{i=201}^{300} (h_\theta(x^{(i)}) - y^{(i)}) * x_j{}^{(i)} + \sum_{i=301}^{400} (h_\theta(x^{(i)}) - y^{(i)}) * x_j{}^{(i)})$$

Every times that an algorithm (gradient descent for linear regression, logistic regression, etc..) can be expressed a sums of functions over the training set, Map-Reduce can be a good candidate for scaling the learning algorithms through a Big Data set.

For further reading, please refer to Jeffrey Dean and Sanjay Ghemawat.

A slight variant to Map Reduce is the **Multi-core Machines**: even if there is only a single computer, map reduce can be applicable. Nowadays, modern computer can have multiple processing cores (i.e., multiple CPUs). Thus, when have a large training set and a single computer with 4 cores, the training set can be split into 4 subsets which can be sent to different cores. Each of the core can carry out the sum over one quarter of the total training set before taking all the partial sum and combine the results in order to get the summation over the entire training set.

The advantage of **Multi-core Machines** compared to Map Reduce that use multiple computer, is that problem such as network latency are solved because all the communication, all the sending of $temp_j{}^{(i)}$ will happens within a single computer.

IV. Metric evaluation

- Start with a **simple algorithm** that can be implemented quickly and test on the cross-validation set.
- Plot **learning Curves** to decide whether adding or discarding more data, features, etc… is likely to help
- **Error analysis**: Manually examine the training examples to spot any systematic trend that might be the source of errors (i.e. try to see which of the features the learning algorithm is doing poorly, having a high errors and is doing correctly, having a high accuracy).

Issue: is there an efficient way of analysing errors in order to avoid false positive or false negative?

1) Error Metrics for Skewed Classes

There is two metrics that is used to analyse errors: **Precision** and **Recall**

Given a cancer classification example where,

A Logistic regression classifier such as $0 \leq h_\theta(x) \leq 1$ is trained on the training sets.

Predict y=1, if $h_\theta(x) \geq 0.5$

Predict y=0, if $h_\theta(x) < 0.5$

This classifier will give some values for Precision and Recall.

Reminder:

Suppose *y=1* in presence of rare class that we want to detect.

Knowing that,

	Actual class = 1 (e.g. having a cancer)	Actual class = 0 (e.g. not having a cancer)
Predicted class = 1 (e.g. predicted to have a cancer)	True Positive	False Positive
Predicted class =0 (e.g. predicted not to have a cancer)	False Negative	True negative

Precision (of all patients where it is predicted y=1, what fraction actually has cancer?):

$$\frac{True\ positives}{Number\ of\ predicted\ positive} = \frac{True\ positives}{True\ positive + False\ positive}$$

Recall (of all patients that actually have cancer, what fraction is correctly detect as having cancer?):

$$\frac{True\ positives}{Number\ of\ actual\ positives} = \frac{True\ positives}{True\ positive + False\ negative}$$

A classifier with high precision and high recall means that the algorithm works well on skewed dataset

Generally speaking, precision and recall are generally used as an **evaluation metrics** for algorithm.

Now,

Suppose we want to predict y=1 only if very **confident**. So instead of having

Predict y=1, if $h_\theta(x) \geq 0.5$

Predict y=0, if $h_\theta(x) < 0.5$

- To make the predict more confident,

  Predict y=1, if $h_\theta(x) \geq \mathbf{0.7}$ → "y=1 is predicted only if the doctors are 70% confident that the patients has cancer"

  Predict y=0, if $h_\theta(x) < \mathbf{0.7}$

This results in having a classifier with a **higher precision** (e.g. cases where cancer has been predicted turns out to be true) but **lower recall** (because y=1 is going to be predicted on a small number of patients)

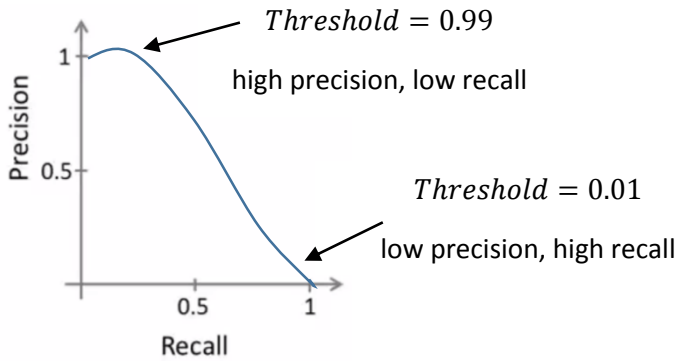
- To avoid missing too many cases of cancer (i.e. avoid false negatives, avoid predicting y=0 while the patient do really have cancer),

  Predict y=1, if $h_\theta(x) \geq \mathbf{0.3}$

  Predict y=0, if $h_\theta(x) < \mathbf{0.3}$

  This results in having a classifier with a **higher recall** (because a higher number of patients will be predicted with y=1) but **lower precision** (because in the higher number of patients predicted with a cancer y=1, there will be a higher fraction that will turn out of not having a cancer, i.e. false positive)

In general, from predict y=1, if $h_\theta(x) \geq \boldsymbol{threshold}$ and As varying the value of this ***threshold*,** it is possible to plot a curve that trade-off precision and recall. It is possible to try manually different values of ***threshold***.



Issue: Could the ***threshold*** be chosen **automatically**?

2) F_1 Score (F score)

Suppose 3 different learning algorithms that have been training on a training set in which Precision and Recall values have been collected.

	Precision (P)	Recall (R)
Algorithm 1	0.5	0.4
Algorithm 2	0.7	0.1
Algorithm 3	0.02	1.0

Issue: Does the algorithm 1 with precision 0.5 and recall 0.4 does better or worse than algorithm 2 with precision 0.7 and recall 0.1? Is it possible to have only a single raw evaluation metric that enable to chosen the best algorithm instead of having 2 separate (i.e. precision and recall)?

Averaging Precision and Recall such as $\frac{P+R}{2}$ might not be a good approach. Indeed, since the algorithm 3 has a very low precision and a really high recall that will predict y=1 all the time, averaging both such as $\frac{0.02+1}{2} = 0.51$ might not help in indicating whether the threshold need to be increased or reduced. In this particular case, algorithm 1 and 2 might work better than algorithm 3, but because their respective average 0.45 and 0.4 are lower than the average in algorithm 3, algorithm 3 seems to be a better choice. Thus, averaging does not help in giving a good classifier at the end.

Solution: F_1 Score give this single raw evaluation metric that enable the trade-off between precision and recall.

F_1 Score:

$$2\frac{P * R}{P + R}$$

Because of *P*R*, either Precision or Recall are equal to 0, the F_1 Score will be equal to 0. Thus, to obtain a large F_1 Score, both Precision and Recall need to be high

P=0 or R=0 → F_1 Score=0

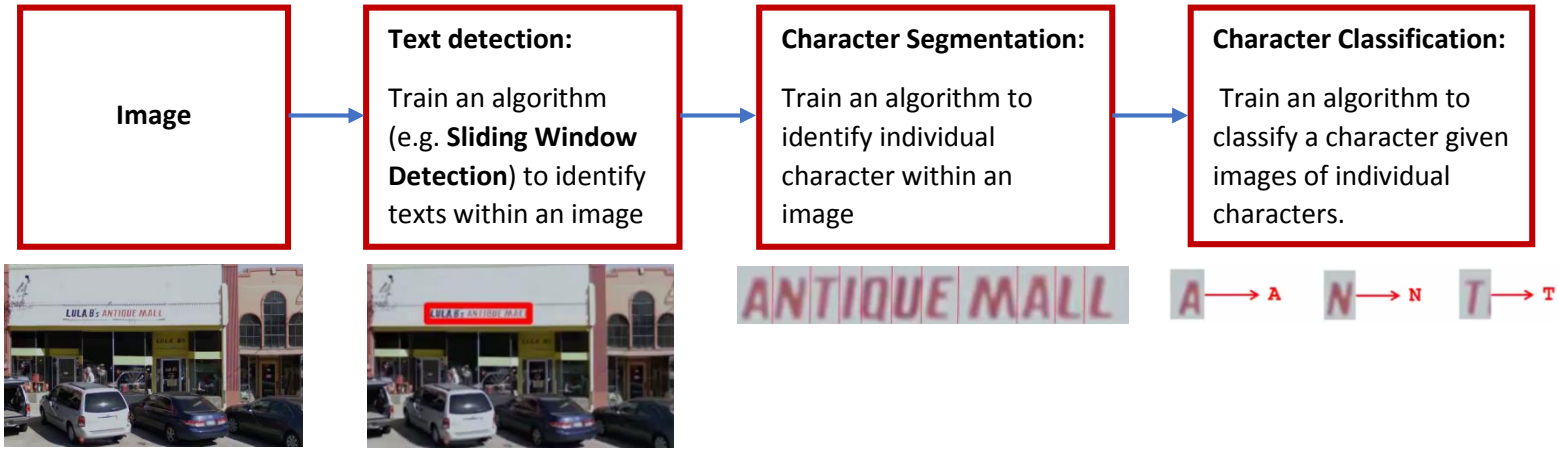
P=1 and R=1 → F_1 Score=1

	Precision (P)	Recall (R)	Average	F_1 Score
Algorithm 1	0.5	0.4	0.45	0.444
Algorithm 2	0.7	0.1	0.4	0.175
Algorithm 3	0.02	1.0	0.51	0.0392

After ranking the F_1 Score of the different algorithm computing different ***threshold***, the algorithm 1 turns out to have the highest F_1 Score. Thus, the algorithm 1 will be chosen to be train on the training set with its respective ***threshold***.

V. Ceiling Analysis

Given the example of a Photo Optical Character Recognition (OCR) pipeline:



When designing a machine learning system, one of the most important decision is to determine the performance of each of the module of the pipeline in order to identify where to dedicate more time and effort in optimizing the algorithms.

Ceiling Analysis is used to estimate the errors of each of the module of the pipeline. First, the ceiling analysis define the number evaluation metric for the overall learning system before moving to the first module of the pipeline (e.g. Text Detection) to manually provide to the algorithm the correct prediction for every single training example (e.g. provide the correct text detection output). This is in order to *simulate* a text detection system with 100% accuracy. In other words, instead of letting the learning algorithm detect the text in the images, the location of the text are manually labelled in the test set image and use these labelled text to feed in the next stages of the pipeline (e.g. Character Segmentation and Character Recognition). Evaluation Metric is then used to these late stages of the pipeline to estimate their accuracy such as

Component	**Accuracy**	
Overall system	72%	
Text Detection	89%	Increase of 17%
Character Segmentation	90%	Increase of 1%
Character Recognition	100%	Increase of 10%

When given a **perfect** text detection, the performance increase to 17% while given a **perfect** character segmentation and character recognition, the performance increase only to 1% and 10%, respectively.

Thus, time and effort should be dedicated to the module which have higher increase when having a **perfect** machine learning system is the Text Detection (17% increase), followed by Character Recognition (10% increase), and finally Character Segmentation (1% increase).

VI. Artificial Data Synthesis

One of the more reliable method to obtain high-performance machine learning system is to have a low bias algorithm and to train it on a huge training set.

Issue: Is it possible to obtain a huge training set without systematically collecting a huge amount of data ourselves or using "Crowd Source" such as Amazon Mechanical Turk?

An alternative to data collection is **artificial data synthesis.** There are two methods to synthesis data artificially:

- **Creating Data from Scratch**: Collect from the Internet random characters and paste them against different random background to create new unlimited supply of training examples.
- **Amplify a Small Dataset**: From a small dataset, select a character examples and apply Distortion, Rotation, or other Transformations to amply the dataset.