– Training Regression Models –

Anca Ralescu
EECS Department
University of Cincinnati
Cincinnati, OH, USA
Anca.Ralescu@uc.edu

1 Notation

- Data will be divided into training, validation, and test sets.
- m: size, (the number of instances) in the dataset you are measuring the RMSE (Root Mean Square Error) on.
  - For example, if you are evaluating the RMSE on a validation set of 2,000 data points then $m = 2,000$.
- $x^{(i)}$ is a vector of all the feature values (excluding the label) of the ith instance in the dataset, and $y(^{i)}$ is its label (the desired output value for that instance).
  - For example, $\mathbf{x}^{(1)} = \begin{pmatrix} -118.29 \\ 33.91 \\ 1,416 \\ 38,372 \end{pmatrix}$ and: $y^{(1)} = 156,400$
- $\mathbf{X}$ is a matrix containing all the feature values (excluding labels) of all instances in the dataset. There is one row per instance and the ith row is equal to the transpose of $\mathbf{x}^{(i)}$, noted $(\mathbf{x}^{(i)})^T$.
  - For example, the matrix $\mathbf{X}$ might be look like this:

$$\mathbf{X} = \begin{pmatrix} (\mathbf{x}^{(1)})^T \\ (\mathbf{x}^{(2)})^T \\ \vdots \\ (\mathbf{x}^{(1999)})^T \\ (\mathbf{x}^{(2000)})^T \end{pmatrix} = \begin{pmatrix} -118.29 & 33.91 & 1,416 & 38,372 \\ \vdots & \vdots & \vdots & \vdots \end{pmatrix}$$

- h is the prediction function, also called a *hypothesis*. When your system is given an instance's feature vector $\mathbf{x}^{(i)}$, it outputs a predicted value $\hat{y}^{(i)} = h(\mathbf{x}^{(i)})$

– For example, if the system predicts $158,400$, for $(\mathbf{x}^{(1)})^T$ then $\hat{y}^{(1)} = f(\mathbf{x}^{(1)}) = 158,400$. The prediction error for this data point is $\hat{y}^{(1)} - y^{(1)} = 2,000$

- $RMSE(\mathbf{X}, h)$ is the cost function measured on the set of examples using the hypothesis h. We use lowercase italic font for scalar values (such as m or $y^{(i)}$) and function names (such as h), lowercase bold font for vectors (such as $\mathbf{x}^{(i)}$), and uppercase bold font for matrices (such as $\mathbf{X}$).

The $RMSE$ is defined as

$$RMS(\mathbf{X}, h) = \sqrt{\frac{1}{m}\sum_{i=1}^{m}\left(h(\mathbf{x}^{(i)}) - y^{(i)}\right)^2}$$

where $\sqrt{\left(h(\mathbf{x}^{(i)}) - y^{(i)}\right)^2}$ is the *Euclidean norm* or l_2 norm, $\|\cdot\|_2$.

The Mean Absolute Error (MAE) is defined as

$$MAE(\mathbf{X}, h) = \frac{1}{m}\sum_{i=1}^{m}\|h(\mathbf{x}^{(i)}) - y^{(i)}\|_1$$

where $\|\cdot\|_1 = |\cdot|$ is the l_1 norm;

2 Linear Regression

Recall the simple prediction model - a linear regression model - for life satisfaction based on a country's GDP

$$life\_satisfaction = \theta_0 + \theta_1 \times GDP\_per\_capita$$

where θ_0 is called the *bias term*. If we denote by θ the vector (θ_0, θ_1), and form the data point $new\_GDP = (1, GDP\_per\_capita)$, then the above equation becomes

$$life\_satisfaction = \theta \cdot new\_GDP$$

where $\cdot$ denotes the *dot product* of two vectors.

Recall, for vectors $A = (a_1, \ldots, a_k)$ and $B = (b_1, \ldots, b_k)$, the dot product is defined as

$$A \cdot B = \sum_{i=1}^{k} a_i b_i$$

Using *matrix multiplication* $\times$, the dot product can be written as

$A \cdot B = A \times B^T$, where T denotes the transpose of the matrix

NOTE: All ML practitioners must become knowledgeable in matrix calculus.

The general *Linear Regression prediction model* is of the form

$$\hat{y} = \theta_0 + \theta_1 x_1 + \cdots + \theta_n x_n,$$

x=(x1, x2, .., xn)

where

- $\hat{y}$ is the predicted value
- n is the number of features
- x_i is the ith feature, i.e. the ith component of the data point $\mathbf{x}$, a row vector in the matrix $\mathbf{X}$
- θ_j is the jth model parameter, including the bias, θ_0; θ is a column; θ^T is a the transpose of the column vector θ.

The training data supplies the pairs $(\mathbf{x}, y)$. Training means to obtain the values of the parameter θ such that the error between $\hat{y}$ and y is minimized.

Each $\mathbf{x}$ is extended by a 1 in the 1st position: It becomes now of dimension $n + 1$

The Linear Regression in vectorized form

$$\hat{y} = h_\theta(\mathbf{x}) = \theta^T \cdot \mathbf{x}$$

Thus, MSE is given by

$$MSE(\theta) = MSE(\mathbf{X}, \theta) = \frac{1}{m} \sum_{i=1}^{m} (\theta^T \cdot \mathbf{x}^{(i)} - y^{(i)})^2$$

To find θ which minimizes MSE there is a closed from:

The Normal Equation

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

Suppose that $\mathbf{X}$ is a one dimensional data of size 100 generated randomly; and $\mathbf{y}$ is generated as $\mathbf{y} = 4 + 3 * X * \epsilon$, where ϵ is random noise. Figure 1 depicts the data set $(\mathbf{X}, \mathbf{y})$

Let $\hat{\theta}$ be the estimate obtained from the normal equation, and $\hat{\mathbf{y}}$ the corresponding estimate $\mathbf{y}$. We might get a value for $\hat{\theta} = [4.3, 2.8]$ and then we can make predictions using $\hat{\theta}$. Typically, the model predictions look as depicted in Fig 2.

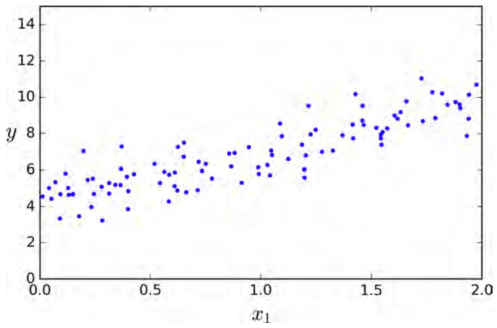


Figure 1: Randomly generated linear data set.

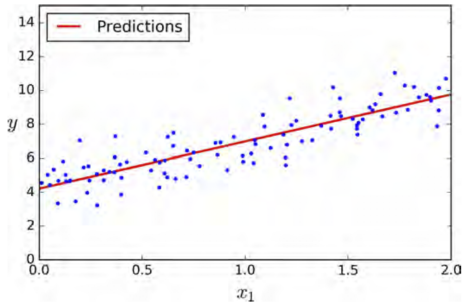


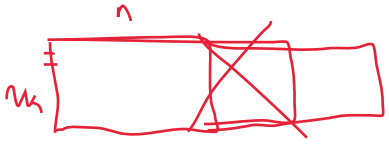
Figure 2: Linear Regression model predictions.

X: m by n
transpose(X): n by m
transpose(X) x X= (n by m) x (m by n)
n by n

Computational Complexity

The Normal Equation computes the inverse of $\mathbf{X}^T \times \mathbf{X}$, which is an $n \times n$ matrix (where n is the number of features). The computational complexity of inverting such a matrix is typically about $O(n^{2.4})$ to $O(n^3)$ (depending on the implementation). In other words, if you double the number of features, you multiply the computation time by roughly $2^{2.4} = 5.3$ to $2^3 = 8$.

- The Normal Equation gets very slow when the number of features grows large (e.g., 100,000).
- However, this equation is linear with regards to the number of instances in the training set (it is $O(m)$), so it handles large training sets efficiently, provided they can fit in memory.
- Once trained, the Linear Regression model (using the Normal Equation or any other algorithm), predictions are very fast:
  - the computational complexity is linear with regards to both the number of instances you want to make predictions on and the number of features (i.e., making predictions on twice as many instances, or twice as many features, will just take roughly twice as much time).



Other ways of training a regression model

Gradient Descent – a *generic optimization algorithm*

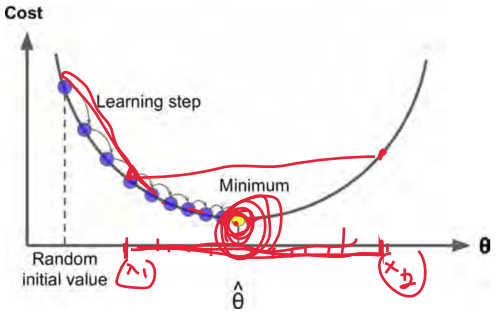
General idea:

- Tweak parameters iteratively in order to minimize a cost function.
- It measures the local gradient of the error function with regards to the parameter vector θ
- it goes in the direction of descending gradient.
- Once the gradient is zero, it reached a minimum!

Concrete steps:

1. *Random initialization:* Fill θ with random values
2. At each step decrease the cost function (e.g., the MSE), until the algorithm converges to a minimum (see Figure 3).

convex function
f[ax1+(1-a)x2]<=af(x1)+(1-a)f(x2)



convex combination
x= (alpha)(a) + (1 - alpha)(b)

0<=alpha<=1

Figure 3: Gradient Descent.

An important parameter in Gradient Descent is the size of the steps, determined by the learning rate *hyperparameter*: η.

- If η is too small, then the algorithm will have to go through many iterations to converge, which will take a long time (see Figure 4).
- If η is too large, then the algorithm will go through few iterations to converge, but may actually miss the solution (see Figure 5).

The Gradient descent works with *quadratic cost/loss* (more generally with *convex*) functions because these look like "bowls", therefore, they have a bottom, i.e., a global minimum. Fortunately, the MSE cost function for a Linear Regression model

1. is a *convex function*, which means that if you pick any two points on the curve, the line segment joining them never crosses the curve *implies* there are no local minima, just one global minimum.

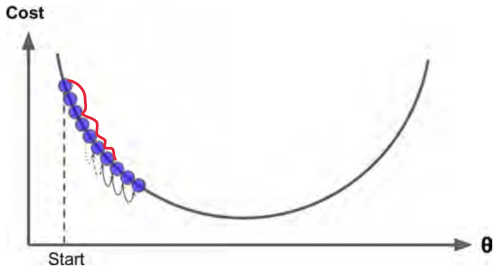


Figure 4: Learning rate too small.

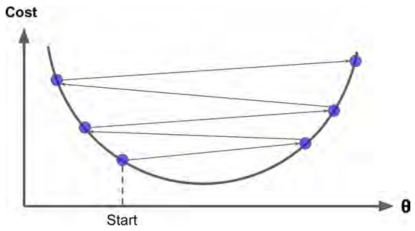


Figure 5: Learning rate too large.

2. It is also a continuous function with a slope that never changes abruptly.

These two facts have a great consequence:

The Gradient Descent is guaranteed to approach arbitrarily close the global minimum (if you wait long enough and if the learning rate is not too high)

.

The MSE cost function has the shape of a bowl, but it can be an elongated bowl if the features have very different scales (see Fig 6).

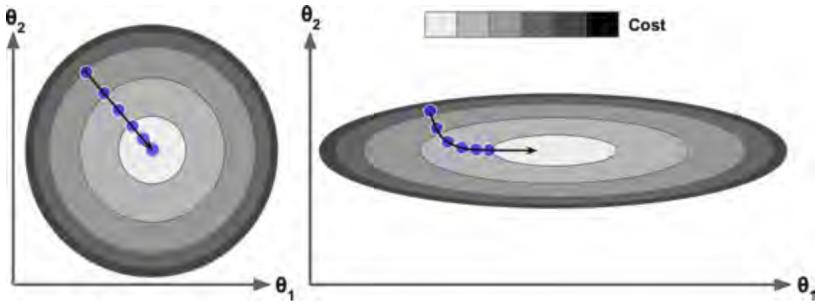


Figure 6: Gradient Descent with and without feature scaling: features 1 and 2 have the same scale (on the left); feature 1 has much smaller values than feature 2 (on the right).

But, not all cost functions look like nice regular bowls. There may be holes, ridges, plateaus, and all sorts of irregular terrains, making convergence to the minimum very difficult, illustrated in Fig. 7, where the two main challenges with Gradient Descent are shown:

- if the random initialization starts the algorithm on the left, then it will converge to a *local minimum*, which is not as good as the global minimum.
- if the random initialization starts the algorithm on the right, then it will take a very long time to cross the plateau, and if stopped too early, it will never reach the global minimum.

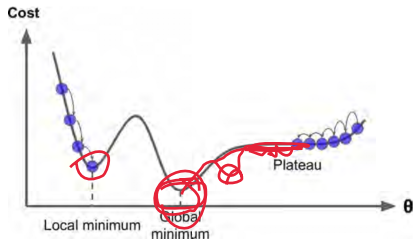


Figure 7: Gradient Descent pitfalls.

Very Important Note 1 *When using Gradient Descent, one should ensure that all features have a similar scale (e.g., using Scikit-Learn's StandardScaler class), or else it will take much longer to converge.*

Batch Gradient Descent

To implement Gradient Descent, we need to compute the gradient of the cost function with regards to each model parameter θ_j, i.e. need to calculate how much the cost function will change if θ_j changes just a little bit: partial derivative

$$\frac{\partial}{\partial \theta_j} MSE(\theta) = \frac{2}{m} \sum_{i=1}^{m} \left(\theta^T \cdot \mathbf{x}^{(i)} - y^{(i)} \right) x_j^{(i)}$$

In matrix form:

$$\nabla_\theta MSE(\theta) = \begin{pmatrix} \nabla_{\theta_0} MSE(\theta) \\ \nabla_{\theta_1} MSE(\theta) \\ \vdots \\ \nabla_{\theta_n} MSE(\theta) \end{pmatrix} = \frac{2}{m} \mathbf{X}^T \times (\mathbf{X} \times \theta - \mathbf{y})$$

- Notice that this formula involves calculations over the full training set $\mathbf{X}$, at each Gradient Descent step! Hence the algorithm is called Batch Gradient Descent: it uses the whole batch of training data at every step.

- As a result it is terribly slow on very large training sets.
- However, Gradient Descent scales well with the number of features: training a Linear Regression model when there are hundreds of thousands of features is much faster using Gradient Descent than using the Normal Equation.
- $\nabla_\theta MSE(\theta)$ points upward, so it needs to be subtracted from θ.

Gradient Descent step

$$\theta^{next\ step} = \theta - \eta \nabla_\theta MSE(\theta)$$

Figure 8 illustrates GD with various learning rates.

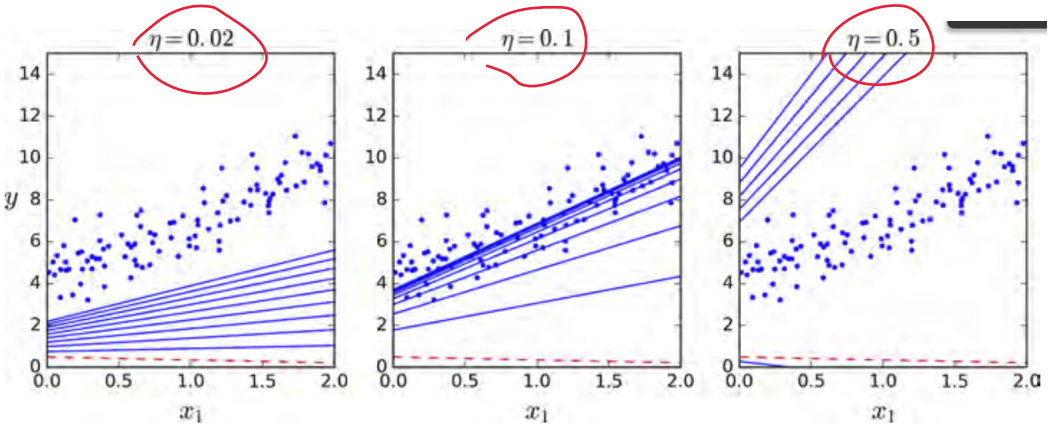
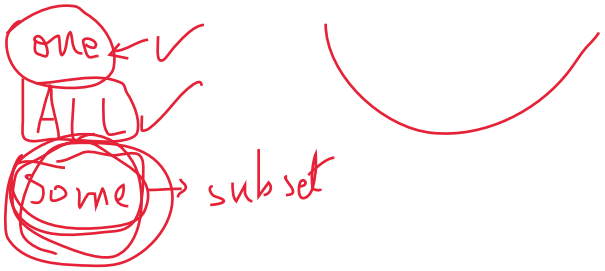
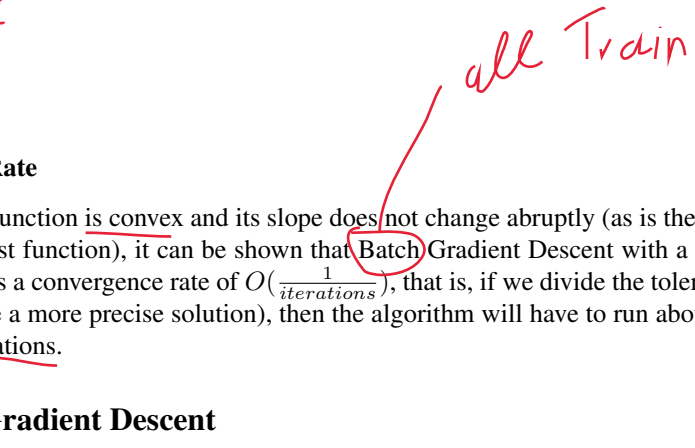


Figure 8: Gradient Descent with various learning rates: (a) On the left, the learning rate is too low: the algorithm will eventually reach the solution, but it will take a long time; (b) In the middle, the learning rate looks pretty good: in just a few iterations, it has already converged to the solution; (c) On the right, the learning rate is too high: the algorithm diverges, jumping all over the place and actually getting further and further away from the solution at every step.

Question How to set the number of iterations? If it is too low, we will still be far away from the optimal solution when the algorithm stops, but if it is too high, we will waste time while the model parameters do not change anymore.

Solution Set a very large number of iterations but to interrupt the algorithm when the gradient vector becomes tiny, i.e., that is, when its norm becomes smaller than a tiny number ϵ (called the *tolerance*), because this happens when Gradient Descent has (almost) reached the minimum.





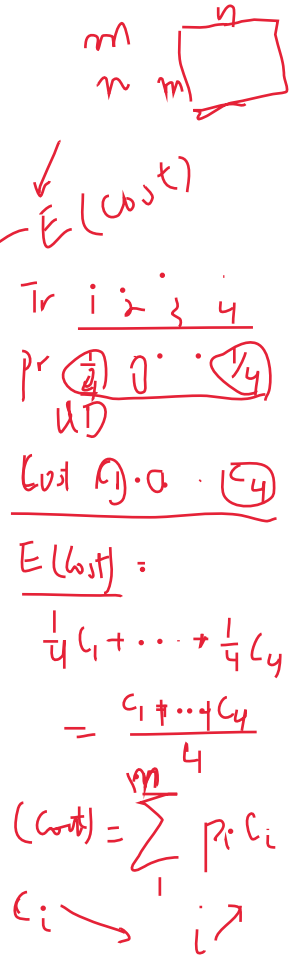
Convergence Rate

When the cost function is convex and its slope does not change abruptly (as is the case for the MSE cost function), it can be shown that Batch Gradient Descent with a fixed learning rate has a convergence rate of $O(\frac{1}{iterations})$, that is, if we divide the tolerance ϵ by 10 (to have a more precise solution), then the algorithm will have to run about 10 times more iterations.

Stochastic Gradient Descent

- The main problem with Batch Gradient Descent is the fact that it uses the whole training set to compute the gradients at every step, which makes it very slow when the training set is large.
- At the opposite extreme, Stochastic Gradient Descent just picks a random instance in the training set at every step and computes the gradients based only on that single instance.
- Obviously this makes the algorithm much faster since it has very little data to manipulate at every iteration.
- It also makes it possible to train on huge training sets, since only one instance needs to be in memory at each iteration.
- On the other hand, due to its stochastic (i.e., random) nature, this algorithm is much less regular than Batch Gradient Descent:
  - Instead of gently decreasing until it reaches the minimum, the cost function will bounce up and down, decreasing only on average.
  - Over time it will end up very close to the minimum, but once it gets there it will continue to bounce around, never settling down (as illustrated in Fig. 9).
  - So once the algorithm stops, the final parameter values are good, but not optimal.
- When the cost function is very irregular (as in Figure 7), this can actually help the algorithm jump out of local minima *implies* Stochastic Gradient Descent has a better chance of finding the global minimum than Batch Gradient Descent does.
- Therefore, randomness is good to escape from local optima, but bad because it means that the algorithm can never settle at the minimum.
- One solution to this dilemma is to gradually reduce the learning rate.
- Simulated annealing[1]

[1] It resembles the process of annealing in metallurgy where molten metal is slowly cooled down.



- The steps start out large (which helps make quick progress and escape local minima),
- Then get smaller and smaller, allowing the algorithm to settle at the global minimum.
- The function that determines the learning rate at each iteration is called the *learning schedule*.
- If the learning rate is reduced too quickly, alg may get stuck in a local minimum, or even end up frozen halfway to the minimum.
- If the learning rate is reduced too slowly, alg. may jump around the minimum for a long time and end up with a suboptimal solution if training is stopped too early.
- By convention, we iterate by rounds of m iterations; each round is called an *epoch*.
- While the Batch Gradient Descent code may iterate1,000 times through the whole training set, SGD may go through the training set only 50 times before it reaches a fairly good solution.

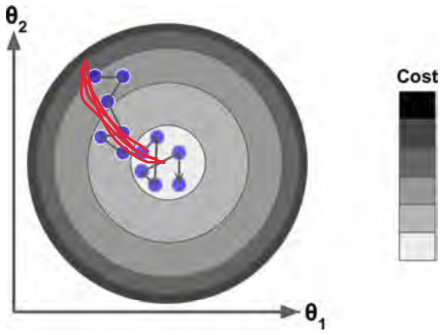


Figure 9: Stochastic Gradient Descents.

Mini-batch Gradient Descent

Combines Batch and Stochastic Gradient Descent:

- At each step, instead of computing the gradients based on the full training set (as in Batch GD) or based on just one instance (as in Stochastic GD), Mini-batch GD computes the gradients on *small random sets of instances* called *minibatches*.
- The main advantage of Mini-batch GD over Stochastic GD is that you can get a performance boost from hardware optimization of matrix operations, especially when using GPUs.
- The algorithm's progress in parameter space is less erratic than with SGD, especially with fairly large mini-batches.

- As a result, Mini-batch GD will end up walking around a bit closer to the minimum than SGD.
- But, on the other hand, it may be harder for it to escape from local minima (in the case of problems that suffer from local minima, unlike Linear Regression as we saw earlier).
- Figure 10 shows the paths taken by the three Gradient Descent algorithms in parameter space during training.
  - They all end up near the minimum, but Batch GD's path actually stops at the minimum, while both Stochastic GD and Mini-batch GD continue to walk around.
  - However, Batch GD takes a lot of time to take each step, and Stochastic GD and Mini-batch GD would also reach the minimum if a a good learning schedule was used

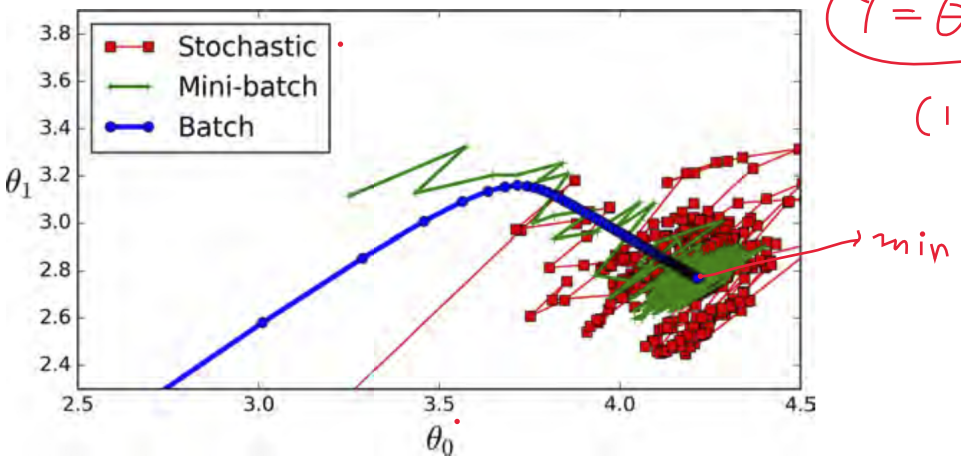


Figure 10: Gradient Descent paths in parameter space.

Table 1 summarizes the behavior of algs for linear regression.

Table 1: Comparison of algs. for Linear Regression

Alg.	Large m	Large n	Hyperparam.	Scaling?
Normal Eq	fast	slow	0	No
Batch GD	slow	fast	2	yes
Stochastic GD	fast	fast	≥ 2	yes
Minbatch GD	fast	fast	≥ 2	yes

Polynomial Regression

- Data is actually more complex than a simple straight line?
- Surprisingly, you can actually use a linear model to fit nonlinear data:
  - add powers of each feature as new features,
  - train a linear model on this extended set of features.

Figure 11 shows a data set generated by a quadratic equation.

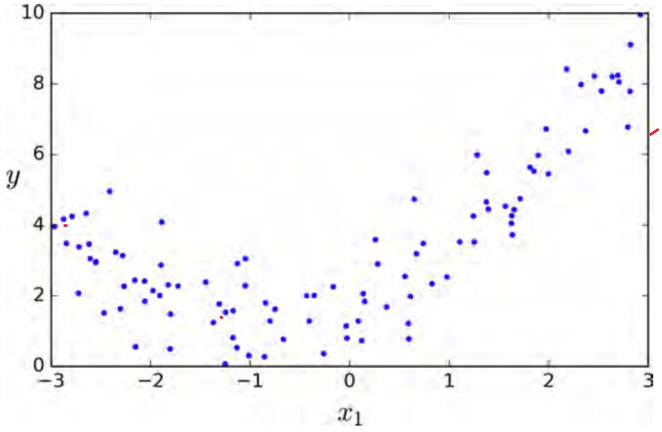


Figure 11: Generated nonlinear and noisy dataset.

Clearly, a straight line will never fit this data properly. Transform the training data, adding the square (2nd-degree polynomial) of each feature. In this case, there was only one feature, f_1, so after transformation there will be two features (f_1, f_1^2).

We now form the model

$$y(f) = \theta_0 + \theta_1 f_1 + \theta_2 f_1^2$$

And after training, the values of $(\theta_0, \theta_1, \theta_2)$ may be such that, the plot of the (x, y), is as shown in Fig 12.

Note: PolynomialFeatures(degree=d) transform an array containing n features into an array containing $\frac{(n+d)!}{d!n!}$ features. Therefore, we must be aware of the combinatorial explosion of the number of features!

Learning Curves

If we perform high-degree Polynomial Regression, we will likely fit the training data much better than with plain Linear Regression. For example, Figure 13 applies a 300-degree polynomial model to the preceding training data, and compares the result with a pure linear model and a quadratic model (2nd-degree polynomial). Notice how the 300-degree polynomial model wiggles around to get as close as possible to the training instances.

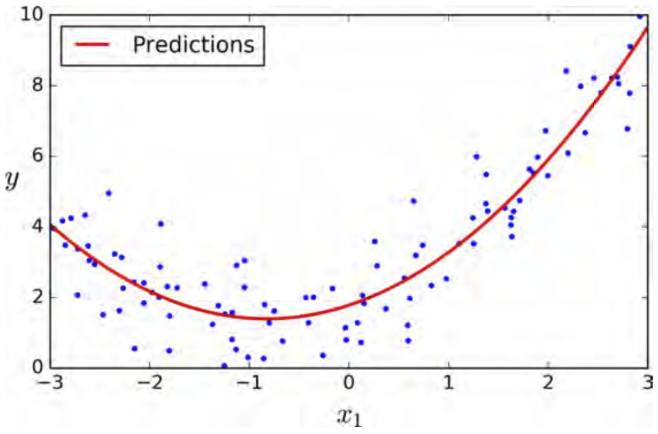


Figure 12: Polynomial Regression model predictions.

- The high-degree Polynomial Regression model is severely overfitting the training data, while the linear model is underfitting it.
- The model that will generalize best in this case is the quadratic model. It makes sense since the data was generated using a quadratic model
- In general you won't know what function generated the data, so how can you decide how complex your model should be?
- How can you tell that your model is overfitting or underfitting the data:
  - Use cross validation
  - Use learning curves: plots of the model's performance on the training set and the validation set as a function of the training set size. To generate the plots, simply train the model several times on different sized subsets of the training set.

Figure 14 illustrates learning curves:

- When there are just one or two instances in the training set, the model can fit them perfectly, which is why the curve starts at zero.
- As new instances are added to the training set, it becomes impossible for the model to fit the training data perfectly, both because the data is noisy and because it is not linear at all.
- So, the error on the training data goes up until it reaches a plateau, at which point adding new instances to the training set doesn't make the average error much better or worse.

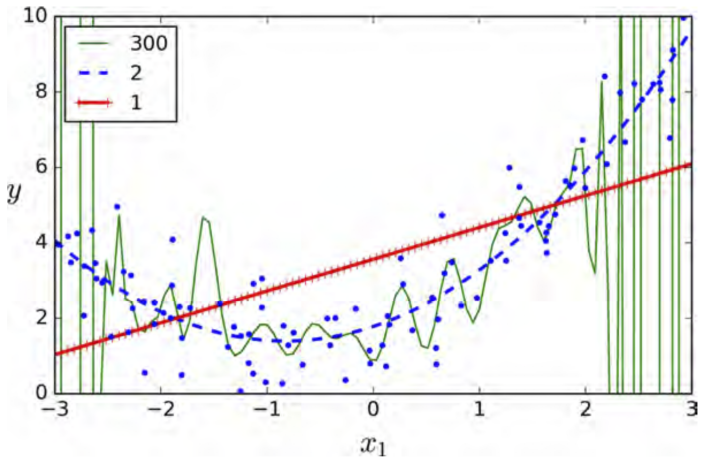


Figure 13: High-degree Polynomial Regression.

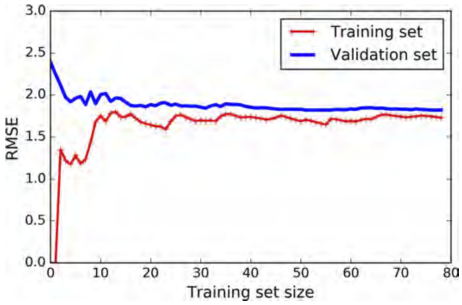


Figure 14: Learning curves.

- On the validation data, when the model is trained on very few training instances, it is incapable of generalizing properly, which is why the validation error is initially quite big.
- Then as the model is shown more training examples, it learns and thus the validation error slowly goes down.
- However, once again a straight line cannot do a good job modeling the data, so the error ends up at a plateau, very close to the other curve.
- These learning curves are typical of an under-fitting model: both curves have reached a plateau; they are close and fairly high.

Note: If model is under-fitting the training data, adding more training examples will not help. We need to use a more complex model or come up with better features.

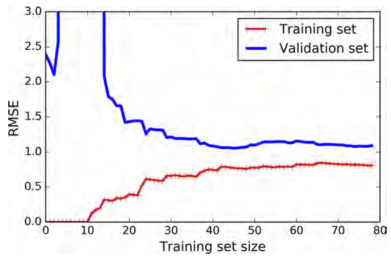


Figure 15: Learning curves for the 10Th degree polynomial model.

Figure 15 shows the learning curves for the 10Th degree polynomial model.
Note that:

- The error on the training data is much lower than with the Linear Regression model.
- There is a gap between the curves: the model performs significantly better on the training data than on the validation data, hence overfitting
- However, with a much larger training set, the two curves would continue to get closer.

Bias versus Variance

The generalization power of a model is based on:

Bias

The part of the generalization error due to wrong selection of the model, e.g., assuming linear when it is actually quadratic. **High-bias model contributes to under-fitting the training data.**

Variance

The part of the error due to the model's excessive sensitivity to small variations in the training data. A model with many degrees of freedom (such as a high-degree polynomial model) is likely to have **high variance, and thus to overfit the training data.**

Irreducible Error

This part of the error is due to the noisiness of the data itself. Must clean up the data: fix the data sources, such as broken sensors, or detect and remove outliers.

Increased complexity $\Longrightarrow$ variance increases and bias decreases.

(12, 3), 1.5
(12, 3) , 3.2

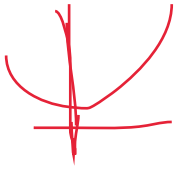
(12, 3) class 1
(12, 3) class 2

classes are mutually exclusive - no overlap

Regularization

Three approaches:

- Ridge Regression
- Lasso Regression
- Elastic Net



Ridge Regression

Add $\alpha \sum_{i=1}^{n} \theta_i^2$. Thus the cost function becomes

$$J(\theta) = MSE(\theta) + \alpha \sum_{i=1}^{n+1} \theta_i^2$$

Figure 16 shows several Ridge models trained on some linear data using different α value.

On the left, plain Ridge models are used, leading to linear predictions.

On the right, the data is first expanded using PolynomialFeatures(degree=10), then it is scaled, and finally the Ridge models are applied to the resulting features: this is *Polynomial Regression with Ridge regularization*.

Increasing α leads to flatter (i.e., less extreme, more reasonable) predictions; this reduces the model's variance but increases its bias.

Closed form equation for Ridge Regression

$$\widehat{\theta} = (\mathbf{X}^T\mathbf{X} + \alpha\mathbf{A})^{-1}\mathbf{X}^T\mathbf{y},$$

where $\mathbf{A}$ is the $(n+1) \times (n+1)$ *identity matrix*, except that $\mathbf{A}(1,1) = 0$

Lasso Regression: Least Absolute Shrinkage and Selection Operator Regression

Cost function

$$J(\theta) = MSE(\theta) + \alpha \sum_{i=1}^{n} |\theta_i|$$

Taking into account that

$$|x| = \begin{cases} -1 & \text{if} \quad x < 0 \\ 0 & \text{if} \quad x = 0 \\ +1 & \text{if} \quad x > 0 \end{cases}$$

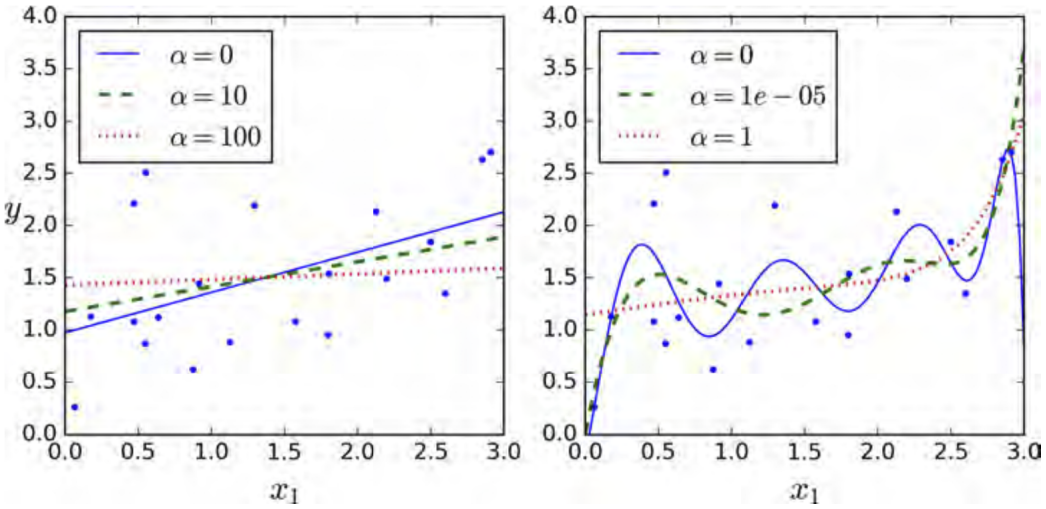


Figure 16: Ridge Regression with regularization: linear model on the left; polynomial model on the right;

We can rewrite the LASSO cost function as

$$J(\theta) = MSE(\theta) - \alpha \sum_{\mathbf{i=1}, \theta_i < 0}^{n} \theta_i + \alpha \sum_{\mathbf{i=1}, \theta_i > 0}^{n} \theta_i$$

The Lasso cost function is not differentiable at $\theta_i = 0$ (for $i = 1, 2, \ldots, n$). Gradient Descent still works fine if you use a *subgradient* vector g instead when any $\theta_i = 0$.

$$g(\theta, J) = \nabla_\theta MSE(\theta) + \alpha \sum_{\mathbf{i=1}}^{n} sign(\theta_i) = \nabla_\theta MSE(\theta) + \alpha(N_+ - N_-) = \nabla_\theta MSE(\theta) + \alpha(2N_+ - n),$$

where N_+, N_- denote the number of $\theta > 0$ and $\theta < 0$ respectively, and n the dimension of θ. Figure 17 illustrates Lasso with various levels of regularization - left linear; right polynomial.

Lasso Regression tends to completely eliminate the weights of the least important features (i.e., set them to zero).

For example, the dashed line in the right plot in Figure 17 ($\alpha = 10^{-7}$) looks quadratic, almost linear: all the weights for the high-degree polynomial features are equal to zero.

That is, Lasso Regression automatically performs feature selection and outputs a sparse model (i.e., with few nonzero feature weights).

may not be correct

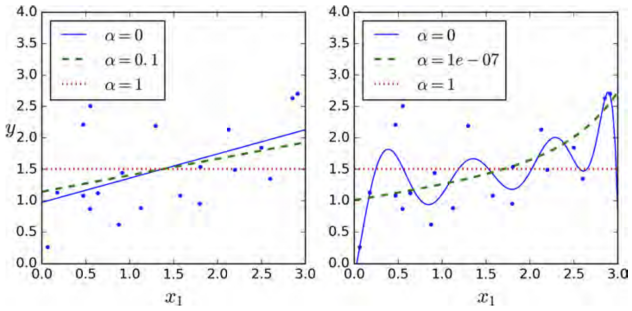


Figure 17: Lasso Regression.

Elastic Net

Middle ground between Ridge Regression and Lasso Regression

$$J(\theta) = MSE(\theta) + r\alpha \sum_{i=1}^{n} |\theta_i| + \frac{1-r}{2}\alpha \sum_{i=1}^{n} \theta_i^2$$

Choosing the regression

- Avoid plain regression
- Prefer Lasso or Elastic Net when one suspects that only some features are useful
- Prefer Elastic to Lasso when $n > m$ or when features are strongly correlated

Early Stop ("beautiful free lunch")

Stop when the validation error reaches a minimum. Figure 18 shows a complex model (in this case a high-degree Polynomial Regression model) being trained using Batch Gradient Descent.

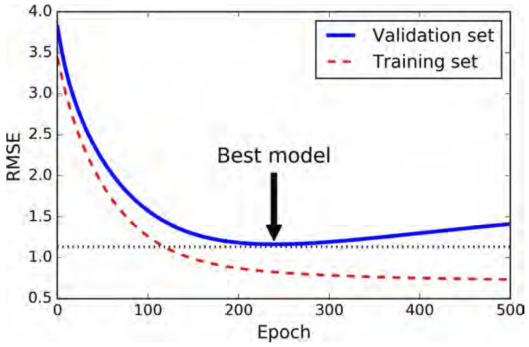


Figure 18: Early stopping regularization.

As the algorithm learns and its prediction error (RMSE) on the training set naturally goes down, and so does its prediction error on the validation set.

However, after a while the validation error stops decreasing and actually starts to go back up: the model has started to overfit the training data.

Logistic Regression

Used to estimate probabilities.

$$\hat{p} = h_\theta(\mathbf{x}) = \sigma(\mathbf{x}^T\theta),$$

where

$$\sigma(t) = \frac{1}{1 + exp^{(-t)}}$$

shown in Fig 19.

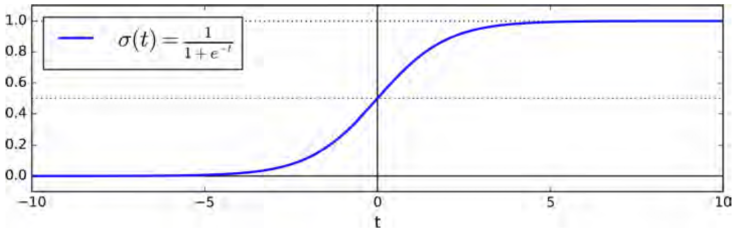


Figure 19: The logistic function.

The model prediction is based on $\hat{p}$

$$\hat{y} = \begin{cases} 0 & \text{if } \hat{p} < 0.5 \\ 1 & \text{if } \hat{p} \geq 0.5 \end{cases}$$

therefore $\hat{y} = 1$ when $\mathbf{x}^T\theta \geq 0$ and 0 if $\mathbf{x}^T\theta < 0$.

Training the logistic regression model

The cost function for a single instance is given by

$$c(\theta) = \begin{cases} -\log \hat{p} & \text{if } y = 1 \\ -\log(1 - \hat{p}) & \text{if } y = 0 \end{cases}$$

We can write $c(\theta)$ in one equation as follows:

$$c(\theta) = -y \log \hat{p} + (1 - y) \log(1 - \hat{p})$$

The cost function over all m training instances is then

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^{m} \left[y^{(i)} \log \hat{p}^{(i)} + (1 - y^{(i)}) \log(1 - \hat{p}^{(i)}) \right]$$

No closed form solution (no normal equation).

Apply Gradient Descent, which works because $J(\theta)$ is convex.

$$\frac{\partial}{\partial \theta_j} J(\theta) = \sum_{i=1}^{m} \left(\sigma(\theta^T \mathbf{x}^{(i)}) - y^{(i)} \right) x_j^{(i)}$$

Once we compute $\nabla J(\theta)$ we can apply Batch/Stochastic/MiniB Gradient Descent.

Decision Boundaries

Use the Iris Data set shown in Figure 20 Training a logistic regression model on the

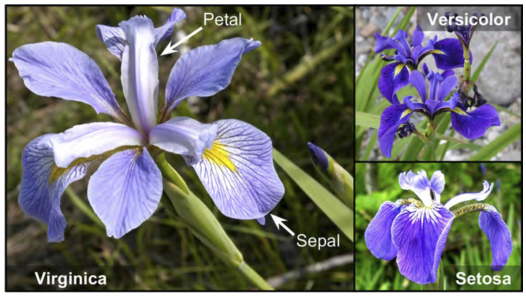


Figure 20: Flowers of three iris flowers.

petal width, yields the decision boundary shown Fig. 21.

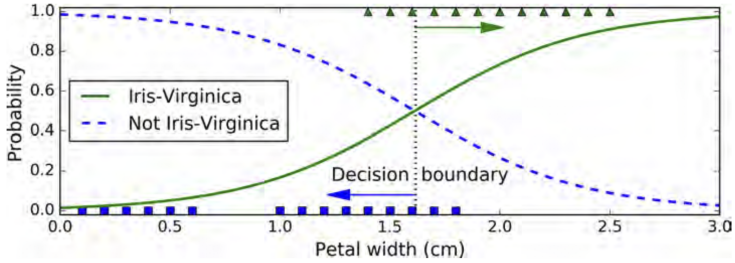


Figure 21: Estimated probabilities and decision boundary based on the petal width.

- The petal width of *Iris-Virginica flowers* are represented by triangles: its range is $[1.4, 2.5]$ cm.
- Other iris flowers petal width are represented by squares: generally, they have a smaller petal width: $[0.1, 1.8]$ cm
- Overlap: $\approx$ 2cm

- Above about 2 cm the classifier is highly confident that the flower is an Iris-Virginica
- Below 1 cm it is highly confident that it is not an Iris-Virginica
- In between these extremes, the classifier is unsure:
  - it predicts the class – whichever class is the most likely.
- Therefore, there is a decision boundary at around 1.6 cm where both probabilities are equal to 0.5:
  - if the petal width is higher than 1.6 cm, the classifier will predict that the flower is an Iris- Virginica,
  - else it will predict that it is not (even if it is not very confident)

Figure 22 shows the same data in the petalLength - petalWidth space. Once trained,

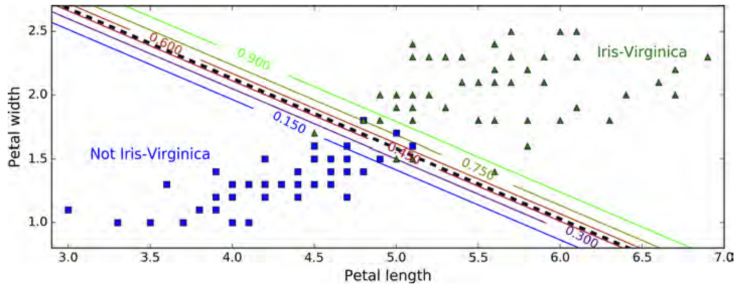


Figure 22: Linear Decision Boundary: dashed line estimates 0.5 probability; bottom left, 0.15, top right 0.90.

the Logistic Regression classifier can estimate the probability that a new flower is an *Iris virginica*. Logistic regression can be regularized.

Softmax Regression / Multinomial Logistic Regression

Generlize the Logistic Regression to support multiple classes directly.

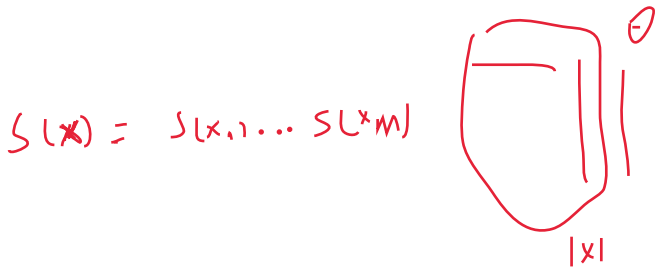
When given an instance x, the Softmax Regression model first computes a score $s_k(x)$ for each class k, then estimates the probability of each class by applying the *softmax* function (also called the normalized exponential) to the scores.

$$s_k(\mathbf{x}) = \mathbf{x}^T \theta^{(k)}$$

where $\theta^{(k)}$ is the parameter vector for class k.
The matrix $\mathbf{\Theta}$ holds all the vectors $\theta^{(k)}$.
The softmax function for class k:

$$\hat{p}_k = \sigma(s(\mathbf{x}))_k = \frac{exp(s_k(\mathbf{x}))}{\sum_{j=1}^{K} exp(s_j(\mathbf{x}))},$$

where



- K is the number of classes
- s(x) is a vector containing the scores for each class for instance $\mathbf{x}$
- $\sigma(\mathbf{s}(\mathbf{x}))_k$ is the estimated probability that $\mathbf{x}$ belongs to class k, given the scores for each class

The softmax regression classifier prediction is

$$\hat{y} = \arg\max_k \sigma(\mathbf{s}(\mathbf{x}))_k = \arg\max_k s_k(\mathbf{x}) = \arg\max_k \left(\left(\theta^{(k)}\right)^T \cdot \mathbf{x} \right)$$

Training the Softmax

Use the cross entropy

$$J(\mathbf{\Theta}) = -\frac{1}{m} \sum_{i=1}^{m} \sum_{k=1}^{K} y_k^{(i)} \log\left(\hat{p}_k^{(i)}\right),$$

where

- $y_k^{(i)}$ is equal to 1 if the target class for the ith instance is k; otherwise, it is equal to 0.

The gradient for the cross entropy for class k is given by

$$\nabla_{\theta^{(k)}} J(\mathbf{\Theta}) = \frac{1}{m} \sum_{i=1}^{m} \left(\hat{p}_k^{(i)} - y_k^{(i)}\right) \mathbf{x}^{(i)}$$