MACH

Training models

EPITA

So far ...

- We have treated ML models and their training algorithms mostly like black bloxes.
- You may have been surprised (and frustrated) by how much you can get done without knowing anything about what is under the hood.
  - You optimized a regression system
  - You improve a digit classifier
  - You even built a spam classifier from scratch
  - All without knowing how they actually work.
- Implementation details are not necessary in many situations.
- However, having a good understanding of how things work can help you quickly home in on:
  - The appropriate model
  - The right training algorithm to use
  - A good set of hyperparameters
- And also:
  - Debug issues and perform error analysis efficiently

Roadmap

- Start with looking at the linear regression model (one of the simplest model)
- Discuss two very different ways to train it:
  - Using a closed-form equation that directly computes the model parameters that best fit the model to the training set.
  - Using an iterative optimization approach called gradient descent (GD) that gradually tweaks the model parameters to minimize the cost function on the training set (eventually converging as the same parameters as the first method).
    - Few variants of GD: batch GD, mini-batch GD, stochastic GD
      - Used in Neural Networks
- Polynomial regression
- Regularization techniques
- Two models commonly used for classification tasks:
  - Logistic regression
  - Softmax regression

Warning: notions of linear algebra and calculus are required.
You need to know:

- Vectors, Matrices
- How to transpose them, multiply them, and inverse them
- What partial derivatives are.

If worry: go back to the tutorial.

Linear Regression

Enter in the game …

Linear Regression

- A linear model makes a prediction by simply computing a weighted sum of the input features, plus a constant called the **bias term** (aka the intercept term)

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots + \theta_n x_n$$

    - $\hat{y}$ is the predicted value
    - n is the number of features
    - x_i is the ith feature
    - θ_j is the jth parameter including the bias term θ_0 and the weights $\theta_1, \theta_2, \ldots, \theta_n$
- This can be written much more concisely …

Vectorized form

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots + \theta_n x_n$$

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

- h_θ is the **hypothesis function** using the **model parameters $\boldsymbol{\theta}$**
- $\boldsymbol{\theta}$ **is the model parameter vector, containing the bias term** θ_0 and the feature weights θ_1 to θ_n
- **x** is the instances' **feature vector**, containing x_0 to x_n with x_0 always equal to 1.
- $\boldsymbol{\theta} \cdot \boldsymbol{x}$ is the **dot product** of the vectors $\boldsymbol{\theta}$ and $\boldsymbol{x}$ which is equal to $\sum_{i=0}^{n} \theta_i x_i$

Matrix Multiplication

- In Machine Learning:
  - Vector are often represented as column vector (2D arrays with a single column)
  - If $\boldsymbol{\theta}$ and $\boldsymbol{x}$ are column vectors, then the prediction is

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

  - Where $\boldsymbol{\theta}^{\boldsymbol{T}}$ is the transpose of $\boldsymbol{\theta}$ (a row vector instead of a column vector)
  - $\boldsymbol{\theta}^{\boldsymbol{T}} \cdot \boldsymbol{X}$ is the matrix multiplication of $\boldsymbol{\theta}^{\boldsymbol{T}}$ and $\boldsymbol{X}$
- The same prediction, except it is now represented as a single-cell matrix rather than a scalar value.

➔ We will use this notation.

How do we train the regression model ?

- Training a model ←→ setting its parameters so that the model best fits the training set.

➔ We need a measure of how well (or poorly) the model fits the training data.

➔ RMSE the most common used

- Train a linear model ➔ find the $\boldsymbol{\theta}$ values that minimize the RMSE
- In practice, it is simpler to minimize mean square error (MSE) than the RMSE and it leads to the same result (the value that minimizes a positive function also minimize its square root).

- Learning algorithms will often **optimize** a different **loss function** during training than the **performance measure** used to evaluate the final model.
- Generally because the function is **easier to optimize** and/or because it has **extra terms needed during training** only (e.g., for regularization).
- A good **performance metric** is as close as possible to the final **business objective**
- A good training **loss** is **easy** to **optimize** and **strongly correlated** with the metric.
- Classifier are often training using a cost function such as the log loss but evaluated by precision/recall. The log loss is easy to optimize and doing so will usually improve precision/recall.

MSE

MSE cost function for a linear regression model:

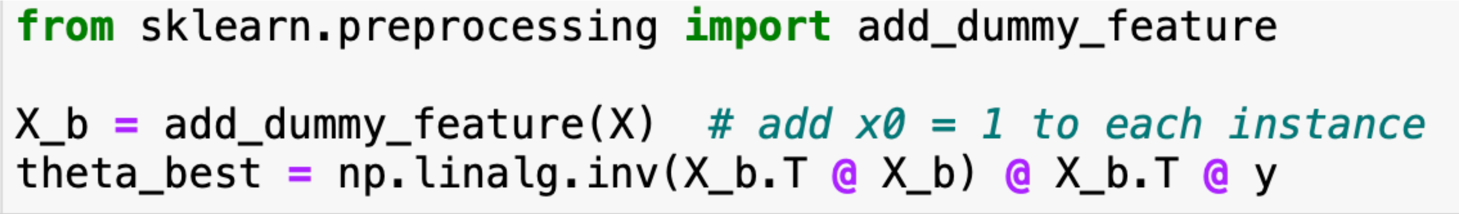
$$MSE(\mathbf{X}, h_\theta) = \frac{1}{m}\sum_{i=1}^{m}\left(\theta^T\mathbf{x}^{(i)} - y^{(i)}\right)^2$$

The Normal Equation

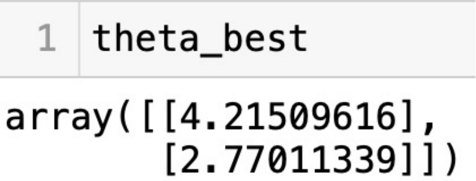
- To find the value of $\boldsymbol{\theta}$ that minimizes the MSE, there exists a *closed-form solution* – a mathematical equation that gives the result directly. This is called the Normal Equation.

$$\hat{\theta} = \left(\mathbf{X}^\top \mathbf{X}\right)^{-1} \mathbf{X}^\top \mathbf{y}$$

- $\hat{\boldsymbol{\theta}}$ is the value of $\boldsymbol{\theta}$ that minimizes the cost function
- **y** is the vector of target values containing $y^{(1)}$ to $y^{(m)}$
- $(X^\top X)^{-1}X^\top$ is the pseudo-inverse of X (to be inversed X must a be square matrix)

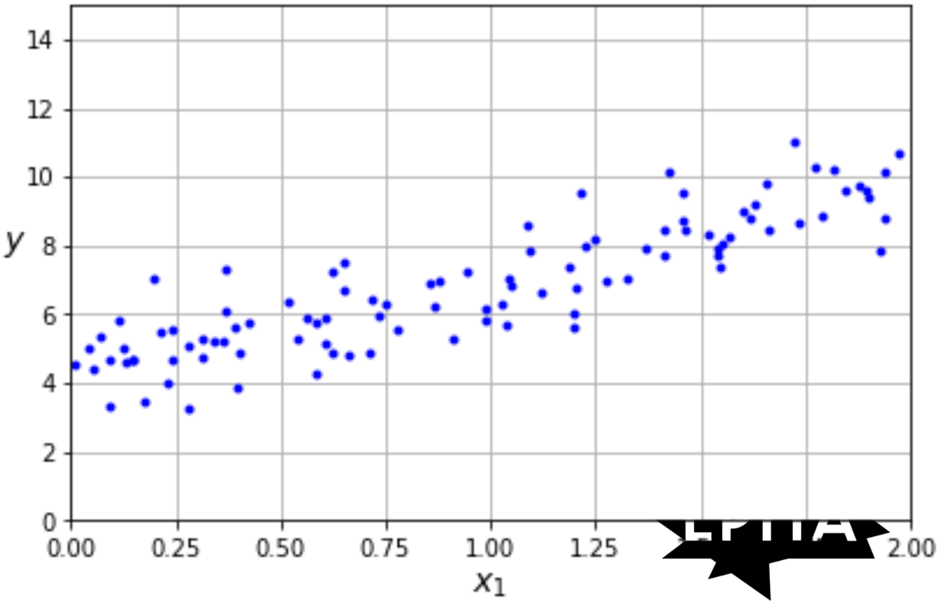


```python
from sklearn.preprocessing import add_dummy_feature

X_b = add_dummy_feature(X)  # add x0 = 1 to each instance
theta_best = np.linalg.inv(X_b.T @ X_b) @ X_b.T @ y
```



```python
theta_best
```

```
array([[4.21509616],
       [2.77011339]])
```



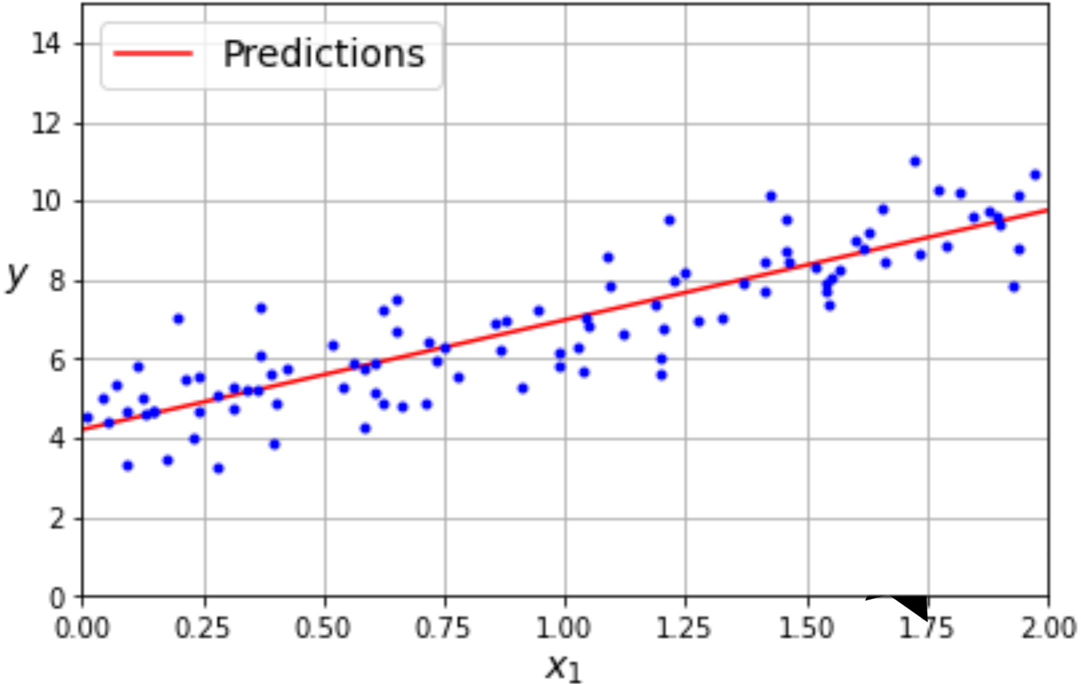
- inv() function from NumPy linear algebra module np.linalg to compute the inverse of a matrix.
- @ performs matrix multiplication
- If A and B are NumPy arrays, then A@B is equivalent to nb.matmul(A,B).
- Many other libraries (TF, Pytorch,..) support @ operator
- However you cannot use @ on pure Python arrays (lists of lists)

- The function we used to generate the data is y=4 +3x + Gaussian noise.
  - ([[4.21509616], [2.77011339]])
  - Not too bad: the Gaussian noise made it impossible to retrieve the exact parameters of the original function.
- The smaller and noisier the dataset, the harder it gets!

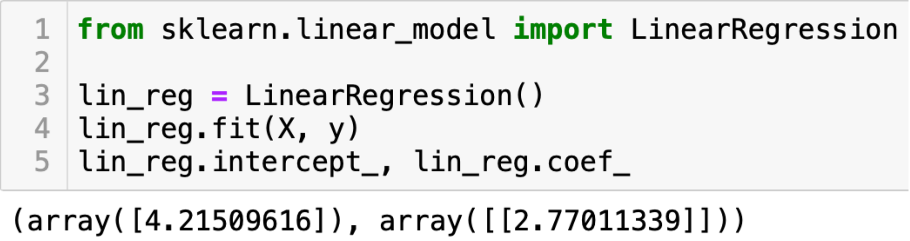
Making predictions using $\boldsymbol{\theta}$

```python
X_new = np.array([[0], [2]])
X_new_b = add_dummy_feature(X_new)  # add x0 = 1 to each instance
y_predict = X_new_b @ theta_best
y_predict
```

```
array([[4.21509616],
       [9.75532293]])
```

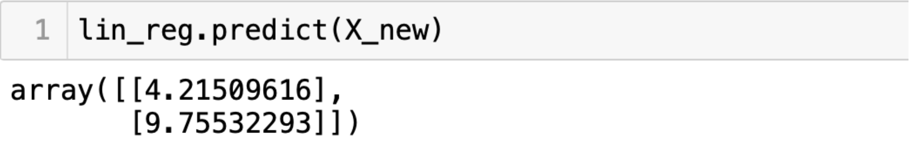


Performing linear regression using Scikit-Learn



```
from sklearn.linear_model import LinearRegression

lin_reg = LinearRegression()
lin_reg.fit(X, y)
lin_reg.intercept_, lin_reg.coef_
```

```
(array([4.21509616]), array([[2.77011339]]))
```



```
lin_reg.predict(X_new)
```

```
array([[4.21509616],
       [9.75532293]])
```

- Notice that Scikit-Learn separates the bias term (intercept_) from the feature weights (coef_).
- The LinearRegression is based on the **scipy.linalg.lstsq()** function (least squares) which can be called directly:

theta_best_svd, residuals, rank, s = np.linalg.lstsq(X_b, y, rcond=1e-6)

This function computes $\widehat{\boldsymbol{\theta}} = \boldsymbol{X}^{+}\boldsymbol{y}$ where $\boldsymbol{X}^{+}$ is the pseudo-inverse of **X** (the Moore-Penrose inverse)

You can use **np.linalg.pinv()** to compute the pseudo-inverse directly:

np.linalg.pinv(X_b) @ y

- The pseudo-inverse is computed using a standard matrix factorization technique called Singular Value Decomposition (SVD)
  - $\boldsymbol{X}\ \boldsymbol{decomposed\ in}\ \boldsymbol{U\Sigma}\mathbf{V}^{\top}$ then $\boldsymbol{X}^{+} = \mathbf{V\Sigma}^{+}\mathbf{U}^{\top}$
  - To compute $\boldsymbol{\Sigma}^{+}$: take $\boldsymbol{\Sigma}$ and set to 0 all values smaller than a tiny threshold value, then replace all the non zero values with their inverse, and finally transpose the resulting matrix.
  - More efficient than computing the Normal equation, plus it handles edge cases nicely:
    - Normal Equation may not work if the matrix $\boldsymbol{X}^{\top}\boldsymbol{X}$ is not invertible (if m<n or if some features are redundant)
    - The pseudo inverse is always defined.

Computational Complexity

- The Normal Equation computes the inverse of $X^\top X$
  - $X^\top X$ is an (n+1)x(n+1) matrix (where n is the number of features)
- Computational complexity of inverting such a matrix is about $O(n^{2.4})$ to $O(n^3)$ depending the implementation.
- SVD approach is about $O(n^2)$
- Both Normal Equation and SVD get very slow when the number of features grows large (e.g., >100,000)
- They behave linearly w.r.t. the number of instances in the training set (O(m))
  - They handle large low-dimensional datasets (provided they can fit in memory).
- Prediction is very fast!
  - Linear with #instances and #features.

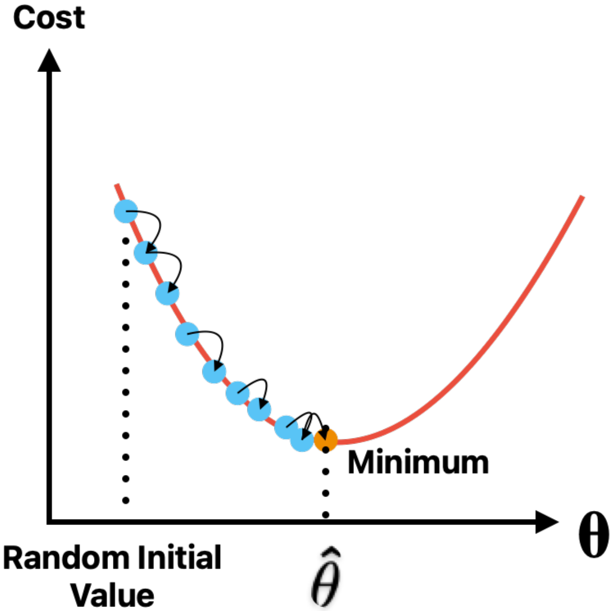
➔ Look at a different and scalable way to train a linear regression model

Gradient Descent

A generic optimization algorithm capable of finding optimal solutions to a wide range of problems.

General idea

- Tweak the parameters iteratively to minimize a cost function.
- Gradient descent in a nutshell:
  - Measures the local gradient of the error function with regard to the parameter vector $\boldsymbol{\theta}$
  - It goes in the direction of the descending gradient
  - Once the gradient is zero, one has reached **a** minimum
- In Practice:
  - Start by filling $\boldsymbol{\theta}$ with random values (aka random initialization)
  - Then improve it gradually, taking one baby step at a time
    - Each step attempting to decrease the cost function (e.g., MSE)
  - Until the algorithm converges to a minimum



Pseudo-code

(i) Start with an initial guess for θ;

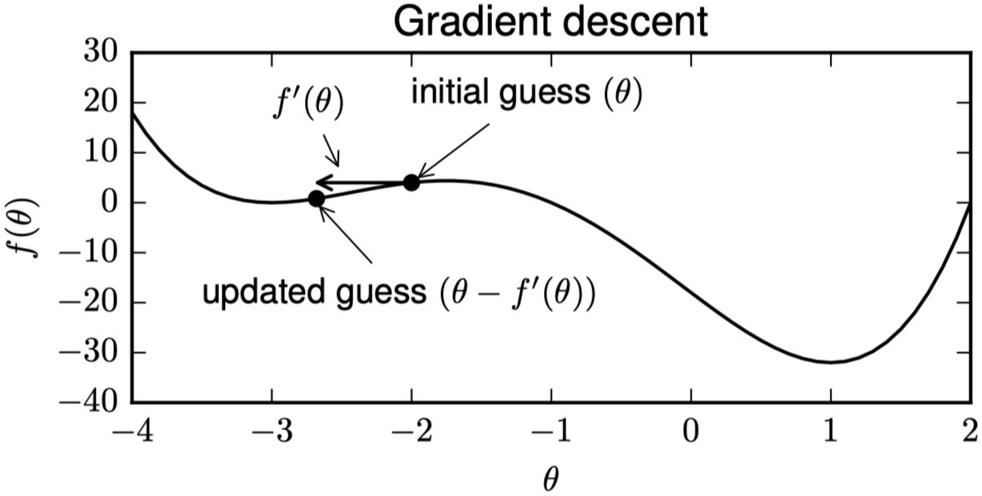
(ii) Compute the derivative (∂θ/∂) f(θ). Here f(θ) is the MSE (or whatever criterion we are optimizing) under our model θ.

(iii) Update our estimate of

$$\theta := \theta - \alpha \cdot f'(\theta);$$

/!\ Simultaneous update

(iv) Repeat Steps (ii) and (iii) until convergence.



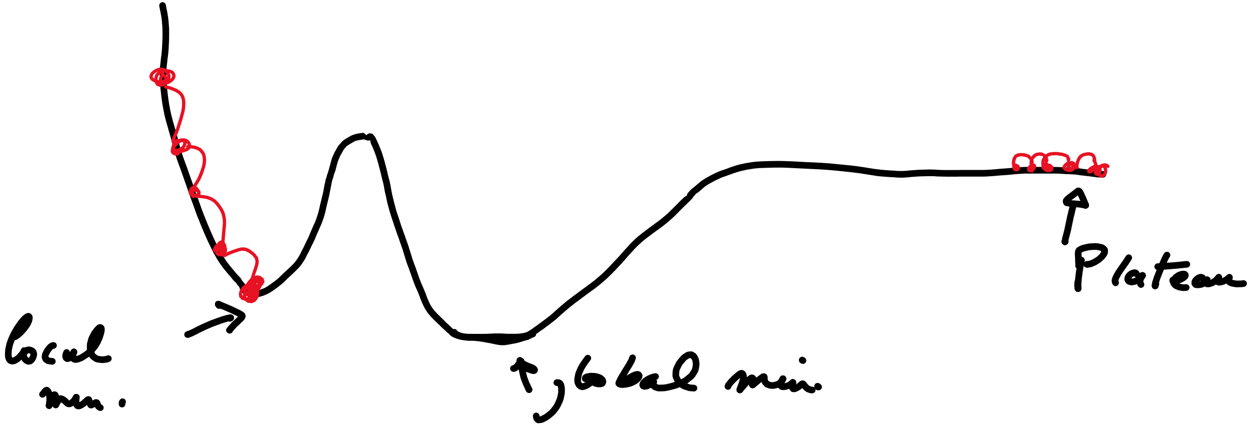
α : learning rate

Learning rate

- An important parameter in gradient descent is the **size of the steps**
  - Determined by the **learning rate** hyperparameter
- If the learning rate is too small:
  - Too many iterations (and time) to converge
- If the learning rate is too high:
  - Jump accross the valley
  - High risk to diverge

The importance of the cost function

- Remember this algorith is generic
- Not all cost functions look nice and regular bowls.
  - Holes, plateau,
  - Risk to reach a local minimum instead of the global one.



MSE cost function

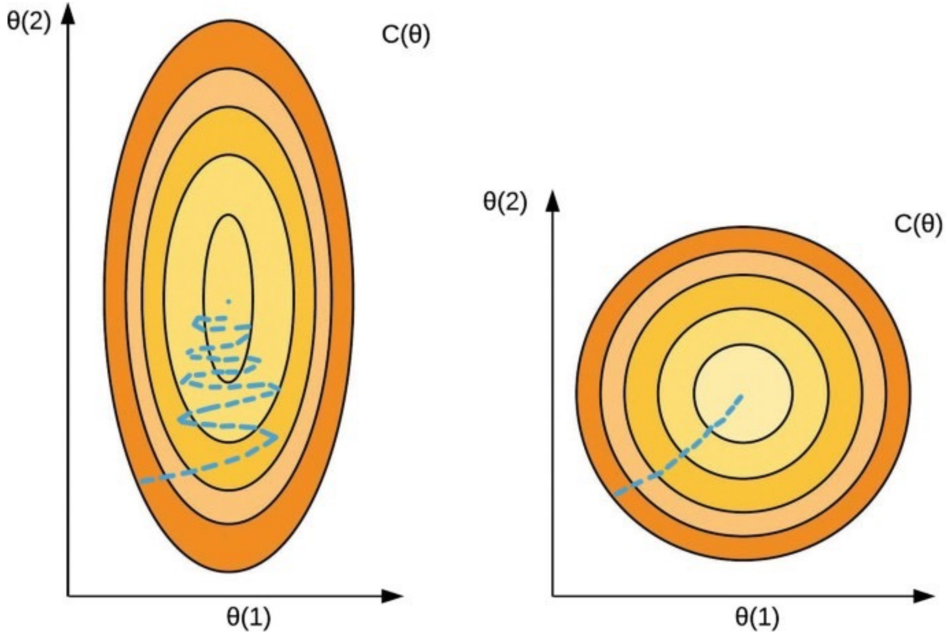
- Convex function
    - If you pick any two points on the curve, the line segment joining them is never below the curve?
    - ➔ There is no local minimum, just a global minimum.
- Continuous function with a slope that never changes abruptly (its derivative is Lipschitz continuous).

➔ Great consequence: gradient descent is guaranteed to approach arbitrarly closely the global minimum (if you wait long enough and if the learning rate is not too high).

The importance of feature scaling

- While the cost function has the form of a bowl, it can be an elongated bowl if the feature have very different scales.
- The gradient descent algorithm goes straight toward the minimum – reaching it quickly with feature scaling
- Without feature scaling:
  - Long march down an almost flat valley
  - Eventually reach the minimum but in a longer time

➔ When Using gradient descent, ensure that all features have a similar scale!



Batch Gradient Descent

- To implement a Gradient Descent:
  - Need to compute the gradient of the cost function w.r.t. each model parameter θ_j
  - ⇔ calculate how much the cost function will change if you change θ_j just a little bit.
    - Called the partial derivative
      - Like asking *«what is the slope of the mountain under my feet if I face east»*? And then asking the same for North …
- Partial derivatives of the cost function:

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

- Instead of computing these derivatives individually, compute them all in one go. The gradient vector contains all the partial derivatives of the cost function

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

Batch Gradient Descent (cn't)

- Once you have the gradient vector, which points uphill, just go in the opposite direction to go downhill.

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

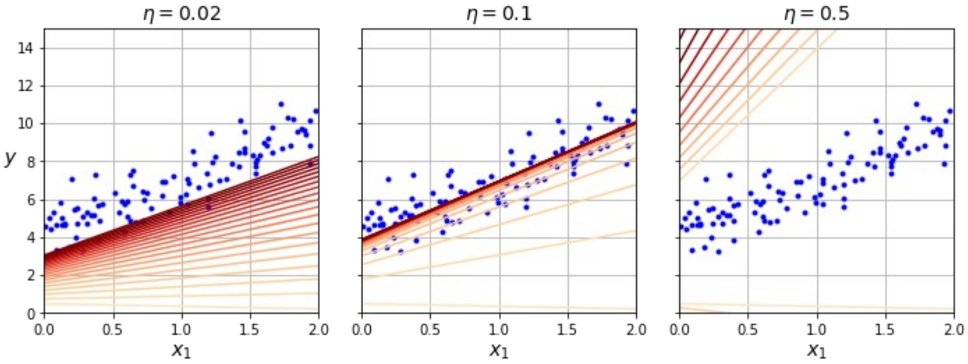
- Notice that this formula involves calculations over the full training set **X,** at each gradient descent step.
- This is why, the algorithm is called batch gradient descent.

➔ Terribly slow on very large training sets.

➔ Scales well with #features.

➔ Training a linear regression model when there are 100,000s features is much faster with Gradient Descent than Normal Equation or SVD decomposition.

Convergence rate



- To find a good convergence rate, you can use a Grid Search
  - However, you may want to limit the number of epochs so that grid search can eliminate models that take too long to converge
- How to set the number of epochs?
  - Too low: risk to be far away from the optimal solution
  - Too high: waste time while the model parameters do not change
  - A simple solution: set a very large number of epochs but interrupt the algorithm when the gradient vector becomes tiny – when its norm becomes smaller than a tiny value ϵ called the tolerance – this happens when gradient descent has almost reached the minimum.
- Convergence rate: when cost function convex and Lipschitz continuous, batch gradient descent will eventually converge to the optimal solution.
  - It can take $O(1/\epsilon)$ iterations.

TODO: (next 30 min)

- Implement Batch Gradient Descent
- Compare it with SVD and the Normal Equation according to #features and #instances (compute the runtime).

Stochastic Gradient Descent

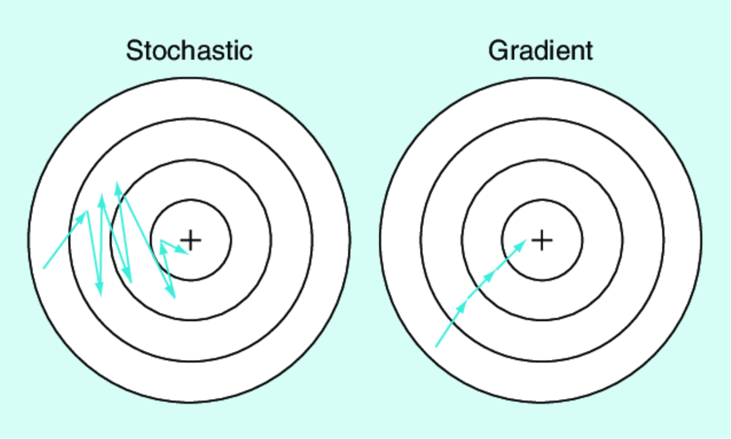
To avoid to take all the training set at each iteration

Batch Gradient Descent drawbacks and SGD motivations

- ❑ BGD uses the whole training set to compute the gradients at every step.
- ➔ Very slow when the training set is large.
- ❑ Stochastic Gradient Descent:
  - ❑ Pick a random instance in the training set at every step.
  - ❑ Compute the gradients based only on that single instance.

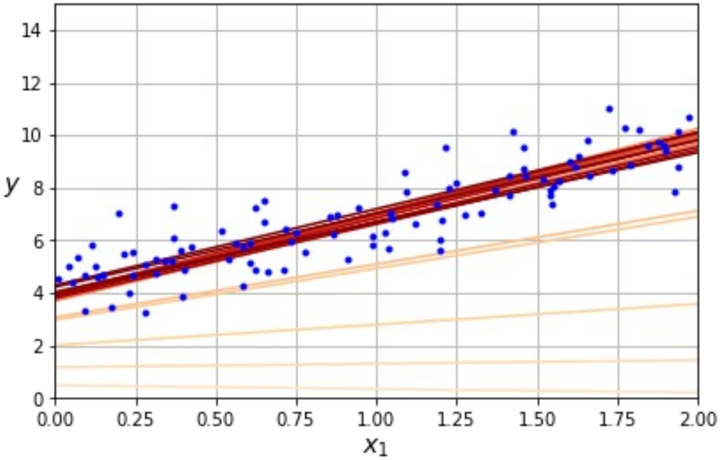
+ Much faster → possible to train on huge data

- - Much less regular than batch gradient descent
- - Final parameters values will be good, but not optimal.



Other advantages

- When the cost function is very irregular, SGD has better chance of finding globad minimum than BGD (random instance can help jump out of local minima).
- 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: gradually reduce the learning rate.
    - ■ Steps start out large (help make quick progress and escape local minima)
    - ■ Then get smaller and smaller, allowing the algorithm to settle at the global minimum.
    - ➔ **Simulated annealing**: inspired by the process in metallurgy of annealing, where molten metal slowly cooled down
  - ○ The function that determines the learning rate at each iteration is called the **learning schedule**.



By convention: we iterate by rounds of m iterations.
Each round is called an epoch.
On the dataset: BGD: 1000 epochs ; SGD 50 epochs with similar results

Warning

- When using SGD: the training instances must be independent and identically distributed (IID) to ensure the parameters get pulled toward the global optimum on average.
  - Simple way to ensure it: shuffle the instances during the training (pick each instance randomly or shuffle the training set at the beginning of each epoch).
- If you do not shuffle the instances:
  - If the data are sorted by labels: SGD will start by optimizing for one label, then the next …. It will not settle close to the global minimum.

TODO

- RTFM: partial_fit() method for scikit-learn estimators. (useful when you need to control the training process).

TODO: (next 0.42*42 min)

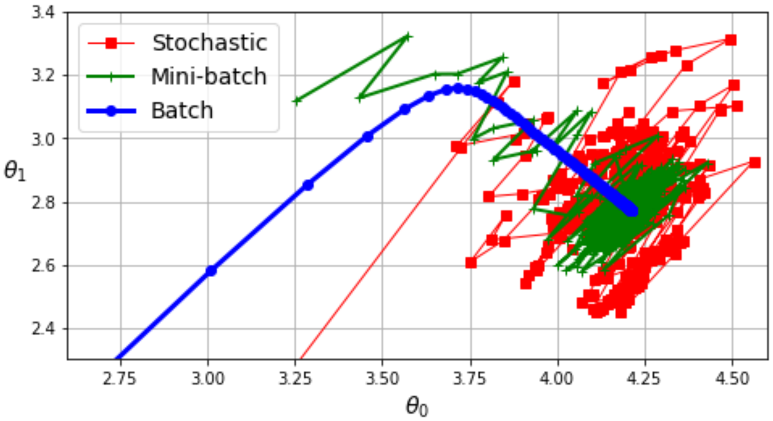
- Implement Stochastic Gradient Descent
- Compare it with BGD according to #features and #instances (compute the runtime, and the performance).

Mini-Batch Gradient Descent

A good trade-off

Mini-batch Gradient Descent

- Straightforward once you know BGD and SGD
- At each step:
  - Instead of computing the gradients based on the full training set (BGD) or based on just an instance (SGD), compute the gradients on small random sets of instances called mini-batches.
- The main advantage over SGD:
  - Get a performance boost from hardware optimization of matrix operations
  - Especially when using GPUs.
- Algorithm's progress in parameter space is less erratic than SGD.



At the end, all end up near the minimum but with different execution times.

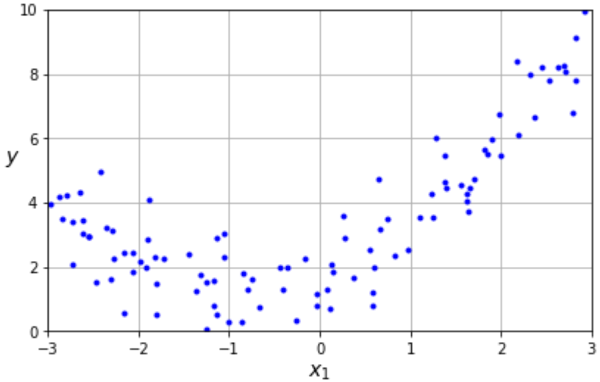
Comparison of algorithms for linear regression

Algorithm	Large m	Out-of-core support	Large n	Hyperparams	Scaling required	Scikit-Learn
Normal Equ.	Fast	No	Slow	0	No	N/A
SVD	Fast	No	Slow	0	No	LinearRegression
Batch GD	Slow	No	Fast	2	Yes	N/A
Stochastic GD	Fast	Yes	Fast	≥ 2	Yes	SGDRegressor
Mini-Batch GD	Fast	Yes	Fast	≥ 2	Yes	N/A

TODO: (next 0.42 (0.42*42) min)

- Implement Mini-Batch Gradient Descent
- Compare it with BGD and SGD according to #features and #instances (compute the runtime, and the performance).

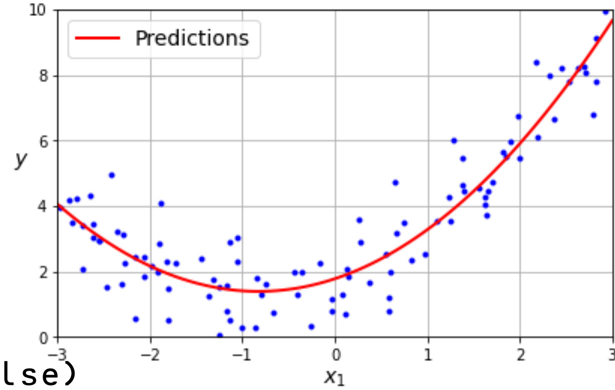
```
y = 0.5 * X ** 2 + X + 2 + np.random.randn(m, 1)
```



Polynomial Regression

When your model is more complex than a straight line … you can use linear model to fit non linear data.

Not learn a polynomial regression, instead a linear regression on polynomila features



```
from sklearn.preprocessing import PolynomialFeatures

poly_features = PolynomialFeatures(degree=2, include_bias=False)
X_poly = poly_features.fit_transform(X)
X[0]→array([-0.75275929])
X_poly[0]→array([-0.75275929, 0.56664654])

lin_reg = LinearRegression()
lin_reg.fit(X_poly, y)
lin_reg.intercept_, lin_reg.coef_
→(array([1.78134581]), array([[0.93366893, 0.56456263]]))
```

Warning: multiple features

- Polynomial regression is capable of finding relationships between features, which is something a plain linear regression model cannot do.
- PolynomialFeatures(degree=3) would not only add the features a^3,b^3,c^3
  - But also the combination ab, a^2b and ab^2.

➔ PolynomialFeatures(degree=d) transforms an array contening n features into an array containing (n+d)!/(d!n!) features.

➔ Beware of the combinatorial explosion of the number of features.

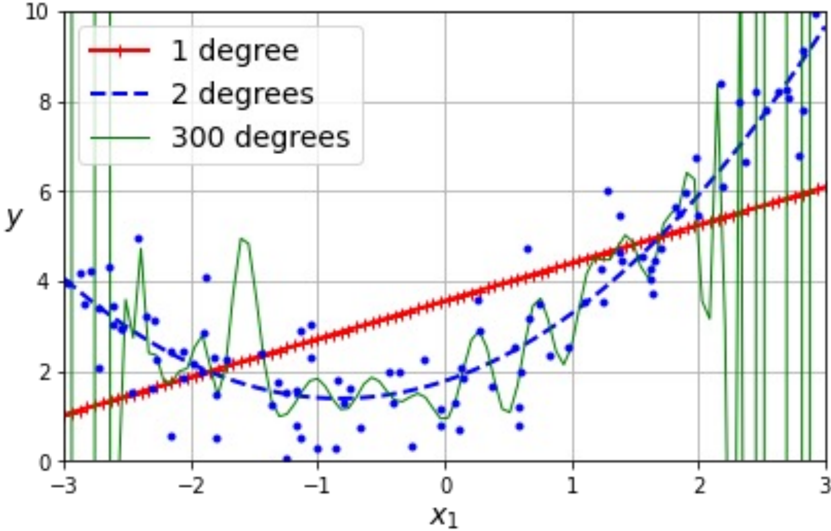
Learning Curves

- If you perform a high-degree polynomial regression, you will likely fit the training data much better than with plain linear regression.
- Ex: 300 degree regression vs quadratic model vs pure linear model

➔ Severely overfitting the training data.

➔ How can you decide how complex your model should be ?

  - ➢ √ Cross-validation
  - ➢ Learning curves

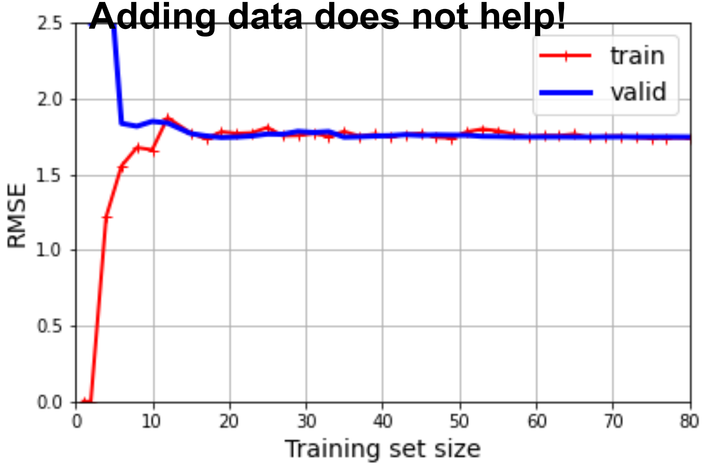


Learning curves

- Plots of the model’s training error and validation error as a function of the training iteration.
    - Just evaluate the model at regular intervals during training on **both** the **training** set and the **validation** set.
    - partial_fit() √ or train model several times with gradually larger subsets of the training set.
    - ScikitLearn has a useful learning_curve() function.
        - It trains and evaluates the model using CV

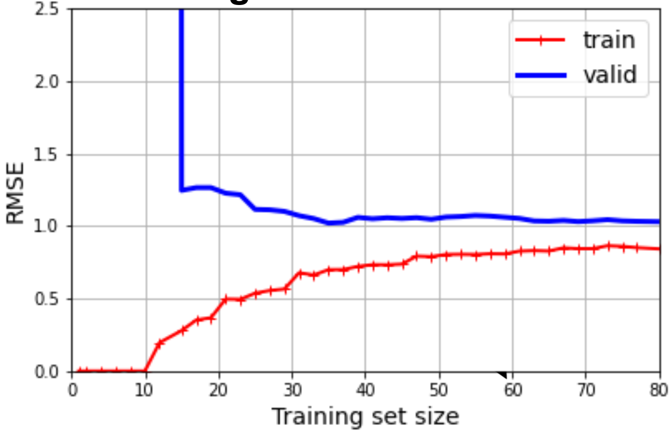
Plain linear model: **underfitting**

Adding data does not help!



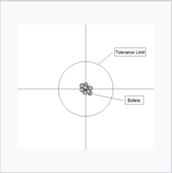
10th degree polynomial model:

Overfitting

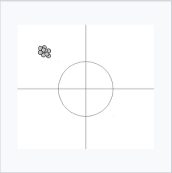


The Bias/Variance Trade-off

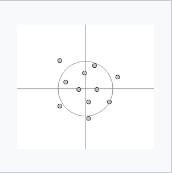
https://en.wikipedia.org/wiki/Bias%E2%80%93variance_tradeoff



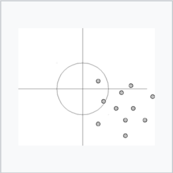
bias low, variance low



bias high, variance low



bias low, variance high



bias high, variance high

- An important theoretical result of statistics and machine learning is the fact that a model's generalization error can be expressed as the sum of 3 very different errors:
  - **Bias**: part of the generalization error due to wrong assumptions (assuming model is linear when it is actually quadratic). **High bias → model underfits.**
  - **Variance**: due to model's excessive sensitivity to small variation in the training data (model with many degree of freedom). **High variance → model overfits**.
  - **Irreducible error**: part due to the noiseness of the data itself. Only way to reduce it is to clean the data.
- Increasing model's complexity ➔ increase its variance and reduce its bias
- Reducing model's complexity ➔ increase its bias and reduce its variance

➔ Trade-off

Regularized Models

Regularize a polynomial model to reduce the number of polynomial degrees and prevent overfitting.

Regularization in linear models

- Achieved by constraining the weights of the model
    - Ridge regression
    - Lasso regression
    - Elastic net regression
- Which implement 3 different ways to constrain the weights.

Ridge Regression (aka Tikhonov regularization)

- A regularized version of the linear regression
- A regularization term equal to is added to MSE: $\frac{\alpha}{m}\sum_{i=1}^{m}\theta_i^2$

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

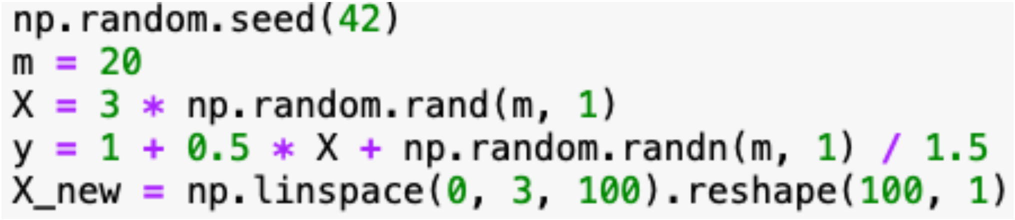
➔ This forces the algorithm to not only fit the data but also keep the model weights as small as possible.

❖ Note that the regularization term should only be added to the cost function during the training. ➔ Once the model is trained, use the unregularized MSE to evaluate the model's performance.

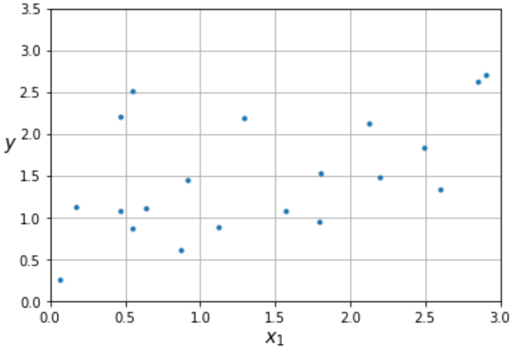
❖ Note that the bias term is not regularized.

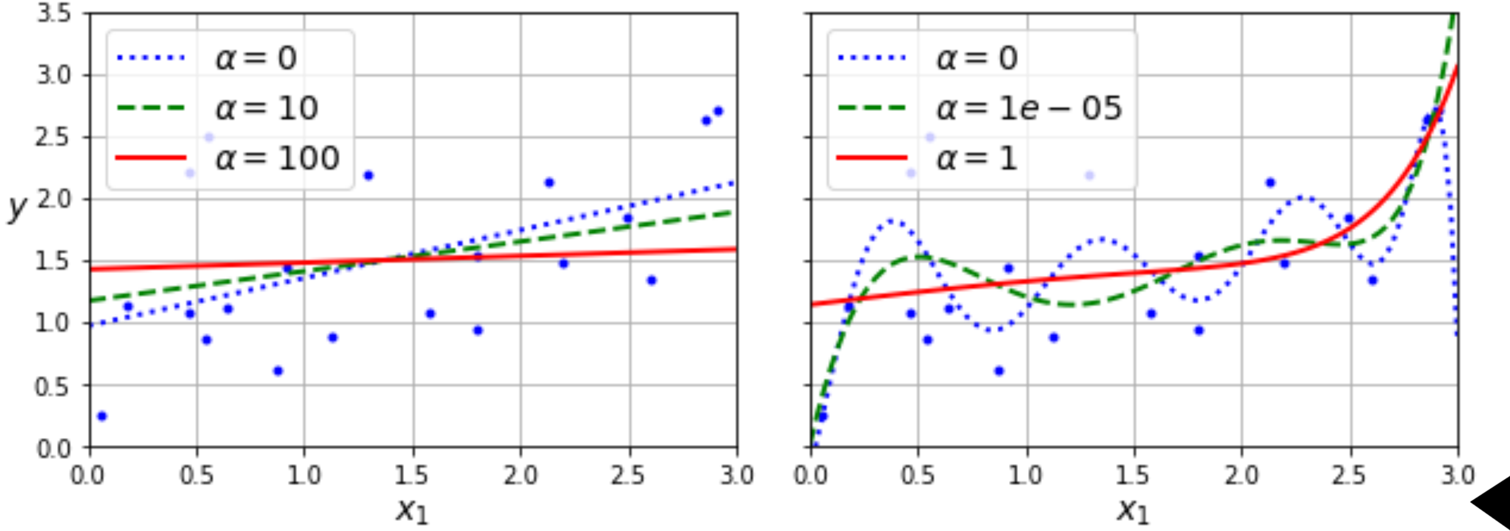
❖ Let **w** be the vector of feature weights (θ_1 to θ_n).

❖ In Norm ℓ_2, just add $2\alpha\mathbf{w}/m$ to the part of the MSE gradient vector without adding anything to the gradient of the bias term.



```
np.random.seed(42)
m = 20
X = 3 * np.random.rand(m, 1)
y = 1 + 0.5 * X + np.random.randn(m, 1) / 1.5
X_new = np.linspace(0, 3, 100).reshape(100, 1)
```





- Ridge regression closed form solution: $\hat{\theta} = \left(\mathbf{X}^\top \mathbf{X} + \alpha \mathbf{A}\right)^{-1} \mathbf{X}^\top \mathbf{y}$
  - The pros and cons remain the same.
- In Scikit-Learn, perform Ridge regression using closed-form solution:
  - ridge_reg = Ridge(alpha=0.1, solver="cholesky")
  - ridge_reg.fit(X, y)
- Using SGD:

```
sgd_reg = SGDRegressor(penalty="l2", alpha=0.1 / m, tol=None,  max_iter=1000, eta0=0.01, random_state=42)
sgd_reg.fit(X, y.ravel())  # y.ravel() because fit() expects 1D targets
sgd_reg.predict([[1.5]])
```

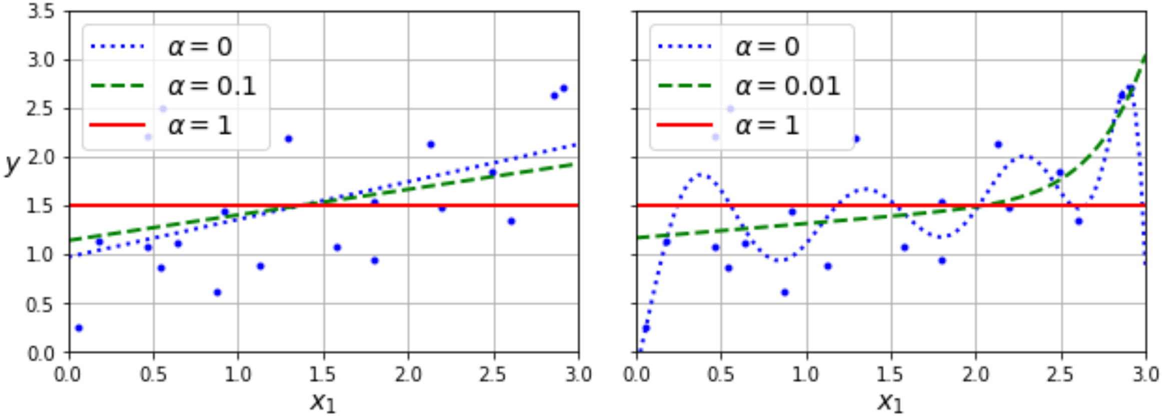
- RidgeCV class also performs ridge regression but it automatically tunes hyperparameters using cross validation. Roughly equivalent to using GridSearchCV but optimized for ridge regression and runs much faster.
- Several other estimators have efficient CV variants.

Lasso Regression

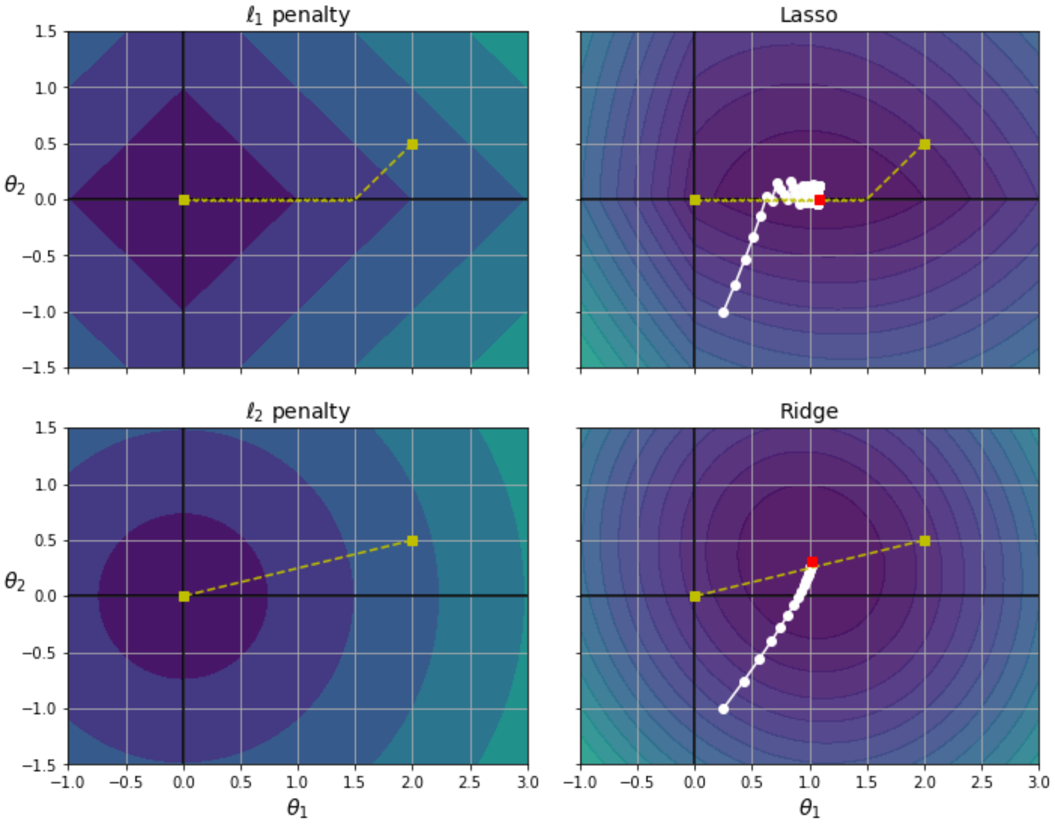
- Least Absolute Shrinkage and selection operator regression.
- Similar to ridge regression except:
  - It uses norm ℓ_1
  - Multiplied by 2α instead α/m.
  - Factors were chosen to ensure that the optimal α value is independent from the training set size.

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

- Important feature: Lasso tends to eliminate the weights of the least important features (i.e., set them to zero).



Lasso vs ridge regularization



Lasso cost function is not differenciable at $\theta_i = 0$ for (i= 1, 2, ..., n)

- But gradient descent still works if you use subgradient vector **g** instead when any $\theta_i = 0$

$$g(\theta, J) = \nabla_\theta MSE(\theta) + \begin{pmatrix} sign(\theta_1) \\ sign(\theta_2) \\ \vdots \\ sign(\theta_n) \end{pmatrix} \text{ where } \mathrm{sign}(\theta_i) = \begin{cases} -1, & \text{if } \theta_i < 0. \\ -0, & \text{if } \theta_i = 0. \\ +1, & \text{if } \theta_i > 0. \end{cases}$$

Elastic Net Regression

- In a middle ground between ridge regression and lasso regression.
- The regularization term is a weighted sum of both ridge and lasso's regularization terms.
- You can control the mix ratio r.
  - When r=0 ➔ ridge regression
  - When r=1 ➔ lasso regression

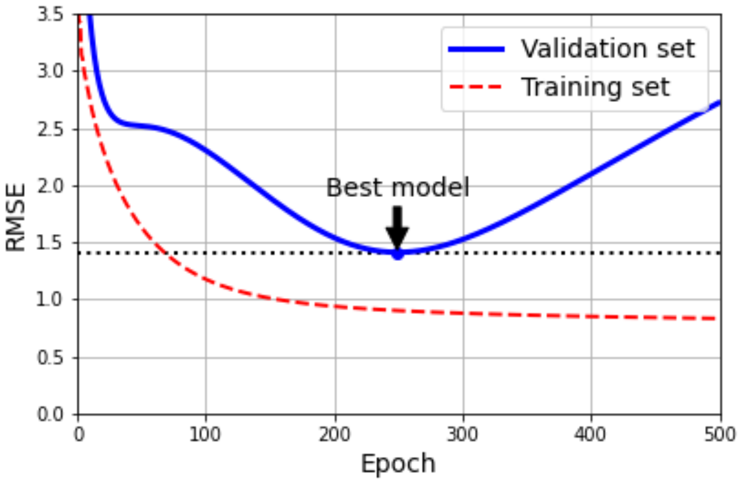
$$J(\theta) = MSE(\theta) + r\left(2\alpha\sum_{i=1}^{n}|\theta_i|\right) + (1-r)\left(\frac{\alpha}{m}\sum_{i=1}^{n}\theta_i^2\right)$$

Early Stopping

Another way to regularize iterative learning algorithm: stop training as soon as good conditions appear.

Early Stopping

- Stop training as soon as the validation error reaches a minimum
  - After the model start to overfit.
- Simple and efficient regularization technique: « *Beautiful Free Lunch* » (Geoffrey Hinton)



- With SGD and MBGD, the curves are not so smooth ➔ it may be hard to know whether you avec reached the minimum or not.
- One solution:
    - Stop only after the validation error has been above the minimum for some time (when you are confident that the model will not do any better).
    - Then roll-back the model parameters to the point where the validation error was at a minimum.
- To implement it with SGDRegressor: use partial_fit() instead of fit().

TODO: (next α 0.42 (0.42*42) min)

- Implement Early Stopping in a Mini-Batch Gradient Descent

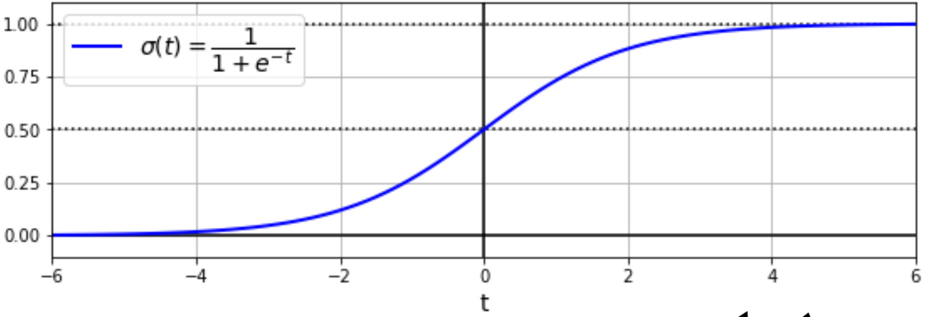
Logistic Regression

Using regression for classification: estimating the probability that an instance belongs to a particular class

- If the estimated probability is greater than a threshold (e.g., 50%) then :
    - The model predicts that the instance belongs to that class (called positive class, labeled ‘1’)
    - Otherwise, it predicts the negative class.
- This makes it a binary classifier.

Estimating probabilities

- Similarly to linear regression, a logistic regression model computes a weighted sum of the input features (plus a bias term).
- Instead of outputing the result directly like the linear regression model does, it outputs the *logistic* of this result :
- $\hat{p} = h_{\boldsymbol{\theta}}(x) = \sigma(\boldsymbol{\theta}^\top \boldsymbol{x})\ with\ \sigma(t) = \frac{1}{1+e^{-t}}$
  - The logistic is a sigmoid function



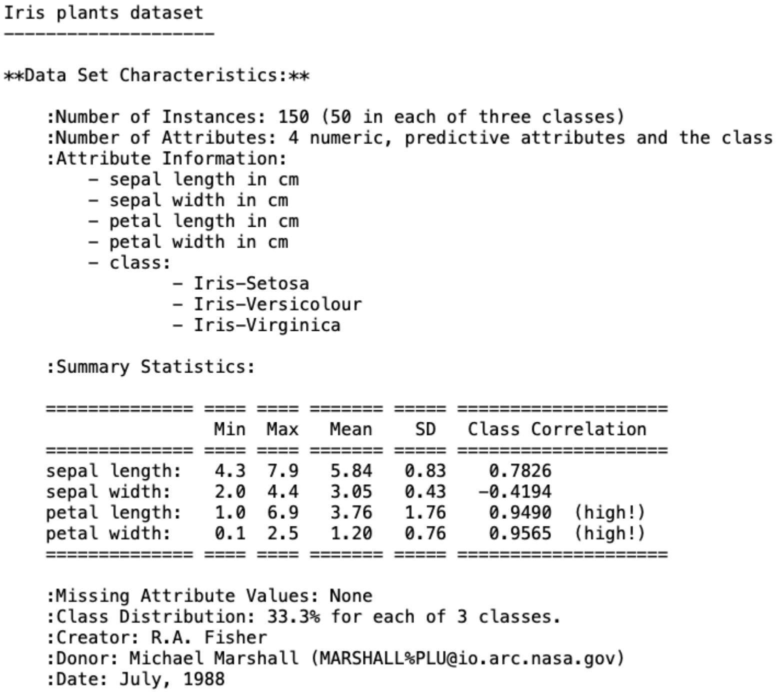
Training and cost function

- For a single training instance:
  - $c(\boldsymbol{\theta}) = \begin{cases} -\log(\hat{p}) \;\; if \; y = 1 \\ -\log(1-\hat{p}) \; if \; y = 0 \end{cases}$
  - -log(t) grows very large when t approaches 0 ➔ cost will be large if the model estimates a probability close to 0 for a positive instance
  - Similarly, if the model estimates a probability close to 1 for a negative instance, cost will be large.
- Cost function
  - $J(\boldsymbol{\theta}) = -\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\log\left(\hat{p}^{(i)}\right) + \left(1-y^{(i)}\right)\log\left(1-\hat{p}^{(i)}\right)\right)$
- A principled function: minimizing this loss ⬄ Maximum Likehood (assuming instances follow a Gaussian distribution around the mean of their class)

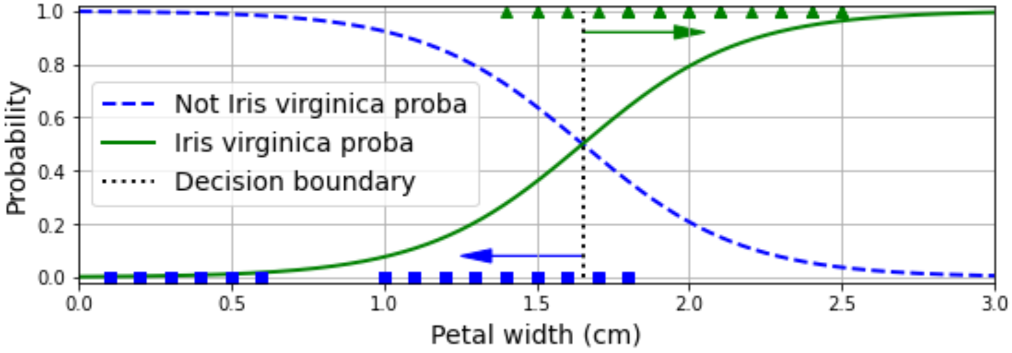
- (Bad?) news : No closed-form equation ➔ no equivalent to Normal Equation.
- Derivatives:
  - $\frac{\partial}{\partial \theta_j} J(\boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^{m} \left(\sigma\left(\boldsymbol{\theta}^\top \boldsymbol{x}^{(i)} \right) - y^{(i)} \right) x_j^{(i)}$

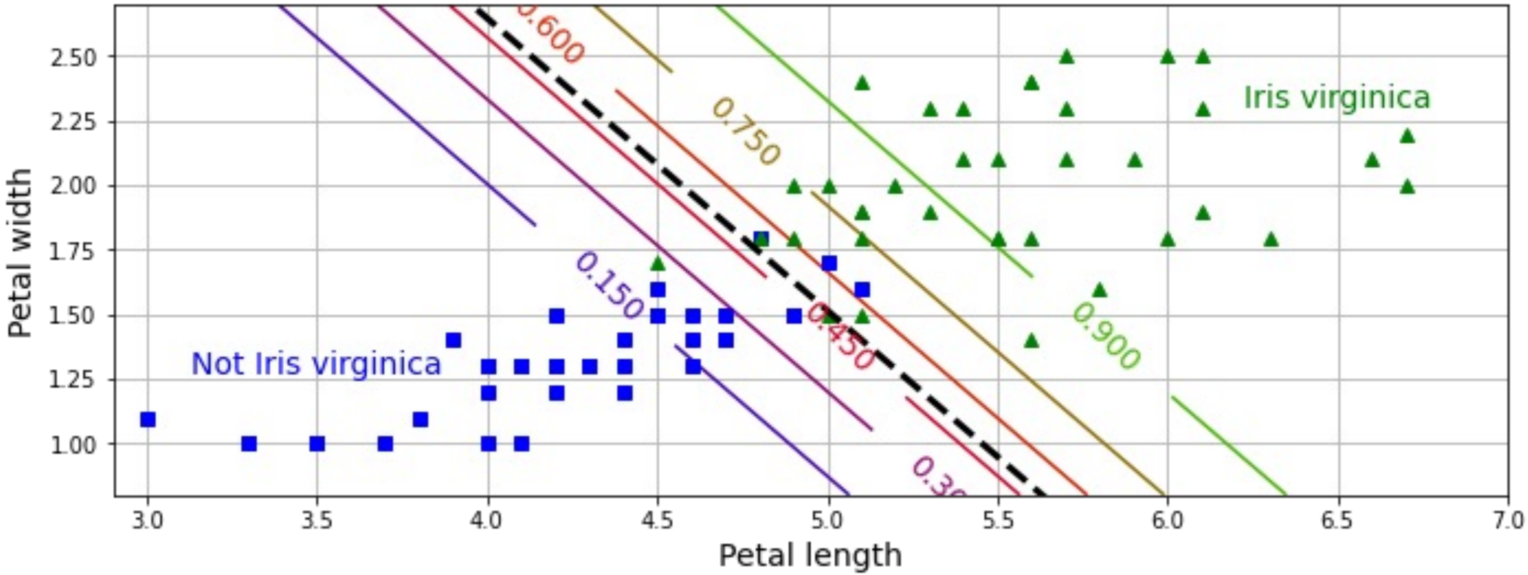
Decision boundaries

- Iris dataset (only petal_width for the moment)



```
Iris plants dataset
--------------------

**Data Set Characteristics:**

    :Number of Instances: 150 (50 in each of three classes)
    :Number of Attributes: 4 numeric, predictive attributes and the class
    :Attribute Information:
        - sepal length in cm
        - sepal width in cm
        - petal length in cm
        - petal width in cm
        - class:
                - Iris-Setosa
                - Iris-Versicolour
                - Iris-Virginica

    :Summary Statistics:

    ============== ==== ==== ======= ===== ====================
                    Min  Max   Mean    SD   Class Correlation
    ============== ==== ==== ======= ===== ====================
    sepal length:   4.3  7.9   5.84   0.83    0.7826
    sepal width:    2.0  4.4   3.05   0.43   -0.4194
    petal length:   1.0  6.9   3.76   1.76    0.9490  (high!)
    petal width:    0.1  2.5   1.20   0.76    0.9565  (high!)
    ============== ==== ==== ======= ===== ====================

    :Missing Attribute Values: None
    :Class Distribution: 33.3% for each of 3 classes.
    :Creator: R.A. Fisher
    :Donor: Michael Marshall (MARSHALL%PLU@io.arc.nasa.gov)
    :Date: July, 1988
```





- With Petal width and length as features
- -- the border: model estimates 50%/50%
  - The model decision boundary
  - Set of points **x** such that $\theta_0 + \theta_1 x_1 + \theta_2 x_2 = 0$ which defines **a straight line**

- The hyperparemeter controlling the regularization strength of Scikit-Learn Logistic Regression model is not alpha (as in other linear models) but its inverse: **C**.
    - The higher the value of C, the less the model is regularized.

Softmax Regression

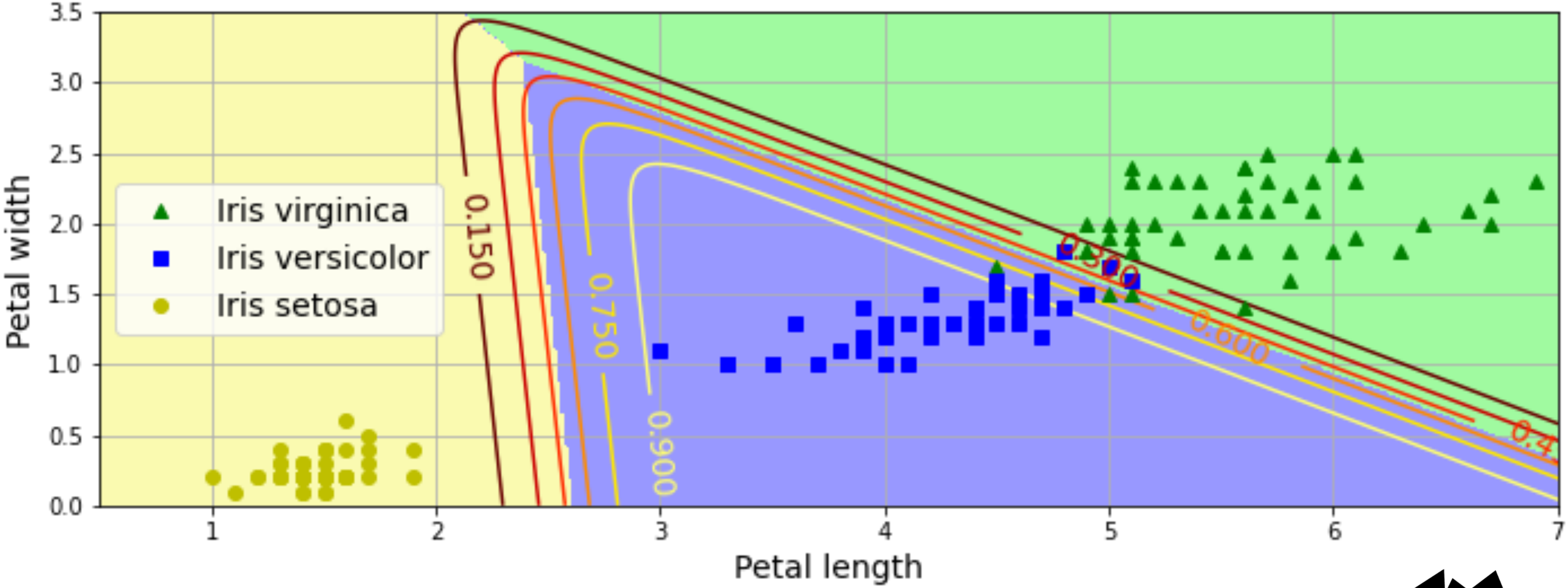
To support multiclass classification

Supporting multiple classes

- Logistic regression can be generalized to support multiple classes directly, without having to train and combine multiple binary classifiers (OVR, OVO)
- Idea:
  - Given an instance **x**, the softmax regression model first computes a score s_k(**x**) for each class k
  - Then estimate the probability of each class by applying the *softmax function* (aka normalized exponential) to the scores.
  - Softmax score for class k : $s_k(\boldsymbol{x}) = \left(\boldsymbol{\theta}^{(k)}\right)^\top \boldsymbol{x}$
  - Each class has its own dedicated vector $\boldsymbol{\theta}^{(k)}$ ➔ all these vector are stored as rows in a **parameter matrix Θ**.
  - Once you have the score ➔estimate the probability $\widehat{p_k}$
  - $\hat{p}_k = \sigma\left(s(\boldsymbol{x})\right)_k = \frac{\exp(s_k(\boldsymbol{x}))}{\sum_{k=1}^{K} \exp(s_j(\boldsymbol{x}))}$
    - K is the number of classes, s(**x**) is a vector containing the scores of each class for the instance x. $\sigma\left(s(\boldsymbol{x})\right)_k$ is the estimated probability that **x** belong ot class k, given the scores of each class for **x**.
  - By default: $\hat{y} = argmax_k\left(s_k(\boldsymbol{x})\right) = argmax_k\left(\left(\boldsymbol{\theta}^{(k)}\right)^\top \boldsymbol{x}\right)$

Cost function and gradient

- The objective is to have a model that estimate high probability for the target class and a low probability for the others.
- Cross Entropy:
  - $J(\Theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log\left(\hat{p}_k^{(i)}\right)$
  - $y_k^{(i)}$ is the target probability that the *ith* instance belongs to class *k*. In general, it is either 1 or 0.
  - When K=2 ➔ Logistic regression cost function (log loss).
- The gradient vector:
  - $\nabla_{\boldsymbol{\theta}^{(k)}} J(\boldsymbol{\Theta}) = \frac{1}{m}\sum_{i=1}^{m}\left(\hat{p}_k^{(i)} - y_k^{(i)}\right)\boldsymbol{x}^{(i)}$
  - Now you can compute the gradient vector for every class, then use gradient descent to find the parameter matrix $\boldsymbol{\Theta}$ that minimizes the cost function.



3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
Petal width
1
2
3
4
5
6
7
Petal length
▲ Iris virginica
■ Iris versicolor
● Iris setosa
0.150
0.750
0.900
0.300
0.600
0.4

Summary

- Various way to train linear models both for regression and classification
  - Closed-form equation and gradient descent to solve linear regression
- Various penalties can be added to the cost function during training to regularize the model
  - Ridge, Lasso, Elastic net
- Early Stopping
- Learning curves
- Logistic / softmax regression

TODO: (now)

- Implement batch gradient descent with early stopping for softmax regression without using Scikit-Learn (only NumPy).
- Use it on two classification tasks (Iris and (Titanic or spam or …))
- Take a Ranking dataset and do retro-engineering to find the formulae and do « if » simulation.

EPITA
ÉCOLE D'INGENIEURS EN INFORMATIQUE

Merci !

EPITA

ÉCOLE D’INGENIEURS EN INFORMATIQUE