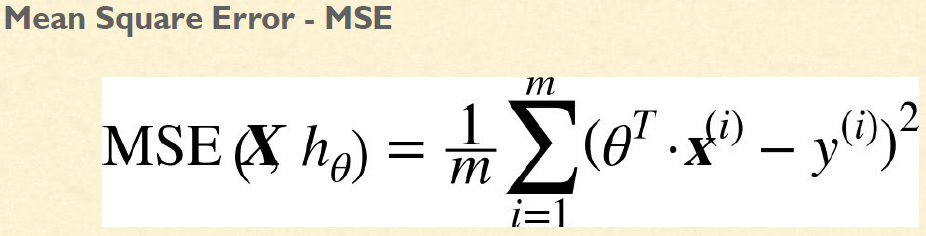
Training Models

# Linear Regression

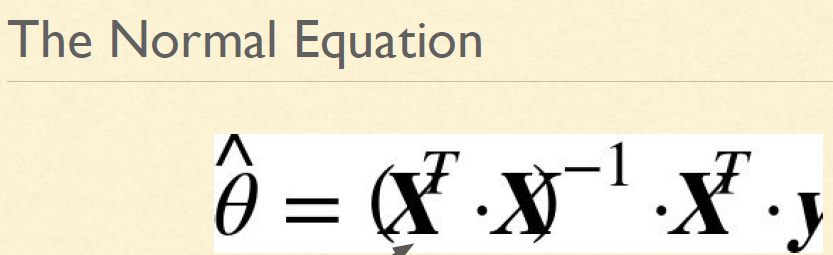
Equation of Line , y = mx+c

C = Intercept/Bias, where it intercepts the y-axis. If c=0, it will pass through origin(0,0). It is constant & doesn’t depends on any of the features.



**There are two approaches to train the linear regression model**

* The normal “closed-form” equation
* Gradient Descent (GD) - Iterative optimization approach



import numpy as np

**#Uniformly Distributed Array**

X= 2 \* np.random.rand(100,1)

y= 4 + 3 \* X + np.random.rand(100,1)

plt.plot(X, y, "b.")

plt.xlabel("$x\_1$", fontsize=18)

plt.ylabel("$y$", rotation=0, fontsize=18)

plt.axis([0, 2, 0, 15])

plt.show()

**#Add Bias to Dataset**

X\_b = np.c\_[np.ones((100,1)),X]

**#Apply the Normal Equation to compute theta hat**

theta\_best = np.linalg.inv(X\_b.T.dot(X\_b)).dot(X\_b.T).dot(y)

**#Make Prediction using theta\_best**

X\_new = np.array([[0],[2]])

X\_new\_b = np.c\_[np.ones((2,1)),X\_new]

Y\_predict = X\_new\_b.dot(theta\_best)

**#Plot the Predictions**

plt.plot(X\_new, y\_predict, "r-", linewidth=2, label="Predictions")

plt.plot(X, y, "b.")

plt.xlabel("$x\_1$", fontsize=18)

plt.ylabel("$y$", rotation=0, fontsize=18)

plt.legend(loc="upper left", fontsize=14)

plt.axis([0, 2, 0, 15])

plt.show()

**#Using Scikit-Learn**

from sklearn.linear\_model import LinearRegression

lin\_reg = LinearRegression()

lin\_reg.fit(X,y)

lin\_reg.intercept\_,lin\_reg.coef\_

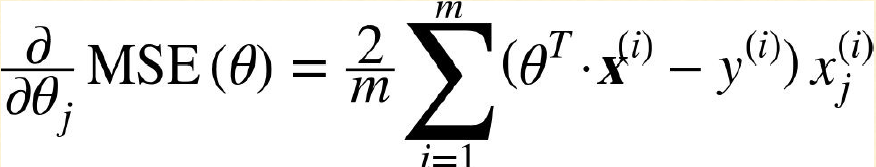
lin\_reg.predict(X\_new)

🡪Computational Complexity = O(n3), where n = no. of features. So if features got increased it will be very slow.

# Gradient Descent

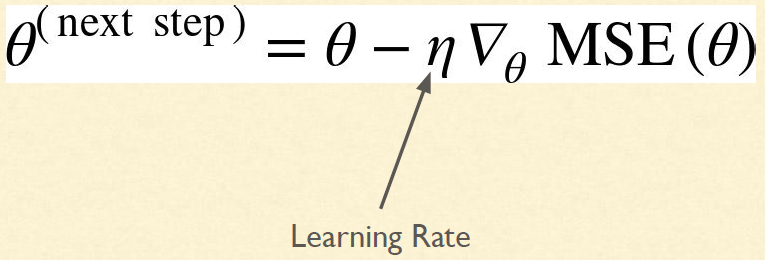
Measures the local gradient of the cost function with regards to the parameter vector θ and it goes in the direction of descending gradient until the gradient becomes zero.

* Start by filling θ with random values - Random initialization
* Each step is also called as **Learning Rate.**  If Learning Rate is too small, then it will take long time to converge. If the Learning Rate is too large, it will keep on bouncing.
* **Problem :** Local Minima,holes,ridges,plateaus
* **FeatureScaling** must need to be implemented while using Gradient Descent.
* **Gradient Descent Algorithms : Batch GD, Stochastic GD & Mini-Batch GD.**
* *Batch Gradient Descent :*
  + Random Initialization : Initialize model Parameter.
  + ***Partial Derivative*** : For each of the weights, we change one weight and keeping others fixed and calculate the change in cost
  + ***Partial Derivative Cost Function*** :



***This formula involves calculations over the full training set X at every step, therefore the name Batch Gradient Descent***

* + *It is better than Linear Regression when no. of features is more. But is very slow on large Datasets since it uses entire Dataset for each step.*



**#Batch Gradient Descent**

eta = 0.01 #Learning Rate

n\_iterations = 100

m = 100

theta = np.random.randn(2,1) #Radom initialization of weight

for iteration in range(n\_iterations):

gradients = 2/m \* X\_b.T.dot(X\_b.dot(theta)-y)

theta = theta-eta\*gradients

if iteration % 50 :

print (theta)

**# Plot Gradient Descent for various learning rates**

theta\_path\_bgd = []

def plot\_gradient\_descent(theta, eta, theta\_path=None):

m = len(X\_b)

plt.plot(X, y, "b.")

n\_iterations = 1000

for iteration in range(n\_iterations):

if iteration < 10:

y\_predict = X\_new\_b.dot(theta)

style = "b-" if iteration > 0 else "r--"

plt.plot(X\_new, y\_predict, style)

gradients = 2/m \* X\_b.T.dot(X\_b.dot(theta) - y)

theta = theta - eta \* gradients

if theta\_path is not None:

theta\_path.append(theta)

plt.xlabel("$x\_1$", fontsize=18)

plt.axis([0, 2, 0, 15])

plt.title(r"$\eta = {}$".format(eta), fontsize=16)

np.random.seed(42)

theta = np.random.randn(2,1) # random initialization

plt.figure(figsize=(10,4))

plt.subplot(131); plot\_gradient\_descent(theta, eta=0.02)

plt.ylabel("$y$", rotation=0, fontsize=18)

plt.subplot(132); plot\_gradient\_descent(theta, eta=0.1, theta\_path=theta\_path\_bgd)

plt.subplot(133); plot\_gradient\_descent(theta, eta=0.5)

save\_fig("gradient\_descent\_plot")

plt.show()

***Find Good Learning Rate - Solution***

*Set a very large number of iterations and Interrupt the algorithm when the gradient vector becomes tiny When norm of a gradient vector becomes smaller than ϵ - tolerance*

# Stochastic Gradient Descent

**Problem with Batch Gradient Descent**

* Slow when the training set is large, as it uses the whole training set to compute the gradient at every step

**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
* Once the Algorithm stops, final parameters values are good but not optimal.
* **Simulated Annealing** is the solution process by which **Learning Rate** gradually decreases.
* **Learning Schedule** is a function which determines the change in **Learning Rate** at each iteration.
* Each Round or iteration is called **EPOCH.**

m = len(X\_b)

np.random.seed(42)

n\_epochs = 50

t0,t1 = 5,50 #Learning Schedule Hyperparameter

def learning\_schedule(t):

return t0/(t+t1)

theta = np.random.randn(2,1) #Random initialization

theta\_path\_sgd = []

for epoch in range (n\_epochs):

for i in range(m):

if epoch ==0 and i<20:

y\_predict = X\_new\_b.dot(theta)

style = “b-” if i > 0 else “r--”

plt.plot(X\_new,y\_predict,style)

random\_index = np.random.randint(m)

xi = X\_b[random\_index:random\_index+1]

yi = y[random\_index:random\_index+1]

gradients = 2\*xi.T.dot(xi.dot(theta)-yi)

eta = learning\_schedule(epoch\*m+i)

theta = theta – eta \* gradients

theta\_path\_sgd.append(theta)

plt.plot(X,y,”b.”)

plt.xlabel("$x\_1$", fontsize=18)

plt.ylabel("$y$", rotation=0, fontsize=18)

plt.axis([0, 2, 0, 15])

save\_fig("sgd\_plot")

plt.show()

**#Linear regression using Scikit SGD Regressor**

from sklearn.linear\_model import SGDRegressor

sgd\_reg = SGDRegressor(n\_iter=50,penalty=None,eta0=0.1,random\_state=42)

sgd\_reg.fit(X,y.ravel())

sgd\_reg.intercept\_,sgd\_reg.coef\_

# Mini-Batch Gradient Descent

Computes gradient on small random sets of instances

theta\_path\_mgd = []

n\_iterations = 50

minibatch\_size = 20

np.random.seed(42)

theta = np.random.randn(2,1)

t0,t1 = 10,1000

def learning\_schedule(t):

return t0/(t+t1)

t= 0

for epoch in range(n\_iterations):

shuffled\_indices = np.random.permutations(m)

X\_b\_shuffled = X\_b[shuffled\_indices]

y\_shuffled = y[shuffled\_indices]

xi = X\_b\_shuffled[:minibatch\_size]

yi = y\_shuffled[:minibatch\_size]

gradients = 2 \* xi.T.dot(xi.dot(theta)-yi)

eta = learning\_schedule(epoch)

theta = theta-eta\*gradients

theta\_path\_mgd.append(theta)

***# Plot paths taken by the 3 Gradient Descent algorithms during training***

theta\_path\_bgd = np.array(theta\_path\_bgd)

theta\_path\_sgd = np.array(theta\_path\_sgd)

theta\_path\_mgd = np.array(theta\_path\_mgd)

plt.figure(figsize=(7,4))

plt.plot(theta\_path\_sgd[:, 0], theta\_path\_sgd[:, 1], "r-s", linewidth=1, label="Stochastic")

plt.plot(theta\_path\_mgd[:, 0], theta\_path\_mgd[:, 1], "g-+", linewidth=2, label="Mini-batch")

plt.plot(theta\_path\_bgd[:, 0], theta\_path\_bgd[:, 1], "b-o", linewidth=3, label="Batch")

plt.legend(loc="upper left", fontsize=16)

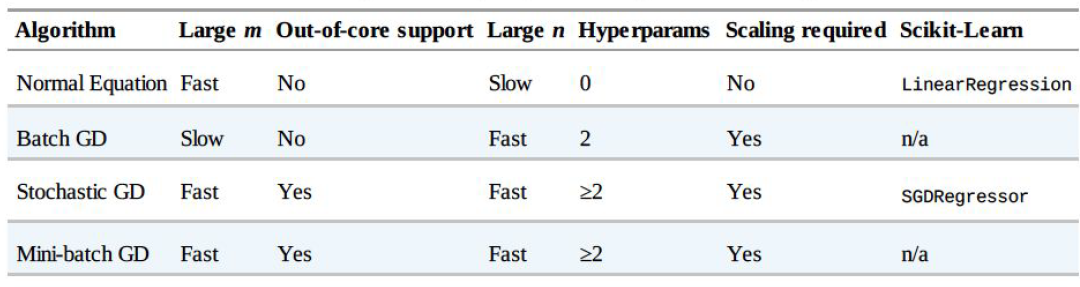
plt.xlabel(r"$\theta\_0$", fontsize=20)

plt.ylabel(r"$\theta\_1$ ", fontsize=20, rotation=0)

plt.axis([2.5, 4.5, 2.3, 3.9])

save\_fig("gradient\_descent\_paths\_plot")

plt.show()



# Polynomial Regression

A linear Model to fit non-linear data.

Feature = n, Degree = d. Transform array = n+d!/n!d!

***# Generate nonlinear and noisy dataset***

import numpy as np

import numpy.random as rnd

np.random.seed(42)

m=100

X = 6 \* np.random.rand(m,1)-3

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

plt.plot(X,y,’b.’)

plt.xlabel(“$X\_1$”,fontsize=18)

plt.ylabel(“$y$”,fontsize=18)

plt.axis([-3,3,0,10])

***# Add new features using Scikit's Learn PolynomialFeatures class***

from sklearn.preprocessing import PolynomialFeatures

poly\_features = PolynomialFeatures(degree=3,include\_bias=False)

X\_poly = poly\_features.fit\_transform(X)

***# Fit LinearRegression model***

from sklearn.linear\_model import LinearRegression

lin\_reg = LinearRegression()

lin\_reg.fit(X\_poly,y)

lin\_reg.intercept\_,lin\_reg.coef\_

***# Polynomial Regression model predictions***

X\_new=np.linspace(-3, 3, 100).reshape(100, 1)

X\_new\_poly = poly\_features.transform(X\_new)

y\_new = lin\_reg.predict(X\_new\_poly)

plt.plot(X, y, "b.")

plt.plot(X\_new, y\_new, "r-", linewidth=2, label="Predictions")

plt.xlabel("$x\_1$", fontsize=18)

plt.ylabel("$y$", rotation=0, fontsize=18)

plt.legend(loc="upper left", fontsize=14)

plt.axis([-3, 3, 0, 10])

plt.show()

* **Learning Curve**

***# High degree polynomials***

**from** **sklearn.preprocessing** **import** StandardScaler

**from** **sklearn.pipeline** **import** Pipeline

**for** style, width, degree **in** (("g-", 1, 300), ("b--", 2, 2), ("r-+", 2, 1)):

polybig\_features = PolynomialFeatures(degree=degree,include\_bias=**False**)

std\_scaler = StandardScaler()

lin\_reg = LinearRegression()

polynomial\_regression = Pipeline([

("poly\_features", polybig\_features),

("std\_scaler", std\_scaler),

("lin\_reg", lin\_reg),

])

polynomial\_regression.fit(X, y)

y\_newbig = polynomial\_regression.predict(X\_new)

plt.plot(X\_new, y\_newbig, style, label=str(degree), linewidth=width)

plt.plot(X, y, "b.", linewidth=3)

plt.legend(loc="upper left")

plt.xlabel("$x\_1$", fontsize=18)

plt.ylabel("$y$", rotation=0, fontsize=18)

plt.axis([-3, 3, 0, 10])

plt.show()

* **If Model is Over-fitting, one of the Solution is ‘Cross-Validation’. Other one is Learning Curve.**

*# Learning curves of the plain Linear Regression model*

**from** **sklearn.metrics** **import** mean\_squared\_error

**from** **sklearn.model\_selection** **import** train\_test\_split

**def** plot\_learning\_curves(model, X, y):

X\_train, X\_val, y\_train, y\_val = train\_test\_split(X, y, test\_size=0.2, random\_state=10)

train\_errors, val\_errors = [], []

**for** m **in** range(1, len(X\_train)):

model.fit(X\_train[:m], y\_train[:m])

y\_train\_predict = model.predict(X\_train[:m])

y\_val\_predict = model.predict(X\_val)

train\_errors.append(mean\_squared\_error(y\_train\_predict, y\_train[:m]))

val\_errors.append(mean\_squared\_error(y\_val\_predict, y\_val))

plt.plot(np.sqrt(train\_errors), "r-+", linewidth=2, label="train")

plt.plot(np.sqrt(val\_errors), "b-", linewidth=3, label="val")

plt.legend(loc="upper right", fontsize=14)

plt.xlabel("Training set size", fontsize=14)

plt.ylabel("RMSE", fontsize=14)

lin\_reg = LinearRegression()

plot\_learning\_curves(lin\_reg, X, y)

plt.axis([0, 80, 0, 3])

plt.show()

**🡪** If **RMSE** is very high, then it means Model is Underfitting. Then either we need to add more dataset or more features need to be added or more complex model needs to be selected.

***# Learning curves of the 10th-degree polynomial model on the same data***

**from** **sklearn.pipeline** **import** Pipeline

polynomial\_regression = Pipeline([

("poly\_features", PolynomialFeatures(degree=10, include\_bias=**False**)),

("lin\_reg", LinearRegression()),

])

plot\_learning\_curves(polynomial\_regression, X, y)

plt.axis([0, 80, 0, 3])

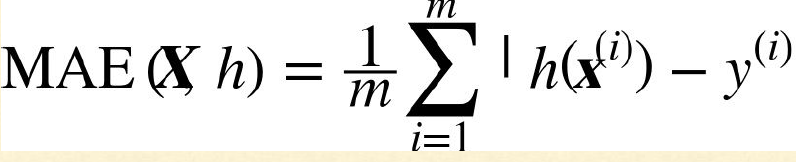
plt.show()

🡪 The Bias / Variance Tradeoff : A model’s generalization error can be expressed as the sum of three very different errors :

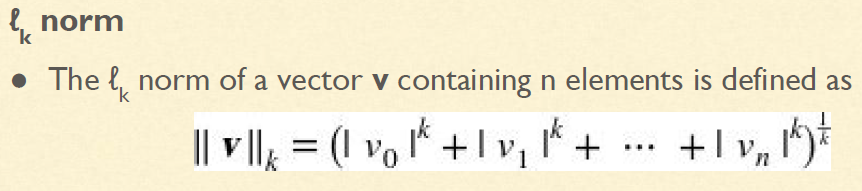
* Bias : High Bias leads to UNDERFIT the model.
* Variance : High Variance/More Features leads to OVERFIT the model.
* Irreducible error : Clean the data.

**More complex model means High Variance and Low Bias & vice-versa.**

* **Regularized Linear Models**
* To regularize a Polynomial Model, reduce the number of polynomial degree.
* To regularize a Linear Model, constrain the weights of the model. Three ways to constrain the weight :
  + **Ridge Regression**
  + **Lasso Regression**
  + **Elastic Net**
* **RMSE** is the preferred performance measure for Regression Task. But, in some case, like-when there are many outliers **MAE –(Mean Absolute Error/Average Absolute Deviation)** can be used.



* **There are various distance measures or norm to calculate distance between two vectors**
  + **Manhattan/L1 Norm :** Orthogonal line. Computing the sum of absolutes (MAE) corresponds to L1 Norm. Denoted as //.//1
  + **Euclidean Norm/L2 norm :** Straight Line between two points. Computing the root of a squares(RMSE) corresponds to L2 norm. Denoted as //.//2 or //.//



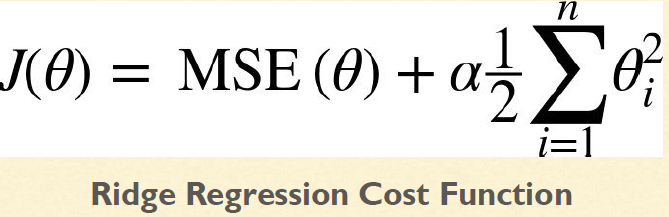
🡪The Higher the Norm index, more it focuses on large values & ignores the small ones.

🡪L1 norm yield sparse matrix, which helps in feature selection. i.e., which feature is important, and which is redundant.

# Ridge Regression

Regularized version of Linear Regression. A regularization term is added to the Cost Function.

* Adding a regularization cost to the cost function, that will force the model not only fit the data & but also keep the model weight small.
* Regularization cost is added only during Training.



* alpha will control cost function. If it is equal to 0, then it is same as Linear Regression.
* Scaling is required.

from sklearn.linear\_model import Ridge

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)

def plot\_model(model\_class,polynomial,alphas,\*\*model\_kargs):

for alpha,style in zip(alphas,(‘b-’,’g--’,’r:’)):

model = model\_class(alpha,\*\*model\_kargs) if alpha > 0 else LinearRegression()

if polynomial:

model = pipeline([

(‘poly\_features’,PolynomialFeatures(degree=10,include\_bias=False)),

(‘std\_scaler’,StandardScaler()),

(‘regul\_reg’,model),

])

model.fit(X,y)

y\_new\_regul = model.predict(X\_new)

lw = 2 if alpha > 0 else 1

plt.plot(X\_new,y\_new\_regul,style,linewidth=lw,label=r”$\alpha = {}$”.format(alpha))

plt.plot(X,y,”b.”,linewidth=3)

plt.legend(loc=”upper left”,fontsize=15)

plt.xlabel(“$X\_1$”,fontsize=18)

plt.axis([0,3,0,4])

plt.figure(figsize=(8,4))

plt.subplot(121)

plot\_model(Ridge,polynomial=False,alphas=(0,10,100),random\_sate=42)

plt.ylabel("$y$", rotation=0, fontsize=18)

plt.subplot(122)

plot\_model(Ridge, polynomial=**True**, alphas=(0, 10\*\*-5, 1), random\_state=42)

plt.show()

**# Ridge Regression with Scikit-Learn using a closed-form solution**

from sklearn.linear\_model import Ridge

ridge\_reg = Ridge(alpha=1,solver=’cholesky’,random\_state=42)

ridge\_reg.fit(X,y)

ridge\_reg.predict([[1.5]])

**# Ridge Regression with Scikit-Learn using SGD**

Sgd\_reg = SGDRegressor(penalty=”l2”,random\_state=42) **#L2**

Sgd\_reg.fit(X,y.ravel())

Seg\_reg.predict([[1.5]])

**# Ridge Regression with Scikit-Learn using Stochastic Average GD**

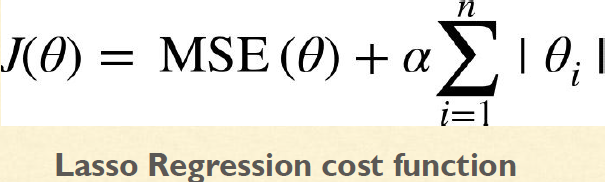
ridge\_reg = Ridge(alpha=1,solver=”sag”,random\_state=42)

ridge\_reg.fit(X,y)

ridge\_reg.predict([[1.5]])

# Lasso Regression

* Like Ridge Regression, but it uses ‘**L1 Norm**’



* It eliminates the weights of Least Important Features by setting it to 0.
* It performs the feature selection and output a Sparse Matrix.

from sklearn.linear\_model import Lasso

plt.figure(figsize=(8,4))

plt.sublplot(121)

plot\_model(Lasso,polynomial=False,alphas=(0,0.1,1),random\_state=42)

plt.ylabel(‘$y$’,rotation=0,fontsize=18)

plt.subplot(122)

plot\_model(Lasso,polynomial=True,alphas=(0,10\*\*-7,1),random\_state=42)

plt.show()

from sklearn.linear\_model import Lasso

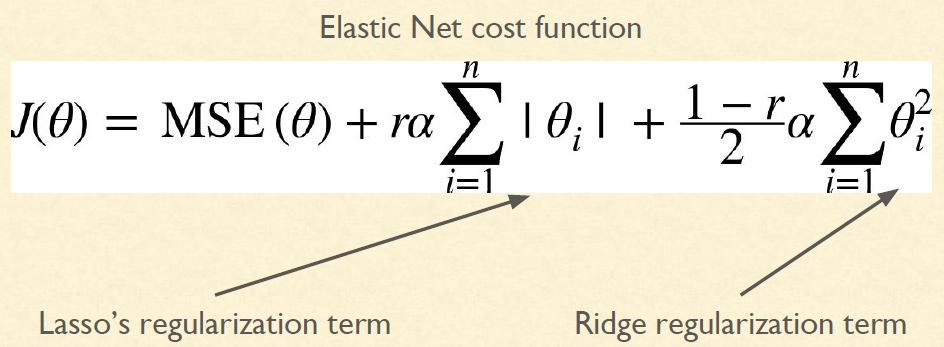
lasso\_reg = Lasso(alpha=0.1)

lasso\_reg.fit(X,y)

lasso\_reg.predict([[1.5]])

# Elastic Net Regression

* Mix of Ridge & Lasso Regression.



from sklearn.linear\_model import ElasticNet

elastic\_net = ElasticNet(alpha=0.1,l1\_ratio=0.5,random\_state=42)

elastic\_net.fit(X,y)

elastic\_net.predict([[1.5]])

* Regular Linear Regression in general should be avoided. Ridge is good default. But if only few features are useful, then either Lasso or Elastic Net should be used. Preferable, Elastic Net should be used.
* **Early Stopping :** To stop the regularize iterative algorithm as soon as validation error reaches a minimum.

np.random.seed(42)

m = 100

X = 6 \* np.random.rand(m, 1) - 3

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

X\_train,X\_val,y\_train,y\_val = train\_test\_split(X[:50], y[:50].ravel(), test\_size=0.5, random\_state=10)

poly\_scaler = Pipeline([

(‘poly\_features’,PolynomialFeatures(degree=90,include\_bias=False)),

(‘std\_scaler’,StandardScaler()),

])

X\_train\_poly\_scaled = poly\_scaler.fit\_transform(X\_train)

X\_val\_poly\_scaled = poly\_scaler.transform(X\_val)

Sgd\_reg = SGDRegressor(max\_iter = 1,

penalty = None,

eta0 = 0.0005,

warm\_start = True,

learning\_rate = ‘constant’,

random\_state = 42)

n\_epochs = 500

train\_errors,val\_errors = [],[]

for epoch in range(n\_epochs):

sgd\_reg.fit(X\_train\_poly\_scaled,y\_train)

y\_train\_predict = sgd\_reg.predict(X\_train\_poly\_scaled)

y\_val\_predict = sgd\_reg.predict(X\_val\_poly\_scaled)

train\_errors.append(mean\_square\_error(y\_train\_predict,y\_train))

val\_errors.append(mean\_square\_error(y\_val\_predict,y\_val))

best\_epoch = np.argmin(val\_errors)

best\_val\_rmse = np.sqrt(val\_errors[best\_epoch])

plt.annotate ( ‘Best model’,

xy = (best\_epoch,best\_val\_rmse),

xytest = (best\_epoch,best\_val\_rmse + 1),

ha= ‘center’

arrowprops=dict(facecolor=’black’,shrink=0.05),

frontsize(16),

)

best\_val\_rmse -= 0.03 *# just to make the graph look better*

plt.plot([0, n\_epochs], [best\_val\_rmse, best\_val\_rmse], "k:", linewidth=2)

plt.plot(np.sqrt(val\_errors), "b-", linewidth=3, label="Validation set")

plt.plot(np.sqrt(train\_errors), "r--", linewidth=2, label="Training set")

plt.legend(loc="upper right", fontsize=14)

plt.xlabel("Epoch", fontsize=14)

plt.ylabel("RMSE", fontsize=14)

plt.show()

* **Stops only after the validation error has been above for some time, then rollback the model parameter to that point where error was minimum**

**# Basic implementation of early stopping with warm\_start=True, when the fit() method is called, it just continues training where it left off instead of restarting from scratch**

from sklearn.base import clone **# deep copy of the model in an estimator without actually copying attached data**

sgd\_reg = SGDRegressor(max\_iter=1,warm\_start=True,penalty=None,learning\_rate=’constant’,eta0=0.0005,random\_state=42)

minimum\_val\_error = float(‘inf’)

best\_epcoh = None

best\_model = None

for epoch in range(1000):

sgd\_reg.fit(X\_train\_poly\_scaled,y\_train) **#Continues where it left off**

y\_val\_predict = sgd\_reg.predict(X\_val\_poly\_scaled)

val\_error = mean\_squared\_error(y\_val\_predict,y\_val)

if val\_error < minimum\_val\_error:

minimum\_val\_error = val\_error

best\_epoch = epoch

best\_model = clone(sgd\_reg)

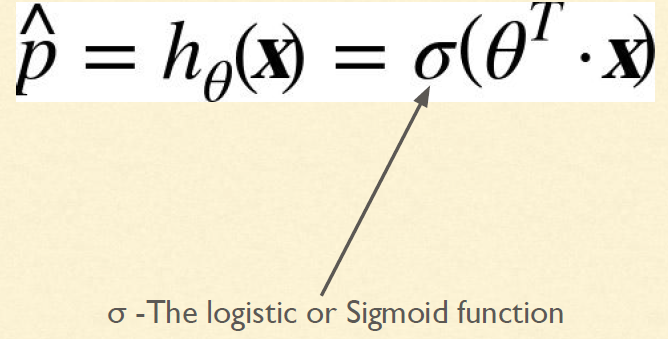
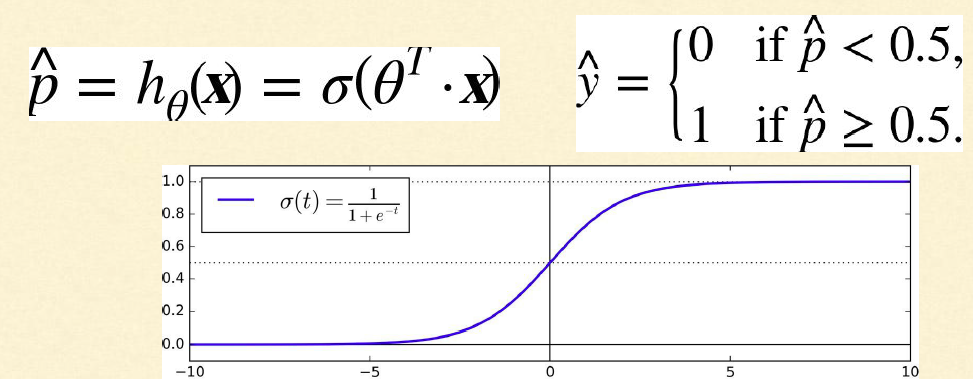
last\_increase = 0

else :

break

# Logistic Regression

* Binary Classifier that estimates the probability that a particular instance belongs to which class. Ex: Email Spam Classifier
* Like Linear Regression, it computes the weighted sum of the input features plus a bias term.

***# Plot Sigmoid function***

t = np.linspace(-10, 10, 100)

sig = 1 / (1 + np.exp(-t))

plt.figure(figsize=(9, 3))

plt.plot([-10, 10], [0, 0], "k-")

plt.plot([-10, 10], [0.5, 0.5], "k:")

plt.plot([-10, 10], [1, 1], "k:")

plt.plot([0, 0], [-1.1, 1.1], "k-")

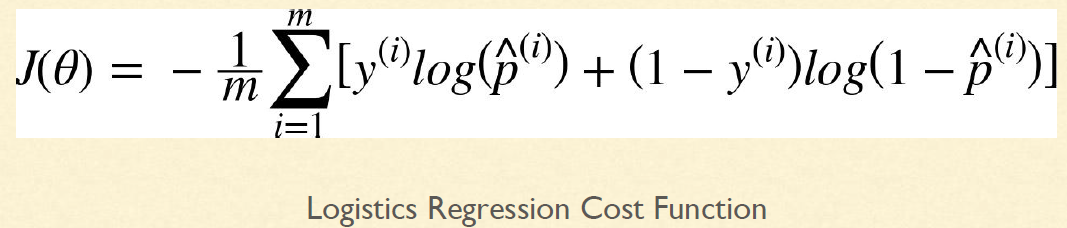
plt.plot(t, sig, "b-", linewidth=2, label=r"$\sigma(t) = \frac**{1}**{1 + e^{-t}}$")

plt.xlabel("t")

plt.legend(loc="upper left", fontsize=20)

plt.axis([-10, 10, -0.1, 1.1])

plt.show()



* **Decision Boundaries**

from sklearn import datasets

iris = datasets.load\_iris()

list(iris.keys())

**# Train a Logistic Regression model**

**# To detect the Iris-Virginica type based only on the petal width feature**

import numpy as np

X = iris[“data”][:,3:] **#petal width**

y = (iris[‘target’] == 2).astype(np.int)

from sklearn.linear\_model import LogisticRegression

log\_reg = LogisticRegression(random\_state=42)

log\_reg.fit(X,y)

log\_reg.predict([[1.0]])

***# Model’s estimated probabilities for flowers with petal widths varying from 0 to 3 cm***

X\_new = np.linspace(0, 3, 1000).reshape(-1, 1)

y\_proba = log\_reg.predict\_proba(X\_new)

decision\_boundary = X\_new[y\_proba[:, 1] >= 0.5][0]

plt.figure(figsize=(8, 3))

plt.plot(X[y==0], y[y==0], "bs")

plt.plot(X[y==1], y[y==1], "g^")

plt.plot([decision\_boundary, decision\_boundary], [-1, 2], "k:", linewidth=2)

plt.plot(X\_new, y\_proba[:, 1], "g-", linewidth=2, label="Iris-Virginica")

plt.plot(X\_new, y\_proba[:, 0], "b--", linewidth=2, label="Not Iris-Virginica")

plt.text(decision\_boundary+0.02, 0.15, "Decision boundary", fontsize=14, color="k", ha="center")

plt.arrow(decision\_boundary, 0.08, -0.3, 0, head\_width=0.05, head\_length=0.1, fc='b', ec='b')

plt.arrow(decision\_boundary, 0.92, 0.3, 0, head\_width=0.05, head\_length=0.1, fc='g', ec='g')

plt.xlabel("Petal width (cm)", fontsize=14)

plt.ylabel("Probability", fontsize=14)

plt.legend(loc="center left", fontsize=14)

plt.axis([0, 3, -0.02, 1.02])

plt.show()

*# Train a Logistic Regression model*

*# To detect the Iris-Virginica type based on the petal width and length features*

**from** **sklearn.linear\_model** **import** LogisticRegression

X = iris["data"][:, (2, 3)] *# petal length, petal width*

y = (iris["target"] == 2).astype(np.int)

log\_reg = LogisticRegression(C=10\*\*10, random\_state=42)

log\_reg.fit(X, y)

x0, x1 = np.meshgrid(

np.linspace(2.9, 7, 500).reshape(-1, 1),

np.linspace(0.8, 2.7, 200).reshape(-1, 1),

)

X\_new = np.c\_[x0.ravel(), x1.ravel()]

y\_proba = log\_reg.predict\_proba(X\_new)

plt.figure(figsize=(10, 4))

plt.plot(X[y==0, 0], X[y==0, 1], "bs")

plt.plot(X[y==1, 0], X[y==1, 1], "g^")

zz = y\_proba[:, 1].reshape(x0.shape)

contour = plt.contour(x0, x1, zz, cmap=plt.cm.brg)

left\_right = np.array([2.9, 7])

boundary = -(log\_reg.coef\_[0][0] \* left\_right + log\_reg.intercept\_[0]) / log\_reg.coef\_[0][1]

plt.clabel(contour, inline=1, fontsize=12)

plt.plot(left\_right, boundary, "k--", linewidth=3)

plt.text(3.5, 1.5, "Not Iris-Virginica", fontsize=14, color="b", ha="center")

plt.text(6.5, 2.3, "Iris-Virginica", fontsize=14, color="g", ha="center")

plt.xlabel("Petal length", fontsize=14)

plt.ylabel("Petal width", fontsize=14)

plt.axis([2.9, 7, 0.8, 2.7])

plt.show()

* Like Linear Models, Logistic Regression models can be regularized using L1 & L2(Default) Penalty.

# Softmax/Multinomial Logistic Regression

* Can be generalized to support multiple classes without have to train and combine multiple binary classifiers.
* Softmax function calculates the probabilities of each target classes. And, return Probability of each Class & Target Class will have maximum probability.
* lbfgs - Broyden–Fletcher–Goldfarb–Shanno (BFGS) algorithm is an iterative method for solving unconstrained nonlinear optimization problems
* C - Hyperparameter for regularizing L2

**# Softmax Regression**

X = iris["data"][:, (2, 3)] **# petal length, petal width**

y = iris["target"]

softmax\_reg = LogisticRegression(multi\_class=’multinimial’,solver=’lbfgs’,c=10,random\_state=42)

softmax\_reg.fit(X,y)

y\_pred = softmax\_reg.predict(X)

softmax\_reg.predict\_proba([[5,2]])

**#Confusion Matrix**

from sklearn.metrics import confusion\_matrix

confusion\_matrix (y,y\_pred)