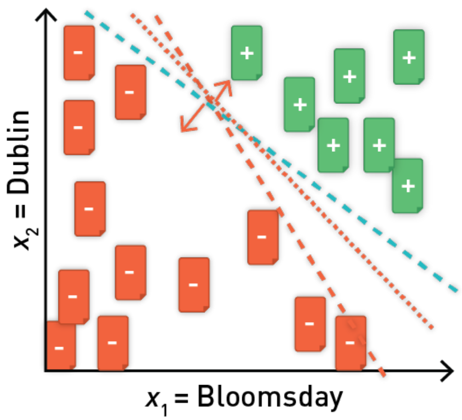
Linear Discriminant Model

Training Data

E.g.	x_1	y
1	3	−1
2		+1
...	...	...
L	0	−1

Many hyperplanes exist.



A linear discriminant function is linear in the components of *X*.

$$y = ax_1 + b = 0$$

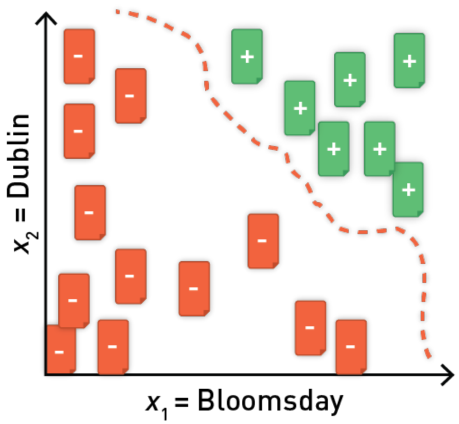
$$f(X) = \begin{cases} -1 & if \;\; y < 0 \\ 0 & if \;\; y = 0 \\ 1 & if \;\; y > 0 \end{cases}$$

$$Class(X) = sign(< W, X > + b)$$

Nonlinear Discriminant Model

Training Data

E.g.	x_1	x_2	y
1	3	0	–1
2			+1
…	…	…	…
L	0	4	–1



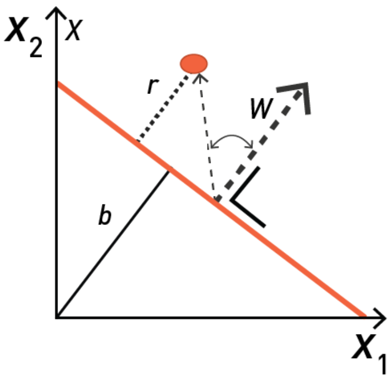
A nonlinear discriminant function is nonlinear in the components of X.

$$y = ax_1^2 + bx_2^3 + c = 0$$

$$f(X) = \begin{cases} -1 & if\ \ y < 0 \\ 0 & if\ \ y = 0 \\ 1 & if\ \ y > 0 \end{cases}$$

$$Class(X) = sign(< W, X > + b)$$

Geometry: Linear Separators



$w_1x_1 + w_2x_2 + \ \dots\ + b = 0$	**Basic hyperplane**
$\sum_{i=1}^{n} w_ix_i + b = 0$	**More general**
$WX + b = 0$	**Dot product**

For a plane given by the equation *ax* + *by* + *cz* = *d*, the vector (*a*, *b*, *c*) is a normal.

Represent a hyperplane, *H*, in terms of vector *W* and scalar *b*.

- *W* determines the orientation of the hyperplane or discriminant plane.
- *b* denotes the offset (perpendicular distance) from plane to origin.

$$r = \frac{W^TX + b}{\sqrt{w_1^2 + \ \dots\ w_n^2}}$$

Perpendicular distance from point *X* to a hyperplane

Empirical Risk Minimization

- This provides a criterion to decide on *h*.

$$\min_{h \in \mathcal{H}} \frac{1}{N} \sum_{i=1}^{N} loss(\boldsymbol{x}_i, \boldsymbol{y}_i; h)$$

- Background preferences over *h* can be included in regularized empirical risk minimization.

$$\min_{h \in \mathcal{H}} \frac{1}{N} \sum_{i=1}^{N} loss(\boldsymbol{x}_i, \boldsymbol{y}_i; h) + R(h)$$

Loss Terms: Zero-One and Hinge Loss

- **Zero-one loss**, or L_{01}, counts the number of mistakes (gold standard).

$$L_{01}(m) = \begin{cases} 0 & \text{if } m \geq 0 \\ 1 & \text{if } m < 0 \end{cases}$$

- **Hinge loss**, or L_{hinge}: Loss term for soft-margin SVM

$$\begin{aligned} J(w) &= \frac{1}{2} \parallel w \parallel^2 + \sum_i \max(0, 1 - y^i w^T x^i) \\ &= \frac{1}{2} \parallel w \parallel^2 + \sum_i \max(0, 1 - m_i(w)) \\ &= R_2(w) + \sum_i L_{hinge}(m_i) \end{aligned}$$

Loss Terms: Log Loss

- This is equivalent to the cross-entropy loss function used to train a logistic regression model.

$$J(w) = \lambda \parallel w \parallel^2 + \sum_i y^i \log g_w(x^{(i)}) + (1 - y^i)(\log 1 - g(x^{(i)})), y^i \in (0, 1)$$

- Simplify log conditional likelihood to get a more succinct loss component.

$$J(w) = \lambda \parallel w \parallel^2 + \sum_i \log 1 + e^{-\bar{y^{(i)}} f_w(x^{(i)})}$$

- Where *m* is defined:

$$L(m) = \log 1 + e^{-m}$$

$$m^i = \bar{y^{(i)}} f_w(x^{(i)})$$

$$\bar{y^{(i)}} = \begin{cases} -1 & \text{if } y^{(i)} = 0 \\ 1 & \text{if } y^{(i)} = 1 \end{cases}$$

Other Loss Terms

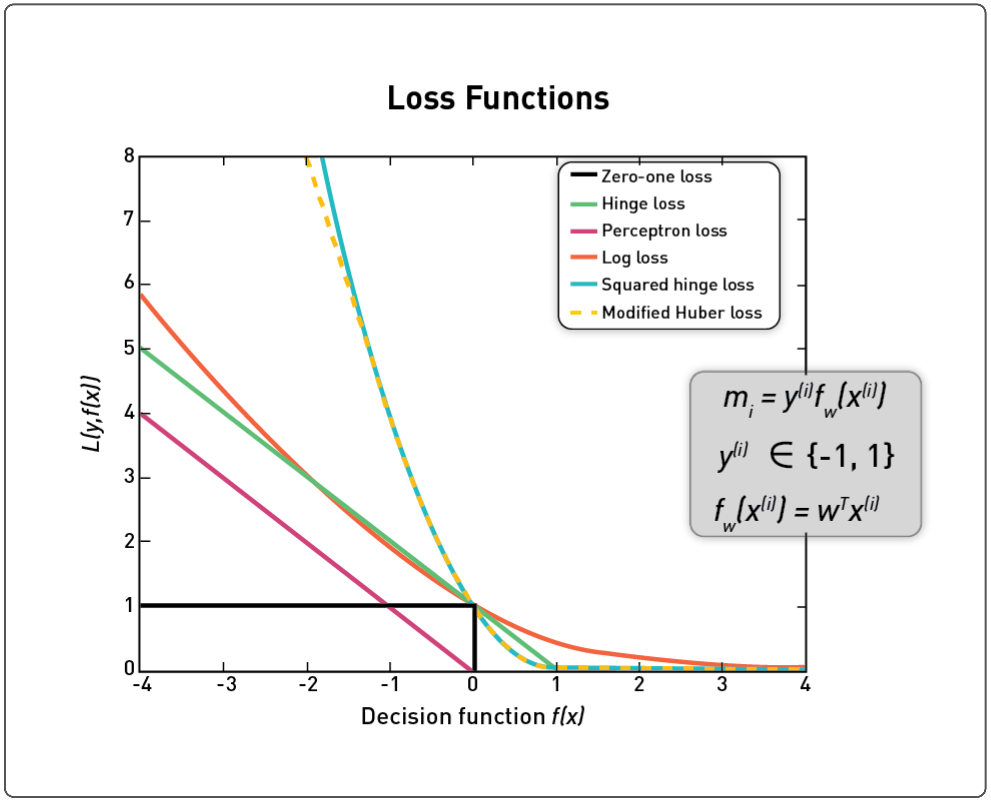
- **Squared-error:** For linear regression

$$L_2(m) = (f_w(x) - y)^2 = (m-1)^2$$

- **Boosting:** A greedy optimization of the exponential loss term

$$J(w) = \lambda R(w) + \sum_i \exp(-y^{(i)} f_w(x^{(i)}))$$

$$L_{exp}(m_i) = \exp(-m_i(w))$$



Loss Functions
Zero-one loss
Hinge loss
Perceptron loss
Log loss
Squared hinge loss
Modified Huber loss
$m_i = y^{(i)} f_w(x^{(i)})$
$y^{(i)} \in \{-1, 1\}$
$f_w(x^{(i)}) = w^T x^{(i)}$
L(y,f(x))
Decision function f(x)

Loss Functions: A Unifying View

Loss function consists of:

- **Loss term** Lm_iw, expressed in terms of the margin of each training example
- **Regularization term** λRw, expressed as a function of the model complexity

$$J(w) = \sum_i \underbrace{\boxed{L\big(m_i(w)\big)}}_{\text{Loss term}} + \underbrace{\boxed{\lambda R(w)}}_{\text{Regularization term}}$$

$$m_i = y^{(i)} f_w\big(x^{(i)}\big)$$

$$y^{(i)} \in \{-1, 1\}$$

$$f_w\big(x^{(i)}\big) = w^T x^{(i)}$$

Machine Learning Objectives

Objective function:

$$J(w) = \sum_i \underbrace{\boxed{L\big(m_i(w)\big)}}_{\text{Loss term}} + \underbrace{\boxed{\lambda R(w)}}_{\text{Regularization term}}$$

Almost all machine learning objectives are optimized using this update:

$$w \leftarrow w - \alpha \cdot \sum_{i=1}^{n} g(J(w)x_i, y_i)$$

Update weight vector with gradient of the objective function.

- *w* is a vector of dimension *d*.
- We're trying to find the best *w* via optimization.

Distributed Gradient Descent

- Ordinary least squares (OLS)
- Logistic regression
- Bayesian logistic regression
- Perceptron
- SVMs and their many learning algorithms

Distributed Gradient Descent: Linear Regression Example

Master-Slave Process

- Initialize model parameters; assume a weight vector *W* = (0, 0, … , 0); Gradient = (0, 0, … , 0)
- While not converged
  - MASTER: Broadcast model (i.e, weight vector) to the worker nodes
  - MASTER launches MapReduce jobs
    - Mappers (*many* mappers) to compute partial gradients over the respective training data subsets (chunks)

      Init *g* = (0, 0, … , 0).

      For each training example:

      Compute partial gradient for each training example.

      Combine in memory: $g = \sum_i gW;X_i,Y_i$.

      Finally yield the partial gradient *g*.

      Linear regression
      Goal: Learn a weight vector *W*.
      Gradient is defined here:
      $gW;X_i,Y_i = y^i - W_i X^i X^i$

    - Reducer (single reducer)

      Initialize full gradient: *G* = (0, 0, … , 0).

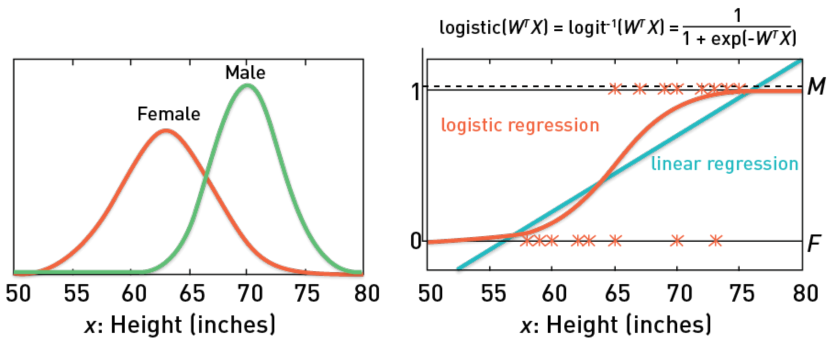
      For each partial gradient *g*:

      Aggregate partial gradients: $g = \sum_m g_m$.

      Yield full gradient *G*.
  - MASTER *W* = *W* +α*G* // update the weight vector
- End-While

Logistic Regression in One Slide

Example: Predict the gender (y = M/F) of a person given their height (x = a number).



$$py = Mx = \beta_0 + \beta_x \text{ (linear regression)}$$

$$\ln \frac{py = Mx}{py = Fx} = \beta_0 + \beta_x \text{ (logistic regression)}$$

Assumptions

- $y \in \{-1, +1\}$, x is feature vector, and p is defined as:

***p* = Pr(*y* = +1 | *x*)**

- Linearly related to features after logit transformation:

$$\log \frac{p}{1-p} = w^T \cdot x - b$$

- Then probability is a logistic function of **w · x**:

$$p = \frac{1}{1 + \exp -w^T \cdot x + b}$$

- Where:
  - w is the coefficient vector.
  - b is the intercept.

Maximum Likelihood Expectation (MLE)

Maximize the log likelihood:

$$I(W) = \ln \prod_i P_i$$

$$= \ln \prod_i \left(\frac{1}{1+\exp\left(-\mathbf{w}^T \cdot \mathbf{x}_i + b\right)}\right)^{\frac{1+y_i}{2}} \left(1 - \frac{1}{1+\exp\left(-\mathbf{w}^T \cdot \mathbf{x}_i + b\right)}\right)^{\frac{1-y_i}{2}}$$

Maximizing this log likelihood is equal to minimizing the following:

$$I(W) = \sum_i \log(1 + \exp\left(-y\mathbf{w}^T \mathbf{x}_i\right))$$

$$L(m) = \log 1 + e^{-m}$$

$$m^i = y^{-(i)} f_w(x^{(i)})$$

- Minimize the **NEG** log joint conditional likelihood.
- The conditional likelihood θ given data x and y is $L(\theta; y|x) = p(y|x) = f(y|x; \theta)$.

Estimating Parameters Using Gradient Descent

- No closed-form solution to maximizing $l(W)$ with respect to W
- One common approach: Use gradient descent, working with the gradient, which is the vector of partial derivatives

Estimating Parameters Using Gradient Descent (cont.)

The *i*th component of the vector gradient has the form:

$$I\left(W\right) = \sum_{i} Y^{l}\left(w_0 + \sum_{i}^{n} w_i X_i^l\right) - \ln\left(1 + \exp\left(w_0 + \sum_{i}^{n} w_i X_i^l\right)\right)$$

$$\frac{\partial l(W)}{\partial w_i} = \sum_{l} X_i^l \left(Y^l - \hat{P}\left(Y^l = 1 \middle| X^l, W\right)\right)$$

$$w_i \leftarrow w_i + \eta \sum_{l} X_i^l \left(Y^l - \hat{P}\left(Y^l = 1 \middle| X^l, W\right)\right)$$

- Beginning with initial weights of zero, we repeatedly update the weights in the direction of the gradient, changing the *i*th weight according to this formula, where η is a small constant (e.g., 0.01), which determines the step size.
- Effectively, we are pulling the weight vector closer to the examples where we make mistakes.

Gradient Descent

- Objective function (logloss):

$$\sum_i \log\left(1 + \exp\left(-y\left(\mathbf{w}^T\mathbf{x}_i + b\right)\right)\right)$$

- Gradient:

$$\nabla\mathbf{w} = \sum_i -y\left(1 - \frac{1}{1+\exp\left(-y\left(\mathbf{w}^T\mathbf{x}_i+b\right)\right)}\right)\cdot\mathbf{x}_i$$

- Update *w* until it converges:

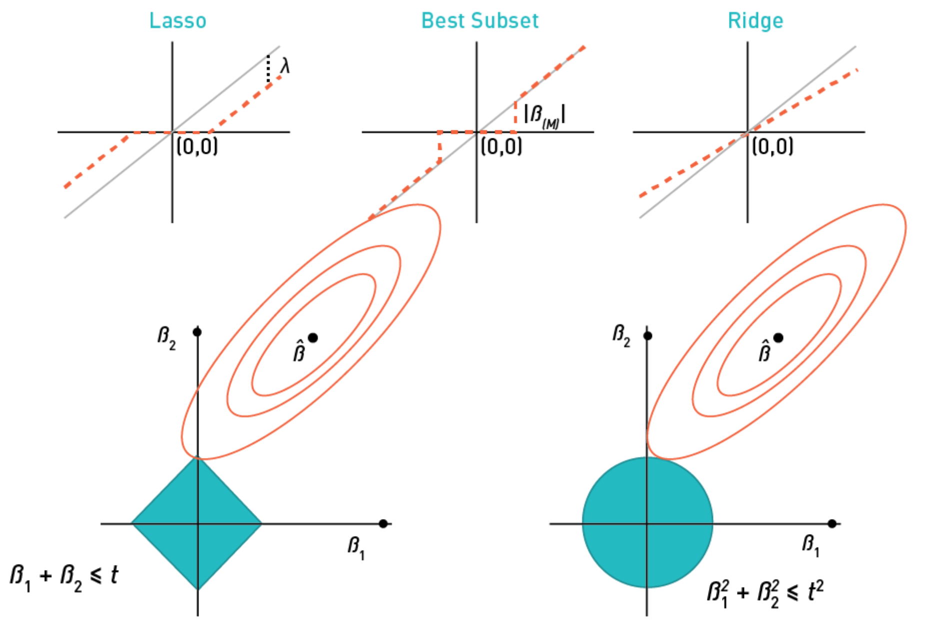
$$\mathbf{w} = \mathbf{w} - \eta \cdot \nabla\mathbf{w}$$

Regularization

Shrinkage of *w*

- **Lasso:** L1 norm regularization, |*w*|
- **Ridge:** L2 norm regularization, w^2

Add regularization to prevent overfitting.



Estimation Picture for Lasso and Ridge (contours of error and constraint functions)

Gradient Descent: Lasso

- Objective function (logloss and lassreg):

$$\sum_i \log\left(1+\exp\left(-y\left(\mathbf{w}^T\mathbf{x}_i+b\right)\right)\right)+\lambda\left|\mathbf{w}\right|$$

- Gradient:

$$\nabla\mathbf{w}=\sum_i -y\left(1-\frac{1}{1+\exp\left(-y\left(\mathbf{w}^T\mathbf{x}_i+b\right)\right)}\right)\cdot\mathbf{x}_i+\lambda\left(\mathbf{1}_{>0}\left(\mathbf{w}\right)\cdot 2-1\right)$$

- Update *w* until it converges:

$$\mathbf{w}=\mathbf{w}-\eta\cdot\nabla\mathbf{w}$$

Gradient Descent: Ridge

- Objective function (logloss and ridgereg):

$$\sum_i \log\left(1+\exp\left(-y\left(\mathbf{w}^T\mathbf{x}_i+b\right)\right)\right)+\lambda\mathbf{w}^2$$

- Gradient:

$$\nabla\mathbf{w}=\sum_i -y\left(1-\frac{1}{1+\exp\left(-y\left(\mathbf{w}^T\mathbf{x}_i+b\right)\right)}\right)\cdot\mathbf{x}_i+\lambda\mathbf{w}$$

- Update *w* until it converges:

$$\mathbf{w}=\mathbf{w}-\eta\cdot\nabla\mathbf{w}$$

Python Notebooks for Logistic Regression

- Notebook for logistic regression on a single node: http://nbviewer.ipython.org/urls/dl.dropbox.com/s/kurbw695jdvxib0/LogisticRegression-Single-Core-Notebook-.ipynb
- Notebook for distributed logistic regression on Spark: http://nbviewer.ipython.org/urls/dl.dropbox.com/s/r20ff7q0yni5kiu/LogisticRegression-Spark-Notebook.ipynb

Logistic Regression: Single-Node Code

Python—Gradient Descent

Input:

- data: feature information
- *y*: label information
- eta: learning rate
- iter_num: maximum iteration number
- regPara: regularization parameter
- stopCriteria: stop criteria

Output

- *w*: coefficients of boundary

[LogisticRegression-Single-Core-Notebook-.ipynb]

Note: This is the single-node implementation of logistic regression. The distributed version will follow momentarily.

Please click on the link and take some time out to review this notebook.

```
# gradient descent (and with NO stochasticity!)
# Objective Function
# minw   λ/2   w'w   +   1/m ∑i log(1+exp(yi(w'xi - b)))
# gradient
#   -y*(1-1/(1+exp(yi(w'xi - b))))*x

def
logisticReg_GD(data,y,w=None,eta=0.05,iter_num=500,regPara=0.01,stopCriteria=
0.0001,reg="Ridge"):
    data = np.append(data,np.ones((data.shape[0],1)),axis=1)
    if w is None:
       w = np.random.normal(size=data.shape[1])
    for i in range(iter_num):
       wxy = np.dot(data,w)*y
       g = np.dot(data.T,-y*)1-1/(1+np.exp(-wxy))))/data.shape[0]
       #Gradient of log loss
       wreg = w*1
       wreg[-1] = 0 #last value of weight vector is bias term;
       ignore in regularization
       if reg == "Ridge":
          wreg = w*1
          wreg[-1] = 0 #last value of weight vector is bias term;
          ignore in regularization
       elif reg == "Lasso":
          wreg = w*1
          wreg[-1] = 0 #last value of weight vector is bias term;
          ignore in regularization
          wreg = (wreg>0).astype(int)*2-1
       else:
          wreg = np.zeros(w.shape[0])
       wdelta = eta*(g+regPara*wreg) #gradient: log loss + regularized term
       if sum(abs(wdelta))<=stopCriteria*sum(abs(w)): #Convergence condition
          break
       w = w - wdelta
    return w
```

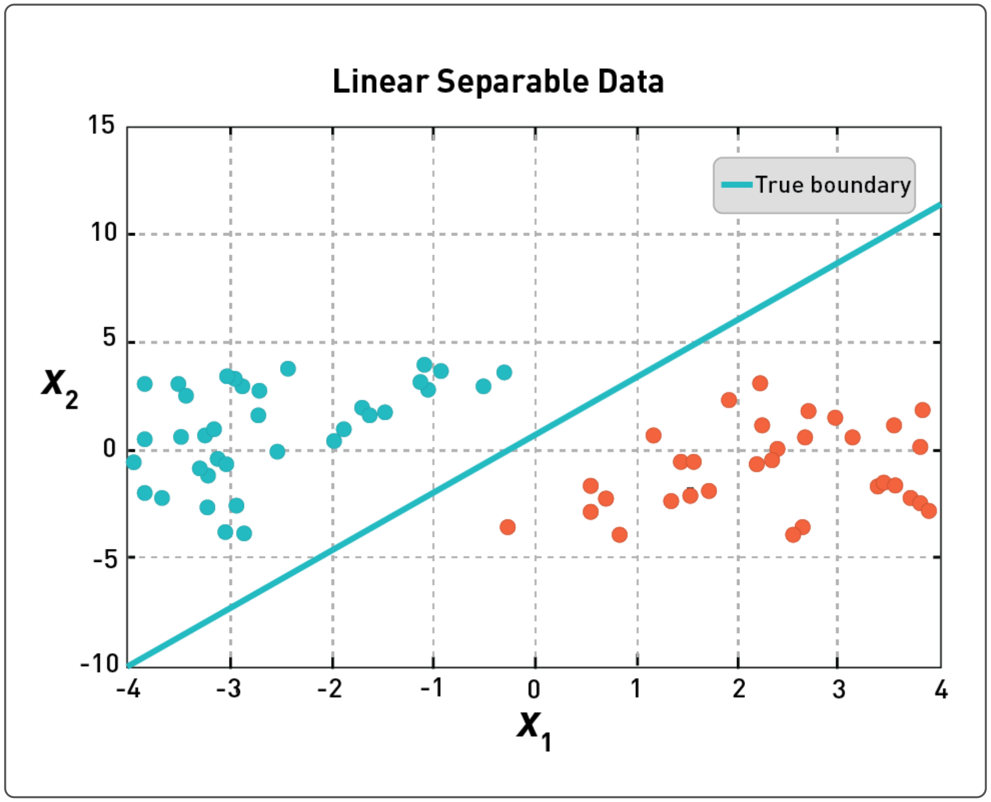
Distributed Logistic Regression: PySpark Code

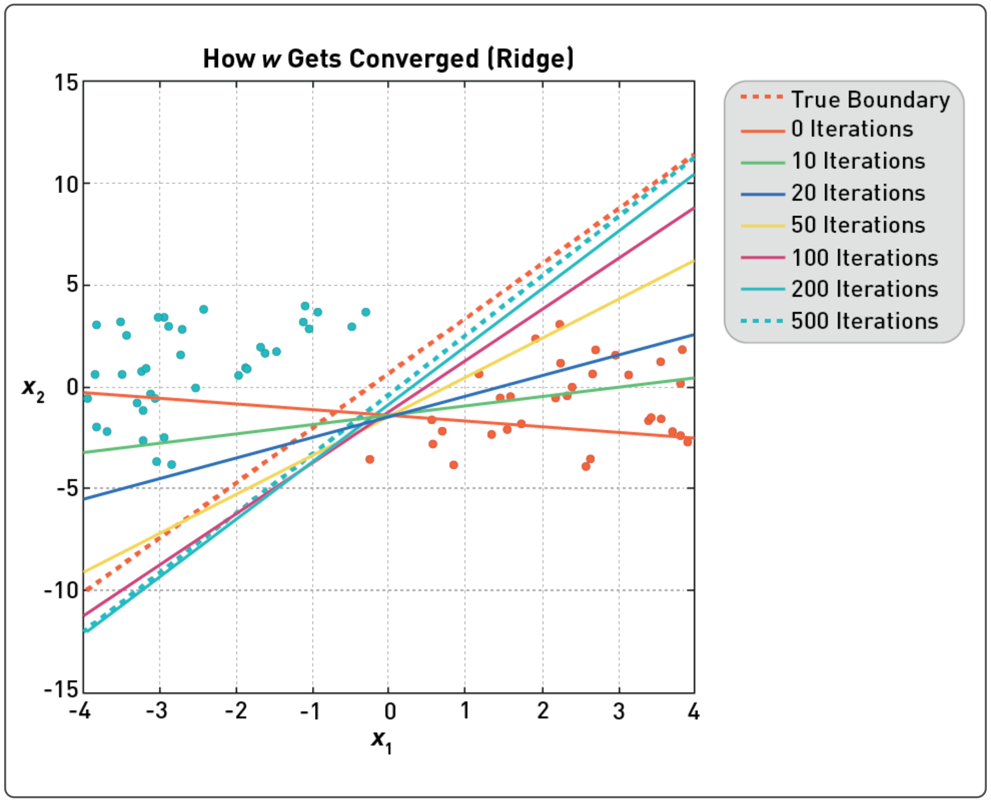
LogisticRegression-Spark-Notebook.ipynb

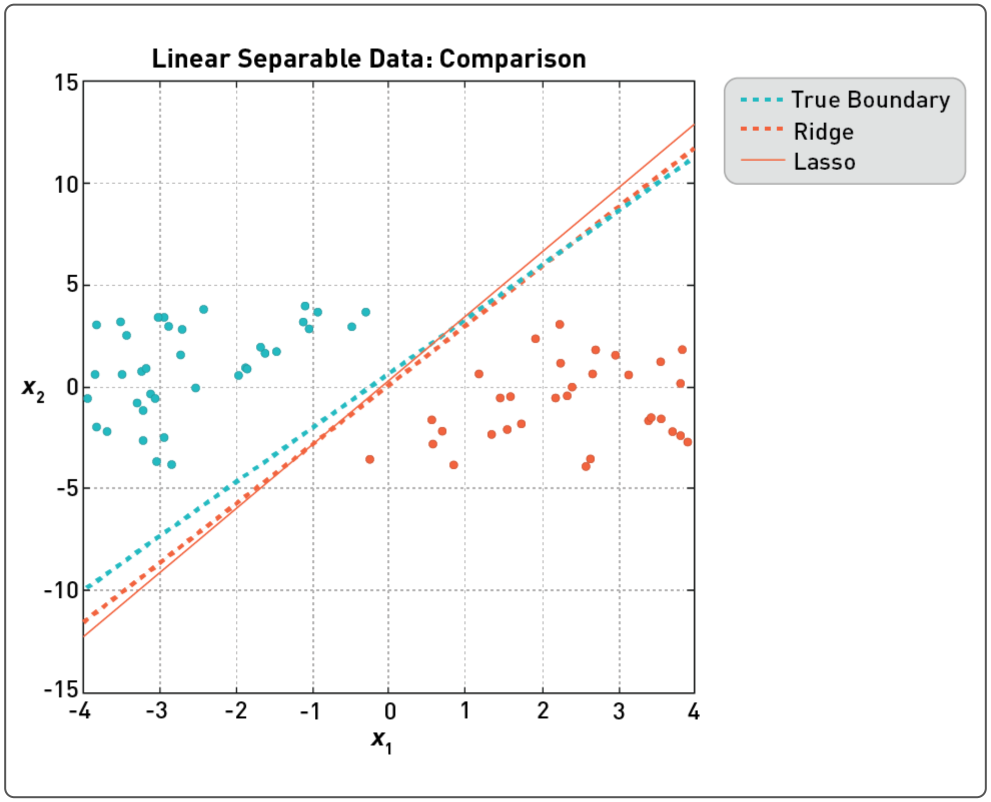
```
# gradient descent (and with NO stochasticity!)
# Objective Function
# minw   λ/2   w'w   +   1/m ∑i log(1+exp(yi(w'xi − b)))
# gradient
#   −y*(1−1/(1+exp(yi(w'xi − b))))*x

def logisticReg_GD_Spark(data,y,w=None,eta=0.05,iter_num=500,regPara=0.01,
stopCriteria=0.0001,reg="Ridge"):
    #eta learning rate
    #regPara
    dataRDD = sc.parallelize(np.append(y[:,None],data,axis=1)).cache()
    if w is None:
       w = np.random.normal(size=data.shape[1]+1)
    for i in range(iter_num):
       w_broadcast = sc.broadcast(w)
       g = dataRDD.map(lambda x: -x[0]*(1-1/(1+np.exp(-x[0]
       *np.dot(w_broadcast.value,np.append(x[1:],1))))) \
             *np.append(x[1:],1)).reduce(lambda x,y:x+y)/data.shape[0]
             # Gradient of logloss
       if reg == "Ridge":
          wreg = w*1
          wreg[-1] = 0 #last value of weight vector is bias term;
          ignore in regularization
       elif reg == "Lasso":
          wreg = w*1
          wreg[-1] = 0 #last value of weight vector is bias term;
          ignore in regularization
          wreg = (wreg>0).astype(int)*2-1
       else:
          wreg = np.zeros(w.shape[0])
       wdelta = eta*(g+regPara*wreg) #gradient: hinge loss + regularized term
       if sum(abs(wdelta))<=stopCriteria*sum(abs(w)): # converged as updates
       to weight vector are small
          break
       w = w - wdelta
    return w
```

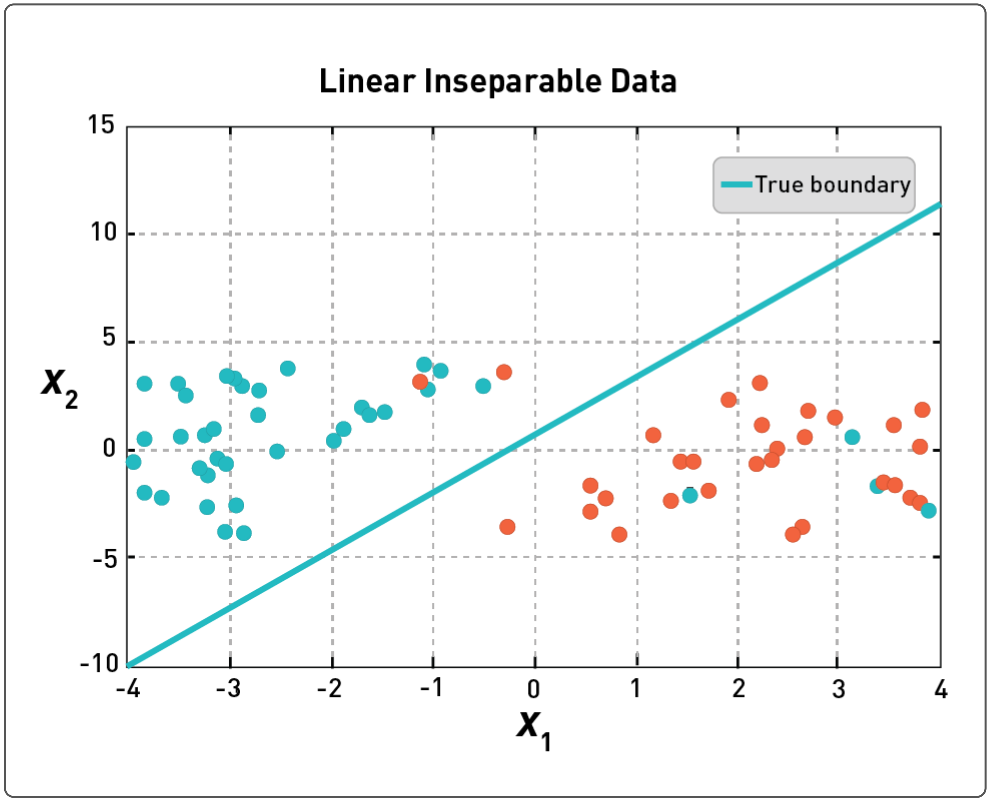
How to Run the Code

- **Ridge logistic regression**
  - ```
    np.random.seed(400)
    w_gd_spark_sep_ridge =
    logisticReg_GD_Spark(data_sep,y_sep,reg="Ridge")
    print w_gd_spark_sep_ridge
    ```
  - `[ 1.84106359 -0.63166074   0.00472968]`
- **Lasso logistic regression**
  - ```
    np.random.seed(400)
    w_gd_spark_sep_lasso =
    logisticReg_GD_Spark(data_sep,y_sep,reg="Lasso")
    print w_gd_spark_sep_lasso
    ```
  - `[ 1.90212754 -0.60763246   0.18656826]`


Linear Separable Data
True boundary
15
10
5
0
-5
-10
-4
-3
-2
-1
0
1
2
3
4
x_2
x_1

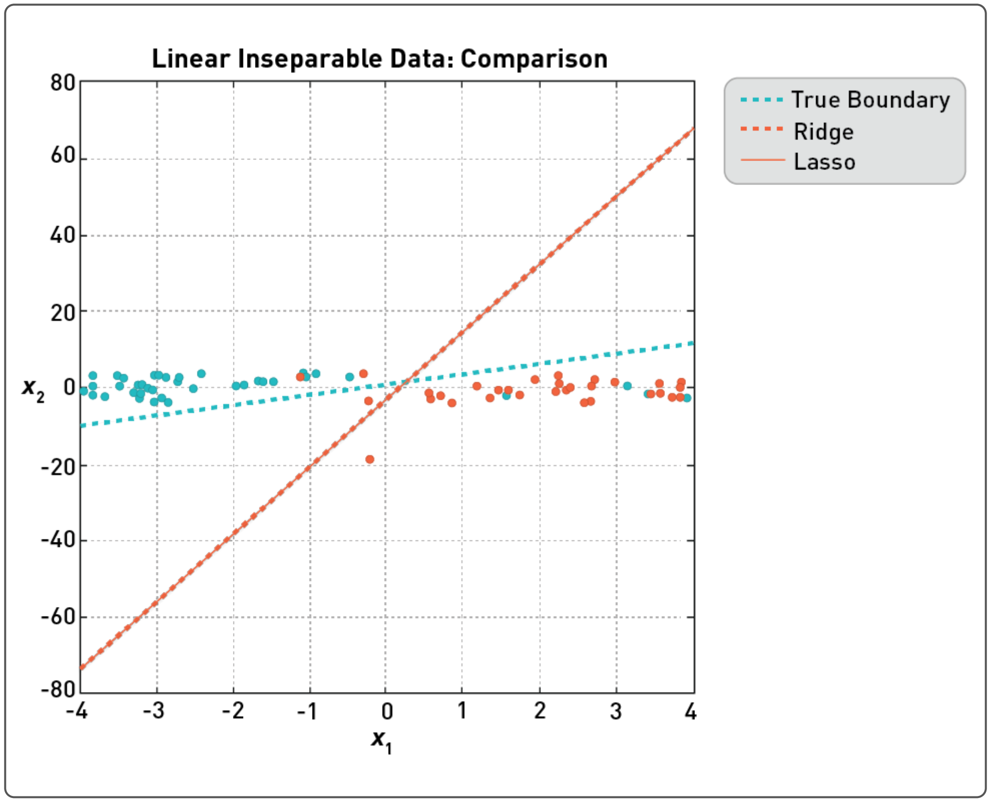

How w Gets Converged (Ridge)
True Boundary
0 Iterations
10 Iterations
20 Iterations
50 Iterations
100 Iterations
200 Iterations
500 Iterations
x1
x2


Linear Separable Data: Comparison
True Boundary
Ridge
Lasso
15
10
5
0
-5
-10
-15
-4
-3
-2
-1
0
1
2
3
4
x_1
x_2


Linear Inseparable Data
True boundary
15
10
5
0
-5
-10
-4
-3
-2
-1
0
1
2
3
4
x_2
x_1

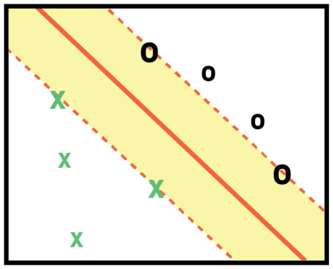
Linear Inseparable Data: Code

- **Ridge logistic regression**
  - ```
    np.random.seed(400)
    w_gd_insep_ridge = logisticReg_GD(data_insep,y_insep,reg="Ridge")
    print w_gd_insep_ridge
    ```
  - `[ 0.88960208 -0.06413379   -0.24526051]`
- **Lasso logistic regression**
  - ```
    np.random.seed(400)
    w_gd_insep_lasso = logisticReg_GD(data_insep,y_insep,reg="Lasso")
    print w_gd_insep_lasso
    ```
  - `[ 0.88745901 -0.0500822   -0.14752302]`


Linear Inseparable Data: Comparison
True Boundary
Ridge
Lasso
80
60
40
20
0
-20
-40
-60
-80
-4
-3
-2
-1
0
1
2
3
4
x_1
x_2

Find Shortest Weight Vector

- Finding a separating hyperplane with the largest margin is equivalent to finding such a hyperplane with minimal *W*.
- Solve using optimization approaches.



Max $\frac{2}{\|w\|^2}$

Subject to constraints

$y_i w \cdot x_i + b - 1 \geq 0$ **∀ *i* = 1, ... , *L***

or

Min $\frac{\|w\|^2}{2}$

Subject to constraints

$y_i w \cdot x_i + b - 1 \geq 0$ **∀ *i* = 1, ... , *L***

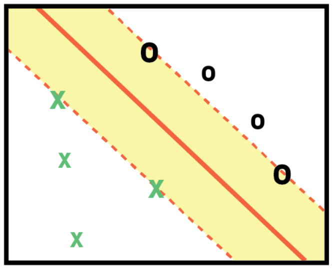
Hard SVMs vs. Soft SVMs

Hard SVM learning algorithm

Must classify each example correctly

$$\min Wb = 0.5 \times \left\| w \right\|^2$$

Subject to:
$y_i W X_i + b \geq 1 \quad \forall\, i = 1, \ldots, L$



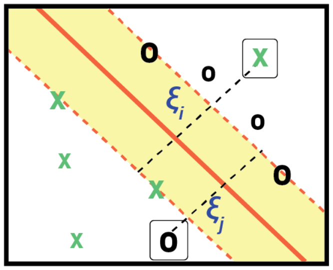
Soft SVM learning algorithm

Has trade-off between margin and error

Associates a slack variable ξ_i with each example

$$\min Wb = 0.5 \times \left\| w \right\|^2 + C \sum_{i=1}^{L} \xi_i$$

Subject to:
$y_i W X_i + b + \xi_i \geq 1 \quad \forall\, i = 1, \ldots, L$
and $\xi_i \geq 0 \quad \forall\, i = 1, \ldots, L$



Kernels

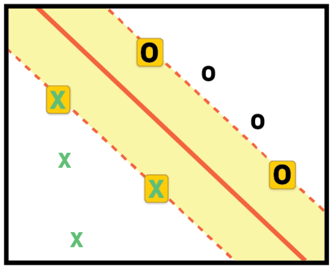
- Linear
- Polynomial
- Radial basis function

Primal and Dual Learning

- **Primal learning** (e.g., perceptron) involves learning weight values associated with each term or feature.
- **Dual learning** involves learning weight values associated with each example so that the margin, λ, is maximum.

Dual Learning

Learning weight values associated with each example so that the margin, λ, is maximum



	Wgt Vector	w_0	w_1	...	...	w_n	
α	**Instance/ Attr**	x_0	x_1	x_2	...	x_n	***y***
1.2	1	1	3	0	...	7	–1
0	2	1			...		+1
⋮	⋮	⋮	⋮	⋮	...	⋮	⋮
4.2	*L*	1	0	4	...	8	–1

Dual Learning (cont.)

The dual problem is solved for multipliers, not for *w* and *b*.

- *W* is then obtained as a linear combination of data vectors, by the constraint:
  - $w - \sum_i \alpha_i y_i x_i = 0$
- *b* can be found by complementary Karush-Kuhn-Tucker (KKT) condition:
  - $\alpha_i = 0 \Rightarrow y_i f \vec{x}_i > 1$
  - $\alpha_i > 0 \Rightarrow y_i f \vec{x}_i = 1$ **SV (active constraint)**

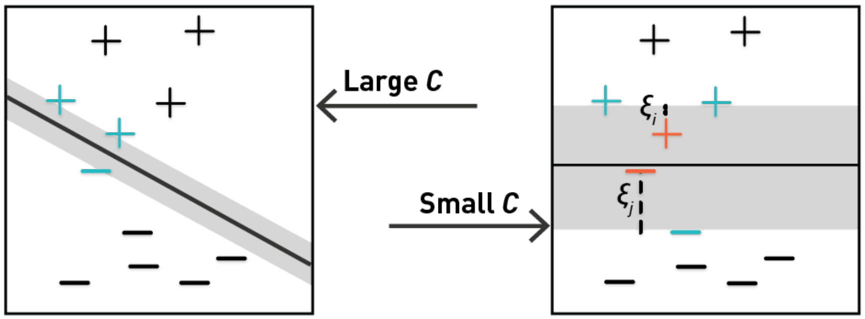
$$\mathrm{W} = \sum_{i=1}^{L} \alpha_i y_i X_i$$
$$\text{and } b = y_{SV} - \langle \mathrm{W}, X_{SV} \rangle$$

$$\langle \mathrm{W}, X \rangle = \left\langle \left(\sum_{i=1}^{L} \alpha_i y_i X_i \right), \mathrm{X} \right\rangle = \sum_{i=1}^{L} \alpha_i y_i \langle X_i, \mathrm{X} \rangle$$

Soft Margin SVM (Primal)

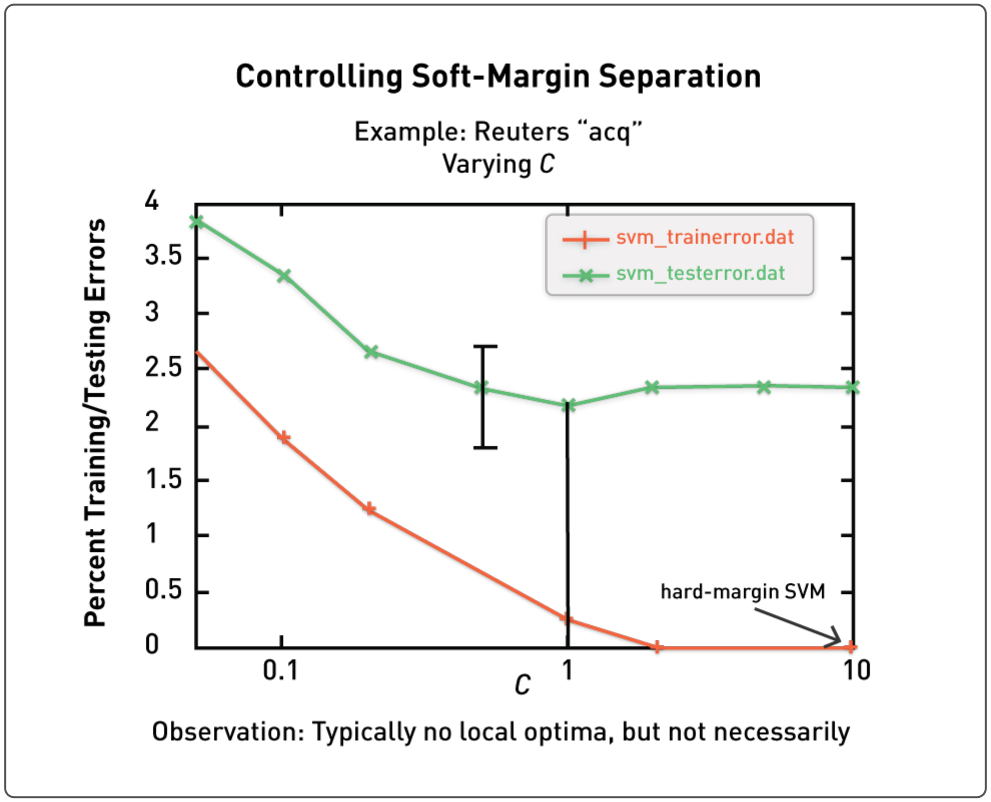
$$\min\ Wb = 0.5 \times \|w\|^2 + C \sum_{i=1}^{L} \xi_i$$

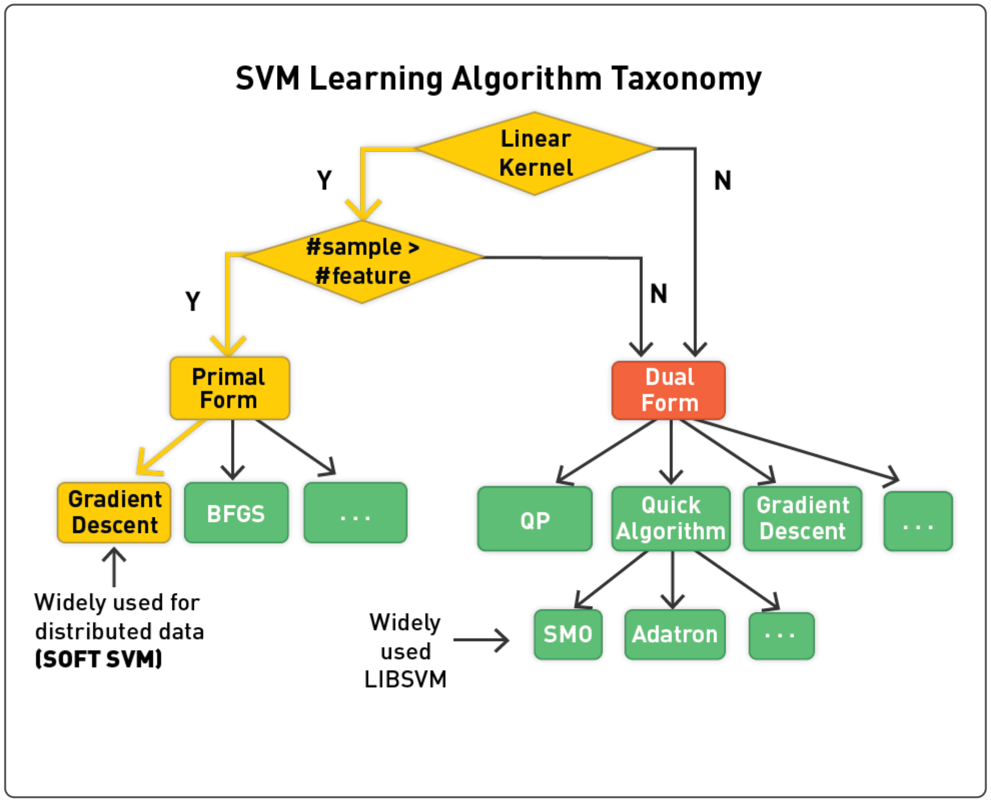
Subject to:
$y_i W X_i + b + \xi_i \geq 1$ **∀ *i* = 1, ... , *L***
and $\xi_i \geq 0$ **∀ *i* = 1, ... , *L***



C > 0 is a *regularization parameter*. It takes the form of a tuning constant that controls the size of the slack variables and balances the two terms in the minimizing function.

- Large *C* → hard margin (allows very few errors)
- Small *C* → allows a lot of slack and therefore a large margin


Controlling Soft-Margin Separation
Example: Reuters "acq"
Varying C
Percent Training/Testing Errors
4
3.5
3
2.5
2
1.5
1
0.5
0
0.1
1
10
C
svm_trainerror.dat
svm_testerror.dat
hard-margin SVM
Observation: Typically no local optima, but not necessarily


SVM Learning Algorithm Taxonomy
Linear Kernel
Y
N
#sample > #feature
Y
N
Primal Form
Dual Form
Gradient Descent
BFGS
. . .
QP
Quick Algorithm
Gradient Descent
. . .
SMO
Adatron
. . .
Widely used for distributed data
(SOFT SVM)
Widely used LIBSVM

Soft SVM via Unconstrained Optimization

Soft margin SVM (primal)

$$\min Wb = 0.5 \times \left\| w \right\|^2 + C \sum_{i=1}^{L} \xi_i$$

- Subject to:

$y_i W X_i + b + \xi_i \geq 1$ $\forall$ i = 1, ... , L

$\xi_i \geq 0$ $\forall$ i = 1, ... , L

Objective function

- $\text{Min} \frac{\left\| w \right\|^2}{2}$
- Subject to constraints: $y_i w \cdot x_i + b - 1 \geq 0$ $\forall$ i = 1, ... , L

Soft SVM via
Unconstrained Optimization (cont.)

Lagrangean function

$$L_p(\mathbf{w}) = \frac{1}{2}\langle \mathbf{w}, \mathbf{w} \rangle - \sum \alpha_i \left[y_i \left(\langle \mathbf{w}, x_i \rangle + b \right) - 1 \right]$$

$$\alpha \geq 0$$

SVM Learning as Optimization Problem

Soft margin SVM (primal)

$$\min Wb = 0.5 \times \left\| w \right\|^2 + C \sum_{i=1}^{L} \xi_i$$

- Subject to:

$$y_i W X_i + b + \xi_i \geq 1 \qquad \forall \text{ i} = 1, \dots, L$$
$$\xi_i \geq 0 \qquad \forall \text{ i} = 1, \dots, L$$

Regularized hinge loss

Expected hinge loss on the training set

$$\min_w \quad \frac{\lambda}{2} \quad w'w \quad + \quad \frac{1}{m}\sum_i 1 \quad - \quad y_i w' x_i - b_{+}$$

Trade-off between margin and loss

Size of the margin

Positive for correctly classified examples, otherwise negative

SVM Learning as Optimization Problem (cont.)

Regularized hinge loss

$$\min_w \tfrac{\lambda}{2} w'w + \tfrac{1}{m} \sum_i (1 - y_i (w'x_i - b))_+$$

- $1 - z_+ = \ \max 01 - z$ (hinge loss)
- First summand is a quadratic function; the sum is a piecewise linear function
- The whole objective: Piecewise quadratic

Gradient of Regularized Hinge Loss

Regularized hinge loss

$$\min_w \tfrac{\lambda}{2} w'w + \tfrac{1}{m} \sum_i (1 - y_i (w'x_i - b))_+$$

Gradient of regularized hinge loss

- Hinge loss and regularization loss:

λw	if $y_i w' x_i - b > 1$
$\lambda w + y_i x_i$	Otherwise

- $W_{t+1} = w_t + \text{average(gradient)}$
- $W^{t+1} = w_t + \text{average(regularization + hinge loss)}$

Distributed Gradient Descent: Linear Soft SVMs

Master-Slave Process

- Initialize model parameters; assume a weight vector *W* = (0, 0, … , 0); Gradient = (0, 0, … , 0)
- While not converged
  - MASTER: Broadcast model (i.e, weight vector) to the worker nodes
  - MASTER launches MapReduce jobs
    - Mappers (*many* mappers) to compute partial gradients over the respective training data subsets (chunks)

      Init *g* = (0, 0, … , 0).

      Combine in memory: $g = \sum_i gW; X_i, Y_i$.

      Finally yield the partial gradient *g*.

      Gradient of regularized hinge loss: λw if $y_i w' x_i - b > 1$; $\lambda w + y_i x_i$ Otherwise

    - Reducer (single reducer)

      Initialize full gradient: *G* = (0, 0, … , 0).

      For each partial gradient *g*

      Aggregate partial gradients: $g = \sum_m g_m$.

      Yield full gradient *G*.
  - MASTER *W* = *W* +α*G* // update the weight vector
- End-While

PySpark: SVM via Gradient Descent (Single Node)

Link to notebook: [SVM-Notebook-Linear-Kernel-2015-06-19.ipynb]

```
#gradient descent (and with no stochasticity!)
#Objective Function
#minw   λ/2   w'w   +   1/m ∑i(1 - yi(w'xi - b))+
#gradient
#   λw           if yi(w'xi - b) > 1   #correctly classified
#   λw + yi xi   Otherwise             #incorrectly classified
def SVM_GD(data,y,w=None,eta=0.01,iter_num=1000,regPara=0.01,
stopCriteria=0.0001):
    data = np.append(data,np.ones((data.shape[0],1)),axis=1)
    if w==None:
       w = np.random.normal(size=data.shape[1])
    for i in range(iter_num):
       wxy = np.dot(data,w)*y #labeled margin
       xy = -data*y[:,None]
       zipv = zip(xy,wxy)
       #Gradient of hinge loss: if wxy<0 then hinge loss is xy
       g = sum((u for u,v in zipv if, v <1])/data.shape[0]
       wreg = w*1 #weight vector
       wreg[-1] = 0
       #wreg = np.array([w[0],w[1],0])
       wdelta = eta*(g+regPara*wreg)
       if sum(abs(wdelta))<=stopCriteria*sum(abs(w)):
          break
       w = w - wdelta
    return w
```

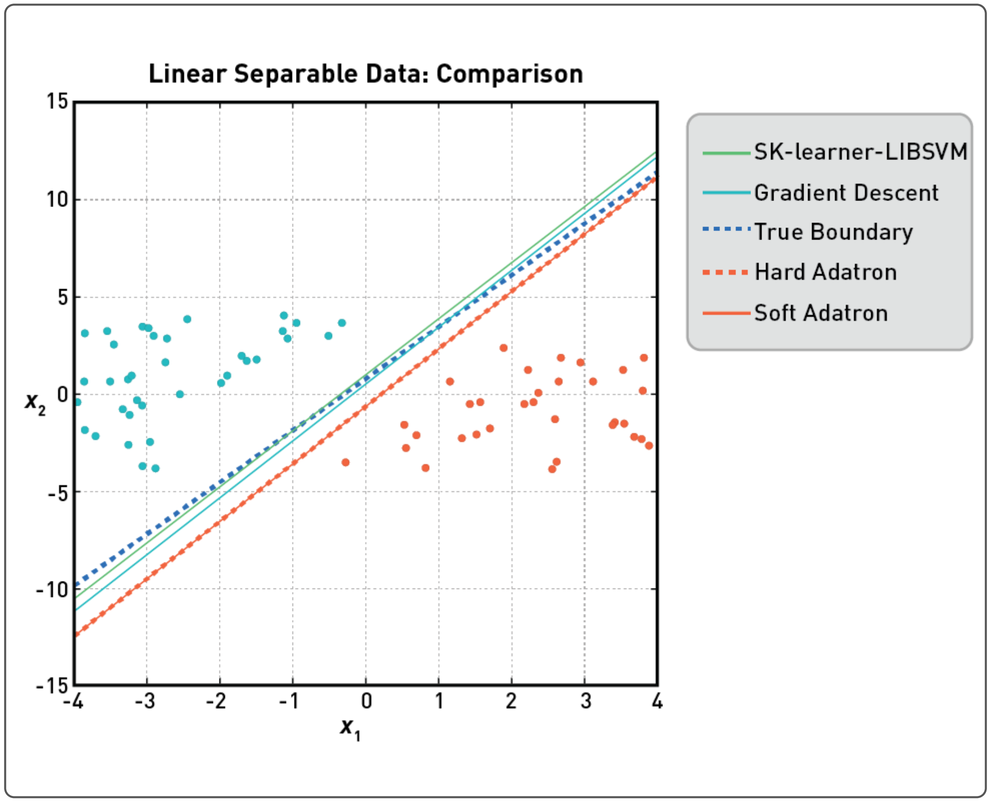
PySpark: Distributed SVM

Link to notebook: [SVM-Notebook-Linear-Kernel-2015-06-19.ipynb]

```
#gradient descent (and with no stochasticity!)
#Objective Function
#minw   λ/2   w'w   +   1/m ∑i(1 - yi(w'xi - b))+
#gradient
#   λw           if yi(w'xi - b) > 1   #correctly classified
#   λw + yi xi   Otherwise             #incorrectly classified

def SVM_GD_SPARK(data,y,w=None,eta=0.01,iter_num=1000,regPara=0.01, stopCriteria=0.0001):
    #eta learning rate
    #regPara
    dataRDD = sc.parallelize(np.append(y[:,None],data,axis=1)).cache() #prepend y to x
    if w==None:
       w = np.random.normal(size=data.shape[1]+1)
    for i in range(iter_num): #label*margin
       sv = dataRDD.filter(lambda x:x[0]*np.dot(w,np.append(x[1:],1))<1) #Support vector? with label*margin<1
       if sv.isEmpty(): #converged as no more updates possible
          break #hinge loss component of gradient y*x and sum up
       g = -sv.map(lambda x:x[0]*np.append(x[1:],1)).reduce(lambda x,y:x+y)/data.shape[0] #gradient: total hinge 1
       wreg = w*1 #temp copy of weight vector
       wreg[-1] = 0 #last value of weight vector is bias term; ignore in regularization
       wdelta = eta*(g+regPara*wreg) #gradient: hinge loss + regularized term
       if sum(abs(wdelta))<=stopCriteria*sum(abs(w)): #converged as updates to weight vector are small
          break
       w = w - wdelta
    return w
```

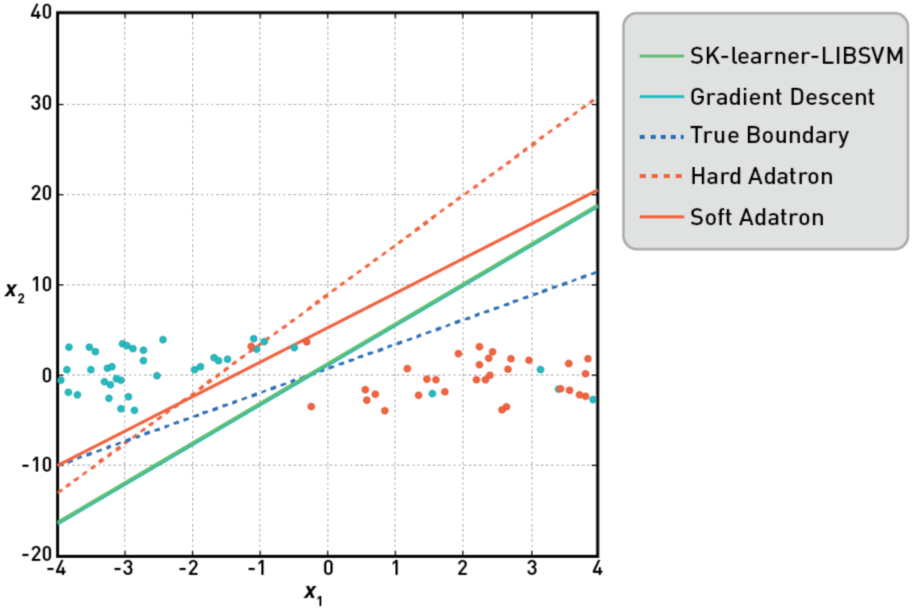
Linearly Separable Data: Code

- ```
  np.randomseed(400)
  w_gd_spark_sep = SVM_GD_SPARK(data_sep,y_sep)
  print w_gd_spark_sep
  ```
- ```
  array([ 0.93510664, -0.31799796,   0.10234458])
  ```


Linear Separable Data: Comparison
SK-learner-LIBSVM
Gradient Descent
True Boundary
Hard Adatron
Soft Adatron
x_1
x_2
15
10
5
0
-5
-10
-15
-4
-3
-2
-1
0
1
2
3
4

Linear Inseparable Data: Comparison

```
: comparison_plot(data_insep,y_insep,w_insep,w_sklearn_libsvm_insep,w_gd_insep,w_hard_adatron_insep,
                  w_soft_adatron_insep,sep=False)
```



Supervised Machine Learning

- Loss functions
- General framework for gradient descent notebook
- Logistic regression at scale
- Perceptron
- SVMs