In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import timeit
```

1 Preliminaries

1.1 Dataset construction

Start by creating a design matrix for regression with $m = 150$ examples, each of dimension $d = 75.$ We will choose a true weight vector θ that has only a few non-zero components:

In [13]:
```python
def create_design_matrix(examples,dimensions,seed=1):
    """
    examples: number of data points
    dimensions: number of features
    seed: for replicating randomization
    """
    np.random.seed(seed)
    X = np.random.rand(examples,dimensions)
    theta = np.zeros(dimensions)
    np.random.seed(seed)

    first_ten = np.random.randint(2, size=10)
    first_ten[first_ten==0] = -10
    first_ten[first_ten==1] = 10
    #theta[0:10] = [10,-10,10,10,-10,-10,10,10,10,-10]
    theta[0:10]=first_ten
    np.random.seed(seed)

    epsilon = 0.1*np.random.randn(examples)
    y = np.dot(X,theta) + epsilon

    return X,y,theta


def train_test_split(X,y,n_train,n_val,n_test):
    assert n_train+n_val+n_test == 150, "Train test split doesn't add up
"
    X_train = X[:n_train,:]
    y_train = y[:n_train]
    X_val = X[n_train:n_train+n_val,:]
    y_val = y[n_train:n_train+n_val]
    X_test = X[n_train+n_val:,:]
    y_test = y[n_train+n_val:]
    return X_train, y_train, X_val, y_val, X_test, y_test

(X_all,y_all,true_theta) = create_design_matrix(150,75)
(X_train, y_train, X_val, y_val, X_test, y_test)=train_test_split(X_all,
y_all,80,20,50)

print "train",X_train.shape
print "val",X_val.shape
print "test",X_test.shape
print "ytrain",y_train.shape
print "yval",y_val.shape
print "ytest",y_test.shape
```

```
train (80, 75)
val (20, 75)
test (50, 75)
ytrain (80,)
yval (20,)
ytest (50,)
```

1.2 Experiments with Ridge Regression

1. Run ridge regression on this dataset. Choose the λ that minimizes the square loss on the validation set.

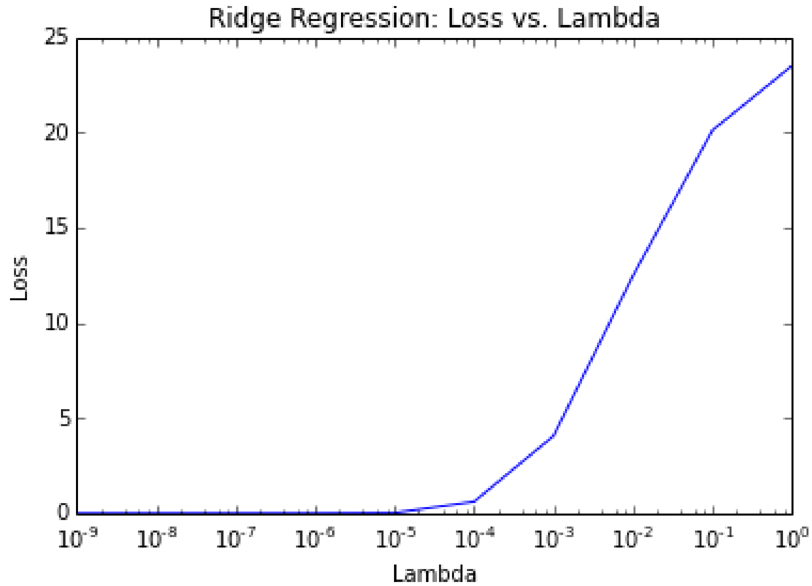
In [3]:
```
from sklearn.linear_model import Ridge
from scipy.optimize import minimize
```

In [4]:
```
def ridge(X,y,Lambda):
    (N,D) = X.shape
    """
    takes a regularization term Lambda and returns objective function ri
dge_obj
    """
    def ridge_obj(theta):
        return ((np.linalg.norm(np.dot(X,theta) - y))**2)/(2*N) + Lambda
*(np.linalg.norm(theta))**2
    return ridge_obj

def compute_loss(X,y,theta):
    (N,D) = X.shape
    """
    Computes loss for given dataset and weight vector theta
    """
    return ((np.linalg.norm(np.dot(X,theta) - y))**2)/(2*N)

def run_ridge_regression():
    (N,D) = X.shape
    w = np.random.rand(D,1)

    #Sklearn implementation
    def ridge_regression(alpha,train,val):
        ridge_model = Ridge(alpha)
        #ridge_model.fit(X, y)

    #Try various Lambdas, optimize theta, and print loss
    t=0
    loss_opt=lambda_opt=w_opt_opt=np.nan
    Lambdas=[]
    loss_hist=[]
    for i in range(-9,1):
        Lambda = 10**i;
        Lambdas.append(Lambda)
        w_opt = minimize(ridge(X_train,y_train,Lambda), w)
        loss=compute_loss(X_val,y_val, w_opt.x)
        loss_hist.append(loss)
        if t==0 or loss<loss_opt:
            loss_opt = loss
            lambda_opt = Lambda
            w_opt_opt = w_opt.x.copy()
        t=t+1

    return w_opt_opt,Lambdas,loss_hist
w_ridge,lambda_ridge,loss_ridge = run_ridge_regression()
```

In [5]:
```
def ridge(X,y,Lambda):
    (N,D) = X.shape
    """
    takes a regularization term Lambda and returns objective function ri
dge_obj
    """
    def ridge_obj(theta):

        return ((np.linalg.norm(np.dot(X,theta) - y))**2)/(2*N) + Lambda
*(np.linalg.norm(theta))**2
    return ridge_obj

def compute_loss(X,y,theta):
    (N,D) = X.shape
    """
    Computes loss for given dataset and weight vector theta
    """
    return ((np.linalg.norm(np.dot(X,theta) - y))**2)/(2*N)

def run_ridge_regression():
    (N,D) = X.shape
    w = np.random.rand(D,1)

    #Sklearn implementation
    def ridge_regression(alpha,train,val):
        ridge_model = Ridge(alpha)
        #ridge_model.fit(X, y)

    #Try various Lambdas, optimize theta, and print loss
    t=0
    loss_opt=lambda_opt=w_opt_opt=np.nan
    Lambdas=[]
    loss_hist=[]
    for i in range(-9,1):
        Lambda = 10**i;
        Lambdas.append(Lambda)
        w_opt = minimize(ridge(X_train,y_train,Lambda), w)
        loss=compute_loss(X_val,y_val, w_opt.x)
        loss_hist.append(loss)
        if t==0 or loss<loss_opt:
            loss_opt = loss
            lambda_opt = Lambda
            w_opt_opt = w_opt.x.copy()
        t=t+1

    print "Best Lambda",lambda_opt
    print "Best Loss", loss_opt

    return w_opt_opt,Lambdas,loss_hist

w_ridge,lambdas_ridge,loss_hist_ridge = run_ridge_regression()
```

```
Best Lambda 1e-06
Best Loss 0.0171323767578
```

In [6]:
```
plt.plot(lambdas_ridge,loss_hist_ridge)
plt.xlabel('Lambda')
plt.ylabel('Loss')
plt.xscale('log')
plt.title('Ridge Regression: Loss vs. Lambda')
plt.show()
```



In [14]:
```
def evaluate_theta_estimate(truth,est, tolerance=0):
    print "tolerance",tolerance
    false_nonzeros = len(est[(truth==0) & (est!=0)])
    false_zeros = len(est[(truth!=0) & (abs(est)<tolerance)])
    print "True theta is zero, estimate is nonzero",false_nonzeros
    print "True theta is nonzero, estimate is zero",false_zeros
    return false_nonzeros,false_zeros

evaluate_theta_estimate(true_theta,w_ridge)
evaluate_theta_estimate(true_theta,w_ridge,tolerance=10**-3)
```

```
tolerance 0
True theta is zero, estimate is nonzero 65
True theta is nonzero, estimate is zero 0
tolerance 0.001
True theta is zero, estimate is nonzero 65
True theta is nonzero, estimate is zero 0
```

Out[14]: (65, 0)

2 Lasso

2.1 Shooting Algorithm

1. Write a function that computes the Lasso solution for a given λ using the shooting algorithm described above. This function should take a starting point for the optimization as a parameter. Run it on the dataset constructed in (1.1), and select the λ that minimizes the square error on the validation set. Report the optimal value of λ found, and the corresponding test error. Plot the validation error vs λ.

> The optimal λ found was 10, corresponding test error was 443.

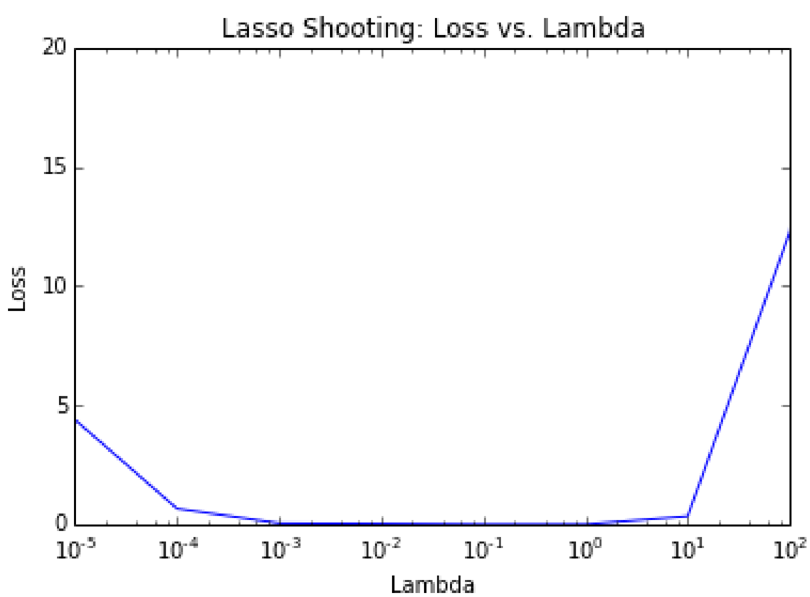
In [39]:
```
def soft(a,d):
    return np.sign(a) * max(abs(a)-d,0)

def lasso_shooting(X,y,Lambda, w_init=[], tolerance=1e-5):
    """
    Takes training data and regularization Lambda,
    performs coordinate descent for lasso, aka the "shooting algorithm",
    and returns a vector of estimated weights
    """
    (N,D) = X.shape

    if len(w_init)==0:
        w=np.ones(D)
    else: w=w_init

    maxIter = 1000
    converged = False
    iteration=1
    while (converged==False) & (iteration<maxIter):
        w_old = w.copy()
        for j in range(D):
            aj=cj=0
            for i in range(1,N):
                aj=aj+2*X[i,j]**2
                cj=cj+2*X[i,j]*(y[i]-np.dot(w,X[i,:])+ w[j]*X[i,j])

            w[j] = soft(cj/aj,Lambda/aj)
        iteration=iteration+1
        converged = (sum(abs(w-w_old)) < tolerance)

    print "Converged:",converged,"Iterations:",iteration
    return w

%timeit lasso_shooting(X_train,y_train,10)
```

```
Converged: True Iterations: 465
Converged: True Iterations: 465
Converged: True Iterations: 465
Converged: True Iterations: 465
1 loops, best of 3: 35 s per loop
```

In [33]:
```
def Lambda_search(lasso_func):
    """
    Runs lasso_shooting on a training set with a variety of
    regularization hyperparameters Lambda.
    Selects the lambda that minimizes square error on the validation set
.
    Plots validation error vs. Lambda
    Prints selected lambda along with corresponding test error.
    """
    #loop through array of lambdas
    t=0
    Lambdas=[]
    loss_hist=[]

    print "Starting Loop"
    for i in range(-5,6):
        Lambda = 10**i
        print "Lambda",Lambda
        Lambdas.append(Lambda)
        w=lasso_func(X_train,y_train,Lambda,w_init=w_ridge)
        loss=compute_loss(X_val,y_val,w)
        loss_hist.append(loss)

        if t==0 or loss<=loss_opt:
            loss_opt = loss
            lambda_opt = Lambda
            w_opt = w.copy()
        t=t+1

    test_loss = compute_loss(X_test,y_test, w_opt)

    print "Best Lambda:",lambda_opt
    print "Square Loss on Test Data:", loss_opt

    return w_opt,Lambdas,loss_hist


w_lasso,lambdas_lasso,loss_hist_lasso=Lambda_search(lasso_shooting)
```

```
Starting Loop
Lambda 1e-05
Converged: False Iterations: 1000
Lambda 0.0001
Converged: False Iterations: 1000
Lambda 0.001
Converged: False Iterations: 1000
Lambda 0.01
Converged: False Iterations: 1000
Lambda 0.1
Converged: False Iterations: 1000
Lambda 1
Converged: True Iterations: 357
Lambda 10
Converged: True Iterations: 148
Lambda 100
Converged: True Iterations: 142
Lambda 1000
Converged: True Iterations: 251
Lambda 10000
Converged: True Iterations: 3
Lambda 100000
Converged: True Iterations: 2
Best Lambda: 1
Square Loss on Test Data: 0.00607817992338
```

In [37]:
```
plt.plot(lambdas_lasso,loss_hist_lasso)
plt.xlabel('Lambda')
plt.ylabel('Loss')
plt.xscale('log')
plt.xlim((1e-5,1e2))
plt.ylim((0,20))
plt.title('Lasso Shooting: Loss vs. Lambda')
plt.show()
```



2. Analyze the sparsity of your solution, reporting how many components with true value zero have been estimated to be non-zero, and vice-versa.

> There were 7 cases where the true value zero has been estimated to be non-zero, but the reverse never happened.

In [38]:
```
evaluate_theta_estimate(true_theta,w_lasso)
```

```
tolerance 0
True theta is zero, estimate is nonzero 7
True theta is nonzero, estimate is zero 0
```

Out[38]:
```
(7, 0)
```

3.Implement the homotopy method described above. Compare the runtime for computing the full regularization path (for the same set of λ's chosen above) using the homotopy method compared to starting with the same intial point every time.

> The time has improved from 454sec per run to 403sec per run.

In [ ]:
```
def get_lambda_max(X,y):
    lambda_max = 2*np.linalg.norm(X.T*y,np.inf)
    return lambda_max

def homotopy_lambda_search():
    """
    Runs lasso_shooting on a training set,
    using Lambdas chosen through the homotopy method.
    Selects the lambda that minimizes square error on the validation set
.
    Plots validation error vs. Lambda
    Prints selected lambda along with corresponding test error.
    """
    (N,D) = X_train.shape
    lambda_max = get_lambda_max(X_train,y_train)
    log_lambda_max = int(np.ceil(np.log10(lambda_max)))
    #loop through array of lambdas
    t=0
    Lambdas=[]
    loss_hist=[]
    w=np.zeros(D)
    for i in range(log_lambda_max,-5,-1):
        w_old=w.copy()
        Lambda = 10**i
        print "Lamba",Lambda
        Lambdas.append(Lambda)

        w=lasso_shooting(X_train,y_train,Lambda,w_init=w_old)
        loss=compute_loss(X_val,y_val, w)
        loss_hist.append(loss)

        if t==0 or loss<=loss_opt:
            loss_opt = loss
            lambda_opt = Lambda
            w_opt = w.copy()
        t=t+1

    test_loss = compute_loss(X_test,y_test, w_opt)

    print "Best Lambda:",lambda_opt
    print "Square Loss on Test Data:", loss_opt

    return w_opt,Lambdas,loss_hist

w_homotopy,lambdas_homotopy,loss_hist_homotopy = homotopy_lambda_search(
)

plt.plot(lambdas_homotopy,loss_hist_homotopy)
plt.xlabel('Lambda')
plt.ylabel('Loss')
plt.xscale('log')
plt.title('Homotopy Loss vs. Lambda')
plt.show()
```

In [40]:
```
def timeme(func,iterations=1,*args,**kwargs):
    """
    Timer wrapper.  Runs a given function, with arguments,
    100 times and displays the average time per run.
    """
    def wrapper(func, *args, **kwargs):
        def wrapped():
            return func(*args, **kwargs)
        return wrapped
    wrapped = wrapper(func,*args,**kwargs)
    run_time = float(timeit.timeit(wrapped, number=iterations))/iteratio
ns
    print "Avg time to run %s after %i trials: %i seconds per trial" %(f
unc,iterations,run_time)
```

In [41]:
```
timeme(Lambda_search,1,lasso_shooting);
timeme(homotopy_lambda_search,1);
```

```
Starting Loop
Lambda 1e-05
Converged: False Iterations: 1000
Lambda 0.0001
Converged: False Iterations: 1000
Lambda 0.001
Converged: False Iterations: 1000
Lambda 0.01
Converged: False Iterations: 1000
Lambda 0.1
Converged: False Iterations: 1000
Lambda 1
Converged: True Iterations: 357
Lambda 10
Converged: True Iterations: 148
Lambda 100
Converged: True Iterations: 142
Lambda 1000
Converged: True Iterations: 251
Lambda 10000
Converged: True Iterations: 3
Lambda 100000
Converged: True Iterations: 2
Best Lambda: 1
Square Loss on Test Data: 0.00607817992338
Avg time to run <function Lambda_search at 0x107a6d2a8> after 1 trials: 4
54 seconds per trial
Lamba 10000
Converged: True Iterations: 2
Lamba 1000
Converged: True Iterations: 251
Lamba 100
Converged: True Iterations: 143
Lamba 10
Converged: True Iterations: 214
Lamba 1

Converged: True Iterations: 411
Lamba 0.1
Converged: False Iterations: 1000
Lamba 0.01
Converged: False Iterations: 1000
Lamba 0.001
Converged: False Iterations: 1000
Lamba 0.0001
Converged: False Iterations: 1000
Best Lambda: 1
Square Loss on Test Data: 0.00607723555481
Avg time to run <function homotopy_lambda_search at 0x10796b488> after 1
trials: 403 seconds per trial
```

4. Derive matrix expressions for computing a_j and c_j

$$a_j = 2\sum_{i=1}^{n} x_{ij}^2 = 2X_j^T X_j = 2[X^T X]_{j,j}$$

$$c_j = 2\sum_{i=1}^{n} x_{ij}(y_i - \mathbf{w}^T \mathbf{x}_i + w_j x_{ij})$$

$$= 2\sum_{i=1}^{n} x_{ij} y_i - 2\sum_{i=1}^{n} x_{ij} \mathbf{w}^T \mathbf{x}_i + 2\sum_{i=1}^{n} x_{ij} w_j x_{ij}$$

$$= 2X_j^T \mathbf{y} - \sum_{k=1}^{d} 2X_k^T X \mathbf{w} + 2X_j^T X_j w_j$$

$$= 2[X^T \mathbf{y}]_j - \sum_{k=1}^{d} 2[X^T X]_k \mathbf{w} + 2[X^T X]_j w_j$$

---

5. Implement the matrix expressions and measure the speedup to compute the regularization path.

Time improved from 454 sec per run to 180 sec per run.

In [45]:

```
def lasso_shooting_matrix(X,y,Lambda, w_init=[], tolerance=1e-5):
    """
    Takes training data and regularization Lambda,
    performs coordinate descent for lasso, aka the "shooting algorithm",
    and returns a vector of estimated weights
    """

    (N,D) = X.shape

    if len(w_init)==0:
        w=np.ones(D)
    else: w=w_init

    maxIter = 1000
    converged = False
    iteration=1
    a=c=0
    while (converged==False) & (iteration<maxIter):
        w_old = w.copy()
        XTX = np.dot(X.T,X)

        XTy = np.dot(X.T,y)
        for j in range(D):
            a=2*XTX[j,j]
            c=2*XTy[j] - sum(2*np.dot(XTX[j,:],w)) + 2*XTX[j,j]*w[j]
            w[j]=soft(c/a,Lambda/a)
            #X_i_sq = np.apply_along_axis(lambda x: x ** 2,1,X[i,:])

        iteration=iteration+1
        converged = (sum(abs(w-w_old)) < tolerance)

    print "Converged:",converged,"Iterations:",iteration
    return w
```

In [45]:

```
timeme(Lambda_search,1,lasso_shooting_matrix);
```

2.3 Feature Correlation

1. Derive the relation between $\hat{\theta}_i$ and $\hat{\theta}_j$, the i^{th} and j^{th} components of the optimal weight vector obtained by solving the Lasso optimization problem.

Assume in the optimal solution that $\hat{\theta}_i = a$ and $\hat{\theta}_j = b$

The lasso objective function, below, must minimize both loss and regularization.

$$\sum_{k=1}^{n}(h(x_k) - y_k)^2 + \lambda \|w\|_1$$

First consider the loss, $\sum_{k=1}^{n}(h(x_k) - y_k)^2$, where $h(x_k) = \mathbf{w}^T x_k$

The loss due to x_i, x_j is $\sum_{k=1}^{n}(\hat{\theta}_i X_{ik} + \hat{\theta}_j X_{jk} - y_k)$

Since $X_i = X_j$, this simplifies to $\sum_{k=1}^{n}((\hat{\theta}_i + \hat{\theta}_j)X_{ik} - y_k)$

Thus the optimal values a and b must sum to another value c that minimizes this expression $\sum_{k=1}^{n}(cX_{ik} - y_k)$

Next we minimize the lasso regularization component, $\lambda \|w\|_1 = \lambda \sum_{l=1}^{d} |w_l|$

The regularization penalty due to x_i, x_j is $|a| + |b|$. If a and b are of opposite sign, and both are nonzero, then $|a| + |b| > |c| + |0|$, and the regularization penalty is not minimized. Therefore a and b must be of the same sign and are constrained by optimal value c such that $a + b = c$

2. Derive the relation between $\hat{\theta}_i$ and $\hat{\theta}_j$, the i^{th} and j^{th} components of the optimal weight vector obtained by solving the ridge regression optimization problem.

The ridge regression objective function, below, must minimize both loss and regularization.

$$\sum_{k=1}^{n}(h(x_k) - y_k)^2 + \lambda ||w||_2^2$$

The loss component is the same as with lasso, so we must minimize the regularization penalty subject to the same constraint as in lasso, which is that $a + b = c$

The regularization penalty due to x_i, x_j is $a^2 + b^2$. Next I will show that $a^2 + b^2 \geq (\frac{c}{2})^2 + (\frac{c}{2})^2$, and therefore a and b must be equal to $\frac{c}{2} = \frac{a+b}{2}$ under these conditions.

Claim: $a^2 + b^2 \geq 2(\frac{c}{2})^2$

Proof: $a^2 + b^2 = a^2 + (c - a)^2$

$= a^2 + c^2 - 2ac + a^2$

$= 2a^2 + c^2 - 2ac$

$= \frac{1}{2}(4a^2 - 4ac + 2c^2)$

$= \frac{1}{2}(2a - c)^2 + \frac{c^2}{2}$

$= \frac{1}{2}(2a - c)^2 + 2(\frac{c}{2})^2$

$\frac{1}{2}(2a - c)^2 \geq 0$, therefore $\frac{1}{2}(2a - c)^2 + 2(\frac{c}{2})^2 \geq 2(\frac{c}{2})^2$

Thus a and b must be equal.

Feedback

1. Approximately how long did it take to complete this assignment?

   20 hours

2. Any other feedback?

   This is exhausting. Thankfully this is my only class this term (I'm also working full-time).