Cody Fizette

Homework 1

In [1]:
```
from hw1_code import *
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import matplotlib
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
```

In [38]:
```
matplotlib.rcParams['figure.figsize'] = [15, 10]
np.random.seed(1337)
```

2. Mathamatical Fundamentals

2.1 Probability

Let $(X_1, X_2, \cdots, X_d)$ have a d-dimensional multivariate Gaussian distribution, with mean vector $\mu \in \mathbb{R}^{\mathrm{d}}$ and covariance matrix $\Sigma \in \mathbb{R}^{\mathrm{d\times d}}$, i.e. $(X_1, X_2, \cdots, X_d) \sim \mathcal{N}(\mu, \Sigma)$. Use μ_i to denote the i^{th} element of μ and Σ_{ij} to denote the element at the i^{th} row and j^{th} column of Σ.

2.1.1

Let $x, y \in R^d$ be two independent samples drawn from $\mathcal{N}(\mu, \Sigma)$. Give expression for $E\|x\|_2^2$ and $E\|x - y\|_2^2$. Express your answer as a function of μ and Σ. $\|x\|_2$ represents the ℓ_2-norm of vector x.

$$
\begin{aligned}
E\|x\|_2^2 &= E\Big(\sum_{i=1}^{d} x_i^2\Big),\\
&= \sum_{i=1}^{d} E(x_i^2),\\
&= \sum_{i=1}^{d} \left(\Sigma_{i,i} + \mu_i^2\right).
\end{aligned}
$$

And

$$
\begin{aligned}
E\|x-y\|_2^2 &= E\Big(\sum_{i=1}^{d}(x_i - y_i)^2\Big),\\
&= \sum_{i=1}^{d}\Big(E(x_i)^2 - 2E(x_* y_i) + E(y_i)^2\Big),\\
&= \sum_{i=1}^{d}\Big(E(x_i^2) - 2E(x_i^2) + E(x_i^2)\Big),\\
&= 0.
\end{aligned}
$$

2.1.2

Find the distribution of $Z = \alpha_i X_i + \alpha_j X_j$, for $i \neq j$ and $1 \leq i, j \leq d$. The answer will belong to a familiar class of distribution. Report the answer by identifying this class of distribution and specifying the parameters.

Z is normally distributed.

$Z \sim \mathcal{N}(\alpha_i\mu_i + \alpha_j\mu_j,\ \alpha_i^2\Sigma_{i,i} + \alpha_j^2\Sigma_{j,j} + 2\alpha_i\alpha_j\Sigma_{i,j})$

2.1.3

Assume W and R are two Gaussian distributed random variables. Is $W + R$ still Gaussian?

No. Proof:

Let $W \sim \mathcal{N}(\mu, \Sigma)$ Now let $R = -W$. Then it follows that $R \sim \mathcal{N}(\mu, \Sigma)$. However,

$$
\begin{aligned}
W + R &= W - W,\\
&= 0.
\end{aligned}
$$

Thus $W + R$ is not Gaussian.

2.2 Linear Algebra

2.2.1

Let A be a $d \times d$ matrix with rank k. Consider the set $S_A := \{x \in R^d | Ax = 0\}$. What is the dimension of S_A?

$dim(S_A) = d - k.$

2.2.2

Assume S_v is a k dimensional subspace in R^d and $v_1, v_2, \cdots, v_k$ form an orthonormal basis of S_v. Let w be an arbitrary vector in R^d. Find

$$x^* = \underset{x \in S_v}{\operatorname{argmin}} \|w - x\|_2,$$

where $\|w - x\|_2$ is the Euclidean distance between w and x. Express x^* as a function of $v_1, v_2, \ldots, v_k$ and w

Solution

$$\begin{aligned} x^* &= \underset{x \in S_v}{\operatorname{argmin}} \|w - x\|_2, \\ &= proj_{s_v}(w), \\ &= \sum_{i=1}^{d} \frac{w \cdot v_i}{v_i \cdot v_i} w_i \end{aligned}$$

3. Linear Regression

3.1 Feature Normalization

```
#######################################
### Feature normalization
def feature_normalization(train, test):
    """Rescale the data so that each feature in the training set is in
    the interval [0,1], and apply the same transformations to the test
    set, using the statistics computed on the training set.

    Args:
        train - training set, a 2D numpy array of size (num_instances, num_featur
es)
        test - test set, a 2D numpy array of size (num_instances, num_features)

    Returns:
        train_normalized - training set after normalization
        test_normalized - test set after normalization
    """
    # Remove columns with constant values
    cols_to_delete = np.all(train==train[0,:], axis=0)
    cols_to_delete = np.argwhere(cols_to_delete==True)
    train = np.delete(train, cols_to_delete, 1)
    test = np.delete(test, cols_to_delete, 1)

    min_arr = np.amin(train, axis=0)
    range_arr = np.amax(train, axis=0) - min_arr

    train_normalized = (train-min_arr)/range_arr
    test_normalized = (test-min_arr)/range_arr

    return train_normalized, test_normalized
```

3.2 Gradient Descent Setup

3.2.1

$$J(\theta) = \frac{1}{m}(X^T\theta - y)^T(X^T\theta - y)$$

3.2.2

$$\nabla J(\theta) = \frac{2}{m}(X^T\theta - y)^T X$$

3.2.3

$$J(\theta + \eta h) - J(\theta) \approx J(\theta) + \eta h^T \nabla J(\theta)$$

3.2.4

$$\theta \leftarrow \theta - \eta \nabla J(\theta)$$

3.2.5

```
#######################################
### The square loss function
def compute_square_loss(X, y, theta):
    """
    Given a set of X, y, theta, compute the average square loss for predicting y
with X*theta.

    Args:
        X - the feature vector, 2D numpy array of size (num_instances, num_featur
es)
        y - the label vector, 1D numpy array of size (num_instances)
        theta - the parameter vector, 1D array of size (num_features)

    Returns:
        loss - the average square loss, scalar
    """
    y_pred = np.matmul(X,theta)

    #return (1/len(y)) * np.matmul((y_pred - y).T, (y_pred - y))

    return np.mean(np.square(y - y_pred))
```

3.2.6

```
#######################################
### The gradient of the square loss function
def compute_square_loss_gradient(X, y, theta):
    """
    Compute the gradient of the average square loss (as defined in compute_square
_loss), at the point theta.

    Args:
        X - the feature vector, 2D numpy array of size (num_instances, num_featur
es)
        y - the label vector, 1D numpy array of size (num_instances)
        theta - the parameter vector, 1D numpy array of size (num_features)

    Returns:
        grad - gradient vector, 1D numpy array of size (num_features)
    """
    y_pred = np.matmul(X, theta)
    n = len(y)
    return (2/n) * np.matmul(X.T, y_pred - y)
```

3.3 Gradient Checker

```
#######################################
### Gradient checker
def grad_checker(X, y, theta, epsilon=0.01, tolerance=1e-4):
    """Implement Gradient Checker
    Check that the function compute_square_loss_gradient returns the
    correct gradient for the given X, y, and theta.

    Let d be the number of features. Here we numerically estimate the
    gradient by approximating the directional derivative in each of
    the d coordinate directions:
    (e_1 = (1,0,0,...,0), e_2 = (0,1,0,...,0), ..., e_d = (0,...,0,1))

    The approximation for the directional derivative of J at the point
    theta in the direction e_i is given by:
    ( J(theta + epsilon * e_i) - J(theta - epsilon * e_i) ) / (2*epsilon).

    We then look at the Euclidean distance between the gradient
    computed using this approximation and the gradient computed by
    compute_square_loss_gradient(X, y, theta).  If the Euclidean
    distance exceeds tolerance, we say the gradient is incorrect.

    Args:
        X - the feature vector, 2D numpy array of size (num_instances, num_featur
es)
        y - the label vector, 1D numpy array of size (num_instances)
        theta - the parameter vector, 1D numpy array of size (num_features)
        epsilon - the epsilon used in approximation
        tolerance - the tolerance error

    Return:
        A boolean value indicating whether the gradient is correct or not
    """
    true_gradient = compute_square_loss_gradient(X, y, theta) #The true gradient
    num_features = theta.shape[0]
    approx_grad = np.zeros(num_features) #Initialize the gradient we approximate

    hs = np.eye(num_features)

    for i, h in enumerate(hs):
        approx_grad[i] = (compute_square_loss(X, y, theta + epsilon*h) - compute_
square_loss(X, y, theta - epsilon*h))/(2*epsilon)

    dist = np.linalg.norm(approx_grad - true_gradient)

    return dist <= tolerance
```

```
#######################################
### Generic gradient checker
def generic_gradient_checker(X, y, theta, objective_func, gradient_func, epsilon=
0.01, tolerance=1e-4):
    """
    The functions takes objective_func and gradient_func as parameters.
    And check whether gradient_func(X, y, theta) returned the true
    gradient for objective_func(X, y, theta).
    Eg: In LSR, the objective_func = compute_square_loss, and gradient_func = com
pute_square_loss_gradient
    """
    true_gradient = gradient_func(X, y, theta) #The true gradient
    num_features = theta.shape[0]
    approx_grad = np.zeros(num_features) #Initialize the gradient we approximate

    hs = np.eye(num_features)

    for i, h in enumerate(hs):
        approx_grad[i] = (objective_func(X, y, theta + epsilon*h) - objective_fun
c(X, y, theta - epsilon*h))/(2*epsilon)

    dist = np.linalg.norm(approx_grad - true_gradient)

    return dist <= tolerance
```

Some helper functions

In [3]:
```
def add_bias(X, b=1):
    n = X.shape[0]
    bias = b*np.ones((n,1))
    return np.hstack((bias, X))
```

In [4]:
```
def load_data(train_size=0.8):
    df = pd.read_csv('data.csv').values
    train, test = train_test_split(df, train_size=train_size)
    train, test = feature_normalization(train, test)
    X = train[:, :-1]
    y = train[:,-1]
    X_test = test[:,:-1]
    y_test = test[:,-1]
    return X, y, X_test, y_test
```

In [5]:
```
X, y, _, __ = load_data(train_size=0.99)
```

```
/home/cfizette/anaconda3/envs/ml/lib/python3.6/site-packages/sklearn/model_
selection/_split.py:2179: FutureWarning: From version 0.21, test_size will
always complement train_size unless both are specified.
  FutureWarning)
```

In [6]:
```
X = add_bias(X)
```

3.4 Batch Gradient Descent

3.4.1

```
#######################################
### Batch gradient descent
def batch_grad_descent(X, y, alpha=0.1, num_step=1000, grad_check=False):
    """
    In this question you will implement batch gradient descent to
    minimize the average square loss objective.

    Args:
        X - the feature vector, 2D numpy array of size (num_instances, num_featur
es)
        y - the label vector, 1D numpy array of size (num_instances)
        alpha - step size in gradient descent
        num_step - number of steps to run
        grad_check - a boolean value indicating whether checking the gradient whe
n updating

    Returns:
        theta_hist - the history of parameter vector, 2D numpy array of size (num
_step+1, num_features)
                     for instance, theta in step 0 should be theta_hist[0], theta
in step (num_step) is theta_hist[-1]
        loss_hist - the history of average square loss on the data, 1D numpy arra
y, (num_step+1)
    """
    num_instances, num_features = X.shape[0], X.shape[1]
    theta_hist = np.zeros((num_step+1, num_features)) #Initialize theta_hist
    loss_hist = np.zeros(num_step+1) #Initialize loss_hist
    theta = np.zeros(num_features) #Initialize theta

    for i in range(num_step+1):
        loss_hist[i] = compute_square_loss(X, y, theta)
        theta_hist[i] = theta

        grad = compute_square_loss_gradient(X, y, theta)

        if grad_check:
            if not grad_checker(X, y, theta):
                warnings.warn('Error computing gradient on iteration {}'.format(i
))
                return theta_hist, loss_hist

        theta -= grad*alpha

    return theta_hist, loss_hist
```

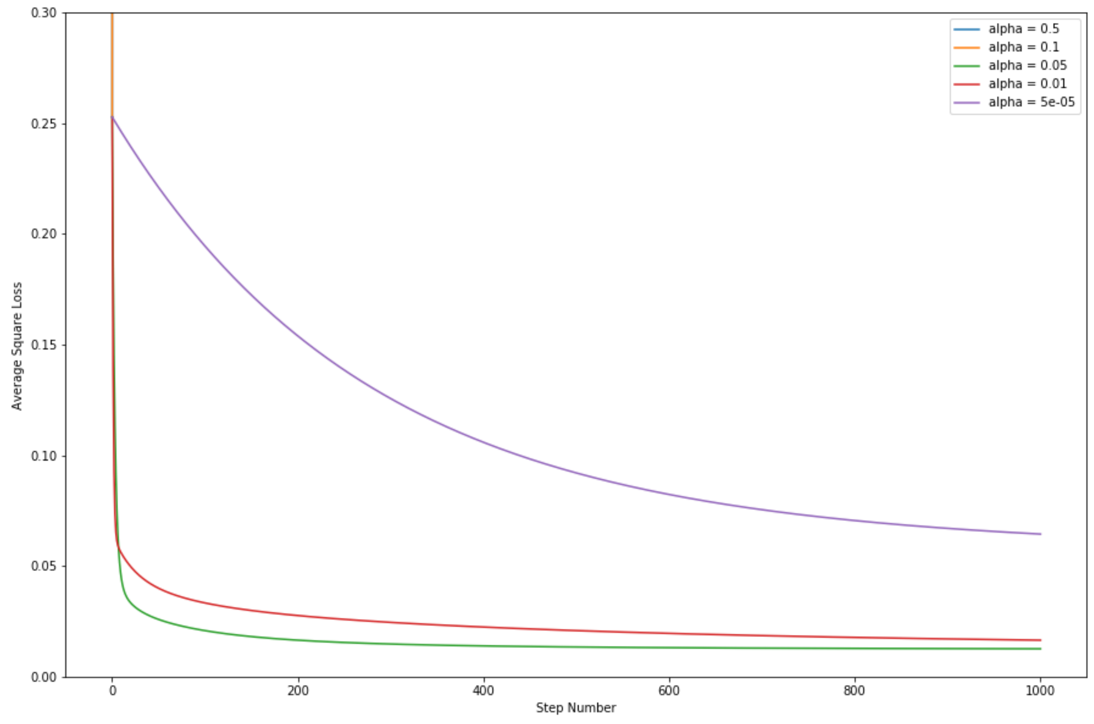
```
In [7]: batch_alphas = [0.5, 0.1, 0.05, 0.01, 0.00005]
        theta_hists = []
        loss_hists = []
        for alpha in batch_alphas:
            theta_hist, loss_hist = batch_grad_descent(X, y, alpha=alpha, num_step=
        1000, grad_check=False)
            theta_hists.append((theta_hist, alpha))
            loss_hists.append((loss_hist, alpha))
```

```
/home/cfizette/anaconda3/envs/ml/lib/python3.6/site-packages/numpy/core/_me
thods.py:75: RuntimeWarning: overflow encountered in reduce
  ret = umr_sum(arr, axis, dtype, out, keepdims)
/home/cfizette/NYU/NYU-Data-Science/DS-1003/hw/hw1/hw1_code.py:62: RuntimeW
arning: overflow encountered in square
  return np.mean(np.square(y - y_pred))
```

3.4.2

```
In [39]: for loss_hist, alpha in loss_hists:
             plt.plot(loss_hist, label='alpha = {}'.format(alpha))
         plt.ylim(0,0.3)
         plt.legend()
         plt.xlabel('Step Number')
         plt.ylabel('Average Square Loss')
```

```
Out[39]: Text(0, 0.5, 'Average Square Loss')
```



Step sizes above 0.0 resulted in divergence.

3.4.3

```
########################################
### Backtracking line search
#Check http://en.wikipedia.org/wiki/Backtracking_line_search for details
def check_ag_condition(X, y, theta, current_loss, alpha, c, p, grad_norm):
    # Checks Armijo–Goldstein condition for linear regression
    # Returns true if condition not satisfied
    return current_loss - compute_square_loss(X, y, theta-alpha*p) < c*alpha*grad
_norm

def normalize(v):
    norm = np.linalg.norm(v)
    if norm == 0:
       return v
    return v / norm , norm

def backtracking_line_search(X, y, max_alpha=1, b=0.5, c=0.5, num_step=1000):
    num_instances, num_features = X.shape[0], X.shape[1]
    theta_hist = np.zeros((num_step+1, num_features)) #Initialize theta_hist
    loss_hist = np.zeros(num_step+1) #Initialize loss_hist
    theta = np.zeros(num_features) #Initialize theta

    for i in range(num_step + 1):
        alpha = max_alpha
        loss_hist[i] = compute_square_loss(X, y, theta)
        theta_hist[i] = theta

        #precompute to avoid unnecessary computation
        current_loss = compute_square_loss(X, y, theta)
        grad = compute_square_loss_gradient(X, y, theta)
        p, grad_norm = normalize(grad)

        # While Armijo-Goldstein condition is not satisfied, shrink alpha
        while check_ag_condition(X, y, theta, current_loss, alpha, c, grad, grad_
norm):
            alpha = b*alpha

        theta -= alpha*p

    return theta_hist, loss_hist
```

In [9]:
```
%timeit backtracking_line_search(X, y, c=0.01, b=0.5, num_step=1000)
```

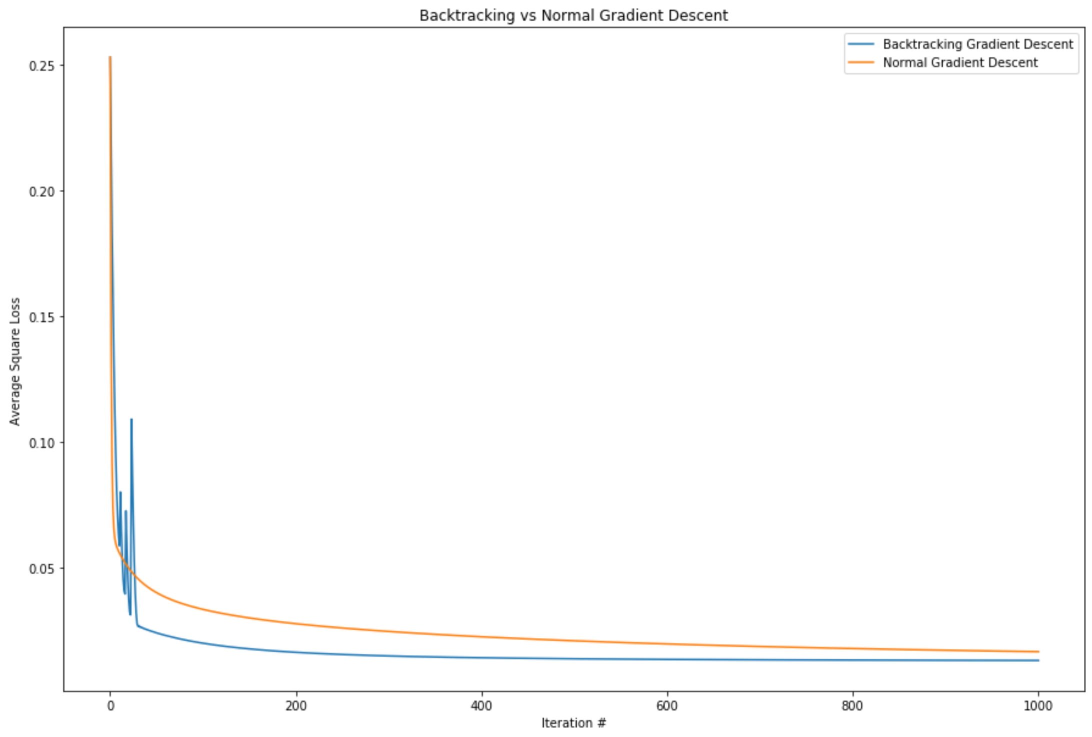
```
262 ms ± 43.4 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
In [10]: %timeit batch_grad_descent(X, y, alpha=0.1, num_step=1000, grad_check=False
         )

         43.8 ms ± 1.63 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
In [11]: hist_backtrack, loss_hist_backtrack = backtracking_line_search(X, y, max_al
         pha=1, c=0.01, b=0.1, num_step=1000)
```

```
In [12]: theta_hist, loss_hist = batch_grad_descent(X, y, alpha=0.01, num_step=1000,
         grad_check=False)
```

```
In [13]: plt.plot(loss_hist_backtrack, label='Backtracking Gradient Descent')
         plt.plot(loss_hist, label='Normal Gradient Descent')
         plt.title("Backtracking vs Normal Gradient Descent")
         plt.xlabel('Iteration #')
         plt.ylabel('Average Square Loss')
         plt.legend()

Out[13]: <matplotlib.legend.Legend at 0x7f79f9292978>
```



The backtracking algorithm is about 6x slower to run 1000 iterations. It is worth noting that the backtracking algorithm slows down as it approaches the optimal solution. This is due to the algorithm needing more iterations to shrink the step size to an appropriate level. However, in terms of iterations, the backtracking algorithm much faster. After 1000 iterations, the normal gradient descent algorithm performs about as well as the backtracking algorithm does in 50 iterations.

3.5 Ridge Regression

3.5.1

$$\nabla J(\theta) = \frac{1}{m}(X\theta - y)^T X + 2\lambda\theta^T$$
$$\theta \leftarrow \theta - \alpha\nabla J(\theta)$$

3.5.2

```
#######################################
### The gradient of regularized batch gradient descent
def compute_regularized_square_loss_gradient(X, y, theta, lambda_reg):
    """
    Compute the gradient of L2-regularized average square loss function given X,
y and theta

    Args:
        X - the feature vector, 2D numpy array of size (num_instances, num_featur
es)
        y - the label vector, 1D numpy array of size (num_instances)
        theta - the parameter vector, 1D numpy array of size (num_features)
        lambda_reg - the regularization coefficient

    Returns:
        grad - gradient vector, 1D numpy array of size (num_features)
    """
    square_loss_gradient = compute_square_loss_gradient(X, y, theta)
    regularization_term = 2 * lambda_reg * theta.T
    return square_loss_gradient + regularization_term
```

3.5.3

```
#######################################
### Regularized batch gradient descent
def regularized_grad_descent(X, y, alpha=0.05, lambda_reg=10**-2, num_step=1000):
    """
    Args:
        X - the feature vector, 2D numpy array of size (num_instances, num_featur
es)
        y - the label vector, 1D numpy array of size (num_instances)
        alpha - step size in gradient descent
        lambda_reg - the regularization coefficient
        num_step - number of steps to run

    Returns:
        theta_hist - the history of parameter vector, 2D numpy array of size (num
_step+1, num_features)
                     for instance, theta in step 0 should be theta_hist[0], theta
in step (num_step+1) is theta_hist[-1]
        loss hist - the history of average square loss function without the regul
arization term, 1D numpy array.
    """
    num_instances, num_features = X.shape[0], X.shape[1]
    theta = np.zeros(num_features) #Initialize theta
    theta_hist = np.zeros((num_step+1, num_features)) #Initialize theta_hist
    loss_hist = np.zeros(num_step+1) #Initialize loss_hist

    for i in range(num_step+1):
        loss_hist[i] = compute_square_loss(X, y, theta)
        theta_hist[i] = theta

        grad = compute_regularized_square_loss_gradient(X, y, theta, lambda_reg)

        theta -= grad*alpha

    return theta_hist, loss_hist
```

3.5.4

Increasing the value of B results in the bias term having a lower coefficient. This decreases the amount that the bias contributes to the overall loss.

3.5.5

Let $X \in R^{m,n+1}$ such that $X_{i,1} = B \; \forall \; i$. That is, the first column of X contains only B. Given

$$J(\theta) = \frac{1}{m}(X^T\theta - y)^T(X^T\theta - y) + \lambda\theta^T\theta = \frac{1}{m}(X^T\theta - y)^T(X^T\theta - y) + \Omega(\theta),$$

we now claim that as $B \to \infty$, $\frac{\partial\Omega(\theta)}{\partial\theta_1} \to 0$. In other words, as $B \to \infty$, the amount of regularization it experiences approaches 0.

Shitty Proof:

$\exists$ some constant $c = X^{(1)^T}\theta_1^*$ that optimizes $J(\theta)$ with respect to $X^{(1)}$

Now note that

$$\frac{\partial\Omega(\theta)}{\partial\theta_1} = 2\lambda\theta_1$$

Now since c is a constant, as $B \to \infty, \theta_1^* \to 0$ and therefore $\frac{\partial\Omega(\theta)}{\partial\theta_1^*} \to 0$

3.5.6

In [14]:
```
X, y, X_test, y_test = load_data()
```

In [15]:
```
Bs = [1,2,3,4,5,6,7,8,9,10]
test_losses = []
for B in Bs:
    X_ = add_bias(X, b=B)
    X_test_ = add_bias(X_test, b=B)
    theta_hist, loss_hist = regularized_grad_descent(X_, y, alpha=0.01)
    theta = theta_hist[-1]
    test_losses.append(compute_square_loss(X_test_, y_test, theta))
```

In [16]:
```
test_losses
```

Out[16]:
```
[0.027767567458916852,
 0.02704512786076287,
 0.026916670073004696,
 0.026873360766006098,
 0.026853768609246494,
 0.026843265680191308,
 0.026836983619430516,
 0.026832927574705828,
 0.02683015665211446,
 5.295606270673492e+210]
```

Test set performance was best when B=9. Beyond that loss increased rapidly.

3.5.7

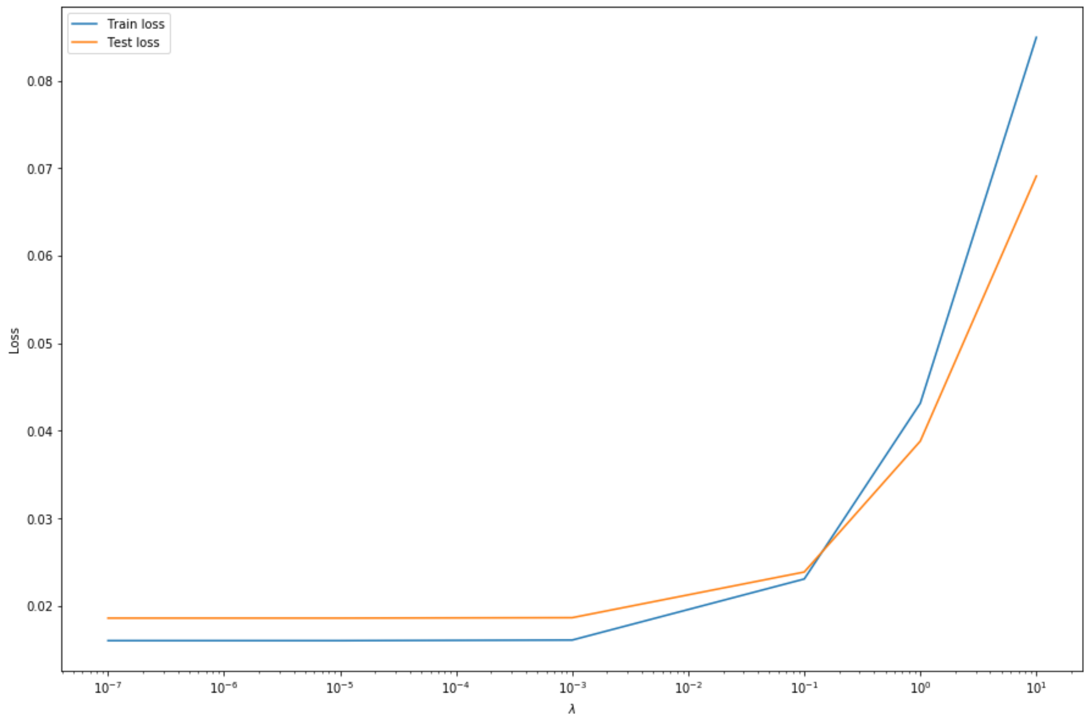
In [17]:
```
# Reload fresh data
X, y, X_test, y_test = load_data()
# Set B=1
X_ = add_bias(X, b=1)
X_test_ = add_bias(X_test, b=1)
```

In [18]:
```
lambdas = [1e-7, 1e-5, 1e-3, 1e-1, 1, 10]
train_losses = []
test_losses = []
```

In [19]:
```
for l in lambdas:
    theta_hist, loss_hist = regularized_grad_descent(X_, y, alpha=0.01, lam
bda_reg=l)
    theta = theta_hist[-1]
    test_losses.append(compute_square_loss(X_test_, y_test, theta))
    train_losses.append(compute_square_loss(X_, y, theta))
```

In [20]:
```
ax = plt.plot(lambdas, train_losses, label='Train loss')
plt.plot(lambdas, test_losses, label='Test loss')
plt.xscale('log')
plt.legend()
plt.xlabel('$\lambda$')
plt.ylabel('Loss')
```

Out[20]:
```
Text(0, 0.5, 'Loss')
```

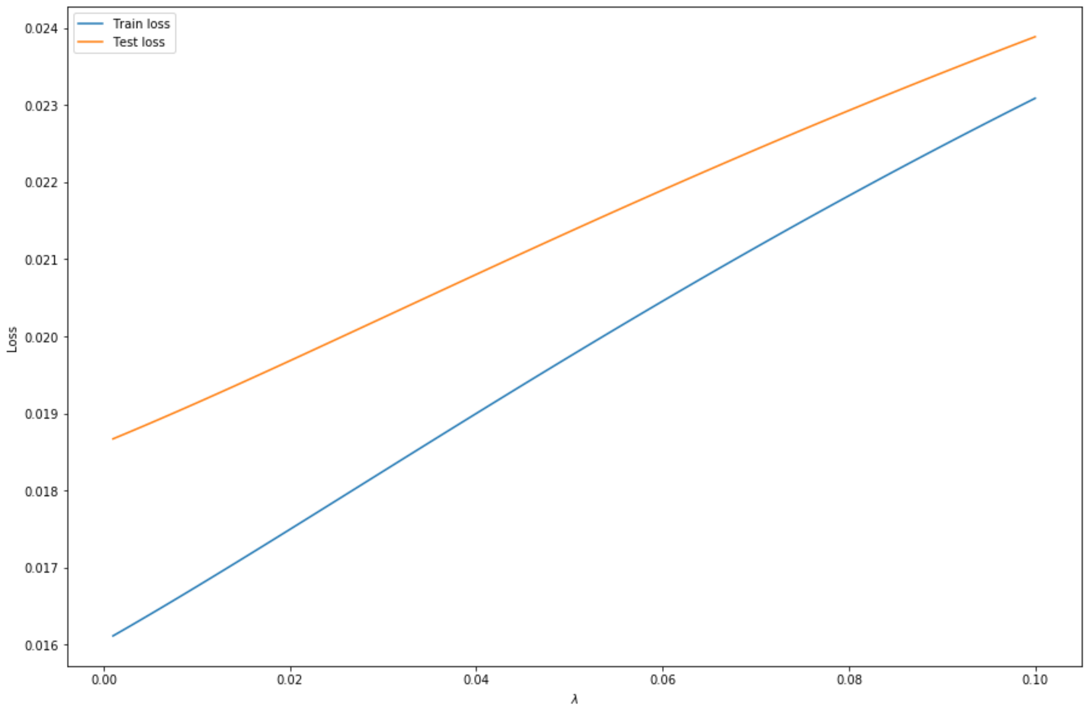


In [21]:
```
lambdas = np.linspace(1e-3, 1e-1, 100)
train_losses = []
test_losses = []
```

```
In [22]: for l in lambdas:
             theta_hist, loss_hist = regularized_grad_descent(X_, y, alpha=0.01, lam
         bda_reg=l)
             theta = theta_hist[-1]
             test_losses.append(compute_square_loss(X_test_, y_test, theta))
             train_losses.append(compute_square_loss(X_, y, theta))
```

```
In [23]: ax = plt.plot(lambdas, train_losses, label='Train loss')
         plt.plot(lambdas, test_losses, label='Test loss')
         #plt.xscale('log')
         plt.legend()
         plt.xlabel('$\lambda$')
         plt.ylabel('Loss')
```

```
Out[23]: Text(0, 0.5, 'Loss')
```



$\lambda = 0$ minimizes loss on the test set.

3.5.8

For deployment, I would use $\lambda = 0.02$. This allows for some regularization without too much of a loss in performance.

3.6 Stochastic Gradient Descent

3.6.1

$$f_i(\theta) = (h_\theta(x_i) - y)^2 + \lambda\theta^T\theta$$

3.6.2

Let

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} f_i(\theta)$$

Then by taking the gradient we see that

$$\nabla J(\theta) = \frac{1}{m}\sum_{i=1}^{m} \nabla f_i(\theta)$$

Now observe that

$$\begin{aligned} \mathbb{E}[\nabla f_i(\theta)] &= \frac{1}{m}\sum_{i=1}^{m} \nabla f_i(\theta) \qquad \text{by definition of expected value,} \\ &= \nabla J(\theta) \end{aligned}$$

3.6.3

$$\theta \leftarrow \theta - \alpha[2(x_i\theta - y_i)x_i + 2\lambda\theta]$$

3.6.4

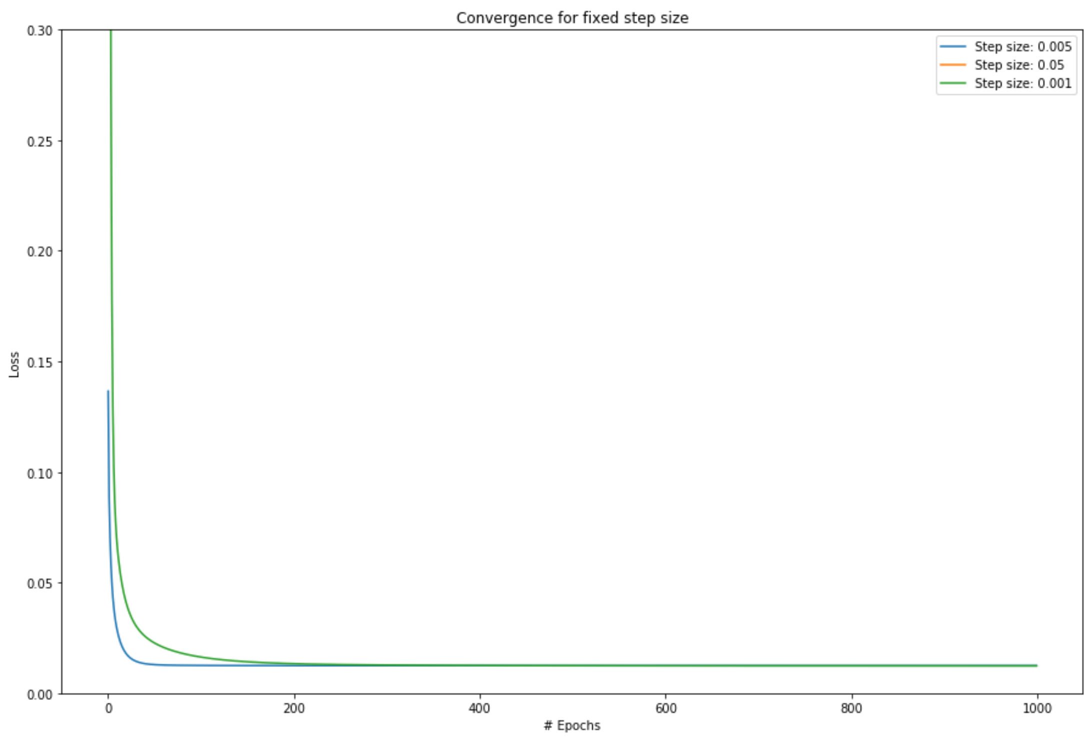
```
#######################################
### Stochastic gradient descent
def stochastic_grad_descent(X, y, alpha=0.01, lambda_reg=10**-2, num_epoch=1000,
C=0.1, averaged=False, eta_0=None):
    """
    In this question you will implement stochastic gradient descent with regulari
zation term

    Args:
        X - the feature vector, 2D numpy array of size (num_instances, num_featur
es)
        y - the label vector, 1D numpy array of size (num_instances)
        alpha - string or float, step size in gradient descent
                NOTE: In SGD, it's not a good idea to use a fixed step size. Usua
lly it's set to 1/sqrt(t) or 1/t
                if alpha is a float, then the step size in every step is the floa
t.
                if alpha == "1/sqrt(t)", alpha = 1/sqrt(t).
                if alpha == "1/t", alpha = 1/t.
        lambda_reg - the regularization coefficient
        num_epoch - number of epochs to go through the whole training set

    Returns:
        theta_hist - the history of parameter vector, 3D numpy array of size (num
_epoch, num_instances, num_features)
                     for instance, theta in epoch 0 should be theta_hist[0], thet
a in epoch (num_epoch) is theta_hist[-1]
        loss hist - the history of loss function vector, 2D numpy array of size (
num_epoch, num_instances)
    """
    num_instances, num_features = X.shape[0], X.shape[1]
    theta = np.ones(num_features) #Initialize theta

    theta_hist = np.zeros((num_epoch, num_instances, num_features)) #Initialize t
heta_hist
    loss_hist = np.zeros((num_epoch, num_instances)) #Initialize loss_hist

    t=1
    mode=alpha

    for i in range(num_epoch):

        for j, (x_j, y_j) in enumerate(zip(X, y)):
            x_j = np.array([x_j])
            y_j = np.array([y_j])
            grad = compute_regularized_square_loss_gradient(x_j, y_j, theta, lamb
da_reg)

            # Adaptive step size methods
            if mode == '1/sqrt(t)':
                alpha = C/math.sqrt(t)
            if mode == '1/t':
                alpha = C/t
```

3.6.5

```
In [24]: l=0.008
         B=9
         # Reload fresh data
         X, y, X_test, y_test = load_data()
         # Set B=1
         X_ = add_bias(X, b=1)
         X_test_ = add_bias(X_test, b=1)
```

```
In [25]: test_losses=[]
         train_losses=[]
         ayes = [0.005, 0.05, 0.001]
         for a in ayes:
             theta_hist, loss_hist = stochastic_grad_descent(X_, y, alpha=a, lambda_
         reg=l)
             train_losses.append(loss_hist)

         /home/cfizette/NYU/NYU-Data-Science/DS-1003/hw/hw1/hw1_code.py:81: RuntimeW
         arning: overflow encountered in multiply
           return (2/n) * np.matmul(X.T, y_pred - y)
```

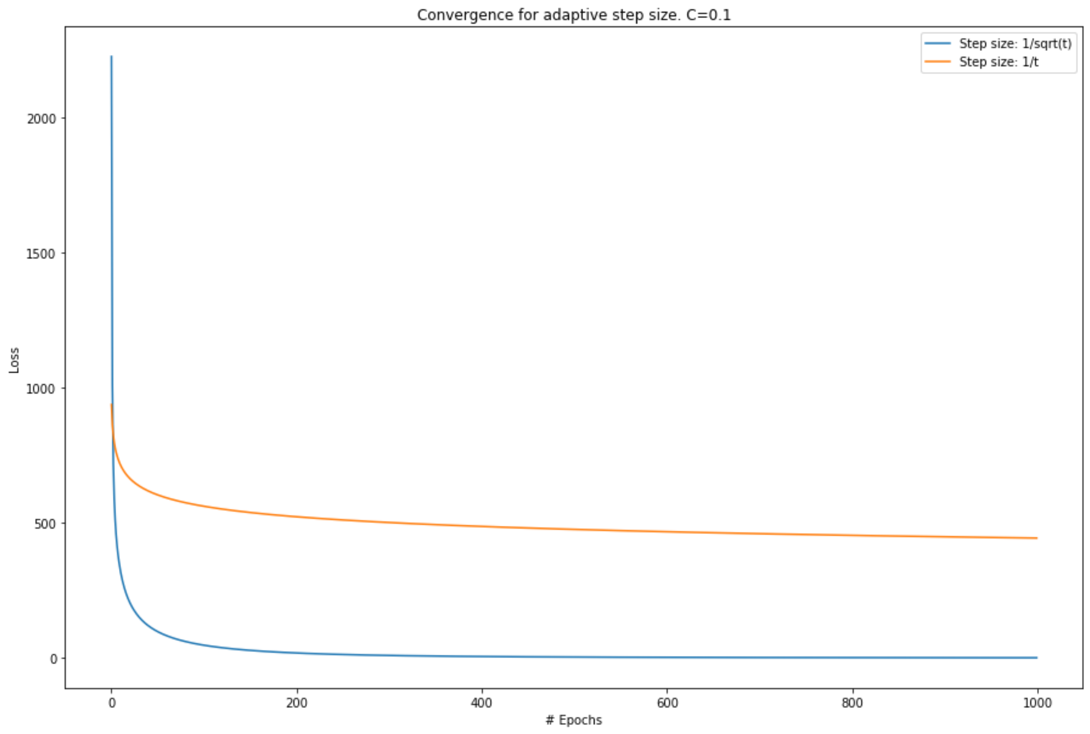
```
In [26]: for a, loss in zip(ayes, train_losses):
             plt.plot(loss[:,-1], label='Step size: {}'.format(a))
         plt.legend()
         plt.xlabel('# Epochs')
         plt.ylabel('Loss')
         plt.ylim(0,0.3)
         plt.title('Convergence for fixed step size')

Out[26]: Text(0.5, 1.0, 'Convergence for fixed step size')
```



In [27]:
```
test_losses=[]
train_losses=[]
ayes = ['1/sqrt(t)', '1/t']
C=0.1
for a in ayes:
    theta_hist, loss_hist = stochastic_grad_descent(X_, y, alpha=a, lambda_
reg=l)
    train_losses.append(loss_hist)
```

In [28]:
```
for a, loss in zip(ayes, train_losses):
    plt.plot(loss[:,-1], label='Step size: {}'.format(a))
plt.legend()
plt.xlabel('# Epochs')
plt.ylabel('Loss')
plt.title('Convergence for adaptive step size. C=0.1')
```

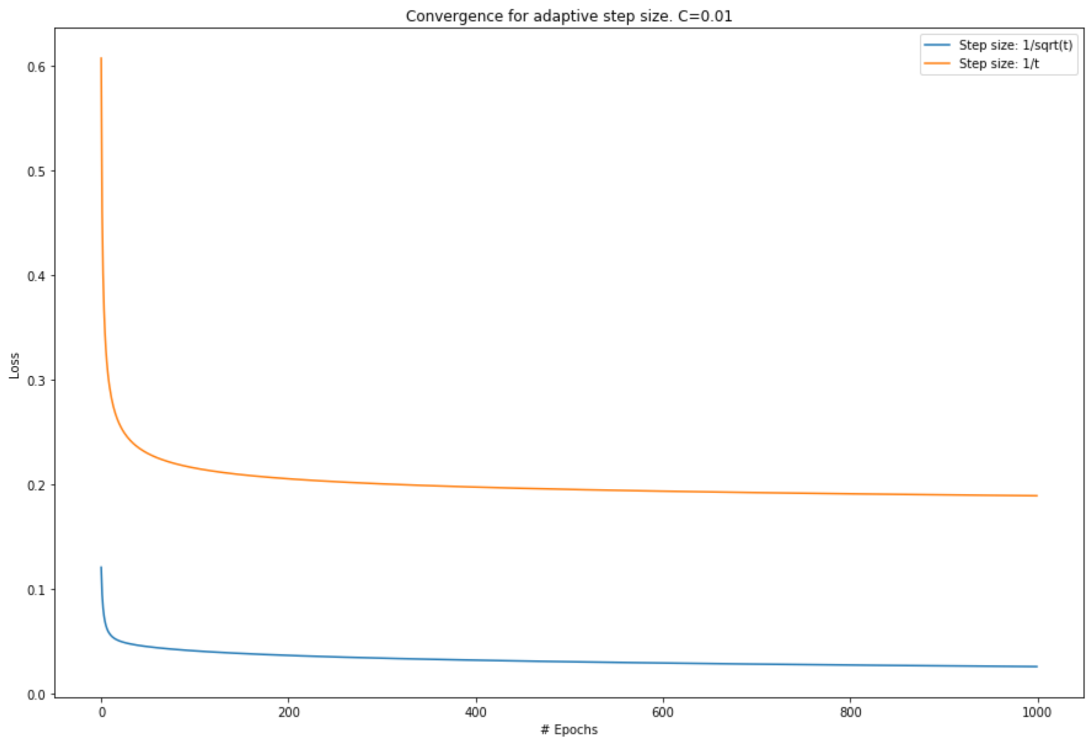
Out[28]:
```
Text(0.5, 1.0, 'Convergence for adaptive step size. C=0.1')
```



In [29]:
```
test_losses=[]
train_losses=[]
ayes = ['1/sqrt(t)', '1/t']
C=0.01
for a in ayes:
    theta_hist, loss_hist = stochastic_grad_descent(X_, y, alpha=a, lambda_
reg=l, C=C)
    train_losses.append(loss_hist)
```

```
In [30]: for a, loss in zip(ayes, train_losses):
             plt.plot(loss[:,-1], label='Step size: {}'.format(a))
         plt.legend()
         plt.xlabel('# Epochs')
         plt.ylabel('Loss')
         plt.title('Convergence for adaptive step size. C=0.01')
```

```
Out[30]: Text(0.5, 1.0, 'Convergence for adaptive step size. C=0.01')
```



With a smaller value of C=0.01, convergence occurs faster.

Averaged SGD

```
In [31]: theta, _ = stochastic_grad_descent(X_ ,y, alpha='1/sqrt(t)', C=0.01, averag
         ed=True)
         loss_averaged = compute_square_loss(X_, y, theta)
         theta_hist, _ = stochastic_grad_descent(X_ ,y, alpha='1/sqrt(t)', C=0.01, a
         veraged=False)
         loss_not_averaged = compute_square_loss(X_, y, theta_hist[-1,-1,:])

         print('Loss with averaging: {} \nLoss without averaging{}'.format(loss_aver
         aged, loss_not_averaged))
```

```
Loss with averaging: 0.03135260773407746
Loss without averaging0.025432554368879978
```

The averaged SGD actually performs worse in this case.

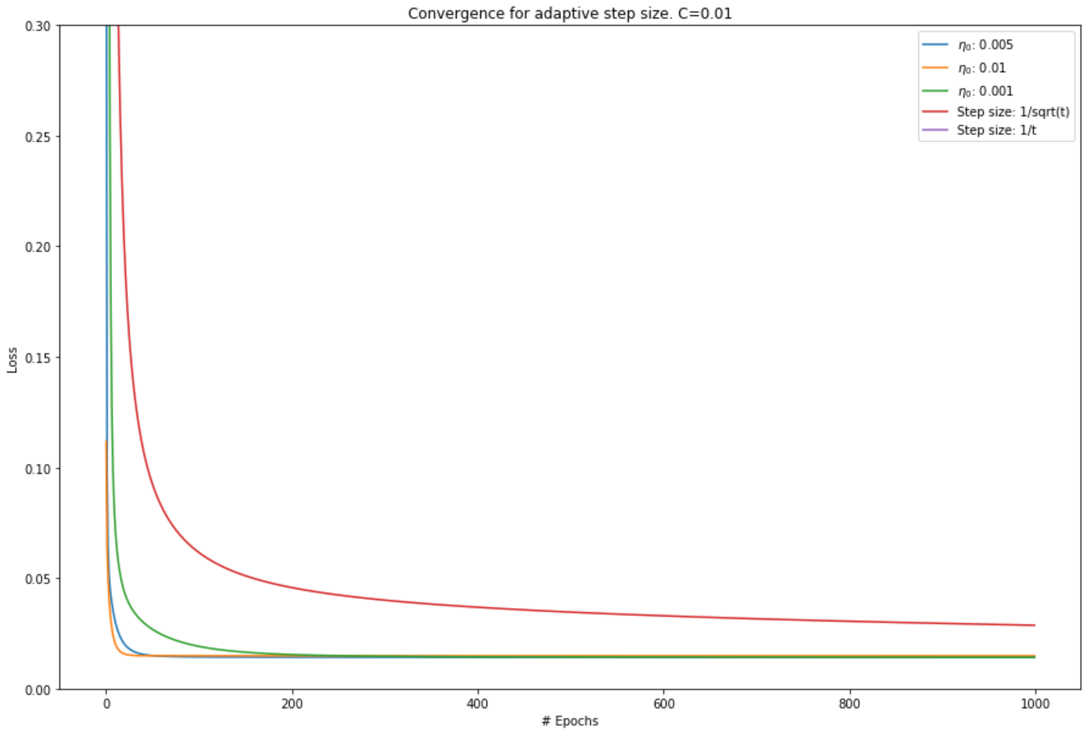
3.6.6

In [32]:
```
l=0.008
B=9
# Reload fresh data
X, y, X_test, y_test = load_data()
# Set B=1
X_ = add_bias(X, b=1)
X_test_ = add_bias(X_test, b=1)
```

In [33]:
```
test_losses=[]
train_losses=[]
etas = [0.005, 0.01, 0.001]
for eta in etas:
    theta_hist, loss_hist = stochastic_grad_descent(X_, y, lambda_reg=l, et
a_0=eta)
    train_losses.append(loss_hist)
```

In [34]:
```
test_losses=[]
train_losses=[]
etas = [0.005, 0.01, 0.001]
for eta in etas:
    theta_hist, loss_hist = stochastic_grad_descent(X_, y, lambda_reg=l, et
a_0=eta)
    train_losses.append(loss_hist)

for eta, loss in zip(etas, train_losses):
    plt.plot(loss[:,-1], label='$\eta_0$: {}'.format(eta))
plt.legend()
plt.xlabel('# Epochs')
plt.ylabel('Loss')
plt.ylim(0,0.3)
plt.title('Convergence for different values of $\eta_0$')

# Compare to other adaptive step size methods ----------------------------
-------------
test_losses=[]
train_losses=[]
ayes = ['1/sqrt(t)', '1/t']
C=0.01
for a in ayes:
    theta_hist, loss_hist = stochastic_grad_descent(X_, y, alpha=a, lambda_
reg=l, C=C)
    train_losses.append(loss_hist)

for a, loss in zip(ayes, train_losses):
    plt.plot(loss[:,-1], label='Step size: {}'.format(a))
plt.legend()
plt.xlabel('# Epochs')
plt.ylabel('Loss')
plt.title('Convergence for adaptive step size. C=0.01')
```

Out[34]:
```
Text(0.5, 1.0, 'Convergence for adaptive step size. C=0.01')
```



This method appears to perform better than some of the other adaptive step size methods we tried, however, the difference in performance could decrease with proper parameter tuning.

4. Risk Minimizatiion

4.1 Square Loss

4.1.1

Find argmin by setting derivative equal to zero and solving for a

$$\begin{aligned} E(\ell') &= 2E(a-y), \\ &= 2(a-Ey) \end{aligned}$$

Setting this equal to 0

$$\begin{aligned} 2(a-Ey) &= 0, \\ a^* &= Ey \end{aligned}$$

Now also observe that

$$\begin{aligned} E(\ell(a^*-y)) &= E((a^*-y)^2) \\ &= E[(Ey-y)^2] \\ &= E[E^2y - 2yEy + y^2] \\ &= E^2y - 2E^2y + Ey^2 \\ &= Ey^2 - E^2y \\ &= Var(y) \end{aligned}$$

4.1.2a

$$\begin{aligned} f^*(x) &= argmin_a E[(a-y)^2|X] \\ &= E(Y|X) \end{aligned}$$

4.1.2b

$$\begin{aligned} E[(f^*(x)-y)^2] &= E[E[f^*(x)-y)^2|X]] \\ &\leq E[E[(f(x)-y)^2|X]] \\ &= E[(f(x)-y)^2] \end{aligned}$$

Thus,

$$E[(f^*(x)-y)^2] \leq E[(f(x)-y)^2]$$