NAME: Parthvi Shah NetID: pss434

COMPUTING RISK!!!

Question 1

In [59]:
```
from IPython.display import Image
Image("/content/sample_data/Screen Shot 2020-02-22 at 9.39.15 AM.png")
```

Out[59]:

Computing Risk.

1. $x \in R^n$ $\{-2, -1, 0, 1, 2\}$ $P = 1/5$

(a) $E[\|x\|_2^2]$

$\rightarrow E[x_1^2 + x_2^2 + \cdots x_n^2]$

$\rightarrow E[x_1^2] + E[x_2^2] + \cdots E[x_n^2]$

$\rightarrow n E[x_i^2]$

$\rightarrow n\left[(-2)^2 \cdot \frac{1}{5} + (2)^2 \frac{1}{5} + (-1)^2 \frac{1}{5} + 0 + (1)^2 \frac{1}{5}\right].$

$\rightarrow \underline{2n}$

(b) $E[\|x\|_{max}]$

$\rightarrow E[\max(|x_1|, |x_2|, \ldots, |x_n|)]$

$\rightarrow \sum_{i=0}^{2} i \cdot p(\max(|x_1| \ldots |x_n|))$

$P(0) = \left(\frac{1}{5}\right)^n$

$P(\max = 2) = 1 - \left(\frac{3}{5}\right)^n$

$P(\max = 1) = \left(\frac{3}{5}\right)^n - \left(\frac{1}{5}\right)^n$

$\therefore E[\|x\|_\infty] = 1 \cdot \left(1 - \left(\left(\frac{3}{5}\right)^n\right) - \left(\frac{1}{5}\right)^n\right) + \left(\frac{3}{5}\right)^n \cdot 2$

Question 1c an 2a

In [62]:
```
Image("/content/sample_data/Screen Shot 2020-02-22 at 9.40.01 AM.png")
```

Out[62]:

c Since x_i's are iids

$cov(x_i, x_j) = Var(x_i) - E(x_i^2) - E^2(x) = 2 - 0 = 2$

$E^2(x_i) = 0$

$E(x_i^2) = 2$

$$\therefore \begin{bmatrix} 2 & & & (0) \\ 0 & 2 & 0 & 0 \\ 0 & 0 & \ddots & 0 \\ (0) & & & 2 \end{bmatrix}$$

1.2

(a) $E[(a-y)^2] = E[a^2 - 2ay + y^2]$

$= E[a^2] + E[y^2] - 2a\,E[y]$

$= Var(a) + var(y) + E[a^2]$

$+ E[y^2] - 2a\,E[y]$

$= var(y) + (a - E[y])^2$

$\therefore$ To minimize $E(a-y)^2$; $a = E[y]$

with risk of $var(y)$

Prove that Bayes risk $= var(y)$

$\longrightarrow$

$E(a^* - y)^2 = E(E(y) - y)^2$

$= E[E(y)^2 - 2yE(y) + y^2]$

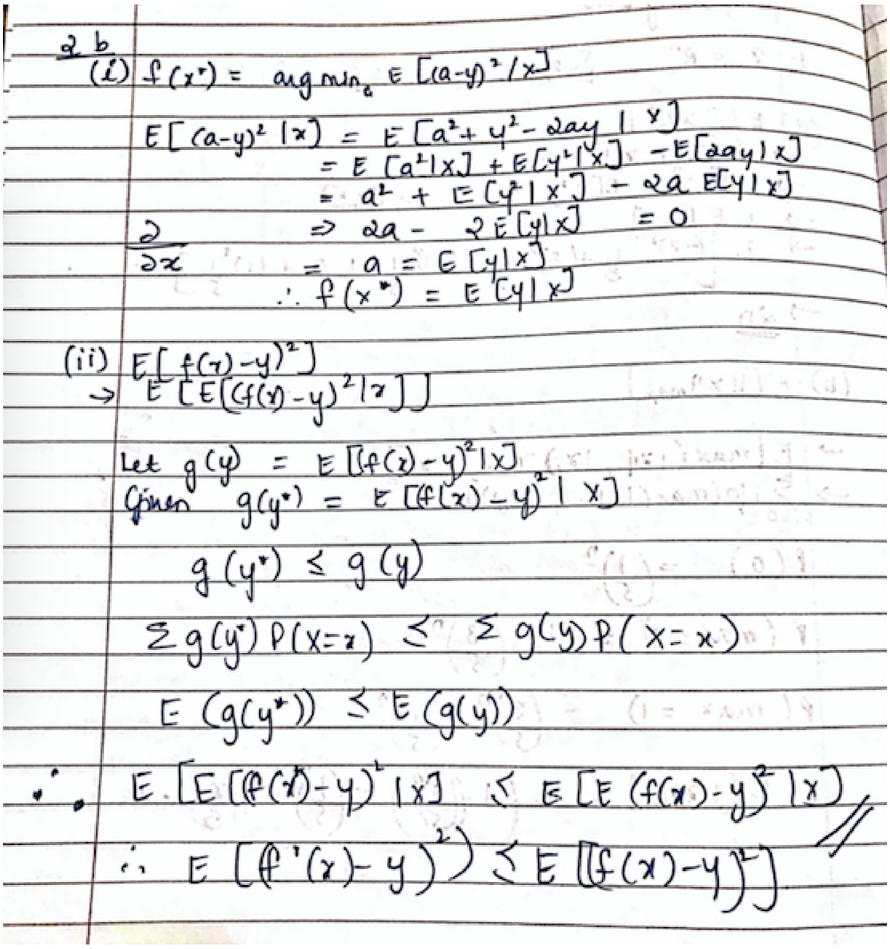
$= E[y]^2 - 2E(y)E(y) + E\,y^2$

$= \underline{Var(y)}$

Question 2b

```
In [63]: Image("/content/sample_data/Screen Shot 2020-02-22 at 9.41.10 AM.png")
```

Out[63]:



2 b

(i) $f(x^*) = \arg\min_a E[(a-y)^2/x]$

$E[(a-y)^2 | x] = E[a^2 + y^2 - 2ay \mid x]$
$= E[a^2|x] + E[y^2|x] - E[2ay|x]$
$= a^2 + E[y^2|x] - 2a\,E[y|x]$

$\frac{\partial}{\partial x} \Rightarrow 2a - 2E[y|x] = 0$
$\Rightarrow a = E[y|x]$
$\therefore f(x^*) = E[y|x]$

(ii) $E[(f(x)-y)^2]$
$\rightarrow E[E[(f(x)-y)^2|x]]$

Let $g(y) = E[(f(x)-y)^2|x]$
Given $g(y^*) = E[(f(x)-y)^2 \mid x]$

$g(y^*) \leq g(y)$

$\sum g(y^*) P(X=x) \leq \sum g(y) P(X=x)$

$E(g(y^*)) \leq E(g(y))$

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

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

```
In [0]: import sys
        import numpy as np
        import pandas as pd
        import matplotlib.pyplot as plt
        from sklearn.model_selection import train_test_split
```

```
In [0]: df = pd.read_csv('/content/sample_data/ridge_regression_dataset.csv', delimiter=',')
```

LINEAR REGRESSION:

1. Normalization :

Numpy's broadcasting can be used here. It makes the calculations easier. It handles array with various sizes and performs calculations which can be used here.

In [0]:

```
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_features)
        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
    """
    train_normedlist = []
    for i,v in enumerate(train):
        result = all(elem == train[i][0] for elem in train[i])
        if result == True:
            train[i] = train[i]
            train_normedlist.append(train[i])
        else:
            for k,v in enumerate(train[i]):
                train[i][k] = (train[i][k] - min(train[i])) / (max(train[i]) - min(train[i
]))
            train_normedlist.append(train[i])

    test_normedlist = []
    for i,v in enumerate(test):
        result = all(elem == test[i][0] for elem in test[i])
        if result == True:
            test[i] = test[i]
            test_normedlist.append(test[i])
        else:
            for k,v in enumerate(test[i]):
                test[i][k] = (test[i][k] - min(train[i])) / (max(train[i]) - min(train[i]))
            test_normedlist.append(test[i])



    return np.array(train_normedlist), np.array(test_normedlist)
```

In [0]:

```
X = df.values[:,:-1]
y = df.values[:,-1]

print('Split into Train and Test')
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size =100, random_state=10)

print("Scaling all to [0, 1]")
X_train, X_test = feature_normalization(X_train, X_test)
X_train = np.hstack((X_train, np.ones((X_train.shape[0], 1))))  # Add bias term
X_test = np.hstack((X_test, np.ones((X_test.shape[0], 1))))  # Add bias term'''
```

```
Split into Train and Test
Scaling all to [0, 1]
```

1. a

Made Design Matrix. dmX (Design Matrix X) = X_train theta = theta response = y_train

In [0]:

```
m =X_train.shape[0]

dmX = np.zeros((m,X_train.shape[1]))
dmX = np.array(X_train)

theta = np.zeros((X_train.shape[1]))
```

In [0]:

```
response=np.ones((m,1))
response = y_train
b = np.ones((m,1))
```

In [0]:

```
dmX.shape
```

Out[0]:

```
(100, 49)
```

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

1. b

> Indented block

gradientjtheta = (2/m)*((np.dot((response - (np.dot(dmX, theta))),dmX)))

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

1. c

In [0]:
```
eta = np.random.rand()
h = np.random.rand(X_train.shape[1],1)
length = eta*np.abs(h)
thetadash = theta+ length
```

$$\theta' = \theta + length$$

$$J(\theta + \eta h) - J(\theta) = \frac{1}{m}(x\theta' - y)^T(x\theta' - y) - J(\theta)$$

1. d

$$\theta = \theta - \frac{\eta}{m}(x\theta - y)^T x$$

1. e

In [0]:
```
def compute_square_loss(dmX, response, theta):
    m = dmX.shape[0]
    costfntheta = (1/m)*(np.dot((np.dot(dmX,theta) - response),(np.dot(dmX,theta) - respons
e)))
    return costfntheta
```

In [0]:
```
compute_square_loss(dmX,response,theta)
```

Out[0]: 7.961518343622414

1. f

In [0]:
```
def compute_square_loss_gradient(X, y, theta):
    iter = X.shape[0]

    gradientjtheta = ((2/iter)*(np.dot(((np.dot(X, theta)) - y ),X)))

    #ones = np.ones(num_instances).reshape(num_instances,1)
    #extended_X = np.append(X,ones,axis = 1)
    '''differences = np.dot(X,theta)-y
    grad = 1.0/num_instances*(np.dot(differences,X))'''
    return gradientjtheta
```

In [0]:
```
compute_square_loss_gradient(dmX, response, theta).shape
```

Out[0]: (49,)

1. a

In [0]:
```
#See http://ufldl.stanford.edu/wiki/index.php/Gradient_checking_and_advanced_optimization
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_features)
        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)
    e=np.zeros((num_features,1))
    for i in range(num_features):
        e=np.zeros(num_features)
        e[i] = 1
        thetaplus = theta+epsilon * e
        thetaminus = theta-epsilon * e
        approx_grad[i] = (compute_square_loss(X, y, thetaplus) - compute_square_loss(X, y,
thetaminus))/(2*epsilon)
    distance = np.linalg.norm(approx_grad-true_gradient)
    return distance<tolerance


    #thetaplusep, thetaminusep = (theta+epsilon),(theta-epsilon)

    #TODO
```

In [0]:
```
grad_checker(dmX, response, theta, epsilon=0.01, tolerance=1e-4)
```

Out[0]: True

1. a

In [0]:
```python
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_features)
        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 when updating

    Returns:
        theta_hist - the history of parameter vector, 2D numpy array of size (num_step+1, n
um_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 array, (num_st
ep+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(1,num_step+1):
        grad = compute_square_loss_gradient(X, y, theta)
        theta = theta-alpha*grad
        loss_hist[i] = compute_square_loss(X,y,theta)
        theta_hist[i] = theta
    return theta_hist, loss_hist
```

In [0]:
```python
theta_hist, loss_hist = batch_grad_descent(X, y, alpha=0.1, num_step=1000, grad_check=False
)
```

```
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:24: RuntimeWarning: invalid v
alue encountered in subtract
```

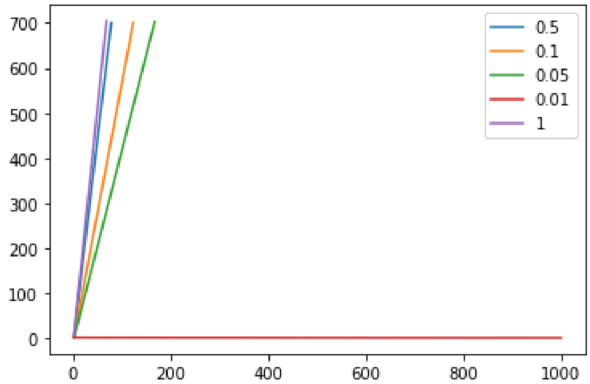
1. b

In [0]:
```python
step_size = [0.5, 0.1, .05, .01, 1]
```

In [0]:
```
for i in range(len(step_size)):
    theta_hist, loss_hist = batch_grad_descent(X, y, alpha=step_size[i], num_step=1000, gra
d_check=False)
    plt.plot(np.log(loss_hist), label = step_size[i])
plt.legend()
```

```
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:3: RuntimeWarning: divide by
zero encountered in log
  This is separate from the ipykernel package so we can avoid doing imports until
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:24: RuntimeWarning: invalid v
alue encountered in subtract
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:3: RuntimeWarning: divide by
zero encountered in log
  This is separate from the ipykernel package so we can avoid doing imports until
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:24: RuntimeWarning: invalid v
alue encountered in subtract
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:3: RuntimeWarning: divide by
zero encountered in log
  This is separate from the ipykernel package so we can avoid doing imports until
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:3: RuntimeWarning: divide by
zero encountered in log
  This is separate from the ipykernel package so we can avoid doing imports until
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:3: RuntimeWarning: divide by
zero encountered in log
  This is separate from the ipykernel package so we can avoid doing imports until
```

Out[0]: <matplotlib.legend.Legend at 0x7f74cae0cba8>



Larger the steps, the faster the loss function value decreases. When step size > 0.01, it starts to diverge.

RIDGE REGRESSION!

1.

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

2.

In [0]:
```
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 thet
a

    Args:
        X - the feature vector, 2D numpy array of size (num_instances, num_features)
        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)

    """


    ridgecostfntheta = compute_square_loss_gradient(X, y, theta) + (lambda_reg*2*theta)
    return ridgecostfntheta
```

In [0]:
```
ridgecostfn = compute_regularized_square_loss_gradient(dmX, response, theta, lambda_reg = 2
)
```

3.

In [0]:
```
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_features)
        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, n
um_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 regularization
 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(1, num_step+1):
        loss_hist[i] = compute_square_loss(X,y,theta)

        grad = compute_regularized_square_loss_gradient(X, y, theta, lambda_reg)
        theta = theta-alpha*grad
        theta_hist[i] = theta


    return theta_hist, loss_hist
```

In [0]:
```
theta_histridge, loss_histridge = regularized_grad_descent(dmX, response, alpha=0.05, lambd
a_reg=10**-2, num_step=1000)
```

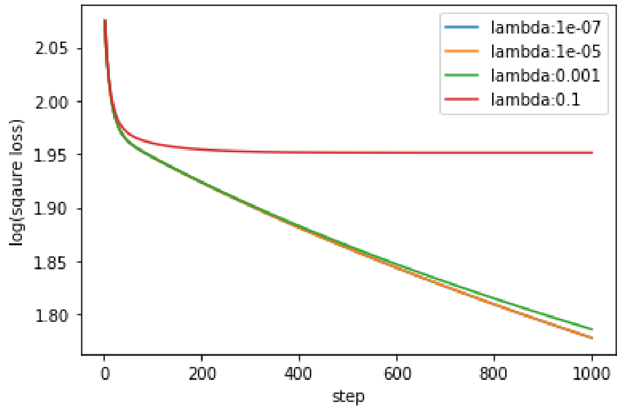
4.

Since B multiplied by the last theta term which is being adjusted. For larger B, the magnitude of theta adjusting is small and the regularization terms is small compared to the entire regularization. Bias is used to regularize the training set.

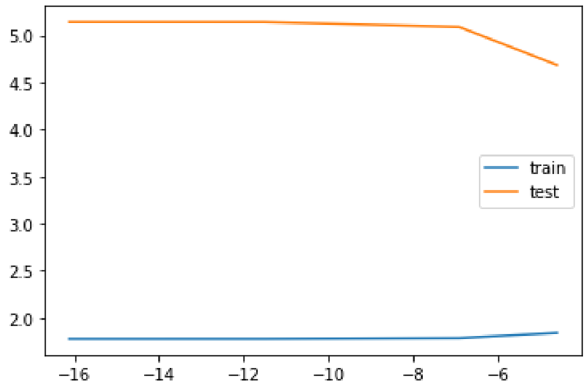
5.

In [0]:
```
for lambda_reg in [10**-7,10**-5,10**-3,10**-1]:
    theta_hist,loss_hist= regularized_grad_descent(X_train,y_train,alpha=0.05,lambda_reg=la
mbda_reg)
    plt.plot(range(len(loss_hist)),np.log(loss_hist),label='lambda:'+str(lambda_reg))
    plt.xlabel("step")
plt.ylabel("log(sqaure loss)")
plt.legend()
plt.show()
```

```
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:3: RuntimeWarning: divide by
zero encountered in log
  This is separate from the ipykernel package so we can avoid doing imports until
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:3: RuntimeWarning: divide by
zero encountered in log
  This is separate from the ipykernel package so we can avoid doing imports until
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:3: RuntimeWarning: divide by
zero encountered in log
  This is separate from the ipykernel package so we can avoid doing imports until
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:3: RuntimeWarning: divide by
zero encountered in log
  This is separate from the ipykernel package so we can avoid doing imports until
```



In [0]:
```
loss_train = []
loss_test = []
lambda_reg =  [10**-7,10**-5,10**-3,0.01]
for i in lambda_reg:
    theta_hist,loss_hist= regularized_grad_descent(dmX,response,lambda_reg=i)
    newtheta = theta_hist[-1]
    loss_hist_train = compute_square_loss(dmX,response,newtheta)
    loss_train.append(loss_hist_train)
    loss_hist_test= compute_square_loss(X_test,y_test,newtheta)
    loss_test.append(loss_hist_test)
plt.plot(np.log(lambda_reg),np.log(loss_train), label = 'train')
plt.plot(np.log(lambda_reg),np.log(loss_test), label = 'test')
plt.legend()
plt.show()
```



I would select theta with value where lambda is 1e-5. It converges the fastest.

Stochastic Gradient Descent

1.

$$fi\theta = \frac{1}{m}\sum \nabla fm\theta + \lambda\theta^T\theta$$

2.

$$\nabla fi\theta = \nabla(x\theta - y)^T(x\theta - y) + \lambda\theta^T\theta$$
$$E[\nabla fi\theta] = \frac{1}{m}\sum \nabla fm\theta$$
$$Therefore,\ E[\nabla fi\theta] = \nabla J(\theta)$$

3.

Initialized all parameters θ

$\theta' = \theta - \eta\nabla J(\theta)$ where $J(\theta)$ is calculated with a random subset of training set.

4.

In [0]:
```
def stochastic_grad_descent(X, y, alpha=0.1, lambda_reg=1, num_epochs=1000, c = c):
    """
    In this question you will implement stochastic gradient descent with a regularization t
erm

    Args:
        X - the feature vector, 2D numpy array of size (num_instances, num_features)
        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 always a good idea to use a fixed step size. Usually
it's set to 1/sqrt(t) or 1/t
                if alpha is a float, then the step size in every iteration is alpha.
                if alpha == "1/sqrt(t)", alpha = 1/sqrt(t)
                if alpha == "1/t", alpha = 1/t
        lambda_reg - the regularization coefficient
        num_iter - number of epochs (i.e number of times) to go through the whole training
 set

    Returns:
        theta_hist - the history of parameter vector, 3D numpy array of size (num_iter, num
_instances, num_features)
        loss hist - the history of regularized loss function vector, 2D numpy array of size
(num_iter, num_instances)
    """
    num_instances, num_features = X.shape[0], X.shape[1]
    theta = np.ones(num_features) #Initialize theta
    num_iter = num_epochs
    theta_hist = np.zeros((num_iter+1, num_features))  #Initialize theta_hist
    loss_hist = np.zeros(num_iter+1) #Initialize loss_hist
    for i in range(1,num_iter+1):
        if alpha=='1/sqrt(t)':
            step_size = c/np.sqrt((i+1.0))
        elif alpha=='1/t':
            step_size = c/(i+1.0)
        else:

          step_size = alpha
        grad = compute_regularized_square_loss_gradient(X_train,y_train,theta,lambda_reg)
        theta = theta-step_size*grad.T
        loss_i = compute_square_loss(X,y,theta)+np.dot(theta,theta)*lambda_reg
        loss_hist[i] = loss_i
        theta_hist[i] = theta
    return loss_hist,theta_hist
```

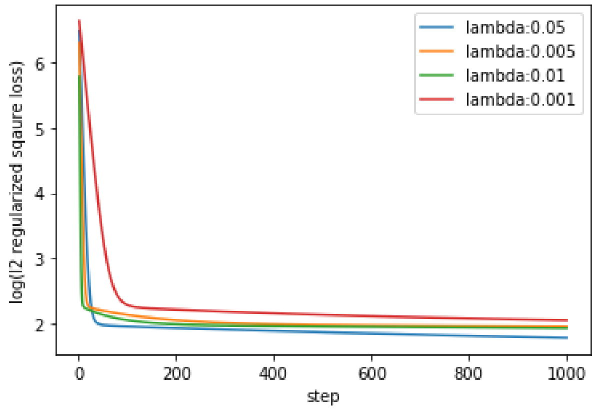
```
In [0]: h=stochastic_grad_descent(dmX, response, alpha=0.1, lambda_reg=1, num_epochs=1000)
```

```
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:34: RuntimeWarning: invalid v
alue encountered in subtract
```

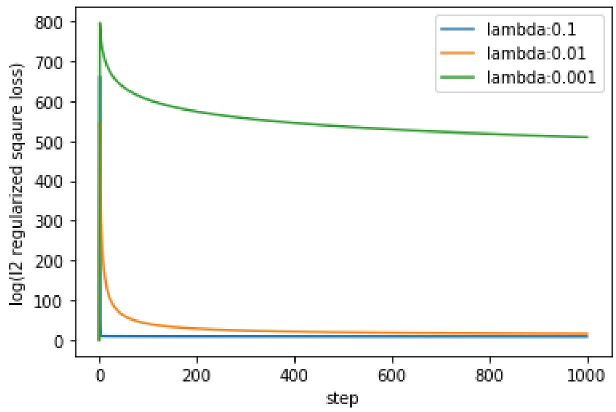
5.

```
In [0]: for step_size in [0.05,0.005, 0.01, 0.001]:
            loss_hist_SGD,theta_hist_SGD = stochastic_grad_descent(dmX,response,lambda_reg=1e-5,alp
        ha=step_size)
            plt.plot(range(len(loss_hist_SGD)),np.log(loss_hist_SGD),label='lambda:'+str(step_size
        ))
        plt.xlabel("step")
        plt.ylabel("log(l2 regularized sqaure loss)")
        plt.legend()
        plt.show()
```

```
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:3: RuntimeWarning: divide by
zero encountered in log
  This is separate from the ipykernel package so we can avoid doing imports until
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:3: RuntimeWarning: divide by
zero encountered in log
  This is separate from the ipykernel package so we can avoid doing imports until
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:3: RuntimeWarning: divide by
zero encountered in log
  This is separate from the ipykernel package so we can avoid doing imports until
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:3: RuntimeWarning: divide by
zero encountered in log
  This is separate from the ipykernel package so we can avoid doing imports until
```

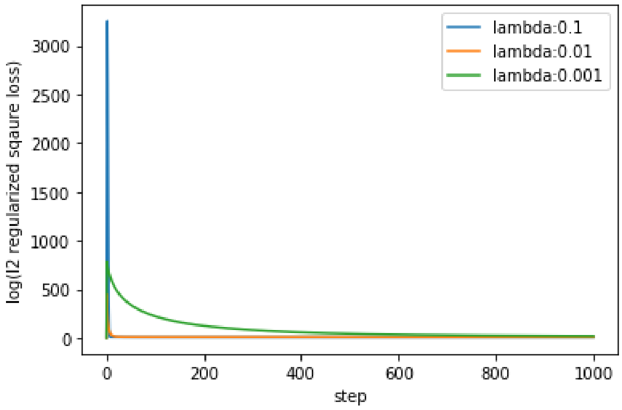



```
In [0]: for c in [0.1,0.01,0.001]:
            loss_hist_SGD,theta_hist_SGD = stochastic_grad_descent(dmX,response,lambda_reg=0.05,alp
        ha='1/t', c= c)
            #print(loss_hist_SGD)
            plt.plot(range(len(loss_hist_SGD)),(loss_hist_SGD),label='lambda:'+str(c))
        plt.xlabel("step")
        plt.ylabel("log(l2 regularized sqaure loss)")
        plt.legend()
        plt.show()
```



In [0]:

```
for c in [0.1,0.01,0.001]:
    loss_hist_SGD,theta_hist_SGD = stochastic_grad_descent(dmX,response,lambda_reg=0.05,alp
ha='1/sqrt(t)', c= c)
    #print(loss_hist_SGD)
    plt.plot(range(len(loss_hist_SGD)),(loss_hist_SGD),label='lambda:'+str(c))
plt.xlabel("step")
plt.ylabel("log(l2 regularized sqaure loss)")
plt.legend()
plt.show()
```



Gradient diverges with decreasing step-size. 0.01 ocnverges faster than 0.001!

Also, 1/sqrt(t) converges faster than 1/t

In [0]: