CS 189 Introduction to Machine Learning
Fall 2017 HW12

This homework is due **Tuesday, November 21 at 10pm.**

1 Getting Started

You may typeset your homework in latex or submit neatly handwritten and scanned solutions. Please make sure to start each question on a new page, as grading (with Gradescope) is much easier that way! Deliverables:

1. Submit a PDF of your writeup to assignment on Gradescope, "HW[n] Write-Up"
2. Submit all code needed to reproduce your results, "HW[n] Code".
3. Submit your test set evaluation results, "HW[n] Test Set".

After you've submitted your homework, be sure to watch out for the self-grade form.

(a) Before you start your homework, write down your team. Who else did you work with on this homework? List names and email addresses. In case of course events, just describe the group. How did you work on this homework? Any comments about the homework?

None
Comment: this monster hw takes way more than 30 hours

(b) Please copy the following statement and sign next to it:

I certify that all solutions are entirely in my words and that I have not looked at another student's solutions. I have credited all external sources in this write up.

I certify that all solutions are entirely in my words & that I have not looked at another student's solutions. I have credited all external sources in my writing

Problem # 2

a.)

$$J_\lambda(w) = \frac{1}{2}\|y - Aw\|_2^2 + \lambda \|w\|_1$$

$$= \frac{1}{2}(y-Aw)^T(y-Aw) + \lambda \sum_1^d |w_i|$$

$$= \underbrace{\frac{1}{2} y^T y}_{g(y)} - \sum_1^d y_i A_i w_i - \frac{1}{2} w^T A^T A w - \lambda |w_i|$$

$$w^T w = \sum w_i^2 \times n$$

$$\sum_1^d f(A_i, y, w_i, \lambda)$$

$$\frac{\partial J_\lambda(w)}{\partial w_i} = 0 \Rightarrow -y_i A_i + n w_i + \lambda \, \text{sign}(w_i) = 0$$

$$\Rightarrow w_i = \frac{y_i A_i - \lambda \, \text{sign}(w_i)}{n}$$

$\Rightarrow$ w_i determined by the i-feature & the output regardless of other features.

b.) $w_i^* > 0 \Rightarrow w_i^* = \dfrac{y_i A_i - \lambda}{n}$

c.) $w_i^* < 0 \Rightarrow w_i^* = \dfrac{y_i A_i + \lambda}{n}$

d.) $w_i^* = 0$ iff $y_i A_i = 0$ or $y_i A_i = \lambda$

e.)

$$J_\lambda(w) = \frac{1}{2}\|y - Aw\|_2^2 + \lambda \|w\|_2^2$$

$$\Rightarrow \frac{\partial J}{\partial w_i} = -y_i A_i + n w_i + 2\lambda w_i = 0$$

$$\Rightarrow w_i^* = \frac{y_i A_i}{n + 2\lambda}$$

λ cannot make w_i^* to be 0

$\Rightarrow$ L2 discourages w_i^* to be 0

$\rightarrow$ that's why L1 norm promotes sparsity

(or L2 norm discourage sparsity.

f.) $\lambda = 0.000001$ is good

See code attached

Problem #3

a.) See code attached

b.) See code & answer attached

c.) See code & answer attached

d.) $$Pr\{|Z_i| \geq t\} \leq e^{-t^2/2\sigma^2}$$

$$t = 2\sigma\sqrt{\log d}$$

$$\Rightarrow Pr\{|Z_i| \geq 2\sigma\sqrt{\log d}\} \leq \underbrace{e^{-\frac{4\sigma^2 \log d}{2\sigma^2}}}_{i := 1 \ldots d}$$

$$\Rightarrow \max_i P\{|Z_i| \geq 2\sigma\sqrt{\log d}\} \leq \frac{1}{d}$$

$$Pr\{\max_i |Z_i| \geq 2\sigma\sqrt{\log d}\} \leq \frac{1}{d}$$

e.) $\hat{w}_{stop}$ returns the top s entries of $\hat{w}_{ls}$

but $$E[\|\hat{w}_{ls} - w^*\|_2^2] = \sigma^2 \, trace[(A^TA)^{-1}]$$

$\Rightarrow$ $\hat{w}_{stop}$ return the top s entries of $w^* + z'$

Problem # 4

a.) See code attached

b.) See code attached Basically:

age : take average

sex : Female = 0, male = 1

Discard ticket & cabin

Fare : take avg of same pclass

embarked : C = 1 0 0

Q = 0 1 0

S = 0 0 1

c.) See code attached

d.) See code attached

e.) See code attached

f.) See code attached

g.) See code attached

h.) skip

i.) skip

j)

simple tree : titanic : Accuracy = 0.86

spam : 0.80

bagged : titanic : 0.86

spam : 0.84

random forest :

titanic : 0.669

spam : 0.709

k.) Submitted

Sparse imaging with LASSO

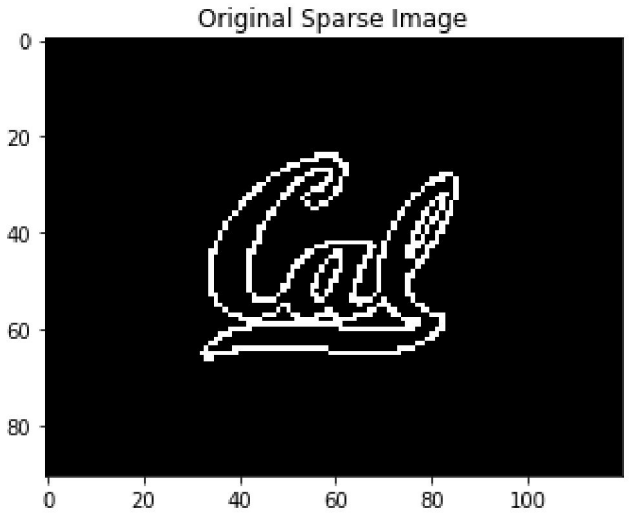
This example generates a sparse signal and tries to recover it using lasso

In [2]:
```
from __future__ import print_function
from __future__ import division
from sklearn import linear_model
import matplotlib.pyplot as plt
import numpy as np
from scipy import misc
from IPython import display
from simulator import *
%matplotlib inline
```

We generate an orthogonal matrix A and compute measurements = Aw+z where w is the vectorized format of the sparse image

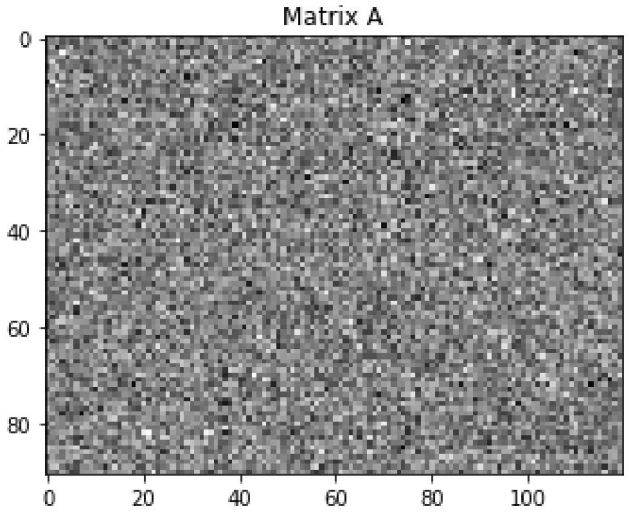
In [3]:
```
measurements,A,I = simulate()

# THE SETTINGS FOR THE IMAGE - PLEASE DO NOT CHANGE
height = 91
width = 120
sparsity = 476
numPixels = len(A[0])

plt.imshow(I, cmap=plt.cm.gray, interpolation='nearest');
plt.title('Original Sparse Image')
```

Out[3]: Text(0.5,1,'Original Sparse Image')



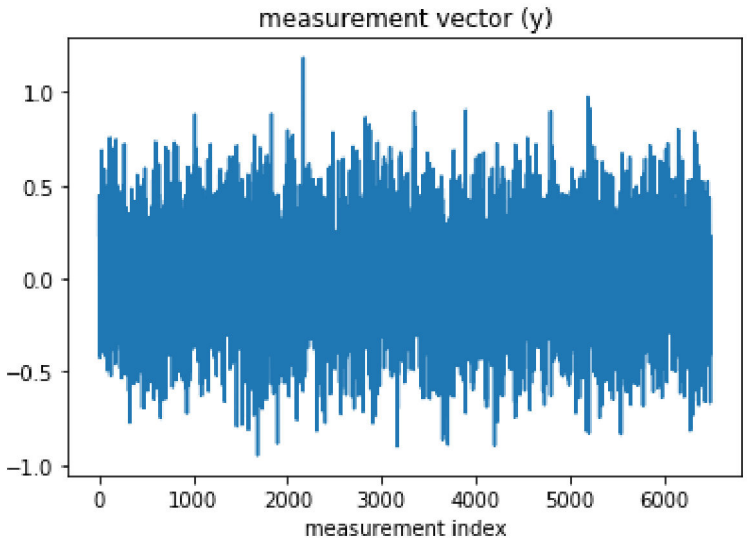
We plot matrix A:

In [4]:
```
chosenMaskToDisplay = 0
M0 = A[chosenMaskToDisplay].reshape((height,width))
plt.title('Matrix A')
plt.imshow(M0, cmap=plt.cm.gray, interpolation='nearest');
```



And here is the plot of measurement vector:

In [5]:
```
# measurements
plt.title('measurement vector (y)')
plt.plot(measurements)
plt.xlabel('measurement index')
plt.show()
```

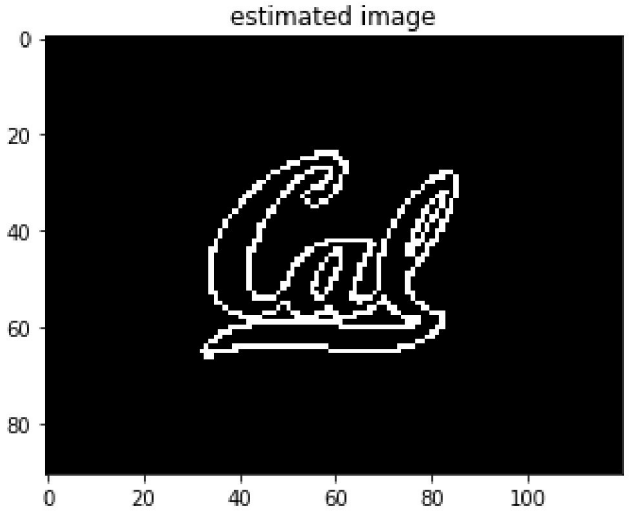


We use lasso to recover the image:

In [6]:
```
def LASSO(imDims, measurements, A, a):
    clf = linear_model.Lasso(alpha=a)
    clf.fit(A,measurements)
    Ihat = clf.coef_.reshape(imDims)
    plt.title('estimated image')
    plt.imshow(Ihat, cmap=plt.cm.gray, interpolation='nearest')
    return clf.coef_
```

Change the lasso regularization parameter to recover the image and report the value.

In [28]:
```
# change the lasso parameter here:
a = 0.0000001
recovered = LASSO((height,width),measurements,A,a)
```



Bias and Variance of Sparse Linear Regression

In this notebook, you will explore numerically how sparse vectors change the rate at which we can estimate the underlying model. This corresponds to parts (a), (b), (c) of Homework 12.

First, some setup. We will only be using basic libraries.

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

The following functions produce the ground truth matrix $A \in \mathbb{R}^{n \times d}$ (denoted by U since it is unitary), as well as the vector $w^* \in \mathbb{R}^d$ and observations $y \in \mathbb{R}^n$. They have been implemented for you, but it is worth going through the code to observe its limitations.

In [22]:
```
def ground_truth(n, d, s):
    """
    Input: Two positive integers n, d. Requires n >= d >=s. If d<s, we let s =
 d
    Output: A tuple containing i) random matrix of dimension n X d with orthon
ormal columns. and
            ii) a d-dimensional, s-sparse wstar with (large) Gaussian entries
    """
    if d > n:
        print("Too many dimensions")
        return None

    if d < s:
        s = d
    A = np.random.randn(n, d) #random Gaussian matrix
    U, S, V = np.linalg.svd(A, full_matrices=False) #reduced SVD of Gaussian m
atrix
    wstar = np.zeros(d)
    wstar[:s] = 10 * np.random.randn(s)

    np.random.shuffle(wstar)
    return U, wstar

def get_obs(U, wstar):
    """
    Input: U is an n X d matrix and wstar is a d X 1 vector.
    Output: Returns the n-dimensional noisy observation y = U * wstar + z.
    """
    n, d = np.shape(U)
    z = np.random.randn(n) #i.i.d. noise of variance 1
    y = np.dot(U, wstar) + z
    return y
```

We now implement the estimators that we will simulate. The least squares estimator has already been implemented for you. You will be implementing the top k and threshold estimators in part (b), but it is fine to skip this for now and compile.

In [21]:
```
def LS(U, y):
    """
    Input: U is an n X d matrix with orthonormal columns and y is an n X 1 v
ector.
    Output: The OLS estimate what_{LS}, a d X 1 vector.
    """
    wls = np.dot(U.T, y) #pseudoinverse of orthonormal matrix is its transpo
se
    return wls


def thresh(U, y, lmbda):
    """
    Input: U is an n X d matrix and y is an n X 1 vector; lambda is a scalar
 threshold of the entries.
    Output: The estimate what_{T}(lambda), a d X 1 vector that is hard-thres
holded (in absolute value) at level lambda.
            When code is unfilled, returns the all-zero d-vector.
    """
    n, d = np.shape(U)
    wls = LS(U, y)
    what = np.zeros(d)

    #print np.shape(wls)
    ##########
    #TODO: Fill in thresholding function; store result in what
    ######################
    #YOUR CODE HERE:
    for i, e in enumerate(wls):
        what[i] = wls[i] if abs(wls[i]) >= lmbda else 0

    ###############
    return what


def topk(U, y, s):
    """
    Input: U is an n X d matrix and y is an n X 1 vector; s is a positive in
teger.
    Output: The estimate what_{top}(s), a d X 1 vector that has at most s no
n-zero entries.
            When code is unfilled, returns the all-zero d-vector.
    """
    n, d = np.shape(U)
    what = np.zeros(d)
    wls = LS(U, y)

    ##########
    #TODO: Fill in thresholding function; store result in what
    ######################
    #YOUR CODE HERE: Remember the absolute value!
```

```
    indices = wls.argsort()[-s:][::-1]
    for i in indices:
        what[i] = wls[i]

    ###############
    return what
```

The following helper function that we have implemented for you returns the error of all three estimators as a function n, d, or s, depending on what you specify. Notice that it has the option to generate the true model with sparsity that need not equal the sparsity demanded by the estimators.

Again, this function can be run without implementing the thresh and topk functions, but some of its returned values should then be ignored.

In [23]:
```
def error_calc(num_iters=10, param='n', n=1000, d=100, s=5, s_model=True, tr
ue_s=5):
    """
    Plots the prediction error 1/n || U(what - wstar)||^2 = 1/n || what - ws
tar ||^2 for the three estimators
    averaged over num_iter experiments.

    Input:
    Output: 4 arrays -- range of parameters, errors of LS, topk, and thresh
 estimator, respectively. If thresh and topk
            functions have not been implemented yet, then these errors are s
imply the norm of wstar.
    """
    wls_error = []
    wtopk_error = []
    wthresh_error = []

    if param == 'n':
        arg_range = np.arange(100, 2000, 50)
        lmbda = 2 * np.sqrt(np.log(d))
        for n in arg_range:
            U, wstar = ground_truth(n, d, s) if s_model else ground_truth(n,
 d, true_s)
            error_wls = 0
            error_wtopk = 0
            error_wthresh = 0
            for count in range(num_iters):
                y = get_obs(U, wstar)
                wls = LS(U, y)
                wtopk = topk(U, y, s)
                wthresh = thresh(U, y, lmbda)
                error_wls += np.linalg.norm(wstar - wls)**2
                error_wtopk += np.linalg.norm(wstar - wtopk)**2
                error_wthresh += np.linalg.norm(wstar - wthresh)**2
            wls_error.append(float(error_wls)/ n / num_iters)
            wtopk_error.append(float(error_wtopk)/ n / num_iters)
            wthresh_error.append(float(error_wthresh)/ n / num_iters)

    elif param == 'd':
```

```
        arg_range = np.arange(10, 1000, 50)
        for d in arg_range:
            lmbda = 2 * np.sqrt(np.log(d))
            U, wstar = ground_truth(n, d, s) if s_model else ground_truth(n,
d, true_s)
            error_wls = 0
            error_wtopk = 0
            error_wthresh = 0
            for count in range(num_iters):
                y = get_obs(U, wstar)
                wls = LS(U, y)
                wtopk = topk(U, y, s)
                wthresh = thresh(U, y, lmbda)
                error_wls += np.linalg.norm(wstar - wls)**2
                error_wtopk += np.linalg.norm(wstar - wtopk)**2
                error_wthresh += np.linalg.norm(wstar - wthresh)**2
            wls_error.append(float(error_wls)/ n / num_iters)
            wtopk_error.append(float(error_wtopk)/ n / num_iters)
            wthresh_error.append(float(error_wthresh)/ n / num_iters)

    elif param == 's':
        arg_range = np.arange(5, 55, 5)
        lmbda = 2 * np.sqrt(np.log(d))
        for s in arg_range:
            U, wstar = ground_truth(n, d, s) if s_model else ground_truth(n,
d, true_s)
            error_wls = 0
            error_wtopk = 0
            error_wthresh = 0
            for count in range(num_iters):
                y = get_obs(U, wstar)
                wls = LS(U, y)
                wtopk = topk(U, y, s)
                wthresh = thresh(U, y, lmbda)
                error_wls += np.linalg.norm(wstar - wls)**2
                error_wtopk += np.linalg.norm(wstar - wtopk)**2
                error_wthresh += np.linalg.norm(wstar - wthresh)**2
            wls_error.append(float(error_wls)/ n / num_iters)
            wtopk_error.append(float(error_wtopk)/ n / num_iters)
            wthresh_error.append(float(error_wthresh)/ n / num_iters)

    return arg_range, wls_error, wtopk_error, wthresh_error
```

We are now ready to perform the parts of the question.

Part (a)

As an example, in the following cell, we run the helper function above to return error values of the OLS estimate for various values of n. You are required to:

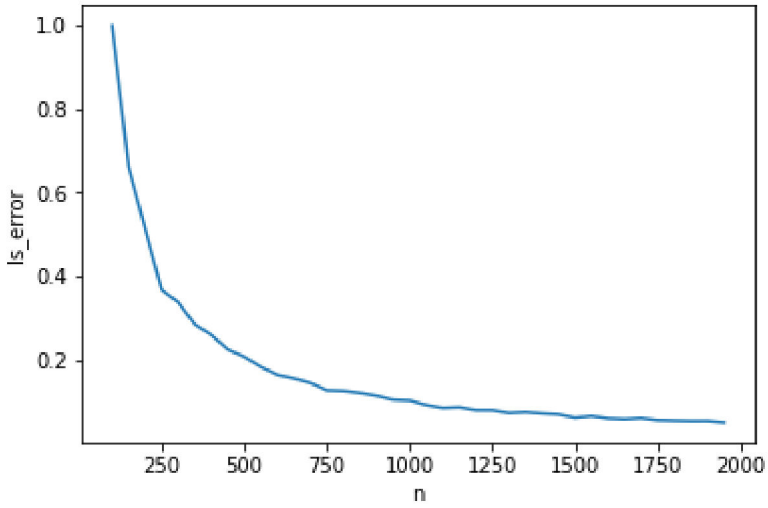
1) Plot the error as a function of n. You may find a log-log plot useful to see the expected bahavior.

2) Run the helper function to return the error as a function of d and s, and plot your results.

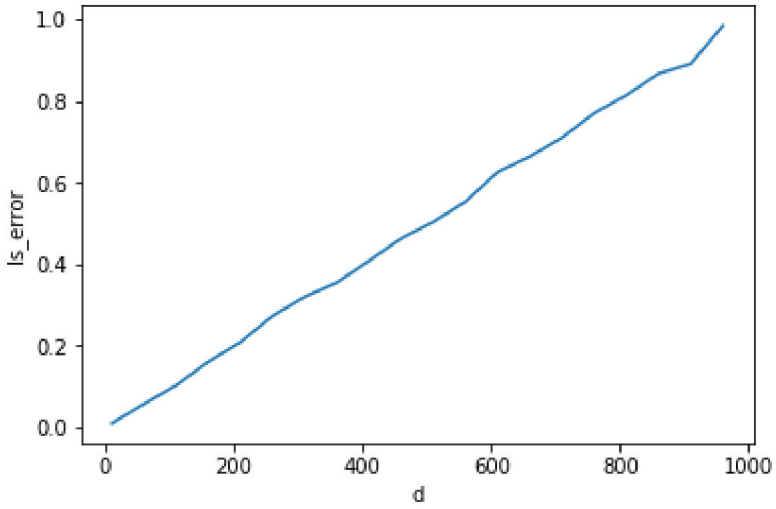
You need to have 3 plots in your answer. Make sure to label axes properly, and to make the plotting visible in general. Feel free to play with the parameters, but ensure that your answer describes your parameter choices. At this point, s_model is True, since we are only interested in the variance of the model.

In [24]:
```
#nrange contains the range of n used, ls_error the corresponding errors for th
e OLS estimate
nrange, ls_error, _, _ = error_calc(num_iters=10, param='n', n=1000, d=100, s=
5, s_model=True, true_s=5)

########
#TODO: Your code here: call the helper function for d and s, and plot everythi
ng
########
#YOUR CODE HERE:
print("ls_error vs n")
plt.plot(nrange, ls_error)
plt.xlabel("n")
plt.ylabel("ls_error")
plt.show()

plt.figure()
drange, ls_error, _, _ = error_calc(num_iters=10, param='d', n=1000, d=100, s=
5, s_model=True, true_s=5)
print("ls_error vs d")
plt.plot(drange, ls_error)
plt.xlabel("d")
plt.ylabel("ls_error")
plt.show()

plt.figure()
srange, ls_error, _, _ = error_calc(num_iters=10, param='s', n=1000, d=100, s=
5, s_model=True, true_s=5)
print("ls_error vs s")
plt.plot(srange, ls_error)
plt.xlabel("s")
plt.ylabel("ls_error")
plt.show()
```

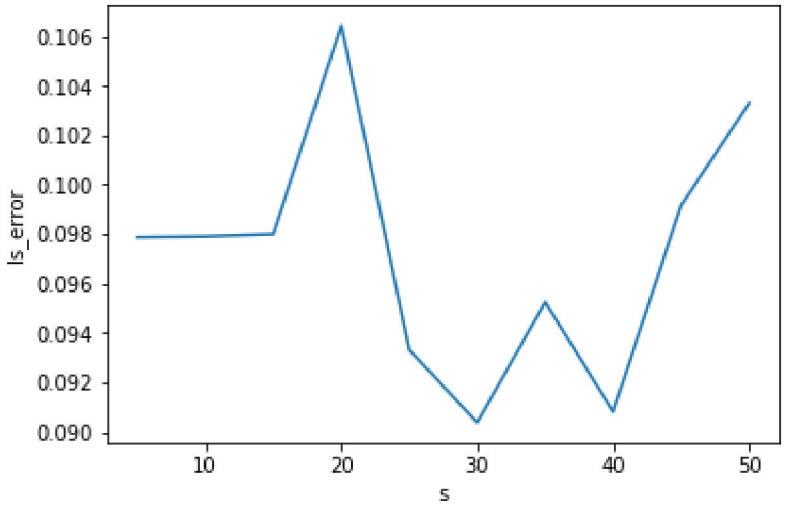
ls_error vs n



ls_error vs d



ls_error vs s



Are these plots as expected? Discuss. Also put down your parameter choices (either here or in plot captions.) It's fine to use the default values, but put them down nonetheless.

Your answer here

n = 2000

d = 0

s = 30

Part (b)

Now fill out the functions implementing the sparsity-seeking estimators: thresh, and topk in the above cells. You should be able to test these functions using some straightforward examples.

We will now simulate the error of all the estimators, as a function of n, d, and s. An example of this for n is given below. You must:

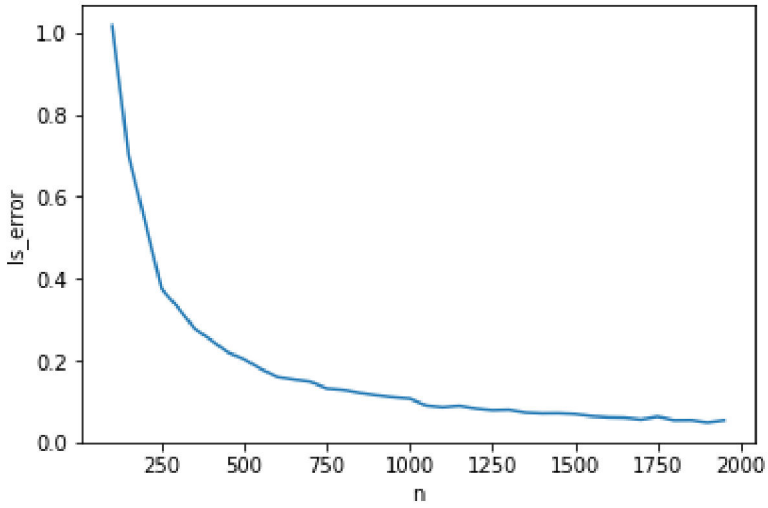
1) Plot the error of all estimators as a function of n.

2) Run the helper function to sweep over d and s, and plot the behavior of all three estimators.

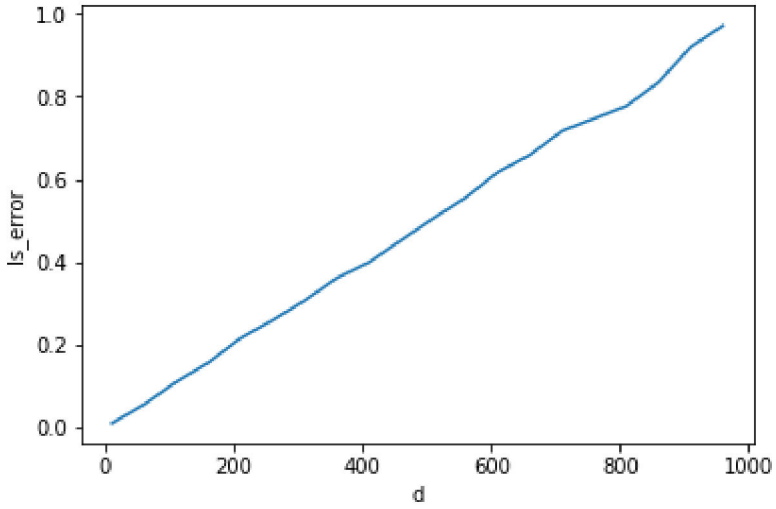
You should report 3 plots here once again. Make sure to make them fully readable.

In [26]:
```
#TODO: Part (b)
##############
#YOUR CODE HERE:

nrange, ls_error, _, _ = error_calc(num_iters=10, param='n', n=1000, d=100, s=
5, s_model=True, true_s=5)
print("ls_error vs n")
plt.plot(nrange, ls_error)
plt.xlabel("n")
plt.ylabel("ls_error")
plt.show()

plt.figure()
drange, ls_error, _, _ = error_calc(num_iters=10, param='d', n=1000, d=100, s=
5, s_model=True, true_s=5)
print("ls_error vs d")
plt.plot(drange, ls_error)
plt.xlabel("d")
plt.ylabel("ls_error")
plt.show()

plt.figure()
srange, ls_error, _, _ = error_calc(num_iters=10, param='s', n=1000, d=100, s=
5, s_model=True, true_s=5)
print("ls_error vs s")
plt.plot(srange, ls_error)
plt.xlabel("s")
plt.ylabel("ls_error")
plt.show()
```

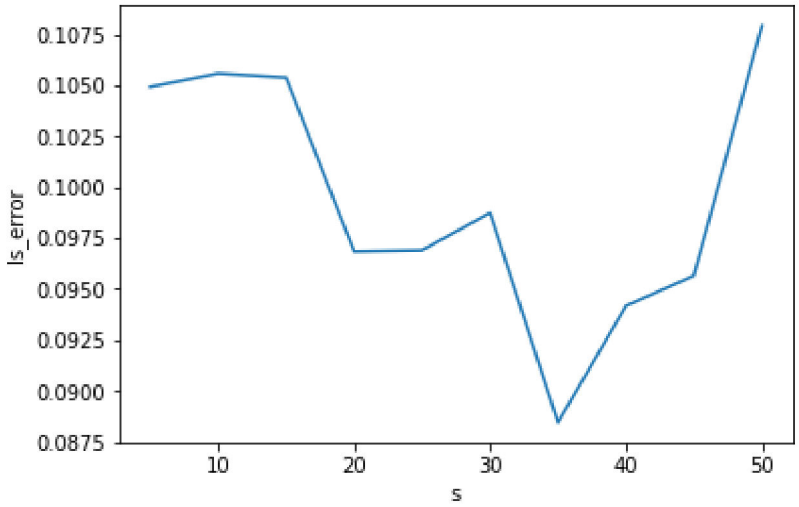
ls_error vs n



ls_error vs d



ls_error vs s



Part (c)

Now, call the helper function with the true sparsity being greater than the sparsity assumed by the top-k estimator. Remember to set s_model to False! Plot the behavior of all three estimators once again, as a function of n, d, s, where s is the assumed sparsity of the top-k model.

You should return 3 plots, and explain what you see in terms of the bias variance tradeoff.

In [30]:
```
#TODO: Part (c)
##############
#YOUR CODE HERE:

nrange, ls_error, _, _ = error_calc(num_iters=10, param='n', n=1000, d=100, s=
5, s_model=False, true_s=50)
print("ls_error vs n")
plt.plot(nrange, ls_error)
plt.xlabel("n")
plt.ylabel("ls_error")
plt.show()

plt.figure()
drange, ls_error, _, _ = error_calc(num_iters=10, param='d', n=1000, d=100, s=
5, s_model=False, true_s=50)
print("ls_error vs d")
plt.plot(drange, ls_error)
plt.xlabel("d")
plt.ylabel("ls_error")
plt.show()

plt.figure()
srange, ls_error, _, _ = error_calc(num_iters=10, param='s', n=1000, d=100, s=
5, s_model=False, true_s=50)
print("ls_error vs s")
plt.plot(srange, ls_error)
plt.xlabel("s")
plt.ylabel("ls_error")
plt.show()
```

In [1]:
```
import numpy as np
from numpy import genfromtxt
import scipy.io
from scipy import stats
from sklearn.tree import DecisionTreeClassifier
from sklearn.base import BaseEstimator, ClassifierMixin
import pandas as pd
from random import randint

eps = 1e-5  # a small number


class DecisionTree:

    def __init__(self, max_depth=3, feature_labels=None):
        self.max_depth = max_depth
        self.features = feature_labels
        self.left, self.right = None, None  # for non-leaf nodes
        self.split_idx, self.thresh = None, None  # for non-leaf nodes
        self.data, self.pred = None, None  # for leaf nodes

    @staticmethod
    def entropy(y):
        # TODO implement entropy function
        n = y.size
        hash_map = {}
        for i in y:
            hash_map[i] = 1 if i not in hash_map else hash_map[i] + 1
            # if i not in hash_map:
            #     hash_map[i] = 1
            # else:
            #     hash_map[i] += 1
        ret = 0
        for i in hash_map:
            ret -= hash_map[i] / n * np.log(hash_map[i] / n)
        return ret


    @staticmethod
    def information_gain(X, y, thresh):
        # TODO implement information gain function
        # print(X)
        index_0 = np.where(X < thresh)[0]
        index_1 = np.where(X >= thresh)[0]
        y0, y1 = y[index_0], y[index_1]
        n0, n1 = y0.size, y1.size
        return DecisionTree.entropy(y) - ( n0 / (n0 + n1) * DecisionTree.ent
ropy(y0) + n1 / (n0 + n1) * DecisionTree.entropy(y1) )

    def split(self, X, y, idx, thresh):
        X0, idx0, X1, idx1 = self.split_test(X, idx=idx, thresh=thresh)
        y0, y1 = y[idx0], y[idx1]
        return X0, y0, X1, y1

    def split_test(self, X, idx, thresh):
```

```
        idx0 = np.where(X[:,idx] < thresh)[0]
        idx1 = np.where(X[:,idx] >= thresh)[0]
        X0, X1 = X[idx0, :], X[idx1, :]
        return X0, idx0, X1, idx1

    def fit(self, X, y):
        if self.max_depth > 0:
            # compute entropy gain for all single-dimension splits,
            # thresholding with a linear interpolation of 10 values
            gains = []
            thresh = np.array([np.linspace(np.min(X[:, i]) + eps,
                                           np.max(X[:, i]) - eps, num=10) fo
r i
                               in range(X.shape[1])])
            for i in range(X.shape[1]):
                gains.append([self.information_gain(X[:, i], y, t) for t in
                              thresh[i, :]])

            gains = np.nan_to_num(np.array(gains))
            self.split_idx, thresh_idx = np.unravel_index(np.argmax(gains),
                                                          gains.shape)
            self.thresh = thresh[self.split_idx, thresh_idx]
            X0, y0, X1, y1 = self.split(X, y, idx=self.split_idx,
                                        thresh=self.thresh)
            if X0.size > 0 and X1.size > 0:
                self.left = DecisionTree(max_depth=self.max_depth-1,
                                         feature_labels=self.features)
                self.left.fit(X0, y0)
                self.right = DecisionTree(max_depth=self.max_depth-1,
                                          feature_labels=self.features)
                self.right.fit(X1, y1)
            else:
                self.max_depth = 0
                self.data, self.labels = X, y
                self.pred = stats.mode(y).mode[0]
        else:
            self.data, self.labels = X, y
            self.pred = stats.mode(y).mode[0]
        return self

    def predict(self, X):
        if self.max_depth == 0:
            return self.pred * np.ones(X.shape[0])
        else:
            X0, idx0, X1, idx1 = self.split_test(X, idx=self.split_idx,
                                                 thresh=self.thresh)
            yhat = np.zeros(X.shape[0], dtype="int")
            yhat[idx0] = self.left.predict(X0)
            yhat[idx1] = self.right.predict(X1)
            return yhat


class BaggedTrees(BaseEstimator, ClassifierMixin):

    def __init__(self, params=None, n=200):
        if params is None:
            params = {}
```

```
        self.params = params
        self.n = n
        self.decision_trees = [
            DecisionTreeClassifier(random_state=i, **self.params) for i in
            range(self.n)]

    def fit(self, X, y):
        row_num, _ = X.shape
        for i in range(self.n):
            X_sampling = []
            y_sampling = []
            for _ in range(row_num):
                index = randint(0, row_num - 1)
                X_sampling.append(X[index])
                y_sampling.append(y[index])
            X_sampling = np.array(X_sampling)
            y_sampling = np.array(y_sampling)
            self.decision_trees[i].fit(X_sampling, y_sampling)


    def predict(self, X):
        ret = []
        for i in range(self.n):
            ret.append(self.decision_trees[i].predict(X))
        ret = np.array(ret)
        ret = np.mean(ret, axis=0)
        array_round = np.vectorize(lambda x: int(round(x)))
        return array_round(ret)


class RandomForest(BaggedTrees):

    def __init__(self, params=None, n=200, m=1):
        if params is None:
            params = {}
        # TODO implement function
        self.params = params
        self.n = n
        self.m = m
        self.decision_trees = [
            DecisionTreeClassifier(random_state=i, **self.params) for i in
            range(self.n)]
        self.features_list = []


    def fit(self, X, y):
        row_num, feature_num = X.shape
        for i in range(self.n):
            self.features_list.append([randint(0, feature_num - 1) for _ in
range(self.m)])
            X_sampling = []
            y_sampling = []
            for _ in range(row_num):
                index = randint(0, row_num - 1)
                X_sampling.append(X[index, self.features_list[i]])
                y_sampling.append(y[index])
            X_sampling = np.array(X_sampling)
```

```
            y_sampling = np.array(y_sampling)
            self.decision_trees[i].fit(X_sampling, y_sampling)


    def predict(self, X):
        ret = []
        for i in range(self.n):
            ret.append(self.decision_trees[i].predict(X[:, self.features_lis
t[i]]))
        ret = np.array(ret)
        ret = np.mean(ret, axis=0)
        array_round = np.vectorize(lambda x: int(round(x)))
        return array_round(ret)


class BoostedRandomForest(RandomForest):

    def fit(self, X, y):
        self.w = np.ones(X.shape[0]) / X.shape[0]  # Weights on data
        self.a = np.zeros(self.n)  # Weights on decision trees
        # TODO implement function
        return self

    def predict(self, X):
        # TODO implement function
        pass


def preprocess_titanic_data(data_path):
    df = pd.read_csv(data_path)
    df.drop(["ticket", "cabin"], axis=1, inplace=True)

    row_indices = []
    for i, row in df.iterrows():
        if pd.isnull(row).all():
            row_indices.append(i)

    df.drop(df.index[row_indices], inplace=True)

    for i, row in df.iterrows():
        df.at[i, "sex"] = 0 if row["sex"] == "female" else 1

    df.at[df.age.isnull(), "age"] = df["age"].mean()
    df.at[df.fare.isnull(), "fare"] = df[df.pclass == 1]["fare"].mean()

    df["e1"], df["e2"], df["e3"] = [0, 0, 0]
    for i, row in df.iterrows():
        if row["embarked"] == 'C':
            df.at[i, "e1"] = 1
        elif row["embarked"] == 'Q':
            df.at[i, "e2"] = 1
        else:
            df.at[i, "e3"] = 1

    df.drop("embarked", axis=1, inplace=True)

    data_path_list = data_path.split('/')
```

```
    data_path_list[-1] = "preprocessed_" + data_path_list[-1]
    preprocessed_data_path = '/'.join(data_path_list)
    df.to_csv(preprocessed_data_path, index=False)
    return preprocessed_data_path


if __name__ == "__main__":
    dataset = "spam"
    params = {
        "max_depth": 20,
        # "random_state": 6,
        "min_samples_leaf": 10,
    }

    if dataset == "titanic":
        # Load titanic data
        path_train = 'datasets/titanic/titanic_training.csv'
        data = genfromtxt(path_train, delimiter=',', dtype=None)
        features = data[0, 1:]  # features = all columns except survived
        y = data[1:, 0]  # label = survived
        class_names = ["Died", "Survived"]

        # TODO implement preprocessing of Titanic dataset
        preprocessed_training_path = preprocess_titanic_data(path_train)
        path_test = 'datasets/titanic/titanic_testing_data.csv'
        preprocessed_test_path = preprocess_titanic_data(path_test)

        training_data = genfromtxt(preprocessed_training_path, delimiter=','
)
        test_data = genfromtxt(preprocessed_test_path, delimiter=',')
        X, Z = training_data[1:, 1:], test_data[1:, :]
        y = training_data[1:, 0]
        y.astype(int)
    elif dataset == "spam":
        features = ["pain", "private", "bank", "money", "drug", "spam",
                    "prescription", "creative", "height", "featured", "diffe
r",
                    "width", "other", "energy", "business", "message",
                    "volumes", "revision", "path", "meter", "memo", "plannin
g",
                    "pleased", "record", "out", "semicolon", "dollar", "shar
p",
                    "exclamation", "parenthesis", "square_bracket", "ampersa
nd"]
        assert len(features) == 32

        # Load spam data
        path_train = 'datasets/spam_data/spam_data.mat'
        data = scipy.io.loadmat(path_train)
        X = data['training_data']
        y = np.squeeze(data['training_labels'])
        Z = data['test_data']
        class_names = ["Ham", "Spam"]
    else:
        raise NotImplementedError("Dataset %s not handled" % dataset)

    print("Features", features)
```

```
print("Train/test size", X.shape, Z.shape)

print("\nPart 0: constant classifier")
print("Accuracy", 1 - np.sum(y) / y.size)

# Basic decision tree
print("\nPart (a-b): simplified decision tree")
dt = DecisionTree(max_depth=3, feature_labels=features)
dt.fit(X, y)
print("Predictions", dt.predict(Z)[:100])
print()

# TODO implement and evaluate parts c-h

print("========== Question 4.c ==========")

print()
print("Titanic ====>")
dataset == "titanic"

path_train = 'datasets/titanic/titanic_training.csv'
data = genfromtxt(path_train, delimiter=',', dtype=None)
features = data[0, 1:]  # features = all columns except survived
y = data[1:, 0]  # label = survived
class_names = ["Died", "Survived"]

preprocessed_training_path = preprocess_titanic_data(path_train)
path_test = 'datasets/titanic/titanic_testing_data.csv'
preprocessed_test_path = preprocess_titanic_data(path_test)

training_data = genfromtxt(preprocessed_training_path, delimiter=',')
test_data = genfromtxt(preprocessed_test_path, delimiter=',')
X, Z = training_data[1:, 1:], test_data[1:, :]
y = training_data[1:, 0]
y.astype(int)

print("Features", features)
print("Train/test size", X.shape, Z.shape)
print()
print("Part (c): simplified decision tree - titanic")
dt = DecisionTree(max_depth=10, feature_labels=features)
dt.fit(X, y)
y_predicted = dt.predict(X)
count = 0
for i, e in enumerate(y):
    count += 1 if abs(y[i] - y_predicted[i]) < 0.01 else 0
print("Accuracy", count/y.size)
y_predicted = dt.predict(Z)
with open("submission_titanic_simpified.txt", "w") as f:
    for i in y_predicted:
        f.write(str(i) + "\n")

print()
print("Part (e): bagged - titanic")
dt = BaggedTrees(params)
dt.fit(X, y)
y_predicted = dt.predict(X)
```

```
    count = 0
    for i, e in enumerate(y):
        count += 1 if abs(y[i] - y_predicted[i]) < 0.01 else 0
    print("Accuracy", count/y.size)
    y_predicted = dt.predict(Z)
    with open("submission_titanic_bagged.txt", "w") as f:
        for i in y_predicted:
            f.write(str(i) + "\n")

    print()
    print("Part (g): random forest - titanic")
    dt = RandomForest(params)
    dt.fit(X, y)
    y_predicted = dt.predict(X)
    count = 0
    for i, e in enumerate(y):
        count += 1 if abs(y[i] - y_predicted[i]) < 0.01 else 0
    print("Accuracy", count/y.size)
    y_predicted = dt.predict(Z)
    with open("submission_titanic_randomforest.txt", "w") as f:
        for i in y_predicted:
            f.write(str(i) + "\n")

    print()
    print("Spam ====>")
    dataset == "spam"

    features = ["pain", "private", "bank", "money", "drug", "spam",
                    "prescription", "creative", "height", "featured", "diffe
r",
                    "width", "other", "energy", "business", "message",
                    "volumes", "revision", "path", "meter", "memo", "plannin
g",
                    "pleased", "record", "out", "semicolon", "dollar", "shar
p",
                    "exclamation", "parenthesis", "square_bracket", "ampersa
nd"]
    assert len(features) == 32

    # Load spam data
    path_train = 'datasets/spam_data/spam_data.mat'
    data = scipy.io.loadmat(path_train)
    X = data['training_data']
    y = np.squeeze(data['training_labels'])
    y.astype(int)
    Z = data['test_data']
    class_names = ["Ham", "Spam"]

    print("Features", features)
    print("Train/test size", X.shape, Z.shape)
    print()
    print("Part (c): simplified decision tree - spam")
    dt = DecisionTree(max_depth=5, feature_labels=features)
    dt.fit(X, y)
    y_predicted = dt.predict(X)
    count = 0
    for i, e in enumerate(y):
```

```
    count += 1 if abs(y[i] - y_predicted[i]) < 0.01 else 0
print("Accuracy", count/y.size)
y_predicted = dt.predict(Z)
with open("submission_spam_simpified.txt", "w") as f:
    for i in y_predicted:
        f.write(str(i) + "\n")

print()
print("Part (e): bagged - spam")
dt = BaggedTrees(params)
dt.fit(X, y)
y_predicted = dt.predict(X)
count = 0
for i, e in enumerate(y):
    count += 1 if abs(y[i] - y_predicted[i]) < 0.01 else 0
print("Accuracy", count/y.size)
y_predicted = dt.predict(Z)
with open("submission_spam_bagged.txt", "w") as f:
    for i in y_predicted:
        f.write(str(i) + "\n")

print()
print("Part (g): random forest - spam")
dt = RandomForest(params)
dt.fit(X, y)
y_predicted = dt.predict(X)
count = 0
for i, e in enumerate(y):
    count += 1 if abs(y[i] - y_predicted[i]) < 0.01 else 0
print("Accuracy", count/y.size)
y_predicted = dt.predict(Z)
with open("submission_spam_randomforest.txt", "w") as f:
    for i in y_predicted:
        f.write(str(i) + "\n")
```

```
Features ['pain', 'private', 'bank', 'money', 'drug', 'spam', 'prescription',
'creative', 'height', 'featured', 'differ', 'width', 'other', 'energy', 'busi
ness', 'message', 'volumes', 'revision', 'path', 'meter', 'memo', 'planning',
'pleased', 'record', 'out', 'semicolon', 'dollar', 'sharp', 'exclamation', 'p
arenthesis', 'square_bracket', 'ampersand']
Train/test size (5172, 32) (5857, 32)

Part 0: constant classifier
Accuracy 0.709976798144

Part (a-b): simplified decision tree
Predictions [0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 1 0 0 0 0 1
1 0 0 1 0
 0 0 0 0 1 0 0 1 0 0 1 0 0 0 1 0 0 0 0 0 0 1 0 1 0 0 1 0 1 0 0 1 0 0 1 0 0
 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 1 0 0 1 0 0 1 1 0 1]

========== Question 4.c ==========

Titanic ====>
Features [b'pclass' b'sex' b'age' b'sibsp' b'parch' b'ticket' b'fare' b'cabi
n'
 b'embarked']
Train/test size (999, 9) (310, 9)

Part (c): simplified decision tree - titanic
Accuracy 0.8698698698698699

Part (e): bagged - titanic
Accuracy 0.8638638638638638

Part (g): random forest - titanic
Accuracy 0.6456456456456456

Spam ====>
Features ['pain', 'private', 'bank', 'money', 'drug', 'spam', 'prescription',
'creative', 'height', 'featured', 'differ', 'width', 'other', 'energy', 'busi
ness', 'message', 'volumes', 'revision', 'path', 'meter', 'memo', 'planning',
'pleased', 'record', 'out', 'semicolon', 'dollar', 'sharp', 'exclamation', 'p
arenthesis', 'square_bracket', 'ampersand']
Train/test size (5172, 32) (5857, 32)

Part (c): simplified decision tree - spam
Accuracy 0.8002706883217324

Part (e): bagged - spam
Accuracy 0.8451276102088167

Part (g): random forest - spam
Accuracy 0.7099767981438515
```