Q1

a) Myself

b) 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.

Q5

What is the main downfall of decision trees and how do random forests solve that?

OVERFITTING.

that can be seen in the training accuracy results of a single decision tree vs its validation accuracy.

Q2

$$\hat{w} = \underset{w}{\arg\min} \left\{ \frac{1}{2} \|y - Xw\|_2^2 + \lambda \|w\|_1 \right\}$$

$$J_\lambda(w) = \frac{1}{2}\|y\|^2 + \frac{1}{2}\|Xw\|^2 - y^T X w + \lambda \|w\|_1$$

$$= \frac{1}{2}\|y\|^2 + \frac{1}{2} w^T X^T X w - y^T X w + \lambda \|w\|_1$$

$$= \frac{1}{2}\|y\|^2 + \frac{1}{2} w^T \Big(\sum_{ij} x_i^T x_j\Big) w - y^T X w + \lambda \|w\|_1$$

$$= \frac{1}{2}\|y\|^2 + \frac{n}{2} w^T w - y^T X w + \lambda \|w\|$$

~~Uncorrelated x_i's~~

$$\leadsto \cancel{E[x_i x_j^T] - E[x_i]E[x_j] = \text{diag for } X = \begin{bmatrix} x_1^T \\ x_2^T \\ \vdots \\ x_n^T \end{bmatrix}}$$

$$\cancel{J_\lambda(w) = \frac{1}{2}\|y - \sum_i w_i x_i\|^2 + \lambda \|w\|_1}$$

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

Solve All together

$$\frac{\partial}{\partial w}(J_\lambda(w)) = n w + \frac{\partial}{\partial w}\|w\|_1 - (X^T y) = 0$$

$$= n w + \text{sig}(w) - X^T y = 0$$

$$w = \frac{1}{n}\left[X^T y - \text{sign}(w)(\lambda)\right]$$

Solve for w_i Seperately

$$\frac{\partial}{\partial w_i} J_{i\lambda}(w_i) = \frac{\partial}{\partial w_i}\left[\frac{n}{2} w_i^2 - (y^T x_i) w_i + \lambda |w_i|\right]$$

$$= n w_i - y^T x_i + \lambda\, \text{sgn}(w_i) = 0$$

$$w_i = \frac{1}{n}\left[y^T x_i - \lambda\, \text{sgn}(w_i)\right]$$

positive

b) $\hat{w}_i = \frac{1}{n}[+y_i^T X_i - \lambda] = +\frac{y_i^T X_i}{n} - \frac{\lambda}{n}$

c) $\hat{w}_i = -\frac{1}{n}[y_i^T X_i + \lambda] = +\frac{y_i^T X_i}{n} + \frac{\lambda}{n}$

- negative

d) if $|y_i^T x_i| < |\lambda|$ then $\hat{w}_i = 0$

e) $\hat{w} = (X^T X + \lambda I)^{-1} X^T y$

$\hat{w}_i = \frac{1}{n+\lambda}(X_i^T y_i)$ $\quad x_i^T y_i = 0$ for $\hat{w}_i = 0$

f) 1e-5 $= \lambda$

Q3

d) $z_i \sim N(0, \sigma^2)$ $\quad Pr(|z| \geq t) \leq e^{-t^2/2\sigma^2}$

$Pr(|z_i| \geq 2\sigma\sqrt{\log d}) \leq e^{-(2\sigma\sqrt{\log d})^2/2\sigma^2} = e^{-\frac{4\sigma^2 \log d}{2\sigma^2}} = 1/d^2$

$d^*(Pr(|z_i| \geq 2\sigma\sqrt{\log d})) \leq (\frac{1}{d^2})\cdot d' \leq 1/d \Rightarrow P(\max|z_i| \geq 2\sigma\sqrt{\log d}) \leq 1/d$

all of them $< 2\sigma\sqrt{\log d}$

e) $w = (X^TX)^{-1}X^T(y) = (X^TX)^{-1}X^T(y^* + z) = w^* + (X^TX)^{-1}Xz$

$= w^* + z'$

$z' \sim (0, \sigma^2 I)$ transformed cov matrix

because $\sigma^2 \cancel{(X^TX)^{-1}X^T}X(X^TX)^{-1}I$

I because has orthonormal columns

$\hat{w} = w^* + z'$

$\hat{w}_{top}(s) = T_s(\hat{w}) = T_s(w^* + z')$

f) $e = \hat{w}_{top}(s) - w^*$ (assuming w^* is s-sparse)

Max error occurs when $\hat{w}_{top}(s)$ predicts s-originally 0 entries in w^* to have some value.

$\max(\hat{w}_{top}(s) - w^*)$ will give s-wrongly predicted non-zero entries

2- negative s entries of original weights

$\max(\text{Sparsity}(e)) = 2s$

g) Given $\max|z_i'| \leq 2\sigma\sqrt{\log d}$

$w_{top}(s) = T_s(w^* + z_i)$

Consider max entry

$e = T_s(w^* + z_i) - w^*$

$e_i = w_i^* + z_i' - w_i^*$

$= z_i$ or $-w_i^*$

$T_s(w_i^* + z_i)$

$\begin{cases} |w_i^* + z_i| & \text{if } i \text{ is top } s \\ 0 & \text{otherwise} \end{cases}$

$$e_i = \begin{cases} -z_i & \max|e_i| \leq 2\sigma\sqrt{\log d} \\ -w_i^* & \leftarrow \text{Assume} \to 0 \text{ because we pr} \end{cases}$$

(h) $Pr(\|\hat{w}_{top(s)} - w^*\|^2 \leq 32\sigma^2 s \log d)$

$\uparrow$ have $\sum_i^n |e_i|^2 \leq (4\sigma\sqrt{\log d})^2 \times 2s$ because $|e_i| \neq 0$ only $2s$ times max

$\leq (16\sigma^2 \log d)\, 2s$

$\leq 32\sigma^2 \log d$

Given that $\mathcal{E}$

from (d) $\Rightarrow Pr(\text{not } \mathcal{E}) \leq 1/d$

$\Rightarrow Pr(\mathcal{E}) \geq 1 - 1/d$

(i) $\|Xv\|^2 = \|v\|^2$ since X has orthonormal columns

$|\lambda_i| = 1 \ \forall \lambda_i$ in $Spec(X)$

$\frac{1}{n}\|X(\hat{w}_{top(s)} - w)\| \leq \frac{1}{n} 32\sigma^2 s \log d$ w/ Prob $(1 - 1/d)$

(j) When Sparsity comes into play

$\left(\frac{1}{n}\|Xe_{tops}\|^2\right) \leq \frac{32\sigma^2 s \log d}{n}$ compared $\frac{\|Xe_{LS}\|}{n} \leq \sigma^2 \frac{d}{n}$

w/ Pr $(1 - 1/d)$

$\Rightarrow$ high probability

$\Rightarrow \frac{1}{n}\|Xe_{tops}\|^2 \leq \frac{1}{n}\|Xe_{WLS}\|^2$

$\frac{32\cancel{\sigma^2} s \log d}{\cancel{n}} \leq \cancel{\sigma^2}\frac{d}{\cancel{n}}$

when

$$\boxed{s \leq \frac{d}{32 \log d}}$$

(4k) The variance would only be σ^2 because that's the variance of the noise

there is a factor of $32 s \log d$ that we pay for not knowing which features to use.

Q4

b) Missing missing features:
the values are filled with the mode of the data feature
Not Numerical Values:
If the string is common enough, we make a one-hot-feature column for it.
Noisy Class label:
Remove it.

e) ~~exclamation~~ < 0.5 for all

g) Spam:

$money < 0.5$
$private < 0.5$
$bank < 0.5$

some common

Most common = 'money'

Titanic:

$sex < 0.5$
class
$Sibsp < 0.5$

Most common = Sex

j) When x has very little counts of anything, means we don't have much information about x then
$W > 1e-15$

When label is not spam it is easy to classify because they mostly only have characters and no keywords of the ones above. then
$W < 1e-35$

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 [5]:
```
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 [6]:
```
def ground_truth(n, d, s):
    """
    Input: Two positive integers n, d. Requires n >= d >=s. If d<s, w
e let s = d
    Output: A tuple containing i) random matrix of dimension n X d wi
th orthonormal columns. and
             ii) a d-dimensional, s-sparse wstar with (large) Gaussia
n 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 G
aussian matrix
    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.

```
    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_t
ruth(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:

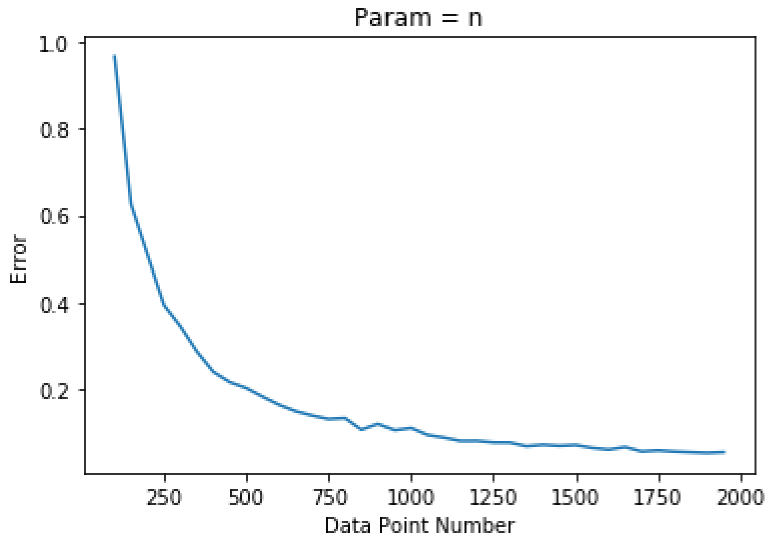
1) Plot the error as a function of n. You may find a log-log plot useful to see the expected bahavior.

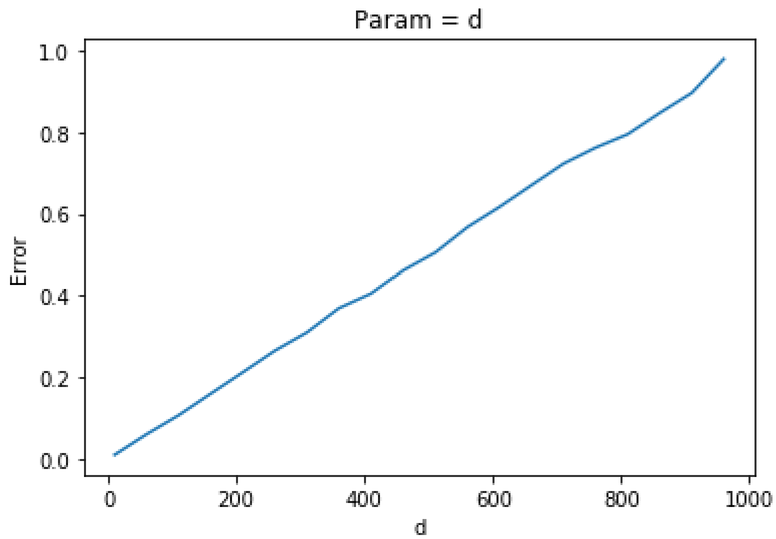
2) Run the helper function to return the error as a function of d and s, and plot your results.

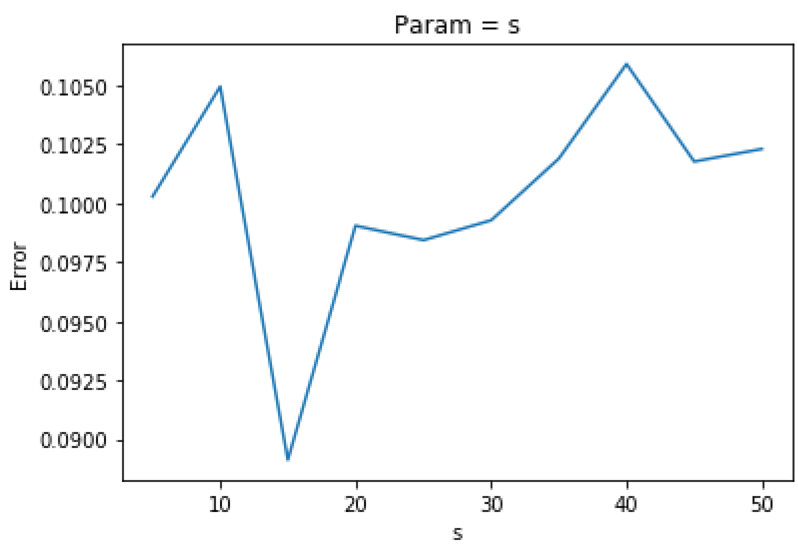
You need to have 3 plots in your answer. Make sure to label axes properly, and to make the plotting visible in general. Feel free to play with the parameters, but ensure that your answer describes your parameter choices. At this point, s_model is True, since we are only interested in the variance of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

In [14]:
```
#nrange contains the range of n used, ls_error the corresponding erro
rs for the OLS estimate
nrange, ls_error, _, _ = error_calc(num_iters=10, param='n', n=1000,
d=100, s=5, s_model=True, true_s=5)

plt.figure()
plt.plot(nrange, ls_error)
plt.xlabel('Data Point Number')
plt.title('Param = n')
plt.ylabel('Error')


nrange, ls_error, _, _ = error_calc(num_iters=10, param='d', n=1000,
d=100, s=5, s_model=True, true_s=5)

plt.figure()
plt.plot(nrange, ls_error)
plt.xlabel('d')
plt.title('Param = d')
plt.ylabel('Error')


nrange, ls_error, _, _ = error_calc(num_iters=10, param='s', n=1000,
d=100, s=5, s_model=True, true_s=5)

plt.figure()
plt.plot(nrange, ls_error)
plt.xlabel('s')
plt.title('Param = s')
plt.ylabel('Error')
```

```
Out[14]: Text(0,0.5,'Error')
```







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.

They are as expected. For LS, the higher the dimensions, the more the bias error. Moreover, the higher the complexity of our Model. S, the sparsity is unrelated to our OLS solver, so as expected, the error is uncorrelated to the sparsity s. Also, as n increases, as expected, the error decreases, as the effect of noise has less effect on our most likely weights. Optimal : d = 1 , n = infinity , s = anything

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.

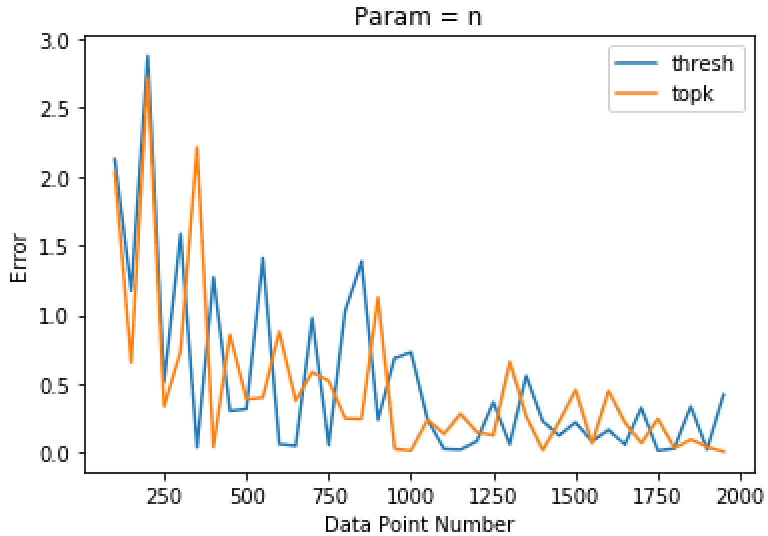
2) Run the helper function to sweep over d and s, and plot the behavior of all three estimators.

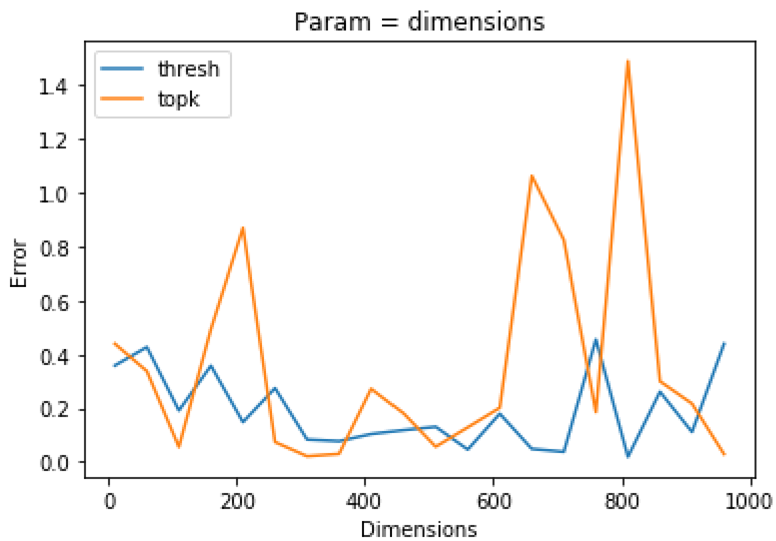
You should report 3 plots here once again. Make sure to make them fully readable.

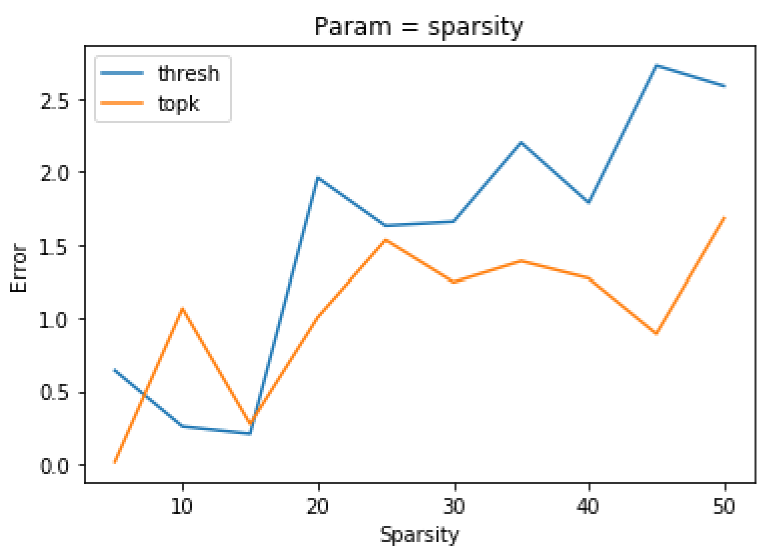
In [34]:
```
nrange, _,wtopk_error, wthresh_error= error_calc(num_iters=10, param=
'n', n=1000, d=100, s=5, s_model=True, true_s=5)

plt.figure()
plt.plot(nrange, wtopk_error, nrange, wthresh_error)
plt.xlabel('Data Point Number')
plt.title('Param = n')
plt.ylabel('Error')
plt.legend({'topk', 'thresh'})

nrange, _,wtopk_error, wthresh_error= error_calc(num_iters=10, param=
'd', n=1000, d=100, s=5, s_model=True, true_s=5)

plt.figure()
plt.plot(nrange, wtopk_error, nrange, wthresh_error)
plt.xlabel('Dimensions')
plt.title('Param = dimensions')
plt.ylabel('Error')
plt.legend()
plt.legend({'topk', 'thresh'})

nrange, _,wtopk_error, wthresh_error= error_calc(num_iters=10, param=
's', n=1000, d=100, s=5, s_model=True, true_s=5)

plt.figure()
plt.plot(nrange, wtopk_error, nrange, wthresh_error)
plt.xlabel('Sparsity')
plt.title('Param = sparsity')
plt.ylabel('Error')
plt.legend()
plt.legend({'topk', 'thresh'})
```

```
No handles with labels found to put in legend.
No handles with labels found to put in legend.
```

```
Out[34]: <matplotlib.legend.Legend at 0x7f9d9652d748>
```







Part (c)

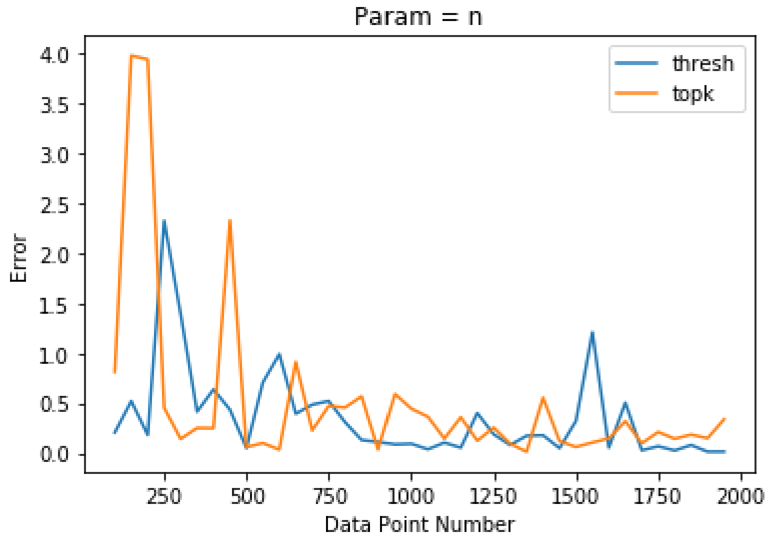
Now, call the helper function with the true sparsity being greater than the sparsity assumed by the top-k estimator. Remember to set s_model to False! Plot the behavior of all three estimators once again, as a function of n, d, s, where s is the assumed sparsity of the top-k model.

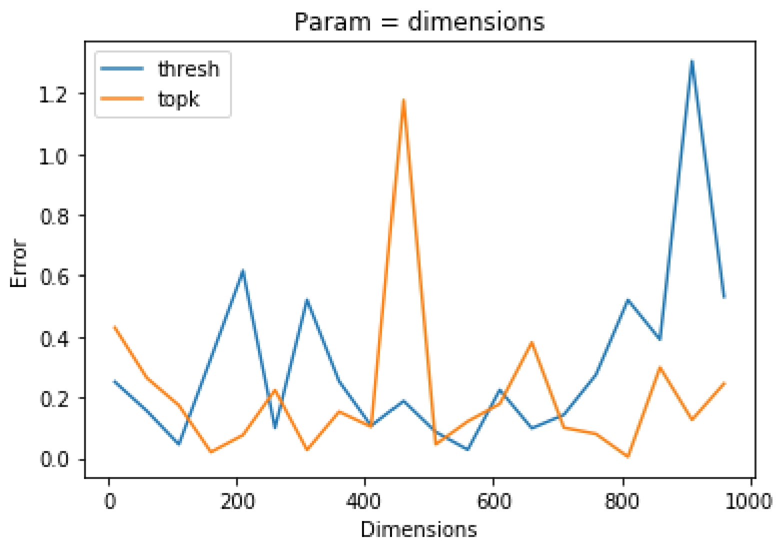
You should return 3 plots, and explain what you see in terms of the bias variance tradeoff.

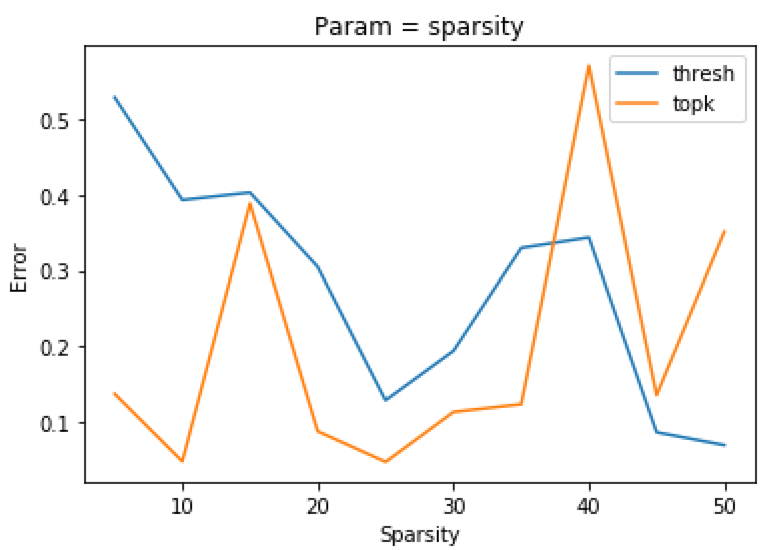
In [38]:
```
nrange, _,wtopk_error, wthresh_error= error_calc(num_iters=10, param=
'n', n=1000, d=100, s=5, s_model=False, true_s=5)

plt.figure()
plt.plot(nrange, wtopk_error, nrange, wthresh_error)
plt.xlabel('Data Point Number')
plt.title('Param = n')
plt.ylabel('Error')
plt.legend({'topk', 'thresh'})

nrange, _,wtopk_error, wthresh_error= error_calc(num_iters=10, param=
'd', n=1000, d=100, s=5, s_model=False, true_s=5)

plt.figure()
plt.plot(nrange, wtopk_error, nrange, wthresh_error)
plt.xlabel('Dimensions')
plt.title('Param = dimensions')
plt.ylabel('Error')
plt.legend()
plt.legend({'topk', 'thresh'})

nrange, _,wtopk_error, wthresh_error= error_calc(num_iters=10, param=
's', n=1000, d=100, s=5, s_model=False, true_s=5)

plt.figure()
plt.plot(nrange, wtopk_error, nrange, wthresh_error)
plt.xlabel('Sparsity')
plt.title('Param = sparsity')
plt.ylabel('Error')
plt.legend()
plt.legend({'topk', 'thresh'})
```

```
No handles with labels found to put in legend.
No handles with labels found to put in legend.
```

```
Out[38]: <matplotlib.legend.Legend at 0x7f9d97c40b70>
```







In the Model we see that with higher dimensional data we don't get more error, that is because we are setting our complexity within our sparsity to be set. Therefore, our variance is set, and our bias is already high. However when looking at the assumed sparsity of our model (which should be 100, since we don't have any sparsity), we see that the error stays the same. The randomness comes from our noise, however there seems to be no visible trend. We expect that as we increase sparsity, the more our error will decrease because we are approaching optimal variance and bias tradeoff as we approach the true model.

In [1]:
```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg

%matplotlib notebook
```

PART A

In [2]:
```
#     @staticmethod
#     def information_gain(X, y, thresh):
#         X0,y0,X1,y1 = split(X,y, thresh)
#         def g(y):
#             pyk_2 = 0
#             for i in range(2):
#                 try:
#                     pyeqk = len(y[y == i])/float(len(y))
#                     pyk_2 = pyeqk**2
#                 except ZeroDivisionError:
#                     pyeqk = 0
#             return 1 - pyk_2

#         return len(X0)*g(y0)/float(len(X)) + len(X1)*g(y1)/float(le
n(X))

#     @staticmethod
#     def gini_impurity(X, y, thresh):
#         X0,y0,X1,y1 = split(X,y, thresh)

#         def h(y) :
#             hy = 0
#             for i in range(2) :
#                 try :
#                     pyeqk = len(y[y == i])/float(len(y))
#                     hy = hy - pyeqk*np.log(pyeqk)
#                 except ZeroDivisionError :
#                     pyeqk = 0

#             return hy

#         return len(X0)*h(y0)/float(len(X)) + len(X1)*h(y1)/float(le
n(X))
```

PART C

PART D (Bagged Trees)

In [ ]:
```
# 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 = [
#             sklearn.tree.DecisionTreeClassifier(random_state=i, **s
elf.params)
#             for i in range(self.n)
#         ]

#     def fit(self, X, y):
#       self.mask = []
#       for tree in self.decision_trees:
#               mask = np.random.randint(0, high = len(X), size= len
(X))
#               Xsampling = X[mask,:]
#               ysampling = y[mask]
#               tree.fit(Xsampling,ysampling)
#               self.mask.append(mask)



#     def predict(self, X):
#       preds = []
#       for tree in self.decision_trees:
#               preds.append(tree.predict(X))
#       return stats.mode(np.array(preds), axis = 0 )[0].reshape(len
(X))
```

PART F (Random Forest)

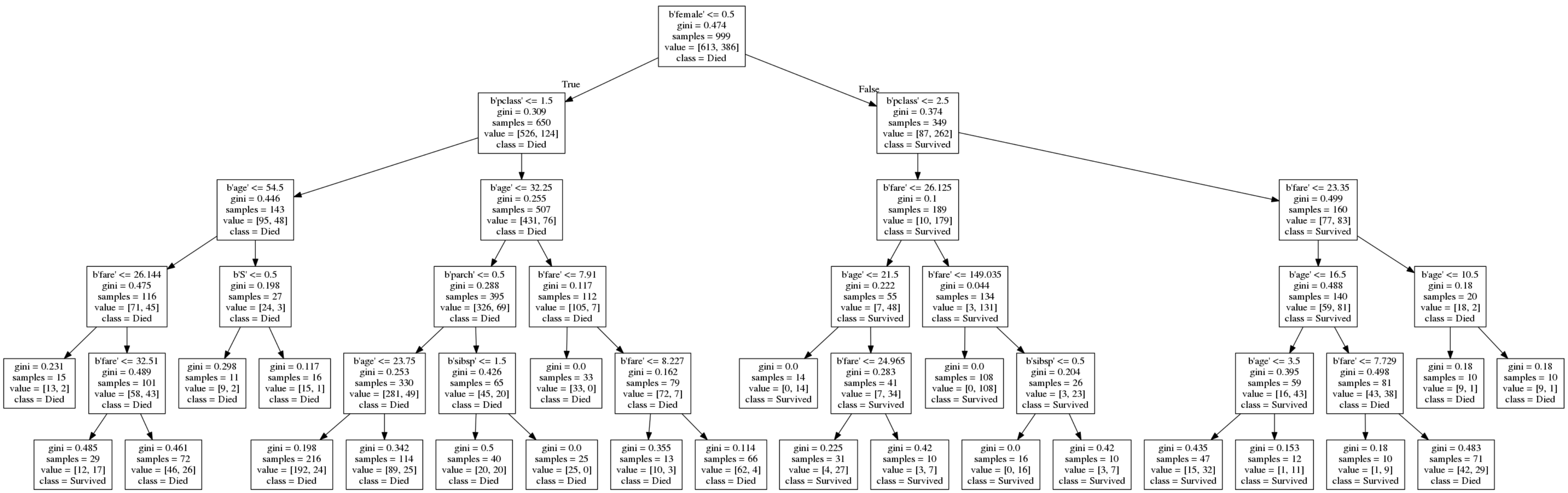
In [ ]:
```
# class RandomForest(BaggedTrees):
#     def __init__(self, params=None, n=200, m=2):
#       super().__init__(params = params , n = n )
#       self.m = m

#     def fit(self, X,y):
#       self.mask = []
#       self.features = []
#       for tree in self.decision_trees:
#               mask = np.random.randint(0, high = len(X), size= len
(X))
#               features = np.random.choice( X.shape[1], size = self.
m )
#               Xsampling = X[mask,:]
#               Xsampling = Xsampling[:,features]
#               ysampling = y[mask]
#               tree.fit(Xsampling,ysampling)
#               self.mask.append(mask)
#               self.features.append(features)
#     def predict(self,X):
#       preds = []
#       k = 0
#       for tree in self.decision_trees:
#               preds.append(tree.predict(X[:,self.features[k]]))
#       return stats.mode(np.array(preds), axis = 0 )[0].reshape(len
(X))
```

PART H (AdaBoost)

In [ ]:
```
# class BoostedRandomForest(RandomForest):
#     def fit(self, X, y ):
#         self.w = np.ones(X.shape[0]) / X.shape[0]  # Weights on dat
a
#         self.a = np.zeros(self.n)  # Weights on decision trees
#         k = 0
#         self.features = []
#         for tree in self.decision_trees:
#               mask = np.random.randint(0, high = len(X), size= len
(X))
#               features = np.random.choice(X.shape[1], size = self.m
 )
#               Xsampling = X[mask,:]
#               Xsampling = Xsampling[:,features]
#               ysampling = y[mask]
#               tree.fit(Xsampling,ysampling)
#               self.features.append(features)

#               ej = 0
#               for j in range(len(Xsampling)):
#                       ej = checkXY(Xsampling[j,:], ysampling[j], tr
ee )*self.w[j] + ej

#               ej = ej/float(sum(self.w))

#               self.a[k] = 0.5*np.log((1-ej)/float(ej))

#               for i in range(len(Xsampling)):
#                       if checkXY(Xsampling[i,:],ysampling[i], tree)
 > 0.5 :
#                               self.w[i] = self.w[i]*np.exp(self.a
[k])
#                       else :
#                               self.w[i] = self.w[i]*np.exp(-self.a
[k])
#               k = k + 1


#     def predict(self, X):
#         classes = list(set(y))
#         preds_tot = []
#         for i in range(len(X)):
#               preds = []
#               for c in classes :
#                       zj = 0
#                       k = 0
#                       for tree in self.decision_trees:
#                               Xcheck = X[:,self.features[k]]
#                               Xcheck = Xcheck[i,:]
#                               zj = zj + self.a[k]*checkXY(Xcheck,
 c, tree)
#                               k = k + 1
#                       preds.append(zj)
#                       preds_tot.append(classes[np.argmax(preds)])
#               return preds_tot
```

Part J (Results)

In [ ]:
```
#TITANIC :
# accuracy = [kfold1, kfold2, kfold3, avg_training]

# DecisionTree
# [0.5993975903614458, 0.63855421686746983, 0.60240963855421692, 0.61
369315342328834]
# BaggedTrees
# accuracy = [kfold1, kfold2, kfold3, avg_training]
# [0.79518072289156627, 0.76204819277108438, 0.77710843373493976, 0.9
7801099450274853]
# RandomForest
# accuracy = [kfold1, kfold2, kfold3, avg_training]
# [0.5993975903614458, 0.4006024096385542, 0.60240963855421692, 0.580
20989505247378]
# Adaboost
# [0.21686746987951808, 0.63855421686746983, 0.21385542168674698, 0.2
4837581209395301]




# SPAM

# DecisionTree
# accuracy = [kfold1, kfold2, kfold3, avg_training]
# [0.5993975903614458, 0.63855421686746983, 0.60240963855421692, 0.61
369315342328834]
# BaggedTrees
# accuracy = [kfold1, kfold2, kfold3, avg_training]
# [0.78915662650602414, 0.76204819277108438, 0.77409638554216864, 0.9
7801099450274853]
# RandomForest
# accuracy = [kfold1, kfold2, kfold3, avg_training]
# [0.5993975903614458, 0.6506024096385542, 0.61144578313253017, 0.628
68565717141422]
# Adaboost
# accuracy = [kfold1, kfold2, kfold3, avg_training]
# [0.5993975903614458, 0.22289156626506024, 0.21987951807228914, 0.24
887556221889059]
```


b'female' <= 0.5
gini = 0.474
samples = 999
value = [613, 386]
class = Died
True
False
b'pclass' <= 1.5
gini = 0.309
samples = 650
value = [526, 124]
class = Died
b'pclass' <= 2.5
gini = 0.374
samples = 349
value = [87, 262]
class = Survived
b'age' <= 54.5
gini = 0.446
samples = 143
value = [95, 48]
class = Died
b'age' <= 32.25
gini = 0.255
samples = 507
value = [431, 76]
class = Died
b'fare' <= 26.125
gini = 0.1
samples = 189
value = [10, 179]
class = Survived
b'fare' <= 23.35
gini = 0.499
samples = 160
value = [77, 83]
class = Survived
b'fare' <= 26.144
gini = 0.475
samples = 116
value = [71, 45]
class = Died
b'S' <= 0.5
gini = 0.198
samples = 27
value = [24, 3]
class = Died
b'parch' <= 0.5
gini = 0.288
samples = 395
value = [326, 69]
class = Died
b'fare' <= 7.91
gini = 0.117
samples = 112
value = [105, 7]
class = Died
b'age' <= 21.5
gini = 0.222
samples = 55
value = [7, 48]
class = Survived
b'fare' <= 149.035
gini = 0.044
samples = 134
value = [3, 131]
class = Survived
b'age' <= 16.5
gini = 0.488
samples = 140
value = [59, 81]
class = Survived
b'age' <= 10.5
gini = 0.18
samples = 20
value = [18, 2]
class = Died
gini = 0.231
samples = 15
value = [13, 2]
class = Died
b'fare' <= 32.51
gini = 0.489
samples = 101
value = [58, 43]
class = Died
gini = 0.298
samples = 11
value = [9, 2]
class = Died
gini = 0.117
samples = 16
value = [15, 1]
class = Died
b'age' <= 23.75
gini = 0.253
samples = 330
value = [281, 49]
class = Died
b'sibsp' <= 1.5
gini = 0.426
samples = 65
value = [45, 20]
class = Died
gini = 0.0
samples = 33
value = [33, 0]
class = Died
b'fare' <= 8.227
gini = 0.162
samples = 79
value = [72, 7]
class = Died
gini = 0.0
samples = 14
value = [0, 14]
class = Survived
b'fare' <= 24.965
gini = 0.283
samples = 41
value = [7, 34]
class = Survived
gini = 0.0
samples = 108
value = [0, 108]
class = Survived
b'sibsp' <= 0.5
gini = 0.204
samples = 26
value = [3, 23]
class = Survived
b'age' <= 3.5
gini = 0.395
samples = 59
value = [16, 43]
class = Survived
b'fare' <= 7.729
gini = 0.498
samples = 81
value = [43, 38]
class = Died
gini = 0.18
samples = 10
value = [9, 1]
class = Died
gini = 0.18
samples = 10
value = [9, 1]
class = Died
gini = 0.485
samples = 29
value = [12, 17]
class = Survived
gini = 0.461
samples = 72
value = [46, 26]
class = Died
gini = 0.198
samples = 216
value = [192, 24]
class = Died
gini = 0.342
samples = 114
value = [89, 25]
class = Died
gini = 0.5
samples = 40
value = [20, 20]
class = Died
gini = 0.0
samples = 25
value = [25, 0]
class = Died
gini = 0.355
samples = 13
value = [10, 3]
class = Died
gini = 0.114
samples = 66
value = [62, 4]
class = Died
gini = 0.225
samples = 31
value = [4, 27]
class = Survived
gini = 0.42
samples = 10
value = [3, 7]
class = Survived
gini = 0.0
samples = 16
value = [0, 16]
class = Survived
gini = 0.42
samples = 10
value = [3, 7]
class = Survived
gini = 0.435
samples = 47
value = [15, 32]
class = Survived
gini = 0.153
samples = 12
value = [1, 11]
class = Survived
gini = 0.18
samples = 10
value = [1, 9]
class = Survived
gini = 0.483
samples = 71
value = [42, 29]
class = Died