Alex Xu
Machine Learning
Professor Alvarez
Problem Set 4

1)

a) Cost of accepting = P[y = +1|x]*0 + P[y = -1|x]*c_a
= **(1-g(x))c_a**
Cost of rejecting = P[y = +1|x]*c_r + P[y -1|x]*0
= **g(x)c_r**

b) We only want to accept when the cost of accepting is ≤ the cost of rejecting.
Plugging in from what we derived in part a :
→ (1-g(x))c_a ≤ g(x)c_r
→ c_a - c_a *g(x) ≤ g(x)c_r
→ c_a ≤ g(x)(c_a + c_r)
→ $\frac{c_a}{c_a+c_r}$ ≤ g(x)

The Threshold = $\frac{c_a}{c_a+c_r}$ hence we reject when $\frac{c_a}{c_a+c_r}$ > g(x)

c) Supermarket threshold $= \frac{1}{10+1} = \boldsymbol{\frac{1}{11}}$

This threshold accommodates the fact that we would want to avoid a false negative more than a false positive.

CIA threshold = $\frac{1}{1000+1} = \boldsymbol{\frac{1}{1001}}$

In this case, we want to avoid false positives since letting in a dangerous acceptance can be detrimental. Hence, the high threshold makes sense.

2)

2a

In [1]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.datasets import load_wine
from sklearn.model_selection import cross_val_score
import numpy as np
import matplotlib.pyplot as plt
import math
```

In [2]:
```
def makePoly(deg):
    coefs = np.random.normal(size=deg+1)
    coefs = np.true_divide(coefs,np.sqrt(np.sum(coefs**2)))
    return coefs
```

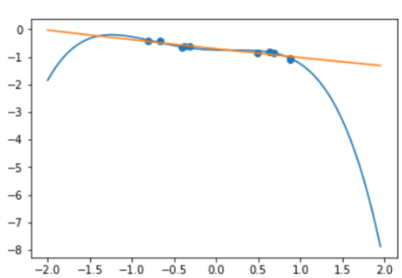
In [3]:
```
def randSample(numPoints, p, noiseStd):
    xi = np.random.uniform(-1, 1, size=numPoints)
    noise = np.random.normal(scale=noiseStd, size=numPoints)
    p_xi = p(xi + noise)
    return xi, p_xi
```

In [4]:
```
def oneFit(numPoints=10, degTarget=4, degFit=2, noiseStd=0.1):
    coefs = makePoly(degTarget)
    p = np.poly1d(coefs)
    xi,pred_xi = randSample(numPoints, p, noiseStd)
    #target function
    plot(p)
    #points
    plt.scatter(xi, pred_xi)
    #predicted graph
    p_prime = np.poly1d(np.polyfit(xi, pred_xi, degFit))
    plot(p_prime)
    #calc e_out
    x = np.arange(-2,2, 0.05)
    y = p(x)
    y_pred = p_prime(x)
    error_out = 0
    for i in range(len(y)):
        error_out += (y[i] - y_pred[i])**2
    error_out = error_out/len(y)
    #calc e_in
    error_in = 0
    for j in range(len(xi)):
        error_in += (p_prime(xi[j]) - pred_xi[j])**2
    error_in = error_in/len(xi)
    return error_in, error_out

```

In [5]:
```
def plot(p):
    x = np.arange(-2,2, 0.05)
    y = p(x)
    plt.plot(x, y)
```

In [6]:
```
oneFit()
```

Out[6]: (0.004506580022704715, 2.610329133969736)



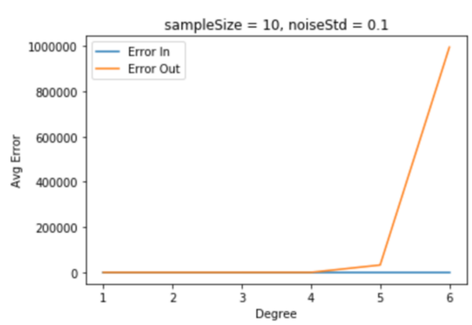
2b

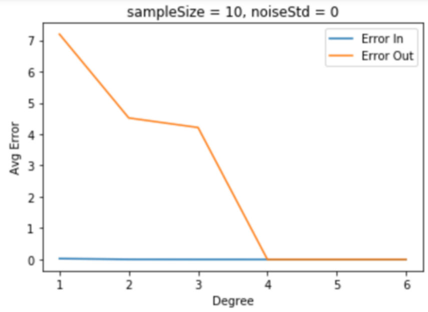
In [7]:
```
#no Plot
def oneFit_noPrint(numPoints=10, degTarget=4, degFit=2, noiseStd=0.1):
    coefs = makePoly(degTarget)
    p = np.poly1d(coefs)
    xi,pred_xi = randSample(numPoints, p, noiseStd)
    p_prime = np.poly1d(np.polyfit(xi, pred_xi, degFit))
    #calc e_out
    x = np.arange(-2,2, 0.05)
    y = p(x)
    y_pred = p_prime(x)
    error_out = 0
    for i in range(len(y)):
        error_out += (y[i] - y_pred[i])**2
    error_out = error_out/len(y)
    #calc e_in
    error_in = 0
    for j in range(len(xi)):
        error_in += (p_prime(xi[j]) - pred_xi[j])**2
    error_in = error_in/len(xi)
    return error_in, error_out
```

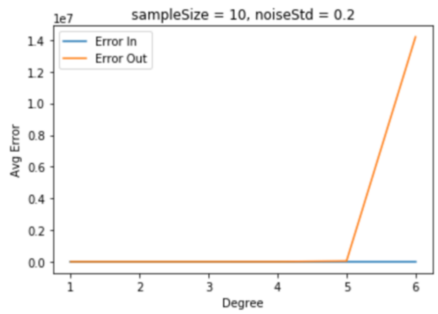
In [8]:
```
def testRangeDegrees(degrees=6, sampleSize=20, degTarget=4, noiseStd=0.1):
    answer = np.empty((degrees, 2))
    for counter in range(6):
        counter_2 = counter + 1
        b_avg = np.empty((1000,2))
        for i in range(1000):
            exp_in, exp_out = oneFit_noPrint(sampleSize, degTarget, counter_2, noiseStd)
            b_avg[i, 0] = exp_in
            b_avg[i, 1] = exp_out
        answer[counter] = np.mean(b_avg, axis=0)
    return answer

a = testRangeDegrees()
a

```

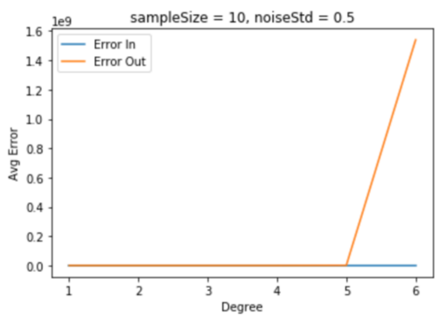
Out[8]:
```
array([[4.44087945e-02, 6.96669436e+00],
       [1.52368202e-02, 4.27435777e+00],
       [9.97849182e-03, 3.70950625e+00],
       [8.73317273e-03, 6.32141119e+00],
       [7.55428524e-03, 1.03322615e+02],
       [7.53490191e-03, 5.01167410e+03]])
```

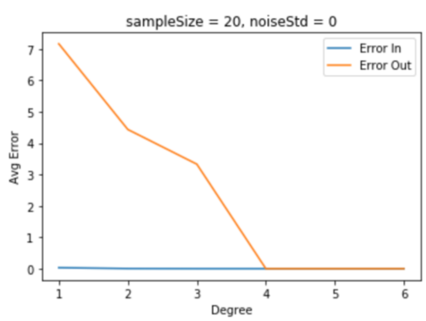
In [9]:
```
samples = [10, 20, 50, 100]
noise = [0, 0.1, 0.2, 0.5]
x_axis = [1, 2, 3, 4, 5, 6]

for size in samples:
    for n in noise:
        avg_error = testRangeDegrees(degrees = 6, sampleSize= size, degTarget= 4, noiseStd=n)
        plt.figure()
        plt.plot(x_axis, avg_error[:, [0]], label = 'Error In')
        plt.plot(x_axis, avg_error[:, [1]], label = 'Error Out')
        plt.ylabel("Avg Error")
        plt.xlabel("Degree")
        plt.title("sampleSize = {}, noiseStd = {}".format(size,n))
        plt.legend()
        plt.show()

```

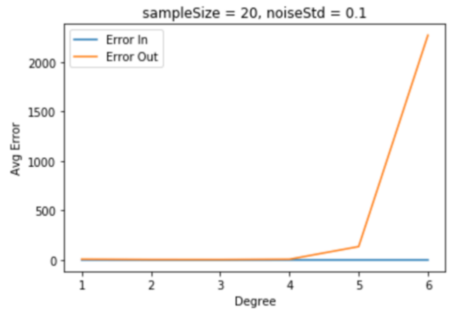


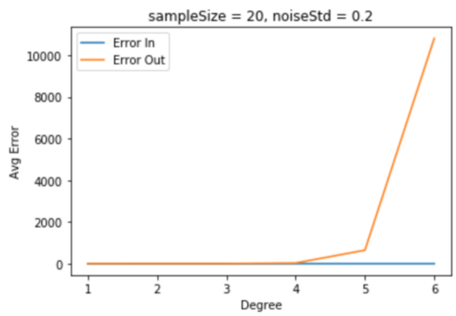

sampleSize = 10, noiseStd = 0
Error In
Error Out
Avg Error
Degree

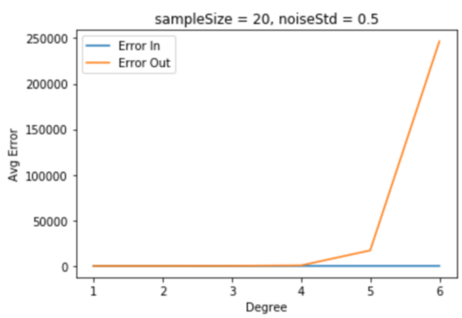

sampleSize = 10, noiseStd = 0.2
1e7
Error In
Error Out
Avg Error
Degree

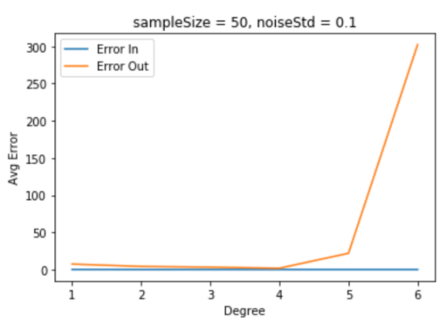

sampleSize = 10, noiseStd = 0.5
1e9
Error In
Error Out
Avg Error
Degree

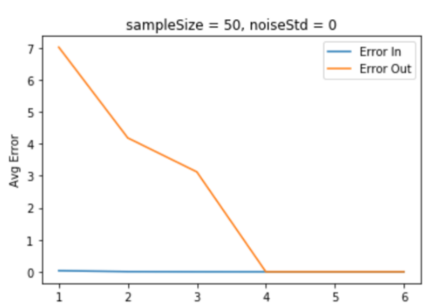

sampleSize = 20, noiseStd = 0
Error In
Error Out
Avg Error
Degree

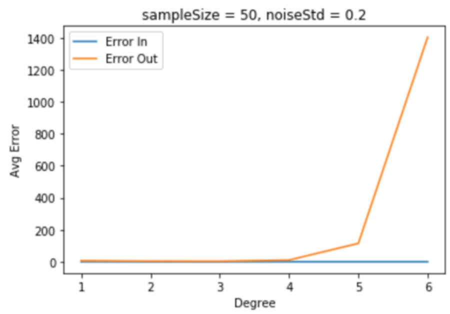

sampleSize = 20, noiseStd = 0.1
Error In
Error Out
Avg Error
Degree

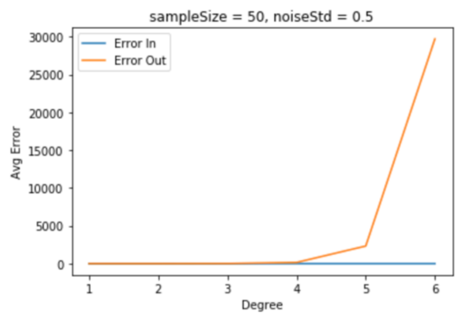

sampleSize = 20, noiseStd = 0.2
Error In
Error Out
Avg Error
Degree

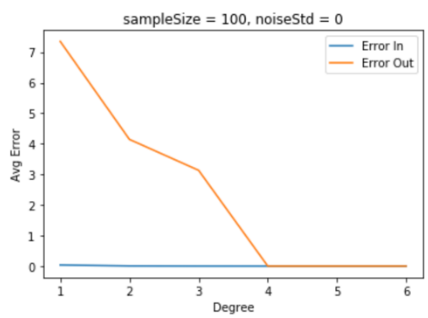

sampleSize = 20, noiseStd = 0.5
Error In
Error Out
Avg Error
Degree

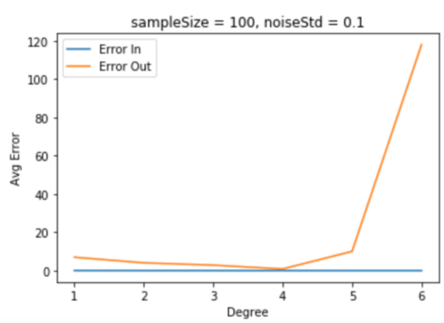

sampleSize = 50, noiseStd = 0.1
Error In
Error Out
Avg Error
Degree

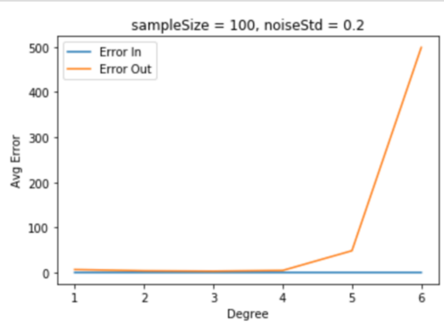

sampleSize = 50, noiseStd = 0
Error In
Error Out
Avg Error
Degree

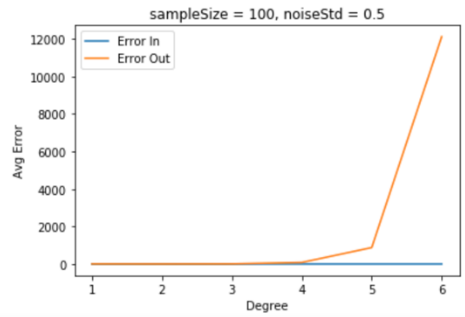

sampleSize = 50, noiseStd = 0.2
Error In
Error Out
Avg Error
Degree
0
200
400
600
800
1000
1200
1400
1
2
3
4
5
6


sampleSize = 50, noiseStd = 0.5
Error In
Error Out
Avg Error
Degree
0
5000
10000
15000
20000
25000
30000
1
2
3
4
5
6


sampleSize = 100, noiseStd = 0
Error In
Error Out
Avg Error
Degree
0
1
2
3
4
5
6
7
1
2
3
4
5
6


sampleSize = 100, noiseStd = 0.1
Error In
Error Out
Avg Error
Degree
0
20
40
60
80
100
120
1
2
3
4
5
6


sampleSize = 100, noiseStd = 0.2
Error In
Error Out
Avg Error
Degree
0
100
200
300
400
500
1
2
3
4
5
6



When noise is = 0, the out of sample error is minimized regardless of sample

3)

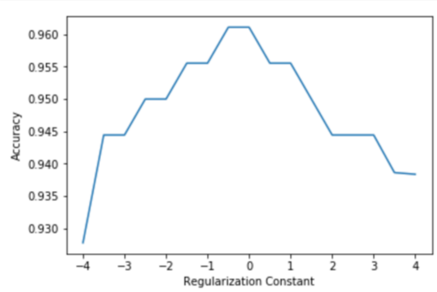
3a

In [10]:
```python
wine = load_wine()
data = wine.data
target = wine.target
```

In [11]:
```python
n = list(np.arange(-8,9))
x_axis = []
acc = []
for element in n:
    c = 2**(0.5*element)
    x_axis.append(math.log2(c))
    clf = LogisticRegression(solver = 'liblinear', C = c)
    clf.fit(data, target)
    score = cross_val_score(clf, data, target, cv=4)
    acc.append(np.mean(score))
```

...

In [12]:
```python
plt.xlabel('Regularization Constant')
plt.ylabel('Accuracy')
plt.plot(x_axis,acc)
plt.show()
```



3b

There looks to be a sweet spot in which the accuracy peeks. This makes sense since the regularization constant, if too big, it would start to block the model from accurately predicting the data. If too small, the noise would still have too much of an effect causing overfitting to occur. The ultimate goal of regularization, when coupled with validation, is to find a constant such that the effect of noise is dimmed while not hurting the model's ability to accurately train on data

4)

4a

1602

Leave one out validation means that one sample is not used when validating. The number of samples in the Wine data set is 178, which means there are 178 C 1 ways to pick the sample left out. Since there are three p values and three c values, it makes since that we must train the classifiers 178 C 1 * 3 * 3 times = 1602

4b

In [13]:

```
from sklearn.model_selection import GridSearchCV, LeaveOneOut
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import classification_report
```

In [14]:

```
numSamples = len(wine.data)
numSamples
```

Out[14]: 178

In [15]:

```
tuned_parameters = [{'kernel': ['poly'], 'C': [0.5, 1, 2], 'degree': [1,2,4]}]
cross_val = LeaveOneOut()


```

In [18]:

```

clf = GridSearchCV(SVC(gamma = 'scale'), tuned_parameters, iid = False)
clf.fit(data, target)

print("Best parameters is:")
print(clf.best_params_)

```

```
/Users/alexanderxu/anaconda3/lib/python3.7/site-packages/sklearn/model_selection/_split.py:2053: FutureWarning: You s
hould specify a value for 'cv' instead of relying on the default value. The default value will change from 3 to 5 in
version 0.22.
  warnings.warn(CV_WARNING, FutureWarning)
```

```
Best parameters is:
{'C': 1, 'degree': 2, 'kernel': 'poly'}
```

In [ ]:

5)

a) $\nabla E_{aug}(w(t)) = \nabla E_{in}(w(t)) + 2\lambda w(t)$

Using this, we can find that:

$w(t+1) \leftarrow w(t) - \eta\nabla E_{aug}(w(t))$

$= w(t) - \eta(\nabla E_{in}(w(t)) + 2\lambda w(t))$

$= w(t) - \eta\nabla E_{in}(w(t)) + 2\eta\lambda w(t)$

$= \mathbf{(1-2\eta\lambda)w(t) - \eta\nabla E_{in}(w(t))}$

b) Given $y_i \mathrel{!}= w^T x_i$

$e_i(w) = \max(0, 1 - y_i w^T x_i)$

Hence:

$$\nabla e_i(w) = \begin{cases} -yx, & h_i(w) \leq 0 \\ 0, & h_i(w) > 0 \end{cases}$$

c) Using the logic from above:

$$\nabla e_i(w) = \begin{cases} -yx, & h_i(w) \leq 0 \\ 0, & h_i(w) > 0 \end{cases}$$

But we can substitute $h_i(w)$ for $1 - y_i w^T x_i$ thus getting:

$$\nabla e_i(w) = \begin{cases} -yx, & 1 - y_i w^T x_i \leq 0 \\ 0, & 1 - y_i w^T x_i > 0 \end{cases}$$

Thus, if $1 - y_i w^T x_i \leq 0$, meaning we need to update, then:

From part a we can sub:

$w(t+1) = (1-2\eta\lambda)w(t) - \eta\nabla E_{in}(w(t))$

And further make this into:

$$w(t+1) = (1-2\eta\lambda)(1 - y_i w^T x_i) - \eta\nabla Ein(w(t))$$
$$w(t+1) = 1 - y_i w^T x_i - 2\eta\lambda + 2\eta\lambda\, y_i w^T x_i$$
$$w(t+1) = 1 - y_i w^T x_i - 2\eta\lambda(1 + y_i w^T x_i)$$
$$w(t+1) = -(y_i w^T x_i - 1) - 2\eta\lambda(1 + y_i w^T x_i)$$

If $1 - y_i w^T x_i > 0$ then we do nothing since no update is needed