In [93]:
```
import math
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
from scipy.stats import multivariate_normal
import pickle

%matplotlib inline
%load_ext autoreload
%autoreload 2
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

1. Deicison Tree Code

In [88]:
```
class Node:
    def __init__(self, depth):
        self.qda = False
        self.left = None
        self.right = None
        self.threshold = None
        self.feature_index = None
        self.depth = depth

        self.isLeaf = False
        self.label = None
        self.mu1, self.sig1, self.mu2, self.sig2, self.prior1, self.prior2 =
```

In [ ]:
```
class DecisionTree:
    def __init__(self, maxDepth=None, n=3, features=None, size_limit=10, qda
        self.qda = qda
        self.root = Node(0)
        self.nodes = []
        self.maxDepth = maxDepth
        self.n_thres = n
        self.features = features
        self.size_limit = size_limit

    def train(self, curr, x, y):
        if not curr:
            pass
        elif (not self.entropy(y)) or (curr.depth == self.maxDepth) or (len(
            counts = np.bincount(y)
            curr.label = np.argmax(counts)
            curr.isLeaf = True
        else:
            curr.feature_index, curr.threshold = self.segmentor(x, y)
            leftX, leftY, rightX, rightY = self.split(x, y, threshold=curr.t

            if self.qda:
                if self.entropy(y) > 0.1 and curr.depth < 2:
                    curr.mu1, curr.sig1, curr.mu2, curr.sig2, curr.prior1, c
                    leftX_Q, leftY_Q, rightX_Q, rightY_Q = self.qda_split(x,
                                                                           mu
                                                                           pr
                    if self.impurity(leftY_Q, rightY_Q) < self.impurity(left
                        leftX, leftY, rightX, rightY = leftX_Q, leftY_Q, rig
                        curr.qda = True
            else:
                if leftY:
                    curr.left = Node(curr.depth+1)
                    self.train(curr.left, leftX, leftY)

                if rightY:
                    curr.right = Node(curr.depth+1)
                    self.train(curr.right, rightX, rightY)

    def predict(self, x):
        y_hat = []
        for sample in x:
            y_hat.append(self._predict(sample))
        return y_hat

    def _predict(self, x): # x is one sample
        curr = self.root
        while not curr.isLeaf:
            if curr.qda and len(x) == len(curr.mu1):
                    curr = curr.left
                else:
                    curr = curr.right
            else:
                if x[curr.feature_index] < curr.threshold:
                    curr = curr.left
                else:
```

```
                curr = curr.right
        return curr.label

    def evaluate(self, x, y):
        y_hat = self.predict(x)
        accuracy = np.mean(np.equal(y, y_hat).astype(int))
        return accuracy

    def entropy(self, s):
        _, counts = np.unique(s, return_counts=True)
        entro = 1.0 * counts / len(s) * np.log(counts / len(s))
        return -np.sum(entro)

    def segmentor(self, x, y):
        # Find best (feature, threshold) pair
        x_reshape = np.transpose(x)
        impurities = []
        for f in range(len(x[0])): # Find best threshold for each feature
            thresholds = np.linspace(int(min(x_reshape[f])), int(max(x_resha
            temp = {}
            for thr in thresholds:
                leftX, leftY, rightX, rightY = self.split(x, y, threshold=th
                imp = self.impurity(leftY, rightY)
                temp[thr] = imp
            min_thr = min(temp, key=temp.get)
            impurities.append([min_thr, temp[min_thr]]) # Threshold, Impuri
        f_i = np.argmin([a[1] for a in impurities])
        return f_i, impurities[f_i][0]

    def split(self, x, y, threshold=None, i=None, mu1=None, sig1=None, mu2=N
        leftX, leftY, rightX, rightY = [], [], [], []
        if self.qda:
            for i in range(len(x)):
                pr1 = multivariate_normal.logpdf(x[i], mu1, sig1 + 0.001 * r
                pr2 = multivariate_normal.logpdf(x[i], mu2, sig0 + 0.001 * r
                if pr1 - pr2 > 0:
                    leftX.append(x[i])
                    leftY.append(y[i])
                else:
                    rightX.append(x[i])
                    rightY.append(y[i])
            return leftX, leftY, rightX, rightY

        for j in range(len(x)):
            if x[j][i] < threshold:
                leftX.append(x[j])
                leftY.append(y[j])
            else:
                rightX.append(x[j])
                rightY.append(y[j])
        return leftX, leftY, rightX, rightY

    def impurity(self, leftY, rightY):
        lenL, lenR = len(leftY), len(rightY)
        h_after = 1.0 * (self.entropy(leftY) * lenL + self.entropy(rightY) *
        return h_after
```

```
    def visualize(self):
        # Feature name, split rule, class
        def _print(node, num):
            if node:
                if not node.isLeaf:
                    print('level ', str(num), '-', 'feature:', self.features
                          ';','split:', node.threshold)
                else:
                    print('level ', str(num), '-', 'class:', str(node.label)

        _print(self.root, 1)
        _print(self.root.left, 2); _print(self.root.right, 2)
        _print(self.root.left.left, 3); _print(self.root.left.right, 3)
        _print(self.root.right.left, 3); _print(self.root.right.right, 3)

    def extract_qda_info(self, x, y):
        length = len(x)
        x1, x2, y1, y2 = [], [], [], []
        for i in range(length):
            if y[i] == 0:
                x1.append(x[i])
                y1.append(y[i])
            else:
                x2.append(x[i])
                y2.append(y[i])
        return np.mean(x_0, axis=0),
                np.cov(x_0, rowvar=False),
                np.mean(x_1, axis=0),
                np.cov(x_1, rowvar=False),
                1.0 * math.log((np.count_nonzero(y == 0.0) + 1) / length,
                1.0 * math.log(np.count_nonzero(y == 1.0) + 1) / length
```

2. Random Forest Code

In [91]:
```
from random import randrange

class RandomForest:
    def __init__(self, max_depth=None, num_trees=None, sample_size=None, fea
        self.trees = []
        self.sample_size = sample_size
        self.num_trees = num_trees
        self.max_depth = max_depth
        self.features = features
        self.n = n

    def train(self, x, y):
        for i in range(self.num_trees):
            sub_x, sub_y = self.sample(x, y)
            tree = DecisionTree(maxDepth=self.max_depth, n=self.n, features=
            tree.train(tree.root, sub_x, sub_y)
            self.trees.append(tree)

    def predict(self, x):
        y_hat = []
        for sample in x: # For each point
            logits = []
            for tree in self.trees: # For each decision tree
                logits.append(tree._predict(sample))
            best = max(set(logits), key=logits.count)
            y_hat.append(best)
        return y_hat

    def evaluate(self, x, y):
        y_hat = self.predict(x)
        accuracy = np.mean(np.equal(y, y_hat).astype(int))
        return accuracy

    def sample(self, x, y):
        sub_x, sub_y = [], []
        while len(sub_x) < self.sample_size:
            i = randrange(len(x))
            sub_x.append(x[i])
            sub_y.append(y[i])
        return sub_x, sub_y
```

3. Implementation Details

(a) How did you deal with categorical features and missing values?

- For categorical features, I vectorize them. For missing values, I replace them with the mode (the value that occurs most frequently. (Implementation details are in preprocess.ipynb

(b) What was your stopping criteria?

- Two criteria: if all samples in this node have the same y values, stop; if the tree has reached the maximum depth i set beforehand, stop.

(c) Did did do anything special to speed up training?

- I did not do anything fancy. I just avoid writing necessary loops in segmantor function

(d) How did you implement random forests?

- I set a sample size n and the number of trees I want in my forest. Everytime I construct a tree, I randomly extract n points from the training data and use my Decision Tree class to construct it.

(e) Anything else cool you implemented?

- Both my classifiers are pretty standard because they have reached a good enough result

4. Evaluation

Census

- Census Decision Tree Training Accuracy: 0.86160
- Census Decision Tree Validation Accuracy: 0.85640
- Census Random Forest Training Accuracy: 0.84057
- Census Random Forest Validation Accuracy: 0.84499

In [44]:
```
# Data
df = pd.read_csv('census/census_clean.csv', sep=',')
df.reindex(np.random.permutation(df.index))
y = np.array(df.label)
del df['label']
df = df.drop(df.columns[0], axis=1)
x = np.array(df)
census_columns = df.columns.tolist()

n = 5000
train_x = x[n:]
train_y = y[n:]
val_x = x[:n]
val_y = y[:n]

test_data = pd.read_csv("census/census_test_clean.csv")
diff = list(set(census_columns) - set(test_data.columns.tolist()))
for c in diff:
    test_data[c] = 0
test_data = test_data[census_columns]
```

In [6]:
```
# Decision Tree
cdt = DecisionTree(maxDepth=10, n=10, features=census_columns)
cdt.train(cdt.root, train_x, train_y)
train_acc = cdt.evaluate(train_x, train_y)
val_acc = cdt.evaluate(val_x, val_y)
print("Census Decision Tree: train acc = {}, validate acc = {}".format(trair
```

```
Census Decision Tree: train acc = 0.8616000577117299, validate acc = 0.85
64
```

In [7]:
```
# Random Forest
crf = RandomForest(max_depth=10, num_trees=50, sample_size=math.sqrt(len(tra
crf.train(train_x, train_y)
train_acc = crf.evaluate(train_x, train_y)
val_acc = crf.evaluate(val_x, val_y)
print("Census Random Forest: train acc = {}, validate acc = {}".format(trair
```

```
Census Random Forest: train acc = 0.838262876929736, validate acc = 0.841
```

In [8]:
```
y = cdt.predict(test_data.values)
df = pd.DataFrame(data = y, columns=["Category"])
df.index += 1
df.index.name = "Id"
df.to_csv("census/census_DT.csv")
```

Titanic

- Titanic Decision Tree Training Accuracy: 0.88666
- Titanic Decision Tree Validation Accuracy: 0.80000
- Titanic Random Forest Training Accuracy: 0.806666
- Titanic Random Forest Validation Accuracy: 0.80000

In [89]:
```
# Data
df = pd.read_csv('titanic/titanic_clean.csv', sep=',')
df.reindex(np.random.permutation(df.index))
y = np.array(df.survived)
del df['survived']
df = df.drop(df.columns[0], axis=1)
x = np.array(df)
titanic_columns = df.columns.tolist()

n = 100
train_x = x[n:]
train_y = y[n:]
val_x = x[:n]
val_y = y[:n]

test_data = pd.read_csv("titanic/titanic_test_clean.csv")
diff = list(set(titanic_columns) - set(test_data.columns.tolist()))
for c in diff:
    test_data[c] = 0
test_data = test_data[titanic_columns]
```

In [10]:
```
# Decision Tree
tdt = DecisionTree(maxDepth=11, n=10, features=titanic_columns)
tdt.train(tdt.root, train_x, train_y)
train_acc = tdt.evaluate(train_x, train_y)
val_acc = tdt.evaluate(val_x, val_y)
print("Titanic Decision Tree: train acc = {}, validate acc = {}".format(trai
```

```
Titanic Decision Tree: train acc = 0.8866666666666667, validate acc = 0.8
```

In [90]:
```
# Random Forest (This is too slow so I run in a local file)
trf = RandomForest(max_depth=10, num_trees=100, sample_size=math.sqrt(len(tr
trf.train(train_x, train_y)
train_acc = trf.evaluate(train_x, train_y)
val_acc = trf.evaluate(val_x, val_y)
print("Titanic Random Forest: train acc = {}, validate acc = {}".format(trai
```

...

In [ ]:
```
y = trf.predict(test_data.values)
df = pd.DataFrame(data = y, columns=["Category"])
df.index += 1
df.index.name = "Id"
df.to_csv("titanic/titanic_RF.csv")
```

Spam

- Spam Decision Tree Training Accuracy: 0.95576
- Spam Decision Tree Validation Accuracy: 0.9485
- Spam Random Forest Training Accuracy: 0.93696
- Spam Random Forest Validation Accuracy: 0.9355

In [96]:
```
df = pd.read_csv('spam/spam_clean.csv', sep=',')
df = df.reindex(np.random.permutation(df.index))
y = np.array(df[df.columns[346]])
df = df.drop(df.columns[[0, 346]], axis=1)
x = np.array(df)
spam_columns = df.columns.tolist()

n = 2000
train_x = x[n:]
train_y = y[n:]
val_x = x[:n]
val_y = y[:n]

test_data = pd.read_csv("spam/spam_test_clean.csv")
test_data = test_data.drop(test_data.columns[0], axis=1)
```

In [97]:
```
# Decision Tree
sdt = DecisionTree(maxDepth=5, n=3, features=spam_columns)
sdt.train(sdt.root, train_x, train_y)
train_acc = sdt.evaluate(train_x, train_y)
val_acc = sdt.evaluate(val_x, val_y)
print("Spam Decision Tree: train acc = {}, validate acc = {}".format(train_a
```

```
Spam Decision Tree: train acc = 0.9558566030780573, validate acc = 0.9525
```

In [ ]:
```
# Random Forest (this runs too slow so I ran it in the cloud)
srf = RandomForest(max_depth=5, num_trees=20, sample_size=math.sqrt(len(trai
srf.train(train_x, train_y)
train_acc = srf.evaluate(train_x, train_y)
val_acc = srf.evaluate(val_x, val_y)
```

In [128]:
```
with open("spam_rf.p", "rb") as f:
    srf = pickle.load(f)
print("Loaded Spam Random Forest")
```

```
Loaded Spam Random Forest
```

In [98]:
```
y = sdt.predict(test_data.values)
df = pd.DataFrame(data = y, columns=["Category"])
# df.index += 1
df.index.name = "Id"
df.to_csv("spam/spam_RF.csv")
```

Kaggle

- Census Kaggle: 0.85643
- Titanic Kaggle: 0.81935
- Spam Kaggle: 0.94480

  Kaggle Name: yika

5. Spam

a) Feature Transformation

Used bag of words approach plus normalization. More specifically, the features are the freqencies of each unique word in the text; after normalization, they become probabilities. For feature adding, I used "A List of Common Spam Words" (https://emailmarketing.comm100.com/email-marketing-ebook/spamwords.aspx (https://emailmarketing.comm100.com/email-marketing-ebook/spamwords.aspx)) as a reference to manually select words that appear most offen and intuitively make sense and added them to featurize.py file.

b) Print Out Path Trace for Each Label

In [20]:
```
# Print out the trace of the path
def trace(self, x): # x is one sample
    curr = self.root
    while not curr.isLeaf:
        print('Split at ', self.features[curr.feature_index],
              ', threshold: ', str(curr.threshold))
        if x[curr.feature_index] < curr.threshold:
            curr = curr.left
        else:
            curr = curr.right
    print('y_hat: ', str(curr.label))
    return curr.label
```

In [99]:
```
temp = pd.read_csv('spam/spam_clean.csv', sep=',')
temp = temp.drop(temp.columns[[0, 346]], axis=1)
one = np.array(temp.iloc[[-5]])[0]
zero = np.array(temp.iloc[[5]])[0]
```

In [100]:
```
# Path of One
trace(sdt, one)
```

```
Split at  278 , threshold:  0.5
Split at  0 , threshold:  0.0
Split at  0 , threshold:  0.0
Split at  0 , threshold:  0.0
Split at  0 , threshold:  0.0
y_hat:  1
```

Out[100]: 1

In [101]:
```
# Path of Zero
trace(sdt, zero)
```

```
Split at  278 , threshold:  0.5
Split at  0 , threshold:  0.0
Split at  0 , threshold:  0.0
Split at  0 , threshold:  0.0
Split at  0 , threshold:  0.0
y_hat:  1
```

Out[101]: 1

c) Most Common Split Rules in Random Forest

In [130]:
```
def counter(self, features):
    splits = {}
    for t in self.trees:
        i = t.root.feature_index
        if str(i) not in splits:
            splits[str(i)] = [t]
        else:
            splits[str(i)].append(t)
    return splits

def max_split(splits):
    most = 0
    f = None
    for key in splits.keys():
        n = len(splits[key])
        if n >= most:
            most = n
            f = key
    return f

def print_max(splits, columns):
    f = max_split(splits)
    if not f:
        return
    f = int(f)
    feature = columns[f]
    threshold = splits[str(f)][0].root.threshold
    num = len(splits[str(f)])
    splits.pop(str(f))
    print("Feature = {}, Threshold = {}, ({} trees)".format(feature, thresho
    return splits
```

In [131]:
```
dic = counter(srf, spam_columns)
dic = print_max(dic, spam_columns)
dic = print_max(dic, spam_columns)
dic = print_max(dic, spam_columns)
```

```
Feature = 278, Threshold = 0.5, (13 trees)
Feature = 0, Threshold = 0.0, (7 trees)
```

6. Census

a) Feature Transformation

I did not transform features, I only vectorized them

b) Print Out Path Trace for Each Label

In [34]:
```
temp = pd.read_csv('census/census_clean.csv', sep=',')
temp = temp.drop(temp.columns[0], axis=1)
del temp['label']
one = np.array(temp.iloc[[4]])[0]
zero = np.array(temp.iloc[[3]])[0]
```

In [35]:
```
# Path of One
trace(cdt, one)
```

```
Split at  marital-status_Married-civ-spouse , threshold:  0.111111111111
Split at  education-num , threshold:  12.6666666667
Split at  education-num , threshold:  8.33333333333
Split at  age , threshold:  34.0
Split at  capital-gain , threshold:  11111.0
Split at  hours-per-week , threshold:  80.8888888889
Split at  occupation_Farming-fishing , threshold:  0.111111111111
y_hat:  1
```

Out[35]: 1

In [36]:
```
# Path of Zero
trace(cdt, zero)
```

```
Split at  marital-status_Married-civ-spouse , threshold:  0.111111111111
Split at  education-num , threshold:  12.6666666667
Split at  education-num , threshold:  8.33333333333
Split at  age , threshold:  34.0
Split at  capital-gain , threshold:  11111.0
Split at  capital-gain , threshold:  7070.0
Split at  education-num , threshold:  9.33333333333
Split at  capital-loss , threshold:  1638.0
Split at  hours-per-week , threshold:  33.6666666667
Split at  occupation_Exec-managerial , threshold:  0.111111111111
y_hat:  0
```

Out[36]: 0

c) Most Common Split Rules in Random Forest

In [56]:
```
print("Ranking for most common split rules:")
dic = counter(crf, census_columns)
dic = print_max(dic, census_columns)
dic = print_max(dic, census_columns)
dic = print_max(dic, census_columns)
```

```
Ranking for most common split rules:
Feature = marital-status_Married-civ-spouse, Threshold = 0.5, (36 trees)
Feature = relationship_Husband, Threshold = 0.5, (10 trees)
Feature = marital-status_Never-married, Threshold = 0.5, (2 trees)
```

d) Evaluate Different Max Depth for Decision Tree

In [92]:

```
df = pd.read_csv('census/census_clean.csv', sep=',')
df.reindex(np.random.permutation(df.index))
y = np.array(df.label)
del df['label']
df = df.drop(df.columns[0], axis=1)
x = np.array(df)
census_columns = df.columns.tolist()

n = int(0.8 * len(x))
train_x = x[n:]
train_y = y[n:]
val_x = x[:n]
val_y = y[:n]
```

In [ ]:

```
# This takes too long I train it in the cloud
accs = []
depths = []
for i in range(2, 41):
    dt = DecisionTree(maxDepth=i, n=10)
    dt.train(dt.root, train_x, train_y)
    acc = dt.evaluate(val_x, val_y)
    accs.append(acc)
```

In [113]:

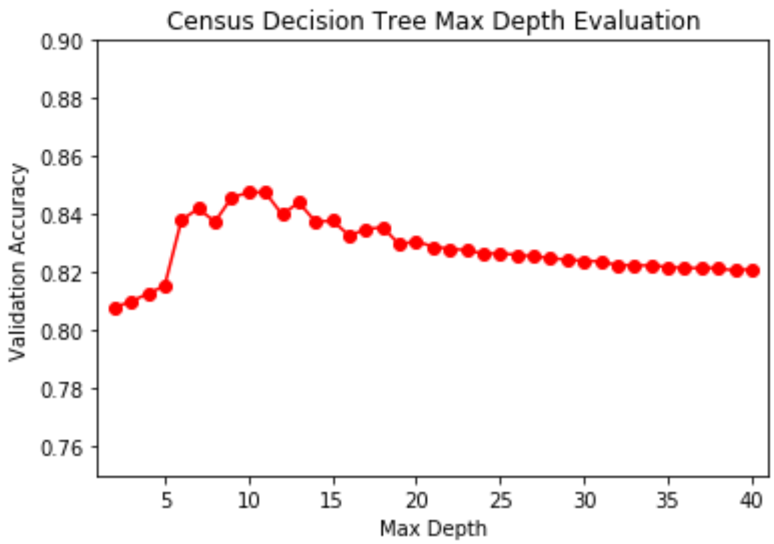
```
depths = list(range(2, 41))
with open("census_accs.p", "rb") as f:
    accs = pickle.load(f)
```

In [117]:
```
fig, ax = plt.subplots()

ax.scatter(depths, accs, color="red")
ax.plot(depths, accs, color="red")

ax.set_xlabel('Max Depth')
ax.set_ylabel('Validation Accuracy')
ax.set_title('Census Decision Tree Max Depth Evaluation')

plt.ylim((0.75, 0.9))
plt.xlim((1, 41))
plt.show()
```



Comments: The Decision Tree reaches a peak of accuracy at the depth around 10. Between 5 and 15 the accuracy fluctuate a relatively more, and then after 15 the accuracy seems to slowly converge.

7. Titanic

In [59]:
```
# Data
df = pd.read_csv('titanic/titanic_clean.csv', sep=',')
df.reindex(np.random.permutation(df.index))
y = np.array(df.survived)
del df['survived']
df = df.drop(df.columns[0], axis=1)
x = np.array(df)
titanic_columns = df.columns.tolist()

n = 100
train_x = x[n:]
train_y = y[n:]
```

In [60]:
```
dt = DecisionTree(maxDepth=3, n=20, features=titanic_columns)
dt.train(dt.root, train_x, train_y)
```

In [62]:
```
print("Visualize Decision Tree (nodes are printed in an order from left to r
dt.visualize()
```

```
Visualize Decision Tree (nodes are printed in an order from left to righ
t)
level  1 - feature: sex_male ; split: 0.0526315789474
level  2 - feature: pclass ; split: 2.05263157895
level  2 - feature: pclass ; split: 1.10526315789
level  3 - feature: cabin_letter_C ; split: 0.0526315789474
level  3 - feature: fare ; split: 32.5263157895
level  3 - feature: age ; split: 12.6315789474
level  3 - feature: age ; split: 3.89473684211
```

In [ ]: