CS 189: Homework 5

Michael Stephen Chen
Kaggle Acct: michaelstchen
SID: 23567341

April 13, 2016

Implemetation Explanation

1. I implemented my **Decision Tree** so that it would take in a couple of hyperparameters: max depth, minimum leaf size, and number of features bagged per node. All three features effect the runtimes of my simulation (i.e. decreasing max depth, number of bagged features, and/or increasing min leaf size all decrease the amount of computation required to generate a tree). In the cases where a growing branch is terminated prematurely, either because it reached the max depth or min leaf size, then we set the label of the leaf by majority vote. At each node features are randomly chosen/bagged, without replacement of course. For quantitative features, we do not test each possible split but instead choose up to 4 equally spaces splits based on the range of the feature's values. This is because it would be computationally intensive to check all possible splits, especially for some of the quantitative features in the census data (i.e. "capital-gain" for which each sample probably has a unique value).

2. For my **Random Forests** implentation, I didn't have to make any additional moifications to my DecisionTree class because I had already implemented feature bagging at every node. Essentially I maintained a list of $T = 30$ trees that were all trained on different data bags that contains randomly chose samples, with replacement, from the overall training data set. I then use all T trees to predict each data point, and take the rounded average as my final prediction. Compared to the single trees, I generally increased the max depth and decreased the min leaf size for my forest trees because I wanted a lower possible bias for each tree. The consequence of this is that each individual tree will have a larger variance, but that is hopefully offset by taking the average of multiple trees. For feature bagging I decided to bag $\sqrt{d}$ features at each node, where d is the total number of features. This decision was made based on Shewchuck's suggestion in lecture; he noted that this generally worked well in practice for decision tree classifiers.

Spam Report

1. I used a custom featurizer, which is just the one given for previous homeworks with additional words added (e.g. "viagra", "re :", etc.). I added 16 additional word features in total, bringing the number of featuers per sample up to 48. For a full list of additional words, see *featurize.py*.

2. When using a sole decision tree, I only trained on 2/3 of the samples and validated on the rest of the 1/3. At each node, I considered all 48 features, so no feature bagging. Depths were limited at 10 and leaf size was limited at 10 (to both prevent overfitting and lower runtimes). The validation accuracy was **0.8485**.

   For my random forests implemetation, I again trained on 2/3, 3448, of the samples and validated on the rest of the 1/3. Each decision tree was trained using a randomly chosen (with replacement)"bag" of 3448 samples (so the same as the number of samples in the training set). At each node, a random "bag" of 7 features was used, corresponding to roughly the square root of the total number of features (48). Depths were limited at 50 and leaf size was limited at 10. My forest consisted of 30 trees. With this setup, the validation set was classified correctly with a rate of **0.8741**.

3. For the Kaggle competiton, I decided to use the random forest impelementation with the same hyperparameters I defined above. My best Kaggle score (random forests) was **0.80247**. In the interest of time I merely ran the script a couple times to get an idea of what hyperaparameter values work. I probably could've probably bumped up my score if I had more rigorously tuned my parameters using cross-validation.

4. I chose a random sample that was classified as 0 (ham):

   i. "cc :" < 0.5
   ii. "!" < 0.5
   iii. "%" < 0.5
   iv. "meter" < 0.5

v. “(” < 0.5
vi. “www” < 0.5
vii. “sex” < 0.5
viii. “&” < 0.5
ix. “;” < 0.5
x. “volumes” > 0.5
xi. label $= 0$

5. For my random forest with 30 trees, the root split counts (“feature”, “value split on”) were as follows:

   i. (“!”, 0.5) - **4 trees**
   ii. (“meter”, 0.5), (“cc :”, 0.5) - **3 trees**
   iii. (“featured”, 0.5), (“&”, 0.5), (“re :”, 1.5), (“sex”, 0.5), (“money”, 0.5), (“cialis”, 0.5), (“microsoft”, 0.5), (“%”, 0.5) - **2 trees**
   iv. (“creative”, 0.5), (“www”, 0.5), (“viagra”, 0.5), (“volumes”, 0.5) - **1 tree**

Census Report

1. I used the features specified in the provided data files, where each sample point had 15 different features. I replaced missing feature values with the mode of the respective feature. Initially I tried to do this using the sklearn.preprocessing.Imputer, however I ran into issues so I instead opted to manually scan the data set (see *featurize_census.py*). I then used sklearn.feature_extraction.DictVectorizer to one-hot encode the categorical features. This resulted in a data matrix consisting of 105 features.

2. When using a sole decision tree, I only trained on 2/3 of the samples, 21816, and validated on the rest. At each node, all 105 features were taken into account. Depths were limited at 10 and leaf size was limited at 100 (to both prevent overfitting and lower runtimes). The validation accuracy was **0.8534**

   For my random forests implemetation, I split up the available data 50/50 into training data and validation data (so 16362 samples each). Each decision tree was trained using a randomly chosen “bag” of 16362 training samples, with replacement. At each node, a random “bag” of 10 features was used, corresponding to roughly the square root of the total number of features (105). Depths were limited at 50 and leaf size was limited at 10. My forest only consisted of 30 trees. With this setup, the validation set was classified correctly with a rate of **0.8657**.

   For the Kaggle competiton, my best Kaggle score (random forests) was **0.75686**. For comparison my Kaggle score using the sole decision tree was **0.6805**. In the interest of time I merely ran the script a couple times to get an idea of what hyperaparameter values work. I probably could’ve probably bumped up my score if I had more rigorously tuned my parameters using cross-validation.

3. The example data point we will classify is: {age=72,workclass=Private,fnlwgt=156310,education=10th,education-num=6,marital-status=Married-civ-spouse,occupation=Other-service,relationship=Husband,race=White,sex=Male, capital-gain=2414,capital-loss=0,hours-per-week=12,native-country=United-States,label=0}

   i. marital-status=Married-civ-spouse > 0.5
   ii. occupation=Prof-specialty < 0.5
   iii. education-num < 12.5
   iv. age > 30.5
   v. capital-gain < 5095.5
   vi. education=Some-college < 0.5
   vii. hours-per-week < 40.5
   viii. occupation=Adm-clerical < 0.5

ix. education=Assoc-acdm < 0.5

x. age > 66.5

xi. label $= 0$

4. For my random forest with 30 trees, the root split counts ("feature", "value split on") were as follows:

   i. (relationship=Husband, 0.5) **- 6 trees**

   ii. (age, 27.5), (education-num, 12.5), (marital-status=Married-civ-spouse, 0.5), (sex=Female, 0.5) **- 3 trees**

   iii. (capital-gain, > 5095.5), (marital-status=Never-married, 0.5), (relationship=own-child, 0.5), (relationship=Unmarried, 0.5) **- 2 trees**

   iv. (education=Bachelors, 0.5), (education=HS-grad, 0.5), (workclass=Private, 0.5) **- 1 tree**

Regarding My High Validation Accuracy But Relatively Low Kaggle Scores

For the census random forests, something interesting of note is that of the total 32724 provided samples, only 24.13% are labeled with a 1 ($> \$50,000$). Similarly in the validation subset 24.11% were labeled a 1, however our predicted labels were 19.23% labeled 1. The predicted labels for the test set were also 18.9% labeled 1. I believe that the skew in the data toward labels of 0 is mostly why my kaggle scores are relatively low compared to my validation accuracy. If there were a way to somehow factor in prior probabilities then maybe I could have addressed this issue.

However this doesn't explain the dsicrepancy between my single tree and random forest Kaggle scores for the census data. They had similar validation accuracies but widely differing Kaggle accuracies.

Appendix: featurize.py

```
'''
**************** PLEASE READ ***************

Script that reads in spam and ham messages and converts each training example
into a feature vector

Code intended for UC Berkeley course CS 189/289A: Machine Learning

Requirements:
-scipy ('pip install scipy')

To add your own features, create a function that takes in the raw text and
word frequency dictionary and outputs a int or float. Then add your feature
in the function 'def generate_feature_vector'

The output of your file will be a .mat file. The data will be accessible using
the following keys:
    -'training_data'
    -'training_labels'
    -'test_data'

Please direct any bugs to kevintee@berkeley.edu
'''

from collections import defaultdict
import glob
import re
import scipy.io

NUM_TRAINING_EXAMPLES = 5172
NUM_TEST_EXAMPLES = 5857

BASE_DIR = './'
SPAM_DIR = 'spam/'
HAM_DIR = 'ham/'
TEST_DIR = 'test/'

# ************* Features *************

# Features that look for certain words
def freq_pain_feature(text, freq):
    return float(freq['pain'])

def freq_private_feature(text, freq):
    return float(freq['private'])

def freq_bank_feature(text, freq):
    return float(freq['bank'])

def freq_money_feature(text, freq):
    return float(freq['money'])
```

```
def freq_drug_feature(text, freq):
    return float(freq['drug'])

def freq_spam_feature(text, freq):
    return float(freq['spam'])

def freq_prescription_feature(text, freq):
    return float(freq['prescription'])

def freq_creative_feature(text, freq):
    return float(freq['creative'])

def freq_height_feature(text, freq):
    return float(freq['height'])

def freq_featured_feature(text, freq):
    return float(freq['featured'])

def freq_differ_feature(text, freq):
    return float(freq['differ'])

def freq_width_feature(text, freq):
    return float(freq['width'])

def freq_other_feature(text, freq):
    return float(freq['other'])

def freq_energy_feature(text, freq):
    return float(freq['energy'])

def freq_business_feature(text, freq):
    return float(freq['business'])

def freq_message_feature(text, freq):
    return float(freq['message'])

def freq_volumes_feature(text, freq):
    return float(freq['volumes'])

def freq_revision_feature(text, freq):
    return float(freq['revision'])

def freq_path_feature(text, freq):
    return float(freq['path'])

def freq_meter_feature(text, freq):
    return float(freq['meter'])

def freq_memo_feature(text, freq):
    return float(freq['memo'])

def freq_planning_feature(text, freq):
    return float(freq['planning'])
```

```
def freq_pleased_feature(text, freq):
    return float(freq['pleased'])

def freq_record_feature(text, freq):
    return float(freq['record'])

def freq_out_feature(text, freq):
    return float(freq['out'])

# Features that look for certain characters
def freq_semicolon_feature(text, freq):
    return text.count(';')

def freq_dollar_feature(text, freq):
    return text.count('$')

def freq_sharp_feature(text, freq):
    return text.count('#')

def freq_exclamation_feature(text, freq):
    return text.count('!')

def freq_para_feature(text, freq):
    return text.count('(')

def freq_bracket_feature(text, freq):
    return text.count('[')

def freq_and_feature(text, freq):
    return text.count('&')

# ---------- Add your own feature methods ----------

def freq_buy_feature(text, freq):
    return text.count('buy')

def freq_cash_feature(text, freq):
    return text.count('cash')

def freq_freetrial_feature(text, freq):
    return text.count('free trial')

def freq_microsoft_feature(text, freq):
    return text.count('microsoft')

def freq_medication_feature(text, freq):
    return text.count('medication')

def freq_viagra_feature(text, freq):
    return text.count('viagra')

def freq_cialis_feature(text, freq):
    return text.count('cialis')
```

```
def freq_sex_feature(text, freq):
    return text.count('sex')

def freq_sexual_feature(text, freq):
    return text.count('sexual')

def freq_penis_feature(text, freq):
    return text.count('penis')

def freq_havebeenselected_feature(text, freq):
    return text.count('have␣been␣selected')

def freq_percent_feature(text, freq):
    return text.count('%')

def freq_www_feature(text, freq):
    return text.count('www')

def freq_100_feature(text, freq):
    return text.count('100')

def freq_re_feature(text, freq):
    return text.count('re␣:')

def freq_cc_feature(text, freq):
    return text.count('cc␣:')

# Generates a feature vector
def generate_feature_vector(text, freq):
    feature = []
    feature.append(freq_pain_feature(text, freq))      #0
    feature.append(freq_private_feature(text, freq))
    feature.append(freq_bank_feature(text, freq))
    feature.append(freq_money_feature(text, freq))
    feature.append(freq_drug_feature(text, freq))
    feature.append(freq_spam_feature(text, freq))     #5
    feature.append(freq_prescription_feature(text, freq))
    feature.append(freq_creative_feature(text, freq))
    feature.append(freq_height_feature(text, freq))
    feature.append(freq_featured_feature(text, freq))
    feature.append(freq_differ_feature(text, freq))  #10
    feature.append(freq_width_feature(text, freq))
    feature.append(freq_other_feature(text, freq))
    feature.append(freq_energy_feature(text, freq))
    feature.append(freq_business_feature(text, freq))
    feature.append(freq_message_feature(text, freq)) #15
    feature.append(freq_volumes_feature(text, freq))
    feature.append(freq_revision_feature(text, freq))
    feature.append(freq_path_feature(text, freq))
    feature.append(freq_meter_feature(text, freq))
    feature.append(freq_memo_feature(text, freq))     #20
    feature.append(freq_planning_feature(text, freq))
    feature.append(freq_pleased_feature(text, freq))
    feature.append(freq_record_feature(text, freq))
    feature.append(freq_out_feature(text, freq))
    feature.append(freq_semicolon_feature(text, freq)) #25
    feature.append(freq_dollar_feature(text, freq))
    feature.append(freq_sharp_feature(text, freq))
    feature.append(freq_exclamation_feature(text, freq))
    feature.append(freq_para_feature(text, freq))
    feature.append(freq_bracket_feature(text, freq)) #30
    feature.append(freq_and_feature(text, freq))

    # --------- Add your own features here ---------
    # Make sure type is int or float

    feature.append(freq_buy_feature(text, freq))
    feature.append(freq_cash_feature(text, freq))
    feature.append(freq_freetrial_feature(text, freq))
    feature.append(freq_microsoft_feature(text, freq))  #35
    feature.append(freq_medication_feature(text, freq))
    feature.append(freq_viagra_feature(text, freq))
    feature.append(freq_cialis_feature(text, freq))
    feature.append(freq_sex_feature(text, freq))
    feature.append(freq_sexual_feature(text, freq))     #40
    feature.append(freq_penis_feature(text, freq))
    feature.append(freq_havebeenselected_feature(text, freq))
    feature.append(freq_percent_feature(text, freq))
    feature.append(freq_www_feature(text, freq))
    feature.append(freq_100_feature(text, freq))       #45
    feature.append(freq_re_feature(text, freq))
    feature.append(freq_cc_feature(text, freq))

    return feature

# This method generates a design matrix with a list of filenames
# Each file is a single training example
def generate_design_matrix(filenames):
    design_matrix = []
    for filename in filenames:
        with open(filename) as f:
            text = f.read() # Read in text from file
            text = text.replace('\r\n', ' ') # Remove newline character
            words = re.findall(r'\w+', text)
            word_freq = defaultdict(int) # Frequency of all words
            for word in words:
                word_freq[word] += 1

            # Create a feature vector
            feature_vector = generate_feature_vector(text, word_freq)
            design_matrix.append(feature_vector)
    return design_matrix

# ************** Script starts here **************
# DO NOT MODIFY ANYTHING BELOW

spam_filenames = glob.glob(BASE_DIR + SPAM_DIR + '*.txt')
spam_design_matrix = generate_design_matrix(spam_filenames)
```

```
ham_filenames = glob.glob(BASE_DIR + HAM_DIR + '*.txt')
ham_design_matrix = generate_design_matrix(ham_filenames)
# Important: the test_filenames must be in numerical order as that is the
# order we will be evaluating your classifier
test_filenames = [BASE_DIR + TEST_DIR + str(x) + '.txt' for x in range(0, NUM_TEST_EXAMPLE
test_design_matrix = generate_design_matrix(test_filenames)

X = spam_design_matrix + ham_design_matrix
Y = [1]*len(spam_design_matrix) + [0]*len(ham_design_matrix)

file_dict = {}
file_dict['training_data'] = X
file_dict['training_labels'] = Y
file_dict['test_data'] = test_design_matrix
scipy.io.savemat('spam_data.mat', file_dict)
```

Appendix: featurize_census.py

```
import numpy as np
import csv
from sklearn.feature_extraction import DictVectorizer

''' Featurizing TRAIN Data '''
train_file = np.array(list(csv.DictReader(open("train_data.csv"))))

feat_keys = train_file[0].keys()
featval_counts = dict.fromkeys(feat_keys)
for feat in featval_counts:
    featval_counts[feat] = {}

for row in train_file:
    for feat in row:
        featval = row[feat]
        if featval.isdigit(): continue
        if featval in featval_counts[feat]:
            featval_counts[feat][featval] += 1
        else:
            featval_counts[feat][featval] = 1

featval_modes = dict.fromkeys(feat_keys)
for feat in featval_counts:
    maxcount = 0
    for val in featval_counts[feat]:
        if featval_counts[feat][val] > maxcount:
            featval_modes[feat] = val
            maxcount = featval_counts[feat][val]


for row in train_file:
    for item in row:
        if row[item].isdigit():
            row[item] = float(row[item])
        if row[item] == '?':
            row[item] = featval_modes[item]
```

```
train_vec = DictVectorizer()

traindv = train_vec.fit_transform(train_file)

train_data = traindv.toarray()

feat_names = train_vec.get_feature_names()
label_ind = feat_names.index('label')

train_labels = train_data[:, label_ind]
train_data = np.delete(train_data, label_ind, 1)


''' Featurizing TEST Data '''
test_file = np.array(list(csv.DictReader(open("test_data.csv"))))

feat_keys = test_file[0].keys()
featval_counts = dict.fromkeys(feat_keys)
for feat in featval_counts:
    featval_counts[feat] = {}

for row in test_file:
    for feat in row:
        featval = row[feat]
        if featval.isdigit(): continue
        if featval in featval_counts[feat]:
            featval_counts[feat][featval] += 1
        else:
            featval_counts[feat][featval] = 1

featval_modes = dict.fromkeys(feat_keys)
for feat in featval_counts:
    maxcount = 0
    for val in featval_counts[feat]:
        if featval_counts[feat][val] > maxcount:
            featval_modes[feat] = val
            maxcount = featval_counts[feat][val]


for row in test_file:
    for item in row:
        if row[item].isdigit():
            row[item] = float(row[item])
        if row[item] == '?':
            row[item] = featval_modes[item]

testdv = train_vec.transform(test_file)

test_data = testdv.toarray()
test_data = np.delete(test_data, label_ind, 1)
```

Appendix: DecisionTree.py

```
import numpy as np
import random

class Node:
    def __init__(self, f, left, right, label):
        self.split_rule = f
        self.left = left
        self.right = right
        self.label = label


class DecisionTree:
    def __init__(self, depth, m, n):
        self.d = depth
        self.m = m
        self.n = n
        self.root = None

    def segmenter(self, data, labels):
        numsamps = len(data)
        #numfeats = len(data[0])
        feats = range(0, len(data[0]))
        random.shuffle(feats)

        best = (None, None)
        best_entropy = float("inf")
        #for f in range(0, numfeats):
        for f in feats[:self.m]:
            all_feat = data[:, f]
            splits = find_splits(np.unique(all_feat))
            for s in splits:
                left_hist = np.zeros(2)
                right_hist = np.zeros(2)
                for i in range(0, numsamps):
                    if all_feat[i] < s:
                        left_hist[labels[i]] += 1
                    else:
                        right_hist[labels[i]] += 1

                curr_entropy = impurity(left_hist, right_hist)
                if (curr_entropy < best_entropy):
                    best_entropy = curr_entropy
                    best = (f, s)

        return best, best_entropy


    def growTree(self, data, labels, depth):
        numData = len(labels)
        numOnes = np.count_nonzero(labels)
        numZeros = numData - numOnes
        if(depth==0 or numOnes==0 or numZeros==0 or numData<self.n):
            if (numOnes > numZeros):
                return Node(None,None,None,1)
            else:
                return Node(None,None,None,0)
        else:
            split, entropy = self.segmenter(data,labels)
            #This occurs when all data have same features
            if (entropy == float("inf")):
                if (numOnes > numZeros):
                    return Node(None,None,None,1)
                else:
                    return Node(None,None,None,0)

            l = np.where(data[:, split[0]] < split[1])
            r = np.where(data[:, split[0]] >= split[1])
            return Node(split, \
                        self.growTree(data[l],labels[l],depth-1), \
                        self.growTree(data[r],labels[r],depth-1), \
                        None)


    def train(self, data, labels):
        split, entropy = self.segmenter(data, labels)
        l = np.where(data[:, split[0]] < split[1])
        r = np.where(data[:, split[0]] >= split[1])
        self.root = Node(split, \
                        self.growTree(data[l],labels[l],self.d), \
                        self.growTree(data[r],labels[r],self.d), \
                        None)

    def predict(self, data):
        p = np.zeros(len(data))
        for i in range(0, len(data)):
            currNode = self.root
            while (currNode.label == None):
                f = currNode.split_rule[0]
                val = currNode.split_rule[1]
                if (data[i][f] < val):
                    currNode = currNode.left
                else:
                    currNode = currNode.right
            p[i] = currNode.label
        return p


def entropy(hist, count):
    p = hist / count
    surprise = -np.multiply(p, np.log2(p + 10e-10))
    return np.sum(surprise)

def impurity(left_hist, right_hist):
    S_l = np.sum(left_hist)
    S_r = np.sum(right_hist)
    weighted_Hl = S_l * entropy(left_hist,S_l)
    weighted_Hr = S_r * entropy(right_hist, S_r)
    return (weighted_Hl + weighted_Hr) / (S_l + S_r)
```

```
def find_splits(dataset):
    splits = (dataset + np.roll(dataset, 1)) / 2.0
    return splits[1::max(len(splits)//4, 1)]
```

Appendix: spam_simple.py

```
import numpy as np
import DecisionTree as dt
from scipy.io import loadmat

def benchmark(pred_labels, true_labels):
    errors = pred_labels != true_labels
    err_rate = sum(errors) / float(len(true_labels))
    indices = errors.nonzero()
    return err_rate, indices

execfile('featurize.py')

#spam_data = loadmat(file_name="spam_data32.mat", mat_dtype=True)
spam_data = loadmat(file_name="spam_data.mat", mat_dtype=True)
train_data = np.array(spam_data['training_data'])
train_labels = np.transpose(np.array(spam_data['training_labels']))
train_labels = train_labels[:, 0]

#train_data = train_data[:3000]
#train_labels = train_labels[:3000]

random_state = np.random.get_state()
np.random.shuffle(train_data)
np.random.set_state(random_state)
np.random.shuffle(train_labels)

k = len(train_data) * 2 / 3
valid_data = train_data[k:]
valid_labels = train_labels[k:]
train_data = train_data[0:k]
train_labels = train_labels[0:k]

dectree = dt.DecisionTree(10, len(train_data[0]), 10)
dectree.train(train_data, train_labels)

pred_labels = dectree.predict(valid_data)
err, ind = benchmark(pred_labels, valid_labels)
```

Appendix: spam_forest.py

```
import numpy as np
import math
import DecisionTree as dt
from random import randint
from scipy.io import loadmat

def benchmark(pred_labels, true_labels):
    errors = pred_labels != true_labels
    err_rate = sum(errors) / float(len(true_labels))
    indices = errors.nonzero()
    return err_rate, indices

execfile('featurize.py')

#spam_data = loadmat(file_name="spam_data32.mat", mat_dtype=True)
spam_data = loadmat(file_name="spam_data.mat", mat_dtype=True)
train_data = np.array(spam_data['training_data'])
train_labels = np.transpose(np.array(spam_data['training_labels']))
train_labels = train_labels[:, 0]

train_data = train_data[:3000]
train_labels = train_labels[:3000]

random_state = np.random.get_state()
np.random.shuffle(train_data)
np.random.set_state(random_state)
np.random.shuffle(train_labels)

ntrain = len(train_data) * 2 / 3
dfeat = len(train_data[0])

valid_data = train_data[ntrain:]
valid_labels = train_labels[ntrain:]
train_data = train_data[0:ntrain]
train_labels = train_labels[0:ntrain]

print("Generating Random Forest...")
T = 30
tree_list = []
for t in range(0, T):
    print("T = " + str(t))
    rind = [randint(0, ntrain-1) for x in range(0, ntrain)]
    dectree = dt.DecisionTree(50, int(math.sqrt(dfeat))+1, 10)
    dectree.train(train_data[rind], train_labels[rind])
    tree_list.append(dectree)

print("Predicting...")
pred_labels = np.zeros(len(valid_labels))
for dt in tree_list:
    pred_labels += dt.predict(valid_data)

pred_labels = np.round(pred_labels / float(T))


err, ind = benchmark(pred_labels, valid_labels)
print("error rate: " + str(err))
```

Appendix: census_simple.py

```
import time
import numpy as np
import DecisionTree as dt

def benchmark(pred_labels, true_labels):
    errors = pred_labels != true_labels
    err_rate = sum(errors) / float(len(true_labels))
    indices = errors.nonzero()
    return err_rate, indices

print("Preprocessing...")
execfile('featurize_census.py')

random_state = np.random.get_state()
np.random.shuffle(train_data)
np.random.set_state(random_state)
np.random.shuffle(train_labels)

k = len(train_data) * 2 // 3
valid_data = train_data[k:]
valid_labels = train_labels[k:]
train_data_sub = train_data[0:k]
train_labels_sub = train_labels[0:k]

depth = 10
m = len(train_data[0])
n = 100
print("Building Decision Tree")
before = time.time()
dectree = dt.DecisionTree(depth, m, n)
dectree.train(train_data_sub, train_labels_sub)
after = time.time()
print("Training took " + str(after - before) + " seconds")

print("Predicting")
before = time.time()
pred_labels = dectree.predict(valid_data)
err, ind = benchmark(pred_labels, valid_labels)
after = time.time()
print("Predicting took " + str(after - before) + " seconds")
```

Appendix: census_forest.py

```
import time
import numpy as np
import math
import DecisionTree as dt
from random import randint
from scipy.io import loadmat

def benchmark(pred_labels, true_labels):
    errors = pred_labels != true_labels
    err_rate = sum(errors) / float(len(true_labels))
    indices = errors.nonzero()
    return err_rate, indices
```

```
print("Preprocessing...")
execfile('featurize_census.py')

random_state = np.random.get_state()
np.random.shuffle(train_data)
np.random.set_state(random_state)
np.random.shuffle(train_labels)

k = len(train_data)//2
valid_data = train_data[k:]
valid_labels = train_labels[k:]
train_data_sub = train_data[0:k]
train_labels_sub = train_labels[0:k]

numtrain = k
depth = 50
m = 10
n = 10

print("Generating Random Forest...")
T = 30
tree_list = []
for t in range(0, T):
    print("T = " + str(t))
    before = time.time()
    rind = [randint(0, k-1) for x in range(0, numtrain)]
    dectree = dt.DecisionTree(depth, m, n)
    dectree.train(train_data_sub[rind], train_labels_sub[rind])
    tree_list.append(dectree)
    after = time.time()
    print("Took " + str(after - before) + " seconds\n")

print("Predicting on Validation...")
pred_val_labels = np.zeros(len(valid_labels))
for dt in tree_list:
    pred_val_labels += dt.predict(valid_data)

pred_val_labels = np.round(pred_val_labels / float(T))


err, ind = benchmark(pred_val_labels, valid_labels)
print("error rate: " + str(err))

print("Predicting on Test...")
pred_test_labels = np.zeros(len(test_data))
for dt in tree_list:
    pred_test_labels += dt.predict(test_data)

pred_test_labels = np.round(pred_test_labels / float(T))

print("Writing to CSV")
with open('test_labels.csv', 'wb') as f:
    writer = csv.writer(f)
    writer.writerow(['Id'] + ['Category'])
```

```
for i in range(0, len(pred_test_labels)):
    writer.writerow([i+1] + [int(pred_test_labels[i])])
```