CS 189 –Spring 2016 — Homework 1 Solutions

Utsav Baral, SID 25694452

DUE: February 10th, 2016

Contents

Problem 3.

Code and report requirements for spam dataset:

I used the default bags of words given to us by the featurize.py. I performed 10 fold cross validation on various different techniques to obtain the following results:

Normal Decision Tree (No Pruning)
validation error rate is: 0.1718146718146718
validation error rate is: 0.19305019305019305
validation error rate is: 0.1760154738878143
validation error rate is: 0.15667311411992263
validation error rate is: 0.18375241779497098
validation error rate is: 0.1644100580270793
validation error rate is: 0.14313346228239845
validation error rate is: 0.1760154738878143
validation error rate is: 0.1528046421663443
validation error rate is: 0.15087040618955513

Bagging (w/o feature sampling)
validation error rate is: 0.1776061776061776
validation error rate is: 0.15444015444015444
validation error rate is: 0.1450676982591876
validation error rate is: 0.16247582205029013
validation error rate is: 0.16054158607350097
validation error rate is: 0.19148936170212766
validation error rate is: 0.16634429400386846
validation error rate is: 0.15473887814313347
validation error rate is: 0.15087040618955513
validation error rate is: 0.17988394584139264

Random Forest (w/ attribute and data bagging)
validation error rate is: 0.1718146718146718
validation error rate is: 0.19305019305019305
validation error rate is: 0.1760154738878143
validation error rate is: 0.15667311411992263
validation error rate is: 0.18375241779497098
validation error rate is: 0.1644100580270793
validation error rate is: 0.14313346228239845
validation error rate is: 0.1760154738878143
validation error rate is: 0.1528046421663443
validation error rate is: 0.15087040618955513

The best Kaggle score I received was using my random forest implementation with *depth_limit* set to float("inf"). (No limit) ***Kaggle Score: 0.72782***

Here is a sample point that got classified positive in the spam test. This was the path it took down the decision tree:

('drug') < 1.0

('width') < 1.0

('memo') < 1.0

('revision') < 1.0

('path') < 1.0

('volumes') < 1.0

('differ') < 1.0

('meter') < 1.0

('prescription') < 1.0

('pain') < 1.0

('spam') < 1.0

('planning') < 1.0

('featured') < 1.0

('creative') < 1.0

('&') < 1.0

('height') < 1.0

('energy') < 1.0

('money') < 1.0

('(') >= 1.0

(';') < 1.0

('#') < 1.0

('message') < 1.0

('(') >= 2.0

('private') < 1.0

('out') < 1.0

('[') < 2.0

('!') >= 1.0

('record') < 1.0

('bank') < 1.0

data point at leaf, and labeled: 1

And, here are the split counts for the random forest, (I used 10 trees in my forest):

('featured', 1.0): (2 trees)

('prescription', 1.0): (2 trees)

('meter', 1.0): (1 trees)

('drug', 1.0): (1 trees)

('pain', 1.0): (1 trees)

('memo', 1.0): (1 trees)

('creative', 1.0): (1 trees)

('volumes', 1.0): (1 trees)

Problem 4.

Code and report requirements for census dataset: Repeat all parts of 3). You will need to do your own feature processing. In part (a), you must report what you did to handle categorical variables and missing features. See the Appendix for detailed instructions.

Data preprocessing for the census data was much trickier than the previous problem because I had to extract the data manually from the CSV file, and fill in the missing data points. I used a differnt CSV library, called pandas, which provided easier handling of columns and allowed me to easily fill in the missing feature point with the mode of all the other sample feature points. After doing this, I used the DictVectorizer module to then binarize the categorical features in order to be able to run it through my Decision Tree, which works for only continuous, not categorical, features.

Here are the cross validation results, Running ten-fold validation proved difficult since the total amount of binarized features was 105, and the dataset was so large, in fact I only used about 1/6 of the dataset in total for the cross-validation, otherwise it was too slow:

Normal Decision Tree:
validation error rate is: 0.2275
validation error rate is: 0.2225
validation error rate is: 0.2725
Random Forest, with 5 trees:
validation error rate is: 0.2
validation error rate is: 0.23
validation error rate is: 0.19
validation error rate is: 0.16
validation error rate is: 0.17
validation error rate is: 0.19
validation error rate is: 0.28
validation error rate is: 0.24
validation error rate is: 0.27
validation error rate is: 0.24

Kaggle Score: 0.63

Couldn't submit anymore, but I forgot to train on a bigger sample size so I could not get a better score :(

Here is a sample point that got classified positive in the spam test. This was the path it took down the decision tree:

('23') $<$ 1.0

('71') $<$ 1.0

('32') $<$ 1.0

('33') $<$ 1.0

('36') $<$ 1.0

('48') $<$ 1.0

('49') $<$ 1.0

('52') < 1.0

('99') < 1.0

('104') < 1.0

('17') < 1.0

('29') < 1.0

('34') < 1.0

('50') < 1.0

('63') < 1.0

('68') < 1.0

('78') < 1.0

('7') < 1.0

('31') < 1.0

('61') < 1.0

('87') < 1.0

('6') < 1.0

('8') < 1.0

('10') < 1.0

('54') < 1.0

('91') < 1.0

('92') < 1.0

('18') >= 1.0

data point at leaf, and labeled: 1

Here are split counts for the random forest on the census dataset:

'native-country=Italy', 1.0): # of Trees = 2,

'education=1st-4th', 1.0): # of Trees = 1,

'education=5th-6th', 1.0): # of Trees = 1,

'marital-status=Married-AF-spouse', 1.0): # of Trees = 1,

'marital-status=Never-married', 1.0): # of Trees = 1,

'native-country=Cuba', 1.0): # of Trees = 1,

'native-country=Dominican-Republic', 1.0): # of Trees = 1,

'native-country=Ecuador', 1.0): # of Trees = 1,

'native-country=El-Salvador', 1.0): # of Trees = 1,

'native-country=India', 1.0): # of Trees = 1,

'native-country=Iran', 1.0): # of Trees = 1,

'native-country=Jamaica', 1.0): # of Trees = 1,

'native-country=Japan', 1.0): # of Trees = 1,

'native-country=Nicaragua', 1.0): # of Trees = 1,

'native-country=Taiwan', 1.0): # of Trees = 1, 'native-country=Vietnam', 1.0): # of Trees = 1,

'occupation=Armed-Forces', 1.0): # of Trees = 1,

'relationship=Other-relative', 1.0): # of Trees = 1,

'relationship=Own-child', 1.0): # of Trees = 1

Problem 5.

Give an explanation of the decision tree techniques you implemented (stopping criteria, pruning, dealing with missing attributes, splitting criteria other than entropy, heuristics for faster training, complex decisions at nodes, cross-validation, Adaboost, bagging etc.).

(a) On top of the simple entropy heuristic, I also added additional stopping criterea on the tree. Such as a depth limit, stopping when 95% of the nodes are the same, and also when there are less then some number of samples, in order to prevent overfitting
(b) For the random forest, I implemented attribute bagging as well as data bagging in order to more efficiently process data and also increase variance. Also no limit to depth on forests in order to have better combined averages of trees.

APPENDIX A: CODE LISTINGS

DecisionTree.py

```
import numpy as np
import math
import random
from collections import defaultdict


class DecisionTree:
    # Node data structure for the Tree Structure
    class Node:
        def __init__(self, left, right, lab, spl_rul):
            self.left_child = left
            self.right_child = right
            self.label = lab
            # Tuple of the split rule (feature index, feature value threshold), None implies
                this Node is a leaf
            self.split_rule = spl_rul

        def is_leaf(self):
            return (self.left_child is None) and (self.right_child is None)

    # Constructor
    def __init__(self, depth=float("inf")):
        self.root = None
        self.depth_lim = depth

    # Takes in the result of a split, and outputs the weighted average entropy of the given split
    # Could use different impurity measure, but standard entropy seems to work well... i think.
    @staticmethod
    def impurity(left_label_hist, right_label_hist):
        def entropy(label_hist):
            avg_surprise = 0
            set_size = sum([label_hist[key] for key in label_hist])
            for label in label_hist:
                if label_hist[label] > 0:
                    prob = label_hist[label] / set_size
                    surprise = -math.log(prob, 2)
                    avg_surprise += prob * surprise
            return avg_surprise

        left_size = len(left_label_hist)
        right_size = len(right_label_hist)
        weighted_avg_entropy = left_size * entropy(left_label_hist) + right_size *
            entropy(right_label_hist)
        weighted_avg_entropy /= (left_size + right_size)
        return weighted_avg_entropy

    @staticmethod
    def extract_subsets(data, labels, sample_index):
        subset_data = np.matrix(np.empty((0, np.size(data, 1))))
        subset_labels = np.matrix(np.empty((0, 1)))
        for i in sample_index:
            subset_labels = np.append(subset_labels, labels[i], 0)
            subset_data = np.append(subset_data, data[i], 0)
        return subset_data, subset_labels

    # This function figures out the split rule for a node using the impurity measure and input
        data
    # Only works for  continuous-valued featured data, we cannot impose such an ordering on
        categorical variables
    @staticmethod
    def segmenter(data, labels, feature_index_set):
        # We will loop through all the possible splits, and then choose the split that maximizes
            the information gain,
        # H(curren+t S) - H_after, so minimize the H_after. Since H(current_s) will remain
            constant during this call
        # Find the impurity for each split and store a tuple of smallest split and impurity.

        curr_min = (float("inf"), None, None)
        for feature in feature_index_set:
            # For a given feature, loop through possible thresholds(i.e values it can take on),
                which is the domain of
            # this feature
            domain = dict()
            for i in range(np.size(data, 0)):
                feat_val_of_samp = data[i, feature]
                lab_of_samp = labels[i, 0]
                if feat_val_of_samp not in domain.keys():
                    domain[feat_val_of_samp] = defaultdict(int)
                domain[feat_val_of_samp][lab_of_samp] += 1

            # The keys to the domain -> hist map are the possible thresholds, so radix sort, and
                scan through list
            # in order, and we can update entropy in O(1) time since we will not have to loop
            sorted_feat_vals = list(domain.keys())
            sorted(sorted_feat_vals)
            num_poss_splits = len(sorted_feat_vals) - 1
            if num_poss_splits == 0:
                # There is only one unique value for given feature among all samples, so a split
                    would be pointless,
                # so we will just move onto the next feature instead
                continue

            # Set up the initial left and right histograms, so left is on the first value, and
                right is on everything to
            # the right of this sorted value, we can update entropy in constant time by scanning
                through the sorted list
            left_hist = domain[sorted_feat_vals[0]]
            right_hist = domain[sorted_feat_vals[1]]
            if num_poss_splits > 1:
                for k in range(2, len(sorted_feat_vals)):
                    for poss_class in domain[sorted_feat_vals[k]]:
                        right_hist[poss_class] += domain[sorted_feat_vals[k]][poss_class]

            for i in range(1, len(sorted_feat_vals)):
                # Left hist and right hist are preconfigured for the start of loop, so we can
                    check the impurity
                if sum([left_hist[key] for key in left_hist]) == 0 or sum([right_hist[key] for
                    key in left_hist]) == 0:
                    break
                impurity_val = DecisionTree.impurity(left_hist, right_hist)
                if impurity_val < curr_min[0]:
                    curr_min = (impurity_val, feature, sorted_feat_vals[i])
                # Set up the left and right hist for the next run of the loop:
                for class_poss in domain[sorted_feat_vals[i]]:
                    left_hist[class_poss] += domain[sorted_feat_vals[i]][class_poss]  # add to
                        the left
                    right_hist[class_poss] -= domain[sorted_feat_vals[i]][class_poss]  #
                        subtract from the right
        # Return tuple of feature index and threshold that we deemed to have the minimum impurity
        return curr_min[1], curr_min[2]

    # Actually grow the decision tree and add nodes based on the labeled training data
    # m_feature_selection paramater is for attribute bagging when doing random forests
    def train(self, data, labels, m_feature_selection=0):
        self.root = None  # reset the root to null

        def grow_tree(sample_index, d):
            # Different Stopping conditions for better performance
            if d == 0:
                sub = self.extract_subsets(data, labels, sample_index)
                return DecisionTree.Node(None, None, np.argmax(np.histogram(sub[1], bins=[0, .5,
                    1])[0]), None)
            if len(sample_index) <= 10:
                sub = self.extract_subsets(data, labels, sample_index)
                return DecisionTree.Node(None, None, np.argmax(np.histogram(sub[1], bins=[0, .5,
                    1])[0]), None)
            if len(sample_index) == 1:
                return DecisionTree.Node(None, None, labels[sample_index[0], 0], None)

            # check if all labels are the same for the given sample_index list
            identical = True
            for i in sample_index:
                if labels[i, 0] != labels[sample_index[0], 0]:
                    identical = False
                    break
            if identical:
                # if all the labels are the same in this (sub)set then we return a leaf node.
                return DecisionTree.Node(None, None, labels[sample_index[0], 0], None)
            else:
                # we need to split samples according to some rule recursively
                data_subset, label_sub = self.extract_subsets(data, labels, sample_index)
                if (np.histogram(label_sub, bins=[0, .5, 1])[0][0] / len(sample_index) > .95) or
                    (np.histogram(label_sub, bins=[0, .5, 1])[1][0] / len(sample_index) > .95):
                    return DecisionTree.Node(None, None, np.argmax(np.histogram(label_sub,
                        bins=[0, .5, 1])[0]), None)
                feature_set = [feat for feat in range(np.size(data_subset, 1))]
                if m_feature_selection > 0:
                    feature_set = random.sample(feature_set, m_feature_selection)
                split = self.segmenter(data_subset, label_sub, feature_set)
                feature_to_split = split[0]
                threshold = split[1]
                if feature_to_split is None or threshold is None:
                    # No good split exists that lessens entropy so we will make this node a leaf
                        and
                    # for the label, we will simply take the class with majority among
                        (sub)samples
                    return DecisionTree.Node(None, None, np.argmax(np.histogram(label_sub,
                        bins=[0, .5, 1])[0]), None)

                left_samps = list()
                right_samps = list()
                for i in sample_index:
                    if data[i, feature_to_split] < threshold:
                        left_samps.append(i)
                    else:
                        right_samps.append(i)

                if len(left_samps) == 0:
                    right_sub = self.extract_subsets(data, labels, right_samps)
                    return DecisionTree.Node(None, None, np.argmax(np.histogram(right_sub[1],
                        bins=[0, .5, 1])[0]),
                                            None)
                if len(right_samps) == 0:
                    left_sub = self.extract_subsets(data, labels, left_samps)
                    return DecisionTree.Node(None, None, np.argmax(np.histogram(left_sub[1],
                        bins=[0, .5, 1])[0]), None)

                return DecisionTree.Node(grow_tree(left_samps, d - 1), grow_tree(right_samps, d
                    - 1), None, split)

        self.root = grow_tree([i for i in range(np.size(data, 0))], self.depth_lim)

    # Traverses the tree rooted at node to find the best label, which is the label at the leaf
        node we end  up at
    @staticmethod
    def trickle_down(node, sample_vector):
        if node.is_leaf():
            return node.label
        else:
            if sample_vector[0, node.split_rule[0]] < node.split_rule[1]:
                return DecisionTree.trickle_down(node.left_child, sample_vector)
            else:
                return DecisionTree.trickle_down(node.right_child, sample_vector)

    def predict(self, data):
        output = np.matrix(np.empty((0, 1)))
        for sample_row in data:
            output = np.append(output, np.matrix(self.trickle_down(self.root, sample_row)), 0)
        return output
```

RandomForest.py

```
import DecisionTree as dt
import numpy as np
import random


class RandomForest:
    def __init__(self, num_trees, m, n_prime, depth=float("inf")):
        self.decision_forest = list()
        self.num_trees = num_trees
        self.m = m
        self.n_prime = n_prime
        self.depth = depth

    def train(self, data, labels):
        self.decision_forest = list()
        index_set = [k for k in range(np.size(data, 0))]
        for i in range(self.num_trees):
            bagged_sample_list = list()
            # create another tree to put in our forest
            for k in range(self.n_prime):
                # Loop n_prime times and generate another sample_index
                bagged_sample_list.append(random.choice(index_set))
            dt_instance = dt.DecisionTree(self.depth)
            data_sub, labels_sub = dt_instance.extract_subsets(data, labels, bagged_sample_list)
            dt_instance.train(data_sub, labels_sub, self.m)
            self.decision_forest.append(dt_instance)

    def predict(self, data):
        output = np.matrix(np.empty((np.size(data, 0), len(self.decision_forest))))
        for j in range(len(self.decision_forest)):
            output[:, j] = np.matrix(self.decision_forest[j].predict(data))[:, 0]

        y_hat = np.matrix(np.empty((np.size(data, 0), 1)))
        for i in range(np.size(output, 0)):
            y_hat[i, 0] = np.argmax(np.histogram(output[i, :], bins=[0, .5, 1])[0])
        return y_hat
```

SpamClassifier.py

```
import numpy as np
import scipy.io.matlab
import DecisionTree as dt
import RandomForest as rf
import random
import math

def k_cross_validation(k, N, X, y, classifier):
    indexSet = set([i for i in range(N)])

    partitionDict = dict()
    for i in range(k):
        partitionDict[i] = [np.matrix(np.empty((0, np.size(X, 1)), X[0, 0].dtype)),
                            np.matrix(np.empty((0, 1), y[0, 0].dtype))]
    for _ in range(N):
        partitionKey = _ % k
        i = random.sample(indexSet, 1)[0]
        indexSet.remove(i)
        partitionDict[partitionKey][0] = np.append(partitionDict[partitionKey][0], X[i], axis=0)
        partitionDict[partitionKey][1] = np.append(partitionDict[partitionKey][1], y[i], axis=0)

    for key in partitionDict:
        validationPartition = partitionDict[key]
        validation_X = validationPartition[0]
        validation_y = validationPartition[1]

        train_X = np.matrix(np.empty((0, np.size(X, 1)), X[0, 0].dtype))
        train_y = np.matrix(np.empty((0, 1), y[0, 0].dtype))

        for otherKey in partitionDict:
            if otherKey != key:
                trainingPartition = partitionDict[otherKey]
                train_X = np.append(train_X, trainingPartition[0], axis=0)
                train_y = np.append(train_y, trainingPartition[1], axis=0)

        classifier.train(train_X, train_y)
        y_hat = np.matrix(classifier.predict(validation_X), dtype=validation_y.dtype)
        print("validation error rate is: {0}".format(
            (np.sum(np.bitwise_xor(y_hat, validation_y)) / np.size(validation_y, 0))))


def decision_tree_classification(X, y, test_dat):
    classifier = dt.DecisionTree()
    classifier.train(X, y)
    y_hat = classifier.predict(test_dat)

    f = open("spam_predictions_decision_tree.csv", 'w')
    f.write("Id,Category\n")
    for i in range(np.size(test_dat, 0)):
        f.write(str(i + 1) + "," + str(int(y_hat[i, 0])) + "\n")
    f.close()
    print("DONE")


def random_forests_classification(X, y, test_dat):
    classifier = rf.RandomForest(5, np.size(X, 1), np.size(X, 0))
    classifier.train(X, y)
    y_hat = classifier.predict(test_dat)
    f = open("spam_predictions_random_forest.csv", 'w')
    f.write("Id,Category\n")
    for i in range(np.size(test_dat, 0)):
        f.write(str(i + 1) + "," + str(int(y_hat[i, 0])) + "\n")
    f.close()
    print("DONE")
```

```
mat = scipy.io.loadmat("spam-dataset/spam_data.mat")
training_data = np.matrix(mat["training_data"])
training_labels = np.matrix(mat["training_labels"]).getT()
test_data = np.matrix(mat["test_data"])

# k_cross_validation(10, np.size(training_data, 0), training_data,
    training_labels,rf.RandomForest(15,
# int(round(math.sqrt(np.size(training_data, 1)))),int(np.size(training_data, 0))))

# k_cross_validation(10, np.size(training_data, 0), training_data, training_labels,
    dt.DecisionTree())

# decision_tree_classification(training_data, training_labels, test_data)
# random_forests_classification(training_data, training_labels, test_data)
```

CensusClassifier.py

```
import numpy as np
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import Imputer
from DecisionTree import DecisionTree
from RandomForest import RandomForest
from sklearn.feature_extraction import DictVectorizer
import random
import pandas as pd
import math


def k_cross_validation(k, N, X, y, classifier):
    indexSet = set([i for i in range(N)])

    partitionDict = dict()
    for i in range(k):
        partitionDict[i] = [np.matrix(np.empty((0, np.size(X, 1)), X[0, 0].dtype)),
                            np.matrix(np.empty((0, 1), y[0, 0].dtype))]
    for _ in range(N):
        partitionKey = _ % k
        i = random.sample(indexSet, 1)[0]
        indexSet.remove(i)
        partitionDict[partitionKey][0] = np.append(partitionDict[partitionKey][0], X[i], axis=0)
        partitionDict[partitionKey][1] = np.append(partitionDict[partitionKey][1], y[i], axis=0)

    for key in partitionDict:
        validationPartition = partitionDict[key]
        validation_X = validationPartition[0]
        validation_y = validationPartition[1]

        train_X = np.matrix(np.empty((0, np.size(X, 1)), X[0, 0].dtype))
        train_y = np.matrix(np.empty((0, 1), y[0, 0].dtype))

        for otherKey in partitionDict:
            if otherKey != key:
                trainingPartition = partitionDict[otherKey]
                train_X = np.append(train_X, trainingPartition[0], axis=0)
                train_y = np.append(train_y, trainingPartition[1], axis=0)

        print("actually train the tree")
        classifier.train(train_X, train_y)
        y_hat = np.matrix(classifier.predict(validation_X), dtype=validation_y.dtype)
        print("validation error rate is: {0}".format(
            (np.sum(np.bitwise_xor(y_hat, validation_y)) / np.size(validation_y, 0))))


def decision_tree_classification(X, y, test_dat):
    classifier = DecisionTree()
    classifier.train(X, y)
    y_hat = classifier.predict(test_dat)

    f = open("spam_predictions_decision_tree.csv", 'w')
    f.write("Id,Category\n")
    for i in range(np.size(test_dat, 0)):
        f.write(str(i + 1) + "," + str(int(y_hat[i, 0])) + "\n")
    f.close()
    print("DONE")


def random_forests_classification(X, y, test_dat):
    classifier = RandomForest(10, round(math.sqrt(np.size(X, 1))), np.size(X, 0), 45)
    # classifier = RandomForest(1, round(math.sqrt(np.size(X, 1))), 100, 45)
    classifier.train(X, y)
    y_hat = classifier.predict(test_dat)
    f = open("census_predictions_random_forest.csv", 'w')
    f.write("Id,Category\n")
    for i in range(np.size(test_dat, 0)):
        f.write(str(i + 1) + "," + str(int(y_hat[i, 0])) + "\n")
    f.close()
    print("DONE")


def parseCSV(df):
    data = np.matrix(np.empty((df.last_valid_index() + 1, 0)))
    keylist = []
    labels = []
    for key in df:
        keylist.append(key)
        if key == "label":
            labels = np.matrix(df[key]).getT()

        elif "?" in np.array(df[key]):
            le = LabelEncoder()
            le.fit(np.array(df[key]))
            imp = Imputer(missing_values=le.transform("?"), strategy='most_frequent', axis=1)
            tform = le.transform(np.array(df[key]))
            imp.fit_transform(tform)
            toapp = le.inverse_transform(np.array(imp.fit_transform(tform), dtype=int))
            data = np.append(data, np.matrix(toapp).getT(), axis=1)
        else:
            data = np.append(data, np.matrix(df[key]).getT(), axis=1)

    return data, labels, keylist


panda_df = pd.read_csv('census_data/train_data.csv')
dataset, train_labels, keylistt = parseCSV(panda_df)

panda_df2 = pd.read_csv('census_data/test_data.csv')
test_dataset, test_labels, keylist_test = parseCSV(panda_df2)

dict_row_list = []
for row in dataset:
    dict_row_list.append(dict(zip(keylistt, row.tolist()[0])))
v = DictVectorizer(sparse=False)
train_data = np.matrix(v.fit_transform(dict_row_list))

dict_row_list = []
for row in test_dataset:
    dict_row_list.append(dict(zip(keylist_test, row.tolist()[0])))
v = DictVectorizer(sparse=False)
test_data = np.matrix(v.fit_transform(dict_row_list))

# N = np.size(train_data, 0)
# k_cross_validation(10, N, train_data, labels, RandomForest(15,
    round(math.sqrt(np.size(train_data, 1))), N, 45))
random_forests_classification(train_data, train_labels, test_data)
```