COMPSCI 289A Homework 5 **Mitch Negus**
3/26/2017 **3032146443**

Problem 1

Python module: **decisiontree.py**

```
"""
decisiontree.py
==============================================
Train a decision tree and predict on test data
==============================================

This python module contains the decision tree class, which can be trained on
labeled data (input as numpy arrays: for data, an Nxd matrix with N rows
corresponding to N sample points and d columns corresponding to d features; for
labels, an N vector with labels corresponding to each of the N sample points)
"""

import numpy as np


class DecisionTree:
    """
    Build and store a decision tree, based on supplied training data.
    Use this tree to predict classifications.
    - treedepth: an integer for the max depth of the tree
    - verbose:   a boolean for descriptive output
    """

    def __init__(self,treedepth=10,verbose=False,params=None):
        self.depth = treedepth
        self.verbose = verbose
        self.tree = self.Node()
        if type(treedepth) is not int:
            print('ERROR: Tree depth must be an integer.')


    def entropy(self,C,D,c,d):
        """
        Calculate entropy based on classifications above and below the
        splitrule.
        - C: sample points in-class below splitrule (left)
        - D: sample points not-in-class below splitrule (left)
        - c: sample points in-class above splitrule (right)
        - d: sample points not-in-class above splitrule (right)
        Returns the entropy.
        """

        if C != 0:
            Cfactor = -(C/(C+D))*np.log2(C/(C+D))
        else:
            Cfactor = 0
        if D != 0:
            Dfactor = -(D/(C+D))*np.log2(D/(C+D))
        else:
            Dfactor = 0
        if c != 0:
            cfactor = -(c/(c+d))*np.log2(c/(c+d))
        else:
            cfactor = 0
        if d != 0:
            dfactor = -(d/(c+d))*np.log2(d/(c+d))
        else:
            dfactor = 0
        H_left = Cfactor + Dfactor
        H_right = cfactor + dfactor
        H = ((C+D)*H_left + (c+d)*H_right)/(C+D+c+d)

        return H


    def segment(self,data,labels):
        """
        March through data and determine split which maximizes info gain.
```

```
        Returns the ideal splitrule as a length-2 list where the first element
        is the index of the splitting feature and the second element is the
        value of that feature to split on.
        """

        totals = np.bincount(labels)
        if len(totals)==1:
            totals = np.append(totals,[0])
        # Quick safety check
        if len(labels) != len(data):
            print('ERROR (DecisionTree.segment): There must be the same number of labels as datapoints.')

        # Calculate the initial entropy, used to find info gain
        C,D = 0,0                     # C = in class left of split; D = not in class left of split
        c,d = totals[1],totals[0]     # c = in class right of split; d = not in class right of split
        H_i = self.entropy(C,D,c,d) # the initial entropy, before any splitting

        # Initialize objects to store optimal split rules for iterative comparison
        maxinfogain = 0
        splitrule = []

        for feature_i in range(len(data[0])):
            # Order the data for determining ideal splits
            lbldat = np.concatenate(([data[:,feature_i]],[labels]),axis=0)
            fv = np.sort(lbldat.T,axis=0)
            lastfeature = np.array(['',''])

            C,D = 0,0                     # Reset the counters
            c,d = totals[1],totals[0]

            for point_i in range(len(fv)-1):

                # Update C,D,c,d to minmize runtime of entrop calc (keep at O(1) time)
                if fv[point_i,1] == 1:
                    C += 1
                    c -= 1
                elif fv[point_i,1] == 0:
                    D += 1
                    d -= 1
                else:
                    print("ERROR (DecisionTree.segment): Classifications can only be 0 or 1.")

                # Skip splitting values that are not separable
                if fv[point_i,0] == fv[point_i+1,0]:
                    continue
                else:
                    H_f = self.entropy(C,D,c,d)
                    infogain = H_i-H_f
                    if infogain > maxinfogain:
                        maxinfogain = infogain
                        splitrule = [feature_i,fv[point_i,0]]

        return splitrule


    def train(self,data,labels,node=1,deep=0):
        """
        Train the decision tree on input data
        - data:   Nxd numppy array with N sample points and d features
        - labels: 1D, length-N numpy array with labels for the N sample points
        - node:   node class passed to function; default is 1, a flag for the head node (INTERNAL USE ONLY)
        - deep:   a counter to determine current depth in tree (INTERNAL USE ONLY)
        """

        # Ensure labels are integers
        labels = labels.astype(int)

        # On the first training cycle, set the current node to the head node
        if node==1:
            node=self.tree

        # Grow decision tree
        depthlim = self.depth
        if deep < depthlim:
            splitrule = self.segment(data,labels)
        else:
            splitrule = []
        if self.verbose is True:
            print(data,labels)
        node.isleaf(data,labels,splitrule)

        # Train deeper if the node splits
        if node.nodetype == 'SplitNode':
            if self.verbose is True:
```

```
                print('rule:',node.rule)
                print('Splitting node left and right')
            deep += 1
            node.left=self.Node()
            node.right=self.Node()
            self.train(node.leftdata,node.leftlabels,node.left,deep)
            self.train(node.rightdata,node.rightlabels,node.right,deep)
        elif node.nodetype == 'LeafNode':
            if self.verbose is True:
                print('You made a leaf node! It has value',node.leaflabel,'and',node.leafcount,'items.')
        else:
            print('ERROR (DecisionTree.train): The node type could not be identified!')


    def predict(self,testdata):
        """
        Predict classfications for unlabeled data points using the previously
        trained decision tree.
        - testdata: Nxd numpy array with N sample points and d features
                    *Note, dimensions N and d must match those used
                    for data array in DecisionTree.train*
        Returns a 1D, length-N numpy array of predictions (one prediction per point)
        """

        npoints = len(testdata)
        predictions = np.empty(npoints)
        for point_i in range(npoints):

            # Print out decisions for a point
            if point_i<10:
                if self.verbose == 'path10':
                    print('Display of choices for point %i' %point_i)

            ParentNode = self.tree
            Rule = ParentNode.rule
            while Rule is not None:
                splitfeat_i = Rule[0]
                splitval = Rule[1]
                if testdata[point_i,splitfeat_i] <= splitval:
                    ChildNode = ParentNode.left
                    if point_i<10:
                        if self.verbose == 'path10':
                            print('Feature #'+str(splitfeat_i+1)+': ',testdata[point_i,splitfeat_i],'<=',splitval)

                else:
                    if point_i<10:
                        if self.verbose == 'path10':
                            print('Feature #'+str(splitfeat_i+1)+': ',testdata[point_i,splitfeat_i],'>',splitval)
                    ChildNode = ParentNode.right
                ParentNode = ChildNode
                Rule = ParentNode.rule
            predictions[point_i]=ParentNode.leaflabel
            if point_i<10:
                if self.verbose == 'path10':
                    print('Point labeled as',ParentNode.leaflabel)
        return predictions.astype(int)



    class Node:
        """
        Store a decision tree node, coupled in series to construct tree;
        includes a left branch, right branch, and splitrule
        """

        def __init__(self):
            self.rule = None
            self.left = None
            self.leftdata = None
            self.leftlabels = None
            self.right = None
            self.rightdata = None
            self.rightlabels = None
            self.leaflabel = None
            self.leafcount = None
            self.nodetype = None


        def isleaf(self,data,labels,splitrule):
            """Determine if this is a leaf node"""
            if splitrule:
                indsabove = self.datainds_above_split(data,splitrule)
                self.rule = splitrule
                self.leftdata,self.leftlabels = self.leftDL(data,labels,indsabove)
```

```
            self.rightdata,self.rightlabels = self.rightDL(data,labels,indsabove)
            self.nodetype = 'SplitNode'

        elif not splitrule:
            self.leaflabel = np.bincount(labels).argmax()
            self.leafcount = len(labels)
            self.nodetype = 'LeafNode'


    def datainds_above_split(self,data,splitrule):
        """
        Collect indices of points with values of the splitting feature
        greater than the split rule
        """

        indsabove = []
        fv = data[:,splitrule[0]]
        for point_i in range(len(fv)):
            if fv[point_i] > splitrule[1]:
                indsabove.append(point_i)

        return indsabove


    def leftDL(self,data,labels,indsabove):
        """Return arrays of only left data and labels"""

        leftdata = np.delete(data,indsabove,axis=0)
        leftlabels = np.delete(labels,indsabove,axis=0)

        return leftdata,leftlabels


    def rightDL(self,data,labels,indsabove):
        """Return arrays of only right data and labels"""

        rightdata = data[indsabove]
        rightlabels = labels[indsabove]

        return rightdata,rightlabels
```

Problem 2

Python module: **randomforest.py**

```
"""
randomforest.py
==============================================
Train a random forest and predict on test data
==============================================

This python module contains the random forest class, which can be trained on
labeled data (input as numpy arrays: for data, an Nxd matrix with N rows
corresponding to N sample points and d columns corresponding to d features; for
labels, an N vector with labels corresponding to each of the N sample points)
"""

import numpy as np


class RandomDecisionTree:
    """
    Build and store a random decision tree, based on supplied training data.
    Use this tree to predict classifications.
    - treedepth: an integer for the max depth of the tree
    - mfeatures: an integer number of random features tested for splits per node
    - verbose:   a boolean for descriptive output
    """

    def __init__(self,treedepth=10,mfeatures=None,verbose=False):
        self.depth = treedepth
        self.mfeatures = mfeatures
        self.nfeatures = None
        self.verbose = verbose
        self.tree = self.Node()
        if type(treedepth) is not int:
            print('ERROR (RandomDecisionTree): Tree depth must be an integer.')
        if mfeatures and type(mfeatures) is not int:
            print('ERROR (RandomDecisionTree): The number of random features must be an integer.')


    def entropy(self,C,D,c,d):
        """
        Calculate entropy based on classifications above and below the
        splitrule.
        - C: sample points in-class below splitrule (left)
        - D: sample points not-in-class below splitrule (left)
        - c: sample points in-class above splitrule (right)
        - d: sample points not-in-class above splitrule (right)
        Returns the entropy.
        """

        if C != 0:
            Cfactor = -(C/(C+D))*np.log2(C/(C+D))
        else:
            Cfactor = 0
        if D != 0:
            Dfactor = -(D/(C+D))*np.log2(D/(C+D))
        else:
            Dfactor = 0
        if c != 0:
            cfactor = -(c/(c+d))*np.log2(c/(c+d))
        else:
            cfactor = 0
        if d != 0:
            dfactor = -(d/(c+d))*np.log2(d/(c+d))
        else:
            dfactor = 0
        H_left = Cfactor + Dfactor
        H_right = cfactor + dfactor
        H = ((C+D)*H_left + (c+d)*H_right)/(C+D+c+d)

        return H


    def pick_random_features(self):
        """Randomly choose a set of m features out of n total features"""

        mrandomfeatures = -1*np.ones(self.mfeatures)
        for i in range(self.mfeatures):
            while mrandomfeatures[i] == -1:
                feature_i = np.random.randint(self.nfeatures)
                if feature_i not in mrandomfeatures:
```

```
                mrandomfeatures[i] = feature_i
        mrandomfeatures = np.sort(mrandomfeatures).astype(int)

        return mrandomfeatures


    def segment(self,data,labels):
        """
        March through data and determine split which maximizes info gain.
        Returns the ideal splitrule as a length-2 list where the first element
        is the index of the splitting feature and the second element is the
        value of that feature to split on.
        """

        totals = np.bincount(labels)
        if len(totals)==1:
            totals = np.append(totals,[0])
        # Quick safety check
        if len(labels) != len(data):
            print('ERROR␣(RandomForest.segment):␣There␣must␣be␣the␣same␣number␣of␣labels␣as␣datapoints.')

        # Calculate the initial entropy, used to find info gain
        C,D = 0,0                       # C = in class left of split; D = not in class left of split
        c,d = totals[1],totals[0]       # c = in class right of split; d = not in class right of split
        H_i = self.entropy(C,D,c,d) # the initial entropy, before any splitting

        # Initialize objects to store optimal split rules for iterative comparison
        maxinfogain = 0
        splitrule = []

        mrandomfeatures = self.pick_random_features()
        for feature_i in mrandomfeatures:
            # Order the data for determining ideal splits
            lbldat = np.concatenate(([data[:,feature_i]],[labels]),axis=0)

            fv = np.sort(lbldat.T,axis=0)
            lastfeature = np.array(['',''])

            C,D = 0,0                       # Reset the counters
            c,d = totals[1],totals[0]

            for point_i in range(len(fv)-1):

                # Update C,D,c,d to minmize runtime of entrop calc (keep at O(1) time)
                if fv[point_i,1] == 1:
                    C += 1
                    c -= 1
                elif fv[point_i,1] == 0:
                    D += 1
                    d -= 1
                else:
                    print("ERROR␣(RandomForest.segment):␣Classifications␣can␣only␣be␣0␣or␣1.")

                # Skip splitting values that are not separable
                if fv[point_i,0] == fv[point_i+1,0]:
                    continue
                else:
                    H_f = self.entropy(C,D,c,d)
                    infogain = H_i-H_f
                    if infogain > maxinfogain:
                        maxinfogain = infogain
                        splitrule = [feature_i,fv[point_i,0]]

        return splitrule


    def train(self,data,labels,node=1,deep=0):
        """
        Train the random decision tree on input data
        - data:   Nxd numppy array with N sample points and d features
        - labels: 1D, length-N numpy array with labels for the N sample points
        - node:   node class passed to function; default is 1, a flag for the head node (INTERNAL USE ONLY)
        - deep:   a counter to determine current depth in tree (INTERNAL USE ONLY)
        """

        # Ensure labels are integers
        labels = labels.astype(int)

        # On the first training cycle, set the current node to the head node
        # If the number of random features has not yet been set, set that too.
        if node==1:
            node=self.tree
            if self.mfeatures is None:
                self.mfeatures = np.int(np.round((np.sqrt(len(data[0])))))  # m random features
```

```
            self.nfeatures = len(data[0])                    # n total features
            if self.mfeatures > self.nfeatures:
                print('WARNING:␣The␣number␣of␣random␣features␣to␣choose␣is␣greater␣than␣the␣total␣number␣of␣features.␣Using␣a
                self.mfeatures = self.nfeatures
        # Grow decision tree
        depthlim = self.depth
        if deep < depthlim:
            splitrule = self.segment(data,labels)
        else:
            splitrule = []
        if self.verbose is True:
            print(data,labels)
        node.isleaf(data,labels,splitrule)

        # Train deeper if the node splits
        if node.nodetype == 'SplitNode':
            if self.verbose is True:
                print('rule:',node.rule)
                print('Splitting␣node␣left␣and␣right')
            deep += 1
            node.left=self.Node()
            node.right=self.Node()
            self.train(node.leftdata,node.leftlabels,node.left,deep)
            self.train(node.rightdata,node.rightlabels,node.right,deep)
        elif node.nodetype == 'LeafNode':
            if self.verbose is True:
                print('You␣made␣a␣leaf␣node!␣It␣has␣value',node.leaflabel,'and',node.leafcount,'items.')
        else:
            print('ERROR␣(RandomForest.train):␣The␣node␣type␣could␣not␣be␣identified!')


    def predict(self,testdata):
        """
        Predict classfications for unlabeled data points using the previously
        trained random decision tree.
        - testdata: Nxd numpy array with N sample points and d features
                    *Note, dimensions N and d must match those used
                    for data array in DecisionTree.train*
        Returns a 1D, length-N numpy array of predictions (one prediction per point)
        """

        npoints = len(testdata)
        predictions = np.empty(npoints)
        for point_i in range(npoints):
            ParentNode = self.tree
            Rule = ParentNode.rule
            while Rule is not None:
                splitfeat_i = Rule[0]
                splitval = Rule[1]
                if testdata[point_i,splitfeat_i] <= splitval:
                    ChildNode = ParentNode.left
                else:
                    ChildNode = ParentNode.right
                ParentNode = ChildNode
                Rule = ParentNode.rule
            predictions[point_i]=ParentNode.leaflabel

        return predictions.astype(int)



    class Node:
        """
        Store a decision tree node, coupled in series to construct tree;
        includes a left branch, right branch, and splitrule
        """

        def __init__(self):
            self.rule = None
            self.left = None
            self.leftdata = None
            self.leftlabels = None
            self.right = None
            self.rightdata = None
            self.rightlabels = None
            self.leaflabel = None
            self.leafcount = None
            self.nodetype = None


        def isleaf(self,data,labels,splitrule):
            """Determine if this is a leaf node"""

            if splitrule:
```

```
            indsabove = self.datainds_above_split(data,splitrule)
            self.rule = splitrule
            self.leftdata,self.leftlabels = self.leftDL(data,labels,indsabove)
            self.rightdata,self.rightlabels = self.rightDL(data,labels,indsabove)
            self.nodetype = 'SplitNode'

        elif not splitrule:
            self.leaflabel = np.bincount(labels).argmax()
            self.leafcount = len(labels)
            self.nodetype = 'LeafNode'


    def datainds_above_split(self,data,splitrule):
        """
        Collect indices of points with values of the splitting feature
        greater than the split rule
        """

        indsabove = []
        fv = data[:,splitrule[0]]
        for point_i in range(len(fv)):
            if fv[point_i] > splitrule[1]:
                indsabove.append(point_i)

        return indsabove


    def leftDL(self,data,labels,indsabove):
        """Return arrays of only left data and labels"""

        leftdata = np.delete(data,indsabove,axis=0)
        leftlabels = np.delete(labels,indsabove,axis=0)

        return leftdata,leftlabels


    def rightDL(self,data,labels,indsabove):
        """Return arrays of only right data and labels"""

        rightdata = data[indsabove]
        rightlabels = labels[indsabove]

        return rightdata,rightlabels


class RandomForest:
    """
    Build and store a random forest, based on supplied training data.
    Use this tree to predict classifications.
    - treedepth: an integer for the max depth of any tree in the forest
    - mfeatures: an integer number of random features tested for splits per node
    - verbose:   a boolean for descriptive output
    """

    def __init__(self,treedepth=10,ntrees=None,mfeatures=None,subsize=None,verbose=False):
        self.treedepth = treedepth
        self.mfeatures = mfeatures
        self.subsize = subsize
        self.verbose = verbose
        self.treecount = ntrees
        self.forest = []
        if type(treedepth) is not int:
            print('ERROR (RandomForest): Tree depth must be an integer.')
        if mfeatures and type(mfeatures) is not int:
            print('ERROR (RandomForest): The number of random features must be an integer.')


    def train(self,data,labels):
        """
        Train (grow) the random forest on input data
        - data:   Nxd numppy array with N sample points and d features
        - labels: 1D, length-N numpy array with labels for the N sample points
        """
        if self.subsize is None:
            self.subsize = len(data)
        if self.treecount is None:
            self.treecount = int(np.sqrt(len(data)))
        elif type(self.treecount) is not int:
            print('ERROR (RandomForest): The number of trees must be an integer.')

        for tree_i in range(self.treecount):
            # choose a random subset of the data, size "subsize", for BAGGING
            subsetindices = np.random.randint(0,self.subsize,self.subsize)
            baggeddata = data[subsetindices]
```

```
            baggedlabels = labels[subsetindices]
            tree = RandomDecisionTree(self.treedepth,self.mfeatures,self.verbose)
            tree.train(baggeddata,baggedlabels)
            self.forest.append(tree)
            if tree_i%5 == 0:
                print('Finished training %i tree(s) out of %i' %(tree_i,self.treecount))

    def predict(self,testdata):
        """
        Predict classfications for unlabeled data points using the previously
        trained random forest.
        - testdata: Nxd numpy array with N sample points and d features
                    *Note, dimensions N and d must match those used
                    for data array in DecisionTree.train*
        Returns a 1D, length-N numpy array of predictions (one prediction per point)
        """

        aggregatedpredictions = np.empty((self.treecount,len(testdata)))
        for tree_i in range(self.treecount):
            treepredictions = self.forest[tree_i].predict(testdata)
            aggregatedpredictions[tree_i]=treepredictions
        forestpredictions = np.round(np.average(aggregatedpredictions,axis=0)).astype(int)

        return forestpredictions
```

Problem 3

a.)

For the Titanic data set, both cabin and ticket number were removed because they were either sparse or not in a common format, so would be difficult (and essentially meaningless) to vectorize.

For the census data set, we removed the final-weight (fnlwgt) category. According to the README, this parameter *only* indicates similarity of demographics for a given state. Without knowing the location of the people in the census data, we cannot be sure that this parameter is valuable.

Other missing values were imputed. Since the vast majority of missing data points in both census and Titanic datasets seemed to be categorical, they were replaced by the mode of their respective feature. Taking the mean of binary vectorized features would not make sense as it would always tend to give a value of zero unless there were either only two classes (so the mean, rounded to the nearest integer 0 or 1, would be the mode). Similarly, it is impossible to take the mean of discrete categories before vectorization.

For the full preprocessing method, see the Jupyter notebook for preprocessing below.

b.)

The stopping criteria (and formation of a leaf node) occurred when either (1) no entropy gain was found after trying every feature over every possible split, or (2) the branch of the tree reached a maximum user-specified depth.

c.)

I did not include any special features to speed up training, other than the common sense approach to entropy evaluation over incremental spits, keeping the runtime at $O(1)$. The code does provide the user with the capacity to adjust all hyperparameters–tree depth for decision trees; tree depth, quantity of random sample points for bagging, random forest feature count, and number of random forest trees–and reducing any of these quantities will achieve a faster run time, though perhaps at a cost of accuracy.

d.)

I implemented random forests by modifying my decision tree class. I created a random forest class which generated a list of "random tree" classes. "Random trees" were decision trees that allowed for a random subsample of features to be used in generating the tree. Furthermore, "random trees" were trained on a bagged (random set, with replacement) set of data by the random forest class.

e.)

Nothing else was implemented.

Problem 4

PERFORMANCE EVALUATION

Spam

Decision Tree
Training Accuracy: 84.9709
Validation Accuracy: 85.2743

Random Forest
Training Accuracy: 82.3833
Validation Accuracy: 81.9409

Kaggle: mnegus 0.79760

Census

Decision Tree
Training Accuracy: 81.5496
Validation Accuracy: 81.9377

Random Forest
Training Accuracy: 85.9942
Validation Accuracy: 84.7188

Kaggle: mnegus 0.76498

Titanic

Decision Tree
Training Accuracy: 72.7778
Validation Accuracy: 71.0000

Random Forest
Training Accuracy: 88.0000
Validation Accuracy: 87.0000

Kaggle: mnegus 0.83226

Problem 5

a.)

No additional packages/features/feature transformations were used.

b.)

Below is the path through the decision tree taken by one of the data points classified as ham:
(”[” = 11) >s 0
(”(” = 1) > 0
(”[” = 11) > 1
(”(” = 1) $\leq$ 1
(”[” = 11) > 4
(”#” = 1) $\leq$ 0
(”[” = 11) > 6
(”[” = 11) $\leq$ 11
(”energy” = 0) $\leq$ 0
(”$” = 10) > 0
(”bank” = 0) $\leq$ 0
(”featured” = 0) $\leq$ 0
Point correctly labeled as 0 (ham)

Below is the path through the decision tree taken by one of the data points classified as spam:
(”[” = 3) > 0
(”(” = 0) $\leq$ 0
(”message” = 2) > 0
(”&” = 0) $\leq$ 0
(”#” = 1) > 0
(”[” = 3) > 2
(”[” = 3) $\leq$ 3
(”#” = 1) $\leq$ 1
(”drug” = 5) > 0
Point correctly abeled as 1 (spam)

Problem 6

a.)

No additional packages/features/feature transformations were used.

b.)

Below is the path through the decision tree tken by one of the data points classified as making over $50,000:
Display of choices for point 0 (Education Number = 9.0) $\leq$ 12
(Hours/Wk = 40.0) $\leq$ 49
(Age = 33) $\leq$ 55
(Occupation = Exec/Manag = 0) $\leq$ 0
(Race = Black = 0) $\leq$ 0
(Age = 33) $\leq$ 47
(Relationship = Unmarried = 1) > 0
(Capital Gains = 0) $\leq$ 3325
Point correctly labeled as 0 (<$50,000)

Below is the path through the decision tree taken by one of the data points classified as making over $50,000:
(Education Number = 13) > 12
(Age = 58) > 40
(Age = 58) > 46
(Age = 58) > 51
(Age = 58) 58 > 56
(Age = 58) 58 $\leq$ 61
(Relationship = Husband = 1) > 0
(Age = 58) > 57
Point correctly labeled as 1 (>$50,000)

Other Code

Python module: **HW05_utils.py**

```
#HW05_utils.py
#-----------------------------------------
# Python module for CS289A HW05
#-----------------------------------------
#-----------------------------------------


import numpy as np
from scipy import io as spio


def load_data(datapath,BASE_DIR,dictkey):
#Load data
    data_dict = spio.loadmat(BASE_DIR+datapath)
    data = data_dict[dictkey]

    return data


def shuffle_data(data,labels):
    datlbl = np.concatenate((data,labels),axis=1)
    np.random.shuffle(datlbl)
    shuffleddata = datlbl[:,:-1]
    shuffledlabels = datlbl[:,-1]

    return shuffleddata,shuffledlabels

def val_partition(data,valfrac):
```

```
    # Separate <valsetsize> items for validation
    valsetsize = int(valfrac*len(data))
    valset = data[:valsetsize]
    trainset = data[valsetsize:]

    return trainset,valset

def val_accuracy(predictions,truelabels):
    count,total = 0,0
    for i in range(len(predictions)):
        if predictions[i] == truelabels[i]:
            count += 1
        total += 1
    valAcc = count/total

    return valAcc
```