In [1]:
```
import scipy
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn import svm
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.neighbors import KNeighborsClassifier
%config InlineBackend.figure_format = 'retina'
```

In [2]:
```
def simple_GridSearchCV_fit(X_train_val, Y_train_val, n_list, fold):
    """
    A simple grid search function for k with cross-validation in k-NN.

    X_train_val: Features for train and val set.
                 Shape: (num of data points, num of features)
    Y_train_val: Labels for train and val set.
                 Shape: (num of data points,)
    n_list:      The list of k values to try.
    fold:        The number of folds to do the cross-validation.

    Return the val and train accuracy matrix of cross-validation.
    """
    val_acc_array = np.zeros(len(n_list))
    train_acc_array = np.zeros(len(n_list))
    for i in range(len(n_list)):
        val_acc_array[i], train_acc_array[i] = simple_cross_validation(X
_train_val, Y_train_val, n_list[i], fold)
    return val_acc_array, train_acc_array
```

In [3]:
```
def simple_cross_validation(X, Y, n, fold):
    """
    A simple cross-validation function for k-NN.

    X_train_val: Features for train and val set.
                 Shape: (num of data points, num of features)
    Y_train_val: Labels for train and val set.
                 Shape: (num of data points,)
    n:           Parameter k for k-NN.
    fold:        The number of folds to do the cross-validation.

    Return the average accuracy on validation set.
    """

    val_acc_list = []
    train_acc_list = []

    topends = 0
    bottomstarts = 0
    val_acc = 0
    train_acc = 0
    for i in range(fold):

        NumPerFold = int(X.shape[0]/fold)
        bottomstarts = NumPerFold*(i+1)
        topends = i * NumPerFold
        Xtrain = np.vstack((X[:topends], X[bottomstarts:]))
        Ytrain = np.vstack((Y[:topends].reshape(-1,1), Y[bottomstarts:].
reshape(-1,1))).reshape(-1)
        Xval = X[topends:bottomstarts]
        Yval = Y[topends:bottomstarts]
        numYval = len(Yval)

        classifier = bubble_KNeighborsClassifier(n=n)
        classifier.fit(Xtrain, Ytrain)
        YvalPred = classifier.predict(Xval)
        YtrainPred = classifier.predict(Xtrain)


        correctval = 0
        for i in range(len(Yval)):
            if Yval[i] == YvalPred[i]:
                correctval = correctval+1
        val_acc = float(correctval)/len(Yval)
        val_acc_list.append(val_acc)

        correcttrain = 0
        for i in range(len(Ytrain)):
            if Ytrain[i] == YtrainPred[i] :
                correcttrain = correcttrain+1
        train_acc = float(correcttrain)/len(Ytrain)
        train_acc_list.append(train_acc)

    return sum(val_acc_list) / len(val_acc_list), \
           sum(train_acc_list) / len(train_acc_list)
```

In [4]:
```
def getNpercentSamples(X, percent):
    n = int(len(X)*percent)
    return n
```

In [5]:
```
def splitSets(X_and_Y, n):
    np.random.shuffle(X_and_Y)
    X = X_and_Y[:, :-1]       # First column to second last column: Feat
ures (numerical values)
    Y = X_and_Y[:, -1:]       # Last column: Labels (0 or 1)
    X_train_val = X[:n, :] # Get features from train + val set.
    X_test      = X[n:, :] # Get features from test set.
    Y_train_val = Y[:n, :].reshape(-1) # Get labels from train + val se
t.
    Y_test      = Y[n:, :].reshape(-1) # Get labels from test set.

    return X_train_val, X_test, Y_train_val, Y_test

#print(X_train_val.shape, X_test.shape, Y_train_val.shape, Y_test.shape)
```

In [6]:
```
def treeClassifier(X_train_val, Y_train_val):
    classifier = DecisionTreeClassifier(criterion="entropy")
    D_list     = [1, 2, 3, 4, 5] # Different D to try.
    param = {'max_depth': D_list}
    clf = GridSearchCV(classifier, param, cv=5)
    return clf.fit(X_train_val, Y_train_val)
```

In [7]:
```
def knnClassifier(X_train_val, Y_train_val):
    classifier = KNeighborsClassifier(algorithm='brute')
    K = [1, 2, 3, 4, 5]
    params = {'n_neighbors': K}

    clf = GridSearchCV(classifier, params, cv=5)
    return clf.fit(X_train_val, Y_train_val)
```

In [8]:
```
def bubble_knnClassifier(X_train_val, Y_train_val):
    classifier = bubble_KNeighborsClassifier()
    nsamples = [5, 6, 7, 8, 9, 10]
    n = [3,4,5,6,7]
    multiplier = [0.125 ,0.25, 0.5, 0.75, 1, 1.5, 1.75, 2]
    param = {'nsamples': nsamples, 'n': n, 'multiplier': multiplier}

    clf = simple_GridSearchCV_fit(X_train_val, Y_train_val, n, fold=5)
    return clf
```

In [9]:
```
def draw_heatmap_knn(acc, acc_desc, n_list):
    plt.figure(figsize = (2,4))
    ax = sns.heatmap(acc, annot=True, fmt='.3f', yticklabels=n_list, xti
cklabels=[])
    ax.collections[0].colorbar.set_label("accuracy")
    ax.set(ylabel='$n$')
    plt.title(acc_desc + ' w.r.t $n$')
    sns.set_style("whitegrid", {'axes.grid' : False})
    plt.show()
```

In [10]:
```
def simple_knnClassifier(X_train_val, Y_train_val):

    classifier = simple_KNeighborsClassifier(k=3)
    K = [1, 2, 3, 4, 5]
    param = {'k': K}

    clf = GridSearchCV(classifier, param, cv=5, scoring='accuracy' )
    return clf.fit(X_train_val, Y_train_val)
```

In [11]:
```
def draw_heatmap_knn(acc, acc_desc, k_list):
    plt.figure(figsize = (2,4))
    ax = sns.heatmap(acc, annot=True, fmt='.3f', yticklabels=k_list, xti
cklabels=[])
    ax.collections[0].colorbar.set_label("accuracy")
    ax.set(ylabel='$n$')
    plt.title(acc_desc + ' w.r.t $n$')
    sns.set_style("whitegrid", {'axes.grid' : False})
    plt.show()
```

In [12]:
```
def svmClassifier(X_train_val, Y_train_val):
    classifier = svm.SVC()
    gamma_list = [1e-7, 1e-6, 1e-5, 1e-4]
    C_list = [0.00001, 0.0001, 0.001, 0.01, 0.1, 1.0] # Different C to t
ry.
    param = {'kernel':['linear','rbf'], 'C': C_list, 'gamma': gamma_list
}
    clf = GridSearchCV(classifier, param, cv=5)
    return clf.fit(X_train_val, Y_train_val)
```

In [13]:
```
carX_and_Y = pd.read_table('auto-mpg.txt', header=None, delim_whitespace
=True)      # Load data from file.
carX_and_Y.rename(columns={8: 'car'}, inplace=True)
carX_and_Y['origin'] = None
carX_and_Y = carX_and_Y[[0, 1, 2, 3, 4, 5, 6, 7, 'car', 'origin']]
carX_and_Y = carX_and_Y[carX_and_Y[3] != '?']

for i, row in carX_and_Y.iterrows():
    car_val = 0
    if row['car'].startswith("chevrolet") or row['car'].startswith("buic
k") or row['car'].startswith("plymouth") or row['car'].startswith("amc")
 or row['car'].startswith("ford") or row['car'].startswith("pontiac") or
 row['car'].startswith("dodge") or row['car'].startswith("chevy") or row
['car'].startswith("mercury") or row['car'].startswith("chrysler") or ro
w['car'].startswith("oldsmobile") or row['car'].startswith("cadillac"):
        car_val = 1
        carX_and_Y.set_value(i,'origin', car_val)
    else:
        carX_and_Y.set_value(i, 'origin', car_val)


carX_and_Y.drop(['car'], axis=1, inplace=True)
carX_and_Y = carX_and_Y.astype('float64')


carX_and_Y = carX_and_Y.as_matrix()
np.random.seed(0)
np.random.shuffle(carX_and_Y)   # Shuffle the data.
```

In [14]:
```
letterX_and_Y = pd.read_table('letter-recognition.txt', header=None, sep
=',')      # Load data from file.
letterX_and_Y.rename(columns={0: 'letter'}, inplace=True)
letterX_and_Y['group'] = None
letterX_and_Y = letterX_and_Y[[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13
, 14, 15, 16, 'letter', 'group']]
letterX_and_Y = letterX_and_Y[:500]

for i, row in letterX_and_Y.iterrows():
    letter_val = 0
    if row['letter'] <= 'M':
        letter_val = 1
        letterX_and_Y.set_value(i,'group', letter_val)
    else:
        letterX_and_Y.set_value(i, 'group', letter_val)


letterX_and_Y.drop(['letter'], axis=1, inplace=True)
letterX_and_Y = letterX_and_Y.astype('float64')

letterX_and_Y = letterX_and_Y.as_matrix()
np.random.seed(0)
np.random.shuffle(letterX_and_Y)   # Shuffle the data.
```

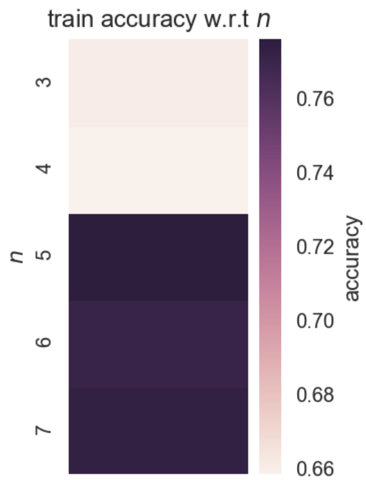
In [15]:
```
coverX_and_Y = pd.read_table('covtype.txt', header=None, sep=',')      #
 Load data from file.
coverX_and_Y.drop([11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 2
4, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34,
                   35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 4
8, 49, 50, 51, 52, 53], axis=1, inplace=True)
coverX_and_Y.rename(columns={54: 'tree'}, inplace=True)
coverX_and_Y['group'] = None
coverX_and_Y = coverX_and_Y[[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 'tree',
'group']].astype(float)
coverX_and_Y.round(2)
coverX_and_Y = coverX_and_Y[:500]

for i, row in coverX_and_Y.iterrows():
    tree_val = 0
    if row['tree'] < 4:
        tree_val = 1
        coverX_and_Y.set_value(i,'group', tree_val)
    else:
        coverX_and_Y.set_value(i, 'group', tree_val)


coverX_and_Y.drop(['tree'], axis=1, inplace=True)
coverX_and_Y = coverX_and_Y.reset_index().values
np.random.seed(0)
np.random.shuffle(coverX_and_Y)   # Shuffle the data.
```

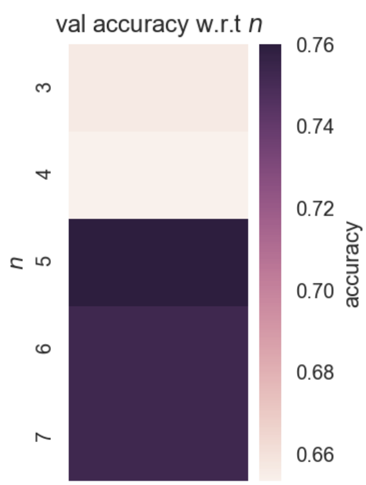
In [16]:
```
class bubble_KNeighborsClassifier(object):
    def __init__(self, n=5, nsamples=10, multiplier=0.5):
        """
        k-NN initialization.
            k: Number of nearest neighbors.
        """

        self.n = n
        self.nsamples = nsamples
        self.multiplier = multiplier

    def fit(self, X_train, Y_train):
        """
        k-NN fitting function.
            X_train: Feature vectors in training set.
            Y_train: Labels in training set.
        """
        self.X_train = X_train
        self.Y_train = Y_train

    def predict(self, X_pred):
        """
        k-NN prediction function.
        X_pred: Feature vectors in training set.
        Return the predicted labels for X_pred. Shape: (len(X_pred), ).

        """
        """Randomly choose n samples of the same class and calculate the
 average
```

```
            distance between them to determine the size of the bubble aro
und a
            sample to predict

        """
        increment=0
        sumdist = 0
        avgDist = 0
        nPoints = []

        zero = np.zeros((1,1))
        for i in range(len(self.Y_train)):
            if self.Y_train[i] == zero:
                nPoints.append(self.X_train[i])
                increment+=1
            if increment == self.nsamples:
                break
        for sampl in nPoints:
            for point in nPoints:
                #if np.array_equal(point,sampl)==False:
                distance = np.linalg.norm(np.array(point) - np.array(sam
pl))
                    #print(distance)
                sumdist += distance
        avgDist = sumdist / (len(nPoints)*len(nPoints))

        Y_pred = []

        maximum = 100
        iterations = 0


        for sample in X_pred:
            count = 0
            bubbleGroupindex = []

            for index, instance in enumerate(self.X_train, start=0):
                if np.array_equal(instance,sample)==False:
                    features = 0
                    for i in range(len(instance)):

                        if np.greater(instance[i],(sample[i]-self.multip
lier*avgDist)) and np.less(instance[i],(sample[i]+self.multiplier*avgDis
t)):
                            features+=1
                        else:
                            break
                        if features == len(instance):
                            bubbleGroupindex.append(index)


                            count+=1
                        if count==self.n:
                            break
                    iterations+=1
                if count==self.n or iterations == maximum:
                    break
```

```
        bubbleindices = np.asarray(bubbleGroupindex)

        label0 = 0
        label1 = 0
        pred = 0

        for index in bubbleindices:
            if self.Y_train[index]==0:
                label0 +=1
            else:
                label1 +=1
        if label1 >= label0:
            pred = 1
        else:
            pred = 0

        Y_pred.append(pred)
    return np.array(Y_pred)
```

In [17]:
```
X_train_val, X_test, Y_train_val, Y_test = splitSets(carX_and_Y, 300)
n = [3,4,5,6,7]
val_acc_array, train_acc_array = bubble_knnClassifier(X_train_val, Y_tra
in_val)
draw_heatmap_knn(train_acc_array.reshape(-1,1), 'train accuracy', n)
draw_heatmap_knn(val_acc_array.reshape(-1,1), 'val accuracy', n)
```





In [18]:
```
# 3) Implement the k-NN.
class simple_KNeighborsClassifier(object):
    def __init__(self, k=5):
        """
        k-NN initialization.
            k: Number of nearest neighbors.
        """
        self.k = k

    def fit(self, X_train, Y_train):
        """
        k-NN fitting function.
            X_train: Feature vectors in training set.
            Y_train: Labels in training set.
        """
        self.X_train = X_train
        self.Y_train = Y_train

    def predict(self, X_pred):
        """
        k-NN prediction function.
            X_pred: Feature vectors in training set.
        Return the predicted labels for X_pred. Shape: (len(X_pred), )
        """
        Y_pred = []
        for i in range(len(X_pred)):
            distances = ((self.X_train - X_pred[i].reshape(1,-1)) \
                          ** 2.0).sum(axis = 1)
            distances_and_labels = [(distances[i], self.Y_train[i]) \
                                    for i in range(len(self.X_train))]
            distances_and_labels.sort()
            top_k_labels = np.array(distances_and_labels)[:self.k,1].rav
el()
            mode, _ = scipy.stats.mode(top_k_labels)
            Y_pred.append(mode[0])
        return np.array(Y_pred)
```

In [19]:
```
def bubbleACC(X_train_val, Y_train_val, X_test, Y_test,n):
    test_acc_list = []

    nsampl = 10
    multipl = .25

    classifier = bubble_KNeighborsClassifier(n=n,nsamples=nsampl,multipl
ier=multipl)
    classifier.fit(X_train_val, Y_train_val)
    YtestPred = classifier.predict(X_test)
    YtrainPred = classifier.predict(X_train_val)


    traincorrectval = 0
    for i in range(len(Y_train_val)):
        if YtrainPred[i] == Y_train_val[i]:
            traincorrectval +=1

    testcorrectval = 0
    for i in range(len(Y_test)):
        if YtestPred[i] == Y_test[i]:
            testcorrectval +=1


    testacc = float(testcorrectval) / len(Y_test)
    trainacc = float(traincorrectval) / len(Y_train_val)

    return trainacc, testacc
```

In [20]:
```
np.random.shuffle(carX_and_Y)
X_train_val, X_test, Y_train_val, Y_test = splitSets(carX_and_Y,300)

test_acc_list = []

n=3
nsamples = 10
multiplier = .25


classifier = simple_KNeighborsClassifier(k=n)
classifier.fit(X_train_val, Y_train_val)
YtestPred = classifier.predict(X_test)

correctval = 0
for i in range(len(Y_test)):
    if YtestPred[i] == Y_test[i]:
        correctval +=1

accuracy = float(correctval) / len(Y_test)
print(accuracy)
```

```
0.8369565217391305
```

In [21]:
```
#1. load and clean carX_and_Y
#   load and clean letterX_and_Y
```

```
#   load and clean coverX_and_Y
trainResults = np.zeros((36,4))
testResults = np.zeros((36,4))

SPLITS = np.array([0.2,0.5,0.8])
DATASETS = [carX_and_Y, letterX_and_Y, coverX_and_Y]

scorerow = 0
avgrow = 27
for splitindex, split in enumerate(SPLITS, start=0):
    for datasetindex, dataset in enumerate(DATASETS, start=0):
        splitIndex = getNpercentSamples(dataset, split)

        treeAccSumtest = 0
        knnAccSumtest = 0
        svmAccSumtest = 0
        treeAccSumtrain = 0
        knnAccSumtrain = 0
        svmAccSumtrain = 0
        bubbleAccSumtest = 0
        bubbleAccSumtrain = 0

        for i in range(3):
            X_train_val, X_test, Y_train_val, Y_test = splitSets(dataset
, splitIndex)
            #Decision Tree
            treeCLF = treeClassifier(X_train_val, Y_train_val)
            bestTree = DecisionTreeClassifier(criterion="entropy", max_d
epth=(treeCLF.best_params_['max_depth'])).fit(X_train_val, Y_train_val)
            treeAccSumtest +=  bestTree.score(X_test, Y_test)
            treeAccSumtrain += bestTree.score(X_train_val, Y_train_val)
            testResults[scorerow,0] = bestTree.score(X_test, Y_test)
            trainResults[scorerow,0] = bestTree.score(X_train_val, Y_tra
in_val)

            #k_NN
            knnCLF = knnClassifier(X_train_val, Y_train_val)
            bestKNN = KNeighborsClassifier(n_neighbors = knnCLF.best_par
ams_["n_neighbors"]).fit(X_train_val, Y_train_val)
            knnAccSumtest += bestKNN.score(X_test, Y_test)
            knnAccSumtrain += bestKNN.score(X_train_val, Y_train_val)
            testResults[scorerow,1] = bestKNN.score(X_test, Y_test)
            trainResults[scorerow,1] = bestKNN.score(X_train_val, Y_trai
n_val)

            #SVM
            svmCLF = svmClassifier(X_train_val, Y_train_val)
            bestSVM = svm.SVC(kernel = (svmCLF.best_params_["kernel"]),
C =(svmCLF.best_params_["C"])).fit(X_train_val, Y_train_val)
            svmAccSumtest += bestSVM.score(X_test, Y_test)
            svmAccSumtrain += bestSVM.score(X_train_val, Y_train_val)
            testResults[scorerow,2] = bestSVM.score(X_test, Y_test)
            trainResults[scorerow,2] = bestSVM.score(X_train_val, Y_trai
n_val)

            #bubble_KNN
            trainACC, testACC = bubbleACC(X_train_val, Y_train_val, X_te
```

```
st, Y_test,n=5)
            bubbleAccSumtest += testACC
            bubbleAccSumtrain += trainACC
            testResults[scorerow,3] = testACC
            trainResults[scorerow,3] = trainACC

            scorerow +=1
            print(scorerow)


        testResults[avgrow,0] = treeAccSumtest / 3.0
        trainResults[avgrow,0] = treeAccSumtrain / 3.0

        testResults[avgrow,1] = knnAccSumtest / 3.0
        trainResults[avgrow,1] = knnAccSumtrain / 3.0

        testResults[avgrow,2] = svmAccSumtest / 3.0
        trainResults[avgrow,2] = svmAccSumtrain / 3.0

        testResults[avgrow,3] = bubbleAccSumtest / 3.0
        trainResults[avgrow,3] = bubbleAccSumtrain / 3.0

        avgrow +=1
```

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
```

In [22]:
```
testResults
```

Out[22]:
```
array([[ 0.99363057,  0.70063694,  0.99363057,  0.73566879],
       [ 0.99044586,  0.75796178,  0.9522293 ,  0.69426752],
       [ 0.99044586,  0.75796178,  0.98407643,  0.77070064],
       [ 0.5875    ,  0.68      ,  0.6125    ,  0.585     ],
       [ 0.6175    ,  0.7175    ,  0.61      ,  0.625     ],
       [ 0.6475    ,  0.715     ,  0.595     ,  0.6375    ],
       [ 0.8825    ,  0.94      ,  0.83      ,  0.96      ],
       [ 0.9425    ,  0.9525    ,  0.83      ,  0.9325    ],
       [ 0.8675    ,  0.9775    ,  0.9025    ,  0.89      ],
       [ 0.98979592,  0.73469388,  0.98979592,  0.73979592],
       [ 1.        ,  0.78571429,  1.        ,  0.82653061],
       [ 0.98469388,  0.80102041,  0.98469388,  0.77040816],
       [ 0.696     ,  0.764     ,  0.728     ,  0.688     ],
       [ 0.64      ,  0.824     ,  0.736     ,  0.692     ],
       [ 0.748     ,  0.748     ,  0.704     ,  0.668     ],
       [ 0.972     ,  0.968     ,  0.852     ,  0.952     ],
       [ 0.972     ,  0.98      ,  0.84      ,  0.96      ],
       [ 0.976     ,  0.972     ,  0.84      ,  0.948     ],
       [ 1.        ,  0.75949367,  1.        ,  0.74683544],
       [ 0.98734177,  0.74683544,  0.98734177,  0.72151899],
       [ 1.        ,  0.82278481,  1.        ,  0.75949367],
       [ 0.71      ,  0.78      ,  0.65      ,  0.67      ],
       [ 0.66      ,  0.8       ,  0.71      ,  0.66      ],
       [ 0.69      ,  0.81      ,  0.76      ,  0.64      ],
       [ 0.94      ,  0.98      ,  0.81      ,  0.95      ],
       [ 0.97      ,  0.98      ,  0.89      ,  0.96      ],
       [ 0.98      ,  0.96      ,  0.85      ,  0.93      ],
       [ 0.99150743,  0.7388535 ,  0.97664544,  0.73354565],
       [ 0.6175    ,  0.70416667,  0.60583333,  0.61583333],
       [ 0.8975    ,  0.95666667,  0.85416667,  0.9275    ],
       [ 0.9914966 ,  0.77380952,  0.9914966 ,  0.77891156],
       [ 0.69466667,  0.77866667,  0.72266667,  0.68266667],
       [ 0.97333333,  0.97333333,  0.844     ,  0.95333333],
       [ 0.99578059,  0.77637131,  0.99578059,  0.74261603],
       [ 0.68666667,  0.79666667,  0.70666667,  0.65666667],
       [ 0.96333333,  0.97333333,  0.85      ,  0.94666667]])
```

In [23]:
```
dfTrain = pd.DataFrame(trainResults, columns=['Decision_Tree', 'K-Neares
t Neighbors', 'SVM', 'newK-Nearest Neighbors'],
                  index=['trainAccuracy(.2)(Auto_MPG) #1','trainAccuracy
(.2)(Auto_MPG) #2','trainAccuracy(.2)(Auto_MPG) #3',
                         'trainAccuracy(.2)(Cover_Type) #1','trainAccura
cy(.2)(Cover_Type) #2','trainAccuracy(.2)(Cover_Type) #3',
                         'trainAccuracy(.2)(Letter_Recognition) #1','tra
inAccuracy(.2)(Letter_Recognition) #2','trainAccuracy(.2)(Letter_Recogni
tion) #3',
                         'trainAccuracy(.5)(Auto_MPG) #1','trainAccuracy
(.5)(Auto_MPG) #2','trainAccuracy(.5)(Auto_MPG) #3',
                         'trainAccuracy(.5)(Cover_Type) #1','trainAccura
cy(.5)(Cover_Type) #2','trainAccuracy(.5)(Cover_Type) #3',
                         'trainAccuracy(.5)(Letter_Recognition) #1','tra
inAccuracy(.5)(Letter_Recognition) #2','trainAccuracy(.5)(Letter_Recogni
tion) #3',
                         'trainAccuracy(.8)(Auto_MPG) #1','trainAccuracy
(.8)(Auto_MPG) #2','trainAccuracy(.8)(Auto_MPG) #3',
                         'trainAccuracy(.8)(Cover_Type) #1','trainAccura
cy(.8)(Cover_Type) #2','trainAccuracy(.8)(Cover_Type) #3',
                         'trainAccuracy(.8)(Letter_Recognition) #1','tra
inAccuracy(.8)(Letter_Recognition) #2','trainAccuracy(.8)(Letter_Recogni
tion) #3',
                         'trainAccAverage(.2)(Auto_MPG)','trainAccAverag
e(.2)(Cover_Type)','trainAccAverage(.2)(Letter_Recognition)',
                         'trainAccAverage(.5)(Auto_MPG)','trainAccAverag
e(.5)(Cover_Type)','trainAccAverage(.5)(Letter_Recognition)',
                         'trainAccAverage(.8)(Auto_MPG)','trainAccAverag
e(.8)(Cover_Type)','trainAccAverage(.8)(Letter_Recognition)',])
dfTrain.index.name = 'AccuracyType(Train%)(Dataset)'

dfTrain
```

Out[23]:

	Decision_Tree	K-Nearest Neighbors	SVM	newK-Nearest Neighbors
AccuracyType(Train%) (Dataset)				
trainAccuracy(.2)(Auto_MPG) #1	0.987179	0.807692	0.987179	0.641026
trainAccuracy(.2)(Auto_MPG) #2	1.000000	1.000000	1.000000	0.730769
trainAccuracy(.2)(Auto_MPG) #3	1.000000	0.871795	1.000000	0.858974
trainAccuracy(.2)(Cover_Type) #1	0.980000	1.000000	0.830000	0.630000
trainAccuracy(.2)(Cover_Type) #2	0.870000	1.000000	0.740000	0.580000
trainAccuracy(.2)(Cover_Type) #3	0.890000	1.000000	0.910000	0.580000
trainAccuracy(.2) (Letter_Recognition) #1	0.980000	1.000000	1.000000	0.870000
trainAccuracy(.2) (Letter_Recognition) #2	1.000000	1.000000	1.000000	0.980000
trainAccuracy(.2) (Letter_Recognition) #3	0.950000	1.000000	0.920000	0.810000
trainAccuracy(.5)(Auto_MPG) #1	0.994898	0.877551	0.994898	0.826531
trainAccuracy(.5)(Auto_MPG) #2	0.984694	1.000000	0.984694	0.729592
trainAccuracy(.5)(Auto_MPG) #3	1.000000	0.903061	1.000000	0.790816
trainAccuracy(.5)(Cover_Type) #1	0.676000	1.000000	0.752000	0.644000
trainAccuracy(.5)(Cover_Type) #2	0.780000	1.000000	0.740000	0.620000
trainAccuracy(.5)(Cover_Type) #3	0.796000	1.000000	0.724000	0.652000
trainAccuracy(.5) (Letter_Recognition) #1	0.992000	0.972000	1.000000	0.936000
trainAccuracy(.5) (Letter_Recognition) #2	0.992000	1.000000	1.000000	0.908000

	Decision_Tree	K-Nearest Neighbors	SVM	newK-Nearest Neighbors
AccuracyType(Train%) (Dataset)				
trainAccuracy(.5) (Letter_Recognition) #3	1.000000	1.000000	1.000000	0.952000
trainAccuracy(.8)(Auto_MPG) #1	0.990415	1.000000	0.990415	0.769968
trainAccuracy(.8)(Auto_MPG) #2	0.993610	0.884984	0.993610	0.725240
trainAccuracy(.8)(Auto_MPG) #3	0.990415	0.884984	0.990415	0.792332
trainAccuracy(.8)(Cover_Type) #1	0.802500	1.000000	0.752500	0.695000
trainAccuracy(.8)(Cover_Type) #2	0.807500	1.000000	0.762500	0.710000
trainAccuracy(.8)(Cover_Type) #3	0.752500	1.000000	0.742500	0.735000
trainAccuracy(.8) (Letter_Recognition) #1	0.995000	1.000000	1.000000	0.942500
trainAccuracy(.8) (Letter_Recognition) #2	0.990000	1.000000	1.000000	0.900000
trainAccuracy(.8) (Letter_Recognition) #3	0.992500	1.000000	1.000000	0.927500
trainAccAverage(.2) (Auto_MPG)	0.995726	0.893162	0.995726	0.743590
trainAccAverage(.2) (Cover_Type)	0.913333	1.000000	0.826667	0.596667
trainAccAverage(.2) (Letter_Recognition)	0.976667	1.000000	0.973333	0.886667
trainAccAverage(.5) (Auto_MPG)	0.993197	0.926871	0.993197	0.782313
trainAccAverage(.5) (Cover_Type)	0.750667	1.000000	0.738667	0.638667
trainAccAverage(.5) (Letter_Recognition)	0.994667	0.990667	1.000000	0.932000
trainAccAverage(.8) (Auto_MPG)	0.991480	0.923323	0.991480	0.762513
trainAccAverage(.8) (Cover_Type)	0.787500	1.000000	0.752500	0.713333

	Decision_Tree	K-Nearest Neighbors	SVM	newK-Nearest Neighbors
AccuracyType(Train%) (Dataset)				
trainAccAverage(.8) (Letter_Recognition)	0.992500	1.000000	1.000000	0.923333

In [24]:
```
dfTest = pd.DataFrame(testResults, columns=['Decision_Tree', 'K-Nearest
 Neighbors', 'SVM', 'newK-Nearest Neighbors'],
                  index=['testAccuracy(.2)(Auto_MPG) #1','testAccuracy(.
2)(Auto_MPG) #2','testAccuracy(.2)(Auto_MPG) #3',
                         'testAccuracy(.2)(Cover_Type) #1','testAccuracy
(.2)(Cover_Type) #2','testAccuracy(.2)(Cover_Type) #3',
                         'testAccuracy(.2)(Letter_Recognition) #1','test
Accuracy(.2)(Letter_Recognition) #2','testAccuracy(.2)(Letter_Recognitio
n) #3',
                         'testAccuracy(.5)(Auto_MPG) #1','testAccuracy(.
5)(Auto_MPG) #2','testAccuracy(.5)(Auto_MPG) #3',
                         'testAccuracy(.5)(Cover_Type) #1','testAccuracy
(.5)(Cover_Type) #2','testAccuracy(.5)(Cover_Type) #3',
                         'testAccuracy(.5)(Letter_Recognition) #1','test
Accuracy(.5)(Letter_Recognition) #2','testAccuracy(.5)(Letter_Recognitio
n) #3',
                         'testAccuracy(.8)(Auto_MPG) #1','testAccuracy(.
8)(Auto_MPG) #2','testAccuracy(.8)(Auto_MPG) #3',
                         'testAccuracy(.8)(Cover_Type) #1','testAccuracy
(.8)(Cover_Type) #2','trainAccuracy(.8)(Cover_Type) #3',
                         'testAccuracy(.8)(Letter_Recognition) #1','test
Accuracy(.8)(Letter_Recognition) #2','testAccuracy(.8)(Letter_Recognitio
n) #3',
                         'testAccAverage(.2)(Auto_MPG)','testAccAverage
(.2)(Cover_Type)','testAccAverage(.2)(Letter_Recognition)',
                         'testAccAverage(.5)(Auto_MPG)','testAccAverage
(.5)(Cover_Type)','testAccAverage(.5)(Letter_Recognition)',
                         'testAccAverage(.8)(Auto_MPG)','testAccAverage
(.8)(Cover_Type)','testAccAverage(.8)(Letter_Recognition)',])
dfTest.index.name = 'AccuracyType(Train%)(Dataset)'

dfTest
```

Out[24]:

	Decision_Tree	K-Nearest Neighbors	SVM	newK-Nearest Neighbors
AccuracyType(Train%) (Dataset)				
testAccuracy(.2)(Auto_MPG) #1	0.993631	0.700637	0.993631	0.735669
testAccuracy(.2)(Auto_MPG) #2	0.990446	0.757962	0.952229	0.694268
testAccuracy(.2)(Auto_MPG) #3	0.990446	0.757962	0.984076	0.770701
testAccuracy(.2)(Cover_Type) #1	0.587500	0.680000	0.612500	0.585000
testAccuracy(.2)(Cover_Type) #2	0.617500	0.717500	0.610000	0.625000
testAccuracy(.2)(Cover_Type) #3	0.647500	0.715000	0.595000	0.637500
testAccuracy(.2)(Letter_Recognition) #1	0.882500	0.940000	0.830000	0.960000
testAccuracy(.2)(Letter_Recognition) #2	0.942500	0.952500	0.830000	0.932500
testAccuracy(.2)(Letter_Recognition) #3	0.867500	0.977500	0.902500	0.890000
testAccuracy(.5)(Auto_MPG) #1	0.989796	0.734694	0.989796	0.739796
testAccuracy(.5)(Auto_MPG) #2	1.000000	0.785714	1.000000	0.826531
testAccuracy(.5)(Auto_MPG) #3	0.984694	0.801020	0.984694	0.770408
testAccuracy(.5)(Cover_Type) #1	0.696000	0.764000	0.728000	0.688000
testAccuracy(.5)(Cover_Type) #2	0.640000	0.824000	0.736000	0.692000
testAccuracy(.5)(Cover_Type) #3	0.748000	0.748000	0.704000	0.668000
testAccuracy(.5)(Letter_Recognition) #1	0.972000	0.968000	0.852000	0.952000
testAccuracy(.5)(Letter_Recognition) #2	0.972000	0.980000	0.840000	0.960000

	Decision_Tree	K-Nearest Neighbors	SVM	newK-Nearest Neighbors
AccuracyType(Train%) (Dataset)				
testAccuracy(.5) (Letter_Recognition) #3	0.976000	0.972000	0.840000	0.948000
testAccuracy(.8)(Auto_MPG) #1	1.000000	0.759494	1.000000	0.746835
testAccuracy(.8)(Auto_MPG) #2	0.987342	0.746835	0.987342	0.721519
testAccuracy(.8)(Auto_MPG) #3	1.000000	0.822785	1.000000	0.759494
testAccuracy(.8)(Cover_Type) #1	0.710000	0.780000	0.650000	0.670000
testAccuracy(.8)(Cover_Type) #2	0.660000	0.800000	0.710000	0.660000
trainAccuracy(.8)(Cover_Type) #3	0.690000	0.810000	0.760000	0.640000
testAccuracy(.8) (Letter_Recognition) #1	0.940000	0.980000	0.810000	0.950000
testAccuracy(.8) (Letter_Recognition) #2	0.970000	0.980000	0.890000	0.960000
testAccuracy(.8) (Letter_Recognition) #3	0.980000	0.960000	0.850000	0.930000
testAccAverage(.2) (Auto_MPG)	0.991507	0.738854	0.976645	0.733546
testAccAverage(.2) (Cover_Type)	0.617500	0.704167	0.605833	0.615833
testAccAverage(.2) (Letter_Recognition)	0.897500	0.956667	0.854167	0.927500
testAccAverage(.5) (Auto_MPG)	0.991497	0.773810	0.991497	0.778912
testAccAverage(.5) (Cover_Type)	0.694667	0.778667	0.722667	0.682667
testAccAverage(.5) (Letter_Recognition)	0.973333	0.973333	0.844000	0.953333
testAccAverage(.8) (Auto_MPG)	0.995781	0.776371	0.995781	0.742616
testAccAverage(.8) (Cover_Type)	0.686667	0.796667	0.706667	0.656667

	Decision_Tree	K-Nearest Neighbors	SVM	newK-Nearest Neighbors
AccuracyType(Train%) (Dataset)				
testAccAverage(.8) (Letter_Recognition)	0.963333	0.973333	0.850000	0.946667

In [ ]: