```
import sys
```

```
# To add your own Drive Run this cell.
from google.colab import drive
drive.mount('/content/drive/')
```

```
    Mounted at /content/drive/
```

```
# Please append your own directory after ‘/content/drive/My Drive/'
# where you have nutil.py and adult_subsample.csv
### ========== TODO : START ========== ###
# for example: sys.path += ['/content/drive/My Drive/cs146/hw2_code']
sys.path += ['/content/drive/My Drive/hw2_code']
### ========== TODO : END ========== ###
```

```
from nutil import *
```

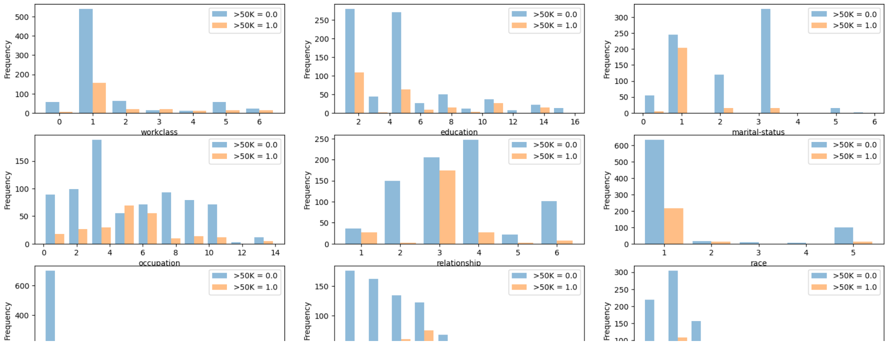
```
# Use only the provided packages!
import math
import csv

from collections import Counter

from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_score, cross_validate
from sklearn import metrics
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import StratifiedShuffleSplit
from sklearn.model_selection import learning_curve
```

```
######################################################################
# Immutatble classes
######################################################################

class Classifier(object) :
    """
    Classifier interface.
    """

    def fit(self, X, y):
        raise NotImplementedError()

    def predict(self, X):
        raise NotImplementedError()


class MajorityVoteClassifier(Classifier) :

    def __init__(self) :
        """
        A classifier that always predicts the majority class.

        Attributes
        --------------------
            prediction_ -- majority class
        """
        self.prediction_ = None

    def fit(self, X, y) :
        """
        Build a majority vote classifier from the training set (X, y).

        Parameters
        --------------------
            X    -- numpy array of shape (n,d), samples
            y    -- numpy array of shape (n,), target classes

        Returns
        --------------------
            self -- an instance of self
        """
        majority_val = Counter(y).most_common(1)[0][0]
        self.prediction_ = majority_val
        return self

    def predict(self, X) :
        """
        Predict class values.

        Parameters
        --------------------
            X    -- numpy array of shape (n,d), samples

        Returns
        --------------------
            y    -- numpy array of shape (n,), predicted classes
        """
        if self.prediction_ is None :
            raise Exception("Classifier not initialized. Perform a fit first.")

        n,d = X.shape
        y = [self.prediction_] * n
        return y



######################################################################
# Mutatble classes
######################################################################

class RandomClassifier(Classifier) :

    def __init__(self) :
        """
        A classifier that predicts according to the distribution of the classes.

        Attributes
        --------------------
            probabilities_ -- class distribution dict (key = class, val = probability of class)
        """
        self.probabilities_ = None

    def fit(self, X, y) :
        """
        Build a random classifier from the training set (X, y).

        Parameters
        --------------------
            X    -- numpy array of shape (n,d), samples
            y    -- numpy array of shape (n,), target classes

        Returns
        --------------------
            self -- an instance of self
        """

        ### ========== TODO : START ========== ###
        # part b: set self.probabilities_ according to the training set
        vals = dict(Counter(y))
        zeros = vals[0]
        ones = vals[1]
        total = zeros + ones
        p_zero = zeros / total
        p_one = ones / total

        self.probabilities_ = {0 : p_zero, 1 : p_one}
        ### ========== TODO : END ========== ###

        return self

    def predict(self, X, seed=1234) :
        """
        Predict class values.

        Parameters
        --------------------
```

```
            X    -- numpy array of shape (n,d), samples
            seed -- integer, random seed

        Returns
        --------------------
            y    -- numpy array of shape (n,), predicted classes
        """
        if self.probabilities_ is None :
            raise Exception("Classifier not initialized. Perform a fit first.")
        np.random.seed(seed)

        ### ========== TODO : START ========== ###
        # part b: predict the class for each test example
        # hint: use np.random.choice (be careful of the parameters)
        n,d = X.shape
        keys = list(self.probabilities_.keys())
        prob = list(self.probabilities_.values())

        y = []

        for i in range(0,n):
          pred = np.random.choice(keys, 1, p = prob)
          y.append(pred)
          i+= 1

        ### ========== TODO : END ========== ###

        return y



######################################################################
# Immutatble functions
######################################################################

def plot_histograms(X, y, Xnames, yname) :
    n,d = X.shape  # n = number of examples, d =  number of features
    fig = plt.figure(figsize=(20,15))
    ncol = 3
    nrow = d // ncol + 1
    for i in range(d) :
        fig.add_subplot (nrow,ncol,i+1)
        data, bins, align, labels = plot_histogram(X[:,i], y, Xname=Xnames[i], yname=yname, show = False)
        n, bins, patches = plt.hist(data, bins=bins, align=align, alpha=0.5, label=labels)
        plt.xlabel(Xnames[i])
        plt.ylabel('Frequency')
        plt.legend() #plt.legend(loc='upper left')

    plt.savefig ('histograms.pdf')


def plot_histogram(X, y, Xname, yname, show = True) :
    """
    Plots histogram of values in X grouped by y.

    Parameters
    --------------------
        X     -- numpy array of shape (n,d), feature values
        y     -- numpy array of shape (n,), target classes
        Xname -- string, name of feature
        yname -- string, name of target
    """

    # set up data for plotting
    targets = sorted(set(y))
    data = []; labels = []
    for target in targets :
        features = [X[i] for i in range(len(y)) if y[i] == target]
        data.append(features)
        labels.append('%s = %s' % (yname, target))

    # set up histogram bins
    features = set(X)
    nfeatures = len(features)
    test_range = list(range(int(math.floor(min(features))), int(math.ceil(max(features)))+1))
    if nfeatures < 10 and sorted(features) == test_range:
        bins = test_range + [test_range[-1] + 1] # add last bin
        align = 'left'
    else :
        bins = 10
        align = 'mid'

    # plot
    if show == True:
        plt.figure()
        n, bins, patches = plt.hist(data, bins=bins, align=align, alpha=0.5, label=labels)
        plt.xlabel(Xname)
        plt.ylabel('Frequency')
        plt.legend() #plt.legend(loc='upper left')
        plt.show()

    return data, bins, align, labels



######################################################################
# Mutatble functions
######################################################################

def error(clf, X, y, ntrials=100, test_size=0.15) :
    """
    Computes the classifier error over a random split of the data,
    averaged over ntrials runs.

    Parameters
    --------------------
        clf         -- classifier
        X           -- numpy array of shape (n,d), features values
        y           -- numpy array of shape (n,), target classes
        ntrials     -- integer, number of trials

    Returns
    --------------------
        train_error -- float, training error
        val_error  -- float, validation error
        f1_score    -- float, validation "micro" averaged f1 score
    """

    ### ========== TODO : START ========== ###
    # part f:
    # compute cross-validation error using StratifiedShuffleSplit over ntrials
    # hint: use StratifiedShuffleSplit (be careful of the parameters)
    t_error = []
    v_error = []
    f_score = []

    ss = StratifiedShuffleSplit(n_splits = 100, test_size = test_size, random_state = 0)
    for i, (train_index, test_index) in enumerate(ss.split(X, y)):
      xtr = X[train_index]
      xte = X[test_index]

      clf.fit(xtr, y[train_index])
      y_pred_tr = clf.predict(xtr)
      t_error.append(1 - metrics.accuracy_score(y[train_index], y_pred_tr, normalize = True))

      y_pred_te = clf.predict(xte)
      v_error.append(1 - metrics.accuracy_score(y[test_index], y_pred_te, normalize = True))

      f_score.append(metrics.f1_score(y[test_index], y_pred_te, average='micro'))
      i += 1

    train_error = np.mean(t_error)
    val_error = np.mean(v_error)
    f1_score = np.mean(f_score)


    ### ========== TODO : END ========== ###

    return train_error, val_error, f1_score
```

```
######################################################################
# Immutatble functions
######################################################################


def write_predictions(y_pred, filename, yname=None) :
    """Write out predictions to csv file."""
    out = open(filename, 'wb')
    f = csv.writer(out)
    if yname :
        f.writerow([yname])
    f.writerows(list(zip(y_pred)))
    out.close()



######################################################################
# main
######################################################################

# load adult_subsample dataset with correct file path

### ========== TODO : START ========== ###
# for example data_file =  "/content/drive/My Drive/cs146/hw1/adult_subsample.csv"
data_file = "/content/drive/My Drive/hw2_code/adult_subsample.csv"
### ========== TODO : END ========== ###

data = load_data(data_file, header=1, predict_col=-1)

X = data.X; Xnames = data.Xnames
y = data.y; yname = data.yname
n,d = X.shape  # n = number of examples, d =  number of features



plt.figure()
#========================================
# part a: plot histograms of each feature
print('Plotting...')
plot_histograms (X, y, Xnames=Xnames, yname=yname)
```

```
Plotting...
<Figure size 640x480 with 0 Axes>
```



```
#========================================
# train Majority Vote classifier on data
print('Classifying using Majority Vote...')
clf = MajorityVoteClassifier() # create MajorityVote classifier, which includes all model parameters
clf.fit(X, y)                  # fit training data using the classifier
y_pred = clf.predict(X)        # take the classifier and run it on the training data
train_error = 1 - metrics.accuracy_score(y, y_pred, normalize=True)
print('\t-- training error: %.3f' % train_error)
```

```
Classifying using Majority Vote...
        -- training error: 0.240
```

```
### ========== TODO : START ========== ###
# part b: evaluate training error of Random classifier
bc = RandomClassifier()
bc.fit(X, y)
y_p = bc.predict(X)
train_e = 1 - metrics.accuracy_score(y, y_p, normalize = True)
print('\t-- training error: %.3f' % train_e)
### ========== TODO : END ========== ###
```

```
        -- training error: 0.374
```

```
### ========== TODO : START ========== ###
# part c: evaluate training error of Decision Tree classifier
cc = DecisionTreeClassifier(criterion = 'entropy')
cc.fit(X,y)
y_pc = cc.predict(X)
train_ec = 1 - metrics.accuracy_score(y, y_pc, normalize = True)
print('\t-- training error: %.3f' % train_ec)
### ========== TODO : END ========== ###
```

```
        -- training error: 0.000
```

```
### ========== TODO : START ========== ###
# part d: evaluate training error of k-Nearest Neighbors classifier
# use k = 3, 5, 7 for n_neighbors
cd1 = KNeighborsClassifier(n_neighbors=3)
cd1.fit(X,y)
y_pcd1 = cd1.predict(X)
te_cd1 = 1 - metrics.accuracy_score(y, y_pcd1, normalize = True)
print('3 n = \t-- training error: %.3f' % te_cd1)

cd2 = KNeighborsClassifier(n_neighbors=5)
cd2.fit(X,y)
y_pcd2 = cd2.predict(X)
te_cd2 = 1 - metrics.accuracy_score(y, y_pcd2, normalize = True)
print('5 n = \t-- training error: %.3f' % te_cd2)

cd3 = KNeighborsClassifier(n_neighbors=7)
cd3.fit(X,y)
y_pcd3 = cd3.predict(X)
te_cd3 = 1 - metrics.accuracy_score(y, y_pcd3, normalize = True)
print('7 n = \t-- training error: %.3f' % te_cd3)
### ========== TODO : END ========== ###
```

```
3 n =   -- training error: 0.153
5 n =   -- training error: 0.195
7 n =   -- training error: 0.213
```

```
### ========== TODO : START ========== ###
# part e: evaluate training error of Logistic Regression
# use lambda_ = 0.1, 1, 10 for n_neighbors
# Note: Make sure you initialize your classifier with the appropriate parameters: random_state=0 and max_iter=1000, using the defa
ce1 = LogisticRegression(C=10, random_state=0, max_iter=1000)
ce1.fit(X,y)
y_pce1 = ce1.predict(X)
te_ce1 = 1 - metrics.accuracy_score(y, y_pce1, normalize = True)
print('lambda 0.1 = \t-- training error: %.3f' % te_ce1)

ce2 = LogisticRegression(C=1, random_state=0, max_iter=1000)
ce2.fit(X,y)
y_pce2 = ce2.predict(X)
te_ce2 = 1 - metrics.accuracy_score(y, y_pce2, normalize = True)
print('lambda 1 = \t-- training error: %.3f' % te_ce2)

ce3 = LogisticRegression(C=0.1, random_state=0, max_iter=1000)
ce3.fit(X,y)
y_pce3 = ce3.predict(X)
te_ce3 = 1 - metrics.accuracy_score(y, y_pce3, normalize = True)
print('lambda 10 = \t-- training error: %.3f' % te_ce3)
### ========== TODO : END ========== ###
```

```
lambda 0.1 =    -- training error: 0.208
lambda 1 =      -- training error: 0.208
lambda 10 =     -- training error: 0.220
```

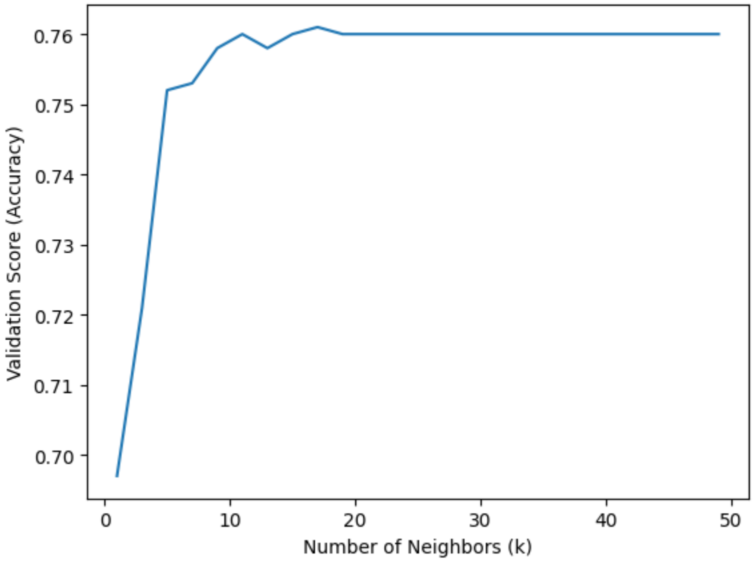
```
### ========== TODO : START ========== ###
# part f: use cross-validation to compute average training and validation error of classifiers
print('Investigating various classifiers...')
rclf = RandomClassifier()
rc_re, rc_se, rc_f = error(clf=rclf, X=X, y=y)
print('random = ', rc_re, rc_se, rc_f)

dclf = DecisionTreeClassifier(criterion = 'entropy')
dc_re, dc_se, dc_f = error(clf=dclf, X=X, y=y)
print("decision = ", dc_re, dc_se, dc_f)

kn = KNeighborsClassifier(n_neighbors = 5)
k_re, k_se, k_f = error(clf=kn, X=X, y=y)
print("k neighbors = ", k_re, k_se, k_f )

lr = LogisticRegression(C=1, random_state=0, max_iter=1000)
l_re, l_se, l_f = error(clf=lr, X=X, y=y)
print("log regression = ",l_re, l_se, l_f)
### ========== TODO : END ========== ###
```

```
Investigating various classifiers...
random =  0.37240000000000006 0.3666666666666667 0.6333333333333333
decision =  0.0 0.20199999999999999 0.7980000000000002
k neighbors =  0.19977647058823528 0.2544 0.7455999999999999
log regression =  0.20736470588235292 0.21240000000000003 0.7875999999999999
```

```
### ========== TODO : START ========== ###
# part g: use 5-fold cross-validation to find the best value of k for k-Nearest Neighbors classifier
print('Finding the best k...')
k = []
cv_scores = []

k_r = list(range(1,51,2))
print(k_r)

for kv in k_r:
  knn = KNeighborsClassifier(n_neighbors = kv)
  score = cross_val_score(knn, X, y, cv=5, scoring="accuracy")
  k.append(kv)
  cv_scores.append(score.mean())

plt.plot(k, cv_scores)
plt.xlabel('Number of Neighbors (k)')
plt.ylabel('Validation Score (Accuracy)')
plt.show()

best_k = k[cv_scores.index(max(cv_scores))]
best_score = max(cv_scores)
```

```
print(best_k)
print(best_score)

### ========== TODO : END ========== ###
```

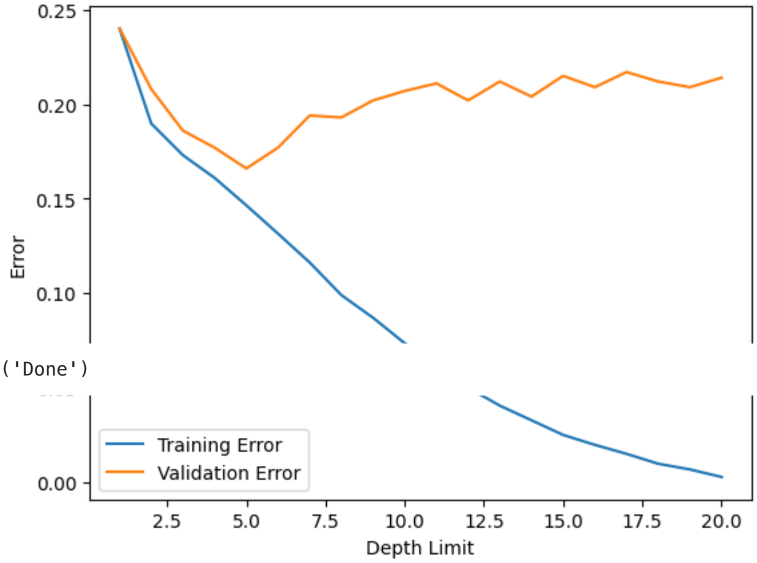
```
Finding the best k...
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39, 41, 43, 45, 47, 49]
```



```
17
0.7609999999999999
```

```
### ========== TODO : START ========== ###
# part h: investigate decision tree classifier with various depths
print('Investigating depths...')

depths = list(range(1,21))

ate = []
ave = []

for d in depths:
  clf = DecisionTreeClassifier(criterion = 'entropy', max_depth = d)
  scores = cross_validate(clf, X, y, cv=5, scoring='accuracy', return_train_score=True)

  train_errors = 1 - scores['train_score']
  val_errors = 1 - scores['test_score']

  ate.append(np.mean(train_errors))
  ave.append(np.mean(val_errors))

plt.plot(depths, ate, label = "Training Error")
plt.plot(depths, ave, label = "Validation Error")
plt.xlabel('Depth Limit')
plt.ylabel('Error')
plt.legend()
plt.show()

print(depths)
print(ate)
print(ave)
### ========== TODO : END ========== ###
```

```
Investigating depths...
```



```
print('Done')
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
[0.24, 0.18975, 0.173, 0.161, 0.14650000000000002, 0.1315, 0.11624999999999999,
[0.24, 0.20799999999999996, 0.186, 0.177, 0.166, 0.177, 0.19399999999999998, 0.1
```