02-750

Assignment Two

Team:
Zeyuan Zuo (zeyuanz)
Tianqin Li (tianqinl)

Q1_ZLG

March 19, 2021

0.1 Question 1. ZLG algorithm implementation (50 points)

You are to implement the ZLG algorithem for this problem. We will use a subset of multiclass data where the label is a protein subcellular localization. The 8 features are extracted from the protein sequence. For this problem we are only using points with labels 'MIT' or 'NUC'.

First, read the paper and answer the following questions. #### 1. What is the idea behind the ZLG algorithm (5 points)? Data are represented as nodes in a graph, with edge weights represets the similarity between data points. Active learning is performed based on gaussian random field graph by greedily selecting queries from the unlabeled data to minimize the estimated expected risk. Since the graph is modeled as gausian random field, the risk can be efficiently computed using matrix operations. #### 2. What are the assumptions behind the ZLG algorithm (5 points)? - Assumes gaussian distribution over the unlabeled nodes - Labels are binary - If two unlabeled data point are similar, they should have the same label - Labeled data points have {0,1} labels while unlabeled data points can have continuous labels in [0,1]

3. What are the pros and cons of the ZLG algorithm (5points)?

- Pros: Algorithm is easy to implement. Due the property of gaussian random field, some calculations have closed form optimal solutions.
- Cons: It takes long to run, $O(n^3)$ per iteration. It suffers from sampling bias.

[1]:
```
import numpy as np
import pandas as pd
from scipy.spatial import distance_matrix
from sklearn.preprocessing import LabelEncoder
```

A total of 892 data points have labels 'MIT' (244) or 'NUC' (429). We start with the labels of only the first 200 data points (set Y_k). The other 792 points are in Y_u.

[2]:
```
data = pd.read_csv('data/data.csv')
data_CYTNUC = data.loc[data['Label'].isin(['MIT','NUC'])].values
X = data_CYTNUC[:,:8]
y = LabelEncoder().fit_transform(data_CYTNUC[:,-1])

n_l = 200

Xk = X[:n_l,:]
Yk = y[:n_l]
```

```
Xu = X[n_l:,:]
Yu = y[n_l:]
```

(5 points) Let's first construct the weight matrix W. Use t = 0, σ as the standard deviation of X. Then calculate the D matrix and the Laplacian matrix (Delta).

```
[3]: def Laplacian_matrix(X):
         # change X to float type
         X = X.astype(float)
         # initialize weight matrix of shape (n_data * n_data)
         weight_matrix = np.zeros(shape = (X.shape[0], X.shape[0]))
         # set sigma = standard deviation of X
         sigma = np.std(X)
         for i in range(weight_matrix.shape[0]):
             # calculate RBF weight matrix
             weight_matrix[i,:] = np.exp(-np.sum((X - X[i,:]) ** 2, axis = 1) /␣
      ↪(sigma ** 2))
         Delta = np.diag(weight_matrix.sum(axis = 1)) - weight_matrix
         return Delta

     Delta = Laplacian_matrix(X)
```

(5 points) Now complete the subroutine to compute the minimum-energy solution for the unlabeled instances. (Hint: Use the formula in page 38, Lecture 7.) The function also outputs one submatrix that we will use to select points to query.

```
[4]: def minimum_energy_solution(Delta,n_l,fl):
         """
         Args:
             Delta: The Laplacian matrix.
             n_l: Number of labeled points. Notice that Delta should have the upper␣
      ↪left submatrix
                 corresponding to these n_l points. So when new points get labeled,␣
      ↪you may need
                 to rearrange the matrix.
             fl: Known labels.
         Returns:
             Delta_uu_inv: Inverse matrix of the submatrix corresponding to␣
      ↪unlabeled points.
             fu: Minimum energy solution of all unlabeled points.
         """
         ## TODO ##
         Delta_uu = Delta[n_l:, n_l:]
         Delta_ul = Delta[n_l:, :n_l]
         Delta_uu_inv = np.linalg.inv(Delta_uu)
         fu = -Delta_uu_inv.dot(Delta_ul).dot(fl)
         return Delta_uu_inv, fu
```

```
Delta_uu_inv, fu = minimum_energy_solution(Delta,n_l,Yk)
```

(15 points) We would like to query the points that minimize the expected risk. To do so, we want to be able to calculate the expected estimated risk after querying any point k. The variable Rhat_fplus_xk refers to $\hat{R}(f^{+x_k})$. fu_xk0 is $f_u^{+(x_k,0)}$ and vice versa for fu_xk1.

```
[5]: def expected_estimated_risk(Delta_uu_inv,k,fu):
         """
         Args:
             Delta_uu_inv: Inverse matrix of the submatrix corresponding to␣
      ↪unlabeled points.
             k: index of one unlabeled point with respect to the uu submatrix (not␣
      ↪the entire Delta)
             fu: Minimum energy solution of all unlabeled points.
         Returns:
             Rhat_fplus_xk: Expected estimated risk after querying node k
         """
         ## fu plus xk, yk = 0
         fu_xk0 = fu + (0 - fu[k])*Delta_uu_inv[:,k]/Delta_uu_inv[k,k]
         ## fu plus xk, yk = 1
         fu_xk1 = fu + (1 - fu[k])*Delta_uu_inv[:,k]/Delta_uu_inv[k,k]

         ## TODO ##
         Rhat_fplus_xk0 = np.minimum(fu_xk0, 1-fu_xk0).sum()
         Rhat_fplus_xk1 = np.minimum(fu_xk1, 1-fu_xk1).sum()
         Rhat_fplus_xk = (1 - fu[k])*Rhat_fplus_xk0 + fu[k]*Rhat_fplus_xk1
         return Rhat_fplus_xk
```

(5 points) Compute the above expected estimated risk for all unlabeled points and select one to query.

```
[6]: def zlg_query(Delta_uu_inv,n_l,fu,n_samples):
         """
         Args:
             Delta_uu_inv: Inverse matrix of the submatrix corresponding to␣
      ↪unlabeled points.
             n_l: Number of labeled points.
             fu: Minimum energy solution of all unlabeled points.
             n_samples: Number of samples.
         Returns:
             query_idx: the idx of the point to query, wrt the unlabeled points
                     (idx is 0 if it's the first unlabeled point)
         """
         n_u = n_samples - n_l
         query_idx = 0
```

```
    min_Rhat = np.inf
    ## TODO ##
    for i in range(n_u):
        tmp_Rhat = expected_estimated_risk(Delta_uu_inv, i, fu)
        if tmp_Rhat < min_Rhat:
            min_Rhat = tmp_Rhat
            query_idx = i
    return query_idx
```

Let's try query 100 points. Which points are queried? Compare with random queries and make a plot.

[7]:
```
def accuracy(fu, Yu):
    acc = 0.0
    for i in range(len(fu)):
        if fu[i] > 0.5 and Yu[i] == 1:
            acc += 1
        if fu[i] < 0.5 and Yu[i] == 0:
            acc += 1
    return acc / len(Yu)
```

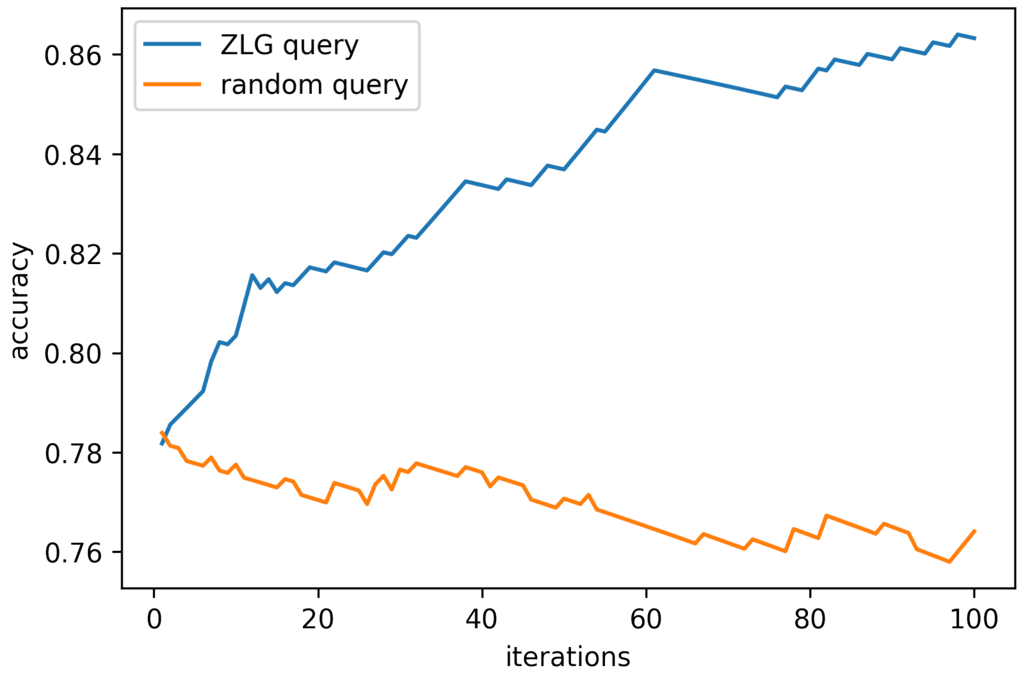
[8]:
```
# ZLG query
n_iter = 100
n_samples = X.shape[0]
accuracy_history_ZLG = []
for t in range(n_iter):
    ## edit this block ##
    query_idx = zlg_query(Delta_uu_inv,n_l,fu,n_samples)
    Yk = np.append(Yk,Yu[query_idx])
    Yu = np.delete(Yu,query_idx)
    Xk = np.append(Xk,[Xu[query_idx,:]],axis=0)
    Xu = np.delete(Xu,query_idx, 0)
    n_l += 1
    Delta = Laplacian_matrix(np.concatenate((Xk,Xu),axis=0))
    Delta_uu_inv, fu = minimum_energy_solution(Delta,n_l,Yk)
#     print(query_idx)
    ## TODO ##
    accuracy_history_ZLG.append(accuracy(fu, Yu))
```

[9]:
```
# random query
n_l = 200
accuracy_history_random = []
Xk = X[:n_l,:]
Yk = y[:n_l]
Xu = X[n_l:,:]
Yu = y[n_l:]
for t in range(n_iter):
    ## edit this block ##
    query_idx = np.random.randint(0, Yu.shape[0])
    Yk = np.append(Yk,Yu[query_idx])
    Yu = np.delete(Yu,query_idx)
    Xk = np.append(Xk,[Xu[query_idx,:]],axis=0)
    Xu = np.delete(Xu,query_idx, 0)
    n_l += 1
    Delta = Laplacian_matrix(np.concatenate((Xk,Xu),axis=0))
    Delta_uu_inv, fu = minimum_energy_solution(Delta,n_l,Yk)
#     print(query_idx)
    ## TODO ##
    accuracy_history_random.append(accuracy(fu, Yu))
```

[10]:
```
import matplotlib.pyplot as plt
```

[11]:
```
time = np.linspace(1,n_iter,n_iter)
fig, ax = plt.subplots(dpi = 300)
ax.plot(time, accuracy_history_ZLG, label = 'ZLG query')
ax.plot(time, accuracy_history_random, label = 'random query')
ax.set_xlabel('iterations')
ax.set_ylabel('accuracy')
ax.legend()
```

[11]:
```
<matplotlib.legend.Legend at 0x7ff00a6542e0>
```



0.1.1 Explaination

In 100 iterations, ZLG algorithm significantly achieves a higher accuray than random sampling. It shows the benefits of active learning.

Bonus question (Your grade will not exceed 100 for this homework): For this dataset, how many labeled data points do you actually need, to train the model sufficiently well? And why?

Q2_DH_tq

March 19, 2021

1 Question 2: DH algorithm (50 points)

In this question we are going to implmeneted the DH algorithm according to this paper:https://icml.cc/Conferences/2008/papers/324.pdf, in which we try to predict protein localization sites in Eukaryotic cells.

[1]:
```
import copy
import warnings
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from numpy.random import choice
from scipy.cluster.hierarchy import linkage
from sklearn.datasets import make_blobs
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.linear_model import LogisticRegression
seed = 2021
warnings. filterwarnings("ignore")
```

1.1 2.0 Data loading and hierarchical clustering

DH algorithm is based on hierarchical clustering of the dataset, we will use the DH algorithm on this classification problem: Protein Localization Prediction, the first step is to load the dataset and conduct a hierarchical clustring using the **Scipy** package. *This part has been implemented, read through the code to make sure you understand what is being done.*

[2]:
```
def load_data(seed = 2021):
    """ Loads "Protein Localizataion Prediction" data. Computes linkage from␣
↪hierarchical clustering.
    :returns X_train: data matrix 538x8
    :returns Y_train: true labels 538x1
    :returns X_test: data matrix 135x8
    :returns Y_test: true labels 135x1
    :returns T: 3 element tree
        T[0] = linkage matrix from hierarchical clustering.  See https://docs.
↪scipy.org/doc/scipy/reference/generated/scipy.cluster.hierarchy.linkage.html
```

```
                for details. If you are unfamiliar with hierarchical clustering␣
↪using scipy, the following is another helpful resource (We won't use␣
↪dendrograms
                here, but he gives a nice explanation of how to interpret the␣
↪linkage matrix):
                https://joernhees.de/blog/2015/08/26/
↪scipy-hierarchical-clustering-and-dendrogram-tutorial/

        T[1] = An array denoting the size of each subtree rooted at node i,␣
↪where i indexes the array.
                ie. The number of all leaves in subtree rooted at node i (w_i in␣
↪the paper).

        T[2] = dict where keys are nodes and values are the node's parent
        """
    df = pd.read_csv('./data/data.csv')
    np.unique(df.Label,return_counts = True)
    filter_class = ['MIT','NUC']
    mask = df.Label ==0
    for x in filter_class:
        mask = mask | (df.Label==x)
    df = df[mask]
    X = df.iloc[:,:8].to_numpy()
    y = df.Label.astype('category').cat.codes.to_numpy()
    X, X_test, y, y_test = train_test_split(X,y,test_size = 0.2, random_state =␣
↪seed)
    n_samples = len(X)
    Z = linkage(X,method='ward')
    link = Z[:,:2].astype(int)
    subtree_sizes = np.zeros(link[-1,-1]+2)
    subtree_sizes[:n_samples] = 1
    parent = {}
    parent[2*(n_samples-1)] = -1 #set root node as 0
    for i in range(len(link)):
        left = link[i,0]
        right = link[i,1]
        current = i + n_samples
        subtree_sizes[current] = subtree_sizes[left] + subtree_sizes[right]
        parent[left] = current
        parent[right] = current

    T = [link,subtree_sizes,parent]

    return X.astype("float"), y, X_test, y_test, T

X_train, y_train, X_test, y_test, T = load_data()
```

```
[ ]: ## Logistic Regression
     lr = LogisticRegression()
     lr.fit(X_train, y_train)

     ## Random Forest
     N_estimator_rf = 20
     MAX_depth_rf = 6
     rf = RandomForestClassifier(n_estimators = N_estimator_rf,
                                 max_depth = MAX_depth_rf, random_state = seed)
     rf.fit(X_train, y_train)

     ## Gradient Boosting Decision Tree
     N_estimator_gbdt = 20
     gbdt_max_depth = 6
     gbdt = GradientBoostingClassifier(n_estimators = N_estimator_gbdt,
                                       learning_rate = 0.1,
                                       max_depth = gbdt_max_depth,
                                       random_state = seed)
     gbdt.fit(X_train,y_train)

     ## 3-Layer fully connected NN
     import torch
     from torch.utils.data import Dataset
     from torch import optim
     from torch.utils.data import DataLoader, TensorDataset
     torch.manual_seed(seed)
     class NNClassifier(object):
         def __init__(self,
                      feature_n,
                      class_n,
                      hidden_n = 30,
                      learning_rate = 4e-3,
                      weight_decay = 1e-5):
             self.model = torch.nn.Sequential(torch.nn.Linear(feature_n,hidden_n),
                                              torch.nn.SiLU(),
                                              torch.nn.Linear(hidden_n,hidden_n),
                                              torch.nn.SiLU(),
                                              torch.nn.Linear(hidden_n,class_n))
             self.model =self.model.cuda()
             self.lr = learning_rate
             self.wd = weight_decay
         def fit(self,X_train,y_train,epoches = 300,batch_size = 50):
             X_t = torch.from_numpy(X_train.astype(np.float32))
             y_t = torch.from_numpy(y_train.astype(np.int64))
             dataset = TensorDataset(X_t,y_t)
             loader = DataLoader(dataset,batch_size = batch_size,shuffle = True)
             loss_fn = torch.nn.CrossEntropyLoss(reduction = 'mean').cuda()
```

```
        optimizer = optim.Adam(self.model.parameters(), lr=self.
↪lr,weight_decay=self.wd)
        loss_record = 0.0
        report_epoch = 50
        for epoch_i in range(epoches):
            for batch in loader:
                x_batch,y_batch = batch
                x_batch = x_batch.cuda()
                y_batch = y_batch.cuda()
                y_pred = self.model(x_batch)
                loss = loss_fn(y_pred,y_batch)
                self.model.zero_grad()
                loss.backward()
                optimizer.step()
                loss_record += loss.item()
            if epoch_i%report_epoch == report_epoch-1:
                print("[%d|%d] epoch loss:%.2f"%(epoch_i+1,epoches,loss_record/
↪report_epoch))
                loss_record = 0.0
            if epoch_i>=epoches:
                break

    def score(self,X_test,y_test):
        X_test_tensor = torch.from_numpy(X_test.astype(np.float32))
        y_pred_test = self.model(X_test_tensor)
        y_output = torch.argmax(y_pred_test,axis = 1).numpy()
        return (y_output == y_test).mean()

nn = NNClassifier(feature_n = X_train.shape[1],class_n = len(np.
↪unique(y_train)))
nn.fit(X_train,y_train)

## Accuracy of 4 classifiers.
print('Accuracy of logistic regression: \t{:.3f}'.format(lr.
↪score(X_test,y_test)))
print('Accuracy of random forest: \t\t{:.3f}'.format(rf.score(X_test,y_test)))
print('Accuracy of Gradient Boosting Decision Tree: \t\t{:.3f}'.format(gbdt.
↪score(X_test,y_test)))
print('Accuracy of Neural Network: \t\t{:.3f}'.format(nn.score(X_test,y_test)))
```

2 2.0.1 Supervised classification methods.

Following we provide several classifiers that can be used, choose your favourite one. To use the Neural Network classifier, you need to install pytorch.

2.0.1 Choose and initialize your classifier:

The classifier is going to be used in 2.2, the choose of classifier won't influence your grade

[5]:
```python
#TODO: Uncomment one line to choose your classifier.
#classifier = LogisticRegression()

#classifier = RandomForestClassifier(n_estimators = N_estimator_rf,max_depth =␣
 ↪MAX_depth_rf, random_state = seed)

#classifier = GradientBoostingClassifier(n_estimators = N_estimator_gbdt,
#                                    learning_rate = 0.1,
#                                    max_depth = gbdt_max_depth,
#                                    random_state = seed)

classifier = NNClassifier(feature_n = X_train.shape[1],class_n = len(np.
 ↪unique(y_train)))
```

2.1 2.1 Implement the DH algorithm (Hierarchical Sampling for Active Learning). (30 points)

Please complete the functions to implement the DH algorithm and run the active learning algorithm on the training dataset. The utils functions has been implemented and attached in the homework folder, including **update_empirical.py, best_pruning_and_labeling.py, assign_labels.py and get_leaves.py**, please read them and finish the following functions to implement the DH algorithm.

[6]:
```python
from update_empirical import update_empirical
from best_pruning_and_labeling import best_pruning_and_labeling
from assign_labels import assign_labels
from get_leaves import get_leaves
```

[27]:
```python
def compute_error(L,labels):
    """Compute the error

    :param L: labeling of leaf nodes
    :param labels: true labels of each node

    :returns error: error of predictions"""

    wrong = 0
    wrong = (L[:len(labels)]!=labels).sum()
    error = wrong/len(labels)
    return error

def select_case_1(data,labels,T,budget,batch_size):
    """DH algorithm where we choose P proportional to the size of subtree␣
 ↪rooted at each node
```

```
    :param data: Data matrix 1200x8
    :param labels: true labels 1200x1
    :param T: 3 element tree
        T[0] = linkage matrix from hierarchical clustering.  See https://docs.
↪scipy.org/doc/scipy/reference/generated/scipy.cluster.hierarchy.linkage.html
               for details. If you are unfamiliar with hierarchical clustering␣
↪using scipy, the following is another helpful resource (We won't use␣
↪dendrograms
               here, but he gives a nice explanation of how to interpret the␣
↪linkage matrix):
               https://joernhees.de/blog/2015/08/26/
↪scipy-hierarchical-clustering-and-dendrogram-tutorial/

        T[1] = An array denoting the size of each subtree rooted at node i,␣
↪where i indexes the array.
               ie. The number of all children + grandchildren + ... + the node␣
↪itself

        T[2] = dict where keys are nodes and values are the node's parent
    :param budget: Number of iterations to make
    :param batch_size: Number of queries per iteration"""

    n_nodes = len(T[1]) #total nodes in T
    n_samples = len(data) #total samples in data
    L = np.zeros(n_nodes) #majority label
    p1 = np.zeros(n_nodes) #empirical label frequency (for label 1)
    n = np.zeros(n_nodes) #number of points sampled from each node
    error = []#np.zeros(n_samples) #error at each round
    root = n_nodes-1 #corresponds to index of root
    P = np.array([root])
    L[root] = 1

    for i in range(budget):
        selected_P = []
        for b in range(batch_size):

            #TODO: select a node from P proportional to the size of subtree␣
↪rooted at each node
            size_p = np.array([len(get_leaves([],v,T,n_samples)) / n_samples␣
↪for v in P])
            probs = size_p / sum(size_p)
            select_node_index = np.random.choice(np.arange(len(P)), p=probs)
            select_node = P[select_node_index]
            selected_P.append(select_node)
            ##TODO: pick a random leaf node from subtree Tv and query its label
            leafs = get_leaves([],select_node,T,n_samples)
            z = np.random.choice(leafs)
            z_l = labels[z]

            #TODO: update empirical counts and probabilities for all nodes u on␣
 ↪path from z to v
            n, p1 = update_empirical(n,p1,select_node,z,z_l,T)

        for p in selected_P:
            #TODO: update admissible A and compute scores; find best pruning␣
 ↪and labeling
            P_best, L_best = best_pruning_and_labeling(n,p1,p,T,n_samples)
            #TODO: update pruning P and labeling L
            P = np.array([node_ for node_ in P if node_ != p] + [node_ for␣
 ↪node_ in P_best])
            L[p] = L_best


        #TODO: temporarily assign labels to every leaf and compute error
        L_temp = L.copy()
        for i in range(len(P)):
            L_temp = assign_labels(L_temp,P[i],P[i],T,n_samples)
        error_i = compute_error(L[:n_samples],labels)
        error.append(error_i)

    for i in range(len(P)):
        L = assign_labels(L,P[i],P[i],T,n_samples)

    return L, np.array(error)

def select_case_2(data,labels,T,budget,batch_size):
    """DH algorithm where we choose P by biasing towards choosing nodes in␣
 ↪areas where the observed labels are less pure

    :param data: Data matrix 1200x8
    :param labels: true labels 284x1
    :param T: 3 element tree
        T[0] = linkage matrix from hierarchical clustering.  See https://docs.
 ↪scipy.org/doc/scipy/reference/generated/scipy.cluster.hierarchy.linkage.html
               for details. If you are unfamiliar with hierarchical clustering␣
 ↪using scipy, the following is another helpful resource (We won't use␣
 ↪dendrograms
               here, but he gives a nice explanation of how to interpret the␣
 ↪linkage matrix):
```

```
            https://joernhees.de/blog/2015/08/26/
↪scipy-hierarchical-clustering-and-dendrogram-tutorial/

        T[1] = An array denoting the size of each subtree rooted at node i,␣
↪where i indexes the array.
            ie. The number of all children + grandchildren + ... + the node␣
↪itself

        T[2] = dict where keys are nodes and values are the node's parent
    :param budget: Number of iterations to make
    :param batch_size: Number of queries per iteration"""

    n_nodes = len(T[1]) #total nodes in T
    n_samples = len(data) #total samples in data
    L = np.zeros(n_nodes,dtype = int) #majority label
    p1 = np.zeros(n_nodes) #empirical label frequency
    n = np.zeros(n_nodes) #number of points sampled from each node
    error = []#np.zeros(n_samples) #error at each round
    root = n_nodes-1 #corresponds to index of root
    P = np.array([root])
    L[root] = 1

    for i in range(budget):
        selected_P = []
        for b in range(batch_size):
            #TODO: select a node from P biasing towards choosing nodes in areas␣
↪where the observed labels are less pure
            size_p = np.array([len(get_leaves([],v,T,n_samples)) / n_samples␣
↪for v in P])
            weights = size_p / sum(size_p)
            probs = weights
            for idx, prune in enumerate(P):
                lv = L[prune]
                if lv == 1:
                    pvlv = p1[prune]
                else:
                    pvlv = 1 - p1[prune]
                upper_pvlv = min(1, pvlv - (1 / (1e-12 + n[prune]) + \
                                            np.sqrt((pvlv * (1 - pvlv)) /␣
↪(1e-12 + n[prune])))
                                )

                probs[idx] = weights[idx] * (1 - upper_pvlv)

            if probs.sum() != 0:
                probs = probs / probs.sum()
            else:
                probs = None
            select_node_index = np.random.choice(np.arange(len(P)), p=probs)
            select_node = P[select_node_index]
            selected_P.append(select_node)

            #TODO: pick a random leaf node from subtree Tv and query its label
            leafs = get_leaves([],select_node,T,n_samples)
            z = np.random.choice(leafs)
            z_l = labels[z]
            #TODO: update empirical counts and probabilities for all nodes u on␣
↪path from z to v
            n, p1 = update_empirical(n,p1,select_node,z,z_l,T)

        for p in selected_P:
            #TODO: update admissible A and compute scores; find best pruning␣
↪and labeling
            P_best, L_best = best_pruning_and_labeling(n,p1,p,T,n_samples)
            #TODO: update pruning P and labeling L
            P = np.array([node_ for node_ in P if node_ != p] + [node_ for␣
↪node_ in P_best])
            L[p] = L_best

        #TODO: temporarily assign labels to every leaf and compute error
        L_temp = L.copy()
        error_i = compute_error(L[:n_samples],labels)
        error.append(error_i)

    for i in range(len(P)):
        L = assign_labels(L,P[i],P[i],T,n_samples)

    return L, np.array(error)
```

2.2 2.2 Run the sample code (10 points)

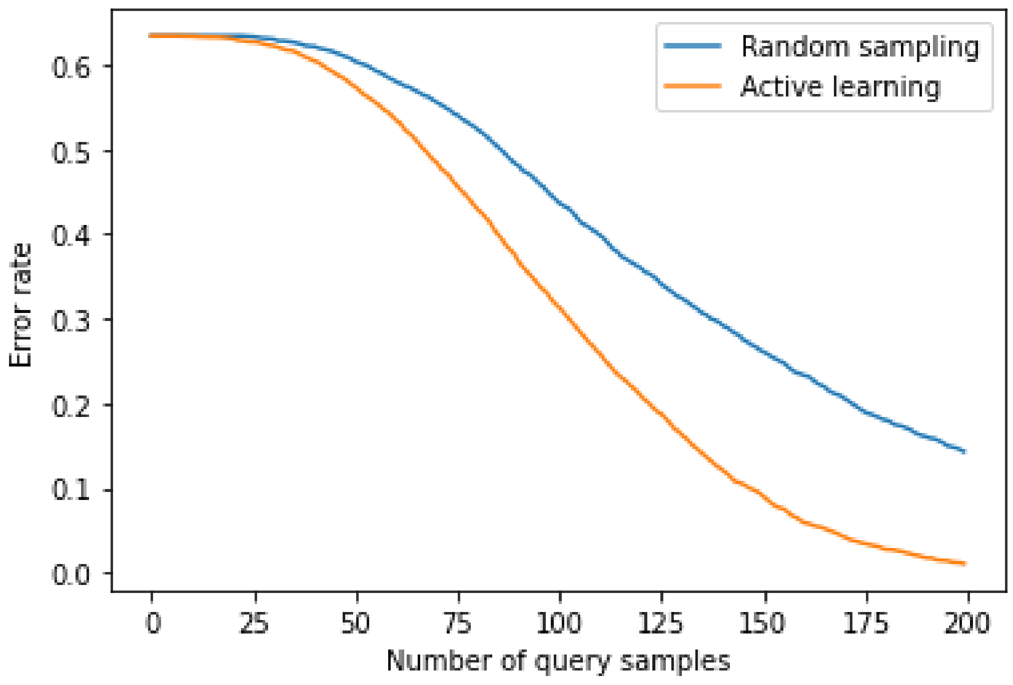
Run the following sample code and compare the two figures.

[29]:
```
def call_DH(part,clf,budget):
    """Main function to run all your code once complete.  After you complete
       select_case_1() and select_case_2(), this will run the DH algo for each
       dataset and generate the plots you will submit within your write-up.

       :param part: which part of the homework to run
       :param clf: The classifier used to predcit on the dataset.
       :param budget: The number of times that one can query a label from the␣
↪oracle.
    """
```

```
part = part.lower()
num_trials = 5
batch_size = 10
clf2 = copy.deepcopy(clf)
axs = plt.subplot()
if part == "b":
    print("Running part B")
    X_train, y_train, X_test, y_test, T = load_data()
    l = np.zeros(budget)
    for i in range(num_trials):
        print("Currently on iteration {}".format(i))
        L, error = select_case_1(X_train,y_train,T,budget,batch_size)
        l += error
    l /= num_trials

    ## TODO: train the classifier clf on the predicted label.
    #raise(NotImplementedError)
    clf.fit(X_train,L[:len(X_train)])

    print('Accuracy of classifier trained on random sampling dataset: \t{:.
↪3f}'.format(clf.score(X_test,y_test)))
    axs.plot(np.arange(budget),l,label = "Random sampling")

elif part == "c":
    print("Running part C")
    X_train, y_train, X_test, y_test, T = load_data()
    l = np.zeros(budget)
    for i in range(num_trials):
        print("Currently on iteration {}".format(i))
        L, error = select_case_2(X_train,y_train,T,budget,batch_size)
        l += error
    l /= num_trials

    ## TODO: train the classifier clf2 on the predicted label.
    #raise(NotImplementedError)
    clf2.fit(X_train,L[:len(X_train)])

    print('Accuracy of classifier trained on active learning dataset: \t{:.
↪3f}'.format(clf2.score(X_test,y_test)))
    axs.plot(np.arange(budget),l,label = "Active learning")

else:
    print("Incorrect part provided. Either 'b', 'c', 'd', or 'e' expected")
axs.set_xlabel("Number of query samples")
axs.set_ylabel("Error rate")
plt.legend()
```

```python
    plt.savefig("q2_2_bc.png")

BUDGET = 200 #You can change this number to a smaller one during testing.
for part in "bc":
    call_DH(part,classifier,BUDGET)
```

```
Running part B
Currently on iteration 0
Currently on iteration 1
Currently on iteration 2
Currently on iteration 3
Currently on iteration 4
[50|300] epoch loss:5.32
[100|300] epoch loss:5.18
[150|300] epoch loss:5.07
[200|300] epoch loss:5.00
[250|300] epoch loss:4.95
[300|300] epoch loss:4.90
Accuracy of classifier trained on random sampling dataset:      0.844
Running part C
Currently on iteration 0
Currently on iteration 1
Currently on iteration 2
Currently on iteration 3
Currently on iteration 4
[50|300] epoch loss:3.96
[100|300] epoch loss:3.76
[150|300] epoch loss:3.71
[200|300] epoch loss:3.64
[250|300] epoch loss:3.55
[300|300] epoch loss:3.49
Accuracy of classifier trained on active learning dataset:      0.867
```



2.3 Explain

From the figure we can see that the active learning approach can produce better error rate comparing to random sampling, indicating active learning can approximate more accurate lables than random sampling.

Additionally, the active learning also converge faster than random sampling and produce higher accuarcy at the test time.

2.4 2.3 Questions (10 points):

2.4.1 What is a "admissible pair" according to the paper (5 points)?

2.4.2 Please explain the sampling bias that is dealt with in the DH algorithm and why it would be a problem if we just query the unlabeled point which is closest to the decision boundary (5 points)?

2.5 1. "Admissible pair"

$$A_{v,l}(t) = True \iff (1 - p_{v,l}^{LB}(t)) < \beta min_{l' \neq l}(1 - p_{v,l'}^{UB}(t))$$

On the left, it means the max error if v is paired with l, and the right represents the min error if v pair with other labels l'. So we consider (v, l) to be admissible, i.e. we are confident abount assigning l to v when even the worst mistake we can make by assigning l to v is better than the best other assignment can produce.

2.6 2. Sampling bias

The initial random sample would be bias so methods that are based on the initial sample to partition the hypothesis space would be biased since some rare case could be overlooked. If we then follow the initial samples to query points around the boundary, then we will only get suboptimal solution and potentially ignore the true decision boundary.

On the other hand, if we incorporate clusters in the data, and query based on the agreement and the size within each cluster, it would avoid overlook any data cluster and hence reduce sampling bias when our budgets are limited.

[ ]:

```

```