1

```
from cgi import print_arguments
import math
import numpy as np
from sklearn.decomposition import PCA
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import LinearSVC
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt


def generate_blob(mu,cov_root,N):
    # generate N samples with Gaussian distribution
    # mu: the mean vector, numpy array
    # cov_root: sqrt(covariance), numpy array
    D = len(mu)
    X = np.random.randn(N,D) @ cov_root.T + mu
    return X

def generate_data(N,ratio_pos=0.5,theta=0):
    # generate a mixture of two Gaussian blobs
    # N: number of samples
    # ratio_pos: P(y=1)
    # theta: the rotation angle for target domain, in degrees
    theta = theta/180*math.pi
    R = np.array([[math.cos(theta), math.sin(theta)], [-math.sin(theta),
math.cos(theta)]])
    mu_pos = R @ np.array([-3,-1])
    cov_pos = R @ np.sqrt([[10,0],[0,1]])
    mu_neg = R @ np.array([3,1])
    cov_neg = R @ np.sqrt([[10,0],[0,1]])

    N_pos = int(N*ratio_pos)
    N_neg = N-N_pos
    X_pos = generate_blob(mu_pos, cov_pos, N_pos)
    X_neg = generate_blob(mu_neg, cov_neg, N_neg)
    X = np.concatenate((X_pos,X_neg), axis=0)
    Y = -np.ones((N,))
    Y[:N_pos] = 1
    return X,Y

def get_clf(Q,k=31):
    # return a base classifier
    # Q: name of classifier, str
    # k: number of neighbors for KNN
    if Q=='LDA':
        clf = LinearDiscriminantAnalysis()
    elif Q=='KNN':
```

```
        clf = KNeighborsClassifier(n_neighbors=k)
    elif Q=='SVM':
        clf = LinearSVC(random_state=0, tol=1e-5)
    else:
        clf = None
        print(f'Non-implemented clf {Q}')
    return clf

def standardize(X, mu=None, std=None):
    # standardize the data X with provided mu and std.
    # if mu or std is None, calculate mu and std from X.
    # X: [num_of_sample, feat_dim]
    if mu is None or std is None:
        mu = np.mean(X, axis=0)
        std = np.std(X, axis=0, ddof=1)
    X = (X-mu)/std
    return X, mu, std

def display(res_sa, res_sl, Q, k=31):
    # draw accuracy vs. theta for SA and SL on the same plot
    # res_sa, res_sl: list of accuracies
    # Q: name of classifier, str
    # k: number of neighbors for KNN
    plt.figure(figsize=(3,3))
    plt.plot(range(0,181,30), res_sa, label='SA')
    plt.plot(range(0,181,30), res_sl, label='SL')
    plt.xlabel('theta'), plt.ylabel('accuracy'), plt.title(f'{Q}' if Q!='KNN' else
f'{Q},k={k}')
    plt.xticks(range(0,181,30))
    for i,theta in enumerate(range(0,181,30)):
        delta1 = 0.03 if res_sa[i]>=res_sl[i] else -0.05
        delta2 = -0.05 if res_sa[i]>=res_sl[i] else 0.03
        plt.text(theta, res_sa[i]+delta1, res_sa[i], size=8, color='b',
ha="center")
        plt.text(theta, res_sl[i]+delta2, res_sl[i], size=8, color='r',
ha="center")
    plt.grid('on')
    plt.legend()
    plt.show()
    return

def
experiment(Ns=1000,Nt=1000,Ntest=2000,Ntl=20,theta=0,ratio_pos=0.5,Q='LDA',k=31,st
andardized=True,sign_flip=True):
    # return SA testing acc and SL testing acc, under the setting specified in
arguments.
    # Ns: number of samples in source domain
    # Nt: number of samples in target domain
    # Nt: number of samples for testing in target domain
    # Ntl: number of labeled training samples in target domain
    # theta: the rotation angle for target domain, in degrees
    # ratio_pos: P(y=1)
    # Q: name of classifier, str
    # k: number of neighbors for KNN
```

```
    # standardized: standardize each feature dimension or not
    # sign_flip: flip the sign or not

    np.random.seed(2022)

    # TO-DO: draw Ds, DT, Dtest
    # notations: Xs, Ys, Xt, Yt, Xtest, Ytest

    Xs, Ys = generate_data(Ns, ratio_pos, 0)
    Xt, Yt = generate_data(Nt, ratio_pos, theta)

    # Xs_pos = []
    # Xs_neg = []
    # Xt_pos = []
    # Xt_neg = []
    # plt.figure()
    # for i in range(len(Xs)):
    #     if Ys[i] == 1:
    #         Xs_pos.append(Xs[i])
    #     elif Ys[i] == -1:
    #         Xs_neg.append(Xs[i])

    #     if Yt[i] == 1:
    #         Xt_pos.append(Xt[i])
    #     elif Yt[i] == -1:
    #         Xt_neg.append(Xt[i])

    # Xs_pos = np.array(Xs_pos)
    # Xs_neg = np.array(Xs_neg)
    # Xt_pos = np.array(Xt_pos)
    # Xt_neg = np.array(Xt_neg)

    # plt.scatter(Xt_pos[:, 0], Xt_pos[:, 1], color="red", marker="+")
    # plt.scatter(Xt_neg[:, 0], Xt_neg[:, 1], color="red", marker="_")
    # plt.scatter(Xs_pos[:, 0], Xs_pos[:, 1], color="blue", marker="+")
    # plt.scatter(Xs_neg[:, 0], Xs_neg[:, 1], color="blue", marker="_")

    # plt.legend(["Xs +1", "Xs -1", "Xt +1", "Xt -1"])
    # plt.title("theta = " + str(theta))
    # plt.show()

    # # TO-DO: standardize Ds, DT, Dtest if needed
    if standardized:
        Xs, _, _ = standardize(Xs)
        Xt, _, _ = standardize(Xt)

    # select Ntl points with labels, for sign flipping.
    # Xt, Yt are feature matrix and labels for target domain (after
standardization if there is)
    Xtl = None
    Ytl = None
    if sign_flip:
        Xtl = np.concatenate((Xt[:Ntl//2],Xt[-Ntl//2:]), axis=0)
        Ytl = np.concatenate((Yt[:Ntl//2],Yt[-Ntl//2:]), axis=0)
```

```
    Xtest, Ytest = generate_data(Ntest, ratio_pos, theta)

    # TO-DO: train and test SA
    # notations: score_SA (testing accuracy of SA, in [0,1])

    sa = SA_ee660(get_clf(Q), 2)
    sa.fit(Xs, Ys, Xt, Xtl, Ytl)
    score_SA = sa.score(Xtest, Ytest)

    # TO-DO: train and test SL
    # notations: score_SL (testing accuracy of SL, in [0,1])

    clf_SL = get_clf(Q)
    if sign_flip:
        clf_SL.fit(np.concatenate((Xs, Xtl), axis=0), np.concatenate((Ys, Ytl),
axis=0))
    else:
        clf_SL.fit(Xs, Ys)
    y_pred = clf_SL.predict(Xtest)
    score_SL = accuracy_score(Ytest, y_pred)

    print(f'theta={theta}: score_SA {score_SA}, score_SL {score_SL}')
    return score_SA, score_SL

class SA_ee660:
    def __init__(self, estimator, n_components):
        # n_components: feature dimension after PCA
        self.estimator = estimator
        self.n_components = n_components
        self.sign_mtx = np.array([[1,0],[0,1]])

    def fit(self, Xs, Ys, Xt, Xtl=None, Ytl=None):
        # training of SA
        # Xs: source domain feature matrix, [Ns, D_feat]
        # Ys: labels in source domain
        # Xt: target domain feature matrix, [Nt, D_feat]
        # Xtl, Ytl: labeled target data. Do sign flipping if they are not None.

        self.pca_src_ = PCA(self.n_components)
        self.pca_src_.fit(Xs)

        self.pca_tgt_ = PCA(self.n_components)
        self.pca_tgt_.fit(Xt)

        self.M_  = self.pca_src_.components_ @ self.pca_tgt_.components_.T

        Xs_tf = self.transform(Xs, domain="src")
        self.estimator.fit(Xs_tf, Ys)

        if Xtl is not None and Ytl is not None:
            besti, bestj = 1, 1
            bestacc = 0
            for i in (1,-1):
```

```
                for j in (1,-1):
                    self.sign_mtx = np.array([[i,0],[0,j]])
                    acc = self.score(Xtl, Ytl)
                    if acc>bestacc:
                        bestacc, besti, bestj = acc, i, j
            self.sign_mtx = np.array([[besti,0],[0,bestj]])
        return

    def score(self, X, y):
        # test trained SA on testing set X and y
        # return the testing accuracy, value in [0,1]
        X_tf = self.transform(X, domain="tgt")
        if hasattr(self.estimator, "score"):
            score = self.estimator.score(X_tf, y)
        elif hasattr(self.estimator, "evaluate"):
            if np.prod(X.shape) <= 10**8:
                score = self.estimator.evaluate(X_tf, y, batch_size=len(X))
            else:
                score = self.estimator.evaluate(X_tf, y)
            if isinstance(score, (tuple, list)):
                score = score[0]
        else:
            raise ValueError("Estimator does not implement score or evaluate
method")
        return score

    def transform(self, X, domain="tgt"):
        if domain in ["tgt"]:
            return self.pca_tgt_.transform(X) @ self.sign_mtx
        elif domain in ["src"]:
            return self.pca_src_.transform(X) @ self.M_
        else:
            raise ValueError("`domain `argument should be `tgt` or `src`, got,
%s"%domain)

if __name__=="__main__":
    # always call the experiment function to get results for a certain setting.
    # Do not remove or change the random seed setting in that function so that
your answers align with our solution.

    # LDA_SA = []
    # LDA_SL = []
    # SVM_SA = []
    # SVM_SL = []
    # KNN_SA = []
    # KNN_SL = []
    # for theta in range(0, 180 + 30, 30):
    #     score_SA, score_SL =
experiment(Ns=1000,Nt=1000,Ntest=2000,Ntl=20,theta=theta,ratio_pos=0.5,Q='LDA',sta
ndardized=True,sign_flip=False)
    #     LDA_SA.append(score_SA)
    #     LDA_SL.append(score_SL)

    #     score_SA, score_SL =
```

```
experiment(Ns=1000,Nt=1000,Ntest=2000,Ntl=20,theta=theta,ratio_pos=0.5,Q='SVM',sta
ndardized=True,sign_flip=False)
    #     SVM_SA.append(score_SA)
    #     SVM_SL.append(score_SL)

    #     score_SA, score_SL =
experiment(Ns=1000,Nt=1000,Ntest=2000,Ntl=20,theta=theta,ratio_pos=0.5,Q='KNN',sta
ndardized=True,sign_flip=False)
    #     KNN_SA.append(score_SA)
    #     KNN_SL.append(score_SL)

    # # example usage of the plotting function. Feel free to create your own.
    # display(LDA_SA, LDA_SL, Q='LDA')
    # display(SVM_SA, SVM_SL, Q='SVM')
    # display(KNN_SA, KNN_SL, Q='KNN')


    # LDA_SA = []
    # LDA_SL = []
    # SVM_SA = []
    # SVM_SL = []
    # KNN_SA = []
    # KNN_SL = []
    # for theta in range(0, 180 + 30, 30):
    #     score_SA, score_SL =
experiment(Ns=1000,Nt=1000,Ntest=2000,Ntl=20,theta=theta,ratio_pos=0.5,Q='LDA',sta
ndardized=True,sign_flip=True)
    #     LDA_SA.append(score_SA)
    #     LDA_SL.append(score_SL)

    #     score_SA, score_SL =
experiment(Ns=1000,Nt=1000,Ntest=2000,Ntl=20,theta=theta,ratio_pos=0.5,Q='SVM',sta
ndardized=True,sign_flip=True)
    #     SVM_SA.append(score_SA)
    #     SVM_SL.append(score_SL)

    #     score_SA, score_SL =
experiment(Ns=1000,Nt=1000,Ntest=2000,Ntl=20,theta=theta,ratio_pos=0.5,Q='KNN',sta
ndardized=True,sign_flip=True)
    #     KNN_SA.append(score_SA)
    #     KNN_SL.append(score_SL)

    # # example usage of the plotting function. Feel free to create your own.
    # display(LDA_SA, LDA_SL, Q='LDA')
    # display(SVM_SA, SVM_SL, Q='SVM')
    # display(KNN_SA, KNN_SL, Q='KNN')

    # plt.figure()
    # for Ntl in [0, 6, 20, 50, 100]:
    #     LDA_SA = []
    #     LDA_SL = []
    #     for theta in range(0, 180 + 30, 30):
    #         score_SA, score_SL =
experiment(Ns=1000,Nt=1000,Ntest=2000,Ntl=Ntl,theta=theta,ratio_pos=0.5,Q='LDA',st
```

```
andardized=True,sign_flip=True)
    #         LDA_SA.append(score_SA)
    #         LDA_SL.append(score_SL)
    #     print(LDA_SA, LDA_SL)
    #     plt.subplot(211)
    #     plt.plot(range(0,181,30), LDA_SA, label="SA " + str(Ntl))
    #     plt.subplot(212)
    #     plt.plot(range(0,181,30), LDA_SL, label="SL " + str(Ntl))
    # plt.legend()
    # plt.show()

    score_SA, score_SL =
experiment(Ns=1000,Nt=1000,Ntest=2000,Ntl=20,theta=30,ratio_pos=0.5,Q='LDA',standa
rdized=False,sign_flip=True)
    print(score_SA, score_SL)
    score_SA, score_SL =
experiment(Ns=1000,Nt=1000,Ntest=2000,Ntl=20,theta=30,ratio_pos=0.5,Q='LDA',standa
rdized=True,sign_flip=True)
    print(score_SA, score_SL)
    score_SA, score_SL =
experiment(Ns=1000,Nt=1000,Ntest=2000,Ntl=20,theta=30,ratio_pos=0.5,Q='LDA',standa
rdized=False,sign_flip=False)
    print(score_SA, score_SL)
    score_SA, score_SL =
experiment(Ns=1000,Nt=1000,Ntest=2000,Ntl=20,theta=30,ratio_pos=0.5,Q='LDA',standa
rdized=True,sign_flip=False)
    print(score_SA, score_SL)
```

2

```
import numpy as np
import math
Nt = 1000
Ns = 100000
N = Ns + Nt
beta = Nt / N

for alpha in [0.1, 0.5, 0.9]:
  eab = 2 * (1 - alpha) * (0.5 * 0.1) \
  + 4 * math.sqrt(alpha ** 2 / beta + (1 - alpha) ** 2 / (1 - beta)) \
  * math.sqrt(2 / N * 10 * np.log(2 * (N + 1)) + 2 / N * np.log(8 / 0.1))

  print(eab)

import numpy as np
import math
```

```
import matplotlib.pyplot as plt

def get_eab(alpha, beta, N):
  eab = 2 * (1 - alpha) * (0.5 * 0.1) \
  + 4 * math.sqrt(alpha ** 2 / beta + (1 - alpha) ** 2 / (1 - beta)) \
  * math.sqrt(2 / N * 10 * np.log(2 * (N + 1)) + 2 / N * np.log(8 / 0.1))
  return eab

plt.figure(figsize=(8, 6), dpi=80)
Ns = 1000
for Nt in [10,100,1000,10000]:
  N = Ns + Nt
  beta = Nt / N
  alpha = np.linspace(0, 1, 201)
  eab = [get_eab(a, beta, N) for a in alpha]
  plt.plot(alpha, eab, label="Nt =" + str(Nt) + " beta = " + str(beta))
plt.legend()
plt.xlabel("alpha")
plt.ylabel("epsilon")
plt.show()

import numpy as np
import math
import matplotlib.pyplot as plt

def get_eab(alpha, beta, N):
  eab = 2 * (1 - alpha) * (0.5 * 0.1) \
  + 4 * math.sqrt(alpha ** 2 / beta + (1 - alpha) ** 2 / (1 - beta)) \
  * math.sqrt(2 / N * 10 * np.log(2 * (N + 1)) + 2 / N * np.log(8 / 0.1))
  return eab

plt.figure(figsize=(8, 6), dpi=80)
Nt = 100
for Ns in [10,100,1000,10000]:
  N = Ns + Nt
  beta = Nt / N
  alpha = np.linspace(0, 1, 201)
  eab = [get_eab(a, beta, N) for a in alpha]
  plt.plot(alpha, eab, label="Ns =" + str(Ns) + " beta = " + str(beta))
plt.legend()
plt.xlabel("alpha")
plt.ylabel("epsilon")
plt.show()

import numpy as np
import math
import matplotlib.pyplot as plt

def get_eab(alpha, beta, N):
  eab = 2 * (1 - alpha) * (0.5 * 0.1) \
  + 4 * math.sqrt(alpha ** 2 / beta + (1 - alpha) ** 2 / (1 - beta)) \
  * math.sqrt(2 / N * 10 * np.log(2 * (N + 1)) + 2 / N * np.log(8 / 0.1))
  return eab
```

```
plt.figure(figsize=(8, 6), dpi=80)

for beta in [0.01,0.1,0.5]:
  alpha = 0.5
  N = np.linspace(1000, 100000, 201)
  eab = [get_eab(alpha, beta, n) for n in N]
  plt.plot(N, eab, label="alpha =" + str(alpha) + " beta = " + str(beta))
plt.legend()
plt.xlabel("N")
plt.ylabel("epsilon")
plt.show()

import numpy as np
import math
import matplotlib.pyplot as plt

def get_eab(alpha, beta, N):
  eab = 2 * (1 - alpha) * (0.5 * 0.1) \
  + 4 * math.sqrt(alpha ** 2 / beta + (1 - alpha) ** 2 / (1 - beta)) \
  * math.sqrt(2 / N * 10 * np.log(2 * (N + 1)) + 2 / N * np.log(8 / 0.1))
  return eab

plt.figure(figsize=(8, 6), dpi=80)

for beta in [0.01,0.1,0.5]:
  alpha = beta
  N = np.linspace(1000, 100000, 201)
  eab = [get_eab(alpha, beta, n) for n in N]
  plt.plot(N, eab, label="alpha =" + str(alpha) + " beta = " + str(beta))
plt.legend()
plt.xlabel("N")
plt.ylabel("epsilon")
plt.show()
```