**Codice MLPR**

* Bayes decision model

def compute\_optimal\_Bayes\_binary\_llr(llr, prior, Cfn, Cfp):  
 th = -numpy.log((prior \* Cfn) / ((1 - prior) \* Cfp))  
 return numpy.int32(llr > th)  
  
  
def compute\_confusion\_matrix(predictedLabels, classLabels):  
 nClasses = classLabels.max() + 1  
 M = numpy.zeros((nClasses, nClasses), dtype=numpy.int32)  
 for i in range(classLabels.size):  
 M[predictedLabels[i], classLabels[i]] += 1  
 return M  
  
  
def computeDCF\_Binary(confusionMatrix, prior, Cfn, Cfp, normalize=False):  
 Pfn = confusionMatrix[0, 1] / (confusionMatrix[0, 1] + confusionMatrix[1, 1])  
 Pfp = confusionMatrix[1, 0] / (confusionMatrix[1, 0] + confusionMatrix[0, 0])  
 bayesError = prior \* Pfn \* Cfn + (1 - prior) \* Pfp \* Cfp  
 if normalize:  
 return bayesError / numpy.minimum(prior \* Cfn, (1 - prior) \* Cfp)  
 return bayesError  
  
  
def compute\_Pfn\_Pfp\_allThresholds(llr, classLabels):  
 llrSorter = numpy.argsort(llr)  
 llrSorted = llr[llrSorter] *# We sort the llrs* classLabelsSorted = classLabels[llrSorter] *# we sort the labels so that they are aligned to the llrs* Pfp = []  
 Pfn = []  
  
 nTrue = (classLabelsSorted == 1).sum()  
 nFalse = (classLabelsSorted == 0).sum()  
 nFalseNegative = 0 *# With the left-most theshold all samples are assigned to class 1* nFalsePositive = nFalse  
  
 Pfn.append(nFalseNegative / nTrue)  
 Pfp.append(nFalsePositive / nFalse)  
  
 for idx in range(len(llrSorted)):  
 if classLabelsSorted[idx] == 1:  
 nFalseNegative += 1 *# Increasing the threshold we change the assignment for this llr from 1 to 0, so we increase the error rate* if classLabelsSorted[idx] == 0:  
 nFalsePositive -= 1 *# Increasing the threshold we change the assignment for this llr from 1 to 0, so we decrease the error rate* Pfn.append(nFalseNegative / nTrue)  
 Pfp.append(nFalsePositive / nFalse)  
  
 *# The last values of Pfn and Pfp should be 1.0 and 0.0, respectively  
 # Pfn.append(1.0) # Corresponds to the numpy.inf threshold, all samples are assigned to class 0  
 # Pfp.append(0.0) # Corresponds to the numpy.inf threshold, all samples are assigned to class 0* llrSorted = numpy.concatenate([-numpy.array([numpy.inf]), llrSorted])  
  
 *# In case of repeated scores, we need to "compact" the Pfn and Pfp arrays (i.e., we need to keep only the value that corresponds to an actual change of the threshold* PfnOut = []  
 PfpOut = []  
 thresholdsOut = []  
 for idx in range(len(llrSorted)):  
 if idx == len(llrSorted) - 1 or llrSorted[idx + 1] != llrSorted[  
 idx]: *# We are indeed changing the threshold, or we have reached the end of the array of sorted scores* PfnOut.append(Pfn[idx])  
 PfpOut.append(Pfp[idx])  
 thresholdsOut.append(llrSorted[idx])  
  
 return numpy.array(PfnOut), numpy.array(PfpOut), numpy.array(  
 thresholdsOut) *# we return also the corresponding thresholds*def compute\_minDCF\_binary(llr, classLabels, prior, Cfn, Cfp, returnThreshold=False):  
 Pfn, Pfp, th = compute\_Pfn\_Pfp\_allThresholds(llr, classLabels)  
 minDCF = (prior \* Cfn \* Pfn + (1 - prior) \* Cfp \* Pfp) / numpy.minimum(prior \* Cfn, (  
 1 - prior) \* Cfp) *# We exploit broadcasting to compute all DCFs for all thresholds* idx = numpy.argmin(minDCF)  
 if returnThreshold:  
 return minDCF[idx], th[idx]  
 else:  
 return minDCF[idx]

* Dim reduction
  + PCA

def PCA\_function(D, m):  
 mu = 0  
 C = 0  
 mu = D.mean(axis=1) *# è un vettore, cioè una matrice riga* DC = D - ut.vcol(mu) *# per centrare i dati* C = np.dot(DC, DC.T) / float(D.shape[1]) *# matrice di covarianza* s, U = np.linalg.eigh(C)  
 P = U[:, ::-1][:, 0:m] *# matrice di proiezione  
 # print("P", P)  
 # U,s,Vh=np.linalg.svd(C)  
 # P=U[:,0:m]#matrice di proiezione* return s, P

* + LDA

def compute\_Sv\_Sb(D, L):  
 num\_classes = L.max() + 1  
 *# separate the data into classes* D\_c = [D[:, L == i] for i in range(num\_classes)]  
 *# number of elements for each class* n\_c = [D\_c[i].shape[1] for i in range(num\_classes)]  
  
 *# mean for all the data* mu = D.mean(1)  
 mu = ut.vcol(mu)  
  
 *# mean for each class* mu\_c = [ut.vcol(D\_c[i].mean(1)) for i in range(len(D\_c))]  
  
 S\_w, S\_b = 0, 0  
 for i in range(num\_classes):  
 Dc = D\_c[i] - mu\_c[i]  
 C\_i = np.dot(Dc, Dc.T) / Dc.shape[1]  
 S\_w += n\_c[i] \* C\_i  
 diff = mu\_c[i] - mu  
 S\_b += n\_c[i] \* np.dot(diff, diff.T)  
  
 S\_w /= D.shape[1]  
 S\_b /= D.shape[1]  
 return S\_w, S\_b  
  
  
def LDA\_function(D, L, m):  
 *# compute Sw and Sb  
 # print("D", D)  
 # print("L", L)* Sw, Sb = compute\_Sv\_Sb(D, L)  
 *# print("Sw", Sw)  
 # print("Sb", Sb)  
 # compute the eigenvalues and eigenvectors of Sw^-1\*Sb* s, U = scipy.linalg.eigh(Sb, Sw)  
 W = U[:, ::-1][:, 0:m]  
  
 return W

* Gaussian density

def compute\_mu\_C(D):  
 mu = ut.vcol(D.mean(1))  
 C = ((D - mu) @ (D - mu).T) / float(D.shape[1])  
 return mu, C  
  
  
def logpdf\_GAU\_ND(X, mu, C):  
 Y = []  
 *# get the number of features* N = X.shape[0]  
 *# for each input data* for x in X.T:  
 x = ut.vcol(x)  
 *# compute the constant term* const = N \* np.log(2 \* np.pi) *# compute the second term* logC = np.linalg.slogdet(C)[1] *# compute the third term* mult = np.dot(np.dot((x - mu).T, np.linalg.inv(C)), (x - mu))[0, 0]  
 *# append the result of the function for this input data* Y.append(-0.5 \* (const + logC + mult))  
  
 *# return the result array* return np.array(Y)  
  
  
def predict\_labels(DVAL, TH, LLR, class1, class2):  
 PVAL = np.zeros(DVAL.shape[1], dtype=np.int32)  
 PVAL[LLR >= TH] = class2  
 PVAL[LLR < TH] = class1  
 return PVAL  
  
  
def log\_likelihood(X, mu, C):  
 return logpdf\_GAU\_ND(X, mu, C).sum()

* Gaussian model

def compute\_log\_likelihood(D, hParams):  
 S = np.zeros((len(hParams), D.shape[1]))  
 for lab in range(S.shape[0]):  
 S[lab, :] = gd.logpdf\_GAU\_ND(D, hParams[lab][0], hParams[lab][1])  
 return S  
  
  
def compute\_mu\_c\_MVG(D, L):  
 labelSet = set(L)  
 hParams = {}  
 for lab in labelSet:  
 DX = D[:, L == lab]  
 hParams[lab] = gd.compute\_mu\_C(DX)  
 return hParams  
  
  
def compute\_mu\_C\_Tied(D, L):  
 labelSet = set(L)  
 hParams = {}  
 hMeans = {}  
 CGlobal = 0  
 for lab in labelSet:  
 DX = D[:, L == lab]  
 mu, C\_class = gd.compute\_mu\_C(DX)  
 *# DX.shape[1] è il numero di campioni di quella classe* CGlobal += C\_class \* DX.shape[1]  
 hMeans[lab] = mu  
 *# qui viene diviso per il numero totale di campioni* CGlobal = CGlobal / D.shape[1]  
 *# viene semplicemente assegnato lo stesso valore di covarianza a tutte le classi* for lab in labelSet:  
 hParams[lab] = (hMeans[lab], CGlobal)  
 return hParams  
  
  
def compute\_mu\_C\_Naive(D, L):  
 labelSet = set(L)  
 hParams = {}  
 for lab in labelSet:  
 DX = D[:, L == lab]  
 mu, C = gd.compute\_mu\_C(DX)  
 *# C moltiplicato per la matrice identità* hParams[lab] = (mu, C \* np.eye(D.shape[0]))  
 return hParams  
  
  
def compute\_logPosterior(S\_logLikelihood, v\_prior):  
 *# probabilità congiunta* SJoint = S\_logLikelihood + ut.vcol(np.log(v\_prior))  
 *# probabilita marginale che è uguale al prodotto delle probabilità congiunte* SMarginal = ut.vrow(scipy.special.logsumexp(SJoint, axis=0))  
 *# probabilità a posteriori, sottrai la probabilità marginale dalla probabilità congiunta in modo che tutti abbiamo probabilità massimo 1* SPost = SJoint - SMarginal  
 return SPost  
  
  
def calculate\_MVG(DTR, LTR, DVAL, LVAL):  
 hParams\_MVG = compute\_mu\_c\_MVG(DTR, LTR)  
 LLR = gd.logpdf\_GAU\_ND(DVAL, hParams\_MVG[1][0], hParams\_MVG[1][1]) - gd.logpdf\_GAU\_ND(DVAL, hParams\_MVG[0][0],  
 hParams\_MVG[0][1])  
 PVAL = gd.predict\_labels(DVAL=DVAL, TH=0, LLR=LLR, class1=0, class2=1)  
 print("MVG 2-Class problem - Error rate: {:.6f}%".format(error.error\_rate(PVAL, LVAL)))  
 return LLR  
  
  
def calculate\_Tied(DTR, LTR, DVAL, LVAL):  
 hParams\_Tied = compute\_mu\_C\_Tied(DTR, LTR)  
 LLR = gd.logpdf\_GAU\_ND(DVAL, hParams\_Tied[1][0], hParams\_Tied[1][1]) - gd.logpdf\_GAU\_ND(DVAL, hParams\_Tied[0][0],  
 hParams\_Tied[0][1])  
 PVAL = gd.predict\_labels(DVAL=DVAL, TH=0, LLR=LLR, class1=0, class2=1)  
 print("Tied 2-Class problem - Error rate: {:.6f}%".format(error.error\_rate(PVAL, LVAL)))  
 return LLR  
  
  
def calculate\_Naive(DTR, LTR, DVAL, LVAL):  
 hParams\_Naive = compute\_mu\_C\_Naive(DTR, LTR)  
 LLR = gd.logpdf\_GAU\_ND(DVAL, hParams\_Naive[1][0], hParams\_Naive[1][1]) - gd.logpdf\_GAU\_ND(DVAL, hParams\_Naive[0][0],  
 hParams\_Naive[0][1])  
 PVAL = gd.predict\_labels(DVAL=DVAL, TH=0, LLR=LLR, class1=0, class2=1)  
 print("Naive 2-Class problem - Error rate: {:.6f}%".format(error.error\_rate(PVAL, LVAL)))  
 return LLR

def correlation(DTR, LTR):  
 hParams\_MVG = compute\_mu\_c\_MVG(DTR, LTR)  
  
 C0 = hParams\_MVG[0][1]  
 C1 = hParams\_MVG[1][1]  
  
 print("C0\n", C0)  
 print("C1\n", C1)  
  
 Corr0 = C0 / (ut.vcol(C0.diagonal() \*\* 0.5) \* ut.vrow(C0.diagonal() \*\* 0.5))  
 Corr1 = C1 / (ut.vcol(C1.diagonal() \*\* 0.5) \* ut.vrow(C1.diagonal() \*\* 0.5))  
  
 heatmap(DTR, LTR, plt, "Correlation")  
 plt.show()  
  
 for i in range(Corr0.shape[0]):  
 row\_Corr0 = ' '.join('{:<10.2f}'.format(x) for x in Corr0[i])  
 print("Corr0[{}]: {} ".format(i, row\_Corr0))  
 print("\n")  
 for i in range(Corr1.shape[0]):  
 row\_Corr1 = ' '.join('{:<10.2f}'.format(x) for x in Corr1[i])  
 print(" Corr1[{}]: {}".format(i, row\_Corr1))  
  
 return Corr0, Corr1

* GMM

class GMM:  
 def \_\_init\_\_(self, alpha=0.1, n0Components=2, n1Components=2, psi=0.01, covType='Full'):  
 self.alpha = alpha  
 self.n0Components = n0Components  
 self.n1Components = n1Components  
 self.psi = psi  
 self.covType = covType  
  
 def \_\_logpdf\_GAU\_ND(self, X, mu, C):  
 invC = np.linalg.inv(C)  
 \_, log\_abs\_detC = np.linalg.slogdet(C)  
 M = X.shape[0]  
 return - M / 2 \* np.log(2 \* np.pi) - 0.5 \* log\_abs\_detC - 0.5 \* ((X - mu) \* np.dot(invC, X - mu)).sum(0)  
  
 def logpdf\_GMM(self, X, gmm):  
 S = np.zeros((len(gmm), X.shape[1]))  
  
 for g in range(len(gmm)):  
 (w, mu, C) = gmm[g]  
 S[g, :] = self.\_\_logpdf\_GAU\_ND(X, mu, C) + np.log(w)  
  
 logdens = scipy.special.logsumexp(S, axis=0)  
 return S, logdens  
  
 def GMM\_algorithm\_EM(self, X, gmm, psi=0.01, cov='Full'):  
 thNew = None  
 thOld = None  
 N = X.shape[1]  
 D = X.shape[0]  
  
 while thOld == None or thNew - thOld > 1e-6: *# finchè non diverge* thOld = thNew  
 logSj, logSjMarg = self.logpdf\_GMM(X, gmm)  
 thNew = np.sum(logSjMarg) / N  
  
 P = np.exp(logSj - logSjMarg) *# Responsabilità che è uguale alla probabilita a posteriori* if cov == 'Diag':  
 newGmm = []  
 for i in range(len(gmm)):  
 gamma = P[i, :]  
 Z = gamma.sum()  
 F = (gamma.reshape(1, -1) \* X).sum(1)  
 S = np.dot(X, (gamma.reshape(1, -1) \* X).T)  
 w = Z / N  
 mu = (F / Z).reshape(-1, 1)  
 sigma = S / Z - np.dot(mu, mu.T)  
 sigma \*= np.eye(sigma.shape[0])  
 U, s, \_ = np.linalg.svd(sigma)  
 s[s < psi] = psi  
 sigma = np.dot(U, s.reshape(-1, 1) \* U.T)  
 newGmm.append((w, mu, sigma))  
 gmm = newGmm  
  
 elif cov == 'Tied':  
 newGmm = []  
 sigmaTied = np.zeros((D, D))  
 for i in range(len(gmm)):  
 gamma = P[i, :]  
 Z = gamma.sum()  
 F = (gamma.reshape(1, -1) \* X).sum(1)  
 S = np.dot(X, (gamma.reshape(1, -1) \* X).T)  
 w = Z / N  
 mu = (F / Z).reshape(-1, 1)  
 sigma = S / Z - np.dot(mu, mu.T)  
 sigmaTied += Z \* sigma  
 newGmm.append((w, mu))  
 gmm = newGmm  
 sigmaTied /= N  
 U, s, \_ = np.linalg.svd(sigmaTied)  
 s[s < psi] = psi  
 sigmaTied = np.dot(U, s.reshape(-1, 1) \* U.T)  
  
 newGmm = []  
 for i in range(len(gmm)):  
 (w, mu) = gmm[i]  
 newGmm.append((w, mu, sigmaTied))  
  
 gmm = newGmm  
  
 elif cov == 'TiedDiag':  
 newGmm = []  
 sigmaTied = np.zeros((D, D))  
 for i in range(len(gmm)):  
 gamma = P[i, :]  
 Z = gamma.sum()  
 F = (gamma.reshape(1, -1) \* X).sum(1)  
 S = np.dot(X, (gamma.reshape(1, -1) \* X).T)  
 w = Z / N  
 mu = (F / Z).reshape(-1, 1)  
 sigma = S / Z - np.dot(mu, mu.T)  
 sigmaTied += Z \* sigma  
 newGmm.append((w, mu))  
 gmm = newGmm  
 sigmaTied /= N  
 sigmaTied \*= np.eye(sigma.shape[0])  
 U, s, \_ = np.linalg.svd(sigmaTied)  
 s[s < psi] = psi  
 sigmaTied = np.dot(U, s.reshape(-1, 1) \* U.T)  
  
 newGmm = []  
 for i in range(len(gmm)):  
 (w, mu) = gmm[i]  
 newGmm.append((w, mu, sigmaTied))  
  
 gmm = newGmm  
  
 else:  
 newGmm = []  
 *# prendi un componente alla volta* for i in range(len(gmm)):  
 gamma = P[i, :]  
 *# calola le statistiche* Z = gamma.sum()  
 F = (gamma.reshape(1, -1) \* X).sum(1)  
 S = np.dot(X, (gamma.reshape(1, -1) \* X).T)  
  
 w = Z / N  
 mu = (F / Z).reshape(-1, 1)  
 sigma = S / Z - np.dot(mu, mu.T)  
 U, s, \_ = np.linalg.svd(sigma)  
 s[s < psi] = psi  
 sigma = np.dot(U, s.reshape(-1, 1) \* U.T)  
 newGmm.append((w, mu, sigma))  
 gmm = newGmm  
  
 return gmm, thNew  
  
 def GMM\_algorithm\_LBG(self, X, alpha, nComponents, psi=0.01, covType='Full'):  
 mean = X.mean(axis=1).reshape(-1, 1)  
 cov = 1 / X.shape[1] \* np.dot(X - mean, (X - mean).T)  
 gmm = [(1, mean, cov)]  
  
 while len(gmm) <= nComponents:  
 gmm, final\_log = self.GMM\_algorithm\_EM(X, gmm, psi, covType)  
  
 if len(gmm) == nComponents:  
 break  
  
 newGmm = []  
 for i in range(len(gmm)):  
 (w, mu, sigma) = gmm[i]  
 U, s, Vh = np.linalg.svd(sigma)  
 d = U[:, 0:1] \* s[0] \*\* 0.5 \* alpha  
  
 newGmm.append((w / 2, mu - d, sigma))  
 newGmm.append((w / 2, mu + d, sigma))  
 gmm = newGmm  
 return gmm, final\_log  
  
 def train(self, Dtrain, Ltrain):  
 self.Dtrain\_c0 = Dtrain[:, Ltrain == 0]  
 self.Dtrain\_c1 = Dtrain[:, Ltrain == 1]  
 self.gmm\_c0, \_ = self.GMM\_algorithm\_LBG(self.Dtrain\_c0, self.alpha, self.n0Components, self.psi, self.covType)  
 self.gmm\_c1, \_ = self.GMM\_algorithm\_LBG(self.Dtrain\_c1, self.alpha, self.n1Components, self.psi, self.covType)  
 return self  
  
 def predict(self, Dtest, labels=False):  
 \_, llr\_0 = self.logpdf\_GMM(Dtest, self.gmm\_c0)  
 \_, llr\_1 = self.logpdf\_GMM(Dtest, self.gmm\_c1)  
 if labels:  
 S = np.vstack([llr\_0.reshape(1, -1), llr\_1.reshape(1, -1)])  
 return np.argmax(S, axis=0)  
 else:  
 return llr\_1 - llr\_0

* LLR

class LinearLogisticRegression:  
 def \_\_init\_\_(self, lbd, prior\_weighted=False, prior=0.5):  
 self.lbd = lbd  
 self.prior\_weighted = prior\_weighted  
 self.prior = prior  
  
 def \_\_logreg\_obj(self, v):  
 w, b = v[0:-1], v[-1]  
 ZTR = 2 \* self.LTR - 1  
 reg = 0.5 \* self.lbd \* np.linalg.norm(w) \*\* 2  
 exp = (np.dot(w.T, self.DTR) + b)  
 avg\_risk = (np.logaddexp(0, -exp \* ZTR)).mean()  
 return reg + avg\_risk  
  
 def \_\_logreg\_obj\_prior\_weighted(self, v):  
 w, b = v[0:-1], v[-1]  
 ZTR = 2 \* self.LTR - 1  
  
 wTrue = self.prior / (ZTR > 0).sum()  
 wFalse = (1 - self.prior) / (ZTR < 0).sum()  
  
 reg = 0.5 \* self.lbd \* np.linalg.norm(w) \*\* 2  
 exp = (np.dot(w.T, self.DTR) + b)  
 avg\_risk\_0 = (np.logaddexp(0, -exp[self.LTR == 0] \* ZTR[self.LTR == 0]) \* wFalse).sum()  
 avg\_risk\_1 = (np.logaddexp(0, -exp[self.LTR == 1] \* ZTR[self.LTR == 1]) \* wTrue).sum()  
 return reg + avg\_risk\_0 + avg\_risk\_1  
  
 def trainLogReg(self, DTR, LTR):  
 self.DTR = DTR  
 self.LTR = LTR  
 x0 = np.zeros(DTR.shape[0] + 1)  
 self.xf = scipy.optimize.fmin\_l\_bfgs\_b(  
 func=self.\_\_logreg\_obj\_prior\_weighted if self.prior\_weighted else self.\_\_logreg\_obj,  
 x0=x0,  
 approx\_grad=True,  
 *# iprint=0* )[0]  
 return self.xf  
  
 def predict(self, DVAL, label=False, threshold=0):  
 w = self.xf[:-1]  
 b = self.xf[-1]  
 sval = np.dot(w.T, DVAL) + b  
 if label:  
 return np.int32(sval > threshold)  
 else:  
 return sval

* QLR

class QuadraticLogisticRegression:  
 def \_\_init\_\_(self, lbd, prior\_weighted=False, prior=0.5):  
 self.lbd = lbd  
 self.prior\_weighted = prior\_weighted  
 self.prior = prior  
  
 def \_\_compute\_zi(self, ci):  
 return 2 \* ci - 1  
  
 def \_\_logreg\_obj(self, v):  
 w, b = v[0:-1], v[-1]  
 z = 2 \* self.Ltrain - 1  
 exp = (np.dot(w.T, self.Dtrain\_exp) + b)  
 reg = 0.5 \* self.lbd \* np.linalg.norm(w) \*\* 2  
 avg\_risk = (np.logaddexp(0, -exp \* z)).mean()  
 return reg + avg\_risk  
  
 def \_\_logreg\_obj\_prior\_weighted(self, v):  
 w, b = v[0:-1], v[-1]  
 z = 2 \* self.Ltrain - 1  
 reg = 0.5 \* self.lbd \* np.linalg.norm(w) \*\* 2  
 exp = (np.dot(w.T, self.Dtrain\_exp) + b)  
 avg\_risk\_0 = np.logaddexp(0, -exp[self.Ltrain == 0] \* z[self.Ltrain == 0]).mean() \* (1 - self.prior)  
 avg\_risk\_1 = np.logaddexp(0, -exp[self.Ltrain == 1] \* z[self.Ltrain == 1]).mean() \* self.prior  
 return reg + avg\_risk\_0 + avg\_risk\_1  
  
 def train(self, Dtrain, Ltrain):  
 self.Dtrain = Dtrain  
 self.Ltrain = Ltrain  
 self.F = Dtrain.shape[0]  
 self.K = len(set(Ltrain))  
 self.N = Dtrain.shape[1]  
 self.Dtrain\_exp = self.\_\_expand\_features\_space(Dtrain)  
 obj\_function = self.\_\_logreg\_obj if self.prior\_weighted is False else self.\_\_logreg\_obj\_prior\_weighted  
 self.x, f, d = scipy.optimize.fmin\_l\_bfgs\_b(func=obj\_function,  
 x0=np.zeros(self.Dtrain\_exp.shape[0] + 1),  
 approx\_grad=True,  
 *# iprint=0* )  
 return self.x  
  
 def \_\_vectorize(self, M):  
 M\_vec = np.hstack(M).reshape(-1, 1)  
 return M\_vec  
  
 def \_\_expand\_features\_space(self, D):  
 D\_exp = np.zeros(shape=(self.F \* self.F + self.F, D.shape[1]))  
 for i in range(D.shape[1]):  
 xi = D[:, i:i + 1]  
 D\_exp[:, i:i + 1] = np.vstack((self.\_\_vectorize(np.dot(xi, xi.T)), xi))  
 return D\_exp  
  
 def predict(self, Dtest, label=True):  
 w, b = self.x[0:-1], self.x[-1]  
 Dtest\_exp = self.\_\_expand\_features\_space(Dtest)  
 S = np.zeros((Dtest\_exp.shape[1]))  
 for i in range(Dtest\_exp.shape[1]):  
 xi = Dtest\_exp[:, i:i + 1]  
 s = np.dot(w.T, xi) + b  
 S[i] = s  
 if label:  
 LP = S > 0  
 return LP  
 else:  
 return S  
  
 def predictThreshold(self, Dtest, threshold):  
 w = self.x[:-1]  
 b = self.x[-1]  
 sval = np.dot(w.T, self.\_\_expand\_features\_space(Dtest)) + b  
  
 return np.int32(sval > threshold)  
  
 def calculateS(self, DVAL):  
 w = self.x[:-1]  
 b = self.x[-1]  
 sval = np.dot(w.T, self.\_\_expand\_features\_space(DVAL)) + b  
 return sval  
  
 def compute\_minDCF\_actDCF(self, LVAL, DVAL, pi\_emp, Cfn=1, Cfp=1, prior=0.5):  
 w = self.x[:-1]  
 b = self.x[-1]  
 sval = np.dot(w.T, self.\_\_expand\_features\_space(DVAL)) + b  
 predictedLabels = np.int32(sval > 0)  
 error\_rate = e.error\_rate(predictedLabels, LVAL)  
 print("Error rate:", error\_rate, "%")  
 sValLLR = sval - np.log(pi\_emp / (1 - pi\_emp))  
 th = -np.log((prior \* Cfn) / ((1 - prior) \* Cfp))  
 predictedLabels = np.int32(sval > th)  
 minDCF = bdm.compute\_minDCF\_binary(sValLLR, LVAL, prior, Cfn, Cfp)  
 confusionMatrix = bdm.compute\_confusion\_matrix(predictedLabels, LVAL)  
 actDCF = bdm.computeDCF\_Binary(confusionMatrix, prior, Cfn, Cfp, normalize=True)  
 print("minDCF:", minDCF)  
 print("actDCF:", actDCF)  
 return minDCF, actDCF

* SVM

class SVM:  
 def \_\_init\_\_(self, hparams, kernel=None, prior=0):  
 self.kernelType = kernel  
 self.C = hparams['C']  
 self.K = hparams['K']  
 self.eps = hparams.get('eps')  
 self.gamma = hparams.get('gamma')  
 self.c = hparams.get('c')  
 self.d = hparams.get('d')  
 self.prior = prior  
  
 def \_\_LDc\_obj(self, alpha):  
 ones\_matrix = np.ones((alpha.shape[0], 1))  
 t = 0.5 \* np.dot(np.dot(alpha.T, self.H), alpha) - np.dot(alpha.T, ones\_matrix).sum(), (  
 np.dot(self.H, alpha) - 1).flatten()  
 return t  
  
 def \_\_polynomial\_kernel(self, X1, X2):  
 ker = (np.dot(X1.T, X2) + self.c) \*\* self.d + self.K \*\* 2  
 return ker  
  
 def \_\_RBF\_kernel(self, X1, X2):  
 *# x = np.repeat(X1, X2.shape[1], axis=1)  
 # y = np.tile(X2, X1.shape[1])  
 # ker = np.exp(  
 # -self.gamma \* np.linalg.norm(x - y, axis=0).reshape(X1.shape[1], X2.shape[1]) \*\* 2) + self.K \*\* 2  
 # return ker* D1Norms = (X1 \*\* 2).sum(0)  
 D2Norms = (X2 \*\* 2).sum(0)  
 Z = vcol(D1Norms) + vrow(D2Norms) - 2 \* np.dot(X1.T, X2)  
 return np.exp(-self.gamma \* Z)  
  
 def train(self, Dtrain, Ltrain):  
 self.Dtrain = Dtrain  
 self.Ltrain = Ltrain  
 self.N = Dtrain.shape[1]  
 self.Ltrain\_z = self.Ltrain \* 2 - 1  
 self.Ltrain\_z\_matrix = self.Ltrain\_z.reshape(-1, 1) \* self.Ltrain\_z.reshape(1, -1)  
 self.bounds = [(0, self.C) for i in self.Ltrain]  
  
 if self.prior != 0:  
 empP = (self.Ltrain == 1).sum() / len(self.Ltrain)  
 self.bounds[self.Ltrain == 1] = (0, self.C \* self.prior / empP)  
 self.bounds[self.Ltrain == 0] = (0, self.C \* (1 - self.prior) / (1 - empP))  
  
 if self.kernelType is not None:  
 if self.kernelType == 'Polynomial':  
 ker = self.\_\_polynomial\_kernel(self.Dtrain, self.Dtrain)  
 elif self.kernelType == 'RBF':  
 ker = self.\_\_RBF\_kernel(self.Dtrain, self.Dtrain)  
 else:  
 return  
 self.H = self.Ltrain\_z\_matrix \* ker  
 else:  
 *# self.expandedD = np.vstack((Dtrain, self.K \* np.ones(self.N)))* self.expandedD = np.vstack([Dtrain, np.ones((1, Dtrain.shape[1])) \* self.K])  
 *# G = np.dot(self.expandedD.T, self.expandedD)  
 # self.H = G \* self.Ltrain\_z\_matrix* self.H = np.dot(self.expandedD.T, self.expandedD) \* self.Ltrain\_z.reshape(self.Ltrain\_z.size,  
 1) \* self.Ltrain\_z.reshape(1,  
 self.Ltrain\_z.size)  
  
 self.alpha, self.primal, \_ = scipy.optimize.fmin\_l\_bfgs\_b(func=self.\_\_LDc\_obj,  
 bounds=self.bounds,  
 x0=np.zeros(Dtrain.shape[1]),  
 factr=1.0)  
 if self.kernelType is None:  
 self.wc = np.sum(  
 self.alpha.reshape(1, self.alpha.size) \* self.Ltrain\_z.reshape(1, self.alpha.size) \* self.expandedD,  
 axis=1)  
  
 self.dual\_value = - self.primal  
 return self  
  
 def compute\_primal\_dual\_value(self):  
 primal\_value = 0.5 \* np.linalg.norm(self.wc) \*\* 2 + self.C \* np.sum(  
 np.maximum(0, 1 - self.Ltrain\_z \* (np.dot(self.wc.T, self.expandedD))))  
 self.primal\_value = primal\_value  
 return self.primal\_value, self.dual\_value  
  
 def compute\_duality\_gap(self):  
 return self.primal\_value - self.dual\_value  
  
 def predict(self, Dtest, labels=False):  
 if self.kernelType is not None:  
 if self.kernelType == 'Polynomial':  
 self.S = np.sum(  
 np.dot((self.alpha \* self.Ltrain\_z).reshape(1, -1), self.\_\_polynomial\_kernel(self.Dtrain, Dtest)),  
 axis=0)  
 elif self.kernelType == 'RBF':  
 self.S = np.sum(  
 np.dot((self.alpha \* self.Ltrain\_z).reshape(1, -1), self.\_\_RBF\_kernel(self.Dtrain, Dtest)), axis=0)  
 else:  
 return  
 else:  
 *# self.wc = np.sum(self.alpha \* self.Ltrain\_z \* self.expandedD, axis=1)  
 # self.w, self.b = self.wc[:-1], self.wc[-1::]* self.w, self.b = self.wc[0:self.Dtrain.shape[0]], self.wc[-1] \* self.K  
 *# self.S = np.dot(self.w.T, Dtest) + self.b \* self.K* self.S = (vrow(self.w) @ Dtest + self.b).ravel() *# \* self.K* if labels is True:  
 predicted\_labels = np.where(self.S > 0, 1, 0)  
 return predicted\_labels  
 else:  
 return self.S