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
```