rapport

March 8, 2025

1 Kernel Kaggle Challenge - MVA 2024-2025

1.0.1 Team : ?

Lilian Say : lilian.say@ens-paris-saclay.fr

Benjamin Deporte : benjamin.deporte@ens-paris-saclay.fr

Link to repo : https://github.com/BenjaminDeporte/MVA_Kernel_Project

1.1 1- Codage des Kernels 'string'

Dans un premier temps, nous avons codé deux Kernels 'string': - le kernel Spectrum - le kernel Mismatch

Nous reproduisons dans ce notebook les deux classes, dont la version à jour figure dans le fichier kernels.py du repo

Chaque classe contient deux méthodes : - k_value(x1,x2) pour calculer $K(x_1, x_2)$ où x_1, x_2 sont deux strings - k_matrix(xs, ys) pour calculer la matrice de Gram $K(x_i, y_j)$

Le fichier kernels.py contient également quelques tests unitaires

1.1.1 Kernel Spectrum

[1]:
```python
class KernelSpectrum():

    dna_alphabet = ['A','G','C','T']

    def __init__(self,k=None):
        # default value for k
        if k is None:
            self.k=3
        else:
            self.k=k
        # create list of k-uplets for faster computation
        # product create an iterator of tuples of cartesian products
        iter_tuples = product(self.dna_alphabet, repeat=self.k)
        # change from tuples of k characters to strings
        self.all_kuplets = [ ''.join(t) for t in iter_tuples]

```

```
def k_value(self,x1,x2):
    """Compute K(x,y)

    Args:
        x1 (_type_): string, or array of one string
        x2 (_type_): string, or array of one string

    Raises:
        NameError: if not string in inputs

    Returns:
        _type_: kernel_spectrum(x1,x2)
    """
    # type check and recast
    if isinstance(x1, np.ndarray):
        x1 = x1.squeeze()
        x1 = x1[0]
    if isinstance(x2, np.ndarray):
        x2 = x2.squeeze()
        x2 = x2[0]
    if isinstance(x1, str) is False or isinstance(x2, str) is False:
        raise NameError('Can not compute a kernel on data not string')

    # list all k-uplets in x1
    x1_kuplets = [ x1[i:i+self.k] for i in range(len(x1)-self.k+1) ]
    c1 = Counter()
    for uplet in x1_kuplets:
        c1[uplet] += 1
    # list all k-uplets in x2
    x2_kuplets = [ x2[i:i+self.k] for i in range(len(x2)-self.k+1) ]
    c2 = Counter()
    for uplet in x2_kuplets:
        c2[uplet] += 1
    # compute kernel value
    kernel = 0
    for uplet, occurences_in_x1 in c1.items():
        occurences_in_x2 = c2.get(uplet, 0)
        kernel += occurences_in_x1 * occurences_in_x2

    return kernel

def k_matrix(self, xs, ys):
    """compute and return Gram matrix K(x_i, y_j)
    for i in range(xs), j in range(xs)

    Args:
        xs (_type_): array of strings
```

```
            ys (_type_): array of strings
        """
        x_data = xs
        y_data = ys
        if isinstance(xs, list) is True:
            x_data = np.array(xs)
        if isinstance(ys, list) is True:
            y_data = np.array(ys)

        if isinstance(x_data, np.ndarray) is False or isinstance(y_data, np.
↪ndarray) is False:
            raise NameError('can not compute design matrix - input is not an␣
↪array')

        nx = x_data.shape[0]
        ny = y_data.shape[0]
        gram = np.zeros((nx, ny))

        for i in range(nx):
            x_i = x_data[i]
            for j in range(ny):
                y_j = y_data[j]
                gram[i,j] = self.k_value(x_i, y_j)

        return gram
```

1.1.2 Kernel Mismatch

[2]:
```
class KernelMismatch():

    dna_alphabet = ['A','G','C','T']

    def __init__(self,k=None):
        # default value for k
        if k is None:
            self.k=3
        else:
            self.k=k
        # create list of k-uplets for faster computation
        # product create an iterator of tuples of cartesian products
        iter_tuples = product(self.dna_alphabet, repeat=self.k)
        # change from tuples of k characters to strings
        self.all_kuplets = [ ''.join(t) for t in iter_tuples]

    def _mismatches(self, kuplet, mismatches=1):
        """Compute all possible mismatches of k-uplet, with at most m mismatches
```

```
        Args:
            kuplet (string): input k-uplet
            m (int, optional): maximum number of allowed mismatches. Defaults␣
↪to 1.
        """
        mismatches_kuplets = []

        for alphabet_kuplet in self.all_kuplets:
            nb_mismatches = np.sum([kuplet[i] != alphabet_kuplet[i] for i in␣
↪range(self.k)])
            if nb_mismatches <= mismatches:
                mismatches_kuplets.append(alphabet_kuplet)

        return mismatches_kuplets

    def k_value(self,x1,x2, mismatches=1, verbose=False):
        """Compute K(x,y)

        Args:
            x1 (_type_): string, or array of one string
            x2 (_type_): string, or array of one string

        Raises:
            NameError: if not string in inputs

        Returns:
            _type_: kernel_spectrum(x1,x2)
        """
        # type check and recast
        if isinstance(x1, np.ndarray):
            x1 = x1.squeeze()
            x1 = x1[0]
        if isinstance(x2, np.ndarray):
            x2 = x2.squeeze()
            x2 = x2[0]
        if isinstance(x1, str) is False or isinstance(x2, str) is False:
            raise NameError('Can not compute a kernel on data not string')

        # list all k-uplets in x1
        x1_kuplets = [ x1[i:i+self.k] for i in range(len(x1)-self.k+1) ]
        # compute dictionnary of unique kuplets in x1 with number of occurences
        c1 = Counter()
        for uplet in x1_kuplets:
            c1[uplet] += 1

        # list all k-uplets in x2
        x2_kuplets = [ x2[i:i+self.k] for i in range(len(x2)-self.k+1) ]
```

```
        # compute dictionnary of unique kuplets in x2 with number of occurences
        c2 = Counter()
        for uplet in x2_kuplets:
            c2[uplet] += 1

        kernel = 0
        # loop over unique kuplets in x1
        for uplet, occurences in c1.items():
            # what are all possible mismatches of this kuplet
            mismatches_kuplet = self._mismatches(uplet, mismatches=mismatches)
            for mismatch in mismatches_kuplet:
                # how many times does this mismatched kuplet appear in x2
                occurences_in_x2 = c2.get(mismatch, 0)
                kernel += occurences * occurences_in_x2
                if occurences_in_x2 > 0 and verbose is True:
                    print(f"uplet in x1 = {uplet}, mismatch in x1 occuring in␣
↪x2 = {mismatch}, number of occurences_in_x2 = {occurences_in_x2}")

        return kernel

    def k_matrix(self, xs, ys):
        """compute and return Gram matrix K(x_i, y_j)
        for i in range(xs), j in range(xs)

        Args:
            xs (_type_): array of strings
            ys (_type_): array of strings
        """
        x_data = xs
        y_data = ys
        if isinstance(xs, list) is True:
            x_data = np.array(xs)
        if isinstance(ys, list) is True:
            y_data = np.array(ys)

        if isinstance(x_data, np.ndarray) is False or isinstance(y_data, np.
↪ndarray) is False:
            raise NameError('can not compute design matrix - input is not an␣
↪array')

        nx = x_data.shape[0]
        ny = y_data.shape[0]
        gram = np.zeros((nx, ny))

        for i in range(nx):
            x_i = x_data[i]
            for j in range(ny):
```

```
                y_j = y_data[j]
                gram[i,j] = self.k_value(x_i, y_j)

        return gram
```

1.2 2- Classifieur SVM

Nous avons réutilisé le code du HomeWork 2. Nos deux classes KernelSVC figurent dans le fichier methods.py

Leurs formulations diffèrent sur deux points mineurs : - l'une optimise en α_i, l'autre en $\alpha_i y_i$ - les contraintes d'inégalité sont encapsulées dans une classe LinearConstraints de scipy pour l'une des classes.

A noter que l'une des classes a été testée vs le classifieur scikit dans le HW2 (Deporte), avec des résultats comparables voire identiques modulo les arrondis numériques.

[ ]:
```
#------------------------------------------
# ALGO SVC LILIAN
#------------------------------------------

class KernelSVCLilian():

    def __init__(self, C, kernel, epsilon = 1e-3):
        self.type = 'non-linear'
        self.C = C
        self.kernel = kernel
        self.alpha = None
        self.support = None # support vectors
        self.epsilon = epsilon
        self.norm_f = None


    def fit(self, X, y):
       #### You might define here any variable needed for the rest of the code
        N = len(y)
        K = self.kernel(X, X)

        # Lagrange dual problem
        def loss(alpha):
            return 0.5*np.dot(alpha*y, np.dot(K, alpha*y)) - np.sum(alpha)␣
 ↪#'''--------------dual loss ------------------ '''

        # Partial derivate of Ld on alpha
        def grad_loss(alpha):
            return np.dot(K, alpha*y)*y - np.ones_like(alpha) #␣
 ↪'''----------------partial derivative of the dual loss wrt alpha␣
 ↪-----------------'''
```

```
        # Constraints on alpha of the shape :
        # -  d - C*alpha  = 0
        # -  b - A*alpha >= 0

        fun_eq = lambda alpha: np.dot(alpha, y)# '''----------------function␣
↪defining the equality constraint------------------'''
        jac_eq = lambda alpha: y #'''----------------jacobian wrt alpha of the ␣
↪equality constraint------------------'''
        fun_ineq = lambda alpha: self.C - alpha # '''---------------function␣
↪defining the inequality constraint-------------------'''
        jac_ineq = lambda alpha: -np.eye(N) # '''---------------jacobian wrt␣
↪alpha of the  inequality constraint-------------------'''
        fun_ineq2 = lambda alpha: alpha # '''---------------function defining␣
↪the inequality constraint-------------------'''
        jac_ineq2 = lambda alpha: np.eye(N) # '''---------------jacobian wrt␣
↪alpha of the  inequality constraint-------------------'''

        constraints = ({'type': 'eq',  'fun': fun_eq, 'jac': jac_eq},
                       {'type': 'ineq', 'fun': fun_ineq, 'jac': jac_ineq},
                       {'type': 'ineq', 'fun': fun_ineq2, 'jac': jac_ineq2}
                      )

        optRes = optimize.minimize(fun=lambda alpha: loss(alpha),
                                   x0=np.ones(N),
                                   method='SLSQP',
                                   jac=lambda alpha: grad_loss(alpha),
                                   constraints=constraints)
        self.alpha = optRes.x

        ## Assign the required attributes
        idx = self.alpha > self.epsilon

        self.support = X[idx]#'''------------------- A matrix with each row␣
↪corresponding to support vectors ------------------'''
        self.support_alpha = self.alpha[idx]
        self.support_y = y[idx]

        self.b =  np.mean(self.support_y - self.separating_function(self.
↪support))#''' -----------------offset of the classifier------------------ '''
        self.norm_f = np.sqrt(np.dot(self.alpha*y, np.dot(K, self.alpha*y)))#␣
↪'''------------------------RKHS norm of the function f␣
↪------------------------------'''
```

```
    ### Implementation of the separting function $f$
    def separating_function(self,x):
        # Input : matrix x of shape N data points times d dimension
        # Output: vector of size N
        K_val = self.kernel(self.support, x)
        return np.sum((self.support_alpha*self.support_y)[:, np.newaxis]*K_val,␣
↪axis=0)


    def predict(self, X):
        """ Predict y values in {-1, 1} """
        d = self.separating_function(X)
        return 2 * (d+self.b> 0) - 1
```

[ ]:
```
#---------------------------------------
# ALGO SVC BEN
#---------------------------------------

class KernelSVCBen():

    def __init__(self, C, kernel, type='non-linear', epsilon = 1e-3):
        self.type = type
        self.C = C
        self.kernel = kernel
        self.alpha = None
        self.support = None # support vectors
        self.epsilon = epsilon
        self.norm_f = None


    def fit(self, X, y):
        #### You might define here any variable needed for the rest of the code
        N = len(y)
        self.X = X
        self.y = y
        # compute gram matrix, we might need it :-)
        self.gram = self.kernel(X,X)
        # vector of ones, size N
        self.ones = np.ones(N)
        # matrix NxN of y_i on diagonal
        self.Dy = np.diag(y)

        # Lagrange dual problem
        def loss(alpha):
            objective_function = 1/2 * alpha @ self.gram @ alpha - alpha @ self.
↪y
            return  objective_function
```

```
        # Partial derivate of Ld on alpha
        def grad_loss(alpha):
            gradient = self.gram @ alpha - self.y
            return gradient

        # equality constraint
        fun_eq = lambda alpha: alpha @ self.ones
        jac_eq = lambda alpha: self.ones
        # inequality constraint avec la classe LinearConstraint de scipy
        inequality_constraint = LinearConstraint(self.Dy, np.zeros(N), self.C *␣
↪self.ones)

        constraints = ( [{'type': 'eq', 'fun': fun_eq, 'jac': jac_eq},
                        inequality_constraint]
                        )

        optRes = optimize.minimize(fun=lambda alpha: loss(alpha),
                                   x0=np.ones(N),
                                   method='SLSQP',
                                   jac=lambda alpha: grad_loss(alpha),
                                   constraints=constraints)
        self.alpha = optRes.x

        ## Assign the required attributes
        # list of indices of support vectors in dataset, None if not a support␣
↪vector
        self.indices_support = np.array([ i if (self.epsilon < self.
↪alpha[i]*self.y[i]) and (self.alpha[i]*self.y[i] <= self.C) else None for i␣
↪in range(N) ])
        self.indices_support = self.indices_support[self.indices_support !=␣
↪None].astype(int)
        # support vectors (data points on margin)
        self.support = self.X[self.indices_support]
        # alphas on support vectors
        self.alpha_support = self.alpha[self.indices_support]
        # compute b by averaging over support vectors
        b = self.y - self.gram @ self.alpha
        b_sv = b[self.indices_support]
        self.b = np.mean(b_sv)
        # '''------------------------RKHS norm of the function f␣
↪-----------------------------'''
        self.norm_f = 1/2 * self.alpha @ self.gram @ self.alpha

        return self
```

```python
    ### Implementation of the separting function $f$
    def separating_function(self,x):
        # Input : matrix x of shape N data points times d dimension
        # Output: vector of size N
        return self.kernel(x, self.support) @ self.alpha_support + self.b


    def predict(self, X):
        """ Predict y values in {-1, 1} """
        d = self.separating_function(X)
        return 2 * (d+self.b> 0) - 1
```

1.3 3- XPs

[ ]:

```python

```

[ ]:

```python

```

[ ]:

```python

```

[ ]:

```python

```

[ ]:

```python

```