TKL USER MANUAL

October 2020

1 Introduction

Kernel methods for classification and regression (and Support Vector Machines (SVMs) in particular) require selection of a kernel. Kernel Learning (KL) algorithms such as those found in [1, 2, 3] automate this task by finding the kernel, $k \in \mathcal{K}$ which optimizes an achievable metric such as the soft margin (for classification). The set of kernels, $k \in \mathcal{K}$, over which the algorithm can optimize, however, strongly influences the performance and robustness of the resulting classifier or predictor.

To understand how the choice of $\mathcal{K}$ influences performance and robustness, three properties were proposed in [4] to characterize the set $\mathcal{K}$ - tractability, density, and universality. Specifically, $\mathcal{K}$ is tractable if $\mathcal{K}$ is convex (or, preferably, a linear variety) - implying the KL problem is solvable using, e.g. [5, 6, 7, 8, 9]. The set $\mathcal{K}$ has the density property if, for any $\epsilon > 0$ and any positive kernel, k^* there exists a $k \in \mathcal{K}$ where $\|k - k^*\| \leq \epsilon$. The density property implies the kernel will perform well on untrained data (robustness or generalizability). The set $\mathcal{K}$ has the universal property if any $k \in \mathcal{K}$ is universal - ensuring the classifier/predictor will perform arbitrarily well on large sets of training data.

In [4], the Tessellated Kernels (TKs) were shown to have all 3 properties, the first known such class of kernels. This work was based on a general framework for using positive matrices to parameterize positive kernels (as opposed to positive kernel matrices as in [7, 8, 10]). Unfortunately, however, the algorithms proposed in [4] were implemented using SemiDefinite Programming (SDP) (thereby limiting the amount of training data) or using SimpleMKL with a randomized linear basis for the kernels (implying loss of density). Thus, while the algorithms in [4] outperformed all other methods (including Neural Nets) as measured by Test Set Accuracy (TSA), the computation times were not competitive. Furthermore, the results in [4] did not address the problem of regression.

In this paper, we extend the TK framework proposed in [4] to the problem of regression. The KL problem in regression has been studied using SDP in [8, 10] and Quadratic Programming (QP) in e.g. [5, 6]. However, neither of these previous works considered a set of kernels with both the tractability and the density property. By generalizing the Tessellated KL framework proposed in [4] to the regression problem, we demonstrate significant increases in performance, as measured by Mean Square Error (MSE), and when compared to the results in [5, 6, 8].

In addition, we show that the SDP-based algorithm [4] for classification, and extended here to regression, can be decomposed into primal and dual sub-problems, OPT_A and OPT_P - similar to the approach taken in [5, 6]. Furthermore, we show that OPT_P (an SDP) admits an analytic solution using the Singular Value Decomposition (SVD) - an approach which allows us to consider higher dimensional feature spaces and more complex TKs. In addition, OPT_A is a convex QP and may be solved efficiently with achieved complexity which scales as $O(m^{2.16})$ where m is the number of data points. We use a two-step algorithm on OPT_A and OPT_P and show that termination at $OPT\_A = OPT\_P$ is equivalent to global optimality. The resulting algorithm, then, does not require the use of SDP and, when applied to several standard test cases, is shown to retain the favorable TSA of [4] for classification, while offering improved MSE for regression, and competitive computation times as compared to other KL and deep learning algorithms.

2 What is kernel learning

Kernel methods for classification and regression (and Support Vector Machines (SVMs) in particular) require selection of a kernel. Kernel Learning (KL) algorithms automate this task by finding the kernel , *k* from

the set *K* which optimizes an achievable metric such as the soft margin (for classification).

To understand how the choice of K influences performance and robustness, three properties were proposed to characterize the set K - tractability, density, and universality.

- K is tractable if K is convex (or, preferably, a linear variety) - implying the KL problem is solvable using.
- The set K has the density property if, for any $e > 0$ and any positive kernel, $\hat{k}$ there exists a k from K such that:

$$\|k - \hat{k}\| \leq e$$

- K has the universal property if any k from K is universal - ensuring the classifier/predictor will perform arbitrarily well on large sets of training data.

More about this property you can find here `https://arxiv.org/pdf/2106.08443.pdf`. Moreover, The Tessellated Kernels (TKs) were shown to have all 3 properties

The general optimization problem can be formulated as

$$\min_{k \in K} \max_{\alpha \in A} -\frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \alpha_i \alpha_j k(x_i, x_j) + \kappa(\alpha)$$

where

$$\kappa(\alpha) = \begin{cases} -\epsilon \sum_{i=1}^{m} |\alpha_i| + \sum_{i=1}^{m} y_i \alpha_i, & \text{If the task is regression} \\ \sum_{i=1}^{m} \alpha, & \text{if the task is classification.} \end{cases}$$

The optimization set is also different

$$A = \begin{cases} \{\alpha \in R^m \ : \ \sum_{i=1}^{m} \alpha_i = 0, \ \alpha_i \in [-C, C]\} & \text{If the task is regression} \\ \{\alpha \in R^m \ : \ \sum_{i=1}^{m} \alpha_i y_i = 0, \ 0 \leq \alpha_i \leq C\} & \text{if the task is classification.} \end{cases}$$

But there are two main questions 1. How to choose K. 2. How to find the optimal solution.

3 Formulation Tesselated Kernel

Tessellated Kernels is a new class of kernel functions, that can be represented in the following way

$$K := \left\{ k \mid k(x, y) = \int_X N(z, x)^T P N(z, y) dz, P \geq 0 \right\}$$

where

$$N_T^d(z, x) = \begin{bmatrix} Z_d(z, x) I(z - x) \\ Z_d(z, x) I(x - z) \end{bmatrix} \quad \text{and} \quad I(z) = \begin{cases} 1 & z \geq 0 \\ 0 & \text{otherwise.} \end{cases}$$

After integrations and long calculations we got the final results

$$k(x, y) = \sum_{i,j=1}^{q} Q_{i,j} g_{i,j}(x, y) + R_{i,j} t_{i,j}(x, y) + R_{i,j}^T t_{i,j}(y, x) + S_{i,j} h_{i,j}(x, y)$$

Where

$$g_{i,j}(x, y) := x^{\delta_i} y^{\delta_j} T(p^*(x, y), b, \gamma_{i,j} + 1)$$

$$t_{i,j}(x, y) := x^{\delta_i} y^{\delta_j} T(x, b, \gamma_{i,j} + 1) - g_{i,j}(x, y)$$

$$h_{i,j}(x,y) := x^{\delta_i} y^{\delta_j} T(a,b,\gamma_i+\gamma_j+1) - g_{i,j}(x,y) - t_{i,j}(x,y) - t_{i,j}(y,x)$$

$$p^*(x,y)_i = \max\{x_i, y_i\}$$

$$T(x,y,\zeta) = \prod_{j=1}^{n} \left(\frac{y_j^{\zeta_j}}{\zeta_j} - \frac{x_j^{\zeta_j}}{\zeta_j} \right).$$

Fortunately, we implemented our kernel and now you can use it. Please, look at kernel functions `https://talitsky.github.io/v1/kernel-functions`.

4 Optimization

We now propose the efficient implementation of the *Frank-Wolfe algorithm*, based on our paper `http://control.asu.edu/Publications/2021/Colbert_NIPS_2021.pdf`. In more details

Algorithm 1: An Efficient FW Algorithm for TKL. Note that the stopping criterion is defined using the duality gap $OPT\_P(\alpha_k) - OPT\_A(P_k) > 0$, which is equivalent to the stopping criterion used in the standard FW algorithm.

`Initialize` $P_0 = I$, $k = 0$, $\alpha_0 = OPT\_A(P_0)$;
`while` $OPT\_P(\alpha_k) - OPT\_A(P_k) \geq \epsilon$
  Step 1a: $\alpha_k = \arg OPT\_A(P_k)$
  Step 1b: $S_k = \arg OPT\_P(\alpha_k)$
  Step 2: $\gamma_k = \arg \min_{\gamma \in [0,1]} OPT\_A(P_k + \gamma(S_k - P_k))$
  Step 3: $P_{k+1} = P_k + \gamma_k(S_k - P_k)$, $k = k+1$
`end while`

Where we denoted OPT_P and OPT_A like

$$OPT\_P(\alpha) = \min_{k \in K} -\frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \alpha_i \alpha_j k(P, x_i, x_j) + \kappa(\alpha)$$

and

$$OPT\_A(P) = \max_{\alpha \in A} -\frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \alpha_i \alpha_j k(P, x_i, x_j) + \kappa(\alpha)$$

5 Performance Measures, Basic Usage, and Code Organization

This section describes PMKL, shows simple examples and the structure of the code.

5.1 Kernel functions

Firstly, we realized the `KernelFunctions` to compute TK matrices and a monomial basis, given x and degree q one can call

```
import numpy as np
from PMKLpy import KernelFunctions
x = np.random.rand(100, 1)
q = 1
Lower, Upper = x.min(axis = 0) - 0.1 , x.max(axis = 0) + 0.1
Kernel = KernelFunctions.Kernel(x, Lower, Upper, q)
```

`Kernel` is a special class for fast computing of TK. The first argument of is data points, the second and third are low and upper bounds for samples and the last one is degree of monomial basis. This object has 2 attributes. The first one is `Kernel.Z`, that is a vector of monomial basis. It stores `numpy.array` object with the shape `(n_samples, q)` where $q = \binom{d+n}{d}$. The second attribute is `Kernel.K` is a interior structure for fast computing of TK.

If you need only monomial basis, then you can use the `monomials(x,d)` function takes a matrix of inputs `x` and computes the monomial basis `Z` of degree `d`

```
from PMKLpy import Transformation
Z = Transformation.monomials(x, 1)
```

To compute TK, the user need to call `makeK` function

```
TK = KernelFunctions.makeK(Kernel, P)
```

`makeK` is a function that returns a Kernel matrix `(n_samples, n_samples)`. It requires a `P` matrix of the form `(2q, 2q)` and an object Kernel.

For general Kernel matrices `K(X1, X2)` we realized a function `TKtest`. This function takes two matrices of inputs, as well as monomial basis of the inputs `Z1, Z2` and a lower `a` and upper `b` bound over which we integrate and a matrix `P` that parameterizes the TK kernel function. Computes the test kernel matrix for a TK kernel.

```
from PMKLpy import Transformation
from PMKLpy import KernelFunctions
x1 = x[:100]
x2 = x[100:]
Lower, Upper = x.min(axis = 0) - 0.1 , x.max(axis = 0) + 0.1
P = np.eye(6)
Z1 = Transformation.monomials(x1, 1)
Z2 = Transformation.monomials(x2, 1)

TK = KernelFunctions.TKtest(x1,x2,Z1,Z2,Lower, Upper,P)
```

5.2 SVM and optimization

The main class of SVM is `PMKL`. It requires several parameters for Kernel computing and C-SVC or ε-SVR.

class PMKLpy.PMKL.PMKL(*C = 10, degree = 1, bound = 0.1, epsilon = 0.1, maxit = 100, tol = 1.e-6, probability = False, to_print= True*)

Parameters:

- *C, float, default=10*
  Regularization parameter, smaller values lead to mappings that are more general.
- *degree, int, default=1*
  Degree of the TK Kernel function
- *bound, float, default=0.1*
  Area of integration is [-*bound*,1+*bound*]
- *epsilon, float, default=0.1* Epsilon parameter for regression problems
- *maxit, float default=100*
  Maximum number of iterations. Each Iteration includes the SVM training
- *tol, float, default=1.e-6*
  Tolerance of the TKL optimization algorithm

- *probability, boolean, default= False*
  If True the algorithm are able to predict probability for binary classification. If False the standard SVM problem will be solved

- *to_print, boolean, default=True*
  If True the the algorithm will be print the objective function for each iteration, else there will not be any outputs

Attributes:

- *Opt, class*
  Our special class for storing objective values, dual gaps and step lengths.

- *Params, class*
  This objects stores key parameters defined in the initialization. After fitting there are P matrix for computing the TK.

- *Kernel, class*
  The Kernel class for fast computing of TK and monomials. It exists only after fitting.

- *xOld, ndarray of shape (n_samples, n_features)*

- *x, ndarray of shape (n_samples, n_features)*
  The scaled train data The original train data

- *y, ndarray of shape (n_samples, 1)*
  The target values for train data

- *scaleFactor, MinMaxScaler() sklearn object*
  Using for scaling input data

- *model, SVM model*
  Fitted SVM for prediction based on precomputted Kernel

Methods

- `fit(X, y)` Fit the SVM model according to the given training data.

- `predict(X)` Perform classification or regression on samples in X.

- `predict_proba` Compute probabilities of possible outcomes for samples in X (only for classification). IT IS NOT IMPLEMENTED

- `get_params`IT IS NOT IMPLEMENTED

Example:

```
from PMKLpy import PMKL
SVM = PMKL.PMKL()
SVM.fit(x, y)
ypred = SVM.predict(x)
```

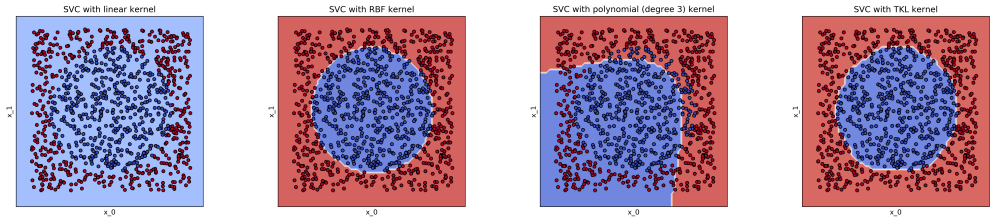

Figure 1: Data Point and decision function for Example 1

5.3 Example 1

The first example is comparing C-SVC with different kernels

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm, datasets
from PMKLpy import PMKL
from myplots import *

X, y = PMKL.loadex1()


C = 1.0

SVM = PMKL.PMKL( C=C, to_print = False)
SVM.fit(X, y)
# yPred = SVM.predict(X)
# prediction = np.round(yPred)

models = [svm.SVC(kernel='linear', C=C),
          svm.SVC(kernel='rbf', gamma=0.7, C=C),
          svm.SVC(kernel='poly', degree=3, gamma='auto', C=C)]

models = [clf.fit(X, y) for clf in models]
models.append(SVM)

# title for the plots
titles = ['SVC with linear kernel',
          'SVC with RBF kernel',
          'SVC with polynomial (degree 3) kernel',
            'SVC with TKL kernel']

fig, sub = plt.subplots(1, 4, figsize = (25, 5))
plt.subplots_adjust(wspace=0.4, hspace=0.4)

X0, X1 = X[:, 0], X[:, 1]
xx, yy = make_meshgrid(X0, X1)

for clf, title, ax in zip(models, titles, sub.flatten()):
    plot_contour_and_points(ax, clf, xx, yy, X0, X1, y, title, cmap=plt.cm.coolwarm, alpha=0.8)
```

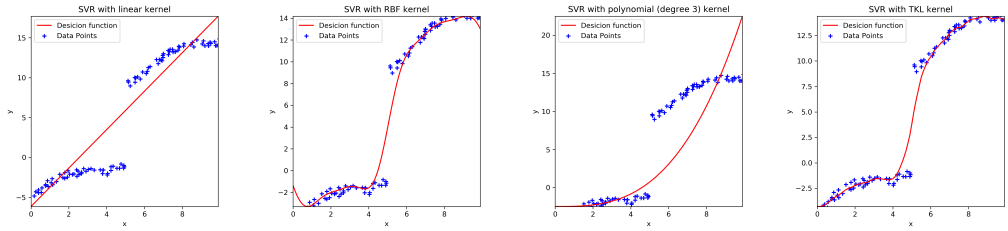

Figure 2: Data points and decision function for example 2

5.4 Example 2

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm, datasets
from PMKLpy import PMKL

num = 100;
x = 10.*np.random.rand(num,1);
y = x + np.sin(x)+5*np.sign(x-5)+ np.random.rand(num,1)-0.5

C = 1.0
eps = 0.1

SVM = PMKL.PMKL( C=C, epsilon= eps, to_print = False)
# yPred = SVM.predict(X)
# prediction = np.round(yPred)

models = [svm.SVR(kernel='linear', C=C, epsilon = eps),
          svm.SVR(kernel='rbf', gamma=0.7, C=C, epsilon = eps),
          svm.SVR(kernel='poly', degree=3, gamma='auto', C=C, epsilon = eps),
          SVM]

models = [clf.fit(x, y) for clf in models]
models.append(SVM)

titles = ['SVR with linear kernel',
          'SVR with RBF kernel',
          'SVR with polynomial (degree 3) kernel',
          'SVR with TKL kernel']

fig, sub = plt.subplots(1, 4, figsize = (25, 5))
plt.subplots_adjust(wspace=0.4, hspace=0.4)

xx = np.arange(0, 10, 0.1)[:, np.newaxis]
for clf, title, ax in zip(models, titles, sub.flatten()):
    yy = clf.predict(xx)
    ax.plot(xx, yy, color = 'r', label = 'Desicion function')
    ax.scatter(x, y, c= 'b', marker = '+', label = 'Data Points')
    ax.set_xlim(xx.min(), xx.max())
    ax.set_ylim(yy.min(), yy.max())
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.legend()
    ax.set_title(title)
```

References

[1] Z. Xu, R. Jin, H. Yang, I. King, and M. Lyu, "Simple and efficient multiple kernel learning by group lasso," in Proceedings of the 27th international conference on machine learning, 2010.

[2] S. Sonnenburg, G. Rätsch, S. Henschel, C. Widmer, J. Behr, A. Zien, F. De Bona, A. Binder, C. Gehl, and V. Franc, "The shogun machine learning toolbox," Journal of Machine Learning Research, 2010.

[3] H. Yang, Z. Xu, J. Ye, I. King, and M. Lyu, "Efficient sparse generalized multiple kernel learning," IEEE Transactions on neural networks, 2011.

[4] B. Colbert and M. Peet, "A convex parametrization of a new class of universal kernel functions," Journal of Machine Learning Research, vol. 21, no. 45, 2020.

[5] A. Rakotomamonjy, F. Bach, S. Canu, and Y. Grandvalet, "SimpleMKL," Journal of Machine Learning Research, 2008.

[6] A. Jain, S. Vishwanathan, and M. Varma, "SPF-GMKL: generalized multiple kernel learning with a million kernels," in Proceedings of the ACM International Conference on Knowledge Discovery and Data Mining, 2012.

[7] G. Lanckriet, N. Cristianini, P. Bartlett, L. El Ghaoui, and M. Jordan, "Learning the kernel matrix with semidefinite programming," Journal of Machine Learning Research, 2004.

[8] S. Qiu and T. Lane, "Multiple kernel learning for support vector regression," Computer Science Department, The University of New Mexico, Albuquerque, NM, USA, Tech. Rep, 2005.

[9] M. Gönen and E. Alpaydın, "Multiple kernel learning algorithms," Journal of Machine Learning Research, 2011.

[10] K. Ni, S. Kumar, and T. Nguyen, "Learning the kernel matrix for superresolution," in 2006 IEEE Workshop on Multimedia Signal Processing, 2006.