# Tensor Decomposition in Multiple Kernel Learning

**Linh Nguyen**

**School of Science**

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 30.07.2017

**Thesis supervisor:**

Prof. Juho Rousu

**Thesis advisors:**

D.Sc. Sandor Szedmak

M.Sc. Anna Cichonska

Aalto University
School of Science

AALTO UNIVERSITY
SCHOOL OF SCIENCE

ABSTRACT OF THE
MASTER'S THESIS

Author: Linh Nguyen

Title: Tensor Decomposition in
Multiple Kernel Learning

Date: 30.07.2017 Language: English Number of pages: 0+65

Department of Computer Science

Professorship:

Supervisor: Prof. Juho Rousu

Advisors: D.Sc. Sandor Szedmak, M.Sc. Anna Cichonska

Modern data processing and analytic tasks often deal with high dimensional matrices or tensors; for example: environmental sensors monitor (time, location, temperature, light) data. For large scale tensors, efficient data representation plays a major role in reducing computational time and finding patterns.
The thesis firstly studies about fundamental matrix, tensor decomposition algorithms and applications, in connection with Tensor Train decomposition algorithm. The second objective is applying the tensor perspective in Multiple Kernel Learning problems, where the stacking of kernels can be seen as a tensor. Decomposition this kind of tensor leads to an efficient factorization approach in finding the best linear combination of kernels through the similarity alignment. Interestingly, thanks to the symmetry of the kernel matrix, a novel decomposition algorithm for multiple kernels is derived for reducing the computational complexity.
In term of applications, this new approach allows the manipulation of large scale multiple kernels problems. For example, with $P$ kernels and $n$ samples, it reduces the memory complexity of $\mathcal{O}(P^2n^2)$ to $\mathcal{O}(P^2r^2 + 2rn)$ where $r < n$ is the number of low-rank components. This compression is also valuable in pair-wise multiple kernel learning problem which models the relation among pairs of objects and its complexity is in the double scale.
This study proposes AlignF_TT, a kernel alignment algorithm which is based on the novel decomposition algorithm for the tensor of kernels. Regarding the predictive performance, the proposed algorithm can gain an improvement in 18 artificially constructed datasets and achieve comparable performance in 13 real-world datasets in comparison with other multiple kernel learning algorithms. It also reveals that the small number of low-rank components is sufficient for approximating the tensor of kernels.

Keywords: Tensor decomposition, kernel learning, multiple kernel learning, multiple kernel approximation

# Preface

Six months ago, I started to do this thesis which firstly is matrix completion on drugs-targets interaction datasets or recommendation systems in general. The manipulation on matrices gives me more insight into the linear algebra world. Gradually, my attention transforms into exploring basic linear algebra operators and multidimensional matrices or tensors. Fortunately, I finally found an application of tensor decomposition on the kernel learning framework. This thesis is the report of this long process. I cannot express the joyfulness when understanding a new definition, deriving equations, and finding a new interested book.

I would like to thank my supervisor Professor Juho Rousu for his kindness and guidance for doing a proper scientific research. Without his funding and support, I can not finish my master study in 2 years. Dr Sandor Szedmak is my instructor and math teacher who inspired me to explore the tensor relations in machine learning. His lessons in math and science are the guidance for my further works and study. I really appreciate his time for carefully answering my questions. I want to thank Anna Cichonska for her experience in drug-interaction data and methodology. My proposed algorithm is inspired from her publication about the tensor kernel alignment algorithm. I want to appreciate Huibin Shen for providing datasets, the experiment section could not be done without his support. Finally, I want to thank all members in the KEPACO research group for your discussion and feedbacks related to my research.

Hope that I can return to do the academic research soon.

Otaniemi, 30.07.2017

Linh Nguyen

# Contents

**Abstract** ii

**Preface** iii

**Contents** iv

**1 Introduction** 1
- 1.1 Notation . . . 2

**2 Tensor decomposition** 3
- 2.1 Singular value decomposition . . . 3
  - 2.1.1 Singular value decomposition . . . 3
  - 2.1.2 Matrix factorization interpretation . . . 4
- 2.2 Tensors definition . . . 5
  - 2.2.1 SVD revisited . . . 17
- 2.3 CP decomposition . . . 17
  - 2.3.1 Tensor rank . . . 18
  - 2.3.2 The algorithm . . . 19
- 2.4 Tucker decomposition . . . 21
  - 2.4.1 The *n-rank* and multilinear rank . . . 21
  - 2.4.2 The Higher Order SVD (HOSVD) . . . 21
  - 2.4.3 Tucker decomposition algorithm . . . 22
- 2.5 Tensor train decomposition . . . 24
  - 2.5.1 Analysis and definition . . . 24
  - 2.5.2 Algorithm . . . 26

**3 Multiple Kernel Learning** 28
- 3.1 Kernel learning . . . 28
  - 3.1.1 Pairwise kernel learning . . . 30
- 3.2 Multiple Kernel Learning . . . 31
  - 3.2.1 MKL algorithms . . . 31
  - 3.2.2 Overall comparison . . . 34
  - 3.2.3 AlignF algorithm . . . 34

**4 Tensor method for Multiple Kernel Learning** 36
- 4.1 Factorized AlignF algorithm . . . 36
- 4.2 The optimal decomposition for the tensor of kernels . . . 38

**5 Experiments** 42
- 5.1 Data and experiment setup . . . 42
  - 5.1.1 Artificially constructed datasets . . . 42
  - 5.1.2 Real datasets . . . 44
  - 5.1.3 Experiments setup . . . 45
- 5.2 Results . . . 47

5.2.1 Artificial constructed datasets . . . . . . . . . . . . . . . . . . . 47
5.2.2 Real-world datasets . . . . . . . . . . . . . . . . . . . . . . . . 52

**6 Discussion** **59**

**7 Acknowledgement** **60**

**References** **61**

**A Appendix** **65**
A.1 The accuracy vs low-rank number . . . . . . . . . . . . . . . . . . . 65

# List of Figures


1 Algorithms to be considered in this thesis . . . . . . 1
2 Singular Vector Decomposition components . . . . . . 4
3 Matrix factorization interpretation for an element . . . . . . 5
4 An *order-3* tensor . . . . . . 6
5 An *order-3* tensor with specific values . . . . . . 6
6 A mode-1 fiber $\mathcal{A}(:, j, k)$, a mode-2 fiber $\mathcal{A}(i, :, k)$, and a mode-3 fiber $\mathcal{A}(i, j, :)$ . . . . . . 7
7 A *3-order* tensor fiber: $\mathcal{A}(:, 1, 1)$ . . . . . . 7
8 A slice of the *order-3* tensor: $\mathcal{A}(:, :, 2)$ . . . . . . 8
9 Rank-one third-order tensor . . . . . . 8
10 The mode-1 unfolding example . . . . . . 9
11 The mode-2 unfolding example . . . . . . 9
12 The mode-3 unfolding example . . . . . . 10
13 Tensor $\mathcal{A}$ of indices . . . . . . 10
14 The CP decomposition for an *order-3* tensor . . . . . . 18
15 The Tucker decomposition components for an *order-3* tensor . . . . . . 21
16 Tensor train decomposition and reconstruction for a specific element in a matrix . . . . . . 25
17 The Tensor train decomposition and its reconstruction for a specific element in an order-3 tensor (cubic) . . . . . . 25
18 Tensor train decomposition components for an order-4 tensor . . . . . . 26
19 Components of the Support Vector Machine algorithm in two dimensional samples. . . . . . . 28
20 The comparison between kernel learning and pair-wise kernel learning 30
21 The framework of Factorization AlignF . . . . . . 37
22 The Tensor Train algorithm decomposition steps for an order 3 tensor which is created by stacking kernels . . . . . . 39
23 The graphical model of the generation process for a sample . . . . . . 42
24 Toy datasets in different scales . . . . . . 43
25 Linear kernels in different scale datasets . . . . . . 44
26 *Toy-data-1*:Accuracy change in different datasets . . . . . . 47
27 *Toy-data-1*:The macro-F1 change in different datasets . . . . . . 48
28 *Toy-data-1*:The micro-F1 change in different datasets . . . . . . 49
29 *Toy-data-2*:Accuracy change in different datasets . . . . . . 49
30 *Toy-data-2*:The macro-F1 change in different datasets . . . . . . 50
31 *Toy-data-2*:The micro-F1 change in different datasets . . . . . . 51
32 *AlignF_TT* performance in different low-rank numbers in *Yeast* data 54
33 *AlignF_TT* performance in different low-rank numbers in *Emotions* data . . . . . . 54
34 *AlignF_TT* performance in different low-rank numbers in *iaprtc12* data 55
35 *AlignF_TT* performance in different low-rank numbers in *psortPos* data 55
36 *AlignF_TT* running time in different low-rank numbers in *Emotions* data . . . . . . 56

37 *AlignF_TT* running time in different low-rank numbers in *Enron* data 57
38 *AlignF_TT* running time in different low-rank numbers in *psortPos* data . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
39 *AlignF_TT* running time in different low-rank numbers in *psortNeg* data . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58
A1 *AlignF_TT* performance in different low-rank numbers in *Enron* data 65
A2 *AlignF_TT* performance in different low-rank numbers in *Fingerprint* data . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66
A3 *AlignF_TT* performance in different low-rank numbers in *Protein* data 66
A4 *AlignF_TT* performance in different low-rank numbers in *corel5k* data 67
A5 *AlignF_TT* performance in different low-rank numbers in *espgame* data 67
A6 *AlignF_TT* performance in different low-rank numbers in *iaprtc12* data 68
A7 *AlignF_TT* performance in different low-rank numbers in *mirflickr* data 68
A8 *AlignF_TT* performance in different low-rank numbers in *pascal07* data 69
A9 *AlignF_TT* performance in different low-rank numbers in *psortPos* data 69
A10 *AlignF_TT* performance in different low-rank numbers in *psortNeg* data 70
A11 *AlignF_TT* performance in different low-rank numbers in *plant* data 70

# 1 Introduction

This thesis covers two major topics: the Tensor decomposition and its application in Multiple Kernel Learning. Related algorithms are described in the Figure 1.

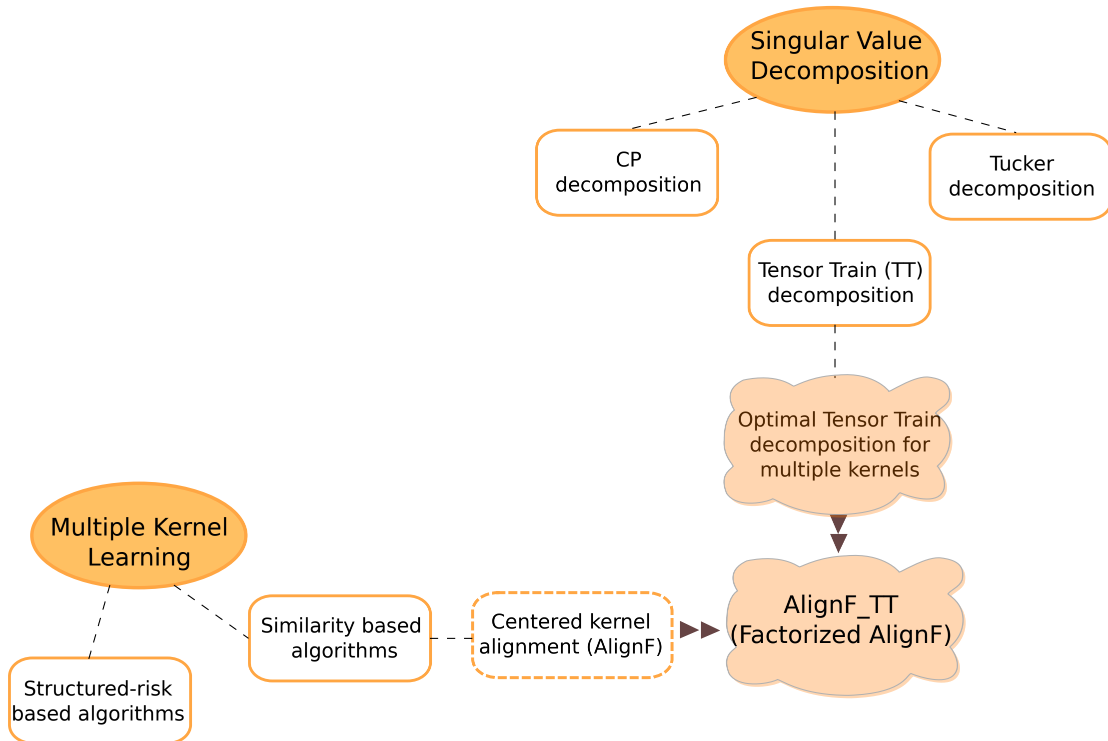



Figure 1: Algorithms to be considered in this thesis

This thesis first gives an overview of fundamental decomposition methods for matrices and tensors. In decomposing a matrix, Singular Value Decomposition (SVD) [1] algorithm represents a matrix as the weighted sum of rank-1 matrices or the multiplication of factor matrices. For a higher dimensional matrix or a tensor, decomposition algorithms are also derived from the SVD. The CANDECOMP/PARAFAC (CP) decomposition [2] takes the perspective of the weighted sum of rank-1 tensors while the Tucker decomposition [3] and Tensor Train decomposition [4] use SVD as the workhorse algorithm. Regarding the application, tensor decomposition methods also are employed in many fields: from linear algebra to machine learning, data mining, computer vision [5]. In Section 2, the primary objective is to derive solutions for these algorithms and relations, and for the potential applications.

Secondly, this study focuses on Multiple kernels learning [6], which has been studied mostly independently from tensor decomposition methods in the literature. The Multiple Kernel Learning (MKL) is a branch of the kernel learning method [7], where a set of kernels is available. The purpose of MKL is combining data from different sources and automatically selecting optimal kernels to improve the learning performance. Typically, these algorithms treat each kernel separately to find an optimal combination of them. To apply the tensor decomposition into these MKL

algorithms; this study focuses on two major MKL approaches based on the structured risk objectives [8] and kernel alignment algorithms [9]. Then Section 3 introduces the kernel learning framework, MKL algorithms and its development.

The fourth section describes two main findings of this study. One important finding is a novel algorithm for decomposing a tensor of kernels. Another interesting finding is a new centered kernel alignment algorithm which is based on the novel decomposition for the tensor of kernels. Then, Section 5 reports datasets and algorithm configurations to evaluate the performance of the proposed kernel alignment algorithm. Two sets of experiments are conducted on artificially constructed and real-world data sets. The first one manipulates in 18 artificially constructed datasets to evaluate the performance of kernel alignment algorithms under different data generation settings. The second experiment in 13 real-world datasets is conducted to compare the performance of algorithms in various kind of data including texts, images, and bioinformatics datasets.

## 1.1 Notation

- A vector containing all ones or zeros is denoted as $\mathbf{1}$ or $\mathbf{0}$, respectively.
- $x$ denotes a column vector and thus $x^T$ describes it in the row format.
- With $(x, y, \ldots, z)$ is the stacking of vectors to create a matrix in column order. The row stacking order as $(x^T; y^T; \ldots; z^T)$ by using the semicolon.
- A full matrix is denoted in the mathbold type fonts: $\mathbf{A}$.
- A general matrix $\mathbf{A} \in \mathbb{R}^{n \times m}$ is a stack of $m$ vector as $(x, y, \ldots, z)$ where each $x \in \mathbb{R}^n$.
- For a matrix $\mathbf{A}$, $A_{ij}$ or $A(i, j)$ is the element at row $i$th and column $j$th.
- The colon notation presents the row or column of a matrix. For example: $A(k, :) = [A_{k1}, A_{k2}, \ldots, A_{km}]$ is $k$th row of a matrix.
- $\Sigma$ is the diagonal matrix.
- $\sigma$ is the singular value.
- Tensors are denoted in calligraphic fonts : $\mathcal{A}$.

# 2 Tensor decomposition

## 2.1 Singular value decomposition

Finding the subspace that captures important properties of a dataset or the data-embedding configurations in a metric space is the crucial task in pattern recognition and machine learning [10]. By projecting data into this subspace and analysing the resulted data, many data-mining tasks can be solved; for example: dimensionality reduction, component interpretation, removing noise, and visualisation. To find this subspace, many well-known algorithms use Singular value decomposition (SVD) method [11] as the workhorse algorithm. For example, Principle Component Analysis aims to find the set of directions that explains most data variance and its solution corresponds to the SVD algorithm. Due to the importance of SVD in machine learning and data mining, this section describes its essential formulas and properties, along with some discussions relating to the tensor decomposition algorithms.

### 2.1.1 Singular value decomposition

Formally, the singular value decomposition of a matrix $\mathbf{A} \in \mathbb{R}^{n \times m}$ returns real matrices $\mathbf{U}, \mathbf{V}$ and a diagonal matrix with non-negative elements $\mathbf{\Sigma}$ which are satisfied:

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T \tag{1}$$

where

- $\mathbf{U} \in \mathbb{R}^{n \times n}$ and $\mathbf{V} \in \mathbb{R}^{m \times m}$ are orthogonal matrices and contain the left and right singular vectors of $\mathbf{A}$ as their columns, respectively.
- $\mathbf{\Sigma} \in \mathbb{R}^{n \times m}$ is the diagonal matrix that contains singular values $\sigma_i$ in decreasing order.

According to [11], the SVD can explicitly show the rank and the null space of a matrix. For the low-rank $r \leq \min\{m, n\}$, there are $r$ vectors in the set of $\mathbf{U} = \{u_1, u_2, \ldots, u_r\}$ spans the column space of $\mathbf{A}$ and the rest $(m - r)$ vectors from $\mathbf{U}_0 = \{u_{r+1}, u_{r+2}, \ldots, u_m\}$ is the left null space of $\mathbf{A}$. The reason is when $r > 0$ then $\mathbf{U}_0^T\mathbf{A} = \mathbf{0}$ and a column in matrix $\mathbf{A}$ is the weighted linear combination from the vector in set $\mathbf{U}$, then this set spans the column space of $\mathbf{A}$. The same definition for the right singular matrix $\mathbf{V}$.

SVD can equivalently written as:

$$\mathbf{A} = \sum_{i=1}^{r} \sigma_i u_i v_i^T \tag{2}$$

where each outer product $u_i v_i^T$ returns a rank-one matrix, so $\mathbf{A}$ is the weighted combination of rank-1 matrices. Pairs of eigenvectors associated with higher eigenvalues contains more information to reconstruct the original matrix.

In dimension reduction, the data of $n$ samples and $m$ features is stored in a matrix $\mathbf{A}$. By using the SVD decomposition, we can efficiently approximate $\mathbf{A}$ by lower

$$\mathbf{A} =$$

$u_1$ $u_r u_{r+1}$ $u_m$ $\sigma_1$ $\sigma_r$ 0 0 $v_1^T$ $v_r^T$ $v_{r+1}^T$ $v_n^T$ row($A$) null($A$) col($A$) null($A$)

Figure 2: Singular Vector Decomposition components

dimensional matrices with a high reconstruction accuracy. This property is proof in the Eckhart-Young theorem [12] which states that the SVD provides the best least square approximation for a matrix. In this case, the left singular matrix $\mathbf{U} \in \mathbb{R}^{n \times k}$ is the solution and new feature values for each sample is the coordinate of this sample in the subspace that contains the largest amount of data variance. Particularly, if $k = 2$, the dataset of $n$ points can be visualized in a 2 dimensional plan. This is due to $\mathbf{U}$ contains eigenvectors of the covariance matrix $\mathbf{A}\mathbf{A}^T$ of the data in $\mathbf{A}$.

$$\mathbf{A}\mathbf{A}^T = (\mathbf{U}\mathbf{\Sigma}\mathbf{V}^T)(\mathbf{V}\mathbf{\Sigma}\mathbf{U}^T) = \mathbf{U}\mathbf{\Sigma}^2\mathbf{U}^T$$

### 2.1.2 Matrix factorization interpretation

The matrix factorization (MF) is the problem of approximating a matrix by the product of two matrices. The task is to find these matrices and typically they have a lower dimension than original data dimension.

$$\mathbf{A} \approx \mathbf{W} \times \mathbf{H} \tag{3}$$

where $\mathbf{A} \in \mathbb{R}^{n \times m}$ and $\mathbf{W} \in \mathbb{R}^{n \times k}$, $\mathbf{H} \in \mathbb{R}^{k \times m}$ and $k \leq \min(m, n)$. For each $i$th column $A(:, i)$ in matrix $\mathbf{A}$, we have:

$$A(:, i) \approx WH(:, i) \tag{4}$$

or each column of the original data is reconstructed by weighted combination of $\mathbf{W}$ matrix with a corresponding column in $\mathbf{H}$ matrix.

In the other hand, $\mathbf{W}$ can be seen as the *feature* matrix and $\mathbf{H}$ is the *coefficient* matrix. A column $A(:, i)$ is the combination of new *feature* matrix $\mathbf{W}$ with weights from corresponding column in the *coefficient* matrix $\mathbf{H}$. In detail, the set of columns in $\mathbf{W} = \{w_1, w_2, \ldots, w_k\}$, then the data column reconstruction as follow:

$$A(:, i) \approx \sum_{j=1}^{k} W(:, j) \times H(j, i)$$

For a specific data element $A_{ij}$, it is the weighted combination of row $i$th of $\mathbf{W}$ and column $j$th of $\mathbf{H}$.

$$A_{ij} \approx W(i,:) \times H(:,j) \quad \text{or} \quad A_{ij} \approx W[i] \times H[j]$$

where $W[i] = W(i,:) \in \mathbb{R}^{1\times k}$ and $H[j] = H(:,j) \in \mathbb{R}^{k\times 1}$, or having the same number of dimension or *low-rank* factors.

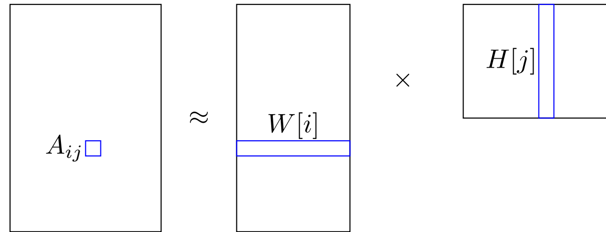



Figure 3: Matrix factorization interpretation for an element

By this kind of representation, each original data element is reconstructed by the production of two vectors in the low-rank space. Then, with $(n+m)$ indices for rows and columns, each index need a vector $\in \mathbb{R}^k$ to store its coefficients, the memory complexity is $\mathcal{O}(k(m+n))$.

In data mining applications, we can analyse properties of new data features through matrix $\mathbf{W}$ and the relations among original data features by using $\mathbf{H}$ [13]. For example, in text mining, the non-negative matrix factorization algorithms [14] are well developed for analyzing text data. The data is a *document-term* matrix where rows are documents and columns are the information of terms in each document. By finding a factorized representation, the matrix is decomposed into *document-feature* and *feature-term* matrices, in which a *feature* stands for the lower dimensional approximation or data contents. Typically, the *document-feature* information is used for clustering documents or *topic modeling*.

Interestingly, the matrix factorization form can be achieved by using SVD. The SVD decomposition of matrix $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$, then the MF solution is:

$$\mathbf{W} = \mathbf{U}_k \quad \texttt{and} \quad \mathbf{H} = \mathbf{\Sigma}_k\mathbf{V}_k^T \tag{5}$$

where $\mathbf{U}_k, \mathbf{\Sigma}_k, \mathbf{V}_k$ is the reduction of original matrix to contain only first largest $k$ eigenvalues and eigenvectors.

## 2.2 Tensors definition

Tensor is the multidimensional array of numerical values. Formally, an *order-d* tensor is a $d$-dimensional array. For example, a scalar is an *order-0* tensor, a vector and a matrix are an *order-1* and *order-2* tensor, respectively. Additionally, the tensor formulation is a compact way to represent a multidimensional dataset. For example, a collection of documents that contains authors, terms, and publish dates can be seen as a *order-3* tensor. A colour image with 3 channels: R, G, B, can be seen as a tensor of [height $\times$ weight $\times$ colour]. Due to the tensor is the generalisation of the matrix,

the tensor decomposition is a generalisation of the low-rank decomposition for a matrix. This section will discuss the definition of tensor operators and decomposition algorithms which focus on CP decomposition (CANDECOMP/PARAFAC), Tucker decomposition [15], and Tensor Train decomposition [4].

**Tensor notation** The order of a tensor is in *modes*; for example, *mode-1* of a matrix is its rows and columns are in *mode-2*. A general *d-order* real tensor is $\mathcal{A} \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_d}$ and an element is $\mathcal{A}(i_1, i_2, \ldots, i_d)$ where the index range for mode $k^{th}$ is $i_k \in [1 : n_k]$. The colon notation ":" is used to indicate all elements in a *mode* or a specific index subset.

For example, the cube in Figure 4, a 3 dimensional array $\mathcal{A} \in \mathbb{R}^{n_1 \times n_2 \times \times n_3}$ is depicted as a cubic. Figure 5 shows the *order-3* tensor: $\mathcal{A} \in \mathbb{R}^{3 \times 4 \times 2}$ in which *mode-1* size is 3 and 4, 2 is for *mode-2*, *mode-3*, respectively.

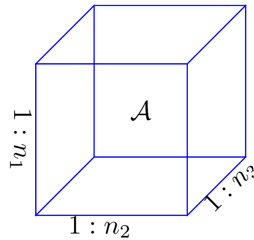



Figure 4: An *order-3* tensor

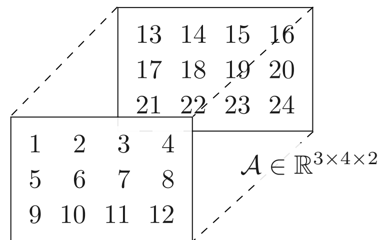



Figure 5: An *order-3* tensor with specific values

**Tensor parts: fiber and slice** Parts of array or subarrays are made by fixing a subset of array indices. In a matrix or a *mode-2* tensor, these parts are rows and columns and denoted as $A(:, i)$ and $A(j, :)$ for column *i-th* and row *j-th*, respectively. In a tensor, due to it has multi-dimensional indices, many possible indexing ways for using colon notation are available. This paragraph describe two special subarrays in tensors which are *fiber* and *slice*.

A *fiber* is a vector that is obtained by fixing all indices but one. For example, fibers are depicted in an *order-3* tensor in Figure 6, 7.

A *slice* is a vector that is obtained by fixing all indices but two. For example, in Figure 8, a matrix is obtained when choosing all elements of two dimension and an index is fixed in the third dimension.

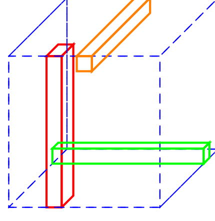


Figure 6: A mode-1 fiber $\mathcal{A}(:, j, k)$, a mode-2 fiber $\mathcal{A}(i, :, k)$, and a mode-3 fiber $\mathcal{A}(i, j, :)$

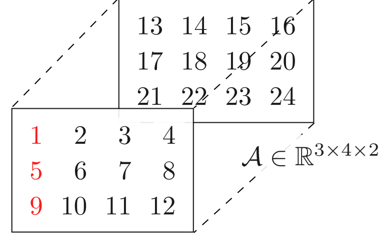



Figure 7: A *3-order* tensor fiber: $\mathcal{A}(:, 1, 1)$

**Rank-one and diagonal tensor** Analogically to the rank-one matrix, the *d-order* rank-one tensor is formed by the outer product [11] of $d$ vectors. For $\mathcal{A} \in \mathbb{R}^{n_1\times n_2\times\cdots\times n_d}$:

$$\mathcal{A} = a^{(1)} \circ a^{(2)} \circ \cdots \circ a^{(d)}$$

where $a^{(k)}$ is the *k-th* vector. Thus, an element $\mathcal{A}(i_1, i_2, \ldots, i_d)$ is the product of corresponding elements in $d$ vectors:

$$\mathcal{A}(i_1, i_2, \ldots, i_d) = a^{(1)}(i_1)a^{(2)}(i_2)\ldots a^{(d)}(i_d)$$

For example, Figure 9 visualizes the third-order rank-one tensor for $\mathcal{A} = a \circ b \circ c$.

A diagonal tensor $\mathcal{A} \in \mathbb{R}^{n_1\times n_2\times\cdots\times n_d}$ that an element $\mathcal{A}(i_1, i_2, \ldots, i_d)$ is 1 if and only if $i_1 = i_2 = \cdots = i_d$.

**Tensor norm** The norm of a tensor $\mathcal{A} \in \mathbb{R}^{n_1\times n_2\times\cdots\times n_d}$ is the square root of the sum of squares of all elements:

$$||\mathcal{A}||_F = \sqrt{\sum_{i_1}^{n_1}\sum_{i_2}^{n_2}\cdots\sum_{i_d}^{n_d}\mathcal{A}(i_1, i_2, \ldots, i_d)^2}$$

**Matricization: *mode-k* unfolding** In tensor computations, the typical step is tensor *unfoldings* or *flattening* which rearranges a tensor to a simpler form: a matrix, to utilize it well-foundation computations and discover patterns in the matrix from. According to [11], there are three main reasons to do that:

- Tensor operations can be reformulated as matrix operators through multiple unfolding steps.

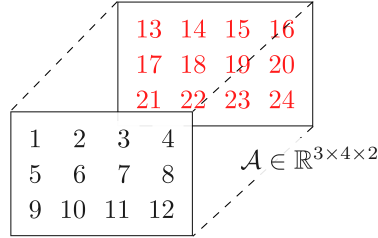



Figure 8: A slice of the *order-3* tensor: $\mathcal{A}(:,:,2)$

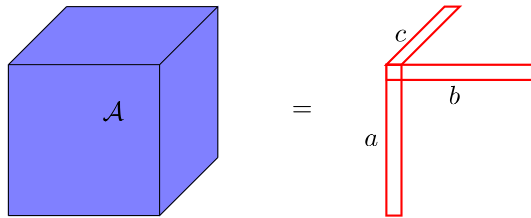



Figure 9: Rank-one third-order tensor

- An iterative tensor optimisation framework contains one or more unfolding steps.
- Hidden patterns of a tensor sometimes can be discovered through unfolding it into matrix forms.

There are several possible ways to assemble a tensor to a matrix. For example, we can rearrange a tensor $\mathcal{A} \in \mathbb{R}^{3\times4\times2}$ in Figure 5 to a $[12 \times 2]$ matrix or a $[4 \times 6]$ matrix. In this section, we discuss an important family of tensor unfolding which is the *mode-k* unfolding. $\mathcal{A}_{(k)}$ denotes the *mode-k* unfolding of the tensor $\mathcal{A} \in \mathbb{R}^{n_1\times n_2\times\cdots\times n_d}$. Formally, the size of $\mathcal{A}_{(k)}$ is $n_k$-by-$(N/n_k)$ where $N = n_1 \times n_2 \times \cdots \times n_d$ and it is the stacking of *mode-k* fibers in the column order. A tensor element $(i_1, i_2, \ldots, i_d)$ maps to the matrix element $(i_k, j)$ by:

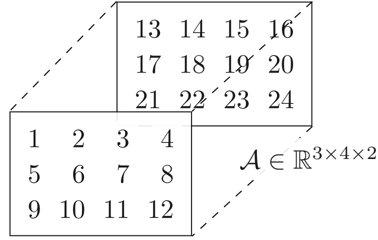
$$j = 1 + \sum_{l=1;\ l\neq k}^{n_d} (i_l - 1)J_l \quad \text{where} \quad J_l = \prod_{l=1;\ l\neq k}^{n_l} n_l \tag{6}$$

$$= 1 + (i_1 - 1)n_1 + (i_2 - 1)n_1n_2 + \cdots + (i_d - 1)(n_1n_2 \ldots n_d)$$

Intuitively, this indexing system is analogous to the matrix indexing system by adding columns in other modes for flattening purpose. On the other hand, this step fixes one index system for a mode and stacking mode by mode for others. For the *mode-k* unfolding, the index in the $k$ position is preserved and all others are folded into one index. For example, the *mode-1* matricization for a tensor in Figure 5 is visualized in Figure 10. Following that, its *mode-2* and *mode-3* unfolding is in Figure 11, 12.

In detail, Figure 13 shows the unfolding example is in the indices format. Then, the *mode-1* unfolding is:

$$\mathcal{A}_{(1)} = \begin{bmatrix} a_{111} & a_{121} & a_{131} & a_{141} & a_{112} & a_{122} & a_{132} & a_{142} \\ a_{211} & a_{221} & a_{231} & a_{241} & a_{212} & a_{222} & a_{232} & a_{242} \\ a_{311} & a_{321} & a_{331} & a_{341} & a_{312} & a_{322} & a_{332} & a_{342} \end{bmatrix}$$




$\mathcal{A}_{(1)} =$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 13 | 14 | 15 | 16 |
| 5 | 6 | 7 | 8 | 17 | 18 | 19 | 20 |
| 9 | 10 | 11 | 12 | 21 | 22 | 23 | 24 |

Figure 10: The mode-1 unfolding example

$\mathcal{A}_{(2)} =$

| | | | | | |
|---|---|---|---|---|---|
| 1 | 5 | 9 | 13 | 17 | 21 |
| 2 | 6 | 10 | 14 | 18 | 22 |
| 3 | 7 | 11 | 15 | 19 | 23 |
| 4 | 8 | 12 | 16 | 20 | 24 |

Figure 11: The mode-2 unfolding example

$$\mathcal{A}_{(2)} = \begin{bmatrix} a_{111} & a_{211} & a_{311} & a_{112} & a_{212} & a_{312} \\ a_{121} & a_{221} & a_{321} & a_{122} & a_{222} & a_{322} \\ a_{131} & a_{231} & a_{331} & a_{132} & a_{232} & a_{332} \\ a_{141} & a_{241} & a_{341} & a_{142} & a_{242} & a_{342} \end{bmatrix}$$

$$\mathcal{A}_{(3)} = \begin{bmatrix} a_{111} & a_{211} & a_{311} & a_{121} & a_{221} & a_{321} & a_{131} & a_{231} & a_{331} & \dots \\ a_{112} & a_{212} & a_{312} & a_{122} & a_{222} & a_{322} & a_{132} & a_{232} & a_{332} & \dots \end{bmatrix}$$

**Tensor product: *mode-k* product** This section considers the tensor-matrix multiplication via the *mode-k* of a tensor. In general, the multiplication of two tensors can be derived as between matrices; this operator is called *tensor contraction* which is clearly described in Chapter 12.4.9 of [11]. The *mode-k* or *modal* product is the special case and important family of the tensor contraction. This product operator needs a tensor, a matrix, and a specific mode. Formally, given a tensor $\mathcal{A} \in \mathbb{R}^{n_1\times n_2\times\cdots\times n_d}$ and a matrix $\mathbf{M} \in \mathbb{R}^{m_k\times n_k}$, the *mode-k* product is the matrix multiplication of the *mode-k* unfolding $\mathcal{A}_{(k)}$ and $\mathbf{M}$, and returns a tensor $\mathcal{Y}$:

$$\mathcal{Y}_{(k)} = \mathbf{M} \times \mathcal{A}_{(k)} \tag{7}$$

or according to [15], the shorted form of this equation is $\mathcal{Y} = \mathcal{A} \times_{(k)} \mathbf{M}$. The configuration of these terms are the tensor $\mathcal{A} \in \mathbb{R}^{n_1\times n_2\times\dots n_{k-1}\times n_k\times n_{k+1}\times\cdots\times n_d}$, the matrix $M \in \mathbb{R}^{m_k\times n_k}$, the tensor $\mathcal{Y} \in \mathbb{R}^{n_1\times n_2\times\dots n_{k-1}\times m_k\times n_{k+1}\times\cdots\times n_d}$. In the element-wise description, an element in $\mathcal{Y}$ is the inner product of two fibers: $M(\alpha_k, :)$ and

$$\mathcal{A}_{(3)} = \begin{array}{|cccccccccccc|} \hline 1 & 5 & 9 & 2 & 6 & 10 & 3 & 7 & 11 & 4 & 8 & 12 \\ 13 & 17 & 21 & 14 & 18 & 22 & 15 & 19 & 23 & 16 & 20 & 24 \\ \hline \end{array}$$

Figure 12: The mode-3 unfolding example

$$\begin{array}{|cccc|} \hline a_{112} & a_{122} & a_{132} & a_{142} \\ a_{212} & a_{222} & a_{232} & a_{242} \\ a_{312} & a_{322} & a_{332} & a_{342} \\ \hline \end{array}$$

$$\begin{array}{|cccc|} \hline a_{111} & a_{121} & a_{131} & a_{141} \\ a_{211} & a_{221} & a_{231} & a_{241} \\ a_{311} & a_{321} & a_{331} & a_{341} \\ \hline \end{array} \quad \mathcal{A} \in \mathbb{R}^{3\times 4\times 2}$$

Figure 13: Tensor $\mathcal{A}$ of indices

$\mathcal{A}(i_1, \ldots, i_{k-1}, :, i_{k+1}, \ldots, i_d)$ for the *mode-k* product on the $k$th dimension.

$$\mathcal{Y}(i_1, \ldots, i_{k-1}, \alpha_k, i_{k+1}, \ldots, i_d) = \sum_{j=1}^{n_k} M(\alpha_k, j)\mathcal{A}(i_1, \ldots, i_{k-1}, j, i_{k+1}, \ldots, i_d) \quad (8)$$

with $\alpha_k \in [1 : m_k]$

For example, if $\mathcal{A} \in \mathbb{R}^{3\times 4\times 2}$ is in Figure 13 and $\mathbf{M} \in \mathbb{R}^{2\times 4}$, then $\mathcal{Y}_{(2)} = \mathbf{M} \times \mathcal{A}_{(2)}$:

$$\mathcal{Y}_{(2)} = \begin{bmatrix} a_{111} & a_{211} & a_{311} & a_{112} & a_{212} & a_{312} \\ a_{121} & a_{221} & a_{321} & a_{122} & a_{222} & a_{322} \end{bmatrix}$$

$$= \begin{bmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \end{bmatrix} \times \begin{bmatrix} a_{111} & a_{211} & a_{311} & a_{112} & a_{212} & a_{312} \\ a_{121} & a_{221} & a_{321} & a_{122} & a_{222} & a_{322} \\ a_{131} & a_{231} & a_{331} & a_{132} & a_{232} & a_{332} \\ a_{141} & a_{241} & a_{341} & a_{142} & a_{242} & a_{342} \end{bmatrix}$$

Intuitively, this multiplication replaces the content of the dimension $k$ of $\mathcal{A}$ by weighted combination of this mode information with a matrix $\mathbf{M}$. For the sequences of multiplications, when assuming component dimensions matches, the final result is independent the order [15].

$$\mathcal{A} \times_k \mathbf{M} \times_h \mathbf{N} = \mathcal{A} \times_h \mathbf{N} \times_k \mathbf{M} \quad (h \neq k) \quad (9)$$

where $\mathbf{N} \in \mathbb{R}^{z_h \times n_h}$. Similarly, when $\mathbf{M}$ is a vector or $\mathbf{M} \in \mathbb{R}^{1\times n_k}$, the *mode-k* product between a tensor and a vector returns a $d-1$ dimensional tensor. It is easy to show that, the returned matrix $\mathcal{Y} \in \mathbb{R}^{n_1\times n_2\times \ldots n_{k-1}\times 1\times n_{k+1}\times \cdots \times n_d}$ and the equivalent representation is $\mathcal{Y} \in \mathbb{R}^{n_1\times n_2\times \ldots n_{k-1}\times n_{k+1}\times \cdots \times n_d}$ .

**Vec operator for a matrix** The vector form is the common representation of a matrix and a tensor with some proper rearrangements. This vector representation or *vec operator* allows to write matrices and tensors in the same form and thus faster the equation reduction step.

The operator $\texttt{vec}$ for a matrix is the reshaping of a matrix $\mathbf{X} \in \mathbb{R}^{m\times n}$ to a vector of $\mathbb{R}^{nm\times 1}$, by stacking columns of $\mathbf{X}$ .

$$\texttt{vec}(\mathbf{X}) = \begin{bmatrix} X(:,1) \\ X(:,2) \\ \dots \\ X(:,n) \end{bmatrix}$$

In detail, for a matrix $\mathbf{X} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix}$ then $\texttt{vec}(\mathbf{X}) = \begin{bmatrix} a_{11} \\ a_{21} \\ \hline a_{12} \\ a_{22} \\ \hline a_{13} \\ a_{23} \end{bmatrix}$

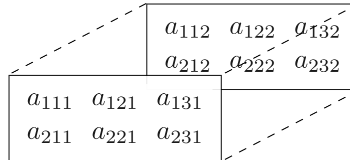
**Vec operator for a tensor** is a generalisation of the matrix vectorization. This operator stacks columns from the last dimensional index as follow, for a *d-dimensional* tensor $\mathcal{A} \in \mathbb{R}^{n_1\times n_2\times\cdots\times n_d}$:

$$\texttt{vec}(\mathcal{A}) = \begin{bmatrix} \texttt{vec}(\mathcal{A}^{(1)}) \\ \texttt{vec}(\mathcal{A}^{(2)}) \\ \dots \\ \texttt{vec}(\mathcal{A}^{(n_d)}) \end{bmatrix}$$

where $\mathcal{A}^{(k)}$ is the $(d-1)$-*dimensional* tensor $\in \mathbb{R}^{n_1\times n_2\times\cdots\times n_{d-1}}$, in which:

$$\mathcal{A}^{(k)}(i_1, i_2, \dots, i_{d-1}) = \mathcal{A}(i_1, i_2, \dots, i_{d-1}, k)$$

Note that, $\mathcal{A}^{(k)}$ is different with $\mathcal{A}_{(k)}$ which is the *mode-k* matrix. $\mathcal{A}^{(k)}$ is the copy of original tensor when the last index $i_d$ is fixed for a specific value $k$. For a $d$ dimensional tensor, this definition is recursive from $\mathcal{A}^{(k)}$ of $\mathcal{A}$, to $(\mathcal{A}^{(k')})^{(k)}$ of $\mathcal{A}^{(k)}$ until fixing $d-1$ indexes to get a scalar, then go back to stack columns of higher order tensor columns. For example, the $\texttt{vec}$ of an *order-3* tensor $\mathcal{A}^{2\times 3\times 2}$:



is given by

$$\mathtt{vec}(\mathcal{A}) = \begin{bmatrix} \mathtt{vec}(\mathcal{A}^{(1)}) \\ \mathtt{vec}(\mathcal{A}^{(2)}) \end{bmatrix} = \begin{bmatrix} \mathtt{vec}(\mathcal{A}(:,:,1)) \\ \mathtt{vec}(\mathcal{A}(:,:,2)) \end{bmatrix} = \begin{bmatrix} a_{111} \\ a_{211} \\ \hline a_{121} \\ a_{221} \\ \hline a_{131} \\ a_{231} \\ \hline a_{112} \\ a_{212} \\ \hline a_{122} \\ a_{222} \\ \hline a_{132} \\ a_{232} \end{bmatrix}$$

**Outer product** The outer product $\circ$ is the product between two coordinate vectors. Formally, for vectors $u \in \mathbb{R}^m, v \in \mathbb{R}^n$ then $\mathbf{A} = uv^T = u \circ v$ is a rank-1 matrix $\in \mathbb{R}^{m \times n}$. For $u = [u_1, u_2, u_3]^T$ and $v = [v_1, v_2]^T$:

$$\mathbf{A} = uv^T = \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} \begin{bmatrix} v_1 & v_2 \end{bmatrix} = \begin{bmatrix} u_1v_1 & u_1v_2 \\ u_2v_1 & u_2v_2 \\ u_3v_1 & u_3v_2 \end{bmatrix} = \left[ v_1 \begin{pmatrix} u_1 \\ u_2 \\ u_3 \end{pmatrix}, v_2 \begin{pmatrix} u_1 \\ u_2 \\ u_3 \end{pmatrix} \right] \tag{10}$$

The outer product of more than 2 vectors returns a tensor. For example, the outer product of 3 vectors returns an rank-1 *order-3* tensor, in which:

$$\mathcal{A} = u \circ v \circ w \quad \text{and} \quad \mathcal{A}(i,j,k) = u(i)v(j)w(k) \tag{11}$$

where $u \in \mathbb{R}^m, v \in \mathbb{R}^n, w \in \mathbb{R}^k$ and $\mathcal{A} \in \mathbb{R}^{m \times n \times k}$. For example, if $u = [u_1, u_2]^T$, $v = [v_1, v_2]^T$, and $w = [w_1, w_2]^T$, the $u \circ v \circ w$ is:

$$\begin{matrix} u_1v_1w_2 & u_1v_2w_2 \\ u_2v_1w_2 & u_2v_2w_2 \end{matrix}$$

$$\begin{matrix} u_1v_1w_1 & u_1v_2w_1 \\ u_2v_1w_1 & u_2v_2w_1 \end{matrix}$$

**Kronecker product** In the case of scalar-matrix multiplication and doing this for many scalars in order to obtain a new block matrix, we need to use the Kronecker operator. Formally, for matrices $\mathbf{B} \in \mathbb{R}^{m_1 \times n_1}$ and $\mathbf{C} \in \mathbb{R}^{m_2 \times n_2}$, the Kronecker product $\mathbf{D} = \mathbf{B} \otimes \mathbf{C} \in \mathbb{R}^{m_1 n_1 \times m_2 n_2}$ is a block matrix, in which a block at $(i,j)$ is a matrix. Thus, $\mathbf{D}$ is the $m_1$-by-$n_1$ block matrix whose $(i,j)$ block is matrix $b(i,j)\mathbf{C}$ of the size $m_2$-by-$n_2$. For example, if $\mathbf{B} \in \mathbb{R}^{3 \times 2}$ and $\mathbf{C} \in \mathbb{R}^{2 \times 2}$ then:

$$\mathbf{D} = \mathbf{B} \otimes \mathbf{C} = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{bmatrix} \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \left[\begin{array}{c|c} b_{11}\mathbf{C} & b_{12}\mathbf{C} \\ \hline b_{21}\mathbf{C} & b_{22}\mathbf{C} \\ \hline b_{31}\mathbf{C} & b_{32}\mathbf{C} \end{array}\right] =$$

$$\left[\begin{array}{cc|cc} b_{11}c_{11} & b_{11}c_{12} & b_{12}c_{11} & b_{12}c_{12} \\ b_{11}c_{21} & b_{11}c_{22} & b_{12}c_{21} & b_{12}c_{22} \\ \hline b_{21}c_{11} & b_{21}c_{12} & b_{22}c_{11} & b_{22}c_{12} \\ b_{21}c_{21} & b_{21}c_{22} & b_{22}c_{21} & b_{22}c_{22} \\ \hline b_{31}c_{11} & b_{31}c_{12} & b_{32}c_{11} & b_{32}c_{12} \\ b_{31}c_{21} & b_{31}c_{22} & b_{32}c_{21} & b_{32}c_{22} \end{array}\right] \tag{12}$$

The power of Kronecker product comes from its structure properties, fast practical algorithms, and connection between tensor and matrix computation [16]. Firstly, the vector form of matrices and tensors operators can be rewritten thanks to the Kronecker product. For matrices, Equation (10) is equivalent with:

$$\mathbf{A} = u \circ v = \begin{bmatrix} u_1v_1 & u_1v_2 \\ u_2v_1 & u_2v_2 \\ u_3v_1 & u_3v_2 \end{bmatrix} = \begin{bmatrix} v_1 \begin{pmatrix} u_1 \\ u_2 \\ u_3 \end{pmatrix}, v_2 \begin{pmatrix} u_1 \\ u_2 \\ u_3 \end{pmatrix} \end{bmatrix} \Rightarrow \mathtt{vec}(\mathbf{A}) = \begin{bmatrix} v_1u_1 \\ v_1u_2 \\ v_1u_3 \\ v_2u_1 \\ v_2u_2 \\ v_2u_3 \end{bmatrix} = \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} \otimes \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix}$$

then

$$\mathtt{vec}(u \circ v) = v \otimes u$$

When the outer product returns a tensor, we do the same procedure for the example of Equation (11). For $u = [u_1, u_2]^T$, $v = [v_1, v_2]^T$, and $w = [w_1, w_2]^T$:

$$\mathcal{A} = u \circ v \circ w = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \circ \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} \circ \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} \Rightarrow \mathtt{vec}(\mathcal{A}) = \begin{bmatrix} u_1v_1w_1 \\ u_2v_1w_1 \\ u_1v_2w_1 \\ u_2v_2w_1 \\ u_1v_1w_2 \\ u_2v_1w_2 \\ u_1v_2w_2 \\ u_2v_2w_2 \end{bmatrix} =$$

$$\begin{bmatrix} w_1 \\ w_2 \end{bmatrix} \otimes \begin{bmatrix} u_1v_1 \\ u_2v_1 \\ u_1v_2 \\ u_2v_2 \end{bmatrix} = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} \otimes \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} \otimes \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}$$

then

$$\mathtt{vec}(u \circ v \circ w) = w \otimes v \otimes u$$

As a tensor is a multi-dimension array, each *mode-k* matrix also can be described by the Kronecker product. By using the above example, $\mathcal{A}_{(1)}$ is:

$$\mathcal{A}_{(1)} = \begin{bmatrix} u_1v_1w_1 & u_1v_2w_1 & u_1v_1w_2 & u_1v_2w_2 \\ u_2v_1w_1 & u_2v_2w_1 & u_2v_1w_2 & u_2v_2w_2 \end{bmatrix} = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \otimes \begin{bmatrix} v_1w_1 & v_2w_1 & v_1w_2 & v_2w_2 \end{bmatrix}$$

$$= \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \otimes \begin{bmatrix} v_1 w_1 \\ v_2 w_1 \\ v_1 w_2 \\ v_2 w_2 \end{bmatrix}^T = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \otimes \left( \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} \otimes \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} \right)^T$$

then

$$\mathcal{A}_{(1)} = u \otimes (w \otimes v)^T$$

These rules describe the Kronecker representation among tensors and matrices:

$$\mathcal{A} = u^{(1)} \circ u^{(2)} \circ \dots u^{(d)} \in \mathbb{R}^{n_1 \times n_2 \times \dots \times n_d}$$

then

$$\texttt{vec}(\mathcal{A}) = u^{(d)} \otimes u^{(d-1)} \cdots \otimes u^{(2)} \otimes u^{(1)} \tag{13}$$

$$\mathcal{A}_{(k)} = u^{(k)} \otimes \left( u^{(d)} \otimes \dots u^{(k+1)} \otimes u^{(k-1)} \cdots \otimes u^{(2)} \otimes u^{(1)} \right)^T \tag{14}$$

A sequence of matrix-matrix multiplications can be represented in the Kronecker product with `vec` operator. For matrices: $\mathbf{Y} \in \mathbb{R}^{m_2 \times m_1}$, $\mathbf{C} \in \mathbb{R}^{m_2 \times n_2}$, $\mathbf{X} \in \mathbb{R}^{n_2 \times n_1}$, and $\mathbf{B} \in \mathbb{R}^{m_1 \times n_1}$

$$\mathbf{Y} = \mathbf{C}\mathbf{X}\mathbf{B}^T \quad \Leftrightarrow \quad \texttt{vec}(\mathbf{Y}) = (\mathbf{B} \otimes \mathbf{C})\texttt{vec}(\mathbf{X}) \tag{15}$$

A very clear example for Equation (15) is in Chapter 1.3.5 of [11]. From the Equation (12), we can see that the structure property of resulted matrix from $\mathbf{B} \otimes \mathbf{C}$ is dependent on the structure of $\mathbf{B}$. Then if $\mathbf{B}$ has a band structure (diagonal, tridiagonal, lower/upper triangular ) then the $\mathbf{B} \otimes \mathbf{C}$ retains the same structure as $\mathbf{B}$.

From [16], some notable properties of the *Kronecker product is*:

$$\begin{aligned} (\mathbf{B} \otimes \mathbf{C})^T &= \mathbf{B}^T \otimes \mathbf{C}^T \\ (\mathbf{B} \otimes \mathbf{C})(\mathbf{D} \otimes \mathbf{F}) &= (\mathbf{B}\mathbf{D}) \otimes (\mathbf{C}\mathbf{F}) \\ (\mathbf{B} \otimes \mathbf{C}) \otimes \mathbf{D} &= \mathbf{B} \otimes (\mathbf{C} \otimes \mathbf{D}) \\ (\mathbf{B} \otimes \mathbf{C})^\dagger &= \mathbf{B}^\dagger \otimes \mathbf{C}^\dagger \end{aligned} \tag{16}$$

**Hadamard product** The Hadamard product definition is quite straight-forward, it is the *elementwise product* for matrices of similar size. Formally, the Hadamard product between two matrices $\mathbf{B}, \mathbf{C} \in \mathbb{R}^{m \times n}$ is:

$$\mathbf{A} = \mathbf{B} * \mathbf{C} \quad \text{and} \quad \mathbf{A} \in \mathbb{R}^{m \times n}$$

$$A(i,j) = B(i,j)C(i,j)$$

Shortly, an element of $\mathbf{A}$ is the multiplication of 2 corresponding elements from $\mathbf{B}$ and $\mathbf{C}$ in terms of row and column index.

**Khatri-Rao product** is the special case of Kronecker product when two matrices contain the same number of columns. Formally, the Khatri-Rao product between two matrices $\mathbf{B} \in \mathbb{R}^{m_1 \times n}$ and $\mathbf{C} \in \mathbb{R}^{m_2 \times n}$ is:

$$\mathbf{A} = \mathbf{B} \odot \mathbf{C} \quad \text{and} \quad \mathbf{A} \in \mathbb{R}^{m_1 m_2 \times n}$$

This product is the *columwise product* where matched column are multiplied by the Kronecker product. For example, the separation of matrices $\mathbf{B}, \mathbf{C}$ into columns are:

$$\mathbf{B} = [b_1 \mid b_2 \mid \ \dots \ \mid b_n] \quad \text{and} \quad \mathbf{C} = [c_1 \mid c_2 \mid \ \dots \ \mid c_n]$$

where $n$ columns are available and each $b_i \in \mathbb{R}^{m_1 \times 1}$ and $c_i \in \mathbb{R}^{m_2 \times 1}$. We can represent the Khatri-Rao product to get $\mathbf{A}$ as follow:

$$\mathbf{A} = [b_1 \otimes c_1 \mid b_2 \otimes c_2 \mid \ \dots \ \mid b_n \otimes c_n]$$

where $a_i = b_i \otimes c_i$ return a column vector of size $m_1$-by-$m_2$ and the number columns of $\mathbf{A}$ remains $n$.

According to [15], the properties of Khatri-Rao product:

$$\begin{aligned} \mathbf{A} \odot \mathbf{B} \odot \mathbf{C} &= (\mathbf{A} \odot \mathbf{B}) \odot \mathbf{C} = \mathbf{A} \odot (\mathbf{B} \odot \mathbf{C}) \\ (\mathbf{A} \odot \mathbf{B})^T (\mathbf{A} \odot \mathbf{B}) &= \mathbf{A}^T \mathbf{A} * \mathbf{B}^T \mathbf{B} \\ (\mathbf{A} \odot \mathbf{B})^\dagger &= ((\mathbf{A}^T \mathbf{A}) * (\mathbf{B}^T \mathbf{B}))^\dagger (\mathbf{A} \odot \mathbf{B})^T \end{aligned} \tag{17}$$

**Multilinear product** is the product among one tensor and many matrices where giving a tensor $\mathcal{A} \in \mathbb{R}^{n_1 \times n_2 \times \dots \times n_d}$, a tensor $\mathcal{S} \in \mathbb{R}^{r_1 \times r_2 \times \dots \times r_d}$, and $d$ matrices $\mathbf{M}_k \in \mathbb{R}^{n_d \times r_d}$. Then, the multilinear product among tensor $\mathcal{S}$ and multiple matrices $\mathbf{M}$ creates a tensor $\mathcal{A}$:

$$\mathcal{A} = \mathcal{S} \times_1 \mathbf{M}_1 \times_2 \mathbf{M}_2 \cdots \times_d \mathbf{M}_d \tag{18}$$

$$\mathcal{A}(i_1, i_2, \dots, i_d) = \sum_{j_1}^{r_1} \sum_{j_2}^{r_2} \cdots \sum_{j_d}^{r_d} \mathcal{S}(j_1, j_2, \dots, j_d) \mathbf{M}_1(i_1, j_1) \mathbf{M}_2(i_2, j_2) \dots \mathbf{M}_d(i_d, j_d)$$

or equivalent with

$$\mathtt{vec}(\mathcal{A}) = (\mathbf{M}_d \otimes \mathbf{M}_{d-1} \otimes \cdots \otimes \mathbf{M}_1) \mathtt{vec}(\mathcal{S}) \tag{19}$$

regarding to the *mode-k* matrix of $\mathcal{A}$

$$\mathcal{A}_{(k)} = \mathbf{M}_k \cdot \mathcal{S}_{(k)} \cdot (\mathbf{M}_d \otimes \mathbf{M}_{d-1} \otimes \dots \mathbf{M}_{k+1} \otimes \mathbf{M}_{k-1} \cdots \otimes \mathbf{M}_1)^T \tag{20}$$

In order to make these definitions clear, we will cover an example for order-2 tensor and make the generalisation:

In the matrices case, the multilinear product according to these matrices: $\mathbf{A} \in \mathbb{R}^{n_1 \times n_2}$, $\mathbf{S} \in \mathbb{R}^{r_1 \times r_2}$, $\mathbf{M}_1 \in \mathbb{R}^{n_1 \times r_1}$, and $\mathbf{M}_2 \in \mathbb{R}^{n_2 \times r_2}$. Then

$$\mathbf{A} = \mathbf{S} \times_1 \mathbf{M}_1 \times_2 \mathbf{M}_2 = \mathbf{M}_1 \mathbf{S} \mathbf{M}_2^T$$

For example, with $\mathbf{S}, \mathbf{M}_1, \mathbf{M}_2 \in \mathbb{R}^{2\times 2}$, then $\mathbf{A} \in \mathbb{R}^{2\times 2}$, in which:

$$\mathbf{M}_1 = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \quad ; \quad \mathbf{S} = \begin{bmatrix} s_{11} & s_{12} \\ s_{21} & s_{22} \end{bmatrix} \quad ; \quad \mathbf{M}_2 = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} \quad ; \quad \mathbf{M}_2^T = \begin{bmatrix} b_{11} & b_{21} \\ b_{12} & b_{22} \end{bmatrix}$$

then

$$\begin{bmatrix} A(1,1) & A(1,2) \\ A(2,1) & A(2,2) \end{bmatrix} = \begin{bmatrix} \begin{matrix}(a_{11}s_{11} + a_{12}s_{21})b_{11} \\ +(a_{11}s_{12} + a_{12}s_{22})b_{12}\end{matrix} & \begin{matrix}(a_{11}s_{11} + a_{12}s_{21})b_{21} \\ +(a_{11}s_{12} + a_{12}s_{22})b_{22}\end{matrix} \\ \\ \begin{matrix}(a_{21}s_{11} + a_{22}s_{21})b_{11} \\ +(a_{21}s_{12} + a_{22}s_{22})b_{12}\end{matrix} & \begin{matrix}(a_{21}s_{11} + a_{22}s_{21})b_{21} \\ +(a_{21}s_{12} + a_{22}s_{22})b_{22}\end{matrix} \end{bmatrix} =$$

$$\begin{bmatrix} a_{11}s_{11}b_{11} + a_{12}s_{21}b_{11} + a_{11}s_{12}b_{12} + a_{12}s_{22}b_{12} & a_{11}s_{11}b_{21} + a_{12}s_{21}b_{21} + a_{11}s_{12}b_{22} + a_{12}s_{22}b_{22} \\ a_{21}s_{11}b_{11} + a_{22}s_{21}b_{11} + a_{21}s_{12}b_{12} + a_{22}s_{22}b_{12} & a_{21}s_{11}b_{21} + a_{22}s_{21}b_{21} + a_{21}s_{12}b_{22} + a_{22}s_{22}b_{22} \end{bmatrix}$$

then

$$\begin{aligned} \mathtt{vec}(\mathbf{A}) = \begin{bmatrix} a_{11}b_{11} + a_{12}b_{11} + a_{11}b_{12} + a_{12}b_{12} \\ a_{21}b_{11} + a_{22}b_{11} + a_{21}b_{12} + a_{22}b_{12} \\ a_{11}b_{21} + a_{12}b_{21} + a_{11}b_{22} + a_{12}b_{22} \\ a_{21}b_{21} + a_{22}b_{21} + a_{21}b_{22} + a_{22}b_{22} \end{bmatrix} \times \begin{bmatrix} s_{11} \\ s_{21} \\ s_{12} \\ s_{22} \end{bmatrix} &= \left( \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} \otimes \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \right) \times \begin{bmatrix} s_{11} \\ s_{21} \\ s_{12} \\ s_{22} \end{bmatrix} \\ &= (\mathbf{M}_2 \otimes \mathbf{M}_1)\, \mathtt{vec}(\mathbf{S}) \end{aligned}$$

From these equations we can infer these facts:

$$A(i,j) = \sum_{r_1} \sum_{r_2} S(r_1, r_2) M_1(i, r_1) M_2(j, r_2)$$

then

$$\begin{aligned} \mathbf{A} &= \sum_{r_1} \sum_{r_2} S(r_1, r_2) \left( M_1(:, r_1) \circ M_2(:, r_2) \right) \\ &= \sum_{r_1} \sum_{r_2} S(r_1, r_2) \left( M_2(:, r_2) \otimes M_1(:, r_1) \right) \end{aligned}$$

or $\mathtt{vec}(\mathbf{A}) = (\mathbf{M}_2 \otimes \mathbf{M}_1)\, \mathtt{vec}(\mathbf{S})$ as in above example. From this point of view, we can make the generalisation to higher order tensor to get the equivalence between Equation (18) and (19).

For the *mode-k* matrix of a multilinear product, we utilize the property of *mode-k* product that:

$$\begin{aligned} \mathcal{A} &= \mathcal{S} \times_1 \mathbf{M}_1 \times_2 \mathbf{M}_2 \cdots \times_d \mathbf{M}_d \\ &= \mathcal{S} \times_k \mathbf{M}_k \times_1 \mathbf{M}_1 \cdots \times_{k-1} \mathbf{M}_{k-1} \times_{k+1} \mathbf{M}_{k+1} \cdots \times_d \mathbf{M}_d \end{aligned}$$

then

$$\mathcal{A}_{(k)} = \mathbf{M}_k \cdot \mathcal{S}_{(k)} \cdot (\mathbf{M}_d \otimes \mathbf{M}_{d-1} \otimes \ldots \mathbf{M}_{k+1} \otimes \mathbf{M}_{k-1} \cdots \otimes \mathbf{M}_1)^T \tag{21}$$

### 2.2.1 SVD revisited

A Singular Value Decomposition for a matrix is equivalent to a multilinear product of matrices in above example. The correspondence is evident by substituting $\mathbf{U} = \mathbf{M}_1$, $\mathbf{\Sigma} = \mathbf{S}$, $\mathbf{V} = \mathbf{M}_2$, and $\mathbf{S}$ is diagonal. Then, there are 2 ways of expression for SVD from multilinear product perspective: $\mathbf{A} = \mathbf{U\Sigma V}^T$:

$$A(i,j) = \sum_{r_1}\sum_{r_2}\Sigma(r_1,r_2)U(i,r_1)V(j,r_2)$$

and

$$\begin{aligned}\mathbf{A} &= \sum_{r_1}\sum_{r_2}\Sigma(r_1,r_2)\left(U(:,r_1)\circ V(:,r_2)\right)\\ &= \sum_{r}\lambda_r\left(U(:,r)\circ V(:,r)\right)\end{aligned}$$

as $S$ is a diagonal matrix and $\lambda_r = \Sigma(r,r)$.

The first direction describes the actual result from multilinear product and the latter can be seen as the weighted sum of rank-1 matrices. These results are matched with our pre-knowledge about SVD and its derivations.

Then an *order-2* tensor has 2 ways of description in order to reconstruct it from other matrices. In order to generalize these properties for higher order tensor, main ideas are still relied on these observation of SVD representation.

There are two principal algorithms to decompose a tensor: CP(CANDECOMP/PARAFAC) and Tucker. The CP decomposition produces a weighted sum of rank-1 tensors and the Tucker decomposition is the multilinear product for high order tensors. In the following sections, we will discuss these decomposition algorithms in detail along with a new way of decomposing a tensor: Tensor-Train algorithm.

## 2.3 CP decomposition

The CP decomposition is introduced as many names from different publications: CANDECOMP [17], PARAFAC [18], and CP in [2]. Formally, given a tensor: $\mathcal{A} \in \mathbb{R}^{n_1\times n_2\times\cdots\times n_d}$, it finds a tensor approximation $\hat{\mathcal{X}}$ that:

$$\min_{\hat{\mathcal{X}}} \quad ||\mathcal{A} - \hat{\mathcal{X}}||_F$$

where

$$\hat{\mathcal{X}} = \sum_{i=1}^{r}\lambda_i\; a_i^{(1)}\circ a_i^{(2)}\cdots\circ a_i^{(k)}\cdots\circ a_i^{(d)} \tag{22}$$

$\hat{\mathcal{X}}$ is the nearest approximation of the tensor $\mathcal{A}$ in terms of weighted sum of rank-1 tensors. The vector component $a_r^{(k)} \in \mathbb{R}^{n_k\times 1}$ stands for the contribution of dimension $k$th in the rank $r$th. This equivalent formula is derived from the property of the multilinear product:

$$\hat{\mathcal{X}} = \sum_{i=1}^{r}\lambda_i\;\mathbf{M}^{(1)}(:,i)\circ\mathbf{M}^{(2)}(:,i)\cdots\circ\mathbf{M}^{(d)}(:,i)$$

where $\mathbf{M}^{(k)} \in \mathbb{R}^{n_k \times r}$. Intuitively, each mode $k$th is compressed into a matrix $n_k$-by-$r$ instead of $n_k$-by-$(n_1 \times n_2 \cdots \times n_{k-1} \times n_{k+1} \dots n_d)$ from the tensor unfolding property. For example, in an *order-3* tensor, the nearest approximation from CP decomposition is showed in the Figure 14. in which:

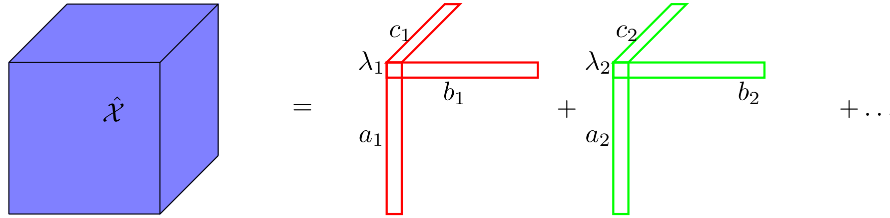



Figure 14: The CP decomposition for an *order-3* tensor

$$\hat{\mathcal{X}} = \sum_{i=1}^{r} \lambda_i \; a_i \circ b_i \circ c_i \tag{23}$$

where $\mathcal{A} \in \mathbb{R}^{n_1 \times n_2 \times n_3}$ and $a_i \in \mathbb{R}^{n_1 \times 1}$, $b_i \in \mathbb{R}^{n_2 \times 1}$, $c_i \in \mathbb{R}^{n_3 \times 1}$

### 2.3.1 Tensor rank

Firstly, we recall the matrix rank that is the number of dimension of the vector space which spans its columns. However, the tensor rank definition is quite different and not a generalisation of the matrix rank.

The rank of a tensor $\mathcal{A}$ is denoted as $\texttt{rank}(\mathcal{A})$ and it is the *smallest* number of required rank-one tensors to exactly reconstruct $\mathcal{A}$ as in Equation (22). For example, a tensor $\mathcal{A} \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_d}$ is called as a *rank-R* tensor if:

$$\mathcal{A} = \sum_{i=1}^{R} \lambda_i \; a_i^{(1)} \circ a_i^{(2)} \cdots \circ a_i^{(k)} \cdots \circ a_i^{(d)} \tag{24}$$

Following the elaborated article [15] and book [11], we will discuss some complication of tensor rank.

- Determining the rank of a tensor is a NP-hard problem.
- There is more than one rank that is available for a tensor. For example from [15], for a $[2 \times 2 \times 2]$ tensor, authors determine that 79% of the space is filled by rank-2 and the rest 21% for rank-3.
- The *maximum rank* is the largest rank can be obtained for a tensor. It is not straight forward as $\max\{n_1, n_2, \dots, n_d\}$.
- The *typical rank* is any rank-$r$ that fit the Equation (24) or any rank that occurs with positive probability.
- A typical rank for an order-3 tensor are already calculated. For example: a tensor $2 \times 2 \times 2$ has a typical rank is $\{2, 3\}$ and this number is $\{3, 4\}$, $\{5, 6\}$ for tensors of size $[3 \times 3 \times 3]$, $[5 \times 3 \times 3]$.

- For a general order-3 tensor: $\mathcal{A} \in \mathbb{R}^{n_1 \times n_2 \times n_3}$, the upper bound for its rank is $\texttt{rank}(\mathcal{A}) \leq \min(n_1 n_2, n_2 n_3, n_1 n_3)$.

### 2.3.2 The algorithm

In this section, we focus on the CP decomposition algorithm based on Alternating Least Square procedure which is proposed in [17], [18]. In order to make the algorithm to be understandable, we firstly find the solution for an order-3 tensor and derive the final algorithm for a general *order-d* tensor.

Given a tensor: $\mathcal{A} \in \mathbb{R}^{n_1 \times n_2 \times n_3}$, the objective is finding an approximation of tensor $\hat{\mathcal{X}}$ that:

$$\min_{\hat{\mathcal{X}}} \quad ||\mathcal{A} - \hat{\mathcal{X}}||_F \tag{25}$$

$$\hat{\mathcal{X}} = \sum_{i=1}^{r} \lambda_i \, \mathbf{M}^{(1)}(:, i) \circ \mathbf{M}^{(2)}(:, i) \circ \mathbf{M}^{(3)}(:, i)$$

where $M^{(k)} \in \mathbb{R}^{n_k \times r}$. This objective function is represented by unfolding its into *mode-k* matrices:

$$||\mathcal{A} - \hat{\mathcal{X}}||_F = ||\mathcal{A}_{(1)} - \hat{\mathcal{X}}_{(1)}||_F = ||\mathcal{A}_{(2)} - \hat{\mathcal{X}}_{(2)}||_F = ||\mathcal{A}_{(3)} - \hat{\mathcal{X}}_{(3)}||_F$$

These objective functions are solved iteratively. Firstly, in the *mode-k* unfolding, we need to find the corresponding CP approximation of this mode in the matrix form. As the CP decomposition objective function (25) is in the multilinear product form where the diagonal tensor $\mathcal{S}$ including $\lambda$, we can rewrite that:

$$\begin{aligned} \hat{\mathcal{X}}_{(1)} &= \sum_{i=1}^{r} \lambda_i \, \mathbf{M}^{(1)}(:, i) \cdot \left(\mathbf{M}^{(3)}(:, i) \otimes \mathbf{M}^{(2)}(:, i)\right)^T \\ &= \mathbf{M}^{(1)} \cdot \text{diag}(\lambda) \cdot \left(\mathbf{M}^{(3)} \odot \ \mathbf{M}^{(2)}\right)^T \end{aligned}$$

where $\text{diag}(\lambda) = \begin{bmatrix} \lambda_1 & \dots \\ \dots & \dots \\ \dots & \lambda_d \end{bmatrix}$ is the diagonal matrix contains all mixing coefficients. The term of $(\mathbf{M}^{(3)}(:, i) \otimes \mathbf{M}^{(2)}(:, i))$ is the Kronecker product of vectors in $\mathbf{M}^{(3)}$ and $\mathbf{M}^{(2)}$ in the same column index. Thus, it is the column-wise Kronecker product or equivalent with the Khatri-Rao product between two matrices of the same number of columns.

Equivalently, we obtain the derivation for $\hat{\mathcal{X}}_{(2)}$ and $\hat{\mathcal{X}}_{(3)}$:

$$\begin{aligned} \hat{\mathcal{X}}_{(2)} &= \mathbf{M}^{(2)} \cdot \text{diag}(\lambda) \cdot \left(\mathbf{M}^{(3)} \odot \ \mathbf{M}^{(1)}\right)^T \\ \hat{\mathcal{X}}_{(3)} &= \mathbf{M}^{(3)} \cdot \text{diag}(\lambda) \cdot \left(\mathbf{M}^{(2)} \odot \ \mathbf{M}^{(1)}\right)^T \end{aligned}$$

There are three equivalent objective functions to the Equation (25) and we need to

find $\{\text{diag}(\lambda), \mathbf{M}^{(1)}, \mathbf{M}^{(2)}, \mathbf{M}^{(3)}\}$ so that these quantities to be minimized:

$$||\mathcal{A}_{(1)} - \mathbf{M}^{(1)} \cdot \text{diag}(\lambda) \cdot \left(\mathbf{M}^{(3)} \odot\ \mathbf{M}^{(2)}\right)^T ||_F$$
$$||\mathcal{A}_{(2)} - \mathbf{M}^{(2)} \cdot \text{diag}(\lambda) \cdot \left(\mathbf{M}^{(3)} \odot\ \mathbf{M}^{(1)}\right)^T ||_F$$
$$||\mathcal{A}_{(3)} - \mathbf{M}^{(3)} \cdot \text{diag}(\lambda) \cdot \left(\mathbf{M}^{(2)} \odot\ \mathbf{M}^{(1)}\right)^T ||_F$$

The Alternating Least Square (ALS) strategy is used by fixing all variables except one and solve the corresponding objective function. For example, when fixing $\mathbf{M}^{(3)}, \mathbf{M}^{(2)}$, we need to find the minimize of this function:

$$||\mathcal{A}_{(1)} - \tilde{\mathbf{M}}^{(1)} \cdot \left(\mathbf{M}^{(3)} \odot\ \mathbf{M}^{(2)}\right)^T ||_F$$

where $\tilde{\mathbf{M}}^{(1)} = \mathbf{M}^{(1)} \cdot \text{diag}(\lambda)$. Then the least squares solution is:

$$\tilde{\mathbf{M}}^{(1)} = \mathcal{A}_{(1)} \left[\left(\mathbf{M}^{(3)} \odot\ \mathbf{M}^{(2)}\right)^T\right]^{\dagger}$$

As the above equation need the inversion of a big matrix $\in \mathbb{R}^{n_3 n_2 \times r}$, nevertheless we can calculate this quantity without doing this inversion thanks to the properties of Khatri-Rao product:

$$\tilde{\mathbf{M}}^{(1)} = \mathcal{A}_{(1)} \left(\mathbf{M}^{(3)} \odot\ \mathbf{M}^{(2)}\right) \left((\mathbf{M}^{(3)})^T \mathbf{M}^{(3)} * (\mathbf{M}^{(2)})^T \mathbf{M}^{(2)}\right)^{\dagger} \tag{26}$$

After getting the solution for $\tilde{\mathbf{M}}^{(1)}$, we can get the solution for $\lambda_j$ and $\mathbf{M}^{(1)}$ after this normalization:

$$\begin{aligned} \lambda_j &= ||\tilde{\mathbf{M}}^{(1)}(:, j)||_2 \\ \mathbf{M}^{(1)}(:, j) &= ||\tilde{\mathbf{M}}^{(1)}(:, j)||/\lambda_j \end{aligned} \tag{27}$$

Finally, we can make a generalisation algorithm for a *d-order* tensor based on Equations (26) and (27).

---

**Algorithm 1:** The CP decomposition algorithm based on Alternating Least Square procedure

---

**Data:** A *d-order* tensor $\mathcal{A} \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_d}$
**Input:** The intended rank $r$
**Output:** $\text{diag}(\lambda), \mathbf{M}^{(1)}, \mathbf{M}^{(2)}, \ldots, \mathbf{M}^{(d)}$
Initialize $\mathbf{M}^{(k)} \in \mathbb{R}^{n_k \times r}$ randomly ;
**while** `!stop-condition` **do**
  **for** $k = [1, 2, \ldots, d]$ **do**
    `/* least squares solution` (26) `*/`
    $\mathbf{V} = \mathbf{M}^{(d)T}\mathbf{M}^{(d)} * \ldots \mathbf{M}^{(k+1)T}\mathbf{M}^{(k+1)} * \mathbf{M}^{(d)T}\mathbf{M}^{(d)} \cdots * \mathbf{M}^{(1)T}\mathbf{M}^{(1)}$;
    $\tilde{\mathbf{M}}^{(k)} = \mathcal{A}_{(k)} \left(\mathbf{M}^{(d)} \odot \ldots \mathbf{M}^{(k+1)} \odot \mathbf{M}^{(k-1)} \cdots \odot \mathbf{M}^{(1)}\right) V^{\dagger}$;
    `/* update the solution as Equation` (27) `*/`
    **for** $j = [1, 2, \ldots, r]$ **do**
      Update $\lambda_j$ ;
      Update $\mathbf{M}^{(k)}(:, j)$ ;

---

## 2.4 Tucker decomposition

The Tucker decomposition is invented by Tucker [19], [3] and also has many names, the most related under the name Higher Order SVD (HOSVD) [20]. The multilinear product description is still the central matter. While the CP decomposition is based on the diagonal tensor $\mathcal{S}$, the Tucker decomposition uses a full one. Formally, given a tensor $\mathcal{A} \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_d}$, a tensor $\mathcal{S} \in \mathbb{R}^{r_1 \times r_2 \times \cdots \times r_d}$, and $d$ matrices in which $\mathbf{M}_k \in \mathbb{R}^{n_d \times r_d}$. The Tucker decomposition objective function is finding a nearest tensor $\hat{\mathcal{X}}$ which is in the form of multilinear product:

$$\min_{\hat{\mathcal{X}}} \quad ||\mathcal{A} - \hat{\mathcal{X}}||_F \tag{28}$$

$$\hat{\mathcal{X}} = \mathcal{S} \times_1 \mathbf{M}_1 \times_2 \mathbf{M}_2 \cdots \times_d \mathbf{M}_d$$

For example, for an *order-3* tensor $\mathcal{A} \in \mathbb{R}^{n_1 \times n_2 \times n_3}$, the decomposition components are illustrated in Figure 15.

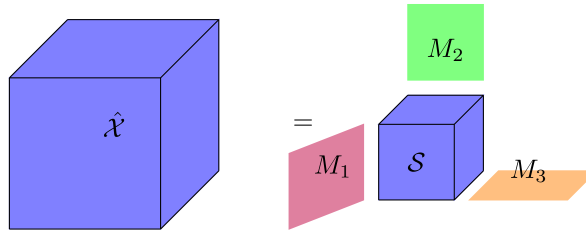



Figure 15: The Tucker decomposition components for an *order-3* tensor

Due to the tensor $\mathcal{S}$ not being diagonal, we need to take care about the rank of each mode: $\{r_1, r_2, \ldots, r_d\}$. We need to define the specific value for $r_i$ or find an efficient value for it. The HOSVD is based on the SVD definition and ranks for each *mode-k* matrix while the Tucker decomposition can achieve a higher compression by user-defined ranks.

### 2.4.1 The *n-rank* and multilinear rank

In order to distinguish the HOSVD and Tucker decomposition, we need to define notation for ranks of tensor modes. There are two common definitions:

- In [15], the *n-rank* for a tensor or $\texttt{rank}_n(\mathcal{A})$ is the rank of the matrix based on *mode-n* unfolding: $\mathcal{A}_{(n)}$. It is the column rank of matrix $\mathcal{A}_{(n)}$.
- In [11], the multilinear rank or $\texttt{rank}_*(\mathcal{A})$ contains all *n-rank* of all modes.

$$\texttt{rank}_*(\mathcal{A}) = \{\texttt{rank}_1(\mathcal{A}), \texttt{rank}_2(\mathcal{A}), \ldots, \texttt{rank}_d(\mathcal{A})\}$$

### 2.4.2 The Higher Order SVD (HOSVD)

We can find an exact solution for Equation (28) when $r_k = \texttt{rank}_k(\mathcal{A})$ for all modes. The solution is given by using the SVD decomposition and providing specific ranks

for each mode. Given a tensor $\mathcal{A}$, the SVD decomposition for each mode is:

$$\mathcal{A}_{(k)} = \mathbf{U}_k \mathbf{\Sigma}_k \mathbf{V}_k^T$$

Recall that $\mathcal{A}_{(k)}$ is the matrix of $n_k$-by-$(n_1 n_2 \dots n_{k-1} n_{k+1} \dots n_d)$, thus the rank $\mathtt{rank}_k(\mathcal{A}) \leq \min(n_k, n_1 n_2 \dots n_{k-1} n_{k+1} \dots n_d)$ and $\mathbf{U}_k$ expands the column space of $\mathcal{A}_{(k)}$. According to [20], its HOSVD is given by:

$$\begin{aligned} \mathcal{A} &= \mathcal{S} \times_1 \mathbf{U}_1 \times_2 \mathbf{U}_2 \cdots \times_d \mathbf{U}_d \\ \mathcal{S} &= \mathcal{A} \times_1 \mathbf{U}_1^T \times_2 \mathbf{U}_2^T \cdots \times_d \mathbf{U}_d^T \end{aligned} \quad (29)$$

where $\mathbf{U}_k^T = \mathbf{U}_k^\dagger$ as the matrix $\mathbf{U}_k$ is columns orthogonal. Noted that the tensor $\mathcal{S}$ is called *core* tensor.

**Algorithm 2:** The HOSVD decomposition algorithm

**Data:** A *d-order* tensor $\mathcal{A} \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_d}$
**Input:**
**Output:** $\mathcal{S}, \mathbf{M}^{(1)}, \mathbf{M}^{(2)}, \dots, \mathbf{M}^{(d)}$
Initialize $\mathbf{M}^{(k)} \in \mathbb{R}^{n_k \times r}$ randomly ;
**for** $k = [1, 2, \dots, d]$ **do**
  `/* SVD decomposition for each mode */`
  $\mathcal{A}_{(k)} = \mathbf{U}_k \mathbf{\Sigma}_k \mathbf{V}_k^T$;
  $\mathbf{M}^{(k)} = \mathbf{U}_k$;
`/* update the core tensor */`
$\mathcal{S} = \mathcal{A} \times_1 \mathbf{U}_1^T \times_2 \mathbf{U}_2^T \cdots \times_d \mathbf{U}_d^T$ ;

Intuitively, the algorithm independently decomposes the tensor in each mode to capture the mode-variance. Then the *core* tensor stores the interaction information among separated modes to reconstruct the tensor.

### 2.4.3 Tucker decomposition algorithm

In the case of $r_k < \mathtt{rank}_k(\mathcal{A})$ in at least one mode or if one is aiming to achieve lower $r_k$, the HOSVD algorithm can be applied by taking less columns for the left singular matrix $\mathbf{U}_k$. However, this *truncated* HOSVD does not give the optimal solution for the Tucker approximation objective function (28).

In order to derive a solution, firstly we analyse the Tucker optimization problem in an *order-3* tensor and make a generalisation latter. From vectorization properties of the multilinear product, the equivalent objective function of (28) is

$$||\mathcal{A} - \hat{\mathcal{X}}||_F = ||\mathtt{vec}(\mathcal{A}) - \left(\mathbf{M}^{(3)} \otimes \mathbf{M}^{(2)} \otimes \mathbf{M}^{(1)}\right) \mathtt{vec}(\mathcal{S})||_2$$

As $\left(\mathbf{M}^{(3)} \otimes \mathbf{M}^{(2)} \otimes \mathbf{M}^{(1)}\right)$ is column orthogonal so the solution for core tensor is:

$$\mathtt{vec}(\mathcal{S}) = \left(\mathbf{M}^{(3)} \otimes \mathbf{M}^{(2)} \otimes \mathbf{M}^{(1)}\right)^T \mathtt{vec}(\mathcal{A})$$

We get the new objective function by putting it back to the objective function:

$$\min_M \quad ||\mathtt{vec}(\mathcal{A}) - \mathbf{M}\mathbf{M}^T \mathtt{vec}(\mathcal{A})||_2$$

where $\mathbf{M} = \left(\mathbf{M}^{(3)} \otimes \mathbf{M}^{(2)} \otimes \mathbf{M}^{(1)}\right)$. Moreover, we can shorten it more by:

$$\begin{aligned}
&||\texttt{vec}(\mathcal{A}) - \mathbf{M}\mathbf{M}^T\texttt{vec}(\mathcal{A})||_2 \\
&\quad = ||\texttt{vec}(\mathcal{A})||_2 - 2\langle \texttt{vec}(\mathcal{A}), \mathbf{M}\mathbf{M}^T\texttt{vec}(\mathcal{A})\rangle + ||\mathbf{M}\mathbf{M}^T\texttt{vec}(\mathcal{A})||_2 \\
&\quad = ||\texttt{vec}(\mathcal{A})||_2 - 2\texttt{vec}(\mathcal{A})^T\mathbf{M}\mathbf{M}^T\texttt{vec}(\mathcal{A}) + ||\mathbf{M}^T\texttt{vec}(\mathcal{A})||_2 \\
&\quad = ||\texttt{vec}(\mathcal{A})||_2 - ||\mathbf{M}^T\texttt{vec}(\mathcal{A})||_2
\end{aligned}$$

where the last terms are equivalent as $\mathbf{M}^T\mathbf{M} = \mathbf{I}$ as $\mathbf{M}$ is column orthogonal. As the term $||\texttt{vec}(\mathcal{A})||_2$ is a constant then the final objective function is to maximize that quantity:

$$|| \left(\mathbf{M}^{(3)T} \otimes \mathbf{M}^{(2)T} \otimes \mathbf{M}^{(1)T}\right) \texttt{vec}(\mathcal{A})||_2 = \begin{cases} ||\mathbf{M}^{(1)T} \cdot \mathcal{A}_{(1)} \cdot (\mathbf{M}^{(3)} \otimes \mathbf{M}^{(2)})||_F \\ ||\mathbf{M}^{(2)T} \cdot \mathcal{A}_{(2)} \cdot (\mathbf{M}^{(3)} \otimes \mathbf{M}^{(1)})||_F \\ ||\mathbf{M}^{(3)T} \cdot \mathcal{A}_{(3)} \cdot (\mathbf{M}^{(2)} \otimes \mathbf{M}^{(1)})||_F \end{cases}$$

At this point, we can apply the Alternating Least Squares to solve objective functions in modes. In each mode, it leads to another optimization problem that given a matrix $\mathbf{A}$ find the best projection $\mathbf{Q}$:

$$\min_{\mathbf{Q}} \quad ||\mathbf{Q}^T\mathbf{A}||_F \quad \text{s.t} \quad \mathbf{Q}^T\mathbf{Q} = \mathbf{I}$$

For example, in solving $\min_{\mathbf{M}^{(1)}} ||\mathbf{M}^{(1)T} \cdot \mathcal{A}_{(1)} \cdot (\mathbf{M}^{(3)} \otimes \mathbf{M}^{(2)})||_F$, the $\mathbf{A}$ matrix is corresponding to $\left(\mathcal{A}_{(1)} \cdot (\mathbf{M}^{(3)} \otimes \mathbf{M}^{(2)})\right)$ and $\mathbf{Q}$ is the matrix $\mathbf{M}^{(1)}$.

Fortunately, this problem solution comes from the decomposition of $\mathbf{A} = \mathbf{U}\mathbf{S}\mathbf{V}^T$ as

$$||\mathbf{Q}^T\mathbf{A}||_F = ||\mathbf{Q}^T\mathbf{U}\mathbf{S}\mathbf{V}^T||_F = ||\mathbf{Q}^T\mathbf{U}\mathbf{S}||_F = \sum_{i=1}^{r} \lambda_i ||\mathbf{Q}^T U(:, i)||_2$$

Then the solution for this nonnegative maximization problem is the top $\hat{r}$ eigen-vectors of the left singular matrix $\mathbf{U}$. Then the final algorithm follows:

**Algorithm 3:** The ALS Tucker decomposition algorithm

**Data:** A *d-order* tensor $\mathcal{A} \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_d}$
**Input:** Input ranks $\{r_1, r_2, \ldots, r_d\}$
**Output:** $\mathcal{S}$, $\mathbf{M}^{(1)}, \mathbf{M}^{(2)}, \ldots, \mathbf{M}^{(d)}$
Initialize $\mathbf{M}^{(k)} \in \mathbb{R}^{n_k \times r}$ randomly or from *truncated* HOSVD;
**for** $k = [1, 2, \ldots, d]$ **do**
  $\mathcal{V}^k = \mathcal{A}_{(k)} \cdot (\mathbf{M}^{(d)} \otimes \ldots \mathbf{M}^{(k+1)} \otimes \mathbf{M}^{(k-1)} \cdots \otimes \mathbf{M}^{(1)})$ ;
  `/* SVD decomposition */`
  $\mathbf{V}^k = \mathbf{U}_k \mathbf{\Sigma}_k \mathbf{V}_k^T$;
  $\mathbf{M}^{(k)} = U_k(:, 1 : r_k)$;
`/* update the core tensor */`
$\mathcal{S} = \mathcal{A} \times_1 \mathbf{U}_1^T \times_2 \mathbf{U}_2^T \cdots \times_d \mathbf{U}_d^T$ ;

## 2.5 Tensor train decomposition

### 2.5.1 Analysis and definition

We already discuss about two fundamental tensor decomposition algorithms: CP and Tucker decomposition. However, these algorithms are not optimal in terms of efficiency and stability. In the ideal case, if we have a *d-order* tensor and each dimension contains $n$ elements, then we need $n^d$ numbers to store it. In the CP decomposition, the approximation from $R$ rank-1 tensors needs $Rdn$ parameters and it is a quite small number in comparison with $n^d$ parameters. However, it is a NP-hard problem in determining $R$ and the CP algorithm is an ill-posedness problem [21]. On the other hand, the Tucker decomposition is optimal but it suffers a high complexity of $\mathcal{O}(R^d + dnR)$ parameters and thus remaining in the bottle neck of the dimensionality curse.

Recently, the Tensor Train decomposition [4] has emerged as an efficient decomposition algorithm that is stable and low complexity. It is a special case of the *tensor network* [22] where a higher order tensor is approximated by many low-order tensors and contraction operators (reshape and multilinear products). This *network* includes nodes as low-order tensors and edges operates the contraction. Formally, given a tensor $\mathcal{A} \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_d}$ where each dimension $k$th the indexing is $i_k \in [1, 2, \ldots, n_k]$, the Tensor Train decomposition of $\mathcal{A}$ is the multiplication of low-rank matrices $\mathcal{G}_i$ in specific indicies for a specific element $\mathcal{A}(i_1, i_2, \ldots, i_d)$:

$$\mathcal{A}(i_1, i_2, \ldots, i_d) = \mathcal{G}_1[i_1]\mathcal{G}_2[i_2] \ldots \mathcal{G}_d[i_d] \tag{30}$$

where $\mathcal{G}_k$ is a 3-order tensor of $r_{k-1} \times n_k \times r_k$ and $\mathcal{G}_k[i_k]$ is a matrix of $r_{k-1} \times r_k$ with $r_k$ is the low-rank. In addition, the boundary conditions that $r_0 = r_d = 1$ in order to get $\mathcal{A}(i_1, i_2, \ldots, i_d)$ is a scalar. On the other hand, the TT can be seen as a tensor factorization method, when a tensor can be described by the multiplication of factors, each factor is a low-order tensor according to one index in a dimension.

**TT decomposition for a matrix** Firstly, a matrix is also an order-2 tensor, so the TT decomposition is equivalent to the matrix factorization. In Figure 16, a 4-by-4 matrix $\mathbf{A}$ is decomposed into 2 smaller matrices $\mathcal{G}_1, \mathcal{G}_2$ (order-2 tensors). In each $\mathcal{G}_k$, there are 4 vectors according to 4 indices that make the low-rank approximation for each specific dimension index of $\mathbf{A}$. To reconstruct an element $A(3, 2)$, the 3rd component of $\mathcal{G}_1$ and the 2nd component of $\mathcal{G}_2$ are taken to do the multiplication. Noted that, in this case of an order-2 tensor then both $\mathcal{G}_k$ are satisfy the boundary conditions that $r_0 = r_3 = 1$ then $\mathcal{G}_1$ is $r_0 \times n_1 \times r_1$ or $1 \times n_1 \times r_1$ and the same for $\mathcal{G}_2$ of $r_1 \times n_2 \times 1$

**TT decomposition for an order-3 tensor** Similarly, for an order-3 tensor we need to add an additional tensor $\mathcal{G}_2$ in decomposition components which stands for the second dimension. Figure 17 shows the TT decomposition for a tensor $\mathcal{A} \in \mathbb{R}^{4 \times 2 \times 3}$:

- $\mathcal{G}_1 \in \mathbb{R}^{1 \times 4 \times r_1}$

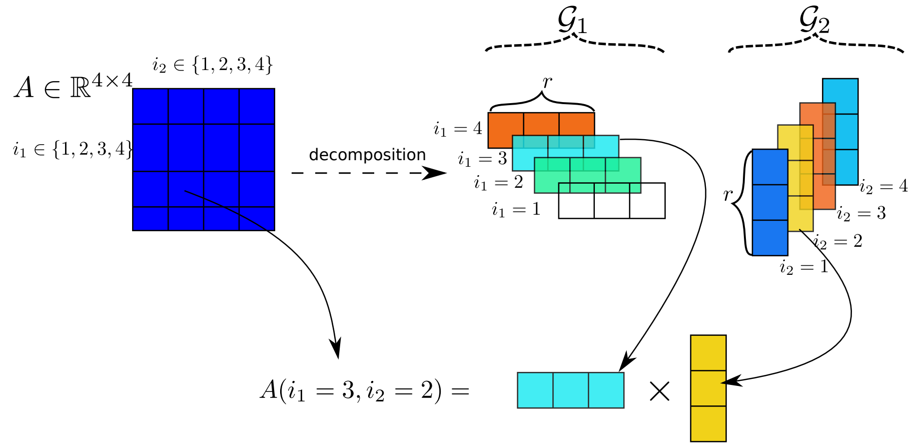



Figure 16: Tensor train decomposition and reconstruction for a specific element in a matrix

- $\mathcal{G}_2 \in \mathbb{R}^{r_1\times2\times r_2}$
- $\mathcal{G}_3 \in \mathbb{R}^{r_2\times3\times1}$

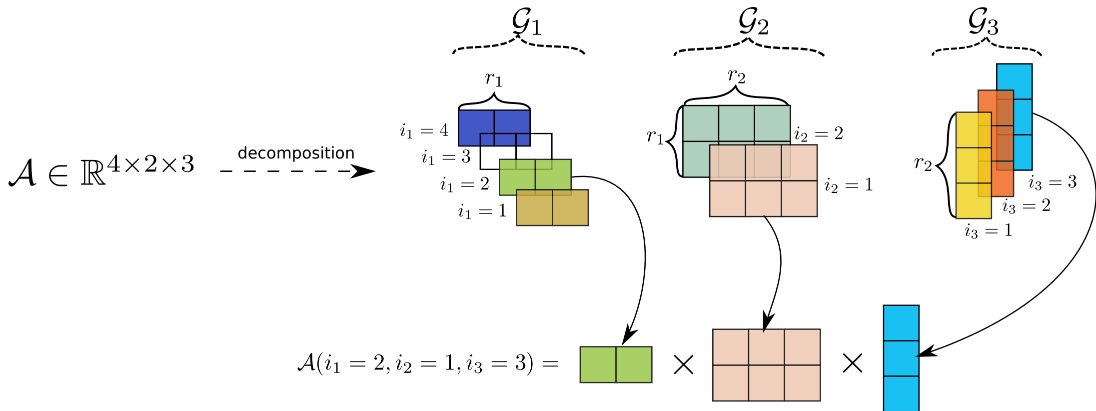



Figure 17: The Tensor train decomposition and its reconstruction for a specific element in an order-3 tensor (cubic)

If $r_1 = 4$ and $r_2 = 2$, then we have an exact TT decomposition for $\mathcal{A}$ and thus it is an approximation when $r_1 < 4$ and $r_2 < 2$. In this case, each $\mathcal{G}_2[i_2]$ is a matrix of $r_1 \times r_2$ for a specific index $i_2$ in the second dimension. Intuitively, this matrix captures interactions of all indices in the first and third dimension that go through a specific index in 2nd dimension. Finally, in order to reconstruct an element from a tensor, we take the multiplication of corresponding low-order tensor elements based on its indices.

**TT decomposition for a order-4 tensor** For a tensor that its order is higher than 3, the visualization comes to the *train* style. For an order-4 tensor $\mathcal{A} \in \mathbb{R}^{n_1\times n_2\times n_3\times n_4}$,

the TT decomposition train in Figure 18 is:

- $\mathcal{G}_1 \in \mathbb{R}^{1 \times n_1 \times r_1}$
- $\mathcal{G}_2 \in \mathbb{R}^{r_1 \times n_2 \times r_2}$
- $\mathcal{G}_3 \in \mathbb{R}^{r_2 \times n_3 \times r_3}$
- $\mathcal{G}_4 \in \mathbb{R}^{r_3 \times n_4 \times 1}$

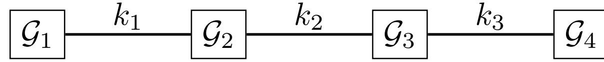



Figure 18: Tensor train decomposition components for an order-4 tensor

For each element of index $(i_1, i_2, i_3, i_4)$, the TT reconstruction is:

$$\begin{aligned}\mathcal{A}(i_1, i_2, i_3, i_4) &= \sum_{k_1=1}^{r_1} \sum_{k_2=1}^{r_2} \sum_{k_3=1}^{r_3} \mathcal{G}_1(i_1, k_1)\mathcal{G}_2(k_1, i_2, k_2)\mathcal{G}_3(k_2, i_3, k_3)\mathcal{G}_4(k_3, i_4) \\ &= \underbrace{\mathcal{G}_1[i_1]}_{1\times r_1} \underbrace{\mathcal{G}_2[i_2]}_{r_1\times r_2} \underbrace{\mathcal{G}_3[i_3]}_{r_2\times r_3} \underbrace{\mathcal{G}_4[i_4]}_{r_3\times 1}\end{aligned}$$

### 2.5.2 Algorithm

At this point we can confirm that the storage complexity of the Tensor Train decomposition is lower than the Tucker decomposition algorithm. The TT complexity is $\mathcal{O}(dR^2n)$ for storing a tensor of $n^d$ elements. By using the SVD decomposition as a workhorse, the TT inherits its reliability and stability, which is the advantage over

the CP decomposition.

**Algorithm 4:** The Tensor train decomposition algorithm

**Data:** A *d-order* tensor $\mathcal{A} \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_d}$
**Input:**
**Output:** $\mathcal{G}_1, \mathcal{G}_2, \ldots \mathcal{G}_d$

```
/* mode-1 unfolding of the tensor */
```
$\mathbf{M}_1 = \mathcal{A}_{(1)}$ ;
```
/* SVD decomposition */
```
$\mathbf{U}_1 \underbrace{\mathbf{\Sigma}_1 \mathbf{V}_1^T}_{\mathbf{S}_1} = \texttt{svd}(\mathbf{M}_1)$ where $\mathbf{\Sigma}_1 \in \mathbb{R}^{r_1 \times r_1}$ and $r_1$ is the rank of $\mathbf{M}_1$ ;
```
/* Get G_1 */
```
$\mathcal{G}_1 = \mathbf{U}_1$ ;
**for** $k = [2, 3 \ldots, d-1]$ **do**
- $\mathbf{M}_k = \texttt{reshape}(\mathbf{S}_{k-1}, [r_{k-1}n_k, n_{k+1}n_{k+2} \cdots n_d])$;
- `/* SVD decomposition */`
- $\mathbf{M}_k = \mathbf{U}_k \underbrace{\mathbf{\Sigma}_k \mathbf{V}_k^T}_{\mathbf{S}_k}$ where $\mathbf{\Sigma}_1 \in \mathbb{R}^{r_k \times r_k}$ and $r_k = \texttt{rank}(\mathbf{M}_k)$ ;
- `/* Get G_k */`
- $\mathcal{G}_k = \texttt{reshape}(\mathbf{U}_k, [r_{k-1}, n_k, r_k])$ ;

```
/* Get G_d */
```
$\mathcal{G}_d = \mathbf{S}_{d-1}$ ;

For example, this algorithm steps in Figure 18 are:

1. For the (1)st dimension:
   - $\mathbf{M}_1 = \texttt{reshape}(\mathcal{A}, [n_1, n_2n_3n_4])$
   - Do SVD: $\mathbf{M}_1 = \mathbf{U}_1\mathbf{S}_1$ and get $\mathcal{G}_1 = \mathbf{U}_1$
2. For the (2)nd dimension:
   - $\mathbf{M}_2 = \texttt{reshape}(\mathcal{S}_1, [r_1n_2, n_3n_4])$
   - Do SVD: $\mathbf{M}_2 = \mathbf{U}_2\mathbf{S}_2$ and get $\mathcal{G}_2 = \texttt{reshape}(\mathbf{U}_2, [r_1, n_2, r_2])$
3. For the (3)rd dimension:
   - $\mathbf{M}_3 = \texttt{reshape}(\mathcal{S}_2, [r_2n_3, n_4])$
   - Do SVD: $\mathbf{M}_3 = \mathbf{U}_3\mathbf{S}_3$ and get $\mathcal{G}_3 = \texttt{reshape}(\mathbf{U}_3, [r_2, n_3, r_3])$
4. For the (4)th dimension:
   - Get $\mathcal{G}_4 = \mathbf{S}_3$

The Tensor Train algorithm iteratively decomposes each dimension in the linear order. In each step, the unfolding *mode-k* of the tensor is inputted to the SVD decomposition and the left singular matrix is always taken to the result. The right singular and diagonal matrix are multiplied and reshaped to create an input matrix for the next SVD step.

# 3 Multiple Kernel Learning

Multiple kernel learning (MKL) refers to a family of algorithms that learn an optimal combination of kernels. This section reviews the definition kernel learning and multiple kernel learning algorithms in the supervised learning framework. In term of storing kernels, the stacking of them can be consider as a tensor; for example: if there are $k$ kernels of size $n$-by-$n$, we get an *order-3* tensor $n$-by-$n$-by-$k$. From the perspective of tensor learning, decomposition a tensor into smaller parts is always beneficial. This section also introduces preliminary algorithms where the tensor learning can apply into the Multiple kernel learning framework.

## 3.1 Kernel learning

In supervised learning, the Support Vector Machine [23], [7] is a discriminative algorithm which is proposed to solve binary classification problems. Given a set of $N$ samples which are generated as independent and identically distributed random instances, we get a training set $\{x_i, y_i\}_{i=1}^{N}$. Each instance $i$th includes a $D$ dimensional feature vector $x_i \in \mathbb{R}^D$ and class label $y_i \in \{+1, -1\}$. The SVM task is to find an optimal linear discriminant hyperplane that maximizes the margin or getting the largest separation between two classes.

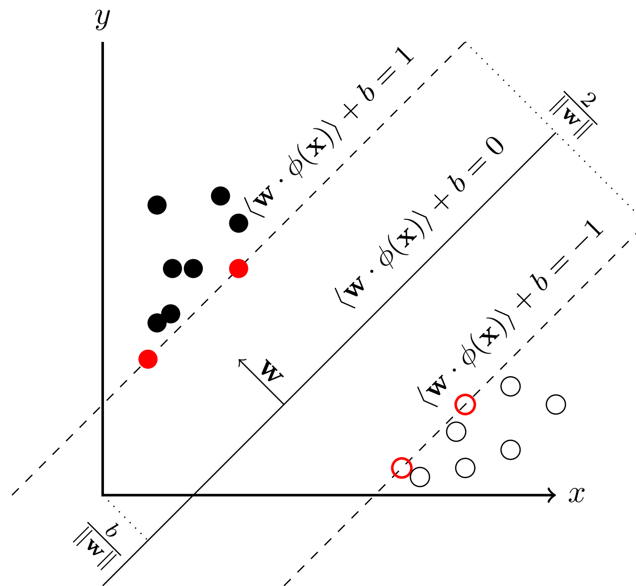



Figure 19: Components of the Support Vector Machine algorithm in two dimensional samples.

Typically, the SVM manipulates on the feature space which is induced through a mapping function $\theta : \mathbb{R}^D \rightarrow \mathbb{R}^S$. Then the linear discriminant function for an instance is:

$$f(x_i) = \langle w, \theta(x_i) \rangle + b$$

where $w$ is weight coefficients or stands for the slope of $f$ in feature space, $b$ is the bias term, and $\langle , \rangle$ is the inner product between two vectors.

Figure 19 illustrates components of SVM algorithms in a two dimensional data. There are two classes of samples: the black samples and the white ones. The *margin* is the distance between two lines where $f(x_i) = \pm 1$ and this distance is $\frac{2}{|||w||}$. The *margin* maximizing is corresponding to minimize this $\frac{|||w||}{2}$ quantity.

The criteria for binary classification task is $f(x_i) \geq 1$ if the sample $i$ belongs to the positive class $+1$ and otherwise. On the other hand, it is equivalent with $f(x_i)y_i \geq 1$ where $x_i$ and $y_i$ belongs to a sample. Then the optimal classifier can be obtained through solving this quadratic optimization problem:

$$\min_{w,b} \quad \frac{1}{2}||w||_2^2 + C\sum_{i=1}^{N}\epsilon_i \quad \text{w.r.t} \quad w \in \mathbb{R}^S, \ b \in \mathbb{R} \tag{31}$$

$$\text{s.t} \quad y_i(\langle w, \theta(x_i)\rangle + b) \geq 1 - \epsilon_i \quad , \epsilon_i \geq 0 \quad , \forall i \in \{1, 2, \ldots, N\}$$

where $\epsilon_i$ are slack variables for each sample and $C$ is the nonnegative trade-off parameter between the model generalisation and the classification correctness.

In the dual space, this objective function can be derived in the form the Lagrangian dual function:

$$\max_{\alpha_i} \quad \sum_{i=1}^{N}\alpha_i - \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i\alpha_j y_i y_j \underbrace{\langle\theta(x_i), \theta(x_j)\rangle}_{\kappa(x_i,x_j)} \quad \text{w.r.t} \quad \alpha \in \mathbb{R}_+^N$$

$$\text{s.t} \quad \sum_{i=1}^{N}\alpha_i y_i = 0 \quad \text{and} \quad 0 \leq \alpha_i \leq C \quad \forall i \in \{1, 2, \ldots, N\}$$

where $\alpha$ is the dual vector of size $N$ according to the number of constraints. We define $\kappa \in \mathbb{R}^D \times \mathbb{R}^D \rightarrow \mathbb{R}$ is the kernel function that specifies the dot product between pairs of samples. After solving the optimal $\alpha$ vector in dual space, we get the primal solution for $w = \sum_{i=1}^{N}\alpha_i y_i \theta(x_i)$, and the linear discriminant function is equivalent with:

$$f(x) = \sum_{i=1}^{N}\alpha_i y_i \kappa(x_i, x) + b$$

The above derivation shows that weight vector can be described by the linear combination of training points. Whenever this property is satisfied, the kernel can be exploited and leads to many other kernel-based algorithms, for example, Kernel Ridge Regression, Kernel PCA, Kernel CCA [10], [24]. These kernel-based algorithms work by the data embedding into the Hilbert space through the kernel function $\kappa$, so the linear relations in this embedded space are explored. This way offers several advantages, for instance, the implicitly calculation inner products between points is faster and easier than explicit projecting and do the inner product later. Moreover, changing the kernel function $\kappa$ enables these algorithms perform a non-linear classification/ regression through implicitly mapping input features into ae higher-dimension space. Beyond that, this way allows these algorithms not only work

on numerical vectors, but also on strings, graphs, images by incorporates domain knowledge, whenever a kernel follows the Mercer's condition [23]. In the other hand, kernel-based algorithms are well-founded under the statistical learning theory and solved as a convex optimization or eigenproblems [10]. These properties make kernel methods is a fundamental algorithm family in pattern analysis and its applications are diverse in many fields from computer vision, natural language processing to bioinformatics.

Finally, learning about kernels is an important topic that can enable us to reduce computation complexity, improve performance, and find new applications of kernel-based algorithms. In following chapters, we will consider the *kernel matrix* $\mathbf{K}_{ij} = \kappa(x_i, x_j)$ or called as *Gram matrix*. This kind of matrix contains all pairwise distance between data points in the embedding space, so it is symmetric and positive semidefinite.

### 3.1.1 Pairwise kernel learning

In [25], a pairwise kernel learning method is firstly introduced to predict the protein-protein interaction. Generally, it is also considered as learning the relation between pairs of objects; for example: link prediction between 2 users in the social network [26] or connectivity prediction on edges between two nodes in a graph. Formally, given two objects dataset $D_1, D_2$, it numeric features is in vectors $x_{D_1} \in \mathbb{R}^{d_1}$ and $x_{D_2} \in \mathbb{R}^{d_2}$, where $d_1, d_2$ are the number of features for each object. Then the training data is in the form of $(x_{D_1}^{(i)}, x_{D_2}^{(j)}, y_{ij})$ in which $y_{ij}$ is the target label according to the $i^{\text{th}}$ and $j^{\text{th}}$ sample in each object dataset.

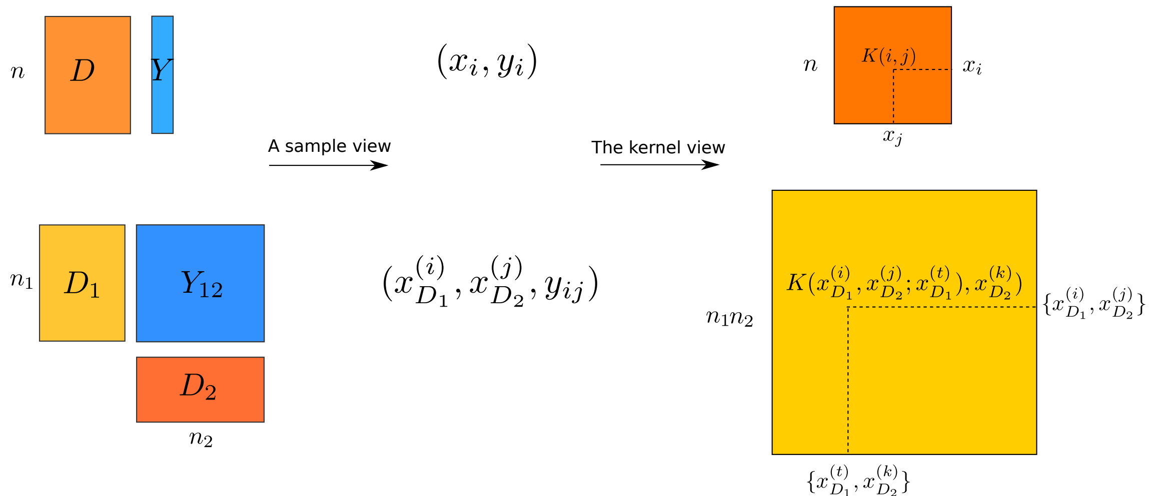



Figure 20: The comparison between kernel learning and pair-wise kernel learning

Given $n_1, n_2$ samples in dataset $D_1, D_2$ respectively, the main task is to build the kernel matrix of size $n_1n_2$-by-$n_1n_2$ that describe that *similarity* among pairs of objects. In using Support Vector Machine algorithm, the kernel matrix is the

crucial point in getting a successful learning process. While building a kernel for two objects are well developed over years, the pairwise kernel is more limited in the way to create and store it. As in Figure 20, a point in the pairwise kernel is the numeric measurement about *similarity* between two pairs $\{x_{D_1}^{(i)}, x_{D_2}^{(j)}\}$ and $\{x_{D_1}^{(t)}, x_{D_2}^{(k)}\}$, thus it requires a quadratic scale to store it. In [25] [27], authors proposed a combination rule to calculate the pairwise kernel from each object kernel $K_{D_1}$ and $K_{D_2}$:

$$K\left(\{x_{D_1}^{(i)}, x_{D_2}^{(j)}\}, \{x_{D_1}^{(t)}, x_{D_2}^{(k)}\}\right) = K_{D_1}(x_{D_1}^{(i)}, x_{D_1}^{(t)})K_{D_2}(x_{D_2}^{(j)}, x_{D_2}^{(k)})$$

This is equivalent to the Kronecker product between two object kernels: $K = K_{D_1} \otimes K_{D_2}$. Then, the Kronecker product kernel is considered as a general-purpose way to create a kernel for the pairwise kernel learning.

## 3.2 Multiple Kernel Learning

In designing machine learning applications, a critical step is finding a good feature representation in order to get higher algorithm performance. In kernel-based algorithms, it is according to learn an optimal way to combine kernels when many kernels are available. There are many kind of kernels which successfully used in literatures and practices. For example: linear kernel, polynomial kernel, and Gaussian kernel are typically used for numerical data. In other applications, there are also many particular kernels such as in bioinformatics [28]. Moreover, a kernel again includes hyper-parameters such as $\sigma$ in the Gaussian kernel, changing its value also produces new kernels.

A typical solution is selecting the set of good kernels by doing cross-validation or relied on the prior knowledge of users. The past decade has seen the emerging of an automatic framework in learning from kernels, which is called Multiple Kernel Learning (MKL) [6] which finds the best combination of kernels rather than picking the best one. Intuitively, this approach may eliminate the biases if only using one kernel and incorporate data information from multiple sources through different kernels.

In a Multiple Kernel Learning problem, where $P$ kernels are exist, the task is seeking an optimal combination function $f_\eta : \mathbb{R}^P \to \mathbb{R}$ that:

$$\mathbf{K}_\eta(x_i, x_j) = f_\eta(\{\mathbf{K}(x_i^m, x_j^m)_{m=1}^P\})$$

where the combined kernel $\mathbf{K}_\eta$ for a specific pair of samples is expressed as a transformation $f_\eta$ for its pairwise distance through $P$ kernels.

### 3.2.1 MKL algorithms

Several algorithms has been proposed to find a good combination function $f_\eta$, a very comprehensive study in [6]. In this study, instead of summarize algorithmic steps to solve a MKL problem, we focus to highlight the formulation aspect and the evolution of algorithms over time.

**MKL without additional optimization steps** Some preliminary works use the simple fixed rules to combine kernels. In order to keep the $\mathbf{K}_\eta$ is positive semidefinite, they take the *summation* and *multiplication* of kernels:

$$\mathbf{K}_\eta(x_i, x_j) = \sum_{m=1}^{P} \mathbf{K}(x_i^m, x_j^m)$$
$$\mathbf{K}_\eta(x_i, x_j) = \prod_{m=1}^{P} \mathbf{K}(x_i^m, x_j^m)$$

Many heuristic approaches also proposed to get a reasonable combination. Almost of them are represented in the form of linear combination of kernels or $f_\eta$ is linear:

$$\mathbf{K}_\eta(x_i, x_j) = \sum_{m=1}^{P} \eta_m \mathbf{K}(x_i^m, x_j^m) \tag{32}$$

where the kernel weights $\eta_m$ are defined by some heuristic rules. The work in [29] utilizes the conditional class probabilities on data labels to define values for $\eta_m$. The predictive performance (prediction accuracy and Pearson correlation) of each kernel is also used as a measurement for $\eta_m$ [30], [31].

**Parameter MKL: Linear combination** The linear combination of kernels in Equation (32) is the dominant way among MKL algorithms. It requires another computational step to find kernel weights $\eta_m$ that optimize the target functions. These target functions can be divided into two groups: similarity-based and structural risk functions. The learning weights is an important criteria which involves to the development of linear MKL algorithms. Many MKL methods share a target function but different conditions on $\eta_m$. Typically, these conditions are arbitrary: $\{\eta_m \in \mathbb{R}\}$, non-negative: $\{\eta_m \in \mathbb{R}^+\}$, convex: $\{\eta_m \in \mathbb{R}^+$ and $\sum_{m=1}^{P} \eta_m = 1\}$, and unit ball $\{||\eta||_2 = 1\}$ weights. To avoid the abuse of notation, we will discuss the main algorithm formulation with a short reference to the condition of weights.

The **similarity-based** functions are established on the notion of *kernel alignment* [32] to measure the similarity/correlation between two kernels:

$$A(\mathbf{K_1}, \mathbf{K_2}) = \frac{\langle \mathbf{K_1}, \mathbf{K_2} \rangle_F}{\sqrt{\langle \mathbf{K_1}, \mathbf{K_1} \rangle_F \langle \mathbf{K_2}, \mathbf{K_2} \rangle_F}} \tag{33}$$

or it is the cosine angle between two kernels $\mathbf{K_1}$ and $\mathbf{K_2}$. In classification problems, similarity-based MKL algorithms criteria is maximizing the alignment between combined kernel $\mathbf{K}_\eta$ and the *target* kernel $\mathbf{K_y}$.

$$\arg\max_{\eta} \quad A(\mathbf{K}_\eta, \mathbf{K_y})$$

where $\mathbf{K_y} = \mathbf{y}\mathbf{y}^T$ in binary classification problems and in general

$$\mathbf{K_y}(x_i, x_j) = \begin{cases} 1 & \text{if} \quad y_i = y_j \\ -1 & \text{if} \quad y_i \neq y_j \end{cases} \tag{34}$$

Intuitively, the *target* kernel reflects the similarity between samples through its labels, then similar samples should have higher pairwise distance and vice-versa. The learning kernel weights by alignment measurement utilizes this guidance to gain a better kernel combination. Some prominent *similarity-based* MKL articles can be grouped by the weight condition as follow:

- Arbitrary kernel weights: [33] formulates a learning problem as a semidefinite programming problem.
- Nonnegative kernel weights: [34] casts the maximize alignment to an equivalent quadratic programming problem.
- Kernel weights on a simplex algorithms are described in [35] and [36].
- Kernel weights on an unit ball is proposed along with *centered-kernel alignment* [9], [37]. In [9], the analytical solution is derived and it becomes an quadratic programming problem [37] when these weights are non-negative.

The **structural risk-based** MKL algorithms follow the structural risk minimization framework where SVM applied. Regardless the conditions on $\eta$, these algorithms generally try to maximize the SVM dual function with an additional parameter for linear kernel combination as in Equation (32).

$$\text{maximize } J(\eta) = \quad \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j \mathbf{K}_\eta(x_i, x_j) \quad \text{w.r.t} \quad \alpha \in \mathbb{R}_+^N$$

$$\text{s.t} \quad \mathbf{K}_\eta \succeq 0, \quad \sum_{i=1}^{N} \alpha_i y_i = 0, \quad \text{and} \quad 0 \leq \alpha_i \leq C \quad \forall i \in \{1, 2, \ldots, N\}$$

In [33], authors cast this objective as a SDP problem by adding an upper bound for $\mathrm{Tr}(\mathbf{K}_\eta)$. Along with encoding the prior knowledge about kernels, [38] derives a corresponding SOCP problem and propose an alternative QP problem. Typically, there are some extension as regularizer terms to get desire form of $\eta$. These constraints almost are $l_1$-norm regularization and some works on $l_2$-norm in [39], group lasso [40], and $l_1$-block lasso [8].

**Parameter MKL: Nonlinear combination** is another approach that develops the nonlinearity into MKL. In [41], authors propose a general optimization framework in learning general kernel combinations. For example, in gender identification task, the proposed framework performance is significantly better than the linear combination of kernels. In this data, their method learns the optimal weights $\eta$ in the product of Gaussian kernels:

$$\mathbf{K}_\eta(x_i, x_j) = \prod_{m=1}^{P} \exp(-\eta_m (x_i^m - x_j^m)^2)$$

thus the combined kernel is the production of multiple scaling in kernel features.

A polynomial kernel combination is introduced in [42], the combined function is:

$$\mathbf{K}_\eta = \sum_{m_1+m_2+\cdots+m_P=d} \eta_1^{m_1}\eta_2^{m_2}\dots\eta_P^{m_P}\mathbf{K}_1^{m_1}\mathbf{K}_2^{m_2}\dots\mathbf{K}_P^{m_P}$$

where $d$ is the bound of total orders. This method shows a significant improvement on some UCI datasets where the linear combination can not consistently improve the performance.

### 3.2.2 Overall comparison

In [6], authors do a performance testing for MKL algorithms on 10 experiments. There are no algorithm that is always better than others in terms of accuracy. In the other hand, we can observe some perspective from experiments results of [6].

- In linear combination kernels, non-linear MKL algorithms is seem promising when it give a significant improvements in some cases.
- In combining Gaussian kernels, the linear combination is better than others type of combination.
- The *centered-kernel alignment* with conic weights algorithm [37] always gains some accuracy improvement. Interestingly, different from other MKL algorithms, this approach is presented with the theoretical guarantee.
- The untrained, unweighted combination of kernels is not trivial in term of predictive performance, other trained MKL algorithms still can not beat this approach in all experiments.

### 3.2.3 AlignF algorithm

Due to the effectiveness of *centered-kernel alignment* algorithms, we focus on learning and making extension in this direction. Particularly, the main objective is the AlignF algorithm [37] which finds the maximizing centered kernel alignment.

**Centered kernel alignment** is developed from the uncentered kernel alignment which is introduced [32]. In [37], authors figure out that the centered or normalized kernels is the critical component to improve the performance. The centered kernel definition for a kernel matrix $\mathbf{K} \in \mathbb{R}^{n\times n}$ is:

$$\mathbf{K}^C = C\mathbf{K}C \quad ; \quad C = \left[\mathbf{I} - \frac{\mathbf{1}\mathbf{1}^\mathbf{T}}{n}\right]$$

where the sum of rows (columns) of $\mathbf{K}^C$ is zeros. Then the alignment task becomes

$$\arg\max_{\eta} \quad A(\mathbf{K}_\eta^C, \mathbf{K}_\mathbf{y}^C) = \frac{\langle \mathbf{K}_\eta^C, \mathbf{K}_\mathbf{y} \rangle_F}{||\mathbf{K}_\eta^C||_F}$$

$$\text{s.t} \quad ||\eta||_2 = 1, \ \eta \geq 0$$

as $\langle \mathbf{K}_\eta^C, \mathbf{K}_\mathbf{y}^C \rangle_F = \langle \mathbf{K}_\eta^C, \mathbf{K}_\mathbf{y} \rangle_F$ and $||\mathbf{K}_\mathbf{y}^C||_F$ is a constant so we can omit it. In detail, we need to solve this optimization problem:

$$\max_\eta \frac{\langle \sum_{m=1}^P \eta_m \mathbf{K}_m^C, \mathbf{K}_\mathbf{y} \rangle_F}{||\mathbf{K}_\eta^C||_F} = \max_\eta \frac{\sum_{m=1}^P \eta_m \langle \mathbf{K}_m^C, \mathbf{K}_\mathbf{y} \rangle_F}{\sqrt{(\sum_{i=1}^P \sum_{j=1}^P \eta_i \eta_j \langle \mathbf{K}_i^C, \mathbf{K}_j^C \rangle_F)}}$$

$$\text{s.t} \quad ||\eta||_2 = 1, \ \eta \geq 0$$

**AlignF via Quadratic programming** In [37], this problem can be solved by calling a quadratic programming solver that:

$$\min_{\eta \geq 0} \quad -c^T \eta + \frac{1}{2} \eta^T \mathbf{Q} \eta$$

where $c$ is a vector of size $P$ and $\mathbf{Q}$ is a symmetric matrix of $P$-by-$P$ such that:

$$\begin{aligned} c(i) &= \langle \mathbf{K}_i^C, \mathbf{K}_\mathbf{y} \rangle_F \\ Q(i,j) &= \langle \mathbf{K}_i^C, \mathbf{K}_j^C \rangle_F \end{aligned}$$

then the solution is $\eta^*/||\eta^*||$.

**Complexity analysis** In solving the AlignF problem, the heaviest step is getting information for matrix $\mathbf{Q}$. It requires $P(P-1)/2$ times to evaluate the correlation between $P$ kernels and takes $n^2$ loops to calculate the Forbenious product between two kernels. Then, the complexity for getting $\mathbf{Q}$ is $\mathcal{O}(P^2 n^2)$. Similarly, for vector $c$, the complexity is $\mathcal{O}(Pn^2)$.

The main computational burden is getting $\mathbf{Q}$ and it costs much more computational resources when more kernels or more samples are available. For example, in the case of pairwise learning, we need to learn the model for two objects rather than a single one. Follow that, the training kernel is the Kronecker product of kernels from two objects. When multiple kernels appear, we need to select the best combination between all possible of pairwise kernels. Specifically, if there are 1000 samples, 10 kernels for each object, then the complexity of getting matrix $\mathbf{Q}$ for AlignF requires $(10^{16})$ arithmetic operations, which is a considerable huge number for a small dataset.

# 4 Tensor method for Multiple Kernel Learning

In another perspective, when putting $P$ kernels $\mathbf{K} \in \mathbb{R}^{n \times n}$ together we get an order-3 tensor of $\mathcal{K} \in \mathbb{R}^{n \times n \times P}$. The main topic of this section is how to utilize well-founded tensor computation operators and properties in improving Multiple Kernel Learning algorithms in terms of performance and computational complexity.

## 4.1 Factorized AlignF algorithm

For a large dataset, more samples and multiple view information lead to a huge tensor of kernels $\mathcal{K} \in \mathbb{R}^{n \times n \times P}$. It is more critical to find a way to reduce the complexity of AlignF algorithm in large multiple view datasets. By the tensor learning approach, Tensor Train algorithm can factorize $\mathcal{K}$ into low-rank matrices, to lessen the computational burden that is required by the AlignF algorithm.

**Overall framework** The framework steps are described in Figure 21, it needs one more Tensor Train decomposition step in the learning procedure. The tensor $\mathcal{K} \in \mathbb{R}^{n \times n \times P}$ of kernels is firstly reshaped into $\mathcal{K} \in \mathbb{R}^{n \times P \times n}$. The Tensor Train algorithm decomposes this tensor $\mathcal{K}$ into there components:

- $\mathcal{G}_1 \in \mathbb{R}^{1 \times n \times r_1}$
- $\mathcal{G}_2 \in \mathbb{R}^{r_1 \times P \times r_2}$
- $\mathcal{G}_3 \in \mathbb{R}^{r_2 \times n \times 1}$

The $\mathcal{G}_1$ and $\mathcal{G}_3$ are matrices and the $\mathcal{G}_2$ is a tensor which stacks $P$ matrices of size $r_1$-by-$r_2$. Intuitively, $\mathcal{G}_1$ and $\mathcal{G}_3$ structure are not dependent on the order of original kernels, so they are shared parts. The tensor $\mathcal{G}_2$ contains $P$ low-rank matrices corresponding to $P$ kernels, so they are compressed parts for kernels. Nevertheless, the tensor $\mathcal{K}$ is symmetry, then it is more sensible when its compressed parts are also square. Then, we assign the number of low-rank components are equal for the first and second dimensions. This leads to $r_1 = r_2 = r$ and the corresponding components after the Tensor Train decomposition are:

- $\mathbf{G}_1 \in \mathbb{R}^{n \times r}$
- $\mathcal{G}_2 \in \mathbb{R}^{r \times P \times r}$
- $\mathbf{G}_3 \in \mathbb{R}^{r \times n}$

According to (30), the approximation for an element in the tensor $\mathcal{K}$ is:

$$\mathcal{K}(i_1, i_2, i_3) \approx \mathbf{G}_1[i_1]\mathcal{G}_2[i_2]\mathbf{G}_3[i_3] \tag{35}$$

where the quantity of $\mathbf{G}_1[i_1]$ and $\mathbf{G}_3[i_3]$ are vectors of size 1-by-$r$ and $r$-by-1, respectively. Similarly, the $\mathcal{G}_2[i_2]$ is a matrix of the size $r$-by-$r$ , where $i_2 \in [1, 2, \ldots, P]$.

As the second dimension of tensor $\mathcal{K} \in \mathbb{R}^{n\times P\times n}$ indicates the kernel indicies, the formula for approximating each kernel is:

$$\mathcal{K}(:, i_2, :) \approx \mathbf{G}_1[:]\mathcal{G}_2[i_2]\mathbf{G}_3[:] \quad \text{or} \quad \mathcal{K}(:, i_2, :) \approx \mathbf{G}_1\mathcal{G}_2[i_2]\mathbf{G}_3 \tag{36}$$

We denote that $\mathbf{K}^L = \mathbf{G}_1$ and $\mathbf{K}^R = \mathbf{G}_3$ are the left and right common matrices in decomposed parts. There are $P$ matrices in $\mathcal{G}_2$ as $\mathbf{K}^M_{(i)}$ are specific compressed parts for each kernel $\mathbf{K}_{(i)}$. Then, the equivalent description for Equation (36) is:

$$\mathbf{K}_{(i)} \approx \mathbf{K}^L\mathbf{K}^M_{(i)}\mathbf{K}^R \tag{37}$$

By embedding these components as a kernel approximation into the AlignF formu-

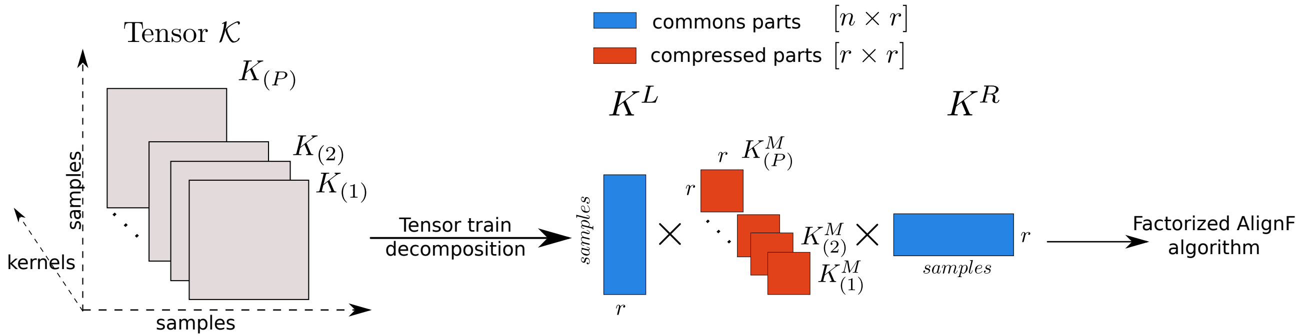



Figure 21: The framework of Factorization AlignF

lation, we get the new factorized AlignF algorithm.

**Factorization into AlignF** There are two main formula in calling AlignF optimization framework which are in getting the correlation between target and input kernels and among input kernels. When replacing AlignF formulas by the approximation form for each kernel at (37), we get

$$\begin{aligned} \langle \mathbf{K}^C_{(i)}, \mathbf{K}_Y \rangle_F &= \mathrm{Tr}(\mathbf{C}\mathbf{K}^L\mathbf{K}^M_{(i)}\mathbf{K}^R\mathbf{C}\mathbf{K}_Y) \\ &= \mathrm{Tr}\{(\mathbf{C}\mathbf{K}^L)\mathbf{K}^M_{(i)}(\mathbf{K}^R\mathbf{C}\mathbf{K}_Y)\} \\ &= \mathrm{Tr}\{(\mathbf{K}^R\mathbf{C}\mathbf{K}_Y\mathbf{C}\mathbf{K}^L)\mathbf{K}^M_{(i)}\} \end{aligned} \tag{38}$$

$$\begin{aligned} \langle \mathbf{K}^C_{(i)}, \mathbf{K}^C_{(j)} \rangle_F &= \mathrm{Tr}(\mathbf{C}\mathbf{K}_{(i)}\mathbf{C}\mathbf{K}_{(j)}) = \mathrm{Tr}(\mathbf{C}\mathbf{K}^L\mathbf{K}^M_{(i)}\mathbf{K}^R\mathbf{C}\mathbf{K}^L\mathbf{K}^M_{(j)}\mathbf{K}^R) \\ &= \mathrm{Tr}\{\mathbf{K}^R\mathbf{C}\mathbf{K}^L\mathbf{K}^M_{(i)}\mathbf{K}^R\mathbf{C}\mathbf{K}^L\mathbf{K}^M_{(j)}\} \end{aligned} \tag{39}$$

In these formulas, we can approximate the n-wise multiple kernels learning into factors for the AlignF algorithm. The first advantage is some parts of these formulas (blue parts) can be precomputed before hand as they are constant value. Moreover, most computationally heavy part is less complex, it is a multiplication of $r$-by-$r$ matrices instead of $n$-by-$n$ matrices. When the low-rank assumption is matched in data kernels, the compression ratio is relatively high, for example: if $n = 10r$ then the ratio is $\frac{n^2}{r^2} = 10^2 = 100$ times.

**Factorization into AlignF for pairwise kernel learning** In pairwise kernel learning, this approximation is a crucial speedup in terms of computational speed and memory. For example, in drug-target interaction learning problem, if there are $n_d$ kernels $\mathbf{K}_d$ of size $n_1$-by-$n_1$ for drugs, $n_t$ kernels $\mathbf{K}_t$ of size $n_2$-by-$n_2$ for targets. Then, there are $n_d n_t$ Kronecker product kernels in the alignment process which will cost in the quadratic scale to finish the computation. Thanks to the Tensor Train approximation for kernels in the drug and target side, the Kronecker product between two kernels are not necessary to be evaluated and stored explicitly. For example, after decomposition multiple kernels in each drug and target side, we can approximate the Kronecker product between each pair of kernel:

$$\begin{aligned} \mathbf{K}_{d(i)} \otimes \mathbf{K}_{t(j)} &\approx (\mathbf{K}_d^L \mathbf{K}_{d(i)}^M \mathbf{K}_d^R) \otimes (\mathbf{K}_t^L \mathbf{K}_{t(j)}^M \mathbf{K}_t^R) \\ &\approx \underbrace{(\mathbf{K}_d^L \otimes \mathbf{K}_t^L)}_{\mathbf{K}_{dt}^L} \underbrace{(\mathbf{K}_{d(i)}^M \otimes \mathbf{K}_{t(j)}^M)}_{\mathbf{K}_{dt(i,j)}^M} \underbrace{(\mathbf{K}_d^R \otimes \mathbf{K}_t^R)}_{\mathbf{K}_{dt}^R} \end{aligned} \tag{40}$$

As the property of the Kronecker product (16), the approximation formula is factorized into 3 components, the left and right common parts $(\mathbf{K}_{dt}^L, \mathbf{K}_{dt}^R)$ are built from shared parts from each side. Interestingly, the middle part $\mathbf{K}_{dt(i,j)}^M$ for each pair is the Kronecker product between middle parts in each side. Moreover, this approximation can be used in speed up other algorithms which manipulate on pairwise kernels. For example, in the AlignF algorithm for pairwise kernels, when putting Equation (40) into (38) and (39), the whole complexity will decrease from the multiplication among $n_1 n_2$-by-$n_1 n_2$ matrices to $r_1 r_2$-by-$r_1 r_2$ matrices, where $r_1$ and $r_2$ is the low-rank number in approximation multiple kernels in the drug and target side.

## 4.2 The optimal decomposition for the tensor of kernels

In decomposing a tensor of kernels, TT algorithm calls the Singular Vector Decomposition two times after reshaping the non-decomposed parts. Figure 22 clearly shows these steps and $\mathbf{K}^L$ is the common left part after the decomposition. It contains left singular vectors of the first SVD step which decomposes the horizontal stacking of kernels. Thus, if a left singular vector in $\mathbf{K}^L$ is denoted as $\vec{a}$ or $a$ and the corresponding right singular vector for each kernel $\mathbf{K}_{(i)}$ is $\vec{b_i}$ or $b_i$, then this step is equivalent to minimize this objective function:

$$\min_{a,b_i} \quad \sum_{i=1}^{P} ||\mathbf{K}_{(i)} - \lambda a b_i^T||_F^2 \quad \text{s.t} \quad ||a||_2 = 1 \ , ||b_i||_2 = 1 \quad \forall i \in [1, 2, \dots, P] \tag{41}$$

where $\lambda$ is the singular value. In one term of this summation, we can get this equivalent formula:

$$||\mathbf{K}_{(i)} - \lambda a b_i^T||_F^2 = \mathrm{Tr}(\mathbf{K}_{(i)}^T \mathbf{K}_{(i)}) - 2\lambda a^T \mathbf{K}_{(i)} b_i + \lambda^2$$

Then the objective function is:

$$\min_{a,b_i} \sum_{i=1}^{P} \left( \mathrm{Tr}(\mathbf{K}_{(i)}^T \mathbf{K}_{(i)}) - 2\lambda a^T \mathbf{K}_{(i)} b_i + \lambda^2 \right) \tag{42}$$

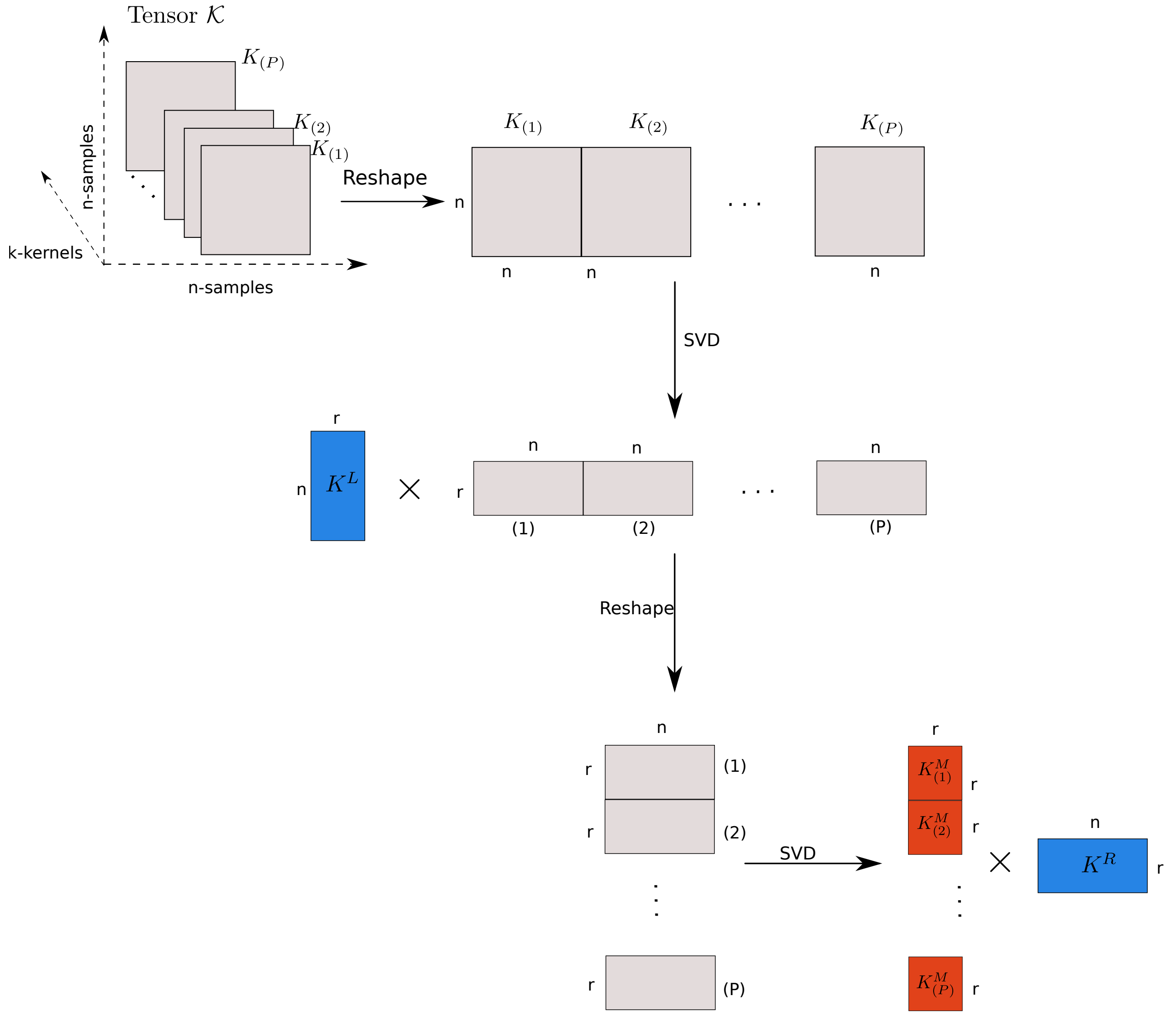



Figure 22: The Tensor Train algorithm decomposition steps for an order 3 tensor which is created by stacking kernels

The objective function (42) is defined as $f = \sum_{i=1}^{P} \left(\mathrm{Tr}(\mathbf{K}_{(i)}^T \mathbf{K}_{(i)}) - 2\lambda a^T \mathbf{K}_{(i)} b_i + \lambda^2\right)$ and its derivatives:

$$\begin{aligned} \frac{\partial f}{\partial a} &= -2\lambda \sum_{i=1}^{P} (\mathbf{K}_{(i)} b_i) \\ \frac{\partial f}{\partial b_i} &= -2\lambda \mathbf{K}_{(i)}^T a \end{aligned}$$

Then, to minimize this objective function, the best gradient direction of $\vec{a}$ and $\vec{b}_i$ are $\sum_{i=1}^{P} (\mathbf{K}_{(i)} b_i)$ and $\mathbf{K}_{(i)}^T a$, respectively. By using the power of iteration method, we

can find the optimal $\vec{a}, \vec{b_i}$ by iteratively calculating two quantities:

$$\begin{aligned} a &= \frac{\sum_{i=1}^{P}(\mathbf{K}_{(i)}b_i)}{||\sum_{i=1}^{P}(\mathbf{K}_{(i)}b_i)||} \\ b_i &= \frac{\mathbf{K}_{(i)}^T a}{||\mathbf{K}_{(i)}^T a||} \end{aligned}$$

or using this equivalent formula for $\vec{a}$:

$$a = \frac{\sum_{i=1}^{P}(\mathbf{K}_{(i)}\mathbf{K}_{(i)}^T)a\frac{1}{||\mathbf{K}_{(i)}^T a||}}{||\sum_{i=1}^{P}(\mathbf{K}_{(i)}\frac{\mathbf{K}_{(i)}^T a}{||\mathbf{K}_{(i)}^T a||})||}$$

This indicates that $\vec{a}$ is a singular vector of the matrix $\sum_{i=1}^{P}(\mathbf{K}_{(i)}\mathbf{K}_{(i)}^T)$. Then the $K^L$ matrix can be obtained by taking $r$ singular vectors corresponding $r$ largest singular values of the decomposition of $\sum_{i=1}^{P}(\mathbf{K}_{(i)}\mathbf{K}_{(i)}^T)$.

Intuitively, as all kernels are symmetric, we can infer that $\mathbf{K}^M$ are also symmetric and thus $\mathbf{K}^L = (\mathbf{K}^R)^T$. We denote the column orthogonal matrix $\mathbf{U} = \mathtt{SVD}(\sum_{i=1}^{P}(\mathbf{K}_{(i)}\mathbf{K}_{(i)}^T))$ and thus $\mathbf{K}^L = \mathbf{U}, \mathbf{K}^R = \mathbf{U}^T$. In the exact decomposition, a kernel is described as:

$$\mathbf{K}_{(i)} = \mathbf{K}^L\mathbf{K}_{(i)}^M\mathbf{K}^R \quad \text{or} \quad \mathbf{K}_{(i)} = \mathbf{U}\mathbf{K}_{(i)}^M\mathbf{U}^T$$

Then we can project back to get the compressed kernel $\mathbf{K}_{(i)}^M$:

$$\mathbf{K}_{(i)}^M = \mathbf{U}^T\mathbf{K}_{(i)}\mathbf{U}$$

The final step is checking whether $\mathbf{K}_{(i)}^M$ is symmetric under these solutions. $\mathbf{K}_{(k)}^M(i,j)$ denotes the element at row, column index $(i,j)$ from $k$th compressed middle kernel. The matrix $\mathbf{U}$ includes $r$ columns according $r$ orthogonal vectors $u_i$ or $\mathbf{U} = [u_1|u_2|\ldots u_{r-1}|u_r]$.

$$K_{(k)}^M(i,j) = u_i^T\mathbf{K}_{(k)}u_j$$

Its transpose is:

$$(K_{(k)}^M(i,j))^T = u_j^T\mathbf{K}_{(k)}u_i = K_{(k)}^M(j,i)$$

as a kernel $\mathbf{K}_{(k)}$ is symmetric then we get above consequence.

**Algorithm 5:** The optimal decomposition for a tensor of kernels

**Data:** A tensor of $P$ kernels $\mathbf{K}_{(i)}$ of $n$ samples: $\mathcal{K} \in \mathbb{R}^{n \times P \times n}$
**Input:** The number of low-rank components: $r$
**Output:** $\mathbf{K}^L, \{\mathbf{K}^M_{(i)}\}_{i=1}^P, \mathbf{K}^R$
$\mathbf{K_A} = \sum_{i=1}^P (\mathbf{K}_{(i)} \mathbf{K}^T_{(i)})$ ;
`/* SVD decomposition */`
$[\mathbf{U}, \mathbf{S}, \mathbf{V}] = \mathtt{svd}(\mathbf{K_A})$ ;
`/* Get` $\mathbf{K}^L, \mathbf{K}^R$ `*/`
$\mathbf{U} = U(:, 1:r)$ ;
$\mathbf{K}^L = \mathbf{U}$; $\mathbf{K}^R = \mathbf{U}^T$ ;
**for** $i = [1, 2, \ldots, P]$ **do**
  `/* Project back */`
  $\mathbf{K}^M_{(i)} = \mathbf{U}^T \mathbf{K}_{(i)} \mathbf{U}$ ;

# 5 Experiments

There are two sets of experiments to evaluate the performance of these algorithms: Factorized AlignF (AlignF_TT), AlignF, and UniMKL which takes the average of kernels as the baseline. The first experiment runs on datasets which are generated by a prior assumption on the data distribution. This experiment also aims to analyse when the similarity alignment algorithms are effective in the multiple kernel learning framework. The real world datasets are used in the second set of experiments, to analyse the number of effective low-rank number for the Factorized AlignF algorithm.

## 5.1 Data and experiment setup

### 5.1.1 Artificially constructed datasets

The artificially constructed dataset is a complex multimodal or mixture of Gaussian distribution [43] in which the generation process can be fully controlled. This data is created by the mixing of clusters; each cluster is independently generated by a Gaussian distribution and identical labels for samples in a cluster. Thus, it is an ideal environment to evaluate the advantages and weakness of machine learning methods.

Figure 23 shows factors: $\sigma_\mu$, $\mu$, and $\sigma$ that influence on the generation of a sample $x$. Firstly, cluster centers $\mu$ are sampled around the origin with a variance $\sigma_\mu$ by the Gaussian distribution $\mathcal{N}(0, \sigma_\mu)$. Secondly, samples in each cluster are generated as $x \sim \mathcal{N}(\mu, \sigma)$. Thus, the parameter $\sigma_\mu$ controls the allocation of clusters over the data space. Similarly, the $\sigma$ handles the diameter of the cloud that samples expanding in a cluster.

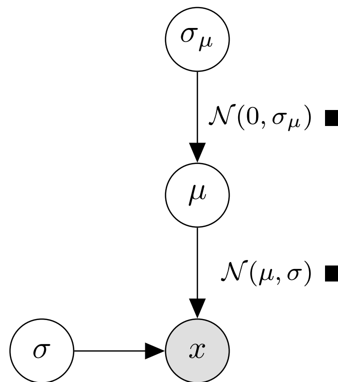



Figure 23: The graphical model of the generation process for a sample

Consequently, changing values of $\sigma_\mu$ and $\sigma$ will generate clusters with different level of overlapping among them. On the other hand, it is corresponding to the separableness of samples and will affect the performance of learning algorithms. Due to these reasons, multiple datasets with different level of overlapping are generated

by increasing the scale of $\sigma_\mu$ with an appropriate large value of $\sigma$.

**Algorithm 6:** Multiple artificial generated datasets generation

**Input:** $ndim$, $nblock$, $nlabel$, $mblock$: the number of dimension, clusters, labels in whole data, and samples in each cluster, respectively.
**Input:** $\sigma_\mu$, $\sigma$: variance parameters for generating centers and samples.
**Input:** $ndata$: the number of datasets
**Output:** Datasets $\{D\}_1^{ndata}$

```
/* Generating cluster centers                                        */
for i = [1, 2, ..., nblock] do
    μ_i ~ N(0, σ_μ) ;
    label_i = rand([1, nlabel]) ;
/* Generating multiple datasets                                      */
for d = [1, 2, ..., ndata] do
    D_d = [] ;
    scale_d = 1.2^d ;
    for i = [1, 2, ..., nblock] do
        μ_id = μ_i × scale_d ;
        for k = [1, 2, ..., mblock] do
            x.feature ~ N(μ_id, σ) ;
            x.label = label_i ;
            D_d.append(x) ;
```

By setting that $ndata = 9, ndim = 2, nlabel = 4, nblock = 8, mblock = 100, \sigma_\mu = 1, \sigma = 0.6$, we can visualize datasets which generated by the Algorithm **??**.

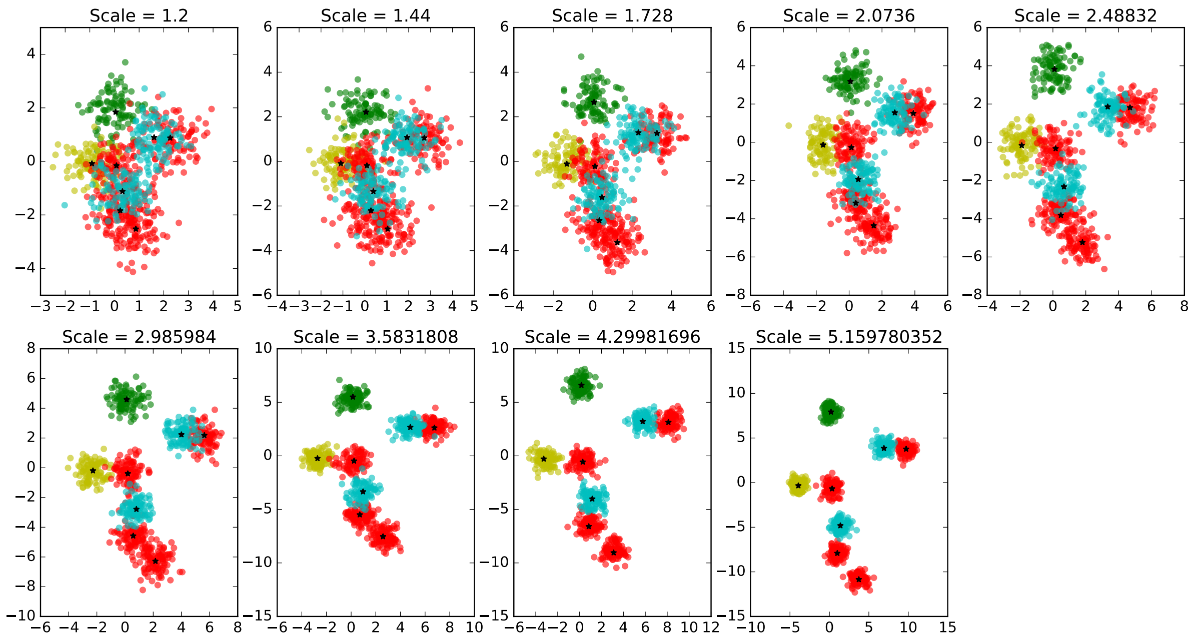



Figure 24: Toy datasets in different scales

The corresponding linear kernels for each dataset are:

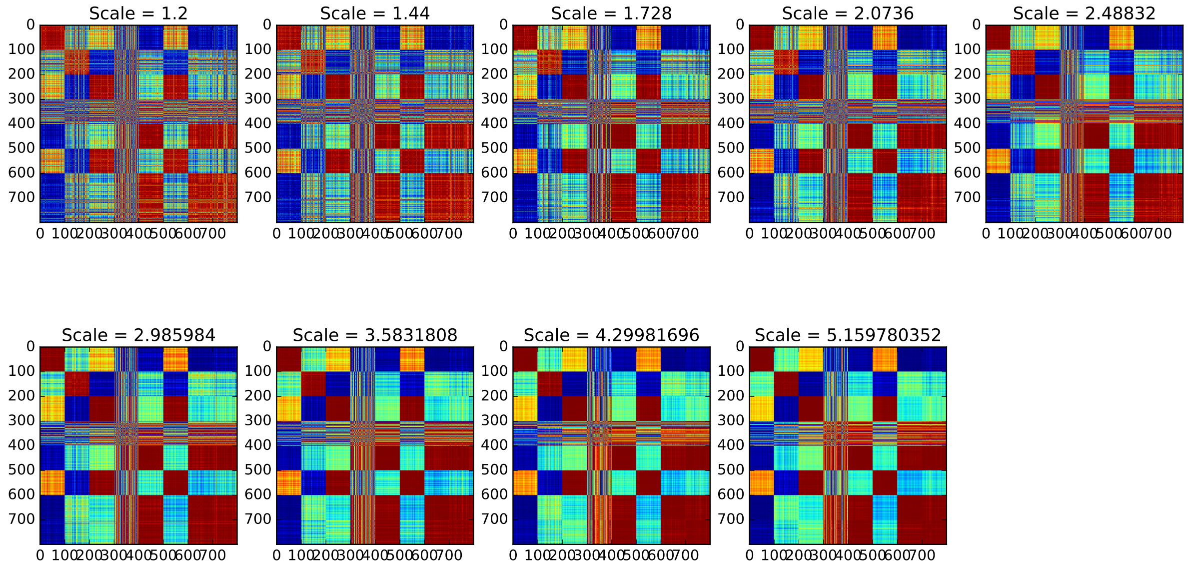



Figure 25: Linear kernels in different scale datasets

Multiple kernels are generated by computing the RBF kernels in different $\gamma$ values. Additionally, one linear kernel $xx^T$ is included to the kernel set. There are 11 values of $\gamma$ in the set $\{10^{-7}, 10^{-6}, \dots, 10^{1}, 10^{0}, 5\}$.

Finally, there are 18 artificial datasets are generated and evaluated. The set *toy-data-1* contains 8 datasets are created by using the Algorithm 6 with these parameters: $ndata = 9, ndim = 20, nlabel = 4, nblock = 8, mblock = 100, \sigma_{\mu} = 1, \sigma = 2.5$. Similarity, the remaining 8 toy datasets in the set *toy-data-2* are generated by using this setting: $ndata = 9, ndim = 20, nlabel = 10, nblock = 40, mblock = 200, \sigma_{\mu} = 1, \sigma = 2.5$. The Table 1 summarizes these datasets in terms of size, label and kernels.

### 5.1.2 Real datasets

There 13 real datasets are evaluated: 3 public datasets, 5 bioinformatics datasets, and 5 image annotation datasets. Three multilabel datasets *Emotions*, *Yeast*, *Enron* are downloaded from the Mulan library (http://mulan.sourceforge.net/datasets-mlc.html). The *Emotions* dataset is used for the emotion detection [44] in music where a piece of song may belong to many classes. It contains 593 songs and 6 clusters of music emotions. The *Yeast* dataset [45] describes 1500 genes in terms of its micro-array expression and phylogenetic profile, along with 14 labels for gene functional classes. The *Enron* dataset includes about 1700 labeled emails and the downloaded data already be preprocessed from texts to binary vectors. There are 53 labels in this dataset, for example: company strategy, personal message, newsletters, joke. For these datasets, multiple kernel matrices are created by the Gaussian kernel with changing $\gamma$ in the set of $[2^{-13}, 2^{-11}, 2^{-9}, 2^{-7}, 2^{-5}, 2^{-3}, 2^{-1}, 2^{1}, 2^{3}]$.

Five bioinformatics datasets are *psortPos*, *psortNeg*, *Plant*, *Protein*, and *Fingerprint*. Three multiclass datasets *psortPos*, *psortNeg*, *Plant* are getting from [46] where kernels are based on the similarity of sequence motif, phylogenetic trees, and BLAST E-values. The *Protein* is also a multiclass dataset which describes

| | Samples | Labels | Kernels |
|---|---|---|---|
| *Toy-data-1-scale=1.1* | 800 | 4 | 11 |
| *Toy-data-1-scale=1.4* | 800 | 4 | 11 |
| *Toy-data-1-scale=1.7* | 800 | 4 | 11 |
| *Toy-data-1-scale=2.0* | 800 | 4 | 11 |
| *Toy-data-1-scale=2.4* | 800 | 4 | 11 |
| *Toy-data-1-scale=2.9* | 800 | 4 | 11 |
| *Toy-data-1-scale=3.5* | 800 | 4 | 11 |
| *Toy-data-1-scale=4.2* | 800 | 4 | 11 |
| *Toy-data-1-scale=5.1* | 800 | 4 | 11 |
| *Toy-data-2-scale=1.1* | 8000 | 10 | 11 |
| *Toy-data-2-scale=1.4* | 8000 | 10 | 11 |
| *Toy-data-2-scale=1.7* | 8000 | 10 | 11 |
| *Toy-data-2-scale=2.0* | 8000 | 10 | 11 |
| *Toy-data-2-scale=2.4* | 8000 | 10 | 11 |
| *Toy-data-2-scale=2.9* | 8000 | 10 | 11 |
| *Toy-data-2-scale=3.5* | 8000 | 10 | 11 |
| *Toy-data-2-scale=4.2* | 8000 | 10 | 11 |
| *Toy-data-2-scale=5.1* | 8000 | 10 | 11 |

Table 1: Toy datasets summarization

the functions of transporter proteins and is downloaded from the TCDB database (http://www.tcdb.org/public/). There are 4 kernels based on BLAST score with UniPro, taxonomy information, protein family, and BLAST score with TCDB. The *Fingerprint* data are taken from [47] which contains 1000 mass spectra and corresponding 101 labels of molecular fingerprints.

Five images datasets *Core5k*, *Espgame*, *Iaprtc12*, *Mirflickr*, and *Pascal* are taken from [47]. Noted that, these are originally come from [48] but this work uses the *smaller* version of them [47] which contains a subset of 1000 samples. There are 15 precomputed numerical vector features for each sample in these datasets and multiple kernels are generated by using the linear kernel. According to [47], some labels are extremely unbalance, so these data labels are filtered to include labels that exist more than 2% of positive.

In summary, the Table 2 contains key information about 13 datasets:

#### 5.1.3 Experiments setup

The Support Vector Machine is the learning algorithm that used in all experiments as datasets are labeled and only using the kernel information. The *Scikit-learn* library in Python [49] is used for SVM algorithm implementation, stratified sampling, kernel centering, grid search parameters, and performance metric evaluation functions.

In a dataset, the 5-fold nested cross validation is done to get the average performance for Multiple Kernel Learning algorithms: AlignF, Factorized AlignF, UniMKL. In a fold, the training kernels is created and centered before inputting it into MKL

| | **Samples** | **Labels** | **Kernels** |
|---|---|---|---|
| *Emotions* | 593 | 6 | 9 |
| *Yeast* | 1500 | 13 | 9 |
| *Enron* | 1702 | 24 | 9 |
| *Fingerprint* | 1000 | 101 | 12 |
| *Protein* | 1060 | 30 | 4 |
| *Corel5k* | 1000 | 37 | 15 |
| *Espgame* | 1000 | 52 | 15 |
| *Iaprtc12* | 1000 | 67 | 15 |
| *Miflickr* | 1000 | 15 | 15 |
| *Pascal07* | 1000 | 6 | 15 |
| *psortPos* | 541 | 1 | 69 |
| *psortNeg* | 1444 | 1 | 69 |
| *Plant* | 940 | 1 | 69 |

Table 2: Real world datasets summarization

algorithms. Each MKL algorithm takes a list of centered training kernels and the normalized target kernel which is built from samples labels. After calling MKL algorithms, they return the linear combination weights for kernels. Follow that, the final training kernel is taken as the weighted sum of training kernels. In training the SVM model, the $C$ parameter is optimized by splitting the training data to 5 parts and evaluated the predictive performance in 1 part with the learned model in 4 parts. The search range of $C$ is in the set of $\{10^{-4}, 10^{-3}, \ldots, 10^{4}, 10^{5}\}$ to pick the best $C$ value in terms of predictive performance.

In the testing phase, the testing kernel are also centered according to the combined training kernel. In calling the SVM solver, we use the C-Support Vector Classification (SVC) of the Scikit-learn library. In multiclass classification problems, two schemes are one-vs-one and one-vs-all SVM. In 13 real world datasets, the label are created as a matrix as there are more than one label for a sample in some datasets. In the label matrix, each row stands for a sample and a column is the binary indicator for all samples that whether it belongs to a specific label or not. Then, each SVM calling for each label is a binary classification problem or one-vs-all in the multiclass classification problems. In toy datasets, we use the default one-vs-one SVM and performance metric functions of the Scikit-learn library for multiclass classification [50].

In these experiments, the target kernel is built one time for all labels rather than one target kernel for a label. For example, the label vectors $y_i = [a, b, c]$ and $y_j = [a, c]$ then $K(y_i, y_j) = 2/3$ as the common label set $[a, c]$ is 2 items over 3 labels in total. For single label datasets, the target kernel follows the rule in (34).

In term of AlignF_TT algorithm, there is one tuning parameter which is the low-rank $r$ of the SVD decomposition. In this work, $r$ is set to be $\frac{n}{10}$ as the default value whenever it is not clearly stated the value, where $n$ is the number of data sample.

**Evaluation measurements** The average of Accuracy, Micro F1, and Marco F1 over 5-fold are used to compare the predictive performance of algorithms.

## 5.2 Results

### 5.2.1 Artificial constructed datasets

**In the set *toy-data-1*** There are 8 datasets which are generated in an identical process with increasing the distance among cluster centers. Figure 26, 27, 28 show how the algorithms performance change in different samples distribution. Each tick in the *x-axis* is according to a dataset and a tick label is the scale number in the generating process. When increasing the scale, data clusters are less clutter and overlapping as clearly showed in Figure 24. Additionally, Table 3, 4, 5 shows evaluation measurements of MKL algorithms on different artificial constructed datasets.

| **Scale=** | *1.2* | *1.44* | *1.73* | *2.07* | *2.49* | *2.99* | *3.58* | *4.3* | *5.16* |
|---|---|---|---|---|---|---|---|---|---|
| *AlignF* | **63.125** | 73.875 | **83.875** | 92.375 | **96.625** | **98.** | **99.75** | **99.875** | 99.875 |
| *AlignF_TT* | 62.75 | 74.125 | **83.875** | **92.625** | 96.5 | **98.** | **99.75** | **99.875** | **100.** |
| *UniMKL* | 62.875 | **75.25** | 83.125 | 91.125 | 94.75 | **98.** | **99.75** | **99.875** | 99.875 |

Table 3: *Toy-data-1*: Accuracy change in different datasets

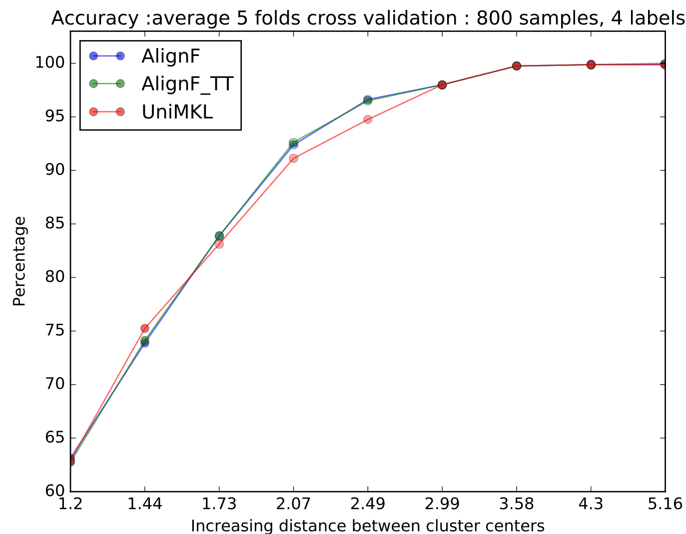



Figure 26: *Toy-data-1*:Accuracy change in different datasets

| **Scale=** | *1.2* | *1.44* | *1.73* | *2.07* | *2.49* | *2.99* | *3.58* | *4.3* | *5.16* |
|---|---|---|---|---|---|---|---|---|---|
| *AlignF* | **62.35** | 73.01 | **83.82** | 92.06 | **96.51** | **97.92** | **99.79** | **99.9** | 99.9 |
| *AlignF_TT* | 61.7 | 73.29 | 83.73 | **92.22** | 96.41 | **97.92** | **99.79** | **99.9** | **100.** |
| *UniMKL* | 62.17 | **74.23** | 82.64 | 91.22 | 95.07 | **97.99** | **99.79** | **99.9** | 99.9 |

Table 4: *Toy-data-1*:The macro-F1 change in different datasets

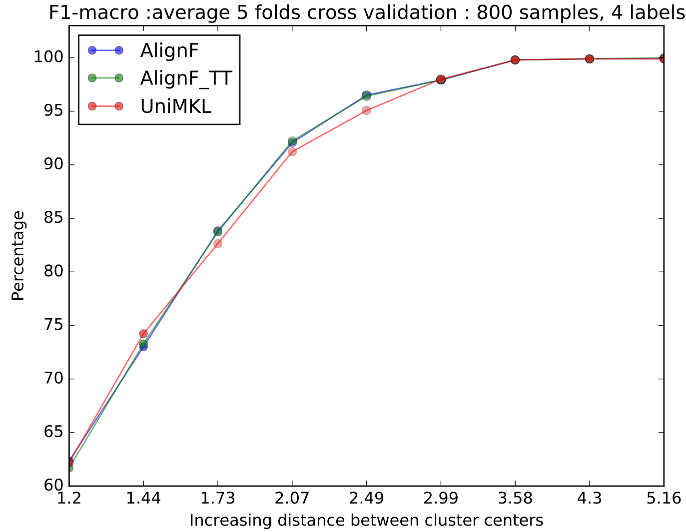



Figure 27: *Toy-data-1*:The macro-F1 change in different datasets

| **Scale=** | *1.2* | *1.44* | *1.73* | *2.07* | *2.49* | *2.99* | *3.58* | *4.3* | *5.16* |
|---|---|---|---|---|---|---|---|---|---|
| *AlignF* | **63.12** | 73.88 | 83.87 | 92.38 | **96.62** | **98.** | **99.75** | **99.88** | 99.88 |
| *AlignF_TT* | 62.75 | 74.12 | **83.88** | **92.62** | 96.5 | **98.** | **99.75** | **99.88** | **100.** |
| *UniMKL* | 62.87 | **75.25** | 83.12 | 91.13 | 94.75 | **98.** | **99.75** | **99.88** | 99.88 |

Table 5: *Toy-data-1*:The micro-F1 change in different datasets

At the *scale* are 1.2, 1.44, the heavy overlapping among clusters are observed at Figure 24 and 25, the alignment algorithms (AlignF, AlignF_TT) do not outperform the simple UniMKL strategy. The performance of AlignF, AlignF_TT are better when the scale increasing or data clusters are less clutter. For example, the accuracy improvement are 0.1%, 1.5% 1.9% when the $scale = 1.73, 2.07, 2.49$, respectively. When the clusters are well separable at the $scale = 2.99, 3.58, 4.3, 5.16$, the performance of 3 algorithms are higher than 95% and almost the same. Interestingly, when the *scale* is 5.16, only the proposed AlignF_TT algorithm is getting 100% in terms of accuracy, micro-F1, and macro-F1.

**In the set *toy-data-2*** There 8 datasets includes more labels (10 vs 4), clusters (40 vs 8), and samples (8000 vs 800) than datasets in the set of *toy-data-1*. In these datasets, the level of overlapping among clusters is higher than datasets in *toy-data-1* as more samples and clusters. Similarly, the accuracy, macro-F1, micro-F1 are shown in Table 6, 7, 8 and Figure 29, 30, 31 .

| **Scale=** | *1.2* | *1.44* | *1.73* | *2.07* | *2.49* | *2.99* | *3.58* | *4.3* | *5.16* |
|---|---|---|---|---|---|---|---|---|---|
| *AlignF* | **19.34** | **25.04** | 35.06 | **42.37** | 50.76 | 56.4 | 58.46 | 60.69 | **62.44** |
| *AlignF_TT* | 18.18 | 23.27 | **35.09** | **42.37** | **51.14** | **57.04** | **59.16** | **60.8** | **62.44** |
| *UniMKL* | 19.66 | 24.4 | 32.66 | 39.29 | 47.4 | 53.55 | 56.54 | 57.37 | 57.71 |

Table 6: *Toy-data-2*: Accuracy change in different datasets

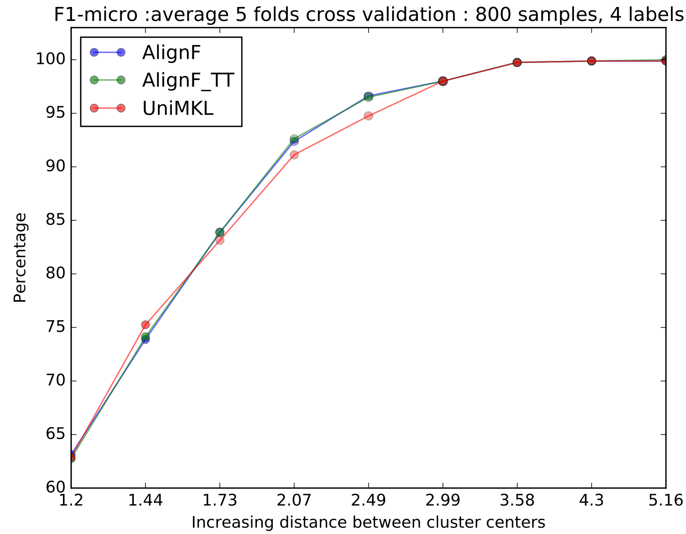



Figure 28: *Toy-data-1*:The micro-F1 change in different datasets

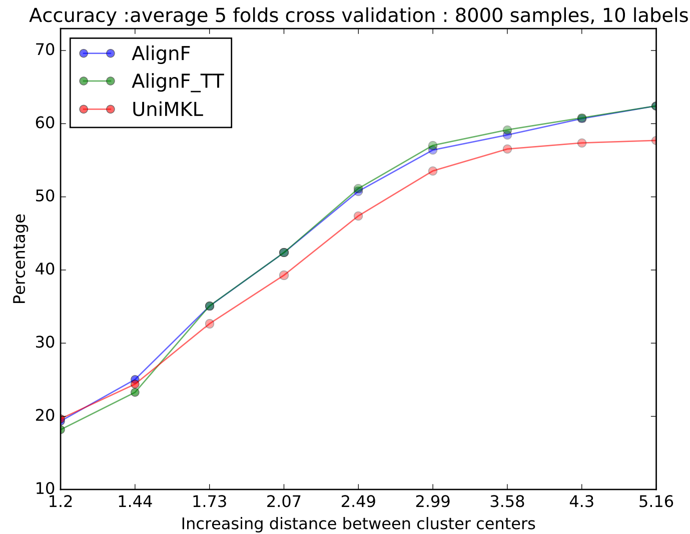



Figure 29: *Toy-data-2*:Accuracy change in different datasets

As in the set *toy-data-1*, there are no considerable performance improvement when datasets are generated in the small scale, at 1.2, 1.44. However, the performance of the AlignF_TT relatively decreases in these cases, around 1% lower than other algorithms. Similarly, increasing the scale shows the effectiveness of kernel alignment algorithms. In terms of accuracy, the performance improvement varies from 3% to around 5%.

These results indicates that the kernel alignment algorithms are useful when data are form by clusters and *enough* separable among them. The study [47] shows that the kernel alignment algorithms is equivalent to the non-negative least square between the vectorization of kernels. As the target kernel is also well-formed to clusters, a possible explanation for this might be that the alignments are more likely

| **Scale=** | *1.2* | *1.44* | *1.73* | *2.07* | *2.49* | *2.99* | *3.58* | *4.3* | *5.16* |
|---|---|---|---|---|---|---|---|---|---|
| *AlignF* | **21.31** | **27.18** | 37.34 | 43.97 | 51.84 | 56.98 | 58.54 | 61.13 | **63.97** |
| *AlignF_TT* | 19.75 | 25.62 | **37.35** | **44.14** | **52.41** | **57.78** | **59.45** | **61.17** | **63.97** |
| *UniMKL* | 21.51 | 26.43 | 34.83 | 40.65 | 48.38 | 53.68 | 55.83 | 56.83 | 57.61 |

Table 7: *Toy-data-2*:The macro-F1 change in different datasets

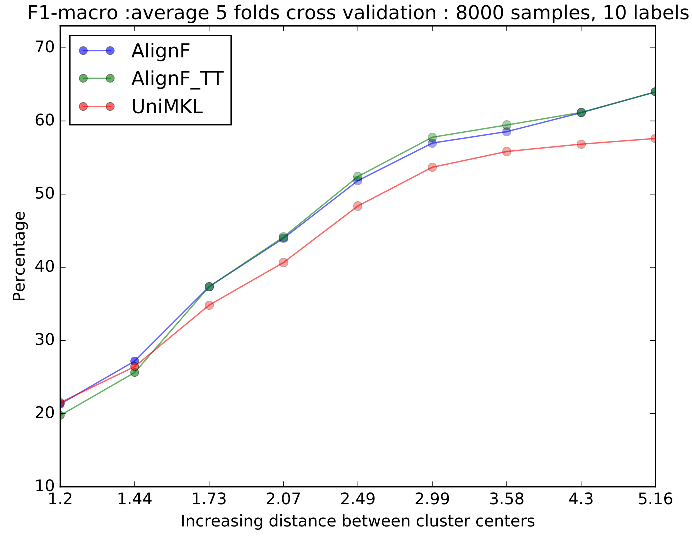



Figure 30: *Toy-data-2*:The macro-F1 change in different datasets

to be effective when other kernels reflect the cluster information.

| **Scale=** | *1.2* | *1.44* | *1.73* | *2.07* | *2.49* | *2.99* | *3.58* | *4.3* | *5.16* |
|---|---|---|---|---|---|---|---|---|---|
| *AlignF* | 19.34 | **25.04** | 35.06 | **42.37** | 50.76 | 56.4 | 58.46 | 60.69 | **62.44** |
| *AlignF_TT* | 18.18 | 23.27 | **35.09** | **42.37** | **51.14** | **57.04** | **59.16** | **60.8** | **62.44** |
| *UniMKL* | **19.66** | 24.4 | 32.66 | 39.29 | 47.4 | 53.55 | 56.54 | 57.37 | 57.71 |

Table 8: *Toy-data-2*:The micro-F1 change in different datasets

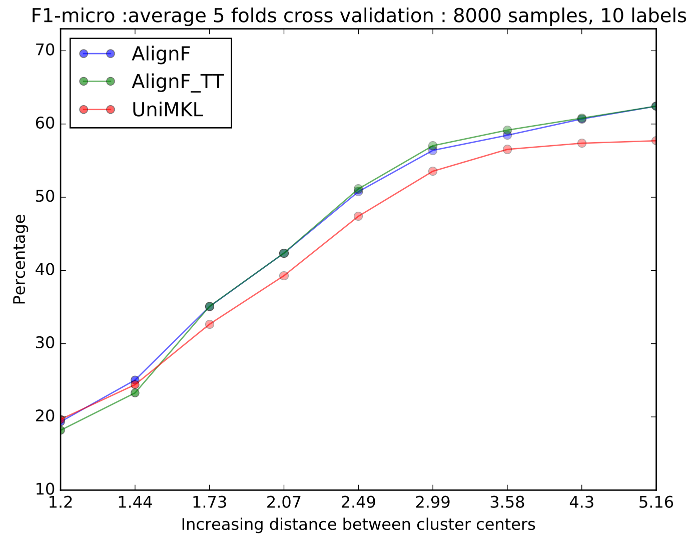



Figure 31: *Toy-data-2*:The micro-F1 change in different datasets

### 5.2.2 Real-world datasets

**Fixing the low-rank number for the AlignF_TT algorithm** The predictive performance of algorithms are reported in the form of average number and variance over 5 folds.

| | **AlignF** | **AlignF_TT** | **UniMKL** |
|---|---|---|---|
| *Emotions* | 81.86±0.09 | 81.64±0.08 | **81.95**±0.08 |
| *Yeast* | 79.75±0.06 | **79.77**±0.05 | 79.55±0.05 |
| *Enron* | **91.3**±0.0 | **91.3**±0.0 | 91.22±0.01 |
| *Fingerprint* | **86.18**±0.04 | 85.51±0.04 | 85.94±0.03 |
| *Protein* | **83.95**±0.07 | 83.87±0.08 | 83.6±0.07 |
| *corel5k* | **96.09**±0.03 | 96.0±0.03 | 96.05±0.03 |
| *espgame* | **94.78**±0.02 | 94.7±0.02 | 94.71±0.02 |
| *iaprtc12* | 94.53±0.02 | **94.57**±0.02 | 94.41±0.02 |
| *mirflickr* | **97.16**±0.02 | 97.14±0.02 | 97.11±0.02 |
| *pascal07* | **97.53**±0.03 | 97.49±0.04 | 97.52±0.04 |
| *psortPos* | **94.83**±0.1 | 94.38±0.19 | 91.52±0.17 |
| *psortNeg* | **96.67**±0.08 | 96.17±0.08 | 94.73±0.11 |
| *plant* | **96.28**±0.07 | 95.48±0.06 | 87.5±0.09 |

Table 9: The accuracy of algorithms over 13 datasets

| | **AlignF** | **AlignF_TT** | **UniMKL** |
|---|---|---|---|
| *Emotions* | 67.97±0.2 | 67.92±0.23 | **68.02**±0.23 |
| *Yeast* | 46.36±0.12 | **46.48**±0.09 | 45.12±0.14 |
| *Enron* | **32.52**±0.11 | 32.21±0.08 | 30.39±0.2 |
| *Fingerprint* | **86.1**±0.06 | 85.52±0.05 | 85.79±0.02 |
| *Protein* | **37.33**±0.19 | 36.57±0.27 | 35.49±0.24 |
| *corel5k* | 49.68±0.5 | 48.66±0.44 | **52.65**±0.53 |
| *espgame* | 9.1±0.14 | 8.47±0.15 | **9.5**±0.07 |
| *iaprtc12* | 10.73±0.17 | **11.76**±0.15 | 10.48±0.06 |
| *mirflickr* | 5.38±0.39 | **5.98**±0.33 | 5.03±0.34 |
| *pascal07* | **28.3**±0.49 | 27.31±0.36 | 29.44±0.6 |
| *psortPos* | **89.17**±0.24 | 87.92±0.44 | 83.16±0.38 |
| *psortNeg* | **91.26**±0.21 | 89.94±0.21 | 86.51±0.25 |
| *plant* | **91.55**±0.14 | 89.94±0.11 | 71.78±0.21 |

Table 10: The Macro-F1 of algorithms over 13 datasets

In bioinfomatics datasets: *psortPos*, *psortNeg*, *plant*, the performance of kernel alignment algorithms are higher than the UniMKL. These numbers of the algorithm *AlignF_TT* are lower than *AlignF* and this gap is around 1%. In other datasets, the predictive performance of algorithms are quite similar and no clear outperformance is witnessed.

| | **AlignF** | **AlignF_TT** | **UniMKL** |
|---|---|---|---|
| *Emotions* | 68.33±0.15 | 68.25±0.13 | **68.44**±0.16 |
| *Yeast* | **65.44**±0.1 | 65.13±0.1 | 65.06±0.1 |
| *Enron* | 61.11±0.08 | **61.2**±0.07 | 60.14±0.04 |
| *Fingerprint* | **86.2**±0.05 | 85.61±0.05 | 85.9±0.02 |
| *Protein* | 58.43±0.14 | **58.72**±0.22 | 57.3±0.22 |
| *corel5k* | 52.2±0.39 | 50.31±0.34 | **53.8**±0.27 |
| *espgame* | **16.91**±0.32 | 15.93±0.2 | 16.89±0.21 |
| *iaprtc12* | 19.97±0.19 | **20.89**±0.24 | 19.74±0.12 |
| *mirflickr* | 7.03±0.46 | **7.38**±0.42 | 6.51±0.39 |
| *pascal07* | 33.02±0.54 | 31.81±0.42 | **33.74**±0.64 |
| *psortPos* | **89.67**±0.2 | 88.76±0.38 | 83.04±0.34 |
| *psortNeg* | **91.68**±0.19 | 90.44±0.19 | 86.83±0.27 |
| *plant* | **92.56**±0.15 | 90.96±0.12 | 75.0±0.17 |

Table 11: The Micro-F1 of algorithms over 13 datasets

**Performance of the *AlignF_TT* in different low-rank numbers** This section presents the performance of *AlignF_TT* algorithm in terms of changing the low-rank numbers $r$. The low-rank $r$ is from 1 to half the number of training sample.

Generally, when increasing the low-rank number, the performance of *AlignF_TT* algorithm also rises. Figures also show that the effective low-rank number is around 10% of data examples in order to get the comparable performance with the *AlignF* algorithm. For example, this observation is clearly recognized in the *Yeast* dataset at Figure 32.

The second observation is that the performance increase is not always proportional to the low-rank increasing. It happens in *Emotions*, *Protein*, and *espgame* datasets. In the *Emotions* dataset, all evaluation measurements are generally increasing in higher low-rank numbers but not in sometimes. For example, Figure 33 shows a gap of decreasing 0.4% accuracy when the rank increasing from 125 to 130.

In the higher ranks, it is also witnessed some performance improvement in around 1% better *AlignF* algorithm in the *Fingerprint*, *iaprtc12*, *mirflickr* data. For instance, Figure 34 presents that the AlignF_TT algorithm procedures a better accuracy and around 2% improvement in terms of Micro and Macro F1 than other algorithms in the *iaprtc12* dataset.

In *psortPos*, *psortNeg* and *plant* datasets, the AlignF algorithm outperforms than other algorithms in all criteria. In these datasets, the AlignF_TT returns a better performance than UniMKL algorithm but lower than the AlignF algorithm. The AlignF_TT performance gap varies from 0.5% to 1.5% decreasing from the AlignF algorithm. We can witness this observation in different low-rank numbers in the *psortPos* dataset at Figure 35.

The remaining figures for other datasets are presented in the *Appendix* section.

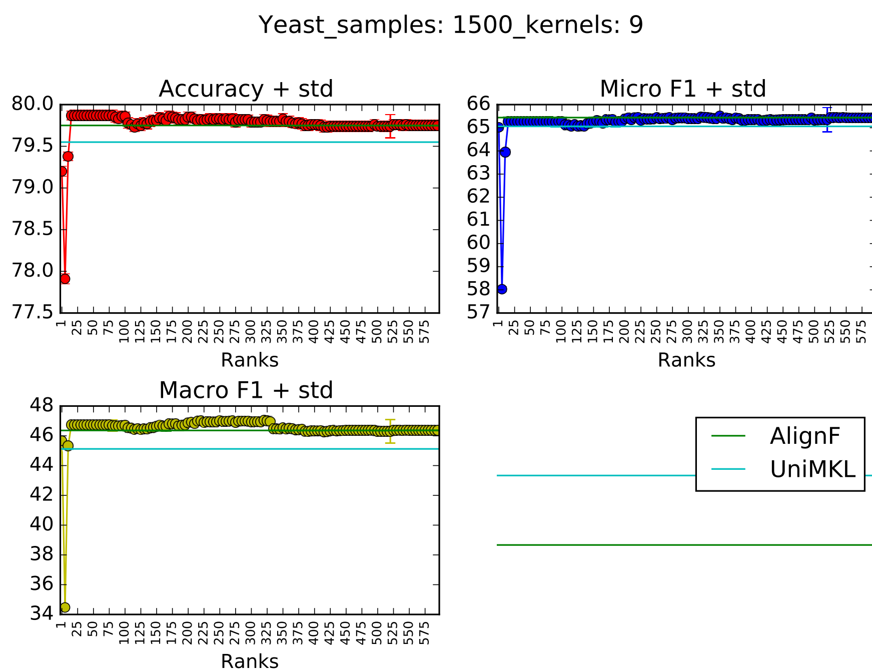



Figure 32: *AlignF_TT* performance in different low-rank numbers in *Yeast* data

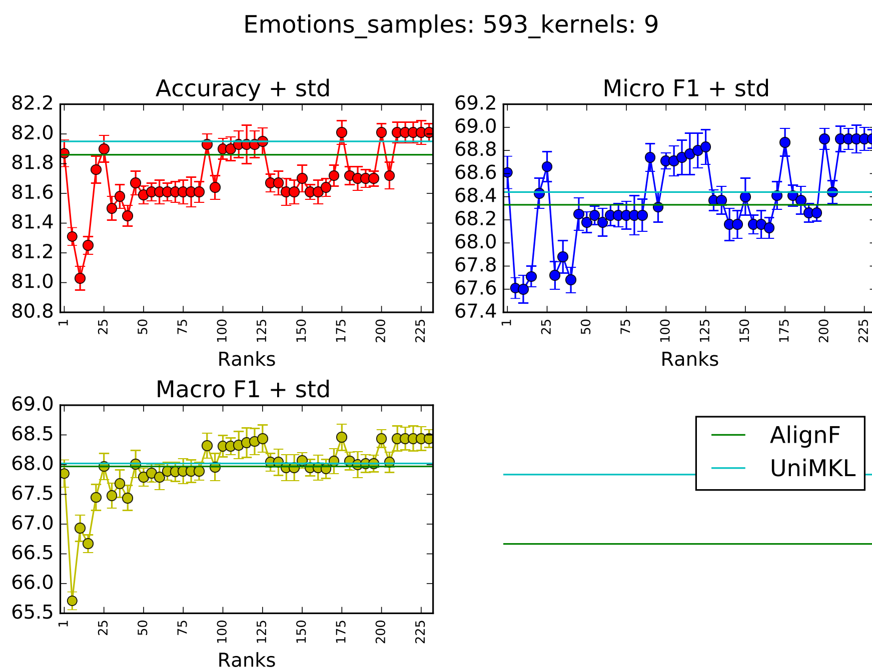



Figure 33: *AlignF_TT* performance in different low-rank numbers in *Emotions* data

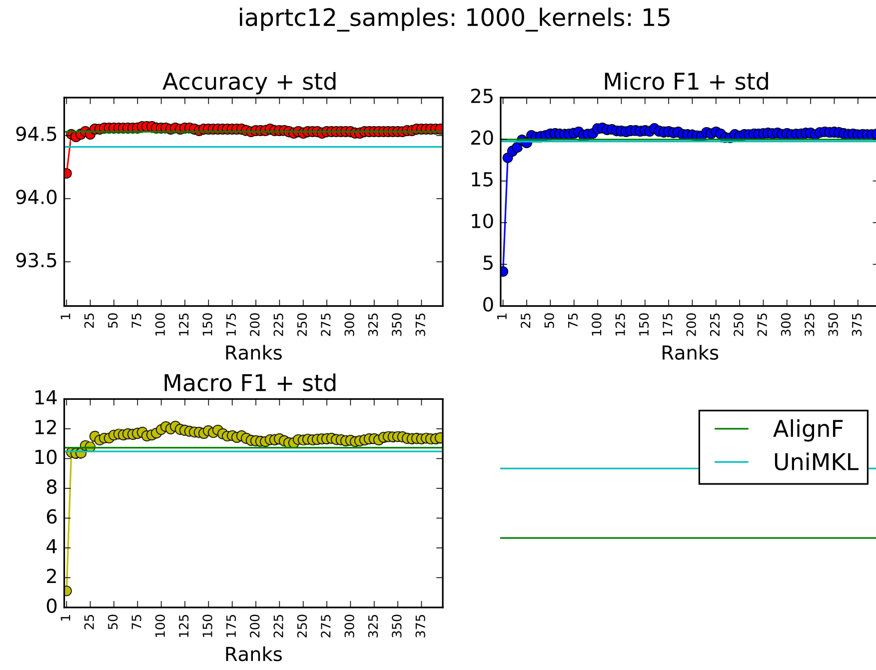



Figure 34: *AlignF_TT* performance in different low-rank numbers in *iaprtc12* data

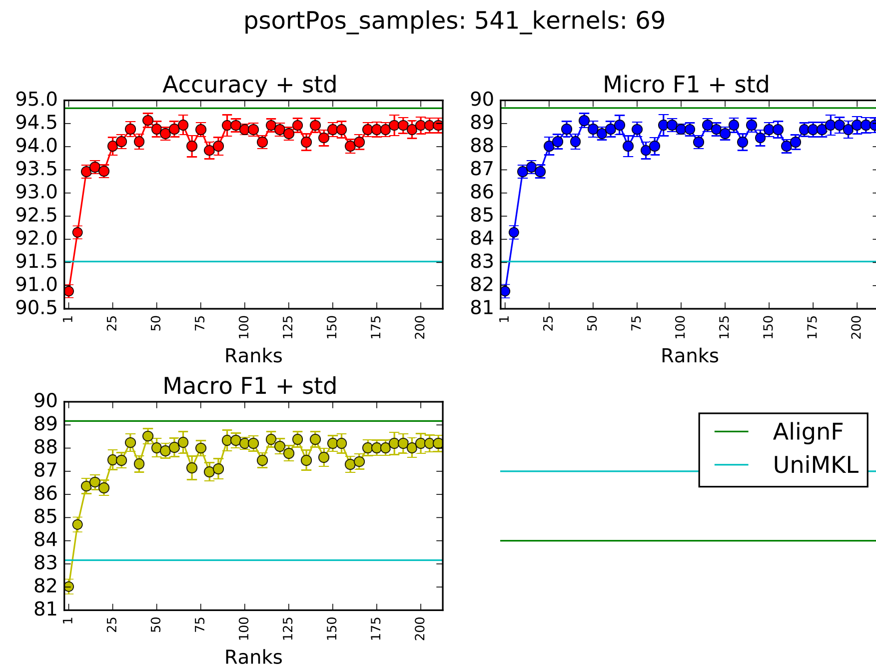



Figure 35: *AlignF_TT* performance in different low-rank numbers in *psortPos* data

**Running time in different low-rank numbers** The running time are reported in 4 datasets: *Emotions*, *Enron*, *psortPos*, *psortNeg*. The reason is *Emotions* contains the small number of samples and kernels whereas *psortNeg* has many of them. The *Enron* data stands for the case of many samples and small number of kernels while *psortPos* contains a lot of kernels and few samples.

The *pre-calculation time* includes the time for centering kernels of *AlignF*. In *AlignF_TT*, this number is the duration for decomposing the tensor of kernels and precomputed common parts of kernel matrices. The *calculation* time is remaining time to run steps to get the kernel alignment result. Moreover, the red and blue line in these figures describe the precalculation time and the total running time of the *AlignF* algorithm, respectively.

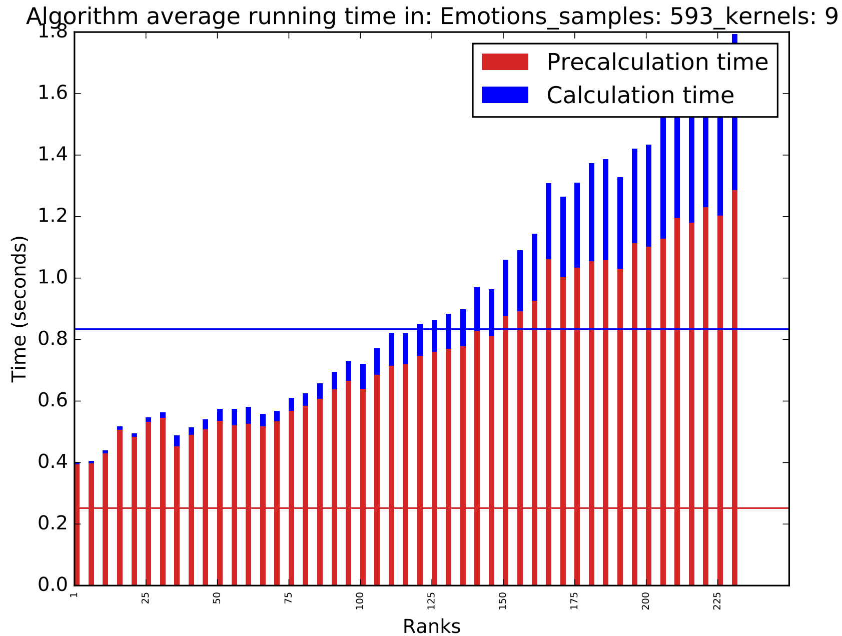



Figure 36: *AlignF_TT* running time in different low-rank numbers in *Emotions* data

Decomposing the tensor of kernels is one of the most costly steps. In *Enron* and *psortNeg* datasets, many data samples requires longer running time for the decomposition step. This decomposition time is even longer than the total running time of AlignF algorithm. For example, in Figure 37, the *AlignF* requires around 3 seconds to run but the decomposition of the tensor $\in \mathbb{R}^{[1702\times1702\times9]}$ needs more than 4 seconds to finish. On the other hand, Figure 36 shows that the *AlignF_TT* algorithm is faster than *AlignF* in terms of running time when calling it with a small number of low-rank components. This happens when decomposing a smaller tensor of kernels which size is $\mathbb{R}^{[593\times593\times6]}$ in the *Emotions* data.

Similarly, in *Emotions* and *Enron* datasets, when the number of kernels is small, the precalculation step always takes the most of the computation time. In these datasets, the calculation step also requires more time when the number of low-rank increasing.

For the *AlignF_TT* algorithm, when there are many kernels available, increasing

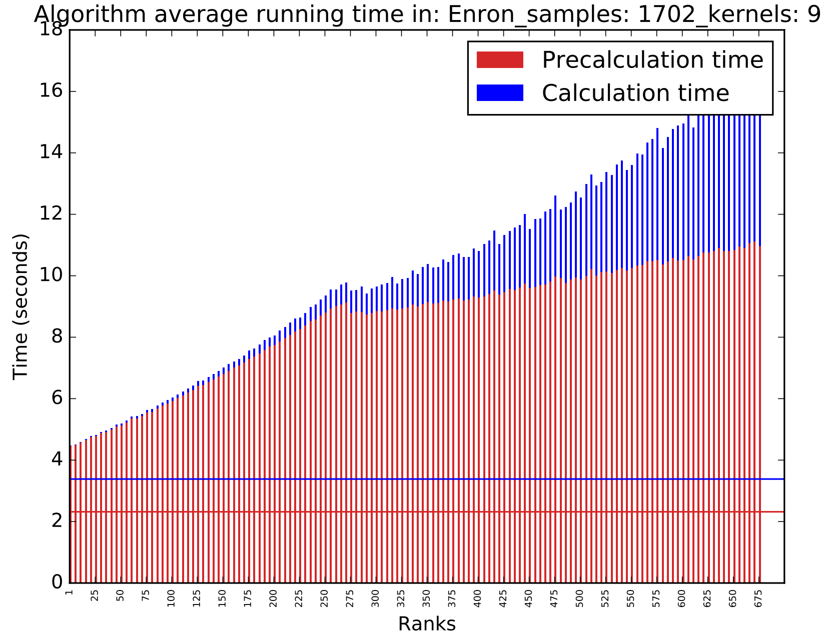



Figure 37: *AlignF_TT* running time in different low-rank numbers in *Enron* data

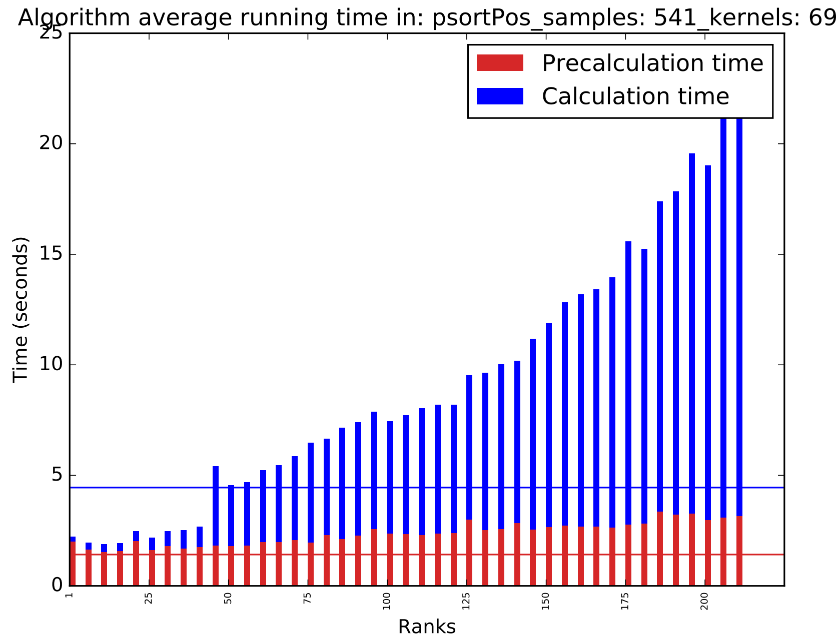



Figure 38: *AlignF_TT* running time in different low-rank numbers in *psortPos* data

the low-rank number leads to cost much more time to finish the calculation step. Figure 38, 39 show this observation in the *psortPos* and *psortNeg* data. In this case, the multiplication of low-rank matrices is not effective when the low-rank is relatively high. Nevertheless, when the low-rank number is less than 10% of training examples, the calculation step of the *AlignF_TT* is faster than the one of *AlignF* algorithm if the data size is small. This happens in the *psortPos Emotions* datasets with 541 and 593 samples, respectively (Figure 38, 36).

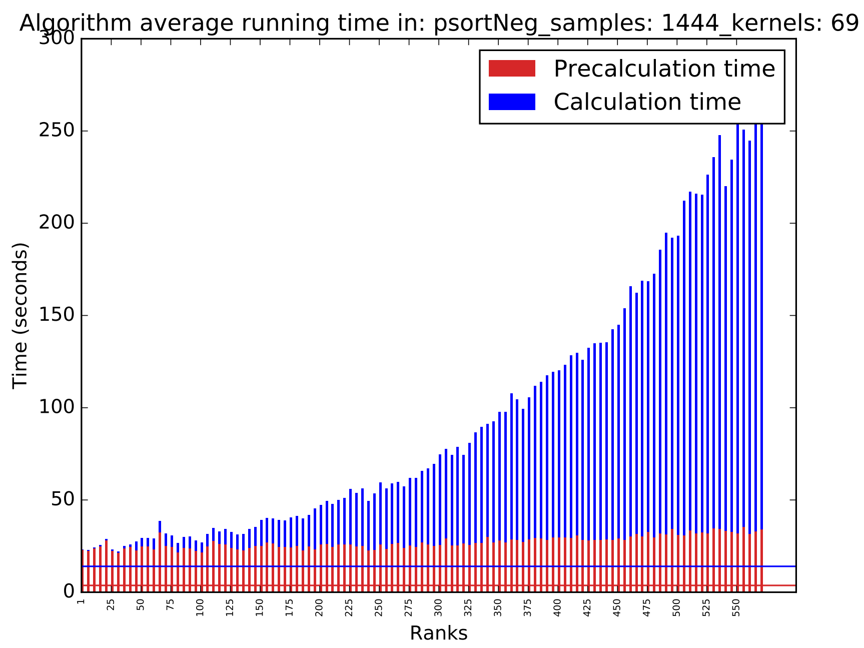



Figure 39: *AlignF_TT* running time in different low-rank numbers in *psortNeg* data

# 6 Discussion

This study set out to explore the tensor learning perspective in multiple kernel learning problems. An optimal decomposition for a tensor of kernels has been proposed after analysing different tensor decomposition algorithms. This decomposition returns a common left and right matrices for all kernels and each kernel specific matrix in the size $[r \times r]$ where $r$ is the number of low-rank components. These decomposed matrices have been used to derive a novel kernel alignment algorithm (AlignF_TT) which is based on the centre kernel alignment algorithm (AlignF).

Experiments show that the proposed algorithm performance is higher than AlignF in artificially constructed datasets and comparable in real-world datasets, regarding to the predictive accuracy. The experiment in artificially constructed datasets also suggests that the kernel alignment algorithms perform well when data samples are taken from a multimodal distribution. The second major finding is that the small number of low-rank components is *enough* for compressing a tensor of kernels and leading to the comparable performance in the AlignF_TT algorithm.

A limitation of the proposed algorithm is that its running time is not faster than the AlignF algorithm in datasets which contain the large samples size. The explanation is the high cost of calling SVD in the decomposition step. To reduce the computational complexity of this step, the Randomize Singluar Value Decomposition algorithm [51] can be used to open the possibility of running AlignF_TT with massive datasets.

Further works will focus on finding new applications for these proposed algorithms: the optimal decomposition for multiple kernels and the AlignF_TT algorithm. The AlignF_TT algorithm is easy to extend to other kernel learning problems; for example: pairwise kernel learning where the Kronecker product kernel is applied. In this case, all computational steps can be done without explicitly computing and storing the Kronecker product of kernels. It would be interesting to do the triple-wise kernel learning which models the relations among three objects which given by multiple kernels. For this problem, the use of the Kronecker product kernel is infeasible and only approximation algorithms can be applied; for example: the proposed algorithms.

# 7 Acknowledgement

This work is supported by the funding from Professor Juho Rousu and the Honours Programme in Computer Science department, Aalto University.

The calculations presented in this thesis were performed using computer resources within the Aalto University School of Science "Science-IT" project.

# References


[1] G. H. Golub and C. Reinsch. Singular value decomposition and least squares solutions. *Numer. Math.*, 14(5):403–420, April 1970.

[2] Henk AL Kiers. Towards a standardized notation and terminology in multiway analysis. *Journal of chemometrics*, 14(3):105–122, 2000.

[3] Ledyard R Tucker. Some mathematical notes on three-mode factor analysis. *Psychometrika*, 31(3):279–311, 1966.

[4] Ivan V Oseledets. Tensor-train decomposition. *SIAM Journal on Scientific Computing*, 33(5):2295–2317, 2011.

[5] Nicholas D Sidiropoulos, Lieven De Lathauwer, Xiao Fu, Kejun Huang, Evangelos E Papalexakis, and Christos Faloutsos. Tensor decomposition for signal processing and machine learning. *IEEE Transactions on Signal Processing*, 65(13):3551–3582, 2017.

[6] Mehmet Gönen and Ethem Alpaydın. Multiple kernel learning algorithms. *Journal of Machine Learning Research*, 12(Jul):2211–2268, 2011.

[7] Nello Cristianini and John Shawe-Taylor. *An introduction to support vector machines and other kernel-based learning methods*. Cambridge university press, 2000.

[8] Francis R Bach. Exploring large feature spaces with hierarchical multiple kernel learning. In *Advances in neural information processing systems*, pages 105–112, 2009.

[9] Corinna Cortes, Mehryar Mohri, and Afshin Rostamizadeh. Two-stage learning kernel algorithms. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 239–246, 2010.

[10] Tijl De Bie, Nello Cristianini, and Roman Rosipal. Eigenproblems in pattern recognition. In *Handbook of Geometric Computing*, pages 129–167. Springer, 2005.

[11] Gene H Golub and Charles F Van Loan. *Matrix computations*, volume 3. JHU Press, 2012.

[12] Carl Eckart and Gale Young. The approximation of one matrix by another of lower rank. *Psychometrika*, 1(3):211–218, 1936.

[13] David Skillicorn. *Understanding complex datasets: data mining with matrix decompositions*. CRC press, 2007.

[14] Daniel D Lee and H Sebastian Seung. Learning the parts of objects by non-negative matrix factorization. *Nature*, 401(6755):788–791, 1999.

[15] Tamara G Kolda and Brett W Bader. Tensor decompositions and applications. *SIAM review*, 51(3):455–500, 2009.

[16] Charles F Van Loan. The ubiquitous kronecker product. *Journal of computational and applied mathematics*, 123(1):85–100, 2000.

[17] J Douglas Carroll and Jih-Jie Chang. Analysis of individual differences in multidimensional scaling via an n-way generalization of "eckart-young" decomposition. *Psychometrika*, 35(3):283–319, 1970.

[18] Richard A Harshman. Foundations of the parafac procedure: Models and conditions for an" explanatory" multi-modal factor analysis. 1970.

[19] Ledyard R Tucker. Implications of factor analysis of three-way matrices for measurement of change. *Problems in measuring change*, 122137, 1963.

[20] Lieven De Lathauwer, Bart De Moor, and Joos Vandewalle. A multilinear singular value decomposition. *SIAM journal on Matrix Analysis and Applications*, 21(4):1253–1278, 2000.

[21] Vin De Silva and Lek-Heng Lim. Tensor rank and the ill-posedness of the best low-rank approximation problem. *SIAM Journal on Matrix Analysis and Applications*, 30(3):1084–1127, 2008.

[22] Andrzej Cichocki, Namgil Lee, Ivan Oseledets, Anh-Huy Phan, Qibin Zhao, Danilo P Mandic, et al. Tensor networks for dimensionality reduction and large-scale optimization: Part 1 low-rank tensor decompositions. *Foundations and Trends® in Machine Learning*, 9(4-5):249–429, 2016.

[23] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[24] John Shawe-Taylor and Nello Cristianini. *Kernel methods for pattern analysis.* Cambridge university press, 2004.

[25] Asa Ben-Hur and William Stafford Noble. Kernel methods for predicting protein–protein interactions. *Bioinformatics*, 21(suppl_1):i38–i46, 2005.

[26] Ben Taskar, Ming-Fai Wong, Pieter Abbeel, and Daphne Koller. Link prediction in relational data. In *Advances in neural information processing systems*, pages 659–666, 2004.

[27] Willem Waegeman, Tapio Pahikkala, Antti Airola, Tapio Salakoski, Michiel Stock, and Bernard De Baets. A kernel-based framework for learning graded relations from data. *IEEE Transactions on Fuzzy Systems*, 20(6):1090–1101, 2012.

[28] Bernhard Schölkopf, Koji Tsuda, and Jean-Philippe Vert. *Kernel methods in computational biology.* MIT press, 2004.

[29] Isaac Martín de Diego, Alberto Muñoz, and Javier M Moguerza. Methods for the combination of kernel matrices within a support vector framework. *Machine learning*, 78(1-2):137, 2010.

[30] Hiroaki Tanabe, Tu Bao Ho, Canh Hao Nguyen, and Saori Kawasaki. Simple but effective methods for combining kernels in computational biology. In *Research, Innovation and Vision for the Future, 2008. RIVF 2008. IEEE International Conference on*, pages 71–78. IEEE, 2008.

[31] Shibin Qiu and Terran Lane. A framework for multiple kernel support vector regression and its applications to sirna efficacy prediction. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 6(2):190–199, 2009.

[32] N ello Cristianini, Andre Elisseeff, John Shawe-Taylor, and Jaz Kandola. On kernel-target alignment. *Advances in neural information processing systems*, 2001.

[33] Gert RG Lanckriet, Nello Cristianini, Peter Bartlett, Laurent El Ghaoui, and Michael I Jordan. Learning the kernel matrix with semidefinite programming. *Journal of Machine learning research*, 5(Jan):27–72, 2004.

[34] Jaz Kandola, John Shawe-Taylor, and Nello Cristianini. Optimizing kernel alignment over combinations of kernel. 2002.

[35] Canh Hao Nguyen and Tu Bao Ho. An efficient kernel matrix evaluation measure. *Pattern Recognition*, 41(11):3366–3372, 2008.

[36] Junfeng He, Shih-Fu Chang, and Lexing Xie. Fast kernel learning for spatial pyramid matching. In *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*, pages 1–7. IEEE, 2008.

[37] Corinna Cortes, Mehryar Mohri, and Afshin Rostamizadeh. Algorithms for learning kernels based on centered alignment. *Journal of Machine Learning Research*, 13(Mar):795–828, 2012.

[38] Manik Varma and Debajyoti Ray. Learning the discriminative power-invariance trade-off. In *Computer Vision, 2007. ICCV 2007. IEEE 11th International Conference on*, pages 1–8. IEEE, 2007.

[39] Corinna Cortes, Mehryar Mohri, and Afshin Rostamizadeh. L 2 regularization for learning kernels. In *Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence*, pages 109–116. AUAI Press, 2009.

[40] Francis R Bach. Consistency of the group lasso and multiple kernel learning. *Journal of Machine Learning Research*, 9(Jun):1179–1225, 2008.

[41] Manik Varma and Bodla Rakesh Babu. More generality in efficient multiple kernel learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 1065–1072. ACM, 2009.

[42] Corinna Cortes, Mehryar Mohri, and Afshin Rostamizadeh. Learning non-linear combinations of kernels. In *Advances in neural information processing systems*, pages 396–404, 2009.

[43] Geoffrey McLachlan and David Peel. *Finite mixture models*. John Wiley & Sons, 2004.

[44] Konstantinos Trohidis, Grigorios Tsoumakas, George Kalliris, and Ioannis P Vlahavas. Multi-label classification of music into emotions. In *ISMIR*, volume 8, pages 325–330, 2008.

[45] André Elisseeff and Jason Weston. A kernel method for multi-labelled classification. In *Advances in neural information processing systems*, pages 681–687, 2002.

[46] Alexander Zien and Cheng Soon Ong. Multiclass multiple kernel learning. In *Proceedings of the 24th international conference on Machine learning*, pages 1191–1198. ACM, 2007.

[47] Huibin Shen, Sandor Szedmak, Céline Brouard, and Juho Rousu. Soft kernel target alignment for two-stage multiple kernel learning. In *International Conference on Discovery Science*, pages 427–441. Springer, 2016.

[48] Matthieu Guillaumin, Jakob Verbeek, and Cordelia Schmid. Multimodal semi-supervised learning for image classification. In *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, pages 902–909. IEEE, 2010.

[49] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12(Oct):2825–2830, 2011.

[50] Scikitlearn developers. Support Vector Machines — scikit-learn 0.18.2 documentation. http://scikit-learn.org/stable/modules/svm.html#multi-class-classification, 2017. [Online; accessed 23-July-2017].

[51] Nathan Halko, Per-Gunnar Martinsson, and Joel A Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM review*, 53(2):217–288, 2011.

# A Appendix

## A.1 The accuracy vs low-rank number

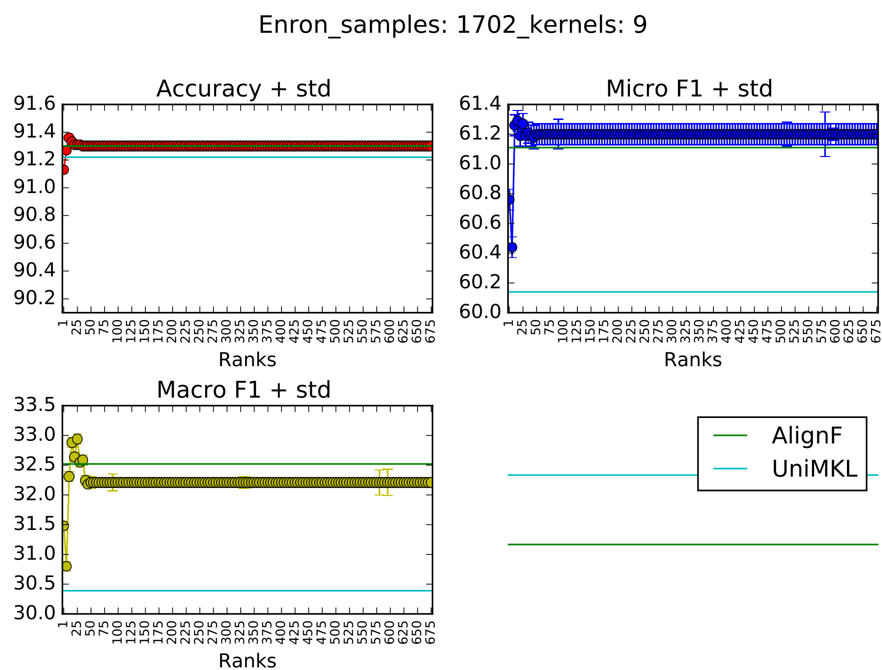



Figure A1: *AlignF_TT* performance in different low-rank numbers in *Enron* data

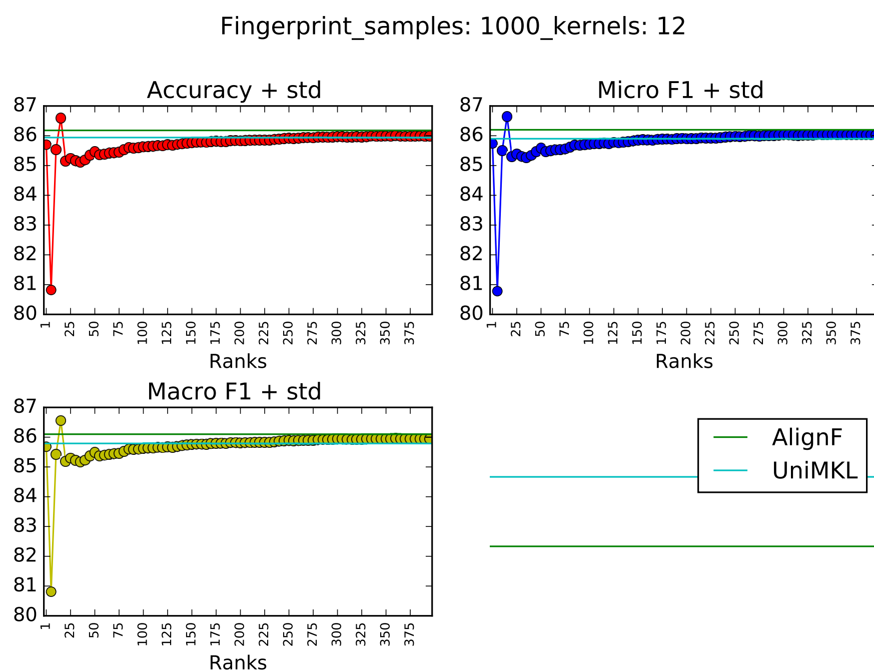



Figure A2: *AlignF_TT* performance in different low-rank numbers in *Fingerprint* data

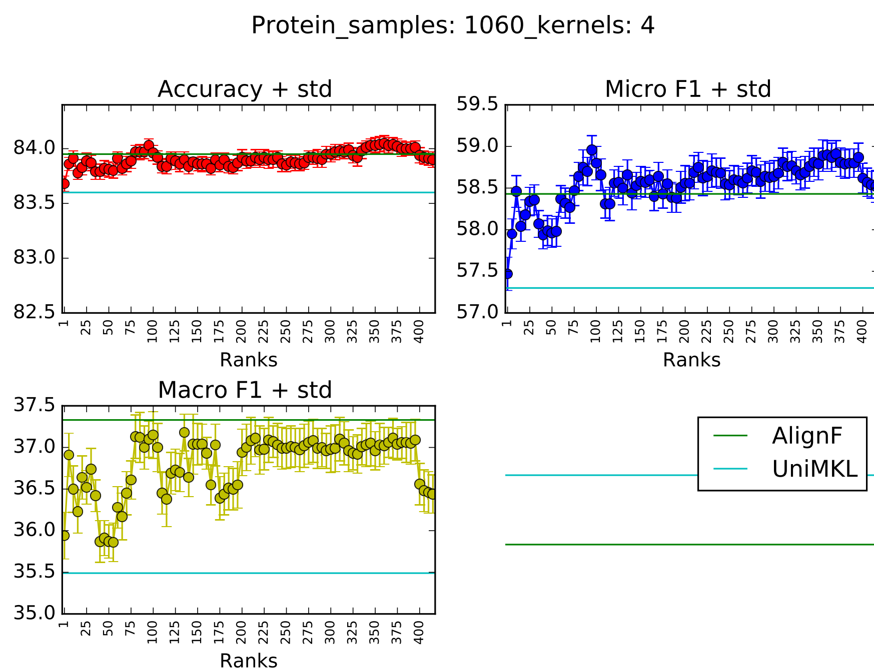



Figure A3: *AlignF_TT* performance in different low-rank numbers in *Protein* data

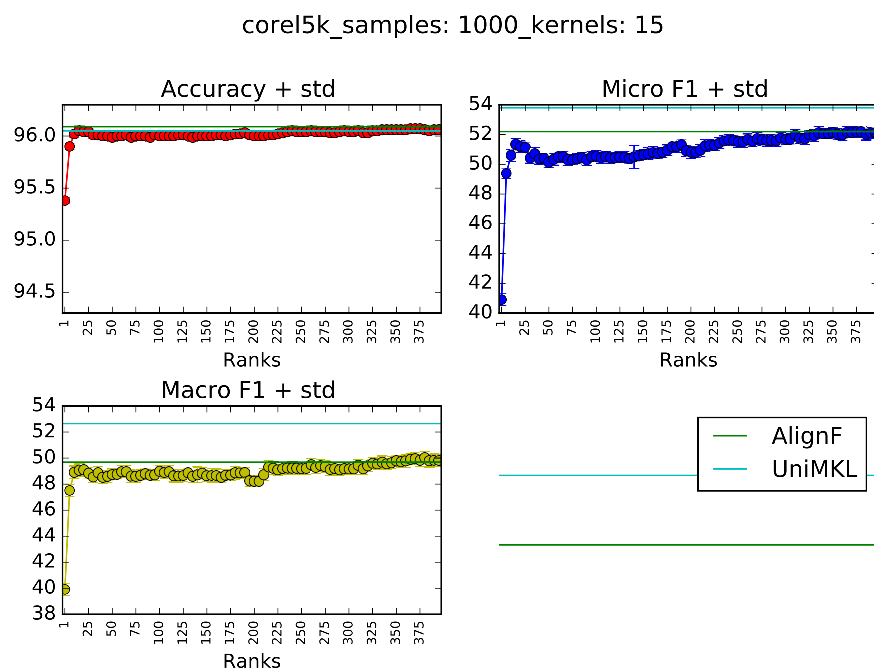



Figure A4: *AlignF_TT* performance in different low-rank numbers in *corel5k* data

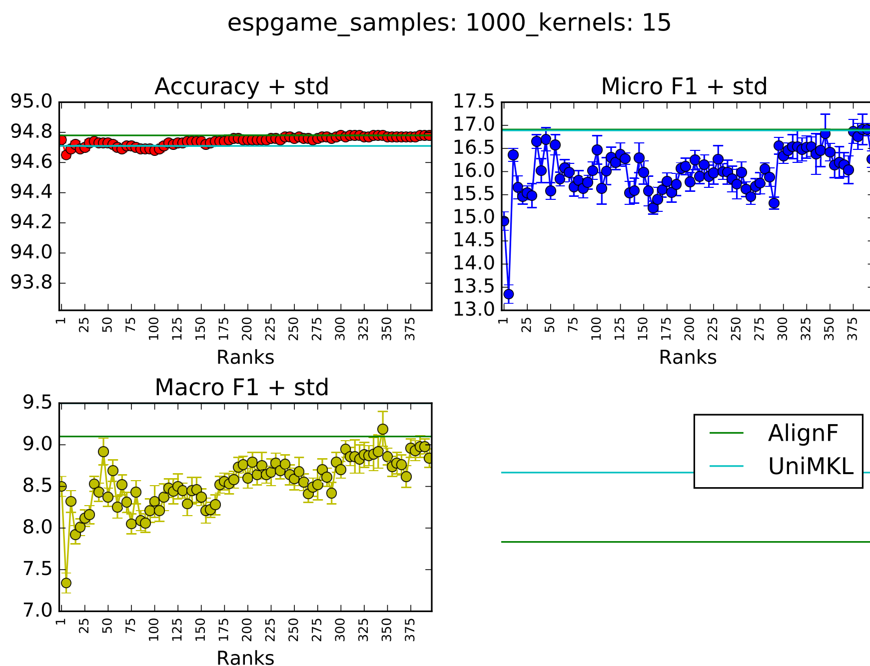



Figure A5: *AlignF_TT* performance in different low-rank numbers in *espgame* data

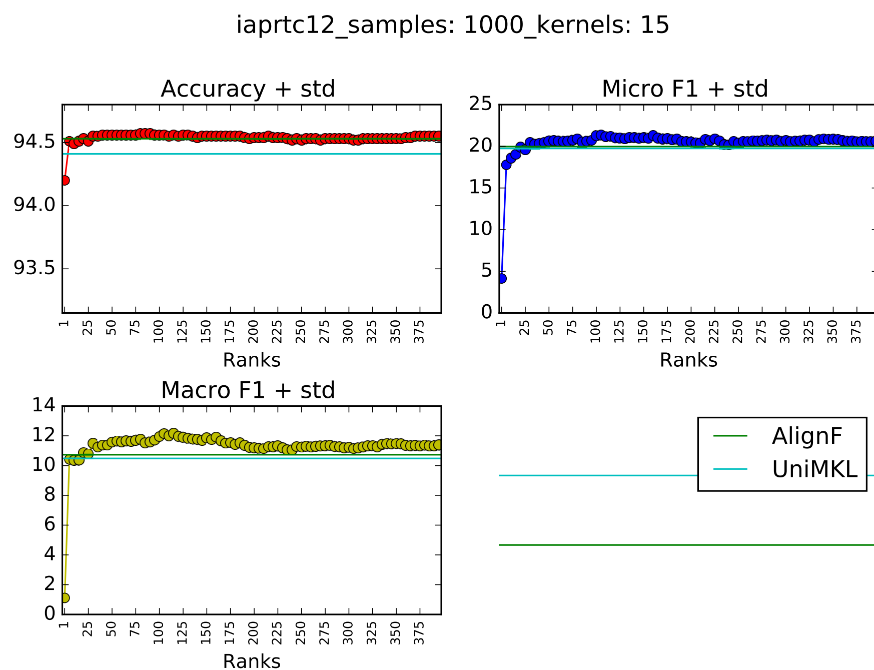



Figure A6: *AlignF_TT* performance in different low-rank numbers in *iaprtc12* data

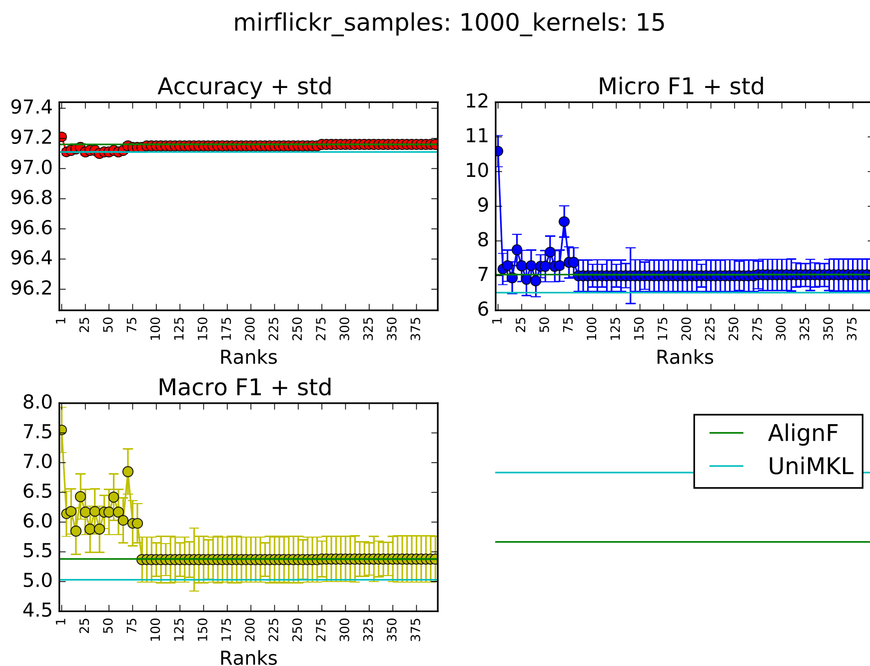



Figure A7: *AlignF_TT* performance in different low-rank numbers in *mirflickr* data

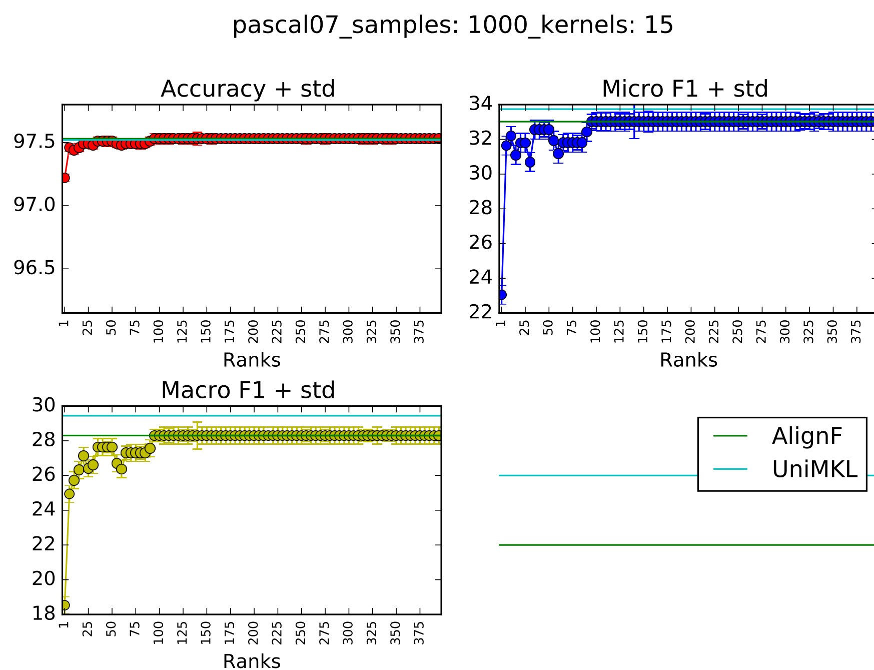



Figure A8: *AlignF_TT* performance in different low-rank numbers in *pascal07* data

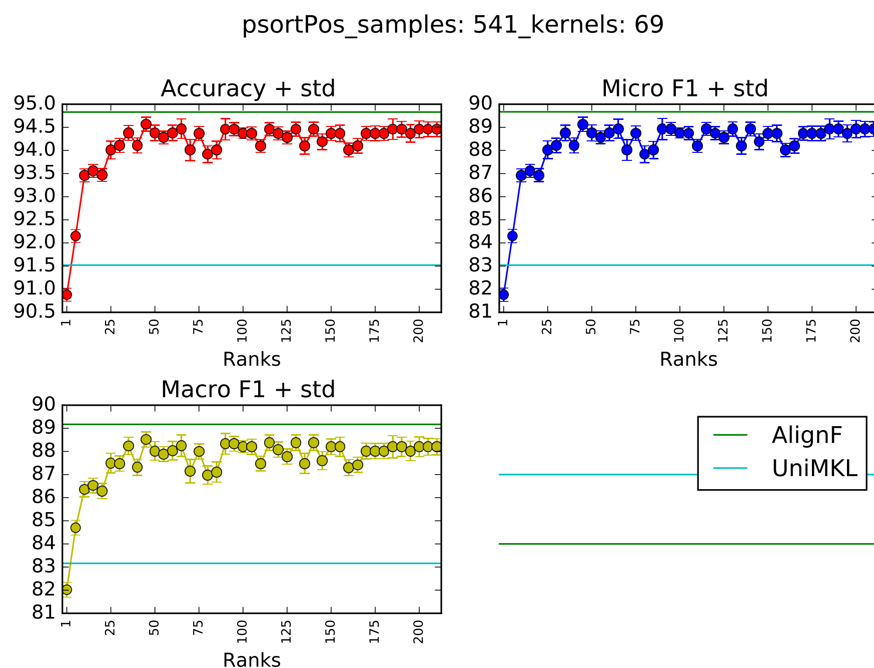



Figure A9: *AlignF_TT* performance in different low-rank numbers in *psortPos* data

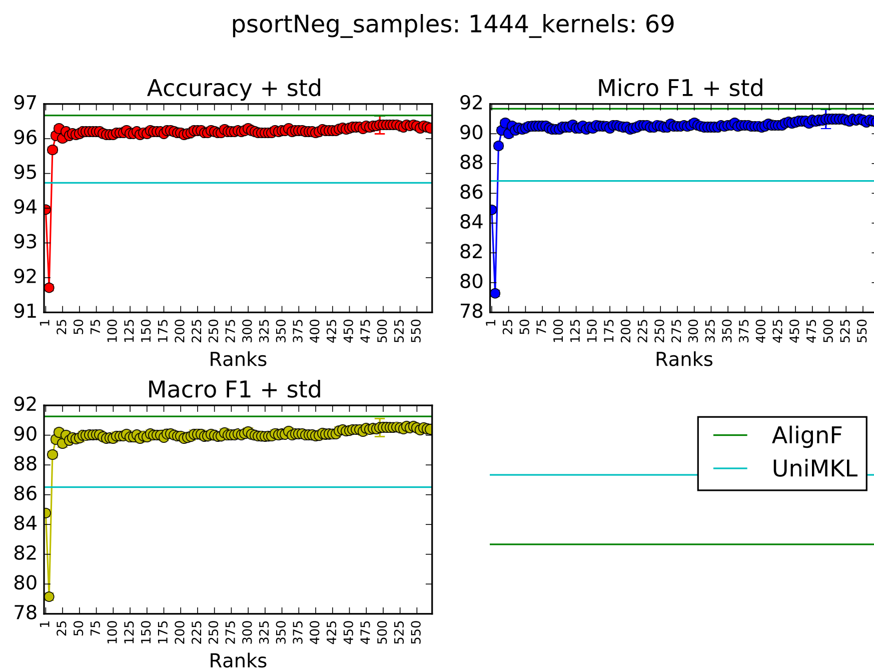



Figure A10: *AlignF_TT* performance in different low-rank numbers in *psortNeg* data

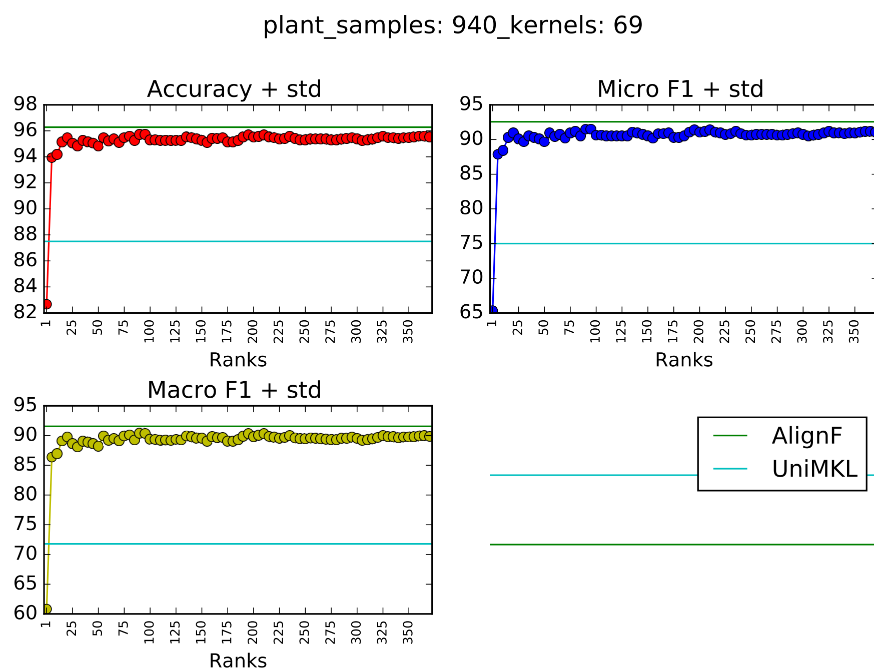



Figure A11: *AlignF_TT* performance in different low-rank numbers in *plant* data