Kernels from Random Partitions

Àlex Batlle Casellas, Aleix Torres i Camps

March 19, 2022

Abstract

We present the Random Partition Kernels, a family of kernels developed in [1] that come from random partitions. We explain the background for these kernels and present the methodology to construct them, and we propose novel approaches using Locality Sensitive Hashing and Markov processes. We show how this kernels offer an improvement for some algorithms in a variety of tasks.

1 Introduction

Kernel methods are a hit in machine learning. Among other things, their utility is to twist linear algorithms into strongly non-linear algorithms, giving them a powerful approach to data that has other kinds of structures that are non-linear. In general, they are also very useful in unsupervised learning tasks, since they do not need a target variable.

Despite there is large variety of kernels, in practice the most used ones are Linear, Polynomial, Periodic and, by far, the Radial Basis Funcion Kernel, which has been shown to give good results in a very large variety of tasks.

In this project we present Random Partition Kernels, a whole new family of kernels firstly introduced in [1]. They are based on random partitions, which have some reminiscence of clustering algorithms. In general terms, these kernels use the clustering information that we can give to them to have an idea of how similar two objects are.

First, we will give all the needed background to understand the whole project, or at least have an idea of what we are talking about. This includes Kernel Methods, Partitions, Distributions over partitions, Iterative methods and preconditioning and, finally, Locality Sensitive Hashing.

After that, we will present the main topic, which is Random Partition Kernels. We prove that they are indeed kernels, and we provide a way to compute an efficient and well-conditioned approximation.

Then, we will present examples of Random Partition Algorithms that can be used. The two first ones are presented in [1], while the next two are new and developed by us. From the latter two, the first one uses Markov Processes to bring up a new way of clustering metric data, and the second one uses Locality Sensitive Hashing, and it is a kernel thought for image datasets. Finally, we present a clustering algorithm for graphs to show a way to generate random partitions from a different type of data.

Finally, we test the kernels obtained from the first four algorithms. We use these kernels in Kernel PCA and in Kernel SVM algorithms to show they performance compared to the unkernelized versions. Additionally, we compare our new algorithms to the ones developed in the paper to see if the new ones offer any improvement.

At the end, we will present our conclusions and give some possible paths for future work.

2 Background

2.1 Reminder on Kernel Methods

To set the background for the project, we will give some definitions and results on kernel methods. Let $\mathcal{X}$ be the space where our data objects exist.

Definition 2.1.1 (Kernel)**.** A kernel is any function $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ such that:

(i) It is *symmetric*: $k(a, b) = k(b, a),\ \forall a, b \in \mathcal{X}$

(ii) It is *positive semi-definite*: for all $n \in \mathbb{N}$ and for all $\mathbf{x} := (x_1, \ldots, x_n) \in \mathcal{X}^n$, the Gram matrix $\mathbf{K}_{ij}[\mathbf{x}] := [k(x_i, x_j)]_{i,j\in\{1,\ldots,n\}}$ is a positive semi-definite matrix.

Definition 2.1.2 (Positive semi-definite matrix)**.** A matrix $A \in \mathcal{M}_{n\times n}(\mathbb{R})$ is positive semi-definite if for any $c \in \mathbb{R}^n \setminus \{\mathbf{0}\}$, the product $c^T Ac$ is non-negative.

Lemma 2.1.3. Let $A \in \mathcal{M}_{n\times n}$ be a symmetric matrix. The following are equivalent:

(i) A is a positive semi-definite matrix.

(ii) All of the eigenvalues of A are non-negative.

(iii) All leading principal minors of A are non-negative.

Lemma 2.1.3 will be useful to prove that a function is a kernel. The concept *kernel method* refers to any machine learning algorithm that has been transformed using the *kernel trick*, that is, expressing everything in terms of dot products. In this way, we can substitute the dot products at play with the kernel function, and this allows for the introduction of non-linearities and representations of the data objects that might make the task easier.

2.2 Partitions

We will now give a definition of the concept of a **partition** of an arbitrary set.

Definition 2.2.1 (Partition)**.** Given a set $\mathcal{D}$, a partition of $\mathcal{D}$ is a collection of subsets $\rho \subseteq 2^{\mathcal{D}}$ such that:

(i) $S \neq \varnothing\ \forall S \in \rho$

(ii) $\forall S, T \in \rho,\ S \cap T = \varnothing$

(iii) $\bigcup_{S\in\rho} S = \mathcal{D}$

Example 2.2.2. If $\mathcal{D} = \{a, b, c, d\}$, then the following two sets are partitions of $\mathcal{D}$:

$$\{\{a, b\}, \{c\}, \{d\}\}, \quad \{\{a\}, \{b\}, \{c\}, \{d\}\}$$

We will usually work with finite $\mathcal{D}$'s. In that context, if we know that $|\mathcal{D}| = N < +\infty$, then the number of partitions of $\mathcal{D}$ is known and it is the nth Bell number. The Bell numbers are calculated using the recursive formula

$$B_{n+1} = \sum_{k=0}^{n} \binom{n}{k} B_k,\ B_0 := 1$$

2.3 Distributions in the partition space

Just like with every other discrete space, we can define a probability distribution over the partition space (the set of all partitions) by associating each partition with a number between 0 and 1, such that all of this numbers add up to 1.

As this is quite impractical for big sets, we state that a distribution over the partition space can be obtained through any stochastic algorithm (as in, non-deterministic) that randomly generates partitions of a set [1]. When using a stochastic algorithm, we will not care about the underlying partition distribution. Partitions sampled from a partition distribution are called **random partitions** and are the basis for the next sections.

2.4 Iterative methods and preconditioning

The main issue with kernel methods is that to calculate the kernel matrix, we usually need $\mathcal{O}(N^2)$ steps, and each of these requires computing the kernel function on two data objects, which may be arbitrarily computationally expensive.

In some cases, though, kernelized machine learning algorithms can be solved using only the ability to compute Kv for arbitrary $v \in \mathbb{R}^n$. These type of solving methods are called *matrix-free*, and may ease computation time. For example, to solve SVMs or PCA, there exist these kind of methods.

The bounds on the number of iterations of matrix-free iterative solutions depend heavily on the condition number of the matrix, $\kappa(K)$: a low condition number leads to a numerically stable solution in a small number of iterations, whereas a large condition number may do just the opposite and maybe not even guarantee convergence of the solution. If we can construct a matrix B, such that we can efficiently evaluate Bv for arbitrary $v \in \mathbb{R}^N$ and such that $\kappa(BK) \ll \kappa(K)$, we can perform the iterative methods on this transformed system. B is known as a *pre-conditioning matrix*.

2.5 Locality Sensitive Hashing

Locality Sensitive Hashing is a type of hashing that uses functions such that similar objects have a high probability to go to the same cluster, and dissimilar objects have a low probability of ending up in the same cluster. In this project, we will use LSH adapted for gray scale images, in the following way:

A hash function, generated randomly, will be determined by a pixel and a threshold. For a particular hash function h, the output of an image will be 1 if the pixel associated with h has a higher or equal value than the threshold, and will return 0 otherwise. Since images usually have a large number of pixels, what we will do is concatenate k of these hash functions creating a big hash function that returns a k-tuple of 0's and 1's.

Notice that similar images have a high probability to output the same tuple, and therefore, go to the same cluster.

3 Random Partition Kernels

In this section we will define a kernel based on a given partition distribution. We will also worry about having a feasible, efficient and well conditioned implementation.

3.1 Random Partition Kernels

Random Partition kernels refer to a kernel defined over a partition distribution in the following way:

Definition 3.1.1 (Random Partition Kernel)**.** Let $\mathcal{P}$ be a given partition distribution, we will denote as $\rho \sim \mathcal{P}$ a sample of it, and for a given object $a \in \mathcal{X}$, $\rho(a)$ will denote the cluster where a belongs according to ρ. Now, for $a, b \in \mathcal{X}$, we define the random partition kernel $k_{\mathcal{P}}$ induced by $\mathcal{P}$ as

$$k_{\mathcal{P}}(a, b) = \mathbb{E}_{\rho \sim \mathcal{P}}[\mathbb{I}[\rho(a) = \rho(b)]]$$

where $\mathbb{I}$ is the indicator function and the expectation is over the partition distribution.

Therefore, the random partition kernel is the probability that two given objects end up in the same cluster according to the partition distribution $\mathcal{P}$. Before doing anything with $k_{\mathcal{P}}$, we have to check that it is indeed a kernel.

Proposition 3.1.2. $k_{\mathcal{P}}$ constitutes a valid kernel.

Proof. To see that $k_{\mathcal{P}}$ is a kernel, we will define k_ρ, for a $\rho \sim \mathcal{P}$, as

$$k_\rho(a, b) = \mathbb{I}[\rho(a) = \rho(b)].$$

By the Central Limit Theorem, we can decompose the expectation into a limit of the average of n samples, where n tends to infinity, in the following way:

$$k_{\mathcal{P}}(a, b) = \mathbb{E}_{\rho \sim \mathcal{P}}[\mathbb{I}[\rho(a) = \rho(b)]] =$$

$$= \lim_{n\to\infty} \frac{1}{n} \sum_{\rho\sim\mathcal{P}}^{n} \mathbb{I}[\rho(a) = \rho(b)] = \lim_{n\to\infty} \frac{1}{n} \sum_{\rho\sim\mathcal{P}}^{n} k_\rho(a, b)$$

Now, using that a linear combination of kernels with positive coefficients is a valid kernel and that a convergent limit of kernels is a kernel, it suffices to show that k_ρ is a valid kernel for any $\rho \sim \mathcal{P}$.

For any dataset of size N, let K be the $N \times N$ kernel matrix of k_ρ. This will contain 1's only in the positions i, j such that $\rho(i) = \rho(j)$, these are, the positions such that the ith and jth element are in the same subset in the partition given by ρ.

Then, K can be permuted into a block diagonal matrix of the following form:

$$ZKZ^T = \begin{pmatrix} \mathbf{1} & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{0} & \mathbf{1} & \dots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \dots & \mathbf{1} \end{pmatrix}$$

where $\mathbf{1}$ are matrices with all entries equal to 1 that can be of different sizes, and similarly $\mathbf{0}$ are matrices with all entries 0. Z is the permutation matrix that groups together all the columns of the same subset in ρ.

Now, using that ZKZ^T is a block matrix of PSD matrices, we can conclude that ZKZ^T is PSD. Moreover, since a permutation does not affect the eigenvalues of a matrix, K is PSD. Therefore, since K is PSD for any dataset, k_ρ is a valid kernel.

□

Notice that this kernel computes the affinity between two elements as the probability that they end up forming part of the same cluster according to $\mathcal{P}$. In practice, this can be very useful, because if we have a good partition distribution of our dataset, this will define a good kernel because it will contain information about the separability of our dataset.

3.2 m-approximate Random Partition Kernel

As we already mentioned, in general, we will not have the partition distribution itself, and only a generator of random partitions. Fortunately, we only need this to achieve a simple approximation scheme using the following definition.

Definition 3.2.1. The m-approximate Random Kernel $\tilde{k}_\mathcal{P}$ is the proportion of times that the partition distribution $\mathcal{P}$ assigns a and b to the same subset, over m samples, i.e.,

$$k_\mathcal{P}(a, b) \approx \tilde{k}_\mathcal{P}(a, b) = \frac{1}{m} \sum_{\rho\sim\mathcal{P}}^{m} k_\rho(a, b)$$

In other words, for practical uses, we substitute the expectation for its estimator. This is a "good" estimator, in the sense that it has low variance when we increase m, according to the following lemma:

Lemma 3.2.2. Let $\mathcal{P}$ be a partition distribution. If we have independent samples from it, the bound on the variance of the estimator $\tilde{k}_\mathcal{P}$ is $O(\frac{1}{m})$.

Proof. Since $\rho \sim \mathcal{P}$ are independent random partitions, they satisfy that

$$k_\rho(a, b) \sim \text{Bernoulli}(k_\mathcal{P}(a, b)).$$

Then, since $\tilde{k}_\mathcal{P}$ is the maximum likelihood estimator for $k_\mathcal{P}$ (because of $k_\rho(a, b)$ following a Bernoulli), its variance is bounded by $\frac{1}{4m}$.

□

In conclusion, we can ensure that the m-approximate Random Partition Kernel $\tilde{k}_\mathcal{P}$ is a feasible approximation of the Random Partition Kernel $k_\mathcal{P}$.

3.3 Efficient evaluation and well-conditioning

In this section we will focus in improving the efficiency of kernel algorithms and in decreasing

the condition number of our kernel matrix, since it is a good practice as we stated in section 2.4.

As we said, some classes of kernel algorithms can be solved using only the ability to compute Kv, where K is the kernel matrix and v is an arbitrary vector. In our case, if K is the kernel matrix of an arbitrary $\rho \sim \mathcal{P}$ and N the size of the dataset, K can be stored in N space, and using that

$$(Kv)_i = \sum_{j \in \rho(i)} v_j$$

the multiplication Kv can be compute in $2N$ operations.

Using the m-approximate Random Partition Kernel, we would require mN space and $2mN$ operations for a matrix-vector product. Finally, if we use an iterative algorithm, the overall complexity will be of $2mNI$ operations. In conclusion, we achieve a matrix-vector product of linear complexity with respect to the size of the dataset (since $m, I \ll N$) which is important in terms of Big Data.

Now, to solve the conditioning problem that the matrix could have, [1] proposes the following pre-condition matrix:

$$B = m \sum_{\rho \in \mathcal{P}}^{m} (K_\rho + \sigma \operatorname{Id})^{-1}$$

where σ is a small constant to ensure that the matrix is non-singular. Due to the next lemma, we can compute $(K_\rho + \sigma \operatorname{Id})^{-1}$ in only $2N$ operations and Bv in $2mN$ operations.

Lemma 3.3.1. Using the same notation, the product $(K_\rho + \sigma \operatorname{Id})^{-1} v$ can be computed in the following way:

$$((K_\rho + \sigma \operatorname{Id})^{-1} v)_i = \frac{1}{\sigma} v_i - \frac{1}{\sigma(|\rho(i)| + \sigma)} \sum_{j \in \rho(i)} v_j$$

Proof. Let w be the vector with w_i equal to the right hand side of the equation and $L = K_\rho + \sigma \operatorname{Id}$. We will show that $L(L^{-1}v) = Lw$, then we will have that $L^{-1}L(L^{-1}v) = L^{-1}Lw$, simplifying, $L^{-1}v = w$, which is the equation from the lemma. Therefore, it suffices to show that $v = Lw$.

Notice that the ith row of $L = K_\rho + \sigma \operatorname{Id}$ is full of 0's except from those positions j such that $j \in \varphi(i)$, where there is a 1, except for the ith position, where there is a $1 + \sigma$. Then, as we know that the ith position in a matrix-vector product results from computing the dot product of the ith row of the matrix and the vector, we have that:

$$((K_\rho + \sigma \operatorname{Id})w)_i = (1+\sigma)w_i + \sum_{j \in \rho(i) \setminus i} w_j =$$

$$= v_i - \frac{\sigma \sum_{j \in \rho(i)} v_j}{\sigma(|\rho(i)| + \sigma)} + \frac{\sum_{j \in \rho(i)} v_j}{\sigma} + \frac{|\rho(i)| \sum_{j \in \rho(i)} v_j}{\sigma(|\rho(i)| + \sigma)}$$

$$= v_i + \left(\frac{1}{\sigma} - \frac{\sigma}{\sigma(|\rho(i)| + \sigma)} - \frac{|\rho(i)|}{\sigma(|\rho(i)| + \sigma)} \right) \sum_{j \in \rho(i)} v_j$$

$$= v_i + 0 \sum_{j \in \rho(i)} v_j = v_i$$

Then $Lw = v$, which was what we had left to prove the lemma.

□

Although we are adding an overhead in the number of operations, we ensure that the condition number decreases, and therefore, we obtain a big reduction in the number of iterations of the iterative solver.

4 Random Partition Algorithms

In this section, we will explain some algorithms that create random partitions out of data. We will first revisit the ones developed in [1], and then we will expand the list by bringing into play other algorithms. These algorithms, when run, create a random partition, which may then be used in creating the Random Partition Kernel matrix.

4.1 Random Forest Partition Sampling Algorithm

The Random Forest Partition Sampling Algorithm is an algorithm that takes advantage of Random Forest models to create clusters of data. It does so by first creating just one tree T from the Random Forest model and sampling a value, d, from a uniform distribution. Then, for every data object a we have in $\mathcal{D}$, it computes in what leaf node of T does a end up. Then, it computes the ancestor of that node at height d, and that ancestor will represent the cluster that a is assigned to. In the end, this procedure outputs a random partition.

Algorithm 1 Random Forest Partition (RFP)

Input: $\mathcal{D} \subseteq \mathcal{X},\ y \in \mathbb{R}^N$
$T \sim \text{RandomForestTree}(\mathcal{D}, y)$
$d \sim \text{DiscreteUniform}(0, \text{height}(T))$
for $a \in \mathcal{D}$ **do**
    $\text{leafnode} = T(a)$
    $\rho(a) = \text{ancestor}(\text{leafnode}, d)$
end for
Output: ρ

The randomness in the partitions outputted by this algorithm comes both from the height d where the tree is "cut" to assign the object a to a certain cluster, and from the Random Forest tree creation algorithm, which samples randomly at each node from a pool of predictors and uses the best one in terms of Gini gain.

[1] does not present a method to compute the kernel matrix between test and train, so we propose the following: first, split the data into training and testing datasets. Then, create the clusters with the training data as the algorithm shows. After that, compute the leaf node of an object in test, and similarly as with the others, compute the corresponding ancestor that gives the corresponding cluster. This gives the cluster that the test object would belong to according to the training data.

4.2 Random Clustering Partition Algorithm

The Random Clustering Partition Algorithm creates random clusters of data by doing the following: it first selects randomly some dimensions of the data, then it samples a random number of cluster representatives (or *centers*). With this variables set, it then assigns each data object to the cluster of the closest center, where the distance from a point to a center is calculated using only the dimensions that were randomly selected at the start of the algorithm.

Algorithm 2 Random Clustering Partition (RCP)

Input: $\mathcal{D} \subseteq \mathcal{X},\ h \in \mathbb{N}\ (h \ll \log N)$
$d \sim \text{Sample}(\text{Bernoulli}(0.5), \dim \mathcal{X})$
$s \sim \text{DiscreteUniform}(0, h)$
$\mathcal{C} \sim \text{Sample}(\text{DiscreteUniform}(1, N), 2^s)$
for $a \in \mathcal{D}$ **do**
    $\rho(a) = \arg\min_{c \in \mathcal{C}} \left(\|(a_c - a) \odot d\|\right)$
end for
Output: ρ

This algorithm is quite reminiscent of the k-Means clustering algorithm, which could also be used, but would be a little slower because of the centroid computation it performs at each step. The algorithm is also similar to k-Nearest Neighbours with $k = 1$, although only with the centers and with some randomness added on the initialization of clusters.

We also thought this algorithm could be expanded on by adding a final piece of probabilistic re-assignment of clusters within the for loop: this would probabilistically reassign points to their actual closest center *using all variables*, and not just a random subset of them. It would obviously require another parameter, $p \in (0, 1)$.

As it also happens with the RFP algorithm, [1] does not present a method to compute the kernel matrix between test and train, so we propose the following: first split the data into training and testing datasets. Then, create the clusters with the train data as the algorithms shows. After that, for each object in the testing data,

select a corresponding cluster by using either the minimum distance to the centers or the average distance to members of a particular cluster.

4.3 Markov Dissemination Process

The Markov Dissemination Process is a clustering algorithm for general metric data, that is, data for which there is a distance function $d : \mathcal{X} \times \mathcal{X} \to \mathbb{R}^+ \cup \{0\}$ defined. We developed this algorithm to generate random partitions by making each data point follow a Markov process in the following way:

Algorithm 3 Markov Dissemination Process (MDP)

Input: $\mathcal{D} \subseteq \mathcal{X}, p \in (0,1), \gamma \in \mathbb{R}^-, T \in \mathbb{Z}^+$
for $a \in \mathcal{D}$ **do**
  $\rho(a) \leftarrow 0$
end for
$\mathcal{C} \leftarrow \{0\}$
for $t \in [0, \dots, T-1]$ **do**
  $q \leftarrow 0.5 * \exp(\gamma t)$
  for $a \in \mathcal{D}$ **do**
    $\pi_1 \sim \text{Bernoulli}(1-q)$
    if not π_1 **then**
      $\pi_2 \sim \text{Bernoulli}(p)$
      if π_2 **then**
        $\rho(a) \leftarrow \arg\min_c(\text{avg}_b(d(a,b)))$
      else
        $c \leftarrow 1 + \max(\mathcal{C})$
        $\rho(a) \leftarrow c$
        $\mathcal{C} \leftarrow \mathcal{C} \cup \{c\}$
      end if
    end if
  end for
end for
Output: ρ

This algorithm takes into account previous knowledge about the data, namely, how the distance (the dissimilarity) between two objects should be calculated. Depending on the values of γ, p and T, it will produce different clusterings, and even fixing γ, p and T produces different results. Note that this algorithm is quadratic on N, i.e., it runs in $\mathcal{O}(TN^2)$ time.

It works by first, considering that all points form a cluster, and iteratively allowing them to escape their cluster and go to a closer one *on average*, or to form another one by themselves, or to stay in their current cluster.

It can produce partitions as one seen in Figure 1, obtained by using $\gamma = -0.02, p = 0.999, T = 200$ and $\mathcal{D}$ a randomly generated dataset of two-dimensional points using two bivariate Gaussian distributions.

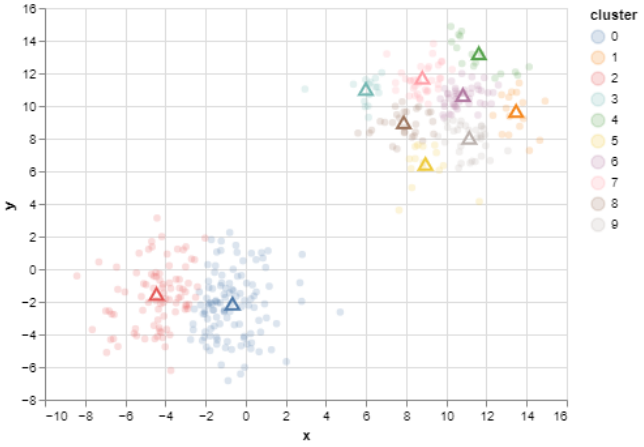

Figure 1: Markov Dissemination Process example.

4.4 LSH Partitioning

Until now, we have explained general algorithms to generate random partitions that can be used in most problems and types of data. But, we do not need to do that always, we can also focus on a single type of problem and try to find an algorithm that generates a partition distribution that may fit better.

This is what we will explain in this section: we have developed an algorithm based on Locality Sensitive Hashing (LSH) that partitions an image dataset, such that similar images have a larger probability to be on the same cluster than unlike ones. The algorithm is the following:

Algorithm 4 LSH Image Partition (LSHIP)

Input: $\mathcal{D} \subseteq \mathcal{X}, (\text{min\_k}, \text{max\_k}) \in \mathbb{N}^2$
$k \sim \text{DiscreteUniform}(\text{min\_k}, \text{max\_k})$
$\mathcal{H} \sim \text{Sample}(\text{DiscreteUniform}(0, (d-1) \times N), k)$
for $\text{im} \in \mathcal{D}$ **do**
    $\rho(\text{im}) = \text{hash}(\mathcal{H}, \text{im})$
end for
Output: ρ

The algorithm takes a image dataset of (presumptive) square grey-scale images and a range for k. Then, it initializes k uniformly within the given range, and generates the hash function $\mathcal{H}$ randomly. After that, it hashes each of the images, and the cluster labels are defined using the buckets the image go into.

The same algorithm can have obvious variations to increase its flexibility. For example, we could add the ability to handle multiple channel images, such as RGB or CMYK. Or, if we want to track larger shapes we may use convolutional hashes and not only hashes that operate pixel-wise. Additionally, we could add regularization with the purpose to have more collisions and less clusters. In the end, we showed this algorithm because we think it is the simplest of its kind.

In this case, it would be easy to compute the kernel matrix between training and testing data, because we would just have to hash the testing images the same way as the training ones and assign them to the corresponding clusters.

4.5 Markov Clustering Algorithm

The Markov Clustering Algorithm, developed in [2], is a clustering algorithm for graphs. It makes use of random walks on graphs to simulate a flow between nodes. It works by performing the usual power iterations of the transition matrix (an operator that is called *expansion* in the development of this algorithm), but it interleaves an intermediate operation called *inflation*, which increments the importance of already important edges and reduces the importance of less important ones. It does so by, for each column of the transition matrix (which is a stochastic matrix), raising its elements to a positive power, and then normalizing the said column to make it sum 1. This operation obviously keeps stochasticity.

Although it is a very promising algorithm, we will not use this one for the experimentation section for two main reasons: first of all, because we have pretty much nothing to compare it to, as all the other algorithms we have presented are valid for many data types, but have to be semi-adapted for graphs. The second reason and the most important one is that the algorithm is quite slow, it takes time in $\mathcal{O}(N^3)$, and it does not even have guaranteed convergence anyway.

5 Experimentation

The main objective of this section is to compare the two novel random partition kernels (MDP and LSHIP) with the random partitions proposed in [1] (RFP and RCP). Additionally, we will also compare the overall performance of the Random Partition Kernels using a benchmark algorithm. So we prepare the following two parallel experiments:

The first one consists in testing the MDP algorithm. We will work on the `bodyfat` dataset. In the first instance, we will apply Kernel PCA, and then Kernel SVM for regression. As we said, we will use the Random Partition Kernel with the three partition algorithms (all but LSHIP), and additionally, with the Linear Kernel. i.e. standard PCA and SVM, to have some grounds to compare performance.

The second one is similar to the previous one, but now it consists in testing the LSHIP algorithm, from which we will work on the `MNIST` dataset, which is, in fact, much harder than the first one because it has a larger number of variables. As before, first we will apply Kernel PCA, and then Kernel SVM, but now, for multi-class classification. As before, we will compare the results with the unkernelized version of the algorithms.

Globally, we will compare the Random Par-

tition Kernel with the unkernelized version of the same algorithm, and try to show its advantages. In particular, we will not only use Kernel PCA, which is an algorithm that only requires the kernel matrix, but we will also use Kernel SVM which, as many other predictor algorithms, needs the kernel similarities computed between the train and test datasets.

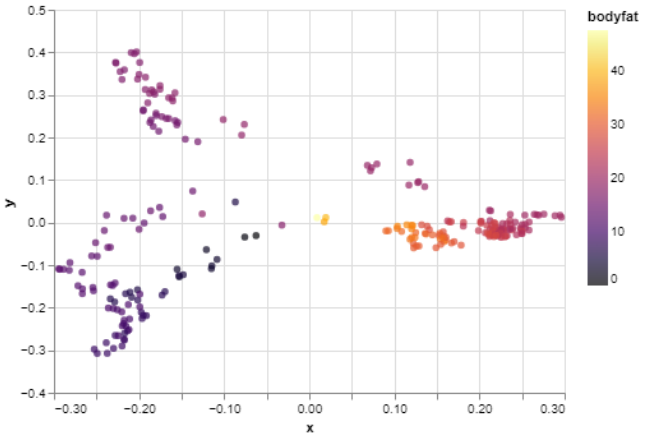

Figure 3: RFP Kernel-PCA on `bodyfat`.

5.1 Kernel PCA for the `bodyfat` dataset

To begin with, we apply Kernel PCA to the `bodyfat` dataset to obtain a dimensionality reduction into two variables. In the following charts, we can see the dimensionality reduction using the described kernels. The points are colored with a gradient to visualize if these kernels distinguish the different levels of the variable "bodyfat".

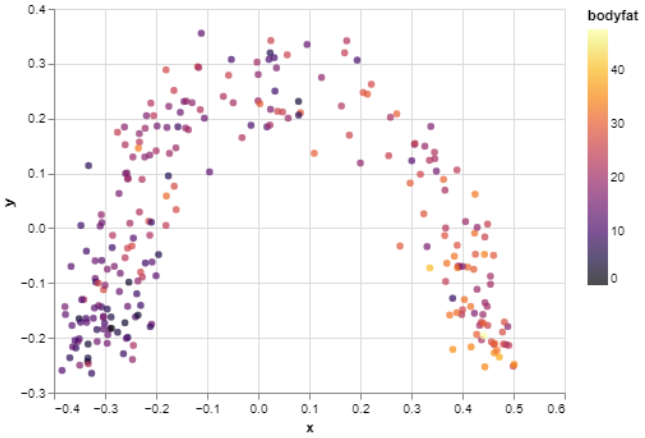

Figure 4: MDP Kernel-PCA on `bodyfat`.

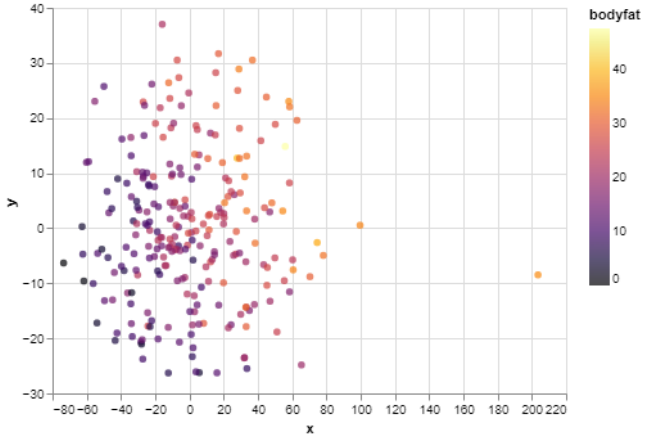

Figure 2: Standard PCA on `bodyfat`.

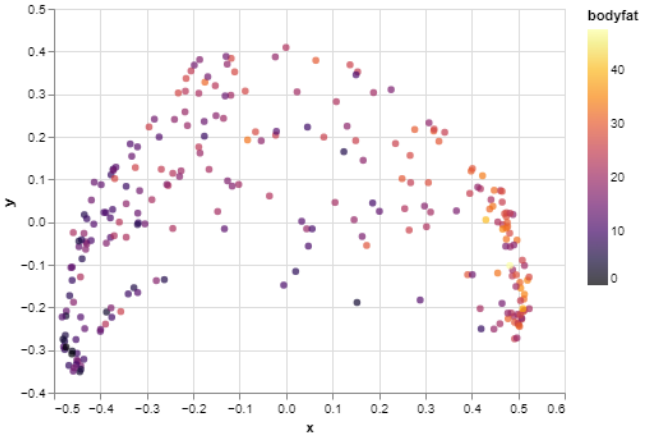

Figure 5: RCP Kernel-PCA on `bodyfat`.

As it can be seen, the simpler shape of the

representation comes from PCA, and on the other hand, the kernelized versions offer different shapes coming from the non-linearities of the method. In all of them, we can observe that the gradient coloring goes somehow along the coordinates of the projected data, telling us that they distinguish this variable as important.

All the kernelized versions offer better results than the linear one, including Fast Cluster PCA, although it is quite similar to MDP PCA. In this case, the best one seems to be Random Forest Partition Kernel PCA, as it seems to distinguish certain groupings of values better.

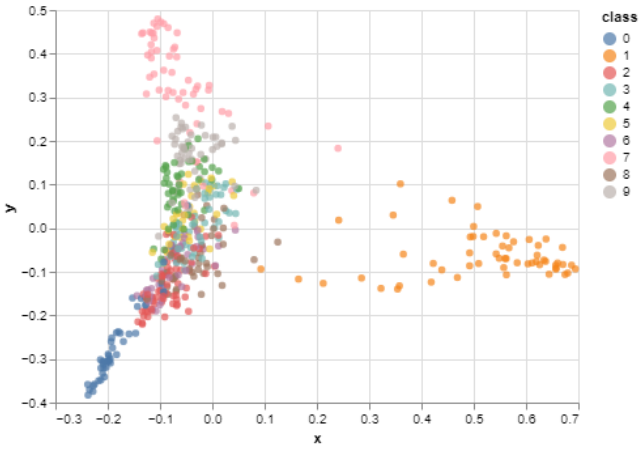

Figure 7: RFP Kernel PCA on `MNIST`.

5.2 Kernel PCA for the `MNIST` dataset

Similarly as we have done in the previous section, here we apply Kernel PCA to the `MNIST` dataset, to obtain a dimensionality reduction into a two variable space. In this case, the points are colorized with the different classes, i.e., the numbers from 0 to 9. Moreover, we have to take into account that `MNIST` has a large number of variables (pixels) for each row, so the dimensionality reduction task is much harder. The dimensionality reduction charts with `MNIST` are the following:

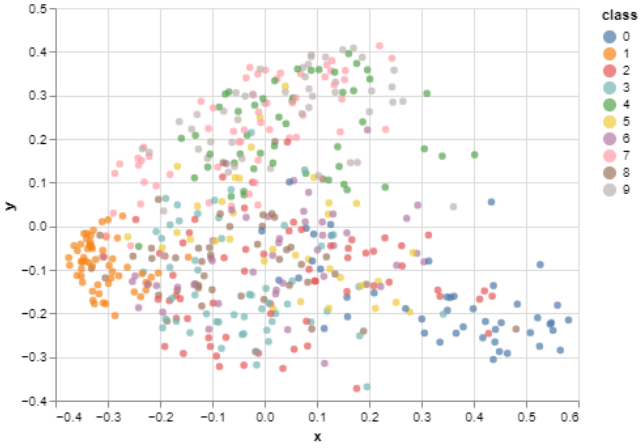

Figure 8: RCP Kernel PCA on `MNIST`.

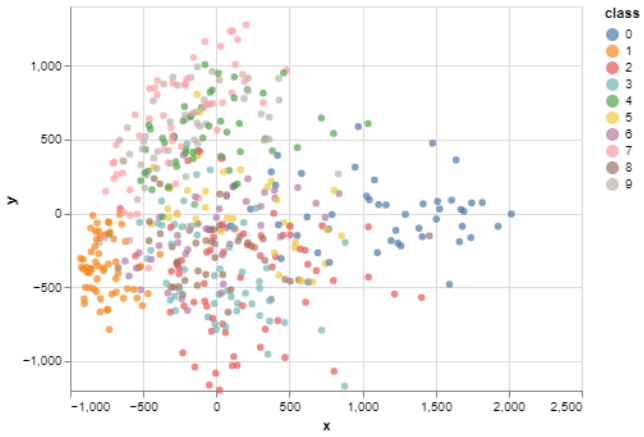

Figure 6: Standard PCA on `MNIST`.

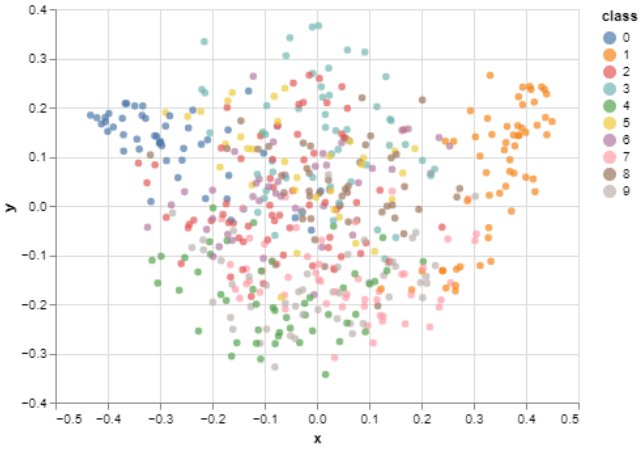

Figure 9: LSH Kernel PCA on `MNIST`.

As it can be seen, all of the methods offer unsatisfactory results. However, it seems that same class points are not far from each other, then it is possible that in higher dimensional representations, the separation between classes is greater. The best method that reflects this property is RFP Kernel PCA, while the other ones are very similar to each other. The RFP is a bit different from the other three, offering slightly better results.

5.3 Kernel SVM for the `bodyfat` dataset

In this part of the section we will test the regression capabilities of the Random Partion Kernels and, in particular, the MDP algorithm performance among them. We will use again the `bodyfat` dataset, and we will apply kernel SVM to predict the variable "bodyfat". The following bar plot shows the MSE of the tested methods:

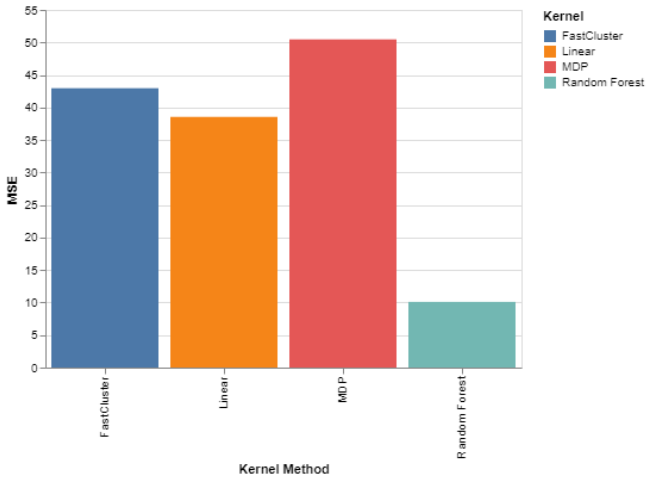

Figure 10: MSE of the Kernel SVM regressors on `bodyfat`.

As it can be seen, only the RFP kernel offers a better performance than the Linear kernel. A possible explanation of that could be that the `bodyfat` dataset is kind of linear, and the only advantage that RFP has is that it is supervised and it uses information of the "bodyfat" variable to generate the partitions, hence the partition it generates is related to the distribution of this variable.

5.4 kSVM for the `MNIST` dataset

In this last section of experiments we test the classification capabilities of the Random Partition Kernels and, in particular, the LSHIP algorithm performance among them. We will use again the `MNIST` dataset, and now, we will apply kernel SVM to predict the number that appears in each image. The following bar plot shows the prediction accuracy of the tested methods:

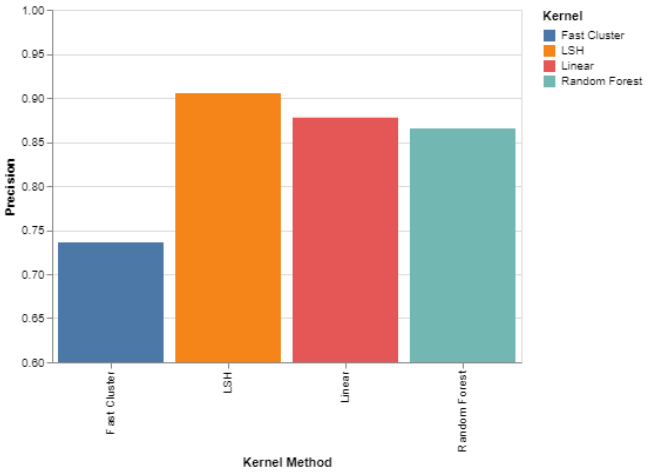

Figure 11: Fast Cluster PCA on `bodyfat`.

As it can be seen, the overall performance of the Random Partition Kernel is clearly superior to the standard SVM, with the RCP algorithm being the lowest performer of the tested algorithms, achieving a similar (but better) performance than the Linear kernel. This may happen because this dataset is far from being easy and the ideal partitions are hard to find. But, there are good news for the other kernels, as they offer a higher performance in this classification task. In particular, the LSHIP algorithm, that was designed to generate partitions in image datasets, is the one that has the highest performance, with more than 85% of accuracy.

6 Conclusions

6.1 This work

As the experiments show, Random Partition Kernels offer a clear improvement in performance in the tasks of data dimensionality reduction, regression and classification with respect to the unkernelized methods.

In general, RFP is the method that has a higher accomplishment of the tasks. This can be explained by recalling that RFP tries to predict one of the variables from the others in the dataset, and hence extracts more information about the data distribution than the other algorithms do, for the most part.

About the algorithms that we developed, it seems that introducing Markovian properties does not help quite a lot. The results offered by MDP are very similar to the ones from RCP, which is much faster in computation time. Nevertheless, the LSHIP seems to have a higher performance in the classification task than the others including RFP, which shows that designing a partition method for your specific problem should improve the performance of a model.

6.2 Future work

For the future of this work, one could develop more Random Partition generators, or even improve the ones that appear in this project. To create a partition method one could search a clustering algorithm and add some sort of randomness to make it a generator of random partitions. In our opinion, it seems more interesting to search for algorithms that are very specific to one single problem or type of data, and compare their performance with the other general ones to see if it is worth.

Additionally, one could think of more experiments to compare the Random Partition Kernels with other kernels or even with other non kernelized methods to see their competitive performance in real world datasets. In this project we decided not to do that as it would be too long.

In the line of looking for other random partition algorithms, one could also combine two or more algorithms that already generate random partitions to the same kind of data, hence modifying their generated kernel matrix, and maybe improving the performance of the associated kernel method. On the research aspect of the work, one could also investigate whether two algorithms that sample from similar distributions really make for similar kernels and performances, and derivate some results for the sensitivity of the kernelized algorithm performance with respect to a distortion in the partition distribution that generates the kernel.

References

[1] Alex Davies and Zoubin Ghahramani. The Random Forest Kernel and other kernels for big data from random partitions, 2014. arxiv.org.

[2] Stijn van Dongen. *Graph Clustering by Flow Simulation*. PhD thesis, University of Utrecht, 2000. Homepage, comprehensive explanation, implementation in Python.

[3] Àlex Batlle Casellas and Aleix Torres Camps. iPython Notebook with the code for this project, 2021-2022. Code hosted in GitHub.

A Code

The visualizations that have been used in this work have been created using code developed by ourselves using Python and interactive Python Notebooks, and the algorithms that we have described and developed have also been implemented using this tool. In [3] we have the code hosted on GitHub.