Advanced Machine Learning
Kernel Methods

Mohammad Poul Doust Robin Khatri

1 Introduction

Kernel methods are powerful techniques to deal with non-linearity in Machine Learning. A kernel mapping aims to project the data points onto a (much) higher dimensional space with hope of getting better representation for the data in that dimension for further tasks such as classification or clustering etc. For instance, in various classification scenarios, linearly inseparable data have higher probability to be linearly separable in higher space. However, this comes with the price of projecting all data points into higher space and apply our machine learning algorithm in that space, which is extremely expensive and complex. This is where kernel methods can be used in various algorithms. In this report, we will discuss the kernalized version of: Principle Component Analysis (PCA), K-Means, Logistic Regression and Support Vector Data Descriptions (SVDD). Moreover, we will compare the normal versions to the kernalized version against different datasets.

1.1 Kernels

Kernels are functions represent an inner product in a high-dimensional space. Hence, there is no need to project the data onto the the new space as long as we can use kernels to calculate the dot products between two data points in the higher-dimensional space:

$$\kappa\left(\mathbf{x_i}, \mathbf{x_j}\right) = \phi\left(\mathbf{x_i}\right)\phi\left(\mathbf{x_j}\right)^T \tag{1}$$

Where ϕ is nonlinear mapping function that maps a point in R^N to a feature space:

$$\Phi : R^N \to F$$

Most common kernels:

1. Polynomials of degree d
$$K(\mathbf{x}, \mathbf{y}) = (\mathbf{x} \cdot \mathbf{y} + c)^d,$$
where $c \geq 0$ is known as coefficient of polynomial kernel. For $c = 0$, the kernel is called homogenous. A slope α can also be added as a multiplier term making the kernel function $(\alpha\mathbf{x} \cdot \mathbf{y} + c)^d$.

2. Gaussian kernel (rbf)
$$K(\mathbf{x}, \mathbf{y}) = \exp\left(-\frac{\|\mathbf{x} - \mathbf{y}\|^2}{2\sigma^2}\right)$$

3. Sigmoid
$$K(\mathbf{x}, \mathbf{y}) = \tanh(\gamma\mathbf{x} \cdot \mathbf{y} + r)$$

There are many other kernel functions and customized kernel functions can also be designed. More information about kernel functions used in Machine Learning can be found in [1] [2]. In further sections, we shall see how kernel methods are highly useful in achieving better performance in dimensionality reduction, classification, regression and outlier detection.

2 Kernel-PCA

2.1 Kernalization

The goal of Principal Component Analysis (PCA) is to linearly project a dataset lying in d dimensional space onto a new space with M dimension where $M < d$. This is done by minimizing the reconstruction error from new space into old space which is equivalent to maximizing the variance in the newly projected space as follows:

$$\max_{\mathbf{u}_1,\ldots,\mathbf{u}_M} \mathbf{U}^T \Sigma \mathbf{U} \tag{2}$$

U is a $d \times d$ matrix. The columns of U are eigen vectors for which is the covariance matrix,

$$\Sigma = \frac{1}{N} \sum_i \mathbf{x}_i \mathbf{x}_i^T, \tag{3}$$

After calculating the covariance matrix from the zero-mean dataset, we find the corresponding M eigen vectors and project the original points in d dimension onto new space of M dimension as follows:

$$t_i = \mathbf{U}^T \mathbf{x}_i \tag{4}$$

If we want to kernalize PCA, we have to project data first into higher space with the goal of having a better representation of the data to be projected into lower dimension preserving most of the variance. This is helpful when PCA fails in capturing non-linearity since it is only a linear transformation of the data. First, we assume the data is normalized In the higher space to have mean of 0:

$$\frac{1}{N} \sum_{i=1}^{N} \phi(\mathbf{x}_i) = \mathbf{0} \tag{5}$$

Equation (3) in this higher dimensional space will be:

$$\mathbf{\Sigma} = \frac{1}{N} \sum_{i=1}^{N} \phi(\mathbf{x}_i) \phi(\mathbf{x}_i)^{\mathrm{T}} \tag{6}$$

The eigenvectors V and eigenvalues λ are given by:

$$\mathbf{\Sigma V} = \lambda \mathbf{V} \tag{7}$$

From (6) and (7) we get:

$$\frac{1}{N} \sum_{i=1}^{N} \phi(\mathbf{x}_i) \left\{ \phi(\mathbf{x}_i)^{\mathrm{T}} \mathbf{v}_k \right\} = \lambda_k \mathbf{v}_k \tag{8}$$

where:

$$k = 1, 2, \cdots, M$$

Thus,

$$v_k = \frac{1}{\lambda_k N} \sum_{i=1}^{N} \phi(x_i) \phi(x_i)^T v_k = \frac{1}{\lambda_k N} \sum_{i=1}^{N} (\phi(x_i) \cdot v_k) \phi(x_i)^T \tag{9}$$

Since the following quantity is a Scalar:

$$(\phi(x_i) \cdot v_k),$$

We can write:

$$v_k = \sum_{i=1}^{N} \alpha_{ki} \phi(x_i) \tag{10}$$

Which means that the solution of V lies in the span of

$$\phi(x_1), \ldots, \phi(x_n)$$

So to find v_k it is enough to find the coefficients α_{ki}

Now by substituting (10) in (8):

$$\frac{1}{N}\sum_{i=1}^{N}\phi(\mathbf{x}_i)\,\phi(\mathbf{x}_i)^{\mathrm{T}}\sum_{j=1}^{N}a_{kj}\phi(\mathbf{x}_j) = \lambda_k\sum_{i=1}^{N}a_{ki}\phi(\mathbf{x}_i) \tag{11}$$

Considering the kernel equation:

$$\kappa(\mathbf{x}_i,\mathbf{x}_j) = \phi(\mathbf{x}_i)^{\mathrm{T}}\,\phi(\mathbf{x}_j), \tag{12}$$

After re-arranging we get:

$$\frac{1}{N}\sum_{i=1}^{N}\kappa(\mathbf{x}_l,\mathbf{x}_i)\sum_{j=1}^{N}a_{kj}\kappa(\mathbf{x}_i,\mathbf{x}_j) = \lambda_k\sum_{i=1}^{N}a_{ki}\kappa(\mathbf{x}_l,\mathbf{x}_i) \tag{13}$$

Using matrix notation we can write:

$$\mathbf{K}\mathbf{a}_k = \lambda_k N\mathbf{a}_k \tag{14}$$

and for new point x to project it onto the k principle component:

$$y_k(\mathbf{x}) = \phi(\mathbf{x})^{\mathrm{T}}\mathbf{v}_k = \sum_{i=1}^{N}\alpha_{ki}\phi(x)^T\phi(x_i) = \sum_{i=1}^{N}a_{ki}\kappa(\mathbf{x},\mathbf{x}_i) \tag{15}$$

2.2 Implementation

The algorithm works as follows:

1. Calculate the centered Gram Matrix:

$$\widetilde{\mathbf{K}} = \mathbf{K} - \mathbf{1}_N\mathbf{K} - \mathbf{K}\mathbf{1}_N + \mathbf{1}_N\mathbf{K}\mathbf{1}_N \tag{16}$$

2. Compute the unit eigenvectors from the centered Gram matrix.

3. Project data points onto the first k eigen vectors using Equation 15.

2.3 Results

We used moons dataset (Sample size = 400) generated from `datasets.make_moons` method of scikit-learn [3] module to test the performance of Kernel PCA. Evidently, that the data is linearly inseparable in the original space. With PCA (or Kernel PCA with linear kernel) as shown in Figure 1, the data is linearly inseparable whether we project on two components or one component. However, Kernel PCA with radial basis function (rbf kernel) successfully projects the data onto a linearly separable space using only one component as depicted in Figure 2.

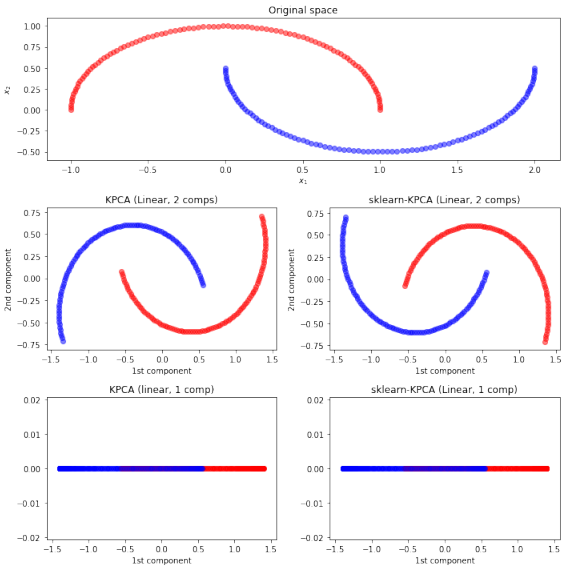

Figure 1: Kernel PCA with linear Kernel (Left: Our implementation, Right: scikit-learn).

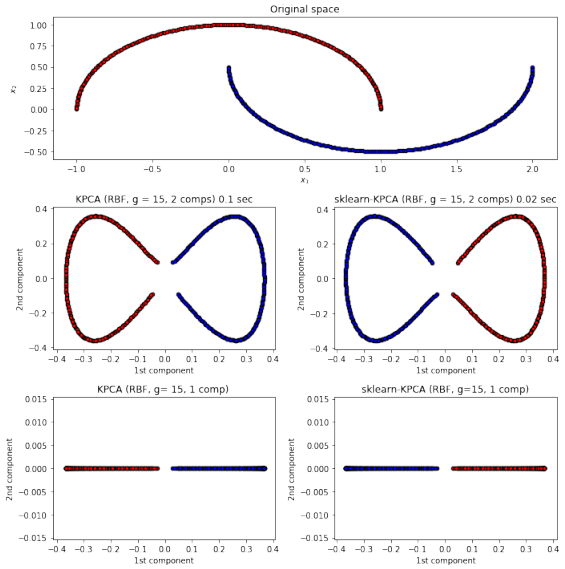

Figure 2: Kernel PCA with rbf Kernel (Left: Our implementation, Right: scikit-learn).

Our implementation is a little slower than scikit-learn's kPCA. On a sample sizes of 400 and 4000 respectively, the difference was around 0.07 and 2.3 seconds respectively. It is possible to work with very large datasets as well. However, in such cases, the time may increase. A reason for this could be the time taken in computation of Gram Matrix. If, however, we used the module `scikit-learn.pairwise` to compute Gram Matrix, the performance was very close to scikit-learn's kPCA.

Figure 3 shows another example of how in complex situations PCA fails to project data to a linearly separable space while with RBF kernel we succeed in doing so.

The choice of kernel depends on the problem at hand. To appropriately choose a kernel, cross-validation can be done. For circles dataset, polynomial kernel with degrees upto 5 fail to project data to a linearly separable space.

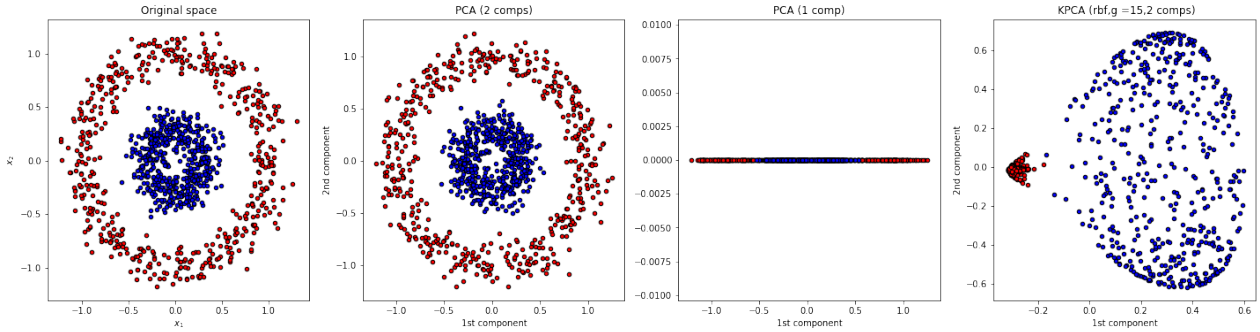

Figure 3: Circles dataset (Sample size: 100, noise level: 0.1), PCA (2 components), PCA (1 components) and kPCA (2 components, RBF Kernel).

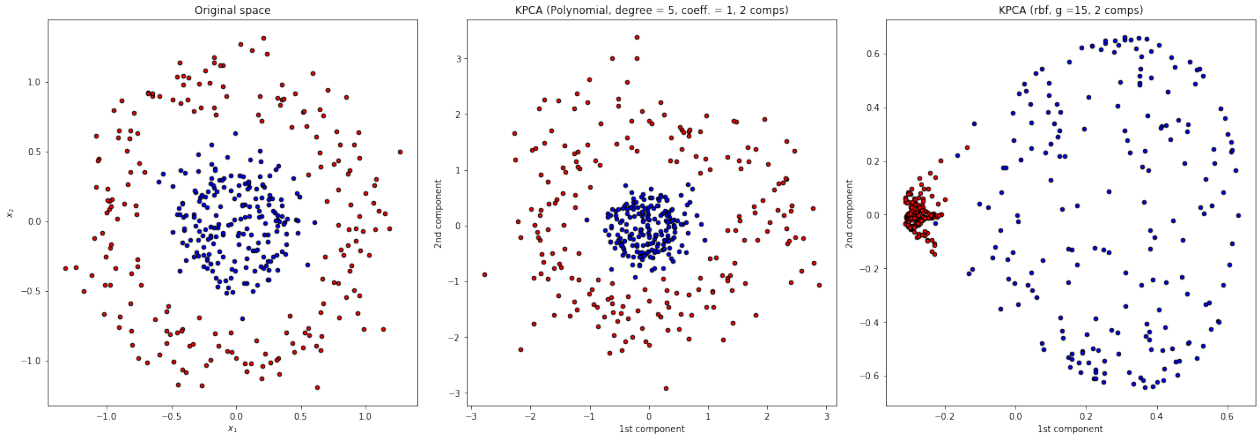

Figure 4: Circles dataset (Sample size: 400, noise level: 0.1), kPCA (Polynomial Kernel, degree = 5), kPCA (RBF Kernel, $\gamma = 10$).

We also tested our implementation on a very popular real dataset - iris [4]. The dataset comprises of three features corresponding to lengths and widths of sepals and petals.There are three classes corresponding to flower species(Setosa, Versicolour and Virginica). The dataset is represented below by three principal components in Figure 5:

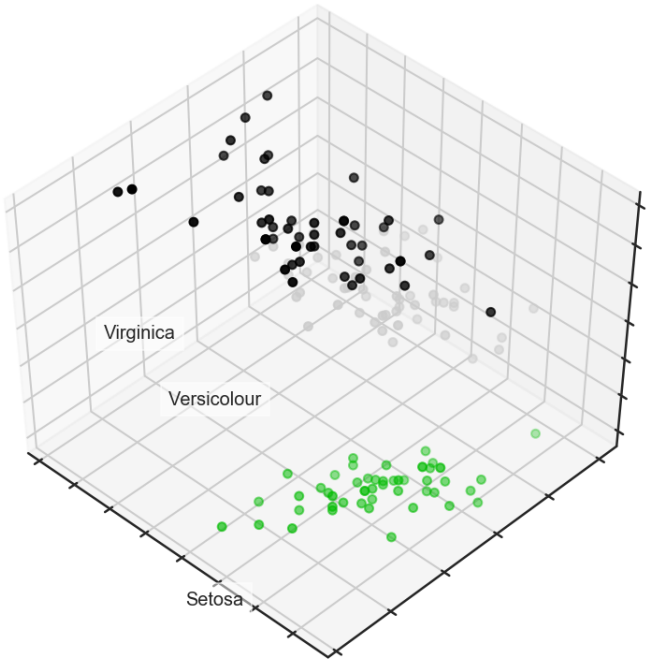

Figure 5: iris dataset [4] - Representation in 3D.

We choose to consider 2 principal components as they account for over 95% of explained variance.

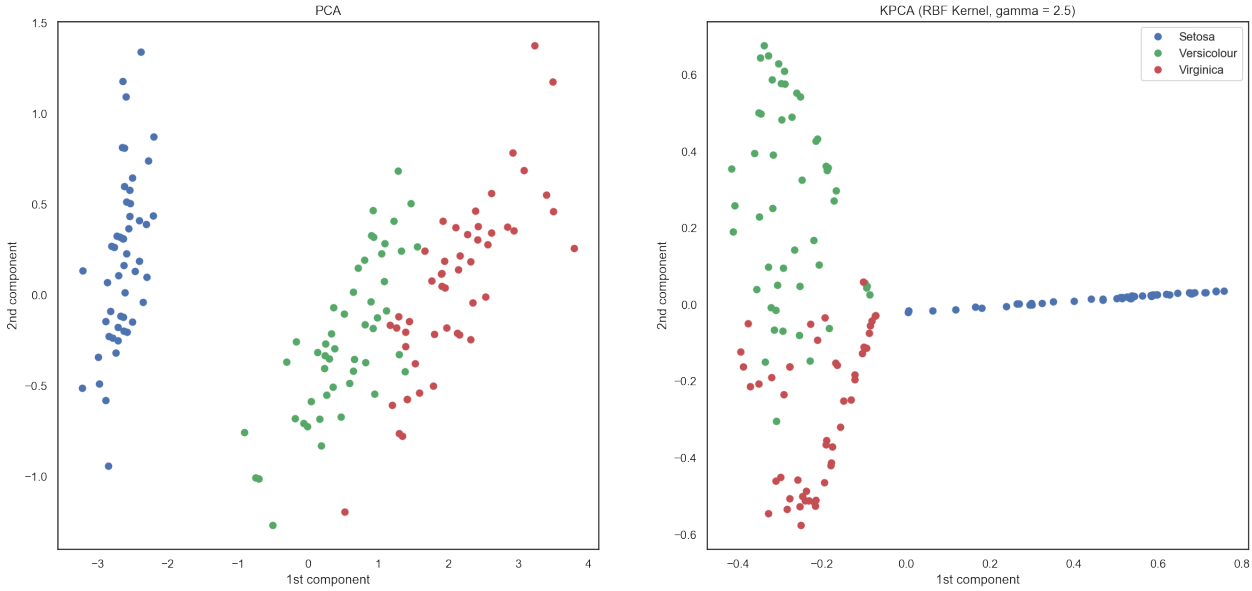

Figure 6: PCA (2 components), kPCA (2 components, RBF Kernel, $\gamma = 2.5$).

In figure 6, 2 principal components from PCA and kPCA (RBF kernel) are shown.

PCA as a pre-processing step for classification

With a decision tree classifier of maximum depth = 2 using `DecisionTreeClassifier` method from `scikit-learn`,

Accuracy on original dataset = Approx. 88%.

For 2 PCA components: Accuracy = Approx. 91.11%.

For 2 kPCA components: Accuracy = Approx. 95.56%.

Clearly, kPCA outperforms regular PCA in this case. Since the dataset is not very large (150 samples), the time taken for kPCA decomposition and PCA was comparable (less than 1 second). In our testing, the choice of γ is very important and the model is sensitive to dramatic changes in γ, therefore a cross-validation is recommended.

3 K-Means

K-Means is a clustering algorithm i.e. unsupervised classification. Its origin is attributed to Lloyd who gave its primal form and similar methods are often regarded as Lloyd algorithm or Lloyd-type methods [5]. Since then many algorithms have been developed to optimize and find the solution of original problem. The notion behind K-Means is to find k clusters given a dataset X. Choice of k is arbitrary. However with cross-validation (In case, we know true labels in a validation set) and/or scree plots (*e.g.* elbow method), we can find optimal k.

We select k random points in X as centroids and calculate their distances from every point. This distance can be eucledean or some other metric based on the problem at hand. Here, we only consider eucledean distances. However, a generalization to other distances is intuitive and easy to manage. Then we assign a point x_i where $i \in \{1, 2, 3...n\}$ to a cluster with centroid k for which the distance (or other similiarity measures *e.g.* for text data, cosine similiarity etc.) $d_{ik}(c_k, x_i)$ is minimum. Doing so, we end up with k clusters. Thereafter, we caluclate mean of all datapoints in a cluster and choose that as centroid of that particular cluster and repeat aforementioned process again until convergence. By convergence, we mean repeating the process until our centroids do not change. Essentially, we want within cluster dissimiliarity (or distance in our case) to be as little as possible while between cluster dissimiliarity to be as high as possible. It may take forever and therefore, we can also limit maximum number of iterations.

As an optimization problem in every iteration, K-Means can be presented as below:

If μ_k is the mean of a cluster C_k, the distance between μ_i and x_i is $||x_i - \mu_k||_2^2$. Initially, μ_k's are k random points. Therefore task is to find,

$$\begin{aligned} &\arg\min \sum \|x_i - \mu_{S_i}\|_2^2, \\ &\text{where, } S_i \in \{1, 2, 3...k\}, \text{ for } i \in \{1, 2, 3...n\}. \end{aligned} \tag{17}$$

After solving, this optimization problem, we select a new centroid,

$$\mu_j = \frac{1}{C_j} \sum x_i, i \in \{1, 2, 3...k\}, \forall j \in \{1, 2, 3...k\}. \tag{18}$$

In Equation 17, in hope of better clustering by projecting it to a higher dimensional space, we can choose a kernel function $\varphi(.)$ such that our new optimization problem in every iteration becomes,

$$\begin{aligned} &\arg \min \sum (||\varphi(x_i) - \mu_{Si}||_2^2), \\ &\text{where, } \varphi(x_i) : \mathcal{X} \times \mathcal{X} \to \mathcal{R} \text{ in new space } \mathcal{H} \\ &\text{and, } S_i \in \{1, 2, 3...k\}, \text{ for } i \in \{1, 2, 3...n\}. \end{aligned} \tag{19}$$

As it can be easily shown that for mean $= K_j$ in $\mathcal{H}$. The new centroid can be selected by solving,

$$\varphi_j = \frac{1}{|C_j|} \sum_{j \in S_i} \varphi(x_i), i \in \{1, 2, 3...|C_j|\}, \forall j \in \{1, 2, 3...k\}. \tag{20}$$

Therefore, our loss function in every iteration can be given by,

$$\arg\min \sum \|\varphi(x_i) - \frac{1}{|C_j|} \sum_{j \in S_i} \varphi(x_i), i \in \{1, 2, 3...|C_j|\}\|_2^2, \tag{21}$$

This optimization problem is equivalent to solving,

$$\arg\min \sum (K(x_i, x_i) - \frac{2}{|C_{S_i}|} \sum_{j \in S_i} K(x_i, x_j) + \frac{1}{|S_i|^2} \sum_{l,j \in C_{S_i}} \sum K(x_j, x_l)) \tag{22}$$

where, $S_i \in \{1, 2, 3...k\}$, for $i \in \{1, 2, 3...|C_j|\}$.

3.1 Implementation

The algorithm of our implementation is as follows:

1. Input: X, kernel parameters (Default kernel: RBF), k, max_iter: integer, random state (To initialize centroids).
2. Get k random points as centroids.
3. Solve Equation 22. For max_iter iterations.
4. When a new datapoint x comes, assign it to a cluster that minimizes $K(x, c_j)$ $\forall j \in \{1, 2, 3...k\}$.

K-Means has a time complexity of $\mathcal{O}(nkdi)$ where n is the number of samples in set X, k is the number of clusters and d is the dimension of data while i is the number of iterations taken for convergence or whenever the algorithm is programmed to stop. This is an expensive algorithm. On top of this, we calculate a gram matrix (complexity of $\mathcal{O}(n^2)$). In this case, the complexity can be given by $\mathcal{O}(n^2ki)$. In this report, we implement this algorithm, however, there are several optimizations available such as by reordering the clustering process to use only a portion of Gram Matrix (Allows for storing Gram matrix and storing it into the memory). There are also versions that use paralellization to compute Gram Matrix and to run iterations. We can also implement an algorithm known as Approximate K-Means which uses sampling to avoid calculating entire Gram Matrix [6]. This is particularly useful in large scale clustering.

3.2 Results

First we tested our model on `circles` dataset generated by our own implementation. The results with both linear and RBF kernel are presented in Figure 7.

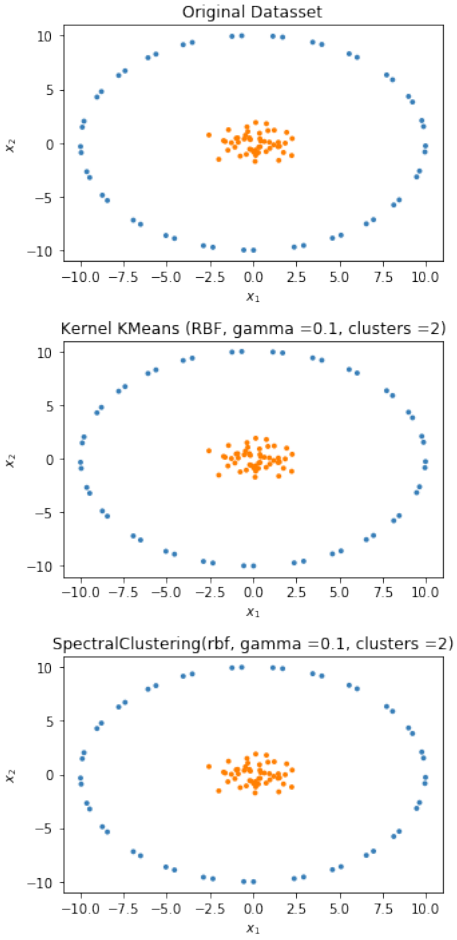

Figure 7: From top to bottom: Original dataset (Sample size: 100), Kernel K-means (RBF, $\gamma = 0.1$) (50 iterations), Spectral Clustering (`scikit-learn`).

On the same dataset K-means gives bad results as presented in figure 8.

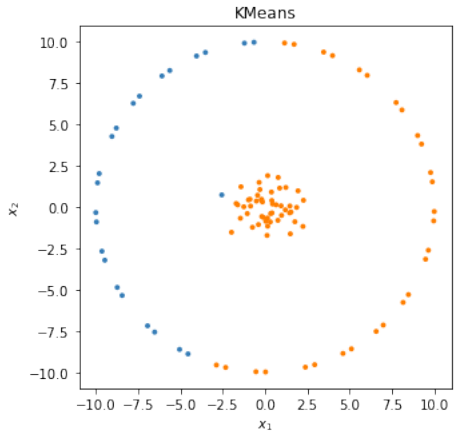

Figure 8: K-means on circles dataset (Sample size = 50)).

We further test our model on a variety of toy datasets as presented in figure 9.

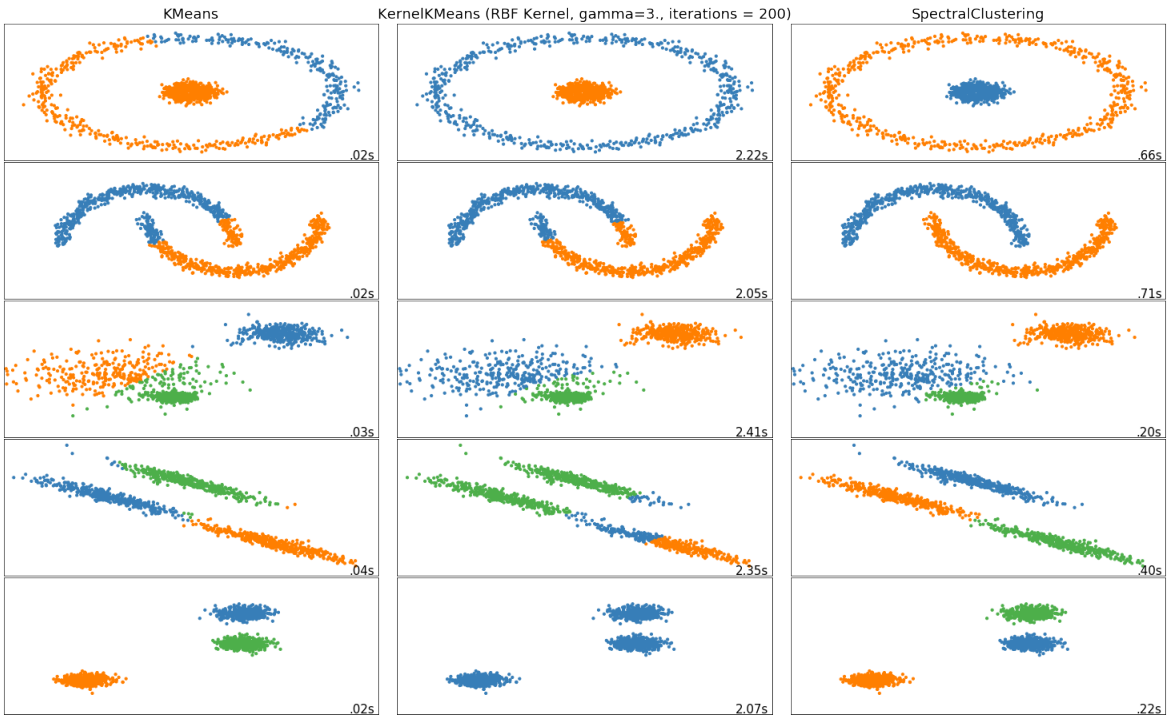

Figure 9: Toy datasets generated from `scikit-learn.datasets.make_circles` with factor 0.1 (Sample size = 1200). From Left to Right: K-means, K-means (RBF Kernel with $\gamma = 3$, iterations =200), and Spectral Clustering (from `scikit-learn`).

Clearly, in Figure 9, Kernel K-means perform better than K-means in case of `circles` and `blobs` but does not perform that much better in case of `moons` dataset. Further, the performance is less better than that of Spectral Clustering which is a clustering algorithm similiar to K-Means with spectral relaxation. Further, the time taken in computing clusters with our implementation is slower than that of K-means and Spectral Clustering. However, even with 1200 samples, the model performs reasonably fast given the complexity of the algorithm as mentioned previously.

The choice of γ is very important and changes in γ may cause errors. Therefore, it is advised to perform cross-validation.

A Kernel K-means is not always better than K-means. Therefore, the choice of using Kernel K-means as opposed to K-means is problem-dependent. More importantly, choosing the right kernel is important. In Figure 10, we can see that with polynomial kernel (degree = 3), the model makes more errors than regular K-means.

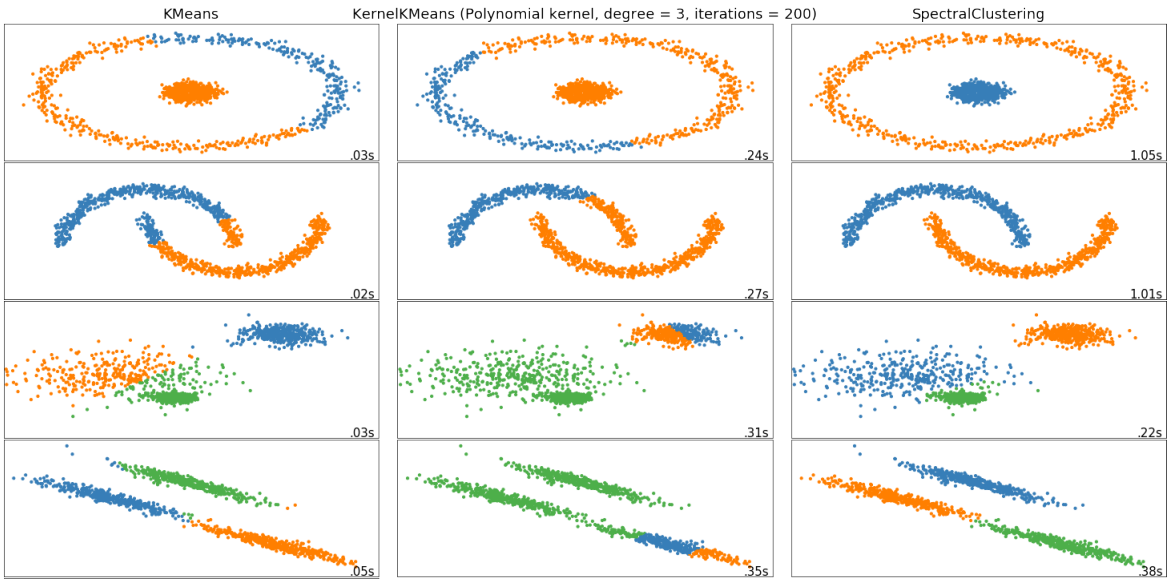

Figure 10: Toy datasets generated from `scikit-learn.datasets.make_circles` with factor 0.1 (Sample size = 1200). From Left to Right: K-means, Kernel K-means (Polynomial Kernel with $= 3, c = 1$, iterations =200), and Spectral Clustering (from `scikit-learn`). c is the coefficient for polynomial kernel as mentioned in section 1.

Further, we tested our model on a real dataset. Similar to kPCA, we chose `iris` dataset. The dataset has 4 features and 3 classes. Firstly, to see the optimal number of clusters, we can use elbow method which checks within cluster squared sum (wcss) of distances with increasing number of clusters. In Figure 11, we can see that after $k = 3$, increasing k further does not result in decrement in wcss. This result was expected since we already know the dataset has three species.

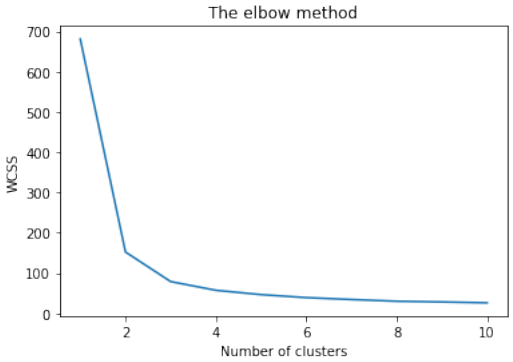

Figure 11: wcss vs k. k=3 is optimal.

We computed clusters using both KMeans and Kernel K-means. γ was chosen such that the model performs best on validation sets. After clustering, since we already have the labels, we computed accuracies for both methods. Figures 13 and 14 present this clustering.

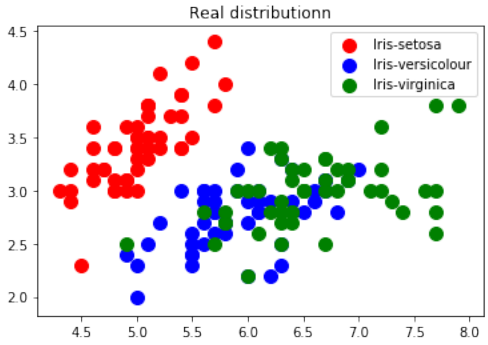

Figure 12: Original dataset.

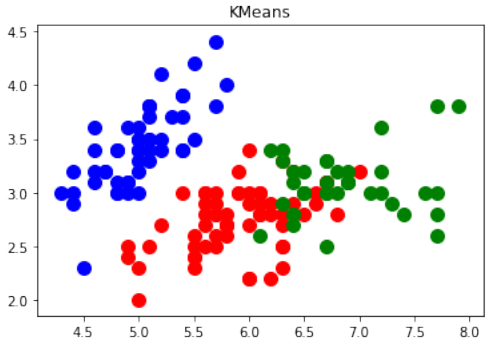

Figure 13: K-means

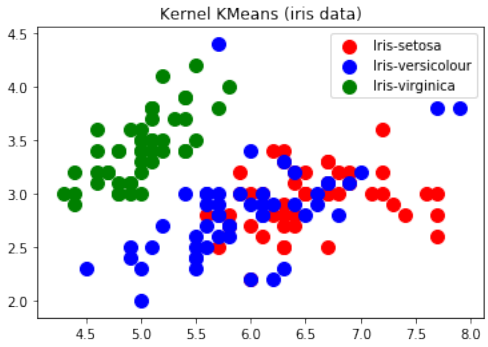

Figure 14: Kernel KMeans ($\gamma = 2.5$)

Accuracy: K-means: 89.33% Kernel K-means ($\gamma = 2.5$): 93.33 %

.

4 Kernel Logistic Regression

Let's start by the simple Logistic Regression. We know that the hypothesis of Logistic regression is given by the following equation:

$$h(w,x) = \frac{1}{1+\exp\left(-w^\top x\right)} = Sigmoid\left(w^\top x\right) \tag{23}$$

The loss function is:

$$J(w) = \frac{1}{m}\left(-y^T \log(h) - (1-y)^T \log(1-h)\right) \tag{24}$$

(24) is usually optimized using Gradient Descent by differentiating the loss function to update weight at each step:

$$\frac{\partial J(w)}{\partial w} = \frac{1}{m} X^T (h-y) \tag{25}$$

The update rule:

$$w = w - \alpha \frac{\partial J(w)}{\partial w} \tag{26}$$

4.1 Kernalization

To kernalize Logistic regression we can change (23) by projecting x to higher space:

$$h(w,x) = \frac{1}{1+\exp\left(-w^\top \phi(x)\right)} \tag{27}$$

According to representer theorem we can write :

$$h(w,x) = \frac{1}{1+\exp\left(-\sum_{i=1}^{n} \alpha_i y_i \phi\left(x_i\right)^\top \phi(x)\right)}$$

and hence:

$$h(w,x) = \frac{1}{1+\exp\left(-\sum_{i=1}^{n} \alpha_i y_i k\left(x_i, x\right)\right)} \tag{28}$$

For gradient descent, it works the same by replacing X with K(X,X):

$$\frac{\partial J(\theta)}{\partial \theta} = \frac{1}{m} K(X_{train}, X_{train})^T (h-y) \tag{29}$$

4.2 Implementation

The algorithm works as follows:

1. Calculate the Kernel Matrix:
2. Optimize (24) using Gradient Descent/Stochastic Gradient Descent, gradient calculated using (29)
3. Predict labels according to (28)

Algorithm *SGD* for Kernel Logistic Regression
procedure $SGD\left(D, \theta^{(0)}\right)$
$\theta \leftarrow \theta^{(0)}$
while not converged do
  for $i \in$ shuffle $(\{1, 2, \ldots, N\})$ do
    for $k \in \{1, 2, \ldots, M\}$ do
      $\theta_k \leftarrow \theta_k - \lambda\left(\mu^{(i)} - y^{(i)}\right) K(X_k, X)^{(i)}$
      where $\mu^{(i)} := h_\theta\left(\mathrm{x}^{(i)}\right) = 1/\left(1+\exp\left(-\theta^T K(X_{train}, X_{train})\right)\right)$
    return θ

4.3 Results

In this section, we compare three different implementation of Logistic Regression:

1. Our implementation of Logistic Regression
2. scikit-learn Logistic Regression
3. Our implementation of Kernalized Logistic Regression

Several Kernels was evaluated and tuned with respect to hyper-parameter

1. Linear Kernel (no kernel)
2. Radial Basis Function Kernel (RBF).
3. Polynomial Kernel (Poly).
4. Sigmoid Kernel

The mentioned implementation was tested against several datasets:

1. Our generated dataset.
2. scikit-learn "Circles" dataset.
3. scikit-learn "Make Moons" dataset.

	LR	**scikit-learn LR**	**KLR(poly, degree=7)**	**KLR(rbf, b= 15)**
Accuracy	47.05%	47.05%	100 %	100%
Time	1.04 sec	0.003 sec	1.851 sec	1.930 sec

Table 1: S Logistic Regression Results - Circles Data

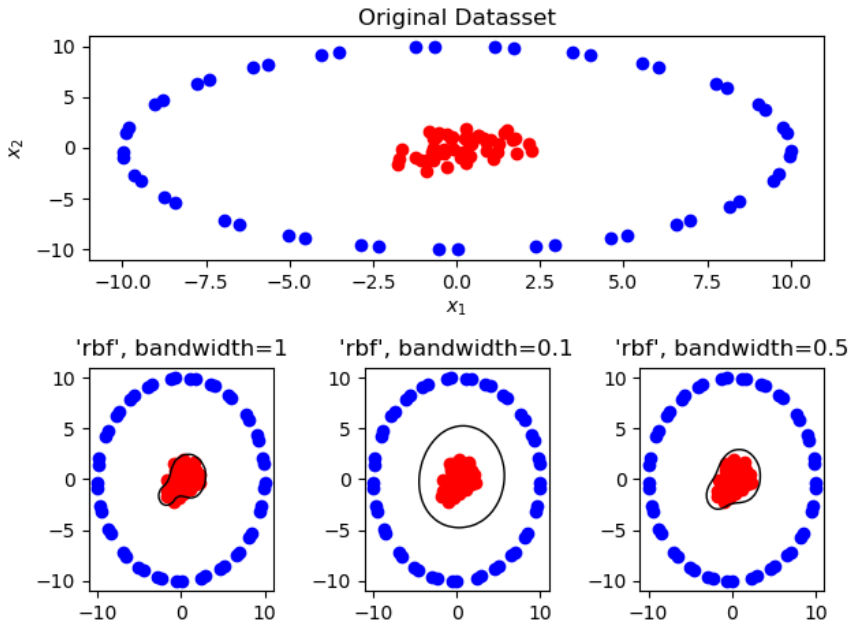

Figure 15: Comparison on synthetic data: the dataset is generated to plot two overlapping circles with radius of 10. We can see that rbf is fitting the data correctly with different parameters

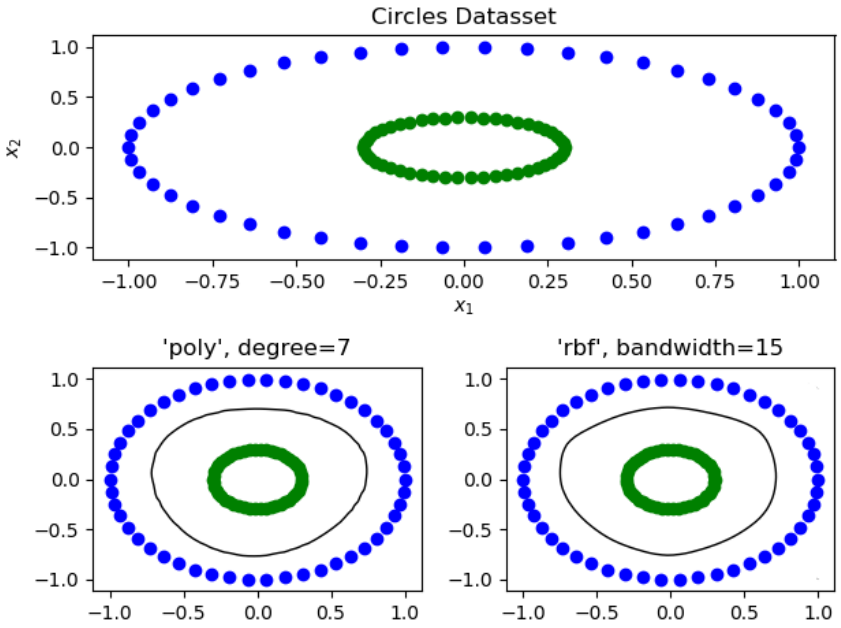

Figure 16: Comparison on scikit-learn generated dataset "circles" with factor of 0.2. We can see that rbf kernel is able to capture the nonlinear trend in the data

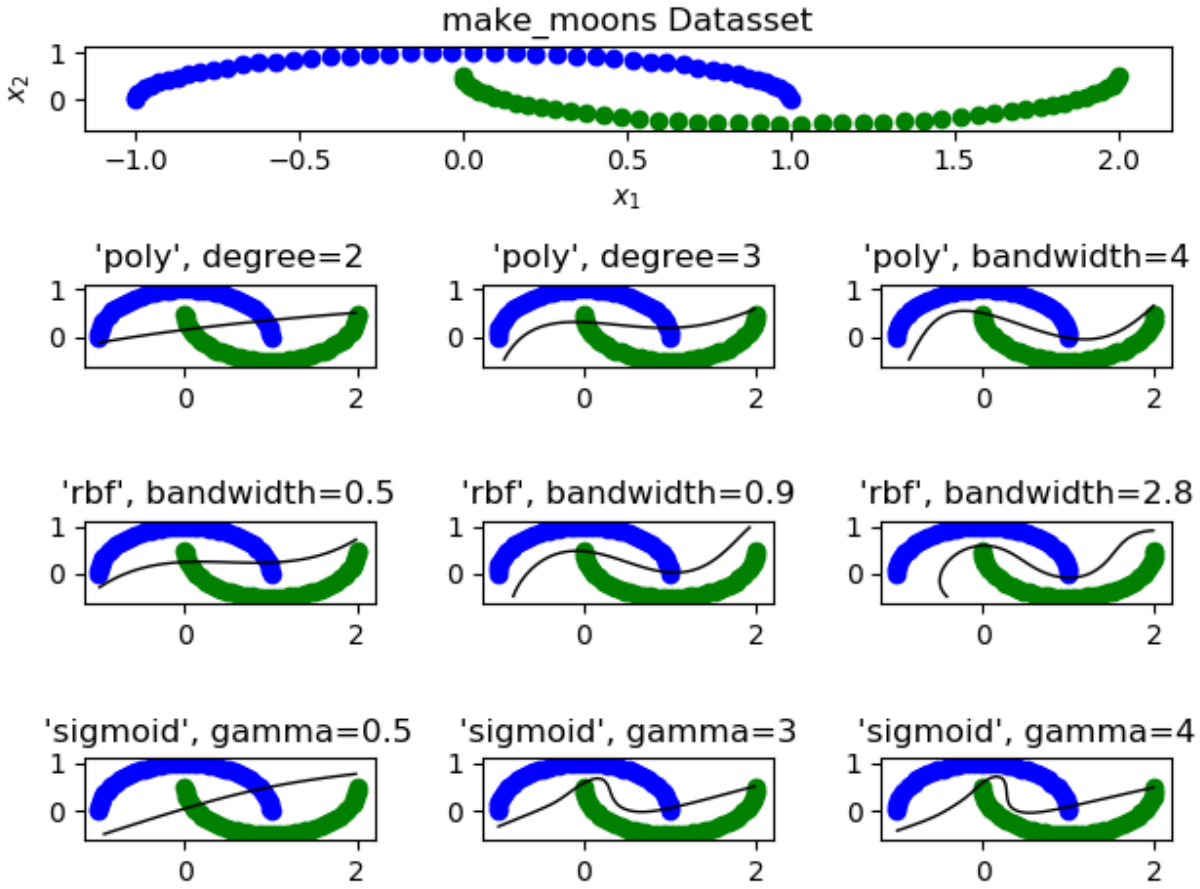

Figure 17: Toy dataset generated from `scikit-learn.datasets.make_moons` (linearly inseprable)with Sample size = 100). different kernels tested (Polynomial, RBF, Sigmoid) with different hyper-parameters

Kernel	**Accuracy (in %)**	**Time (in seconds)**
Poly(deg=2)	88	14.4
Poly(deg=3)	94.0	14.3
Poly(deg=4)	100	14.6
RBF(band=0.5)	92	14.7
RBF(band=0.9)	98	14.5
RBF(band=2.8)	100	14.5
Sigmoid(gamma=0.5)	85	14.4
Sigmoid(gamma=3)	99	14.6
Sigmoid(gamma=4)	97	15.1

Table 2: Kernalized Logistic - Comparison of time and accuracy for different Kernels and their parameters. Dataset generated from `make_moons` of scikit-learn.

5 Support Vector Data Descriptors

In this section, we shall discuss Support vector Data description (SVDD), a model that can be used in a varierty of settings to detect outliers and also can be extended to a two-class problem.

SVDD can be proposed as either a hyperplane or hypersphere approach. Both approaches give comparable solutions. In this report, we discuss the hypersphere approach given by . Simply speaking, the model learns a hypersphere minimizing its volume to fit a target dataset. The datapoints that lie at the boundary of such hypersphere are known as support vectors. To deal with more complex situation of modelling the target, it is also possible to kernalize the model.

5.1 SVDD as an optimization problem

5.1.1 One-Class Case

In classic SVM, we learn a function $f(x; w)$ forming a boundary to separate two classes of a target set. In one class case, this can be viewed as finding a hypersphere around the target set as shown in figure 19 [7][8].

If R is the radius with center at c, our purpose is to minimize the R^2 such that all points in the training set X lie inside or on the hypersphere.

$$\begin{aligned} &\min\ R^2 \\ \text{s.t. }\ &||x_i - a|| \leq R^2,\ \forall\ x_i \in X, i = 1, 2...n. \\ &n \text{ is the number of samples in } X \end{aligned} \tag{30}$$

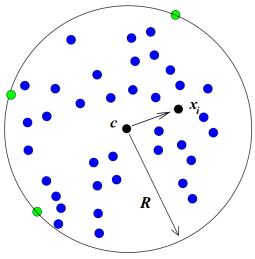

Figure 18: Learning a hypersphere with center C and radius R to model the target data (represented in blue). Datapoints on the sphere boundary are called support vectors similar to support vectors in SVM.

Using Lagrangian multipliers α_i, we can obtain the dual form of this optimization problem. Its dual form can be given as following:

$$\begin{aligned} &\min\ \alpha^T G\alpha - \alpha^T diag(G) \\ \text{s.t. }\ &\sum_{i=1}^{n} \alpha_i = 1 \text{ or equivalently, } e^T\alpha = 1 \\ &0 \leq \alpha_i\ \forall i, \end{aligned} \tag{31}$$

Where G is variance-covariance matrix of X with element $G_{k,l}$ being $< x_k \cdot x_l >$, $k, l \in \{1, 2...m\}$, m is the number of features dataset X. Here, $< \cdot >, \cdot >$ represnts inner product between two vectors. This optimization problem is in the form of a quadratic programming problem and can be easily solved.

R is the distance from center c of the hypersphere to one of the points (x_k) at the boundary *support vectors*. It is given by,

$$R^2 =< x_v, x_v > + \sum_{i,j} \alpha_i \alpha_j < x_v, x_v > -2 \sum_i \alpha_i < x_v, x_v >, \tag{32}$$

We can also introduce the notion of making errors by introducing slack variables in the inequality constraint. By introducing slack, we allow model to have c not strictly smaller than R^2. This makes model more robust and helps in outlier detection.

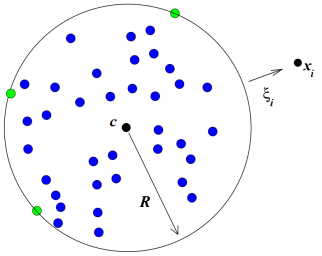

Figure 19: Learning a hypersphere with center C and radius R to model the target data with errors.

On introducing slack variables ξ_i, new optimization problem is,

$$\begin{aligned} &\min\ R^2 + C \sum_{i=1}^{n} \xi \\ &\text{s.t. } ||x_i - c|| \leq R^2 + \xi_i, \forall\ x_i \in X, i = 1, 2...n, \\ &\quad \xi_i = 0, i = 1, 2...n. \\ &\quad n \text{ is the number of samples in } X, \end{aligned} \tag{33}$$

Here C allows for a trade-off between proper data description *i.e.* fitting and errors. $C \geq 1$ is the no-slack case. C controls the influence of a datapoint on the final model. For $\alpha_i = C$, the object is an outlier [9]

By introducing lagrangian multipliers α_i and γ_i, we can obtain the dual form of this optimization problem. Its dual form can be given as following:

$$\begin{aligned} &\min\ \alpha^T G \alpha - \alpha^T diag(G) \\ &\text{s.t. } \sum_{i=1}^{n} \alpha_i = 1 \text{ or equivalently, } e^T \alpha = 1 \\ &\quad \gamma_i < \alpha_i < C - \gamma_i \text{ or equivalently, } 0 \leq \alpha_i \leq C\ \forall i, \\ &\quad \gamma_i \geq 0\ \forall i. \end{aligned} \tag{34}$$

The formulation of R^2 remains same as in equation 32.

5.1.2 Two-Class case

If dataset X has two classes with one being our target class (labelled +1) and other being the outlier class (labelled -1). In this case, we treat negative classes to be outliers. The formulation of the optimization

problem remains same as in Equation 34, except we seperate slack variables ξ_i to two sets ξ_i and ξ_l, both ≥ 0 as before. This is done to include separate errors for these classes. So the new optimization problem becomes,

$$\begin{aligned}
&\min R^2 + C\sum_{i=1}^{p}\xi_i + C\sum_{l=1}^{m}\xi_l, p = \text{ No. of positive samples }, m = \text{ No. of negative samples.} \quad (35)\\
&\text{s.t. } ||x_i - c|| \leq R^2 + \xi_i, \forall\ x_i \in \text{ positive samples }, i = 1, 2...p,\\
&\text{s.t. } ||x_l - c|| > R^2 - \xi_l, \forall\ x_l \in \text{ negative samples }, l = 1, 2...m,\\
&\quad \xi_i = 0, i = 1, 2...p.\\
&\quad \xi_l = 0, l = 1, 2...m.\\
&\quad p + m = n \text{ is the number of samples in } X,
\end{aligned}$$

Similarily we introduce two sets of lagrangian multipliers α_i and α_l for each of the class. However, since the labels $= \pm 1$, we have multipler,

$$\alpha_i' = y_i\alpha_i. \quad (36)$$

The dual formulation is identical to Equation 35 except instead α_i, we use α_i'. Evidently, two class SVDD is simply a generalization of the one-class SVM in two-class case as if there is no outlier class, $\alpha_i' = y_i\alpha_i$. The optiimzation process also remains the same as before and involves solving quadratic programming problem. Only computational time for optimization is increased because of increased number of lagrangian multipliers due to new α_i for outlier samples. [9]. In following subsections, we shall discuss the performance of SVDD in two-class case in comparison to classic SVM. We shall also look into the effect of presence of two-class in the time complexity.

5.1.3 Kernelized Version

Since a hypersphere is a very well defined shape, to allow for a more modelling in complex settings, we can kernalize the variance-covariance matrix to obtain a gram-matrix G. The form of dual optimization problem remains same except for G. In this case,

$$G(k, m) = K(x_j, x_m), \quad (37)$$

where $k(\cdot, \cdot)$ is a kernel function.

5.2 Implementation

The algorithm steps are as follows:

1. Input: Dataset X, $C = 0.01, \gamma = 3.$, degree = 1, kernel = None. Kernel choices: linear, polynomial and rbf (Default). In two class case, a 1-D array y with target class = 1 and outlier class = -1.
2. In one class case, create an array of 1's, size(y) = sample_size(X).
3. Calculate kernel mapping and its derivative for every sample (From gram matrix)
4. Find the nearest positive semi-definite (PSD) matrix of the gram matrix. Call it G.
5. Initialize α_i, Set γ_i = 1e-5 or similar small quantity.
6. Solve (positive semidefinite) quadratic programming problem given in Equation 34. To solve the problem,we used `quadprog` that uses the algorithm described in [10]. We also wrote a programme to do that, however, it is much slower but can be used for small problems. `quadprog` and many existing solvers are written in Cython and therefore much faster.

7. Computer R^2 and get a decision rule:

   (a) Get support vectors SV_k: For $|\alpha_i|$'s $> \gamma_i$.

   (b) $\forall\ k$ in SV Compute R_k^2 Using Equation 32.

   (c) θ = mean of R^2 for all support vectors.

8. For any new point x_i', classify it to be target class or an outlier class based on the rule $sign(\theta - R_i^2)$.

The complexity of this algorithm is $\mathcal{O}(n^3)$. Recently, faster versions of it has been developed such as Fast Incremental SVDD (FISVDD) which has a complexity of $\mathcal{O}(k^2)$ where k is the number of support vectors [11]. In the following sections, we shall see the performance of SVDD in both one class and two class settings including the run-time and accuracy.

5.3 One Class Case

Data description

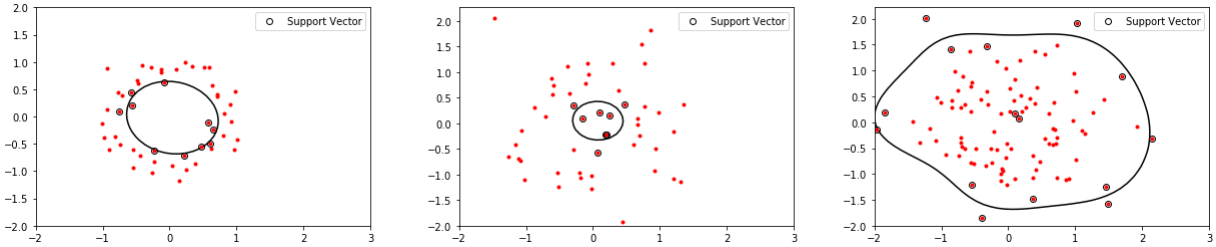

Figure 20: Using Kernels in SVDD. With increased noise in data, we fail to learn data well. All Images are generated using `make_circles` method from `scikit-learn.datasets` module. Noise levels in images (From left to right): 0.1, 0.5, 0.5. First two images use linear kernel with C = 0.1 and in the last setting, we used an RBF kernel.

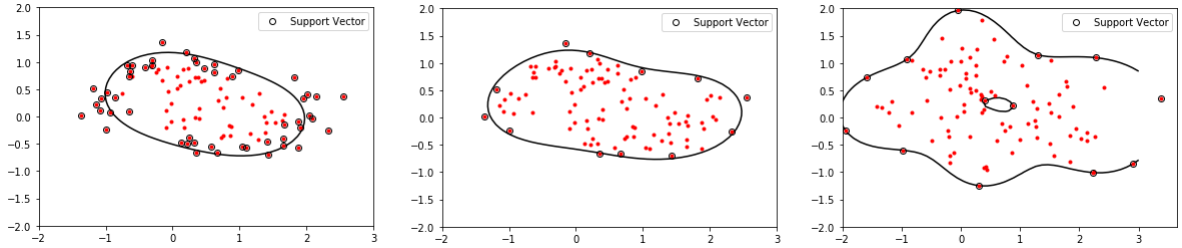

Figure 21: Effect of C (Upper bound) on the learned boundary. From left to right: $C = 0.02$, $C = 0.10$, $C \geq 0.5$ (All use RBF kernel). $C \geq 1$ is the no error case *i.e.* we don't allow a trade-off with outliers. In this testing, for this dataset, increasing C after 0.5 did not have much impact on data description.

Outlier Detection (Unsupervised)

Our model can be utilized to detect outliers in an unsupervised setting. By controlling for C, we allow the model to make errors while training. Below we compare our implementation with One-Class SVM from scikit-learn and other outlier detection algorithm. For comparison, we make use of similiar testing setting as used in [12]. To generalise our algorithm, we have used both X and y as arguments in method `fit` of our class, and therefore in case of only one class in training set X, we supply an array of ones making it compatible with both one-class and two-class settings.

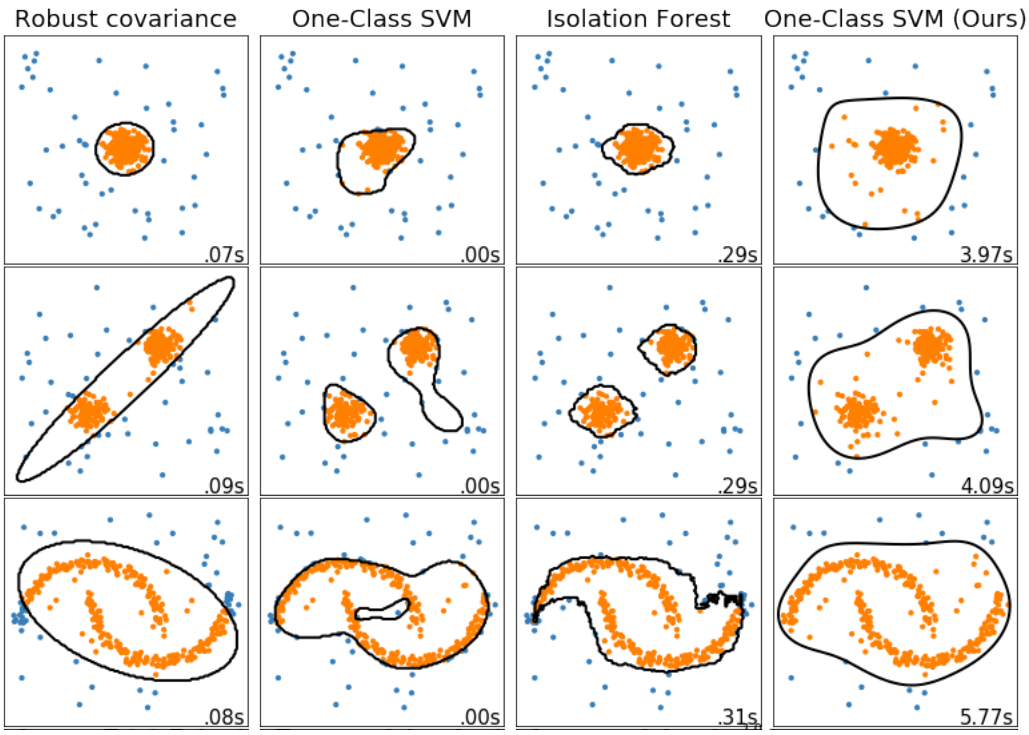

Figure 22: Comparison of our model with other outlier and novelty detection algorithms. Blue points are outliers. The testing template was taken from [12].
From left to right - Robust Covariance, One-Class SVM (scikit-learn), Isolation Forest and One-Class SVM (Our implementation).
For our model, we used $C = 0.02, \gamma = 0.1$. For One-class SVM (scikit-learn): RBF kernel was used with ν (outlier fraction)$= 0.15$. and $\gamma = 0.1$.

Evidently, while the model is not perfect, it is able to detect outliers well. Comparing our version with scikit-learn's `oneClassSVM`, we notice that they are not completely similiar for same gamma parameter. The reason for this could be difference in algorithm implementations. We make use of algorithm given by Tax [8] and so choose a bound C while in scikit-learn, we choose an outlier-fraction ν for the primal form in algorithm proposed by Schölkopf et. al. [13]. While both these parameters are to control for making errors, there intuitions are different. C controls for the upper bound of lagrangian parameters $\alpha_i, i \in \{1, 2, 3...n\}$, while ν in algorithm according to Schölkopf describes an upper bound for fraction of outliers and a lower bound on the number of training examples used as support-vectors. Lastly, our implementation is a quite slower than One-Class SVM in scikit-learn. Perhaps because we are solving a quadratic programming problem.

5.4 Two-class case and comparison with SVM

Below, we can see that, we can train our model in two class case as well. For demonstration, we make use of `make_moons` dataset with two classes. Our target class is labelled $+1$ and the other class is the outlier class labelled -1. This is essentially a supervised outlier detection.

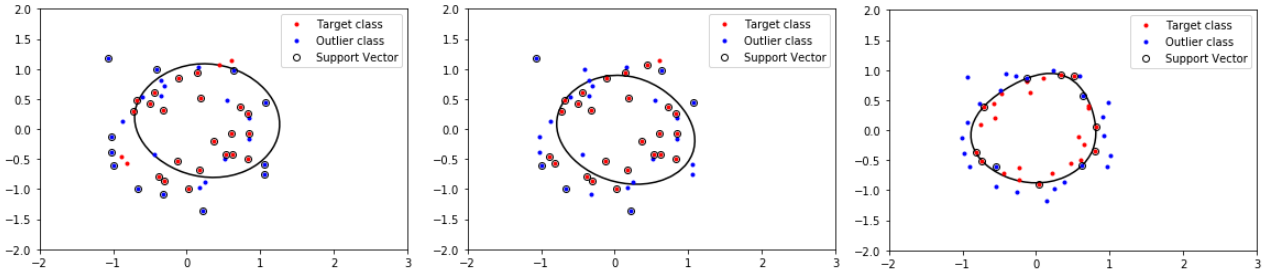

Figure 23: Classification task (From left to right): Linear Kernel, Polynomial Kernel (Degree=3, coefficient=0), RBF Kernel ($\gamma = 0.5$). For complex classification tasks, using kernelizaed version is recommended. Our model uses RBF kernel by default.

Performance comparison with SVM:

For comparison in performance, we train both SVM and our implementation of SVDD on 80 % examples of generated `make_moons` datasets at different noise levels and test on remaining 20 % examples on respective datasets. The test accuracies are given below:

Table 3: SVM and SVDD for classification: Accuracy over test set (in %). Dataset: moons - with noise 0.2 and 0.7). {Parameter selection through Cross-validation.}

		RBF Kernel(γ=0.7)	**Polynomial kernel (degree=2)**
Noise = 0.2	SVM	95	85
	SVDD (C=0.3)	90	80
Noise = 0.7	SVM	70	70
	SVDD (C=0.5)	63.30	65

Evidently, in supervised setting of outlier detection, the model performs well with accuracy comparable to that of SVM. This is obviously intuitive to understand and also since in Equation 36, we have included different slack variables for target and outlier classes, it is easier to understand that the optimization takes into account both outliers and target points.

Table 4: SVM and SVDD for classification: Time (in seconds) comparison of Kernelized versions on moons dataset

		RBF kernel(γ=0.7)	**Polynomial kernel (degree=2)**
Sample size: 50	SVM	6.55e-5	0.0007
	SVDD	0.037	0.041
Sample size: 100	SVM	6.38e-5	0.0008
	SVDD	0.129	0.1561
Sample size: 500	SVM	8.53e-5	0.002
	SVDD	5.49	2.04

6 Conclusion and Recommendation

In conclusion, we can see clearly that kernel methods are very powerful when it comes to learn non-linear pattern. They should be employed in cases where data is linearly separable and where traditional linear algorithms do not succeed. We have demonstrated the results on a variety of datasets and the performance, in general, gets better with the kernelized version of discussed algorithms.

However, this comes with the price of complexity (in time and space). Additionally, the difficulties in choosing the right kernel and tune its hyperparameters. *e.g.* example, see Figure 10. Here polynomial Kernel K-means performs worse than K-means weareas on the same data RBF Kernel performed much better. See Figure

To that goal, it would be interested to implement a parameter-tuning algorithm as Grid Search for example, or elbow test for Kernel K-means. Moreover, the methods implemented in this report follows scikit-learn conventions, hence, they could be easily adapted to the library. However, there is much work needed in optimizing the methods since some time they are performing poorly compared to already existing scikit-learn methods, e.g. See Figure 22.

References

[1] Shun-ichi Amari and Si Wu. Improving support vector machine classifiers by modifying kernel functions. *Neural Networks*, 12(6):783–789, 1999.

[2] César Souza. Kernel functions for machine learning applications, Mar 2010.

[3] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[4] Ronald A Fisher and Michael Marshall. Iris data set. *UC Irvine Machine Learning Repository*, 440, 1936.

[5] Rafail Ostrovsky, Yuval Rabani, Leonard J Schulman, and Chaitanya Swamy. The effectiveness of lloyd-type methods for the k-means problem. *Journal of the ACM (JACM)*, 59(6):28, 2012.

[6] Radha Chitta, Rong Jin, Timothy C Havens, and Anil K Jain. Approximate kernel k-means: Solution to large scale kernel clustering. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 895–903. ACM, 2011.

[7] D Tax, Alexander Ypma, and R Duin. Support vector data description applied to machine vibration analysis. In *Proc. 5th Annual Conference of the Advanced School for Computing and Imaging*, volume 54, pages 15–23. Citeseer, 1999.

[8] David MJ Tax and Robert PW Duin. Support vector data description. *Machine learning*, 54(1):45–66, 2004.

[9] David Martinus Johannes Tax. One-class classification: Concept learning in the absence of counter-examples. 2002.

[10] Donald Goldfarb and Ashok Idnani. A numerically stable dual method for solving strictly convex quadratic programs. *Mathematical programming*, 27(1):1–33, 1983.

[11] Hansi Jiang, Haoyu Wang, Wenhao Hu, Deovrat Kakde, and Arin Chaudhuri. Fast incremental svdd learning algorithm with the gaussian kernel. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 3991–3998, 2019.

[12] Alexandre Gramfort and Albert Thomas. Comparing anomaly detection algorithms for outlier detection on toy datasets. https://scikit-learn.org/stable/auto$_e$*xamples*/*plot*$_a$*nomaly*$_c$*omparison.htmlsphx* − *glr* − *auto* − *examples* − *plot* − *anomaly* − *comparison* − *py. Accessed* : 2019 − 10 − 24.

[13] Bernhard Schölkopf, Robert C Williamson, Alex J Smola, John Shawe-Taylor, and John C Platt. Support vector method for novelty detection. In *Advances in neural information processing systems*, pages 582–588, 2000.