A Support Vector Machine Classifier - KPCA + LDA Implementation and Application On An Image Multi-class Classification Task

Christos Psychalas
Contact: chrispsyc@yahoo.com

30 December, 2023

Abstract

Support Vector Machines, or SVMs for short, is a supervised machine learning algorithm, based on statistical learning frameworks, used both for classification and regression tasks. It was developed by Vladimir Vapnik and Alexey Chervonenkis back in the 1964 [1]. V. Vapnik and his colleagues in 1992 [2] expanded the classical SVM to include nonlinear classification problems by introducing the kernel trick. It is a robust algorithm that can be used for multi-class classification problems as well, using the "one vs all" or the "one vs one" techniques.

1 Introduction

In this first part of the assignment we focus on implementing a Support Vector Machine Classifier from scratch, using a Quadratic Programming solver, based on a Python library called "cvxopt" [3]. We also implement some of the most basic kernels used with SVMs, such as Radial Basis Function kernel [4], sigmoid [5] and polynomial [6] kernels. After the implementation we proceed with applying our SVM Classifier on the cifar-10 [7] dataset[1]. This dataset includes 10 different classes, such as aeroplanes, cars, trucks, etc. We choose to keep only three classes out of ten, which are the 'cat', 'deer' and 'horse' classes, to present the application of SVMs on a multi-classification problem. There are two approaches here, the "one vs all" and the "one vs one". For our example we chose the first one, constructing three different classifiers and finally combine them to have the predicted result among our three classes. First, we try to tune each classifier separately to achieve the best possible score, based on a validation set. After finding the best parameters for our classifiers we try to fine tune their combination. Finally, the results we get after the combination of the three do not coincide with the scores achieved by each classifier separately. More specifically, we notice that the final result is $\sim$10% lower than that of each individual classifier. As we explain in the *Conclusions* section, this is a clear overfitting case and we propose some solutions to deal with it. In the second part we implement some dimensionality reduction algorithms that will help us deal with the overfitting situation. In particular, we begin by implementing a simple PCA (Principal Components Analysis) algorithm, a variance based unsupervised method for dimensionality reduction, which we later enrich by using the kernel trick, as in the SVM case, to form the Kernel PCA. We also implement another dimensionality reduction method that is supervised, taking into consideration the different classes, trying to separate them as much as possible while keeping the samples of each class as closer as possible. This is the LDA (Linear Discriminant Analysis) method. Finally, we apply these two methods, the KPCA and the LDA as a combination, on our original problem and try to solve the overfitting.

[1]Downloaded from https://www.cs.toronto.edu/~kriz/cifar.html

Part I
SVMs

2 SVM Implementation

2.1 Problem Definition

The problem is as follows: We have a set of examples which belong to two classes, either '+' or '-' class. This is a binary classification problem. So what we need is a separator that splits the data in two based on their class value, assuming that the data are separable. There can be many separators, or hyperplanes as we say, that can divide them, but the question that lies is which one is the optimal. Intuitively, we would choose the hyperplane that gives us the maximum possible distance from the points closest to the hyperplane. Achieving this, our classifier becomes more resistant to noise, as our measurements may include errors. So, the problem's definition is to find the hyperplane with the maximum margin, where margin is the distance from the closest samples.

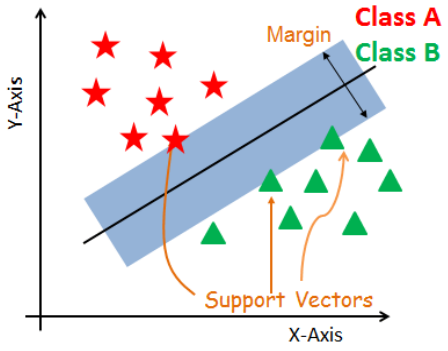


Figure 1: An SVM classifier (source: https://www.analyticsvidhya.com/blog/2021/06/build-an-image-classifier-with-svm/).

Let's continue with the mathematical formulation of our problem. The equation of a hyperplane (just a line in 2-D space) is given as follows [8]:

$$H = \left\{ \boldsymbol{x} : \boldsymbol{w}^T \cdot \boldsymbol{x} + b = 0 \right\}, \tag{1}$$

where $\boldsymbol{x}, \boldsymbol{w} \in \mathcal{R}^d$ and $b \in \mathcal{R}$. The dimension d is the dimension of the feature space. We are in search for the hyperplane with the maximum margin. Once found, the classifier takes the form: $g(\boldsymbol{x}) = sign(\boldsymbol{w}^T \cdot \boldsymbol{x} + b)$ and either predicts $+1$ or -1, but to reach this solution we need to find an equation for the margin in order to maximize it.

We get an arbitrary example of the samples, let that be x_i, and find its distance from the hyperplane H. Let that distance be d, as can be seen in Figure 2. It can be shown that the vectors $\vec{w}$ [2] and $\vec{d}$ are parallel. Then the distance d can be written as:

$$\boldsymbol{d} = \lambda \boldsymbol{w} \tag{2}$$

The norm of $\boldsymbol{d}$ is $||\boldsymbol{d}|| = \sqrt{\boldsymbol{d}^T \cdot \boldsymbol{d}} = \sqrt{\lambda \boldsymbol{w}^T \cdot \lambda \boldsymbol{w}}$, by substituting equation (2). Also let $\boldsymbol{x}_p$ be the projection of $\boldsymbol{x}_i$ on the hyperplane, thus the $\boldsymbol{x}_p$ satisfies equation (1):

$$\boldsymbol{w}^T \cdot \boldsymbol{x}_p + b = 0 \tag{3}$$

If we substitute $\boldsymbol{x}_p$ with $\boldsymbol{x}_p + \boldsymbol{d} = \boldsymbol{x}_i \rightarrow \boldsymbol{x}_p = \boldsymbol{x}_i - \boldsymbol{d}$[3] in equation (3), then we get:

$$\boldsymbol{w}^T \cdot (\boldsymbol{x}_i - \boldsymbol{d}) + b = 0 \xrightarrow{eq.\ (2)} \boldsymbol{w}^T \cdot (\boldsymbol{x}_i - \lambda \boldsymbol{w}) + b = 0 \rightarrow \lambda = \frac{\boldsymbol{w}^T \cdot \boldsymbol{x}_i + b}{\boldsymbol{w}^T \cdot \boldsymbol{w}} \tag{4}$$

[2] We will use the bold writing for vectors and the vector symbol as unit vector

[3] This computations refer to vectors with starting point on the origin and endpoints on the points shown in Figure 2

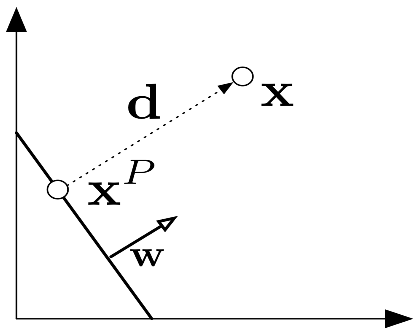


Figure 2: Distance of sample from hyperplane (source: [8]).

Now, *eq.* (2) $\xrightarrow{eq.\ (4)}$ $||\boldsymbol{d}|| = \sqrt{\frac{(\boldsymbol{w}^T\cdot\boldsymbol{x}_i+b)^2}{\boldsymbol{w}^T\cdot\boldsymbol{w}}\boldsymbol{w}^T\cdot\boldsymbol{w}} = \frac{|\boldsymbol{w}^T\cdot\boldsymbol{x}_i+b|}{\boldsymbol{w}^T\cdot\boldsymbol{w}}$. We have the advantage of making any choice about the $|\boldsymbol{w}^T \cdot \boldsymbol{x}_i + b|$ and make it as large or as small we want. It is obvious that we choose it to be equal to one, in order to simplify calculations. So, now we have:

$$D(\boldsymbol{w}, b) = min_{x\in D}\left(\frac{1}{\boldsymbol{w}^T\cdot\boldsymbol{w}}\right)^{4} \tag{5}$$

The above equation means that the margin is the distance from the closest points in both classes, thus the $min_{x\in D}$ (notice how the parameter b now is appearing in the constraints, eq. (6)) and our goal now is to maximize $D(\boldsymbol{w}, b)$. But there is a problem with this. We don't want any hyperplane whose $D(\boldsymbol{w}, b)$ is maximum, as there are many such hyperplanes reaching infinity. We need the hyperplane with the maximum margin that lies between the two classes. To achieve that we impose the following constraints:

$$\begin{aligned} \boldsymbol{w}^T\cdot\boldsymbol{x}_i + b \geq +1, \quad for \quad y_i = +1 \\ \boldsymbol{w}^T\cdot\boldsymbol{x}_i + b \leq -1, \quad for \quad y_i = -1, \end{aligned} \tag{6}$$

where $y \in \{-1, +1\}$ are the labels of our samples that either belong to class a $\{-1\}$ or class b $\{+1\}$. The above constraints (notice that because of the regularization of the $|\boldsymbol{w}^T \cdot \boldsymbol{x}_i + b|$ to one our inequalities have one as their bounds) make sure that the points of one class is on one side of the hyperplane and the points of the other class lie on the other side. The equations in (6) can be combined and yield one final constraint:

$$y_i(\boldsymbol{w}^T\cdot\boldsymbol{x}_i + b) \geq 1 \tag{7}$$

Now we get the final problem formulation that needs solution:

$$max\,(D(\boldsymbol{w}, \boldsymbol{x}, b)) = max\left(\frac{1}{\boldsymbol{w}^T\cdot\boldsymbol{w}}\right), \quad \forall i \quad y_i(\boldsymbol{w}^T\cdot\boldsymbol{x}_i + b) \geq 1 \tag{8}$$

2.2 General SVM Solution

2.2.1 Soft Margin Classifier

The problem stated above, that is to find the hyperplane that separates our classes with the maximum margin is applicable only for linear separable data. What can we do when the data are noisy, that is we have points of one class mixed with points of the other class. These noisy data can be still linearly separable, if we give our classifier a bit of a slack. This means that we allow our classifier to have some misclassifications. In this case the formulation written above can still be applied, if we add some extra variable that regulates how many such misclassifications we allow. Let the errors of misclassification

[4] As we defined the margin there should be a factor of 2, because we measure it for both classes. But this factor makes no difference to the nature of the problem so it is omitted.

be ϵ_i, as can be seen in Figure 3. Now we can rewrite our problem in such a way that allows some errors to occur:

$$max\left(\frac{1}{\boldsymbol{w}^T\cdot\boldsymbol{w}}\right)+C\sum_{i=1}^{k}\epsilon_i,\quad\forall i\quad\begin{cases}y_i(\boldsymbol{w}^T\cdot\boldsymbol{x}_i+b)\geq 1-\epsilon_i\\ \epsilon_i\geq 0\end{cases}\tag{9}$$

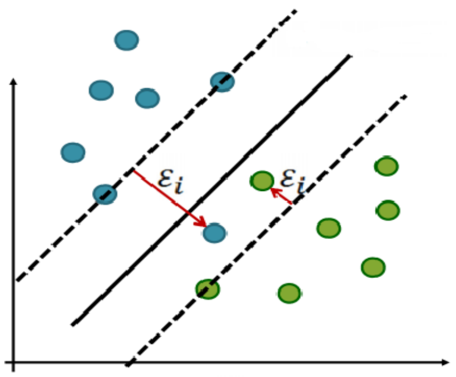


Figure 3: An SVM classifier with some errors (source: https://www.quora.com/What-are-the-objective-functions-of-hard-margin-and-soft-margin-SVM).

The C coefficient is a regularizer (penalizes) of the permitted errors. This is a hyperparameter that is chosen by us, depending on the problem. As the number C gets lower we allow more mistakes, making our classifier softer, while larger values tend to make our classifier more rigid. The first problem formulation is called 'Hard Margin', while the one we rewrote adding the slack is called 'Soft Margin'. Now it depends on the value of C.

2.2.2 Lagrange Formulation

So with this new formulation we can have a classifier for linearly separable data, as in the beginning, but also a classifier for noisy data. But we are not done yet. There is one last case left to handle. What happens if our data are not linearly separable, even with very low values of C we cannot manage to have the desired result. In this case we need a new formulation that would be able to allow us to raise our data to higher dimensions where they become separable. For this purpose we will use the Lagrange formulation [9]. But first of all let's rewrite our problem in the form of minimization rather than maximization:

$$min\left(\boldsymbol{w}^T\cdot\boldsymbol{w}\right)+C\sum_{i=1}^{k}\epsilon_i,\quad\forall i\quad\begin{cases}y_i(\boldsymbol{w}^T\cdot\boldsymbol{x}_i+b)\geq 1-\epsilon_i\\ \epsilon_i\geq 0\end{cases}\tag{10}$$

To formulate the Lagrange equivalent we add a Lagrange multiplier for each constraint. For our case we have two constraints, so we will be using the multipliers a_i and μ_i, where $a_i, \mu_i \geq 0$. The Lagrangian formulation is then:

$$\mathcal{L}(\boldsymbol{w},\boldsymbol{\epsilon},\boldsymbol{a},\boldsymbol{\mu},b)=\boldsymbol{w}^T\cdot\boldsymbol{w}+C\sum_{i=1}^{N}\epsilon_i-\sum_{i=1}^{N}a_i(y_i(\boldsymbol{w}^T\cdot\boldsymbol{x}_i+b)+\epsilon_i-1)-\sum_{i=1}^{N}\mu_i\epsilon_i,\tag{11}$$

where N is the number of samples. This is called the primal formulation [5] of the Lagrangian and the new goal now is:

$$min_{\boldsymbol{w},b,\boldsymbol{\epsilon}}\left(max_{\boldsymbol{a},\boldsymbol{\mu}\geq 0}\left(\mathcal{L}(\boldsymbol{w},\boldsymbol{\epsilon},\boldsymbol{a},\boldsymbol{\mu},b)\right)\right)\tag{12}$$

The dual equivalent is the maximum of the minimum, or:

$$max_{\boldsymbol{a},\boldsymbol{\mu}\geq 0}\left(min_{\boldsymbol{w},b,\boldsymbol{\epsilon}}\left(\mathcal{L}(\boldsymbol{w},\boldsymbol{\epsilon},\boldsymbol{a},\boldsymbol{\mu},b)\right)\right)\tag{13}$$

[5]For our convex problem to be feasible the Karush-Kühn-Tucker must apply [10], which is the case

This is easier as now we can take the derivatives to zero and replace into the initial equation (11) making it an equation with variables a, μ which will be in a form compliant with the one required by the Quadratic Programming optimizer. So, in order to find the minimum we must differentiate [6] with respect to $\boldsymbol{w}, \boldsymbol{\epsilon}$ and b:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{w}} = 0 \rightarrow \boldsymbol{w} = \sum_{i=1}^{N} a_i y_i \boldsymbol{x}_i \tag{14}$$

$$\frac{\partial \mathcal{L}}{\partial b} = 0 \rightarrow \sum_{i=1}^{N} a_i y_i = 0 \tag{15}$$

$$\frac{\partial \mathcal{L}}{\partial \epsilon} = 0 \rightarrow \sum_{i=1}^{N} (a_i + y_i) = C \sum_{i=1}^{N} 1 \tag{16}$$

Inserting equations (14), (15), (16) into equation (11), we obtain:

$$\mathcal{L}(\boldsymbol{a}) = \sum_{i=1}^{N} a_i - \sum_{i=1}^{N}\sum_{j=1}^{N} a_i y_i a_j y_j \boldsymbol{x}_i \boldsymbol{x}_j^T \quad subject\ to \begin{cases} \boldsymbol{a}, \boldsymbol{\mu} \geq 0 \\ \boldsymbol{a} + \boldsymbol{\mu} = \boldsymbol{C} \quad (from\ eq.\ (16)) \\ \boldsymbol{y}^T \cdot \boldsymbol{a} = 0 \quad (from\ eq.\ (15)) \end{cases} \tag{17}$$

As can be seen in eq. (17) out of the two parameters $\boldsymbol{a}, \boldsymbol{\mu}$ only $\boldsymbol{a}$ appears in the final form. But, the second Lagrange multiplier $\boldsymbol{\mu}$ is hidden in the constraints. Combining the first two constraints [11] we arrive at the final dual solution:

$$\mathcal{L}(\boldsymbol{a}) = \sum_{i=1}^{N} a_i - \sum_{i=1}^{N}\sum_{j=1}^{N} a_i y_i a_j y_j \boldsymbol{x}_i \boldsymbol{x}_j^T \quad subject\ to \begin{cases} 0 \leq \boldsymbol{a} \leq \boldsymbol{C} \\ \boldsymbol{y}^T \cdot \boldsymbol{a} = 0 \end{cases} \tag{18}$$

If you remember we had to maximize the minimization. After substituting equations (14), (15), (16) into equation (11), we received eq. (18), which we must now find its maximum. This will be done using the Python library [3], which will yield all the a multipliers. We will expand this later on subsection 'Solving Using CVXOPT', but for now let's say we solved it and got our unknown a variables. The final goal is to find the optimal hyperplane H and to find it we must specify the weights, $\boldsymbol{w}$ and the bias b. So, knowing alphas, we substitute on eq. (14) and we get our $\boldsymbol{w}$. But there is no equation for the beta. We can find it implicitly by substituting a value of $\boldsymbol{x}$ in eq. (7), after weights are known. If we take a closer look on the constraints on eq. (18) we will notice that the solution is based on the constraint that alphas are greater than 0 and lower than the hyperparameter C. These are special points called support vectors and after finding them we don't need any other of our points in the dataset. We needed them to calculate and find the points closer to the hyperplane, but now only the closest ones participate in the final solution. Finally to obtain b [10], substituting the value of a support vector along with its label on eq. (7), we obtain:

$$\begin{gathered} y_s(\boldsymbol{w}^T \cdot \boldsymbol{x}_s + b) = 1 \xrightarrow{eq.\ (14)} y_s(\sum_{i=1}^{S} a_i y_i \boldsymbol{x}_i^T \boldsymbol{x}_s + b) = 1 \rightarrow b = \frac{1}{y_s} - \sum_{i=1}^{S} y_i a_i \boldsymbol{x}_i^T \boldsymbol{x}_s \\ \xrightarrow{y \in [-1,1]} b = y_s - \sum_{i=1}^{S} y_i a_i \boldsymbol{x}_i^T \boldsymbol{x}_s, \end{gathered} \tag{19}$$

where $\boldsymbol{x}_s$ is the support vector and y_s is its label. The S in the summation limit is the number of support vectors. After all this cumbersome process, having acquired both $\boldsymbol{w}$ and b, we are able to write the function of the classifier that is able to handle linear separable, noisy but also non linear separable data (after substituting a kernel function as discussed below). We have:

$$g(\boldsymbol{x}) = sign\left(\boldsymbol{w}^T \cdot \boldsymbol{x} + b\right) = sign\left(\sum_{i=1}^{S} a_i y_i \boldsymbol{x}_i^T \boldsymbol{x} + y_s - \sum_{i=1}^{S} y_i a_i \boldsymbol{x}_i^T \boldsymbol{x}_s\right) \tag{20}$$

[6]Full equation solutions can be found on *Appendix A.2 Equations*

2.3 The Kernel Trick

In equation (18) we had the inner products $\boldsymbol{x}_i \cdot \boldsymbol{x}_j^T$. For the case of non linearly separable data we mentioned that we need to move our data to higher dimensions. It's this formulation of the Lagrangian that serves us to replace these dot products with the products in higher dimensions, as computed by the kernel function. Let $\Phi : \mathcal{R}^d \rightarrow \mathcal{R}^{\tilde{d}}$ be the transformation that takes our original data from feature space d to feature space $\tilde{d}$. Then the transformed data would be $\boldsymbol{z} = \Phi(\boldsymbol{x})$ and we would have to substitute $\boldsymbol{x}$ in our previous solution on equation (20) with $\boldsymbol{z}$ and solve in the $\tilde{d}$ space. This would be very computational consuming as data dimensions grow bigger. The kernel trick help us take the required products as if data were transformed, but without the need to apply computations on the $\tilde{d}$ space. So the kernel is the equivalent of the dot products in the higher dimensions:

$$K(\boldsymbol{x}, \boldsymbol{x}^*) \equiv \Phi(\boldsymbol{x})^T \cdot \Phi(\boldsymbol{x}^*)$$

If we substitute K on equation (20) we get a general solution where K can be any kernel:

$$g(\boldsymbol{x}) = sign\left(\boldsymbol{w}^T \cdot \boldsymbol{x} + b\right) = sign\left(\sum_{i=1}^{S} a_i y_i K(\boldsymbol{x}_i, \boldsymbol{x}) + y_s - \sum_{i=1}^{S} y_i a_i K(\boldsymbol{x}_i, \boldsymbol{x}_s)\right) \tag{21}$$

K is also called the *Gram Matrix* and is an $N \times N$ matrix.

In this implementation three basic kernels are used: the Radial Basis Function kernel (or rbf), the Polynomial and the Sigmoid kernels.

- Polynomial Kernel [12]: $K(\boldsymbol{x}_i, \boldsymbol{x}_j) = (\boldsymbol{x}_i^T \cdot \boldsymbol{x}_j + c)^d$, where d is the degree of the polynomial and c is a coefficient that controls the fit of the data in respect with the size of the margin. Large c values give lower misclassifications, but may result in overfitting. The degree controls our SVM's complexity. The more complex a model is the more prone is to overfitting.

- Sigmoid Kernel [13]: $K(\boldsymbol{x}_i, \boldsymbol{x}_j) = \tanh(\gamma\, \boldsymbol{x}_i^T \cdot \boldsymbol{x}_j + c)$, where γ acts as a scaling factor for the data and c is an offset coefficient. Higher values of c tend to dominate the product result. As kernels can be viewed also as similarity measure, the c parameter here can be seen as a bias that favors one class over the other (for higher values).

- RBF Kernel [14]: $K(\boldsymbol{x}_i, \boldsymbol{x}_j) = \exp(-\gamma\, ||\boldsymbol{x}_i - \boldsymbol{x}_j||^2)$. Expanding it we get:

  $$K(\boldsymbol{x}_i, \boldsymbol{x}_j) = \exp(-\gamma\,(||\boldsymbol{x}_i||^2 + ||\boldsymbol{x}_j||^2) - 2\boldsymbol{x}_i \cdot \boldsymbol{x}_j)$$

  The γ factor here refers to the similarity of samples. The greater the value the more each sample is considered as an individual case with no similarity with other samples. The higher the γ values the more prone we are in overfitting.

2.4 Solving Using CVXOPT

After introducing the kernel general solution as described in equation (21), we had taken the solution of finding the alpha multipliers for granted. To get the alphas we have to use a Quadratic Programming solver to find the optimum solution in our maximization problem ([15], [4]), as described with equations (13) and (18). The Python's library CVXOPT will give us the solution when we provide the $\boldsymbol{x}, y$ and C. But it requires that our problem has the following format:

$$minimize \quad (1/2)\boldsymbol{x}^T P \boldsymbol{x} + \boldsymbol{q}^T \boldsymbol{x} \quad subject\ to \begin{cases} \boldsymbol{G}\boldsymbol{x} \leq \boldsymbol{h} \\ \boldsymbol{A}\boldsymbol{x} = b \end{cases}$$

First of all we need to reformulate equation (18), so the maximization becomes a minimization. To do so, all we have to do is multiply $\mathcal{L}(\boldsymbol{a})$ and the constraints by -1. Now (18), becomes:

$$\mathcal{L}(\boldsymbol{a}) = \sum_{i=1}^{N}\sum_{j=1}^{N} a_i y_i a_j y_j \boldsymbol{x}_i \boldsymbol{x}_j^T - \sum_{i=1}^{N} a_i \quad subject\ to \begin{cases} -\boldsymbol{C} \leq -\boldsymbol{a} \leq 0 \;\; or \;\; -\boldsymbol{a} \leq 0 \; and \; \boldsymbol{a} \leq \boldsymbol{C} \\ \boldsymbol{y}^T \cdot \boldsymbol{a} = 0 \end{cases} \tag{22}$$

In our case the $\boldsymbol{x}$ variable is the alpha multipliers $\boldsymbol{a}$ and the P matrix, based on equation (22), is:

$$P = \sum_{i=1}^{N}\sum_{j=1}^{N} y_i y_j \boldsymbol{x}_i \boldsymbol{x}_j^T = \sum_{i=1}^{N}\sum_{j=1}^{N} y_i y_j K(\boldsymbol{x}_i, \boldsymbol{x}_j)$$

Again from equation (22), $\boldsymbol{q} = -\sum_{i=1}^{N} 1$. Regarding the constraints for the first one we have two matrices that make the $\boldsymbol{G}$ and two that make the $\boldsymbol{h}$:

$$\begin{cases} \boldsymbol{G}_{(2N\times N)} = \begin{bmatrix} \boldsymbol{g1} \\ \boldsymbol{g2} \end{bmatrix}, \textit{where} \begin{cases} \boldsymbol{g1} = -\boldsymbol{1}_{(N\times N)} \\ \boldsymbol{g2} = \boldsymbol{1}_{(N\times N)} \end{cases} \\ \\ \boldsymbol{h}_{(2N)} = \begin{bmatrix} \boldsymbol{h1} \\ \boldsymbol{h2} \end{bmatrix}, \textit{where} \begin{cases} \boldsymbol{h1} = \boldsymbol{0}_{(N)} \\ \boldsymbol{h2} = \boldsymbol{C}_{(N)} \end{cases} \end{cases}$$

Finally, what is left is the $\boldsymbol{A}$ matrix and the b. We derive them from the second constraint, as seen in equation (22):

$$\begin{cases} \boldsymbol{A}_{(1\times N)} = \boldsymbol{y}^T \\ b = 0 \end{cases}$$

After all parameters are set we can import them in the QP (Quadratic Programming) solver and yield the solution that gives us back the alphas. Remember though, as we mentioned earlier the final solution for our hyperplane depends only on the support vectors. To get the support vectors from our QP solution, we have to set some conditions on the alphas, which are not other than the first constraint in equation (18). Here we can have two types of support vectors, the so called *'free support vectors'* and the *'bounded support vectors'*. The first ones comply with this constraint: $0 < a_i < C$, while the others are the ones for which $a_i = C$. The free support vectors are guaranteed to be on the boundary of the hyperplane, while the bounded support vectors can be on the boundary, slightly violating it but still correctly predicted, or seriously violating the boundary and be erroneously predicted [10].

3 Application

In this second part we test our implementation of SVM Classifier on an image dataset, the cifar-10, and construct a three class predictor based on the 'one vs all' approach. The code can be accessed via GitHub, `https://github.com/Xritsos/SVM-from-scratch-.git`. We experiment on tuning the parameters of our classifiers and then test their combination on the test set for the final results. The dataset consists of 60000 pictures of size 32×32 with 3 channels, Red, Green, Blue, split into 6 batches of data containing 10000 pictures each. The first five batches are for training purposes and the last one is the test batch. The pictures in the dataset correspond to 10 different classes, such as aeroplane, truck, car, etc. For this application we choose only 3 classes for demonstration purposes. The target classes for our problem are:

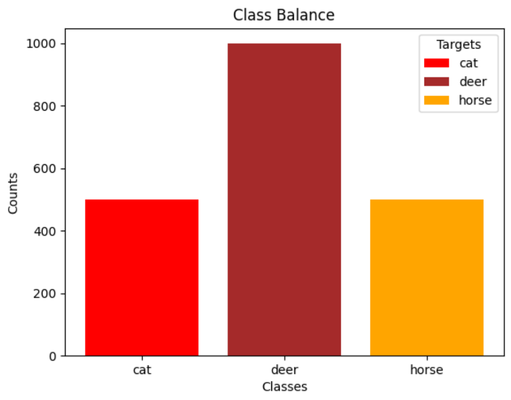
- 'Cat' with label number 3
- 'Deer' with label number 4
- 'Horse' with label number 7

For training our classifiers we keep batches 1, 2 and 3 for each classifier respectively and use batch 5 for validation. The test batch remains untouched in order to evaluate our final classifier.

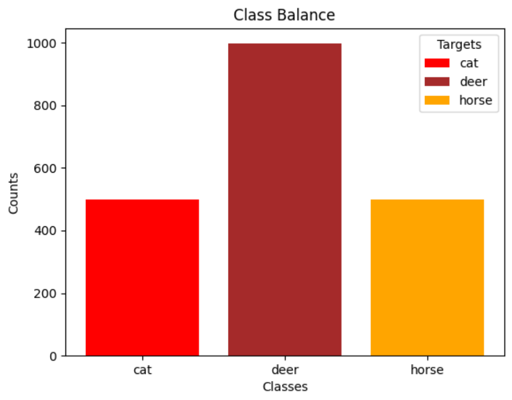
3.1 Deer Classifier

First we choose a random class to start with. Let that be the 'Deer' class. In order to use the 'one vs all' approach, we first merge the two other classes, 'Cat' and 'Horse'. Now our labels become $y \in [-1, 1]$, where 1 corresponds to the target class, for this case 'Deer', and the -1 corresponds to the other two merged classes. Each batch of data contains approximately an equal number of samples, ~ 1000. In order our classifier to be able to predict both cases, we subsample from the merged classes so there is an equal number of samples for the two classes, as can be seen in Figure 4. Our train samples and labels are in the form: x_train = (n_samples, n_features), y_train = (n_samples, 1) or with numbers:

- x_train = (1997, 3072)
- x_valid = (1993, 3072)



(a) Train Set



(b) Validation Set

Figure 4: Balance of classes for train and validation sets for the Deer Classifier

Regarding the data scaling we experiment on two different normalizations:

1. $\boldsymbol{x} \in [0, 1]$ and
2. $\boldsymbol{x} \in [-1, 1]$

First we fit the scaler on the training data and apply the transformations on training and test sets afterwards. Now we present our experiments for determining the best parameters for our first classifier.

Table 1: Test 1: Sigmoid kernel - Scaling $[0, 1]$

Run	C	γ	$coef$[7]	F1[8]	Precision[9]	Recall	Best Params.
1	$[0.01, 0.1, 1, 10]$	$[0.001, 0.01, 0.1, 0.5, 1]$	$[0, 1]$	61%	61%	61%	$C = 1, \gamma = 0.001, coef = 0$
2	$[0.5, 1, 2, 3]$	$[0.0007, 0.001, 0.005]$	$[-0.1, 0, 0.1]$	63%	63%	63%	$C = 2, \gamma = 0.0007, coef = -0.1$
3	$[1.5, 2, 2.5]$	$[0.0004, 0.0007, 0.0008]$	$[-0.2, -0.1]$	64%	64%	64%	$C = 2, \gamma = 0.0004, coef = -0.2$
4	$[2]$	$[0.0002, 0.0004, 0.0005]$	$[-0.3, -0.2]$	64%	64%	64%	$C = 2, \gamma = 0.0004, coef = -0.2$

Table 2: Test 2: Sigmoid kernel - Scaling $[-1, 1]$

Run	C	γ	$coef$	F1	Precision	Recall	Best Params.
1	$[0.01, 0.1, 1, 10]$	$[0.001, 0.01, 0.1, 0.5, 1]$	$[0, 1]$	58%	58%	58%	$C = 10, \gamma = 0.01, coef = 0$
2	$[2, 5, 10, 15]$	$[0.005, 0.01, 0.05]$	$[-0.1, 0, 0.1]$	59%	59%	59%	$C = 10, \gamma = 0.05, coef = 0.1$
3	$[9, 10, 11]$	$[0.04, 0.05, 0.06]$	$[0.08, 0.1, 0.5]$	59%	59%	59%	$C = 10, \gamma = 0.04, coef = 0.08$
4	$[10]$	$[0.02, 0.03, 0.04]$	$[0.04, 0.05, 0.07, 0.08]$	59%	59%	59%	$C = 10, \gamma = 0.04, coef = 0.05$

Table 3: Test 3: Polynomial kernel - Scaling $[0, 1]$

Run	C	$degree$	$coef$	F1	Precision	Recall	Best Params.
1	$[0.01, 0.1, 1, 10]$	$[2, 3, 4]$	$[0, 1]$	46%	46%	46%	$C = 10, degree = 2, coef = 0$
2	$[5, 10, 100]$	$[2, 4, 8]$	$[0.1, 0, 1]$	55%	55%	55%	$C = 100, degree = 8, coef = 1$
3	$[90, 100, 110]$	$[7, 8, 10, 15, 20]$	$[1, 2, 4, 6]$	56%	56%	56%	$C = 110, degree = 15, coef = 1$

[7]coef is the c as mentioned in kernel section
[8]All scores are computed as macro, that is the average for the two classes
[9]All scores are rounded to 2 decimals

Table 4: Test 4: Polynomial kernel - Scaling $[-1, 1]$

Run	C	$degree$	$coef$	F1	Precision	Recall	Best Params.
1	$[0.1, 1, 10, 100]$	$[2, 4]$	$[-0.1, 0, 0.1, 1]$	47%	54%	52%	$C = 10, degree = 2, coef = -0.1$
2	$[5, 20, 50]$	$[10, 20]$	$[-1, -0.5, -0.007]$	35%	53%	50%	$C = 50, degree = 10, coef = -0.007$

Table 5: Test 5: RBF kernel - Scaling $[0, 1]$

Run	C	γ	F1	Precision	Recall	Best Params.
1	$[0.1, 1, 10, 100]$	$[0.001, 0.01, 0.1, 0.5, 1]$	75%	75%	75%	$C = 10, \gamma = 0.01$
2	$[5, 10, 20, 50]$	$[0.007, 0.02, 0.03]$	75%	75%	75%	$C = 10, \gamma = 0.02$
3	$[8, 10, 12]$	$[0.01, 0.015, 0.02]$	75%	75%	75%	$C = 12, \gamma = 0.02$
4	$[11, 12, 14, 16]$	$[0.02, 0.025]$	75%	75%	75%	$C = 12, \gamma = 0.02$

Table 6: Test 6: RBF kernel - Scaling $[-1, 1]$

Run	C	γ	F1	Precision	Recall	Best Params.
1	$[0.1, 1, 10, 100]$	$[0.001, 0.01, 0.1, 0.5, 1]$	75%	75%	75%	$C = 100, \gamma = 0.001$
2	$[90, 100, 120]$	$[0.0007, 0.001, 0.003]$	75%	75%	75%	$C = 100, \gamma = 0.001$

Let's make a recap about the best scores achieved by each trial:

Table 7: Best scores for each test

Test	F1	Precision	Recall
1	64%	64%	64%
2	59%	59%	59%
3	56%	56%	56%
4	47%	54%	52%
5	75%	75%	75%
6	75%	75%	75%

From Table 7 we notice that both tests 5 and 6 yield same scores. We choose the parameters from test 6 as the value of the γ parameter is lower, which makes it a little safer regarding overfitting. A lower value extends the similarity of samples, while a higher tends to regard each sample as separate. Also, we notice that the value of parameter C is higher, which as we mentioned earlier means that our classifier is more strict, permitting fewer misclassifications. These facts make the parameters from test 6 to be our best chosen ones for our Deer Classifier:

- Scaling: $[-1, 1]$
- Kernel: Radial Basis Function
- C: 100
- γ: 0.001

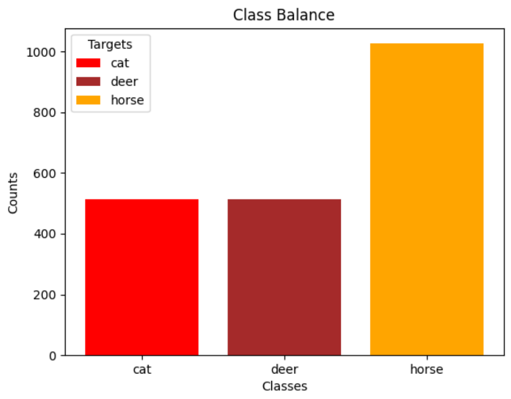
3.2 Horse and Cat Classifiers

For the remaining two classifiers we won't run tests from scratch. First, we try the best parameters as found in Deer Classifier's tuning. For the Horse Classifier, plugging the above parameters, we have:

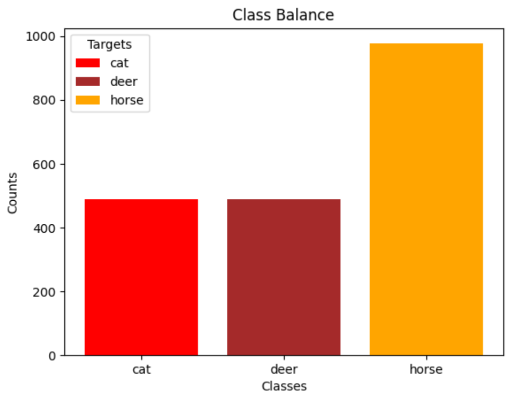
- x_train = (2052, 3072)
- x_valid = (1953, 3072)
- F1 = 71%

- Precision = 73%
- Recall = 71%

The train and validation sets classes can be seen below on Figure 5.



(a) Train Set



(b) Validation Set

Figure 5: Balance of classes for train and validation sets for the Horse Classifier

We proceed trying to tune it a little more running the following test:

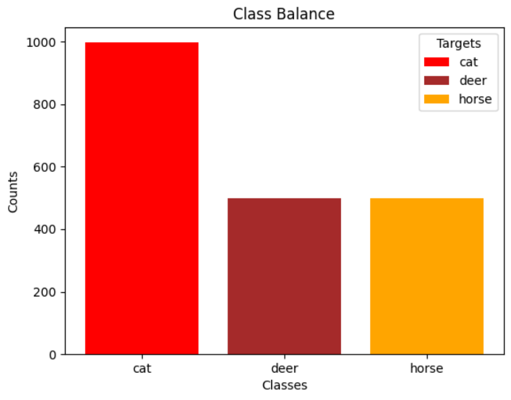
Table 8: RBF kernel - Scaling $[-1, 1]$

C	γ	F1	Precision	Recall	Best Params.
$[90, 100, 110]$	$[0.0007, 0.001, 0.003]$	74%	75%	74%	$C = 100, \gamma = 0.0007$

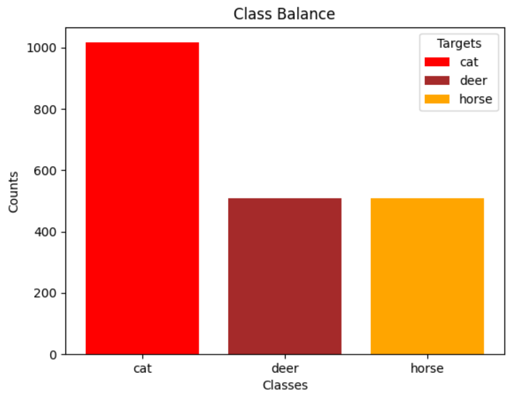
Now we achieved better results almost similar with the Deer Classifier. We continue with the last classifier:

- x_train = (1993, 3072)
- x_valid = (2032, 3072)
- F1 = 73%
- Precision = 76%
- Recall = 73%

The train and validation sets classes can be seen below on Figure 6.



(a) Train Set



(b) Validation Set

Figure 6: Balance of classes for train and validation sets for the Cat Classifier

We proceed trying to tune it a little more running the following test:

Table 9: RBF kernel - Scaling $[-1, 1]$

C	γ	F1	Precision	Recall	Best Params.
$[90, 100, 110]$	$[0.0007, 0.001, 0.003]$	75%	76%	75%	$C = 110, \gamma = 0.0007$

Finally, we have all parameters for all of three classifiers:

Table 10: Best Parameters per Classifier

Classifier	C	γ
Deer	100	0.001
Horse	100	0.0007
Cat	110	0.0007

3.3 Combining Classifiers

The next step is to plug in the best parameters found for each classifier and assess their combination on a new training and validation set. For this purpose out of the five batches of training data we pick the batch 5 as our train set and batch 4 for validation, as we try a few more tuning steps. But first of all let's discuss the voting process of our 'one vs all' approach. To get the final result first we train each classifier independently and then test them on the same test. We sum up all three results and come up with the following scenarios:

- Label $= 2$, which came up from deer classifier predicting 4 and the two others -1 and -1, thus it's clearly a deer.
- Label $= 5$, which with the same logic denotes that the test sample is a horse.
- Label $= 1$, which denotes that the result is a cat.
- Label $= 10$, both Deer and Horse classifiers predict at the same time. In this case we handle the conflict by assigning the result of the classifier with the highest F1 score.
- Label $= 9$, Cat - Horse conflict, managed as before.
- Label $= 6$, Deer - Cat conflict.
- Label $= 14$, all three classifiers conflict, which again is solved by choosing the best one, based on F1 score.

- Label $= -3$, none of the classifiers predict. In this case we choose the classifier with the lowest F1 score as the ones with higher should be able to predict it.

Now moving on with testing and tuning the classifier's combination:

- Deer Classifier: x_train = (1993, 3072)
- Horse Classifier: x_train = (1961, 3072)
- Cat Classifier: x_train = (1952, 3072)
- Test Set (batch 4): x_test = (2961, 3072)

For these sets and parameters from the previous subsection, we can see the results (macro scores) of each individual classifier in Table 11 and their combination in Table 12.

Table 11: Individual Classifiers' scores

Classifier	F1	Precision	Recall
Deer	73%	73.5%	73%
Horse	73%	74%	73%
Cat	76%	77%	76%

Table 12: Combined Classifier

F1	Precision	Recall
61%	64%	62%

Let's try to tweak the C parameter and see the results on the combined classifier. Below in the table we show the results. As C becomes large there's no noticeable difference in the results, but very low values lead to worse results.

Table 13: Combined Classifier C parameter experiments

C	F1	Precision	Recall
1000	60%	63%	61%
0.001	47%	50%	50%

Next, we keep the parameter $C = 100$ steady and try to experiment with the γ parameter and see if we can improve our scores on the combination.

Table 14: Combined Classifier γ parameter experiments

Classifier (tweak)	γ	F1	Precision	Recall
Horse, Cat	0.0003	61%	63%	62%
Horse, Cat	0.0005	62%	65%	63%
Cat	0.0008	59%	63%	60%
Deer	0.0008	62%	64%	63%

Looking at Table 14 we managed to insignificantly change our scores by 1% for these new parameters:

- Deer Classifier: $C = 100, \gamma = 0.001$
- Horse Classifier: $C = 100, \gamma = 0.0005$
- Cat Classifier: $C = 100, \gamma = 0.0005$

The final step is to train and combine our classifiers to more data. For this purpose we perform a training using 4 batches of data and perform the final test on the test batch of the cifar-10 dataset. From all five batches we fit the batches 2, 3, 4 and 5 (due to memory issues). This leads to the following training and test sets:

- Total Train Set: $(11984, 3072)$ - Total Test Set: $(3000, 3072)$
- Deer Train Set: $(8001, 3072)$
- Horse Train Set: $(7997, 3072)$
- Cat Train Set: $(7968, 3072)$

Below are the final test results of our 'one vs all' approach classifier.

Table 15: Combined Classifier Test

F1	Precision	Recall
69%	71%	69%

And these are the results of each individual classifier:

Table 16: Classifiers' scores on the final testing

Classifier	F1	Precision	Recall
Deer	77%	77%	77%
Horse	78%	78%	78%
Cat	79%	79%	79%

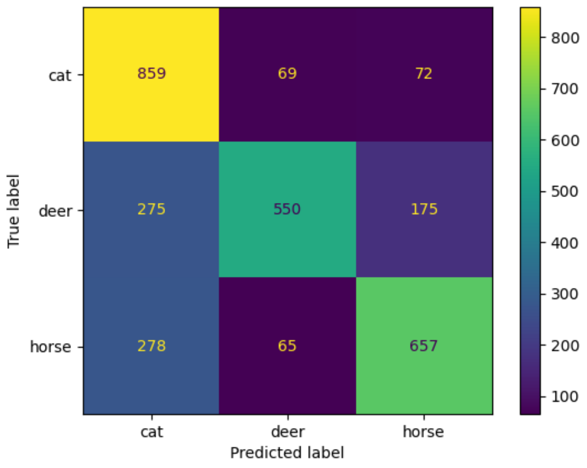


Figure 7: Confusion Matrix

Part II

Dimensionality Reduction

4 Principal Components Analysis

Reducing dimensions is a crucial part of machine learning algorithms' training. It is widely used as it leads to better generalization and less resource consumption. The first method is PCA, a simple and robust technique for dimensionality reduction, which focuses on maintaining the maximum amount of information possible. It is an old technique named with different titles, such as Karhunen-Loève transformation, the Hotelling transformation, the method of empirical orthogonal functions [16]. PCA can be used to reduce noise in the dataset as well. The method is unsupervised and transforms the data linearly by projecting them onto new dimensions, rather than just cropping the existing ones, which are not correlated with each other [17].

Our goal is to start with $\boldsymbol{x} \in \mathcal{R}^p$ and move to a lower dimensional (feature) space where $\boldsymbol{x} \in \mathcal{R}^q$, $q < p$. So starting from space p we end up projecting our data to the subspace q. To do so there are different alternatives. One way would be to minimize the distances of the points to the projected line, or an easier way, to maximize the distance (variance) of the projected points from the origin, as seen in Figure 8, below. Both ways are equivalent. To begin with the easier way first we must center our data, that is each feature has zero mean. If $\bar{\boldsymbol{x}} = \frac{1}{N}\sum_{i=1}^{N}\boldsymbol{x}_i$, is the mean then $\boldsymbol{x}_{centered} = \boldsymbol{x} - \bar{\boldsymbol{x}}.$, where N is the number of samples. This centering is used to derive the covariance matrix so from now on $\boldsymbol{x} = \boldsymbol{x}_{centered}$.

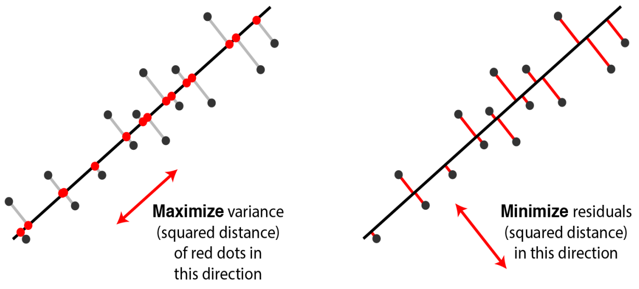


Figure 8: Two ways to PCA (source: http://alexhwilliams.info/itsneuronalblog/2016/03/27/pca/).

Let the variance on the projected hyperplane ($\boldsymbol{w}$) be $\sigma_w^2 = \frac{1}{N}\sum_{i=1}^{N}(\boldsymbol{x}_i \cdot \boldsymbol{w})^2$ (squared so positive and negative terms won't cancel out). Expanding we have:

$$\sigma_w^2 = \frac{1}{N}\sum_{i=1}^{N}(\boldsymbol{x}_i \cdot \boldsymbol{w})^2 = \frac{1}{N}(\boldsymbol{x} \cdot \boldsymbol{w})^T(\boldsymbol{x} \cdot \boldsymbol{w}) \quad = \boldsymbol{w}^T \cdot \frac{\boldsymbol{x}^T \cdot \boldsymbol{x}}{N} \cdot \boldsymbol{w} = \boldsymbol{w}^T \cdot \boldsymbol{v} \cdot \boldsymbol{w}, \tag{23}$$

where $\boldsymbol{v}$ is the covariance matrix. In order to maximize the above, one solution would be $w \to \infty$, but we are not interested in the magnitude of $\boldsymbol{w}$ rather in its direction. So we apply the constraint that $\boldsymbol{w}^T \cdot \boldsymbol{w} = 1$. One way to solve this, is using the same path as in the SVM problem and form its Lagrangian:

$$\mathcal{L}(\boldsymbol{w}, \lambda) = \sigma_w^2 - \lambda(\boldsymbol{w}^T \cdot \boldsymbol{w} - 1)$$

which ends up in the following: $\boldsymbol{v} \cdot \boldsymbol{w} = \lambda\boldsymbol{w}$, an eigen-decomposition problem, with $\boldsymbol{w}$ being the eigenvectors, or the principal components, and λ the corresponding eigenvalues. The first eigenvector (PC_1) is the vector whose σ_w^2 is maximum and each of the next components have the next maximum values in descending order. Moreover each of the components is perpendicular to the others. But there is another way to obtain the eigenvalues and the eigenvectors, the Singular Value Decomposition (SVD) method. According to SVD, for every matrix $X_{(N\times M)}$ there exists a factorization of the following form:

$$X_{(N\times M)} = U_{(N\times N)} \cdot \Sigma_{(N\times M)} \cdot V^T_{(M\times M)}, \tag{24}$$

where X is our centered (train) data. The matrices U and V are orthonomal, that is $U \cdot U^T, V \cdot V^T = I$ and the Σ matrix is diagonal. We should mention that the values of the Σ matrix (called singular values) are zero except for the m principal components and the number of these components cannot be more than the $min(samples, features) = min(N, M)$. So now we need to solve for U, V, Σ. First step, we exploit the matrices U and V that are orthonomal and compute the $X^T \cdot X$ and $X \cdot X^T$ [17]:

$$\begin{aligned} X^T \cdot X &= (U^T \cdot \Sigma^T \cdot V)(U \cdot \Sigma \cdot V^T) = V \cdot \Sigma \cdot U^T \cdot U \cdot \Sigma \cdot V^T = V \cdot \Sigma^2 \cdot V^T \\ X \cdot X^T &= (U \cdot \Sigma \cdot V^T)(U^T \cdot \Sigma^T \cdot V) = U \cdot \Sigma \cdot V^T \cdot V \cdot \Sigma \cdot U^T = U \cdot \Sigma^2 \cdot U^T \end{aligned} \tag{25}$$

From the above equation and eq. (24) we solve for U, V and Σ, this is the SVD method. Now if you look closely at eq. (25) and $\boldsymbol{v} \cdot \boldsymbol{w} = \lambda \boldsymbol{w}$ of the eigen-decomposition, they are the same (if we multiply both sides of eq. (25) with V. Then the values of V correspond to $\boldsymbol{w}$, that is the eigenvectors and the eigenvalues are equal to the values of Σ^2/N. Using the SVD method we obtain the values of the eigenvectors and eigenvalues for the covariance matrix. But why use the SVD method over the eigen-decomposition [18]:

- SVD can be used for any matrix of the form $N \times M$, while the eigen-decomposition (EVD) only applies on $M \times M$ matrices
- SVD is computationally effective, as for high dimensional data ($M >> N$) the covariance matrix computations are heavy
- The use of the covariance matrix affects the precision of the results
- Most of the SVD implementations use a more stable approach

After having calculated the eigenvectors, we project our data to them (dot product) and obtain the transformed data. We mention that except for the obtained eigenvectors it's crucial that we keep also the mean of the original (train) data, so we can transform any new data by centering them to this mean and projecting them to the yielded eigenvectors. A question that arises is how many of the components (eigenvectors) should we keep each time. There is a measure of information (variation) that can be applied to determine the maximum number of components. This information is given by the eigenvalues. Summing all eigenvalues we get the total variance. We keep so many components so as their variance summation over the total variance is $\sim 90 - 95\%$. This ratio is called explained variance.

5 Kernel PCA

The PCA method is a linear transformation method, that is the principal components derived are a linear combination of the original feature space [19]. Consequently, we can use it only on linearly separable data. But we faced the problem of non linearly separable data above, during the formulation of the SVM classifier. The application we chose, consists of such data (RGB images), and as we shown we can solve it by introducing the Kernel Trick [20] as mentioned in the corresponding section. Again we rise our data to higher dimensions (Hilbert Space) in order to find linear projections, that is our data become linearly separable. Starting from our data $\boldsymbol{x} \in \mathcal{R}^N$ we apply a transformation $\phi : \mathcal{R}^N \to \mathcal{R}^D$ and the new data become $z = \phi(\boldsymbol{x})$. As already mentioned above, in the SVM case, computations in such spaces are infeasible. If you recall we replaced such computations with the Gram Matrix, which is a $N \times N$ matrix. This is the case here as well. Instead of calculating the covariance matrix's eigenvalues and eigenvectors for $\boldsymbol{x}$'s transformation, $\phi(\boldsymbol{x}) \cdot \phi(\boldsymbol{x})^T$, we do it for the Gram Matrix $K(\boldsymbol{x}) = \phi(\boldsymbol{x}) \cdot \phi(\boldsymbol{x})^T$.

But there's a catch. In the original PCA we calculated the covariance matrix which uses the mean of our data. This isn't the case for the kernel PCA. We need the centered transformed data, $\phi_{centered}(\boldsymbol{x}) = \phi(\boldsymbol{x}) - \bar{\phi}(\boldsymbol{x})$, where $\bar{\phi}(\boldsymbol{x})$ is the mean of the transformed data, for which we do not know the new dimensions. Instead, we perform an indirect computation by centering accordingly the Gram Matrix:

$$\boldsymbol{K}_{centered}(\boldsymbol{x}_i, \boldsymbol{x}_j) = \boldsymbol{K} - \frac{1}{N}\mathbf{1} \cdot \boldsymbol{K} - \frac{1}{N}\boldsymbol{K} \cdot \mathbf{1} + \frac{1}{N^2}\mathbf{1} \cdot \boldsymbol{K} \cdot \mathbf{1}, \tag{26}$$

where both $\boldsymbol{K}$ and $\mathbf{1}$ are $N \times N$ matrices. After this, we proceed with eigenvalue decomposition of the $K_{centered}$. For the new (test) data that come in, we calculate the Gram Matrix of the new and the fitted (train) data: $K(\boldsymbol{x}_{test}, \boldsymbol{x}_{train})$ and project it (dot product) onto the previously acquired eigenvectors. It can be seen clearly that the original PCA is scaled with the number of features, whereas the Kernel PCA is scaled with the number of samples.

6 Linear Discriminant Analysis

As presented, the PCA method finds the directions where the maximum variance is achieved, although this isn't always the goal in classification tasks, as the different classes remain entangled. There exists another, supervised, method capable of finding the best projections, such that each class becomes separable from the others. This method is called Linear Discriminant Analysis (LDA) and its goals are two: 1) maximize the distances between classes and 2) keep the samples of each class as close as possible.

For the first goal the idea would be to maximize the distance between the projected means for all classes. Let m_i be the mean of the C_i class in the original space, with number of samples N_i, that is: $\boldsymbol{m}_i = \frac{1}{N_i}\sum_{j=1}^{N_i}\boldsymbol{x}_j$, $\boldsymbol{x} \in C_i$. Then for the mean of the projected, μ_i and the new direction $\boldsymbol{w}$, it follows that: $\mu_i = \boldsymbol{w}^T \cdot \boldsymbol{m}_i$. Then for each pair of classes we want to maximize the: $|\mu_1 - \mu_2|$. But there are cases where this maximization is not enough, as depicted in the figure below. Although the means of the projected $\mu_1^{'}, \mu_2^{'}$ are away from each other the samples are entangled.

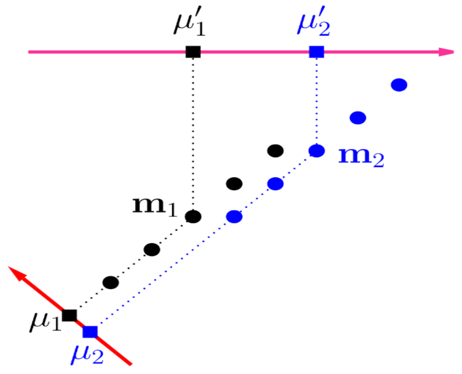


Figure 9: LDA cases(source: [21])

The next step is try to minimize the variance of each class' projected samples, that is try to compress them as much as possible to avoid such entanglements. Let the projected samples of class C_i be a_i then its variance, as showed before in the PCA, is: $s_i^2 = \sum_{i=1}^{C}(a_j - \mu_i)^2$, $a_j \in C_i$. So we need all the variances of all classes' samples to minimize and the their in between variance maximize. But there is the same constraint set to PCA as well, that is we are looking to achieve this by finding the best directions of $\boldsymbol{w}$ and not by setting its magnitude to infinity. Wrapping all this up together we have:

$$max\left(\mathcal{J}(\mu, s) = \frac{(\mu_i - \mu_j)^2}{s_i^2 + s_j^2}\right), \, subject\, to \; ||\boldsymbol{w}|| = 1 \tag{27}$$

We have to calculate the differences for each pair of all the classes. Instead it can be shown that this is equivalent with the difference of each projected mean from the total projected mean, or to be more precise the weighted mean, to ensure that all classes are represented equally, even with lower number of samples. The weighted projected mean is: $\bar{\mu} = \frac{1}{N}\sum_{i=1}^{C} N_i\mu_i$, where N_i are the samples of class C_i. Now the nominator (weighted) of eq. (27) can be written as [21]:

$$\sum_{i=1}^{C} N_i(\boldsymbol{\mu}_i - \bar{\boldsymbol{\mu}})^2 = \sum_{i=1}^{C} N_i(\boldsymbol{w}^T \cdot (\boldsymbol{\mu}_i - \bar{\boldsymbol{\mu}}))^2 = \sum_{i=1}^{C} N_i(\boldsymbol{w}^T \cdot (\boldsymbol{\mu}_i - \bar{\boldsymbol{\mu}}) \cdot (\boldsymbol{\mu}_i - \bar{\boldsymbol{\mu}})^T \cdot \boldsymbol{w})$$
$$= \boldsymbol{w}^T \cdot \sum_{i=1}^{C} N_i(\boldsymbol{\mu}_i - \bar{\boldsymbol{\mu}}) \cdot (\boldsymbol{\mu}_i - \bar{\boldsymbol{\mu}})^T \cdot \boldsymbol{w} = \boldsymbol{w}^T \cdot S_b \cdot \boldsymbol{w} \tag{28}$$

The S_b matrix is called Between Class Scatter Matrix and is $M \times M$, where M is the number of features. Then we proceed with the denominator:

$$\sum_{i=1}^{C} s_i^2 = \sum_{i=1}^{C} \sum_{j=1}^{N_i} (\boldsymbol{a}_j - \boldsymbol{\mu}_i)^2 = \sum_{i=1}^{C} \sum_{j=1}^{N_i} (\boldsymbol{w}^T \cdot \boldsymbol{x}_j - \boldsymbol{w}^T \cdot \boldsymbol{m}_i)^2 = \sum_{i=1}^{C} \sum_{j=1}^{N_i} (\boldsymbol{w}^T \cdot (\boldsymbol{x}_j - \boldsymbol{m}_i))^2$$
$$= \sum_{i=1}^{C} \sum_{j=1}^{N_i} \boldsymbol{w}^T \cdot (\boldsymbol{x}_j - \boldsymbol{m}_i) \cdot (\boldsymbol{x}_j - \boldsymbol{m}_i)^T \cdot \boldsymbol{w} = \boldsymbol{w}^T \cdot S_w \cdot \boldsymbol{w}, \tag{29}$$

where the S_w is called Within Class Scatter Matrix and is an $M \times M$ matrix as well. Now on eq. (27), we plug in equations (28), (29) and we get:

$$max \left(\frac{\boldsymbol{w}^T \cdot S_b \cdot \boldsymbol{w}}{\boldsymbol{w}^T \cdot S_w \cdot \boldsymbol{w}} \right), \, subject\, to \, ||\boldsymbol{w}|| = 1, \tag{30}$$

which is a generalized problem for C number of classes. The solution of the above equation is the solution of the generalized eigenvalue problem:

$$S_b \cdot \boldsymbol{w} = \boldsymbol{\lambda} S_w \cdot \boldsymbol{w}$$

The number of eigenvectors ($\boldsymbol{w}$) we can find is specified by the number of non zero eigenvalues ($\boldsymbol{\lambda}$) and we end up with maximum number of discriminatory directions equal to $C - 1$, where C is the number of classes. The problem that arises is that the Within Class Scatter Matrix needs to be non singular which is not the case when we have high dimensional data. For this purpose we should first apply PCA to reduce the dimensions and then feed the transformed data to LDA, in order to ensure S_w's non singularity.

7 KPCA + LDA

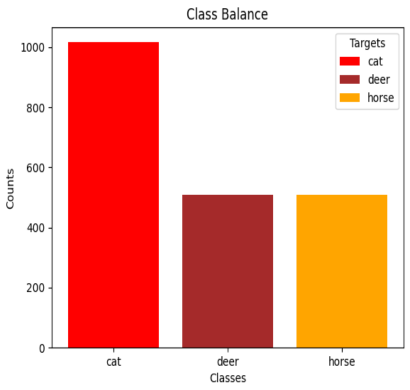
After implementing the methods we will use them to reduce dimensions by transforming our data. We proceed by applying them to our problem of three class classification, from the cifar-10 dataset. As suggested in [22], a powerful method is that of using initially KPCA and then apply its transformed data to the LDA, in order to reduce dimensions and separate the classes as much as possible. This is crucial in our case where the classifier used, an SVM, depends heavily on the number of samples while the dataset is high dimensional, leading to overfitting as seen above. With this technique we should be able to avoid overfitting by reducing the dimensions to a number lower than than of the number of samples.

7.1 Cat Classifier

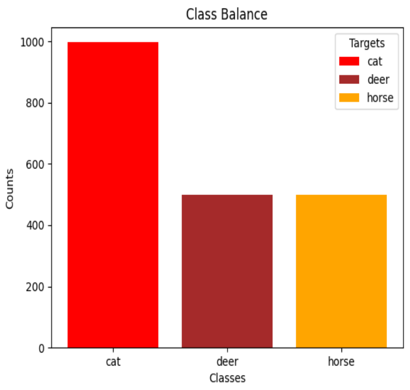
In the first part, we started tuning randomly the Deer Classifier. In this part we begin with the Cat Classifier, but first let's describe the whole process to be followed:

1. Hold data: train set: 'data_batch_1', validation set: 'data_batch_2', test set: 'data_batch_3'
2. Subsample the train, val and test set to ensure class balance
3. 'One vs All' : merge the two other classes together, set labels to $-1, 1$
4. Scale data: in the range $[-1, 1]$

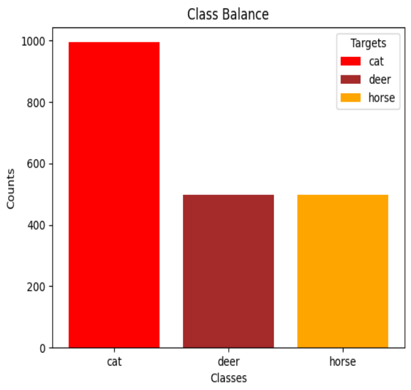
5. Find the maximum number of components LDA can handle (due to S_w's singularity)
6. First apply KPCA for different number of components
7. Apply LDA on the KPCA transformed data
8. Feed data in our SVM classifier and run tests using the train and validation sets
9. Hold the setup that performs best at the test set
10. Apply to the other classifiers
11. Combine all classifiers together and test them on the 'batch_test'



(a) Train Set



(b) Validation Set



(c) Test Set

Figure 10: Class Balance in Sets

So, as can be seen from the figure above we get a class balance for all our sets, at ~ 1000 samples per class. Next we fit our train data to the scaler and transform the train, validation and test sets. We should now determine the maximum number of components KPCA can fall into in order for the LDA to work. After trials it is found that the maximum number is at 1800 components (features) for the rbf kernel but this changes depended on the kernel used, so we will have to search for it every time we change kernel on KPCA. Remember that LDA can produce tops number of classes - 1 features, so in our case of binary classification ('one vs all' approach) we will end up in one dimension. Now we shall proceed with experimenting on the different hyperparameters for KPCA and SVM. In the table below we can see the different parameters and the validation scores for our first classifier.

Table 17: Cat Classifier Tests Data Scaled $[-1, 1]$

KPCA	SVM	F1	Precision	Recall	Best Param.
$\gamma_1 = [0.0001, 0.001, 0.01, 0.1, 0.5, 1]$ $n\_comp. = [1800, 1500, 1200, 1000, 700, 500, 200]$	$C = [0.01, 0.1, 1, 10, 100]$ $\gamma_2 = [0.0001, 0.001, 0.01, 0.1, 0.5, 1]$	76%	76%	76%	$\gamma_1 = 0.0001$ $\gamma_2 = 0.5$ $C = 10$ $n\_comp = 1000$
$\gamma_1 = [0.00001, 0.0001, 0.0005]$ $n\_comp. = [1100, 1000, 900]$	$C = [5, 10, 20]$ $\gamma_2 = [0.2, 0.5, 0.8]$	76.5%	77%	76.5%	$\gamma_1 = 0.0005$ $\gamma_2 = 0.2$ $C = 5$ $n\_comp = 1100$
$\gamma_1 = [0.00001, 0.0001, 0.001, 0.01]$ $n\_comp. = [1800, 1500, 1000]$	$C = [0.01, 0.1, 1, 10, 100]$ $degree = [4, 3, 2]$ $coef = [-1, 0, 1]$	76%	76%	76%	$\gamma_1 = 0.0001$ $degree = 3$ $coef = -1$ $C = 1$ $n\_comp = 1000$
$\gamma_1 = [0.0001, 0.0005]$ $n\_comp. = [1200, 1000, 800]$	$C = [1]$ $degree = [5, 3]$ $coef = [-2, -1, -0.5]$	76, 5%	76, 5%	76, 5%	$\gamma_1 = 0.0001$ $degree = 5$ $coef = -1$ $C = 1$ $n\_comp = 800$
$\gamma_1 = [0.00001, 0.0001]$ $n\_comp. = [1000, 800]$	$C = [0.1, 1, 10, 100]$ $\gamma_2 = [0.01, 0.1, 1]$ $coef = [-1, 0, 1]$	76.5%	76.5%	76.5%	$\gamma_1 = 0.0001$ $\gamma_2 = 1$ $coef = -1$ $C = 100$ $n\_comp = 800$
$\gamma_1 = [0.00001, 0.0001]$ $n\_comp. = [1000, 800]$	$C = [0.1, 1, 10, 100]$	76, 5%	76, 5%	76, 5%	$\gamma_1 = 0.0001$ $C = 10$ $n\_comp = 800$

As can be seen by the results above, the KPCA + LDA approach led to a simplification of our data that even simple methods such as a linear SVM can solve quite good, achieving scores similar to more sophisticated, kernel, approaches. Also, it can be noticed that all setups end up for their best score at around half the sample size for the KPCA number of components. Now, we should pick one setup and try to see if it fits the other two classifiers as well. The score may seem low, but remember that earlier on, when applied on the combination of all the more samples we gave the more the score became higher, even in the case of overfitting. So the question that lies is which setup to use. One solution would be to just choose the simplest one which will be the fastest as well, but first let's try all of the Table's 17 setups on our test set, to avoid the overfitting of the part I. Below there are the scores on the test set for the above setups with the same order as they appear above.

Table 18: Cat Classifier Scores on the Test Set

Setup	F1	Precision	Recall
$\gamma_1 = 0.0001$ $\gamma_2 = 0.5$ $C = 10$ $n\_comp = 1000$	$73,4\%$	$73,4\%$	$73,4\%$
$\gamma_1 = 0.0005$ $\gamma_2 = 0.2$ $C = 5$ $n\_comp = 1100$	$75,5\%$	$75,7\%$	$75,6\%$
$\gamma_1 = 0.0001$ $degree = 3$ $coef = -1$ $C = 1$ $n\_comp = 1000$	$73,5\%$	$73,5\%$	$73,5\%$
$\gamma_1 = 0.0001$ $degree = 5$ $coef = -1$ $C = 1$ $n\_comp = 800$	$74,3\%$	$74,3\%$	$74,3\%$
$\gamma_1 = 0.0001$ $\gamma_2 = 1$ $coef = -1$ $C = 100$ $n\_comp = 800$	$74,2\%$	$74,2\%$	$74,2\%$
$\gamma_1 = 0.0001$ $C = 10$ $n\_comp = 800$	$74,2\%$	$74,2\%$	$74,2\%$

Although all of the setups performed equally on the validation set, we see that the rbf setup wins the race on the test set results. The scores achieved here for one of the classifiers match the one we obtained on the first part. Now we shall see if any improvement can be achieved on the final combined classifier. The next step is to try our best parameters on the other two classifiers and test their performance as well.

Let's see out of curiosity or final data after KPCA + LDA dropped the dimensions down to one, separated for visual purposes. It can be seen that they actually overlap quite enough, but the rbf kernel of the SVM classifier manages to separate them a bit more.

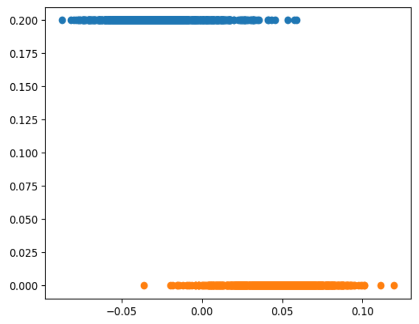


Figure 11: Data to 1D after KPCA+LDA

7.2 Deer and Horse Classifiers

Below we examine the performance of the Deer and Horse Classifiers on the test set using the best setup from above.

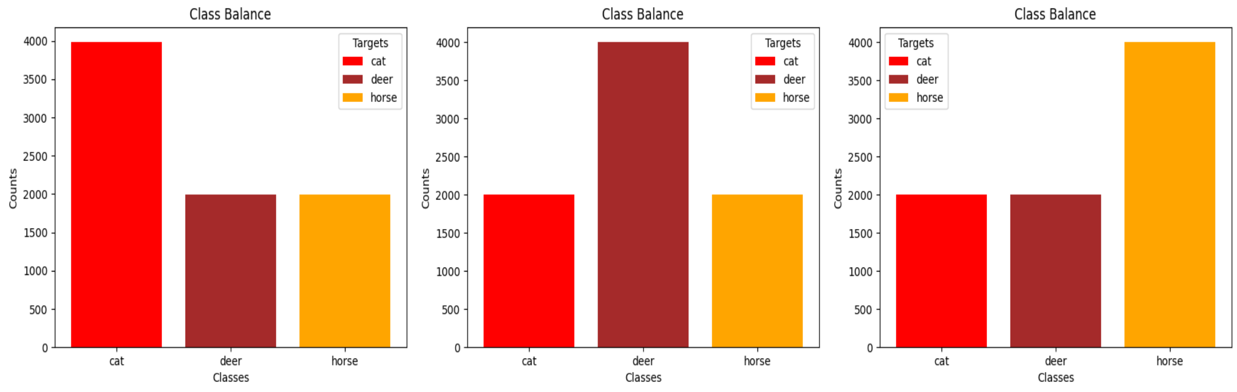
Table 19: Performance of Deer and Horse Classifiers on the Test Set

Classifier	F1	Precision	Recall
Deer	75%	75%	75%
Horse	74, 7%	74, 7%	74, 7%

The results on the test set for the two other classifiers are satisfactory and so we proceed with the same setup for all three of our classifiers.

7.3 Classifier Combination

For the combination we must train and test each classifier individually, as their scores will be used for the voting process of the combination. Now our train test consist of the batches 2-5 and the test set remains the same. There is one batch left but cannot be loaded due to memory shortage. First we fit and transform our data using KPCA and then LDA. We subsample these data and feed each SVM classifier. Then the process of decision is applied as already discussed in the previous part. In the figure below we can see the new train sets. The number of samples are now at ~ 8000.



(a) Cat Classifier Train Set (b) Deer Classifier Train Set (c) Horse Classifier Train Set

Figure 12: Class Balance in Sets

In Table 20 we can see the performance of each individual classifier and in Table 21 the results of the combined classifiers, while in Figure 13 we can see the new confusion matrix.

Table 20: Performance of Individual Classifiers

Classifier	F1	Precision	Recall
Deer	62%	62%	62%
Horse	81%	81%	81%
Horse	66%	66%	66%

Table 21: Performance of the Combined Classifiers

F1	Precision	Recall
52%	53%	55%

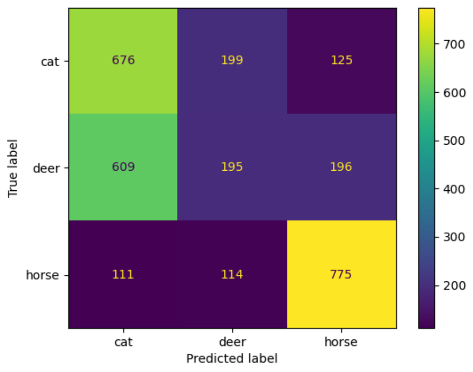


Figure 13: Confusion Matrix of the Combined Classifiers

In Figure 14 we see the difference when our data are projected to a final two dimensional space rather than one dimensional space as depicted in Figure 11.

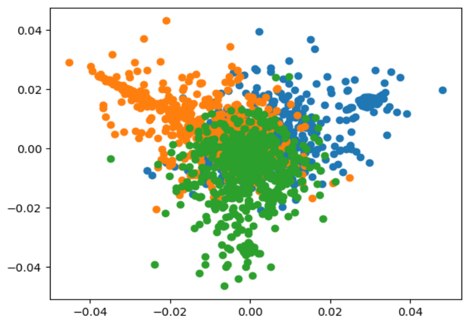


Figure 14: Data to 2D after KPCA+LDA

8 Conclusions

In the first part we noticed that we have a clear case of overfitting as there was a gap of around 10% in the scores of each individual classifier and their combination. It was pointed out that this was due to the number of samples being a lot more than the number of features. If we remember our training samples were ~ 2000 while the features where ~ 3100. We could have prevented this by following the process we did in the second part, that is use validation for tuning and test set for testing each individual classifier, and not just test their combination at the end. To solve the overfitting problem we introduced the method of KPCA + LDA that initially rises our data to higher dimensional space (kernel trick) and tries to maximize their variance and then considering each class tries to separate each label from the others as much as possible. Theoretically this should have given us better results. But again another mistake was made. During tuning of each classifier we missused the KPCA + LDA approach. Our final goal is to create a three class classifier, but instead we focused on each individual without looking at the bigger picture. Instead of fitting our initial data, that is three class data to the KPCA + LDA, we first created and fitted the data of each classifier, that is two class data. This was a huge mistake. When we proceeded with the combination it was clear that the data fit and transformed should be the three class data, but our classifiers had already been misstuned, due to different initial data transformation. The data our classifiers 'saw' during the tuning phase had nothing to do with the three class data transformation. This led to the disappointing scores presented above. The only miracle is the horse classifier which was favored by this miss-transformation and predicted better. We should now be able to learn two very basic lessons: 1) always test before creating ensemble of models and 2) take a look at the bigger picture of your task rather than each part separately.

9 References

[1] Vladimir N Vapnik and A Ya Chervonenkis. On a perceptron class. *Automation and Remote Control*, 25:112–120, 1964.

[2] Bernhard E Boser, Isabelle M Guyon, and Vladimir N Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the fifth annual workshop on Computational learning theory*, pages 144–152, 1992.

[3] CVXOPT, Python Software for Convex Optimization. https://cvxopt.org/.

[4] Randeep Ahlawat. SVM from scratch using Quadratic Programming. https://medium.com/@ahlawat.randeep/svm-from-scratch-using-quadratic-programming-90b4dbc5e1d2.

[5] Major Kernel Functions in Support Vector Machine (SVM), GeeksForGeeks. https://www.geeksforgeeks.org/major-kernel-functions-in-support-vector-machine-svm/.

[6] Polynomial Kernel, Wikipedia. https://en.wikipedia.org/wiki/Polynomial_kernel.

[7] Alex Krizhevsky. Learning Multiple Layers of Features from Tiny Images. https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf, 2009.

[8] Lecture 9: SVM, Cornell, Machine Learning for Intelligent Systems. https://www.cs.cornell.edu/courses/cs4780/2018fa/lectures/lecturenote09.html.

[9] Lagrangian Formulation of the SVM. http://sfb649.wiwi.hu-berlin.de/fedc_homepage/xplore/tutorials/stfhtmlnode64.html.

[10] Yaser Abu-Mostafa, Malik Magdon-Ismail, Hsuan-Tien Lin. Support Vector Machines, e-Chapter 8. https://edisciplinas.usp.br/pluginfile.php/5078086/course/section/5978681/chapSVM.pdf, 2015.

[11] IIT Madras - B.S. Degree Programme, YouTube Channel. Dual formulation for Soft Margin SVM. https://www.youtube.com/watch?v=MZzp4OP8GgQ.

[12] Sidharth. SVM Kernels: Polynomial Kernel - From Scratch Using Python. https://www.pycodemates.com/2022/10/svm-kernels-polynomial-kernel.html.

[13] Carlos Domínguez García. Visualizing the effect of hyperparameters on Support Vector Machines, Medium. https://towardsdatascience.com/visualizing-the-effect-of-hyperparameters-on-support-vector-machines-b9eef6f7357b.

[14] University of Winsconsin-Madison, Computer Science Department. The Radial Basis Function Kernel. https://pages.cs.wisc.edu/~matthewb/pages/notes/pdf/svms/RBFKernel.pdf.

[15] Nikolay Manchev. Fitting Support Vector Machines via Quadratic Programming. https://domino.ai/blog/fitting-support-vector-machines-quadratic-programming.

[16] Cosma Shalizi. Principal Components Analysis, Chapter 18, Carnegie Mellon University, Statistics & Data Science.

[17] Jonathan Hui. Machine Learning — Singular Value Decomposition (SVD) & Principal Component Analysis (PCA), Medium. https://jonathan-hui.medium.com/machine-learning-singular-value-decomposition-svd-principal-component-analysis-pca-1d45e885e491.

[18] Kunyu He. SVD in Machine Learning: PCA Understand what is principal component analysis, how we do it via EVD and SVD, and why the SVD implementation is better, Medium. https://towardsdatascience.com/svd-in-machine-learning-pca-f25cf9b837ae.

[19] PCA and kernel PCA explained, NIRPY RESEARCH. https://nirpyresearch.com/pca-kernel-pca-explained/.

[20] Bernhard Schölkopf, Alexander Smola, and Klaus-Robert Müller. Kernel principal component analysis. In *International conference on artificial neural networks*, pages 583–588. Springer, 1997.

[21] Guangliang Chen. LEC 3: Linear Discriminant Analysis (LDA), A Supervised Dimensionality Reduction Approach, Mathematics & Statistics, San José State University . https://www.sjsu.edu/faculty/guangliang.chen/Math250/lec9lda.pdf.

[22] Jian Yang, Alejandro F Frangi, Jing-yu Yang, David Zhang, and Zhong Jin. Kpca plus lda: a complete kernel fisher discriminant framework for feature extraction and recognition. *IEEE Transactions on pattern analysis and machine intelligence*, 27(2):230–244, 2005.

A Appendix

A.1 Runtimes

Table 22: Runtimes of Classifiers

Classifier	Setup	Runtime
Combination	(samples: $\sim 24,000$, kernel: rbf)	77 *min.*
Combination	(samples: $\sim 24,000$, features: 2, kernel: rbf)	68 *min.*
Individual	(samples: $\sim$ 2000, kernel: rbf)	32 *sec.*
Individual	(samples: $\sim$ 2000, kernel: rbf, features: 1)	17 *sec.*
Individual	(samples: $\sim$ 2000, kernel: poly, degree: 2)	33 *sec.*
Individual	(samples: $\sim$ 2000, kernel: poly, degree: 8)	4 *min.*
Individual	(samples: $\sim$ 2000, kernel: poly, degree= 2, features: 1)	28 *sec.*
Individual	(samples: $\sim$ 2000, kernel: poly, degree= 8, features: 1)	24 *sec.*
Individual	(samples: $\sim$ 2000, kernel: sigmoid)	4 *min.*
Individual	(samples: $\sim$ 2000, kernel: sigmoid, features: 1)	18 *sec.*

Table 23: Runtimes of KPCA and LDA

Dim. Reduction	Setup	Runtime
KPCA	(samples: $\sim$ 2000, kernel: rbf)	3.5 *sec.*
KPCA	(samples: $\sim$ 4000, kernel: rbf)	24 *sec.*
KPCA	(samples: $\sim 12,000$, kernel: rbf)	9 *min.*
KPCA	(samples: $\sim$ 2000, kernel: poly, degree: 2)	4 *sec.*
KPCA	(samples: $\sim$ 4000, kernel: poly, degree: 2)	23 *sec.*
KPCA	(samples: $\sim$ 2000, kernel: poly, degree: 4)	3 *sec.*
KPCA	(samples: $\sim$ 4000, kernel: poly, degree: 4)	23 *sec.*
KPCA	(samples: $\sim$ 2000, kernel: sigmoid)	3 *sec.*
KPCA	(samples: $\sim$ 4000, kernel: sigmoid)	23 *sec.*
LDA	(samples: $\sim$ 2000, features: 1800)	35 *sec.*
LDA	(samples: $\sim$ 4000, features: 1800)	70 *sec.*
LDA	(samples: $\sim$ 2000, features: 1000)	6 *sec.*
LDA	(samples: $\sim$ 4000, features: 1000)	12 *sec.*
LDA	(samples: $\sim$ 2000, features: 500)	0.6 *sec.*
LDA	(samples: $\sim$ 4000, features: 500)	1.3 *sec.*
LDA	(samples: $\sim 12,000$, features: 1100)	50 *sec.*

Processor: Intel® Core™ i7-8550U, 4 GHz (turbo mode)
RAM: 16 Gb
Operating System: Ubuntu 23.10

A.2 Equations

Here we compute analytically the equations presented in section *Lagrange formulation*. These equations are the (14), (15), (16) and (18).

$$eq.\ (14):\ \frac{\partial \mathcal{L}}{\partial \boldsymbol{w}} = 0 \rightarrow \frac{\partial}{\partial \boldsymbol{w}}(1/2^{10}\boldsymbol{w}^T \cdot \boldsymbol{w}) + \frac{\partial}{\partial \boldsymbol{w}}\left(-\sum_{i=1}^{N} a_i(y_i(\boldsymbol{w}^T \cdot \boldsymbol{x}_i + b) + \epsilon_i - 1\right) = 0$$

$$\rightarrow \boldsymbol{w} + \frac{\partial}{\partial \boldsymbol{w}}\left(-\sum_{i=1}^{N} a_i y_i(\boldsymbol{w}^T \cdot \boldsymbol{x}_i + b)\right) + \frac{\partial}{\partial \boldsymbol{w}}\left(\sum_{i=1}^{N} a_i \epsilon_i\right) + \frac{\partial}{\partial \boldsymbol{w}}\left(-\sum_{i=1}^{N} -a_i\right) = 0$$

$$\rightarrow \boldsymbol{w} - \frac{\partial}{\partial \boldsymbol{w}}\left(\sum_{i=1}^{N} a_i y_i \boldsymbol{w}^T \cdot \boldsymbol{x}_i\right) = 0 \rightarrow \boldsymbol{w} - \sum_{i=1}^{N} a_i y_i \boldsymbol{x}_i = 0 \rightarrow \boxed{\boldsymbol{w} = \sum_{i=1}^{N} a_i y_i \boldsymbol{x}_i}$$

$$eq.\ (15):\ \frac{\partial \mathcal{L}}{\partial b} = 0 \rightarrow \frac{\partial}{\partial b}\left(-\sum_{i=1}^{N} a_i(y_i(\boldsymbol{w}^T \cdot \boldsymbol{x}_i + b) + \epsilon_i - 1)\right) = 0 \rightarrow -\frac{\partial}{\partial b}\left(\sum_{i=1}^{N} a_i y_i b\right) = 0 \rightarrow \boxed{\sum_{i=1}^{N} a_i y_i = 0}$$

$$eq.\ (16):\ \frac{\partial \mathcal{L}}{\partial \boldsymbol{\epsilon}} = 0 \rightarrow \frac{\partial}{\partial \boldsymbol{\epsilon}}\left(C\sum_{i=1}^{N} \epsilon_i\right) - \frac{\partial}{\partial \boldsymbol{\epsilon}}\left(\sum_{i=1}^{N} \mu_i \epsilon_i\right) - \frac{\partial}{\partial \boldsymbol{\epsilon}}\left(\sum_{i=1}^{N} a_i \epsilon_i\right) = 0 \rightarrow C\sum_{i=1}^{N} 1 - \sum_{i=1}^{N} \mu_i - \sum_{i=1}^{N} a_i = 0$$

$$\rightarrow \boxed{\sum_{i=1}^{N}(a_i + \mu_i) = C\sum_{i=1}^{N} 1}$$

Substituting eq. (14) in (11), by also involving the results above from equations (15) and (16), we get the final solution of equation (18).

$$eq.\ (18): \mathcal{L}(a) = \frac{1}{2}\left(\sum_{i=1}^{N} a_i y_i \boldsymbol{x}_i^T\right)^T\left(\sum_{j=1}^{N} a_j y_j \boldsymbol{x}_j^T\right) + C\sum_{i=1}^{N} \epsilon_i - \sum_{i=1}^{N} a_i(y_i(\sum_{j=1}^{N} a_j y_j \boldsymbol{x}_j^T \boldsymbol{x}_i + b) - \epsilon_i - 1) - \sum_{i=1}^{N} \mu_i \epsilon_i$$

$$\mathcal{L}(a) = \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} a_i y_i a_j y_j \boldsymbol{x}_i \boldsymbol{x}_j^T + C\sum_{i=1}^{N} \epsilon_i - \sum_{i=1}^{N} \mu_i \epsilon_i - \sum_{i=1}^{N}\sum_{j=1}^{N} a_i y_i a_j y_j \boldsymbol{x}_i \boldsymbol{x}_j^T - \sum_{i=1}^{N} a_i y_i b - \sum_{i=1}^{N} a_i \epsilon_i + \sum_{i=1}^{N} a_i$$

$$\mathcal{L}(a) = -\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} a_i y_i a_j y_j \boldsymbol{x}_i \boldsymbol{x}_j^T + \sum_{i=1}^{N} a_i - \sum_{i=1}^{N} \epsilon_i(\mu_i + a_i) + C\sum_{i=1}^{N} \epsilon_i$$

$$\mathcal{L}(a) = -\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} a_i y_i a_j y_j \boldsymbol{x}_i \boldsymbol{x}_j^T + \sum_{i=1}^{N} a_i + \sum_{i=1}^{N} \epsilon_i(\mu_i + a_i - C)$$

$$\boxed{\mathcal{L}(a) = -\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} a_i y_i a_j y_j \boldsymbol{x}_i \boldsymbol{x}_j^T + \sum_{i=1}^{N} a_i}$$

[10] The 2 factor comes in as we take margin for both classes, as it was said in the beginning it was omitted, but is needed for the final calculations.