Institute of Statistics, Biostatistics and Actuarial Sciences

LDATS2470: Statistical Machine Learning and High Dimensional Data Analysis

Lending club database: insurance policies classification

Authors:
GODFRIN Siméon (DATS2M- 60921800)
RGHIOUI Mehdi (DATS2M- 29912100)
WARNAUTS Aymeric (DATS2M- 87031800)

Professor:
HAFNER Christian

2022-2023

Contents

1 Introduction

For this project, we will work on the *Lending Club* loan database as this is one of the richest loan dataset that provides all loan information for loans granted by the lending club between 2007 and 2011 in the United States. This project will focus on the application of the *Support Vector Machine* which is a supervised machine learning algorithm used for classification and regression analysis. We expect the implementation of Support Vector Machine (SVM) to be useful as this method is able to construct non-linear decision boundaries between different classes of data, which can be helpful for capturing complex relationships between loan default and the various features that are used to predict it. Moreover, we will take advantage of its robustness to high-dimensional data and its ability to handle imbalanced datasets by using techniques such as cost-sensitive learning and class weighting, as we observe in the following table a non symmetric distribution of observations across the levels of our target feature *loan_status*:

	Frequency	Percentage	Cumulative percentage
Fully Paid	32 950	85.4 %	85.5 %
Charged Off	5 627	14.6 %	100 %
Total	38 577	100 %	100 %

Table 1: *loan_status* levels frequencies

As the choice between *hinge loss* and *quadratic loss* depends on the specific question being addressed, the desired trade-off between accuracy and robustness and the type of model used, we will explore both to find an appropriate one. However, the *hinge loss* function is typically used for classification problems with binary labels as our target feature *loan_status* because it is a convex function that is designed to encourage the model to correctly classify data points, while also allowing an error rate. Finally, the *hinge loss* function is recognized for its robustness to outliers compared to the quadratic loss function. This characteristic enables it to effectively handle noisy data. Additionally, SVMs employing the *hinge loss* exhibit sparsity by having fewer support vectors. This sparsity not only contributes to computational efficiency but also enhances their overall performance. We defined this loss as:

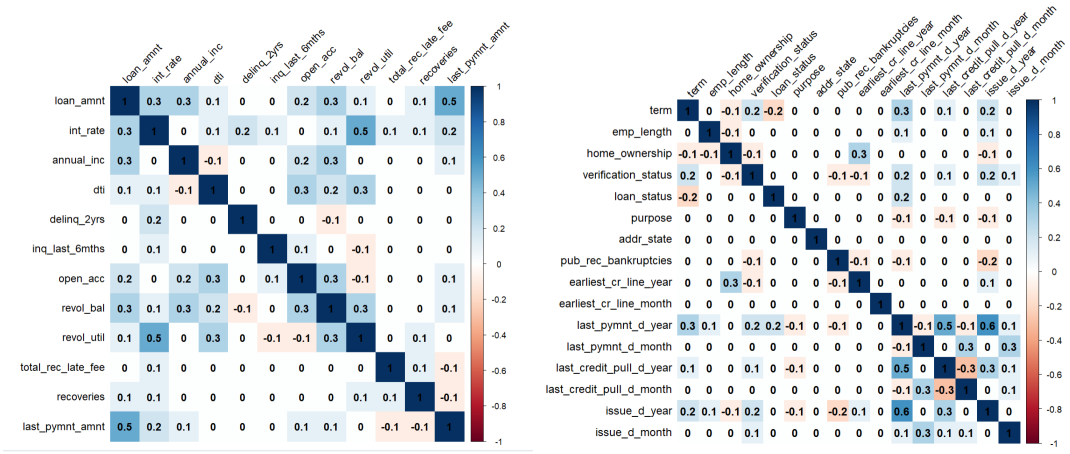
$$\mathcal{L}(y, f(x)) = \max(0, 1 - yh(x))$$

Where y is the true label, $h(x)$ is the predicted score given by the fitted hyperplane equation as $h(x) = w^T x + b$. For a given data point, if the function $h(x)$ yields a positive value, we classify it as belonging to the positive class; conversely, if the function yields a negative value, we classify it as belonging to the negative class. By multiplying y by $h(x)$, the hinge loss function effectively increases the penalty for incorrect predictions. When the predicted score $h(x)$ shares the same sign as the true label $y \in -1, 1$, the product $yh(x)$ is positive, indicating a correct prediction and resulting in a null loss for that data point. Conversely, when the product is negative, it signifies an incorrect prediction, and the loss is proportional to the magnitude of the product $yh(x)$. Larger magnitudes correspond to greater losses, reflecting the model's degree of error.

2 Preprocessing

For our preprocessing part we will not perform specific treatment for outliers, as for *SVM* classification of loan default, outliers may represent borrowers with unique characteristics or circumstances that could have a significant impact on their likelihood of default and thus removing these contracts could result in a biased analysis and potentially lead to incorrect predictions of loan default risk.
Moreover we will discard some features regarding correlation plots of both continuous and categorical features and features related to text and date types. However, regarding the treatment of features such as *desc, emp_title* and *title* we would have used embedding transformation to gain information but this goes beyond the limits of

this course. This leaves us with non-redundant variables that we represent as a matrix of correlations to verify our cleaning:



What concerns *na* values, the strategy will be to fit linear regression to impute missing values in the loan dataset by building a model for each column with missing values and predicting these missing values using other columns as predictors.

3 Optimization by cross-validation

As said, cross-validation is a technique used to validate the performances of a given model in several folds of our training data, especially it's appropriate when fitting models with hyper-parameters tuning as these have to be chosen to improve performances on unseen test data. Since the unobersed testing data set is supposed to be unavailable during training, we cannot simply select the hyper-parameters that give the best performances on the training data as this can lead to over-fitting, suggesting the cutting of the training set in predetermined number of *k-folds* in order to validate our assumptions about the choice of hyper-parameters. The model is then trained on $k-1$ folds and evaluated on the remaining one and this process is repeated k times, with each fold allocated to the validation set exactly once. The performance of the model is then averaged across the folds to give an estimate of the model's performances on new data. In our case, **grid-search** will be the procedure for testing a wide range of different hyper-parameters. By trying out different hyper-parameter values and evaluating the model's performances using cross-validation, we can find the best combination of hyper-parameters that gives the best average performance on new data. Thus, we re-coded the algorithm to keep tracks of some key metrics explained later. Here it is in pseudo-code where we will only list the parameters that will be subject to optimization:

- *Cost* : This is the penalty parameter for misclassifying training data as a larger value of this hyper-parameter increases the penalty for misclassification, which makes the algorithm more likely to choose a decision boundary that separates the classes more accurately. Thus, a smaller value of cost allows for more misclassifications in the training data, which may result in a decision boundary that is more flexible but may overfit the data. Thus, the cost C quantify the way our SVM is over-fitting training data.
- *Gamma* : This parameter controls the width of the *Gaussian* kernel and a larger value of *gamma* makes the kernel narrower, which means that each training sample has a smaller influence on the decision boundary. A smaller value of *gamma* makes the *Gaussian* kernel wider, which means that each training example has a larger influence on the decision boundary. This will be used in the *RBF kernel*, but not in the *polynomial* one.
- *Coef0* : This is the independent term in the *polynomial* kernel function as it determines the offset of the decision boundary from the origin. A non-zero value of this hyperparameter shifts the decision boundary away from the origin, while a zero value places it at the origin.

- *Degree* : This is the degree of the *polynomial* kernel as it controls the complexity of the decision boundary by determining the order of the polynomial. Thus, a larger value of *degree* creates a more complex decision boundary that can fit the training data more closely, but may overfit the data. A smaller value of *degree* creates a simpler decision boundary that may not fit the training data as closely, but may generalize better to new data.

Moreover, these parameters are not independent from each others as increasing the *degree* of the *polynomial* kernel can increase the complexity of the decision boundary, which may require increasing the *cost* parameter to prevent overfitting. On the other hand, increasing the *coef0* parameter can shift the decision boundary away from the origin, which may require decreasing the *degree* parameter to prevent overfitting. This is why we will perform a grid search to find the optimal combination of parameters:

Algorithm 1 Cross-validation with grid search for SVM

Require: A dataset $\mathcal{D}\{y_i, x_i\}_{i=1}^n$, a set of hyperparameters $\mathcal{H} = (C_1, \gamma_1, \dots), (C_2, \gamma_2, \dots), ..., (C_p, \gamma_p, \dots)$ for SVM, and the number of folds k for cross-validation.

Divide $\mathcal{D}$ into k equally sized subsets $\mathcal{D}_1, \mathcal{D}_2, ..., \mathcal{D}_k$
Initialize a variable $best_h$ to $\emptyset$ and a variable $best_{score}$ to ∞
for all $h_i = (C_i, \gamma_i, \dots) \in \mathcal{H}$ **do**
  Initialize a variable $score_i$ to 0
  for $j = 1$ to k **do**
    Define the training set $\mathcal{T}_j$ as $\mathcal{D}$ excluding subset $\mathcal{D}_j$
    Train an SVM model $f_{\mathcal{T}_j}^{(i)}$ on T_j with hyperparameters h_i
    Define the validation set $\mathcal{V}_j$ as $\mathcal{D}_j$
    Evaluate $f_{\mathcal{T}_j}^{(i)}$ on $\mathcal{V}_j$ and record performance metric m_{ij}
    Add m_{ij} to $score_i$
  end for
  Calculate the average performance of h_i: $average_i = \frac{1}{k} \sum_{j=1}^{k} m_{ij}$
  if $average_i < best_{score}$ **then**
    Update $best_{score}$ to $average_i$
    Update $best_h$ to h_i
  end if
end for
Train a final SVM model f on $\mathcal{D}$ with hyperparameters $best_h$
return f the final model

For other nonlinear cases, where we have a lot more parameters and thus combinations to try, we implemented a learning rate. We initially set a 1×3 vector of random values for each hyperparameter to work with. The algorithm updates the initial values for each parameter by selecting the best combination of hyperparameters as a central value. Then we implemented a concentration coefficient based on the current best parameter (either cost of the gamma parameters.) We end up with 1×3 vector for each hyperparameter, all we have to do is restart the loop with those new values till the model does not improve his *detection rate* while maintaining a stable *balanced accuracy*. We will explain these metrics in what follows.

4 Linear Support Vector Machine

Support Vector Machine is a machine learning algorithm that can be used for classification tasks as *SVM* tries to construct a hyperplane in a high-dimensional input space (i.e in d dimensions) that separates the data into two area's that will corresponds to the classification of the *Charged_Off* and *Fully_Paid* labels of the *loan_status* feature. The function of the hyperplane is $h(x)$ as we defined :

$$h(x) = b + w_1 x_1 + w_2 x_2 + ... + w_p x_p$$

where $x_1, x_2, ..., x_p$ are the input variables, $w_0, w_1, w_2, ..., w_p$ are the coefficients of the hyperplane, and p is the number of input variables. All points that lie on the hyperplane have $h(x) = 0$.
The *SVM* tries to find the hyperplane that maximizes the margin while still correctly classifying all the training data. The margin is the distance between the hyperplane and the closest points of each class and can be either hard or soft. The hard margin represents the cases where we do not allow any misclassification. The soft margin approach allows for some degree of misclassification, and is ruled by the cost parameter and the slack variables.
The slack variables," ξ", represent in the case of a misclassified point, the distance between this point and the closest margin to his class. The value is equal to zero if the point is well classified. They embody the overlapping of classes.
In the most plausible case, our dataset is non linearly separable, here follows the objective function:

$$\min_{\boldsymbol{w},b,\boldsymbol{\xi}} \left(\frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_{i=1}^{n} \xi_i^k \right)$$

The parameter C is a tuning parameter that is used to determine the penalty for misclassification of the training data points.

4.1 Linear separability

Linear separability is determined by the ability to find a hyperplane that can completely separate the two classes of our binary target in the feature space. Mathematically, this can be represented as follows, given a set of training data:

$$(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), \ldots, (\boldsymbol{x}_n, y_n)$$

where $\boldsymbol{x}_i$ is a vector of features for the i-th data point, and y_i is its corresponding class label, the problem of linear separability can be formulated as finding a hyperplane $\boldsymbol{w}^T\boldsymbol{x} + b = 0$ such that:

$$\boldsymbol{w}^T\boldsymbol{x}_i + b \geq +1, \text{ if } y_i = +1$$
$$\boldsymbol{w}^T\boldsymbol{x}_i + b \leq -1, \text{ if } y_i = -1$$

where $\boldsymbol{w}$ is the weight vector and b is the bias term.
If these constraints are satisfied, then the training data is linearly separable, and this is equivalent to making the assumption that there exists a margin γ such that:

$$\boldsymbol{w}^T\boldsymbol{x}_i + b \geq +\delta^*, \text{ if } y_i = +1$$
$$\boldsymbol{w}^T\boldsymbol{x}_i + b \leq -\delta^*, \text{ if } y_i = -1$$

where the margin δ^* can be calculated as $\delta^* = \frac{2}{||\boldsymbol{w}||^2}$, where $||\boldsymbol{w}||^2$ is the Euclidean norm of $\boldsymbol{w}$. In fact, the *SVM* will choose the canonical hyperplane that yields the maximum margin among all possible separating hyper-planes, as it's the distance between the decision boundary and the closest data points from either class. Thus as we denote x_- a negative point that lies on the negative side of the hyperplane and x_+ a point that lies on the positive side, we can define the margin to be maximized as:

$$\delta^* = \frac{w^T x_+ + b}{||w||^2} - \frac{w^T x_- + b}{||w||^2}$$

and this defines what we call *support vectors* as these are lying on the margin for both sides of the hyper-planes, then the equations of such points are respectively equal to 1, -1. Thus the optimization for support vector machine algorithms can be formulated as a constrained optimization problem, where the goal is to minimize the classification error while maximizing the margin between the two classes. Then, the optimization involves finding

a set of *Lagrange multipliers*, also known as *dual* variables, which represent the weights of the training samples in the classification problem. The utilization of *Lagrange multipliers* serves the purpose of identifying the support vectors, which are data points positioned in close proximity to the decision boundary. These support vectors hold paramount importance within the training dataset as they provide crucial information, exerting the highest influence on the optimal hyperplane. This is because the decision boundary is constructed as a linear combination of the support vectors and their corresponding *Lagrange multipliers*, further emphasizing their significance. The explanations above have been retrieved from the slides of the *Statistical Machine Learning and High Dimensional Data Analysis* lectures (Hafner, 2021) of the *LSBA* faculty at *UCLouvain*.

4.2 Results

For the linear support vector machine, the cost C will be our regularization parameter that controls the trade-off between achieving a low training error and a low testing error as we already said, a smaller value of this metric leads to a wider margin, but more misclassifications, while a larger value of C leads to a narrower margin, but fewer misclassifications. Moreover, as we want to compare the performances of models with different cost parameters we will introduce some metrics such as the *accuracy* which is the most straightforward metric for evaluating the performance of a classification model because it simply measures the percentage of correctly classified instances out of all instances in the dataset. We will make reference to the *precision* and *recall* percentages which are the proportion of true positive predictions out of all positive predictions made by the model and out of all actual positive instances in the dataset respectively. Finally, to summarize we will compute the *F1 score* that is a weighted average of precision and recall, with equal weighting given to both measures and that provides a single measure of the overall performance of a model. We recall these formula's that will be computed to assess from our classification percentages:

$$\text{Sensitivity} = \frac{TP}{TP + FN} \quad (1) \qquad \text{Precision} = \frac{TP}{TP + FP} \quad (3)$$

$$\text{Recall} = \frac{TP}{TP + FN} \quad (2) \qquad \text{F1-score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \quad (4)$$

$$\text{Balanced Accuracy} = \frac{1}{2}\left(\frac{TP}{TP + FN} + \frac{TN}{TN + FP}\right) \quad (5) \qquad \text{Detection Rate} = \frac{TP}{TP + FP + FN + TN} \quad (6)$$

The first thing we want to verify : are our dataset linearly separable ? In order to find an answer, we compute the accuracy of our linear SVM given that we choose an hard margin, by choosing a cost value $C = \infty$. A perfectly linearly separable dataset would have an accuracy of 100%. Here are our results:

- *bal accuracy*: 75.07%
- *precision*: 29.05%
- *F1-score*: 43.45%
- *margin*: 6.829×10^{-8}
- *detec rate*: 12.62%
- *support V*: 3979

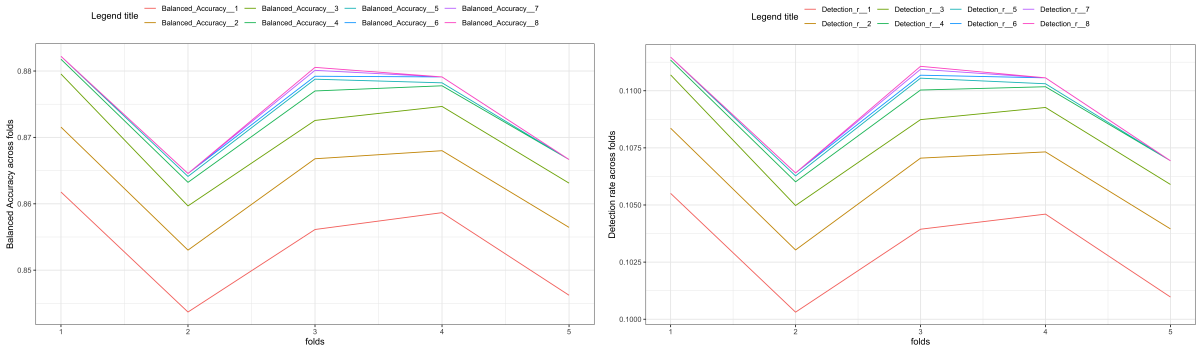
Therefore our dataset is not perfectly linearly separable as we train and test the performances on the whole dataset.

Now that this assumption has been checked, we will use a soft margin and allow misclassification, as explained earlier. We report in the following table the performances of several values of C obtained by grid search and cross-validation (we have introduced these concepts in the section Optimization by cross-validation) , with the mean values of each metric across all folds for the linear *SVM*:

Model	C	error	dispersion	support_V	Sensitivity	F1_score	Balanced_Accuracy	Precision	Recall	Detection_r
SVM Linear 1	5	0.04283	0.004013	4285	0.7065	0.8279	0.8532	1	0.7066	0.1030
SVM Linear 2	7	0.03953	0.003975	4710	0.7263	0.841	0.8631	1	0.7263	0.1059
SVM Linear 3	9	0.03765	0.004361	4701	0.7398	0.8503	0.8699	1	0.7398	0.1079
SVM Linear 4	11	0.03632	0.004166	4627	0.7466	0.8548	0.8733	1	0.7466	0.1089
SVM Linear 5	13	0.03596	0.004016	4610	0.7480	0.8557	0.8740	1	0.7480	0.1091
SVM Linear 6	15	0.03583	0.003925	4591	0.7487	0.8562	0.8743	1	0.7487	0.1092
SVM Linear 7	17	0.03573	0.003904	4659	0.7490	0.8564	0.8745	1	0.7490	0.1093
SVM Linear 8	19	0.03570	0.003867	4611	0.7492	0.8566	0.8746	1	0.7492	0.1093
SVM Linear 9	25	0.0575	0.0036	4582	0.7492	0.8566	0.8746	1	0.7492	0.1093
SVM Linear 10	50	0.0554	0.0037	4490	0.7496	0.8570	0.8748	1	0.7496	0.1094

Table 2: SVM Linear Model Performance

We recall that the objective of this project, as we are working in actuarial setting, is to build an appropriate classifier that detect contracts that are/will be *fully paid* but as we report the *precision metric* that is always equal to 1, the main focus will be to detect as much *charged off* (i.e true negatives) observations as possible. Thus as we know that our percentage of *charged off* contracts in our test $\mathcal{T}$ and train set $\mathcal{D}$ are approximately around 14.6%, we will choose the *SVM* parameters that allows our *detection rate* to be the closest to this value (see the Fire Outbreak Detection System that follows the same approach (Umoh, Udo, & Nyoho, 2019) in the appendices). For the assessment and as we want to follow the propagation of these key values in our 5 folds, we display the trace plots below:



Thus looking at these plots, we can conclude that both the convergences of the *balanced accuracy* and the *detection rate* are reaching optimal values in the folds for $C > 15$. Thus looking at the final table above and more specifically at the error metric we will choose a cost parameter equal to 19 for our final svm in the section PCA and visualisation, as we see that with a highest cost, ther error rate is increasing from 0.0357 to 0.0575.

5 Non-linear Support Vector Machine

The main difference between the linear and the nonlinear case is that we project the points in an enlarged feature space rather than working in the input space. Instead of trying to fit a nonlinear model, we map the problem to a new space by doing a nonlinear transformation using, $v(x_i)$, a suitably chosen basis function and then use a linear model in this new space.

Afterward, the Kernel trick occurs. The idea is to replace the inner product of the basis function, which comes from the Lagrange dual function resolution, by a kernel function. So instead of mapping the whole data points in the new space and working there, we directly apply the kernel function in the original space and work on the Kernel matrix which is on the input space. Thus we can compute the dot product between any pair of data points in the transformed space without explicitly computing the transformation for all data points. We worked with the Kernel matrix and her pairwise similarities. It results in a decreasing of the computational time of our algorithm.

The kernel function can take various form such as the RBF of the polynomial which we are going to explain in

more details in the next sections.
Then, the procedure is the same as before. We use the more general case of the soft margin hyperplane here because we have no guarantee that the problem is linearly separable in this new space. The equation of the hyperplane is given as :

$$h(x) = \sum_{i=1}^{n} \alpha_i y_i K(x_i, x) + b$$

where $K(x_i, x)$ is the kernel function, α_i is a Lagrangian multiplier $0 \leq \alpha_i \leq C$ and y_i is the class belonging (-1, 1).
The objective function stays as follow :

$$\min_{\boldsymbol{w},b,\boldsymbol{\xi}} \left(\frac{1}{2}\|\boldsymbol{w}\|^2 + C \sum_{i=1}^{n} \xi_i^k \right)$$

5.1 Radial Kernel SVM

The RBF kernel corresponds to a transformation that maps the data into an infinite-dimensional space. This means that the transformed data points can have an infinite number of dimensions. It's a popular choice because it can handle non-linearly separable data and has only one hyperparameter to tune. It uses the radial basis function. His distribution graph looks like a Gaussian Distribution curve (see article of (Czarnecki & Tabor, 2014) in the appendices) and is of the following from :

$$K(x_i, x_j) = e^{-\gamma||x_i - x_j||^2}$$

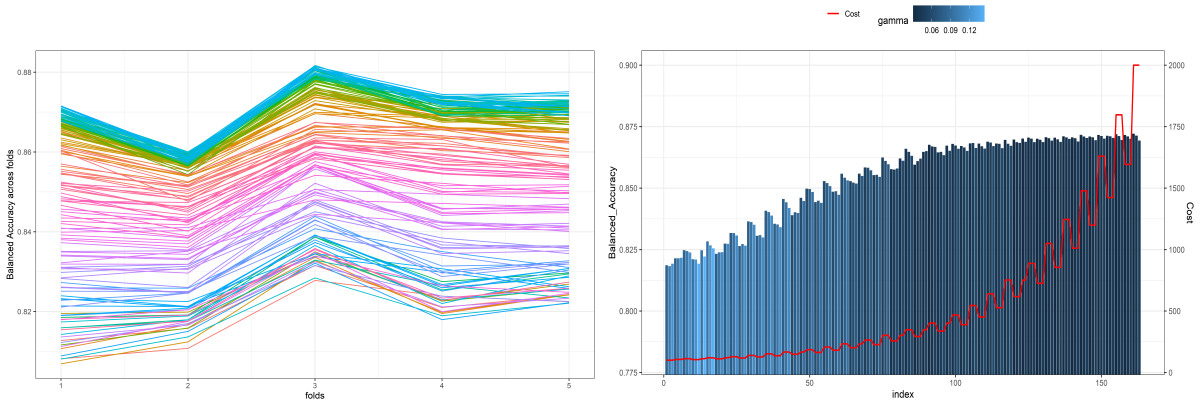
where, x_i and x_j are two data points in the feature space, $\gamma = \frac{1}{2\sigma^2}$ is a hyperparameter that controls the width of the Gaussian density with σ^2 the spread parameter of the Gaussian density function, and $||x_i - x_j||^2$ is the squared Euclidean distance between the two data points, their similarity. The RBF kernel can lead to over-fitting if the gamma parameter is set too high. This can cause the SVM to learn noise in the data instead of the underlying patterns. On the other hand, the small gamma produce smoother decision boundaries because they produce smoother feature space mappings. But by setting it too low could results in the incapacity of capturing the underlying patterns in the data. In other words, gamma sets the influence of the points : with a high gamma the nearby points will have high influence, with a low gamma far away points will also be considered to find the decision boundary.

Model	C	γ	δ^*	support_V	F1_score	Balanced_Accuracy	Precision	Recall	Detection_r	L_SV	R_SV	error
SVM Kernel 1	1	0.005	3.3391 e-4	6682	0.6462	0.7387	1	0.4773	0.0696	3680	3001	0.07624
SVM Kernel 2	1	1.5	2.4077e-04	19957	0.3092	0.5915	0.9569	0.1845	0.0269	15608	4353	0.1202
SVM Kernel 3	10	0.005	4.5204e-05	5880	0.7411	0.7944	0.9997	0.5888	0.0859	3796	2173	0.0600
SVM Kernel 4	10	0.3	5.7754e-05	7569	0.7481	0.7921	0.8899	0.5968	0.0870	5068	2526	0.0696
SVM Kernel 5	17.5	0.0050	2.7470e-05	5745	0.7507	0.8005	1.0000	0.6010	0.0877	3539	2100	0.0582
SVM Kernel 6	17.5	0.1000	3.2864e-05	5094	0.7645	0.8167	0.9513	0.6391	0.0932	3138	1986	0.0574
SVM Kernel 7	17.5	0.5	4.2960e-05	9492	0.6446	0.7613	0.7810	0.5490	0.0801	6671	2930	0.0883
SVM Kernel 8	25	0.1	2.4049e-05	5003	0.7690	0.8215	0.9420	0.6497	0.0948	3115	1936	0.0569
SVM Kernel 9	107.50	0.0614	6.2517e-06	4304	0.8144	0.8529	0.9513	0.7119	0.1038	2684	1683	0.0473
SVM Kernel 10	222.90	0.0393	3.1328e-06	3894	0.8360	0.8660	0.9672	0.7363	0.1074	2368	1559	0.0421
SVM Kernel 11	1150.13	0.0377	7.4137e-07	3554	0.8365	0.8710	0.9472	0.7491	0.1093	2231	1353	0.0427
SVM Kernel 12	1656.18	0.0377	5.3594e-07	3521	0.8356	0.8711	0.9434	0.7500	0.1094	2228	1313	0.0430
SVM Kernel 13	1987.42	0.0302	4.3896e-07	3387	0.8394	0.8721	0.9522	0.7507	0.1095	2136	1300	0.0419

Table 3: SVM RBF Kernel Model Performance using hinge loss

As you can see in on plot below and on our Table 3, our variable of interest, the detection rate, with SVM RBF

Kernel converges to $10,95\%$. To recall it, the target is a detection rate of $14,6\%$. We are quiet happy with this result since we also have a good balanced accuracy, at $87,21\%$ and a small error. The graphs below underline the convergence. As you can see, as the iterations goes by, we are losing precision while converging, but it is not a problem since both the balanced accuracy and the detection rate increase and simultaneously the number of support vectors drops. You will find the results of the RBF kernel using hinge loss in appendices section 8.1.



5.2 Polynomial Kernel SVM

As for the *Radial Kernel SVM*, the kernel trick will be helpful to build the polynomial one as once again, the polynomial kernel will be used to transform the input data from its original space to a higher-dimensional feature space, where it may be easier to separate the 2 classes. Then, he *polynomial kernel* is defined as:

$$K_d(x_i, x_j) = (\gamma x_i^T x_j + r)^d = \sum_{k=1}^{d} \binom{d}{k} r^{d-k} (\gamma x_i^T x_j)^k$$

where x_i and x_j are input feature vectors, d is the degree of the polynomial, γ is a scaling factor, and r is a coefficient and thus each pair of input vectors is represented as a polynomial of degree d with higher degrees allowing for more complex interactions between features.

This kind of kernel is often chosen to improve the classification performance on nonlinear datasets, that is non linearly separable, what we have concluded for our *loan* database. However, it's essential to note that using a *polynomial kernel* with a high degree can lead to over-fitting on the training data, as the decision boundary may become too complex and fit the noise in the data. Therefore, once again the choice of the hyperparameters should be carefully selected through grid-search and cross-validation to avoid this behaviour and maximize generalization performance. Even though we have performed a grid search with a rate covering a wide range of parameters we will only report in the table the most conclusive results. You will find these results in the appendices section Polynomial Kernel results using hinge loss.

6 PCA and visualisation

The goal of PCA is to find a new set of features, called principal components z, which are linear combinations of the original features and that capture the maximum amount of variance in the data. The first principal component, denoted by z_1, is given by the linear combination of the variables $x_1, x_2, \ldots, x_p$:

$$X = \begin{pmatrix} x_{1,1} & x_{1,2} & \dots & x_{1,p} \\ x_{2,1} & x_{2,2} & \dots & x_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n,1} & x_{n,2} & \dots & x_{n,p} \end{pmatrix} \qquad z_1 = \phi_{1,1}x_1 + \phi_{2,1}x_2 + \cdots + \phi_{p,1}x_p$$

where $\phi_{1,1}, \phi_{2,1}, \dots, \phi_{p,1}$ are the coefficients or loading's of the first principal component and are chosen such that the variance of z_1 is maximized subject to the constraint that $\sum_{i=1}^{p} \phi_{i1}^2 = 1$.
Mathematically, *PCA* can be formulated as if we let X be an $n \times p$ matrix containing n observations of p variables then by first centering the data and by subtracting the mean of each variable from each observation, then we can compute the covariance matrix S of the centered data:

$$S = \frac{1}{n-1} X^T X$$

Next, we can compute the eigenvectors and eigenvalues of S as an eigenvector of S is a non-zero vector v that satisfies the equation:

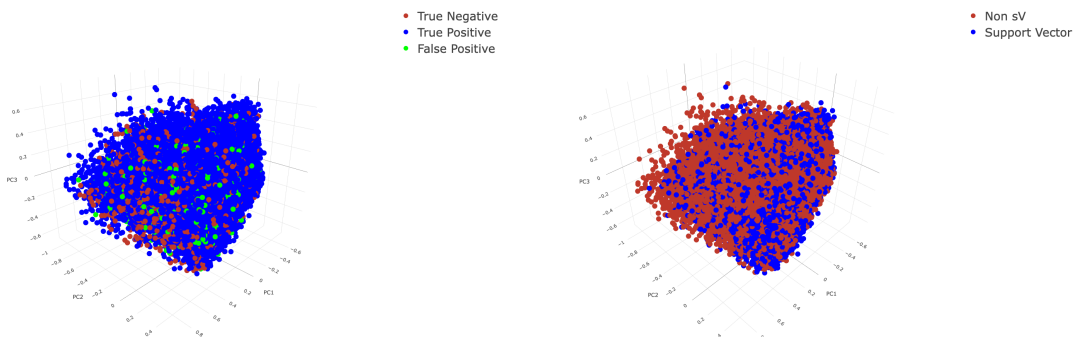
$$Sv = \lambda v$$

where λ is a scalar, called the eigenvalue that corresponds to the amount of variance explained by each eigenvector such that their respective eigenvectors of S form an orthonormal basis for the d-dimensional space. Then, we can use the eigenvectors of S to transform the original data into a new coordinate system as the first principal component is the eigenvector with the largest eigenvalue and thus that captures the biggest amount of variance. As we want to represent our contracts in a *3D* space we will thus choose the first 3 principal components and then the transformed data matrix Z is obtained by multiplying the centered data matrix X by the matrix of 3 first eigenvectors V:

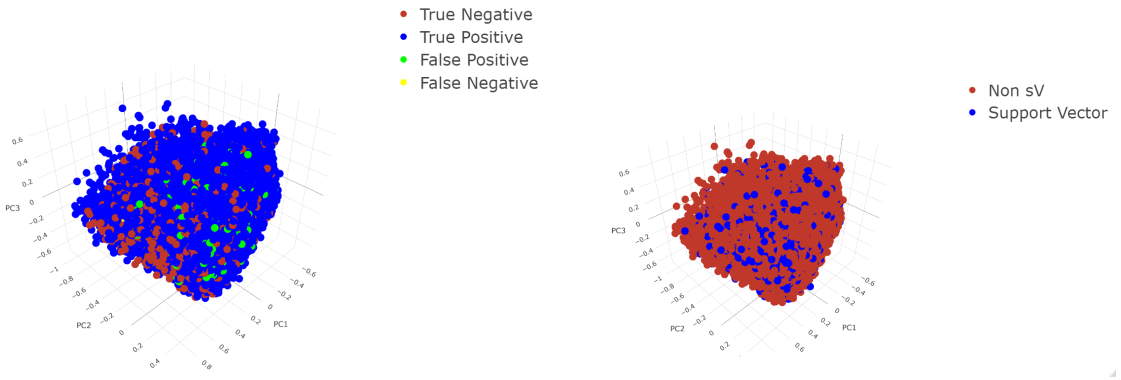
$$Z = XV$$

Thus in order to close this analysis we will display the representation of our data points with PCA and display colors corresponding to *true positive, true negative* that are well classified and *false positive, false negative* that are misclassified. On another plot we color the support vectors to assess for the distribution of points associated to the margin that are the closest to the decision boundary. But we want to add that as it has been done in the paper of (Kempfert, Wang, Chen, & Wong, 2019), it has not been considered to perform other dimensionality reduction such as the *Supervised kernel PCA* prior to our *Support vector machine* classification that may have improved our results.

6.1 PCA visualisation for linear Kernel ($C = 19$)



6.2 PCA visualisation for radial Kernel ($C = 1987.42$, $\gamma = 0.0301$)



7 Conclusion

As it's time to finish our journey through the application of support vector machines to our contract database we will choose the one that gives the highest performances and interpret our results at best. Our results showed that support vector machines are highly effective in predicting loan defaults, achieving a high *accuracy rate* and demonstrating the ability to handle complex and nonlinear relationships between the input features. This project highlights the potential of support vector machines as a valuable tool in the financial industry for risk assessment. It offers lenders an effective means of identifying high-risk loan applicants and reducing the risk of default. Our main objective was to improve the detection rate while maintaining a stable *balanced accuracy*a high *precision* and a low *error*. We have to add that a anomaly analysis has been performed, it's reasonable to expect not perfect results since our dataset contains 4.6% of outliers and that the proportion of anomalies in the *Charged Off* level of the *loan status* target can reach 15% for some features.

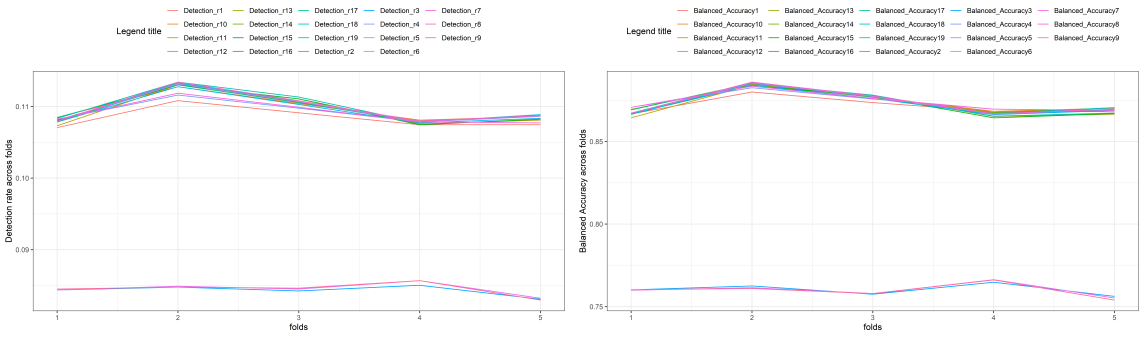
The best performing model is the support vector machine with polynomial kernel 8 ($C = 15, d = 3, \gamma = 1, r = 1$) that gives performances of 87.81% and 11.13% for the *balanced accuracy* and *detection rate* respectively. We keep track of the error rate to make sure that this *svm* configuration is the one that perform better than others and as it's the lowest one we get across all kernels with $error = 0.0336$ and $dispersion = 0.00393$. The second best performing model is with RBF kernel 13 with a quadratic loss function :($C = 1987.42, \gamma = 0.0242$) with a *balanced accuracy* of 87.39% and a *detection rate* of 10.98% while have a small error of 4.08%. Nonetheless every model tried performed almost similarly at optimum, even, at our surprise, the linear one. As it was asked to analyse the results for one of them, we chose to visualize the radial kernel one through PCA as the convergence across the folds was smoother. We encountered problem with the running time and we concluded, as neither expert coder nor yet data scientist, that this could be problematic in the industry. We know that the polynomial kernel with an higher cost and a quadratic loss function would perform better but given our available computational power, we decided to stop there due to lack time.

8 Appendices

8.1 Graph and results about Kernel RBF using a Quadratic loss

Model	C	γ	δ^*	support_V	F1_score	Balanced_Accuracy	Precision	Recall	Detection_r	L_SV	R_SV	error
SVM Kernel 1	1150.13	0.005	5.8541e-07	3466	0.8515	0.8713	0.9971	0.7430	0.1083	1938	1459	0.0378
SVM Kernel 2	1150.13	0.0302	7.1611e-07	3442	0.8427	0.8724	0.9617	0.7500	0.1094	2081	1327	0.0408
SVM Kernel 3	1150.13	0.5	5.3951e-06	8210	0.6038	0.7602	0.6323	0.5779	0.0843	5720	2601	0.1106
SVM Kernel 4	1656.18	0.005	4.2305e-07	3302	0.8541	0.8735	0.9964	0.7475	0.1090	1814	1406	0.0372
SVM Kernel 5	1656.18	0.0302	5.1732e-07	3406	0.8431	0.8732	0.9598	0.7517	0.1097	2067	1292	0.0408
SVM Kernel 6	1656.18	0.5	4.9495e-06	8158	0.6018	0.7602	0.6259	0.5795	0.0845	5664	2600	0.1119
SVM Kernel 7	1987.42	0.005	3.5984e-07	3246	0.8543	0.8737	0.9960	0.7480	0.1091	1788	1375	0.0372
SVM Kernel 8	1987.42	0.0302	4.3923e-07	3379	0.8425	0.8734	0.9568	0.7526	0.1098	2087	1279	0.0410
SVM Kernel 9	1987.42	0.5	4.7890e-06	8143	0.6008	0.7598	0.6240	0.5793	0.0845	5653	2601	0.1123
SVM Kernel 10	1589.94	0.0242	5.1987e-07	3331	0.8464	0.8738	0.9682	0.7519	0.1097	2010	1293	0.0398
SVM Kernel 11	1589.94	0.0302	5.3656e-07	3407	0.8430	0.8731	0.9600	0.7516	0.1096	2063	1297	0.0408
SVM Kernel 12	1589.94	0.0362	5.5236-07	3489	0.8380	0.8717	0.9494	0.7501	0.1094	2161	1301	0.0423
SVM Kernel 13	1987.42	0.0242	4.2535e-07	3286	0.8457	0.8739	0.9655	0.7524	0.1098	1998	1292	0.0408
SVM Kernel 14	1987.42	0.0302	4.3923e-07	3379	0.8425	0.8734	0.9568	0.7526	0.1098	2087	1279	0.0410
SVM Kernel 15	1987.42	0.0362	4.5210e-07	3488	0.8370	0.8719	0.9450	0.7212	0.1096	2168	1293	0.0427
SVM Kernel 16	2384.90	0.0242	3.6052e-07	3264	0.8453	0.8738	0.9649	0.7523	0.1097	1991	1258	0.0401
SVM Kernel 17	2384.90	0.0302	3.7254e-07	3357	0.8410	0.8729	0.9540	0.7521	0.1097	2067	1267	0.0414
SVM Kernel 18	2384.90	0.0362	3.8355e-07	3444	0.8374	0.8726	0.9434	0.7530	0.1098	2165	1280	0.0426
SVM Kernel 19	1907.92	0.0290	4.5290e-07	3361	0.8437	0.8735	0.9605	0.7523	0.1097	2052	1281	0.0406

Table 4: SVM RBF Kernel Model Performance



8.2 Polynomial Kernel results using hinge loss

Model	C	d	γ	r	δ^*	support_V	F1_score	Balanced_Accuracy	Precision	Recall	Detection_r	error	dispersion
SVM Poly 1	7	2	1	1	1.032e-04	11992	0.8491	0.8689	1	0.7378	0.1076	0.03765	0.004361
SVM Poly 2	7	3	10	1	4.775e-03	11673	0.4845	0.7778	0.343	0.8421	0.1228	0.0931	0.00594
SVM Poly 3	7	4	10	1	1.209e-01	6507	0.4601	0.7484	0.3321	0.8171	0.1192	0.1047	0.00754
SVM Poly 4	11	1	1	1	6.132e-05	4644	0.8546	0.8731	1	0.7462	0.1088	0.0401	0.00469
SVM Poly 5	15	1	10	1	5.61e-05	4387	0.8568	0.8748	1	0.749	0.1093	0.03712	0.00456
SVM Poly 6	15	1	10	0	5.617e-05	4476	0.8568	0.8748	1	0.7496	0.1093	0.03668	0.00436
SVM Poly 7	15	3	0.5	1	5.4088e-05	9605	0.854	0.8748	0.9895	0.7510	0.1096	0.03564	0.00427
SVM Poly 8	15	3	1	1	6.349e-05	12776	0.8466	0.8781	0.9542	0.7628	0.1113	0.0336	0.00393

Table 5: SVM nonlinear Model Performance1

8.3 R Code for linear kernel grid search & CV

```r
folds <- createFolds(data_clustering$loan_status, k = 5)


# Define lambda and alpha values to search over
Cost <- c(5,7,9,11,13,15,17,19)
# Initialize matrix to store cross-validation results
cv_results_svm <- matrix(NA, nrow = length(Cost), ncol = 8)
colnames(cv_results_svm) <- c("Cost","support_V", "Sensitivity", "F1_score", "Balanced_
    Accuracy", "Precision", "Recall", "Detection_r")
```

```r
row_idx <- 1
set.seed(87031800)
results_list <- list()

# Loop over lambda and alpha values
for (j in 1:length(Cost)) {
  # Initialize vector to store Dev.Bern results for this combination of lambda and alpha
  support_V <- rep(NA, 5)
  Sensitivity <- rep(NA,5)
  F1_score <- rep(NA, 5)
  Balanced_Accuracy <- rep(NA,5)
  Precision <- rep(NA, 5)
  Recall <- rep(NA, 5)
  Detection_r <- rep(NA, 5)

  # Loop over folds
  nfolds <- 0
  for (fold in 1:5) {
    # Split data into training and test sets for this fold
    train_idx <- setdiff(seq_len(nrow(data_clustering)), folds[[fold]])
    test_idx <- folds[[fold]]
    x_train <- data_clustering[train_idx, ]
    y_train <- data_clustering$loan_status[train_idx]
    x_test <- data_clustering[test_idx, ]
    y_test <- data_clustering$loan_status[test_idx]
    print("go")
    print(Cost[j])
    # Fit model with elastic net regularization
    final_model <- svm(loan_status ~ ., data = x_train, kernel = "linear", cost = Cost[j
    ])
    print("fin")
    # Compute odds

    y_pred <- predict(final_model, newdata = x_test)
    confu <- confusionMatrix(data = y_pred, y_test)
    print(confu)
    sensi <- confu$byClass[["Sensitivity"]]
    f1 <- confu$byClass[["F1"]]
    balanced <- confu$byClass[["Balanced Accuracy"]]
    prec <- confu$byClass[["Precision"]]
    rec <- confu$byClass[["Recall"]]
    det <- confu$byClass[["Detection Rate"]]

    # Store results for this fold
    support_V[fold] <- final_model$tot.nSV
    print(support_V)
    Sensitivity[fold] <- sensi
    print(Sensitivity)
    F1_score[fold] <- f1
    print(F1_score)
    Balanced_Accuracy[fold] <- balanced
    print(Balanced_Accuracy)
    Precision[fold] <- prec
    print(Precision)
    Recall[fold] <- rec
    print(Recall)
    Detection_r[fold] <- det
    nfolds <- nfolds + 1
  }
  results_list <- cbind(results_list, list(support_V, Sensitivity, F1_score, Balanced_
    Accuracy, Precision, Recall, Detection_r))
```

```r
  print(results_list)
  # Compute mean Dev.Bern loss across folds for this combination of lambda and alpha
  mean_support_V <- mean(support_V)
  mean_Sensitivity <- mean(Sensitivity)
  mean_F1_score <- mean(F1_score)
  mean_Balanced_Accuracy <- mean(Balanced_Accuracy)
  mean_Precision <- mean(Precision)
  mean_Recall <- mean(Recall)
  mean_Detection_r <- mean(Detection_r)

  # Store results in cv_results matrix
  cv_results_svm[row_idx, 1] <- Cost[j]
  cv_results_svm[row_idx, 2:8] <- c(mean_support_V, mean_Sensitivity, mean_F1_score,
    mean_Balanced_Accuracy, mean_Precision,mean_Recall, mean_Detection_r)
  print(cv_results_svm)
  # Increment row index
  row_idx <- row_idx + 1
}
```

8.4 R Code for RBF Kernel grid search & CV

```r
folds <- createFolds(data_clustering$loan_status, k = 5)

# Define lambda and alpha values to search over
cost <- c(1150.13,1656.18,1987.42)
gamma = c(0.005, 0.0302, 0.5)
previous_error  = Inf
previous_acc = 0
previous_detection_r = Inf


decay = 0
conver = FALSE
# Initialize matrix to store cross-validation results
cv_results<- matrix(NA, nrow = 9, ncol = 13)
colnames(cv_results) <- c("Cost","gamma","margin",
                          "support_V", "Sensitivity", "F1_score",
                          "Balanced_Accuracy", "Precision", "Recall",
                          "Detection_r","L_SV","R_SV",
                          "error")

results_list <- list()
full_results = list()
#convergebce loop
while (conver == FALSE) {
  print(cost)
  print(gamma)
  set.seed(696460)

  row_idx <- 1
# Loop over gamma and alpha values
  for (c in 1:length(cost)) {
    for (gam in 1:length(gamma)){
    # Initialize vector to store Dev.Bern results for this combination of lambda and
    alpha
    support_V <- rep(NA, 5)
    Sensitivity <- rep(NA,5)
    F1_score <- rep(NA, 5)
    Balanced_Accuracy <- rep(NA,5)
    Precision <- rep(NA, 5)
```

```
    Recall <- rep(NA, 5)
    Detection_r <- rep(NA, 5)

    margin = rep(NA,5)
    L_SV = rep(NA,5)
    R_SV = rep(NA,5)
    error = rep(NA,5)

    # Loop over folds
    nfolds <- 0
    for (fold in 1:5) {
      # Split data into training and test sets for this fold
      train_idx <- setdiff(seq_len(nrow(data_clustering)), folds[[fold]])
      test_idx <- folds[[fold]]
      x_train <- data_clustering[train_idx, ]
      y_train <- data_clustering$loan_status[train_idx]
      x_test <- data_clustering[test_idx, ]
      y_test <- data_clustering$loan_status[test_idx]

      # Fit model with elastic net regularization
      print(c("go in",fold))

      final_model <- svm(loan_status ~ ., data = x_train, kernel = "radial", cost = cost
    [c], gamma = gamma[gam],
                         type = "C-classification", loss = "squared-hinge")
      predictions = predict(final_model, x_test)
      error[fold] = mean(predictions != y_test)

      print("out")

      # Compute odds

      y_pred <- predict(final_model, newdata = x_test)
      confu <- confusionMatrix(data = y_pred, y_test)

      sensi <- confu$byClass[["Sensitivity"]]
      f1 <- confu$byClass[["F1"]]
      balanced <- confu$byClass[["Balanced Accuracy"]]
      prec <- confu$byClass[["Precision"]]
      rec <- confu$byClass[["Recall"]]
      det <- confu$byClass[["Detection Rate"]]


      # Store results for this fold
      support_V[fold] <- final_model$tot.nSV
      Sensitivity[fold] <- sensi
      F1_score[fold] <- f1
      Balanced_Accuracy[fold] <- balanced
      Precision[fold] <- prec
      Recall[fold] <- rec
      Detection_r[fold] <- det

      margin[fold] = 2/norm(final_model$coefs)
      L_SV = final_model$nSV[1]
      R_SV = final_model$nSV[2]
      nfolds <- nfolds + 1
    }
    #end fold loop
    print("out of fold")
```

```
    results_list = cbind(results_list,list(support_V, Sensitivity, F1_score, Balanced_
    Accuracy, Precision, Recall, Detection_r,margin,error,L_SV,R_SV))

    print(results_list)
    # Compute mean Dev.Bern loss across folds for this combination of lambda and alpha
    mean_support_V <- mean(support_V)
    mean_Sensitivity <- mean(Sensitivity)
    mean_F1_score <- mean(F1_score)
    mean_Balanced_Accuracy <- mean(Balanced_Accuracy)
    mean_Precision <- mean(Precision)
    mean_Recall <- mean(Recall)
    mean_Detection_r <- mean(Detection_r)

    mean_margin = mean(margin)
    mean_L_SV = mean(L_SV)
    mean_R_SV = mean(R_SV)
    mean_error = mean(error)


    # Store results in cv_results matrix
    cv_results[row_idx, 1:2] <- c(cost[c],gamma[gam])
    cv_results[row_idx, 3:13] <- c(mean_margin, mean_support_V, mean_Sensitivity,
                                   mean_F1_score, mean_Balanced_Accuracy, mean_Precision
    ,
                                   mean_Recall, mean_Detection_r,mean_L_SV,
                                   mean_R_SV,mean_error)
    print(cv_results)
    # Increment row index
    row_idx <- row_idx + 1
    }
  }
  #######end double if c, gam

  print("end double")
  #to keep the previous retults
  full_results = cbind(full_results,cv_results)
  print(full_results)

  #finding the optimal parameter based on the smallest detection_rate
  #we find the index and take the minimal difference
  opt_detec = abs(0.1406 - cv_results[,10])
  opt_index = which.min(opt_detec)
  best_detection_r = opt_detec[opt_index]

  print(c("opt_detec"))
  print(c("opt_index",opt_index))
  #we assign the best hyperparameters with the opt index
  best_cost = cv_results[opt_index,1]
  best_gamma = cv_results[opt_index,2]
  best_acc = cv_results[opt_index,7]


  #update the new cost and gamma
  # care implementation of the decay hasn't been tested
  cost = c(best_cost - (best_cost /5),best_cost, best_cost + (best_cost/5))
  gamma = c(best_gamma- (best_gamma /5),best_gamma, best_gamma + (best_gamma/5))

  print(c(cost,gamma))

  #test of convergence based on the detection_rate
  if (best_detection_r >= previous_detection_r) {
```

```
    print("it converged")
    conver <- TRUE
  }
  previous_detection_r = best_detection_r

  print(c(best_detection_r,previous_detection_r))
  print(full_results)
}
```

8.5 R Code for polynomial Kernel grid search & CV

```
    folds <- createFolds(data_clustering$loan_status, k = 5)

# Define lambda and alpha values to search over
Cost <- c(7,11,15,19,25, 50,100)
Degree <- c(1,2,3,4)
gamma <- c(0.01,0.1,0.5,1, 10)
Coefs <- c(0, 1)
# Initialize matrix to store cross-validation results
cv_results_svm <- matrix(NA, nrow = length(Cost)*length(Degree)*length(gamma)*length(
    Coefs), ncol = 14)
colnames(cv_results_svm) <- c("Cost","Degree","Gamma","Coef0", "margin", "support_V", "
    Sensitivity", "F1_score", "Balanced_Accuracy", "Precision", "Recall", "Detection_r",
     "Lsv", "Rsv")
row_idx <- 1
results_list <- list()

# Loop over lambda and alpha values
for (i in 1:length(Cost)) {
  for (j in 1:length(Degree)) {
    for (z in 1:length(gamma)){
      for (a in 1:length(Coefs)) {
        set.seed(87031800)
        # Initialize vector to store Dev.Bern results for this combination of lambda and
     alpha
        support_V <- rep(NA, 5)
        Sensitivity <- rep(NA,5)
        F1_score <- rep(NA, 5)
        Balanced_Accuracy <- rep(NA,5)
        Precision <- rep(NA, 5)
        Recall <- rep(NA, 5)
        Detection_r <- rep(NA, 5)
        margin <- rep(NA,5)
        Lsv <- rep(NA,5)
        Rsv <- rep(NA, 5)

        # Loop over folds
        nfolds <- 0
        for (fold in 1:5) {
          # Split data into training and test sets for this fold
          train_idx <- setdiff(seq_len(nrow(data_clustering)), folds[[fold]])
          test_idx <- folds[[fold]]
          x_train <- data_clustering[train_idx, ]
          y_train <- data_clustering$loan_status[train_idx]
          x_test <- data_clustering[test_idx, ]
          y_test <- data_clustering$loan_status[test_idx]
          print("go")
          print(c(Cost[i],Degree[j], gamma[z], Coefs[a]))
          # Fit model with elastic net regularization
```

```
45          final_model <- svm(loan_status ~ ., data = x_train, kernel = "polynomial",
    cost = Cost[i], degree = Degree[j], gamma = gamma[z], coef0 = Coefs[a])
46          print("fin")
47          # Compute odds
48
49          y_pred <- predict(final_model, newdata = x_test)
50          confu <- confusionMatrix(data = y_pred, y_test)
51          print(confu)
52          sensi <- confu$byClass[["Sensitivity"]]
53          f1 <- confu$byClass[["F1"]]
54          balanced <- confu$byClass[["Balanced Accuracy"]]
55          prec <- confu$byClass[["Precision"]]
56          rec <- confu$byClass[["Recall"]]
57          det <- confu$byClass[["Detection Rate"]]
58
59          # Store results for this fold
60          margin[fold] <- 2/norm(final_model$coefs)
61          Lsv[fold] <- final_model$nSV[1]
62          Rsv[fold] <- final_model$nSV[2]
63          support_V[fold] <- final_model$tot.nSV
64          print(support_V)
65          Sensitivity[fold] <- sensi
66          print(Sensitivity)
67          F1_score[fold] <- f1
68          print(F1_score)
69          Balanced_Accuracy[fold] <- balanced
70          print(Balanced_Accuracy)
71          Precision[fold] <- prec
72          print(Precision)
73          Recall[fold] <- rec
74          print(Recall)
75          Detection_r[fold] <- det
76          nfolds <- nfolds + 1
77        }
78        results_list <- cbind(results_list, list(margin, support_V, Sensitivity, F1_
    score, Balanced_Accuracy, Precision, Recall, Detection_r, Lsv, Rsv))
79        print(results_list)
80        # Compute mean Dev.Bern loss across folds for this combination of lambda and
    alpha
81        mean_support_V <- mean(support_V)
82        mean_Sensitivity <- mean(Sensitivity)
83        mean_F1_score <- mean(F1_score)
84        mean_Balanced_Accuracy <- mean(Balanced_Accuracy)
85        mean_Precision <- mean(Precision)
86        mean_Recall <- mean(Recall)
87        mean_Detection_r <- mean(Detection_r)
88        mean_margin <- mean(margin)
89        mean_Lsv <- mean(Lsv)
90        mean_Rsv <- mean(Rsv)
91
92        # Store results in cv_results matrix
93        cv_results_svm[row_idx, 1:4] <- c(Cost[i], Degree[j], gamma[z], Coefs[a])
94        cv_results_svm[row_idx, 5:14] <- c(mean_margin, mean_support_V, mean_Sensitivity
    , mean_F1_score, mean_Balanced_Accuracy, mean_Precision,mean_Recall, mean_Detection_
    r, mean_Lsv, mean_Rsv)
95        print(cv_results_svm)
96        # Increment row index
97        row_idx <- row_idx + 1
98      }
99    }
100 }
```

```
}
```

8.6 R Code for PCA projection

```
# Train an SVM on the transformed data
svm <- svm(loan_status~ ., data = data_PCA, kernel = "radial", cost = 1987.42, gamma =
    0.0302)

# Predict SVM output for each data point
pred_labels <- predict(svm, data_PCA)
confusionMatrix(pred_labels, data_PCA$loan_status)
# Create a color vector for the data points based on the predicted labels
colors <- rep(3, length(pred_labels)) # initialize colors vector with 3s
for(i in 1:length(pred_labels)) {
  if(pred_labels[i] == "Charged Off" && y[i] == "Charged Off") {
    colors[i] <- 1
  } else if(pred_labels[i] == "Fully Paid" && y[i] == "Fully Paid") {
    colors[i] <- 2
  } else if (pred_labels[i]=="Charged Off" && y[i] == "Fully Paid" ) {

    colors[i] = 4
  }
}
to_comp <- table(colors)
to_comp

# PCA
pca <- prcomp(x)
pca.x <- pca$x[,1:3]
library(plot3D)
library(rgl)
library(plotly)
summary(pca)
library(factoextra)

png(filename = "contributions66.png", width = 10, height = 6, units = 'in', res=900) ##
    save the plot in files
fviz_eig(pca)
dev.off()

fviz_pca_var(pca,col.var = "contrib", gradient.cols = c("#00AFBB", "#E7B800", "#FC4E07")
    ,repel = TRUE)
pca

plot3d(pca$rotation[,1:3],
  texts=rownames(pca$rotation),
  col="red", cex=0.8

text3d(pca$rotation[,1:3],
  texts=rownames(pca$rotation),
  col="red", cex=0.8

coords <- NULL
for (i in 1:nrow(pca$rotation)) {
  coords <- rbind(coords,
                  rbind(c(0,0,0),
                        pca$rotation[i,1:3]))
}

lines3d(coords,
```

```
        col="red",
        lwd=1)

dev.off()
var <- get_pca_var(pca)
ind = get_pca_ind(pca)

mean(ind$cos2[,1]) # 0.3439
mean(ind$cos2[,2]) # 0.1778
mean(ind$cos2[,3]) #0.18293
mean(ind$cos2)
ind$coord
var$cos2


library("corrplot")
corrplot(var$cos2, is.corr=FALSE)
fviz_cos2(pca, choice = "var", axes = 1:3)
## Plot for true positive etc...

pca.fin <- as.data.frame(pca.x)
pca.fin <- cbind(pca.fin, colors)
pca.fin$colors <- factor(pca.fin$colors, levels = c("1", "2", "3","4"), labels = c("True
     Negative", "True Positive", "False Positive","False Negative"))

fig1 <- plot_ly(pca.fin, x = ~PC1, y = ~PC2, z = ~PC3, color = ~colors, colors = c('#
    BF382A', 'blue', 'green',"yellow"))
fig1 <- fig1 %>% add_markers()
fig1 <- fig1 %>% layout(scene = list(xaxis = list(title = 'PC1'),
                                     yaxis = list(title = 'PC2'),
                                     zaxis = list(title = 'PC3')),
                        legend = list(font = list(size = 30),
                                      itemsizing = "constant",
                                      itemwidth = 50,
                                      itemheight = 20))

fig1


## Plot for support vectors

supportvectors <- svm$index

pca.fin2 <- as.data.frame(pca.x)
colors <- rep(1, length(pca.fin2))
pca.fin2 <- cbind(pca.fin2, colors)
sv_indices <- match(supportvectors, rownames(pca.fin))

pca.fin2$colors[sv_indices] <- 2
pca.fin2$colors <- factor(pca.fin2$colors, levels = c("1", "2"), labels = c("Non sV", "
    Support Vector"))

fig2 <- plot_ly(pca.fin2, x = ~PC1, y = ~PC2, z = ~PC3, color = ~colors, colors = c('#
    BF382A', 'blue'))
fig2 <- fig2%>% add_markers()
fig2 <- fig2 %>% layout(scene = list(xaxis = list(title = 'PC1'),
                                     yaxis = list(title = 'PC2'),
                                     zaxis = list(title = 'PC3')),
                        legend = list(font = list(size = 30),
                                      itemsizing = "constant",
                                      itemwidth = 50,
```

```
                                        itemheight = 20))
fig2
```

References

Czarnecki, W. M., & Tabor, J. (2014). Cluster based RBF kernel for support vector machines. *CoRR*, *abs/1408.2869*. Retrieved from http://arxiv.org/abs/1408.2869

Hafner, C. (2021). Statistical machine learning and high dimensional data analysis. *Slides K-means & Support Vector Machines*, *22*(43), 1–40. Retrieved from https://moodle.uclouvain.be/course/view.php?id=6116

Hastie, T., Tibshirani, R., & Friedman, J. (2009). *The elements of statistical learning: data mining, inference and prediction* (Second ed.). Springer. Retrieved from http://www-stat.stanford.edu/~tibs/ElemStatLearn/

Kempfert, K. C., Wang, Y., Chen, C., & Wong, S. W. K. (2019). *A comparison study on nonlinear dimension reduction methods with kernel variations: Visualization, optimization and classification.* Retrieved from https://arxiv.org/abs/1910.02114

Umoh, U. A., Udo, E., & Nyoho, E. E. (2019). Support vector machine-based fire outbreak detection system. *CoRR*, *abs/1906.05655*. Retrieved from http://arxiv.org/abs/1906.05655