ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

DEPARTMENT OF STATISTICAL SCIENCES

SECOND CYCLE DEGREE IN

STATISTICAL SCIENCES

LOCAL EFFECTS AND NON-LINEAR MODELING FOR IMPROVED PREDICTION ACCURACY IN REGRESSION AND CLASSIFICATION: AN R PACKAGE IMPLEMENTATION

Supervisor

Prof. Cinzia Viroli

Co-Supervisor

Prof. André Osório Falcão

Defended by

Emanuele Tartaglione

0001037146

Graduation Session II / December / 2023

Academic Year 2022/2023

Acknowledgements

I would like to thank professor Cinzia Viroli and professor André Osório Falcão, my supervisors and guides, who carefully supported my work.

Abstract

A new algorithm has been developed that uses local effects and very small non-linear models for classification and regression. The algorithm is designed to identify non-modelable instances and improve the accuracy of predictions. The algorithm works by constructing a set of small non-linear models, each of which is specialized for a local region of the data space. By using small models and focusing on local effects, the algorithm is able to identify non-modelable instances that would be missed by traditional models. The algorithm has shown to have high accuracy and efficiency in experiments on cheminformatics datasets but it is expected to be of general use. There is an implementation of this algorithm in Python and it is desirable to have running code in R so that the method can reach other communities. An R implementation would further allow to test the performance of the method with R benchmarks. It is expected a full implementation of the algorithm in R, making it available in CRAN and write the full documentation for the library.

Contents

Chapter 1

Introduction

The work that I am presenting with this thesis consists into a translation of a machine learning algorithm developed by Professor André Osório Falcão, from the University of Lisbon operating in the Faculty of Sciences. He was my professor in the course Data Mining during my time as an Erasmus student. His work regards mainly machine learning and data mining techniques for biological systems in bioinformatics and cheminformatics. He recently developed an algorithm called MS-QSAR (Metric Space – Quantitative Structure Activity Relationship) to evaluate, test, predict and screen chemical databases. The algorithm is implemented in Python, and my work consisted in translating a version of MSQSAR in R.

While he is an excellent professor, I wanted to work with Professor A. O. Falcão on the algorithm he proposed because is innovative and can have a huge success in the data science and machine learning community. It will probably have more success in the cheminformatics field since it was created to analyse chemical data, in particular datasets where the values are in between 0 and 1; but working further to optimize the algorithm, I believe there are some possibilities that the algorithm can be a valid alternative to the already known other supervised methods.

The idea behind developing a R version of the algorithm is based on different objectives: first, it will reach a wider community and second, it will be possible to perform the algorithm on all the types of datasets, without the need to scale the datasets. The R version of the MS-QSAR focuses only on the prediction part of the Python version of the algorithm and it can perform both regression and classification. The algorithm makes use of already known supervised learning techniques, like random forest and support vector machines.

Chapter 2

State of the Art

2.1 Supervised Learning

Supervised learning is a paradigm within the broader domain of machine learning where an algorithm is trained on a labelled dataset. In this context, ”labeled” means that each example in the dataset is paired with the correct output. The primary objective of supervised learning is to construct a model that can make accurate predictions on unseen data, based on the patterns it has discerned from the training data.

The initial dataset is split into training data and test data. The former is the dataset on which the algorithm is trained. It consists of input-output pairs, where the inputs are often referred to as features or attributes, and the outputs are often called labels or targets. Hence, training a model with training data means that the model looks for the relationship between inputs and outputs. The model is a more or less complex mathematical system that captures the relationships between given inputs and corresponding outputs. The test dataset is that on which the algorithm is tested: in this way the performance of the algorithm is evaluated. The more the model is trained, the more accurate the results will be. During the training phase, it is possible to calibrate the model, i.e., to find some parameters that allow to obtain the best results. Once the model is trained and results tested with the test dataset, it is possible to perform the predictions on data it has never encountered before.

Types of Supervised Learning:

- Regression: The task where the output variable is continuous. Examples include predicting house prices or stock prices.
- Classification: The task where the output variable is categorical. Examples include identifying whether an email is spam or not or classifying images into categories.

Supervised learning has a myriad of applications across various domains, including finance, healthcare, marketing, and more. For instance, it can be used for credit scoring in banking, tumour detection in medical imaging, and customer segmentation in marketing.

Despite having high accuracy and being very useful in terms of computational complexity, it has also some drawbacks. For instance, the performance of supervised learning models is heavily dependent on the quality of the training data: noisy, inconsistent, or incomplete data can adversely affect model performance. Another challenge regards the model whose results are difficult to interpretable: some advanced models, like deep neural networks, can be ”black boxes,” making it challenging to understand their decision-making process. Moreover, acquiring labelled data can be expensive and time-consuming, especially for large datasets.

In conclusion, supervised learning is a foundational approach in machine learning that has facilitated numerous advancements in various fields. However, like all methodologies, it comes with its own set of challenges that researchers and practitioners must address.

Among the popular algorithms of supervised learning there are the Support Vector Machines (SVM) and Random Forest (RF).

2.1.1 Support Vector Machine

Support vector machines (SVMs) Cortes & Vapnik (1995) operate on a few basic assumptions. Central to SVM performance are the ”support vectors”, which are the data points that are closest to the decision boundary (or hyperplane). These vectors strongly influence the orientation and position of the hyperplane. The hyperplane itself is a linear decision surface that divides the space into two distinct groups. One of the main goals of SVM is to identify the hyperplane that gives the maximum ”margin”. This margin is defined as the distance between the hyperplane and the nearest data point from one of the two classes.

However, not all data is linearly separable at its origin. For such datasets, SVMs use a technique called ”Kernel Trick”. This approach involves mapping the data to a higher dimensional space that can be separated linearly. There are many kernel functions that contribute to this transformation, some popular ones being polynomial kernels, radial basis function (RBF) kernels and sigmoid kernels.

Several advantages come into play when evaluating the utility of SVMs. They perform remarkably well in high-dimensional environments and can perform well even when the number of dimensions exceeds the number of samples. Furthermore, SVMs use only a small portion of the training data to determine decision functions, especially the support vectors, making them more memory efficient.

However, SVM also has a few drawbacks. Their cubic time complexity makes them unsuitable for very large data. Moreover, their performance may deteriorate when there is a large amount of noise in the data set.

2.1.2 Random Forests

Random Forest (RF) Ho (1995) is a supervised learning method used for regression and classification. It is based on creating multiple decision trees during training and assigning a class (for classification) or an average prediction (for regression) of individual trees.

It is based on two main concepts: Bootstrap aggregation (or bagging) and Decision trees. Random Forest applies bagging to sample data points and features. Each tree in the forest is constructed with a drawn sample with replacement from the training set. The Forests are generated using decision trees, especially CART (Classification and Regression Trees) algorithm.

It is a very common technique because it can handle large datasets with high dimensionality, it can address missing values and ensures high accuracy through bagging and provides a nice signal of feature importance. Furthermore, overfitting is reduced, due to the randomness introduced in its design. But it also has some disadvantages: it can be slow to determine if the number of trees is too large and it is not easy interpretable as decision trees.

In summary, although SVMs are powerful for some applications, especially when the data have an obvious separation distance, RFs are more versatile and can be used for a wide range of applications, and results in better performance in many situations.

2.2 Instance Based Learning

Instance-based learning, as opposed to model-based learning, operates by memorizing the training dataset and making predictions based on the similarity between instances. One of the most prominent algorithms in this category is the k-Nearest Neighbours (KNN) algorithm. In KNN, an input instance is classified based on the majority class of its 'k' closest training examples.

2.2.1 K-Nearest Neighbours

K-Nearest Neighbours (KNN) Altman (1992) operates on the principle of feature similarity. When a prediction is required for an unseen instance, the algorithm searches the training dataset to find the 'k' training examples that are closest to the instance. The "closeness" or similarity is typically measured using distance metrics, such as the Euclidean or Manhattan distances. For classification tasks, the input instance is then assigned the class that is most common among its 'k' nearest neighbours. For regression tasks, the average or median of the target values of the 'k' neighbours is taken as the prediction. One of the main advantages of KNN is its non-parametric nature, meaning it makes no explicit assumptions about the underlying data distribution. However, its reliance on distance calculations makes it computationally intensive, especially with large datasets. Despite its simplicity, KNN can achieve remarkable accuracy, especially when the choice of 'k' and the distance metric is appropriate for the given data.

To refine predictions, distance weightings can be incorporated, where closer neighbours exert more influence on the classification than those farther away. This ensures that not all neighbours contribute equally to the decision-making process. Furthermore, the use of kernels, like the Gaussian or Radial Basis Function (RBF) kernel, can transform the input space, enabling the algorithm to capture complex, non-linear relationships. This transformation often aids in improving the accuracy and robustness of the KNN algorithm, especially in scenarios where data is not linearly separable in its original space. In essence, instance-based learning, through methods like KNN, distance weightings, and kernel techniques, offers a flexible and intuitive approach to classification and regression tasks in machine learning.

Chapter 3

Methods

3.1 Algorithm Description

MS-QSAR, as previously mentioned, stands for Metric Space for Quantitative Structure Activity Relationship, and is an algorithm developed to analyse chemical data, typically in contexts of in silico modelling approaches for pharmacological activity. It uses chemical similarities for inference and analysis of the chemical space. It performs four operations:

- Create QSAR models (quantitative and qualitative models).
- Evaluate data set modelability.
- Validate models.
- Screen datasets.

It was fully implemented in Python. It was developed taking advantages of libraries for numerical processing (numpy), machine learning (scikit-learn) and data processing (RDkit). While the first two libraries are famous in the Python community, the latter one is a library that contains several functions for dealing with molecules. A specific guide for installation of the msqsar package in Python and its use can be found at: https://github.com/aofalcao/msqsar .

msqsar is a .py file which contains four methods:

- *eval* - evaluates the quality of a given data set for making inference.
- *test* - checks predictions made against a data set.
- *infer* - makes predictions against a data set – no stats are provided.

- *screen* - screens a large database, identifying the most likely candidates (for classification only).

It has several control parameters:

- *in file_name* - the data set used for model building (required) (.sar format).
- *test file_name* - data set required for method=test (.sar format).
- *scr file_name* - file with data for screening (.smi format) .
- *out file_name* - file where the output is stored. If omitted, redirects to stdout.
- *fpR N* - fingerprint radius (default: 2).
- *fpNB N* - fingerprint number of bits (default: 1024).
- *max_dist f* - maximum distance for prediction (default: 0.9).
- *min_sims N* - minimum number of instances for modelling (default: 5).
- *max_sims* - maximum number of instances for modelling (default: 20).
- *algo (SVM — RF)* - machine learning algorithm (default: RF).
- *ntrees* - Number of trees in random forest RF (default: 20).
- *nprocs* - Number of processes if - parallel option enabled.

Output control options:

- *silent* - no intermediate output at all.
- *detail* - in the final output show the individual predictions.
- *nostats* - do not show end model statistics for methods eval and test.
- *parallel* - run the process in parallel (currently only for screening).

The algorithm works using the idea of the closest neighbours. After splitting the data into train and set, for each element of the test set, the closest neighbours are found according to some parameters:

- minimum size: the minimum number of points to be considered.

- maximum size: the maximum number of points to be considered.
- maximum distance: the maximum distance from the test point to the non-test points to be considered.

If, after controlling for the three parameters, there are not enough elements within the maximum distance, it means that the candidate elements as neighbours of the test point are less than the minimum size, than the point is not considered for the prediction phase. After having found enough neighbours, the algorithm computes the train and the test matrices. The test matrix is simply the matrix of distances between the test point and the neighbours, while the train matrix is the matrix of distances of the neighbours with themselves. To calculate the distances, the Euclidian distance is used, but it can be possible to use another kind of distance like the Manhattan distance or the Jaccard distance. Fig. 3.1 and Fig. 3.2 show the graphical representation of how the train and test matrix are made. The data for this visualization are taken from the dataset Boston Housing [5] (Harrison & Rubinfeld (1978)).

From the dataset Boston, I considered rows 10 to 30 and considering the variables CRIM and LSTAT [5] to have a bi-dimensional graph for a better understanding of the problem. In Fig. 3.1 the test data are represented in light green, whereas the train data are in orange. It is analysed just one point of the test set, the one in dark green. Fig. 3.2 offers a better visualization.

In Fig. 3.2 it is possible to see the distances (in blue) between the test point and the closest neighbours (red dots), that will form a test matrix. The red lines are the distances among the closest neighbours and will form the train matrix. Calling n the number of neighbours, the train matrix will be a $n\ x\ n$ matrix and the test matrix will be a $n\ x\ 1$ matrix. Then a model is performed for each test element, using the train matrix and a new element will be predicted using the test set. The algorithm is written using either a SVM model or a RF model, but theoretically any model can be used. The idea behind this algorithm is to increase the speed of SVM, since they can take long time due to their computational complexity, some work might be done to have a faster version.

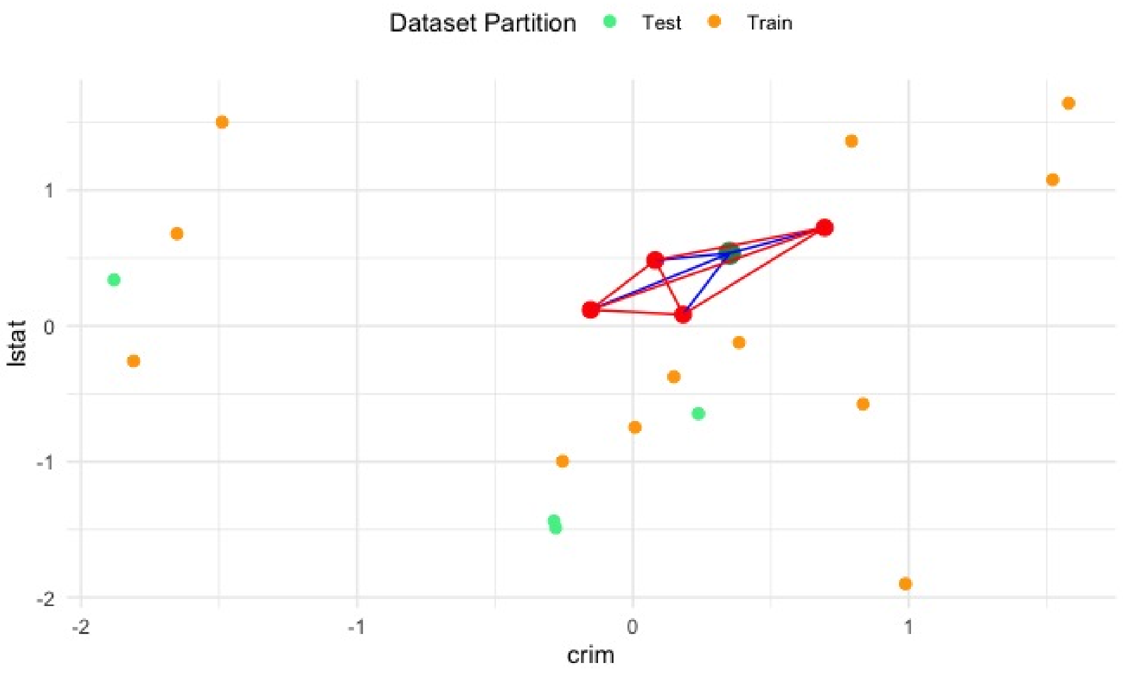


Figure 3.1: Standard Output of a Classification task

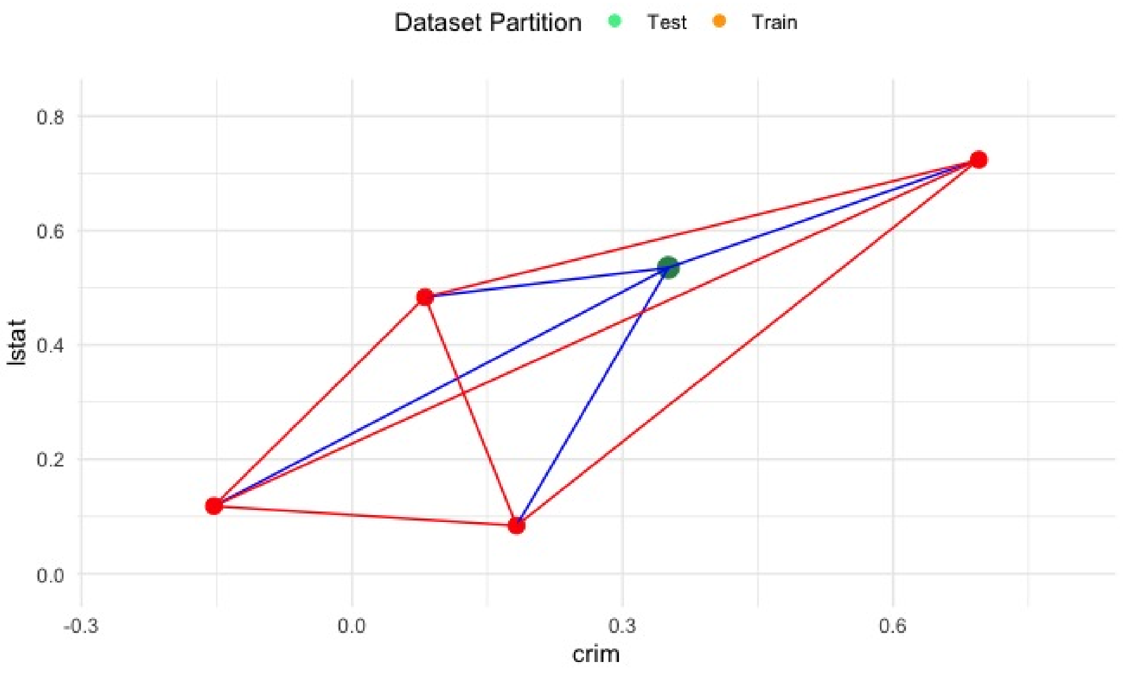


Figure 3.2: Standard Output of a Classification task

Chapter 4

Results

4.1 Implementation in R

Libraries used:

- *dplyr*: it allows the use of the pipe `%>%` operator and functions like *filter()*, *arrange()*, *top_n()*, and *slice_head()*.
- *proxy*: it contains the function dist() that allow to compute the distance matrices.
- *e1071*: the function *svm()* is part of the e1071 package.
- *randomForest*: the function *randomForest()* is part of the randomForest package.
- *mltools*: it has the function *mcc()* that computes the Matthew's Correlation Coefficient (MCC).

The other functions are part of the baseR.

Control parameters:

- *x* - a data frame of independent variables.
- *y* - a vector, dependent variable (for classification, it must be a factor object).
- *p* - percentage of data in the training set (default: 0.8).
- *min_size* - minimum number of points to consider as closest neighbours (default: 3).
- *max_size* - maximum number of point to consider as closest neighbours (default: 7).

- *max_dist* - maximum distance at which the points must be to be considered, when max_dist = 1, it is considered the average distance in the training distance matrix, calculated as $distance = mean(distance\_train\_matrix) \times max\_dist$.
- *seed* - the seed to have always the same results (default: 1).
- *model_type* (SVM — RF) – machine learning algorithm (default: "svm", options: "svm", "rf").
- *k* - the kernel in the SVM algorithm (default: "radial", options: "linear", "radial", "polynomial").
- *cost* - the cost in the SVM algorithm (default: 1).
- *ntrees* - the number of trees in the RF algorithm (default: 500).

4.1.1 Code Description

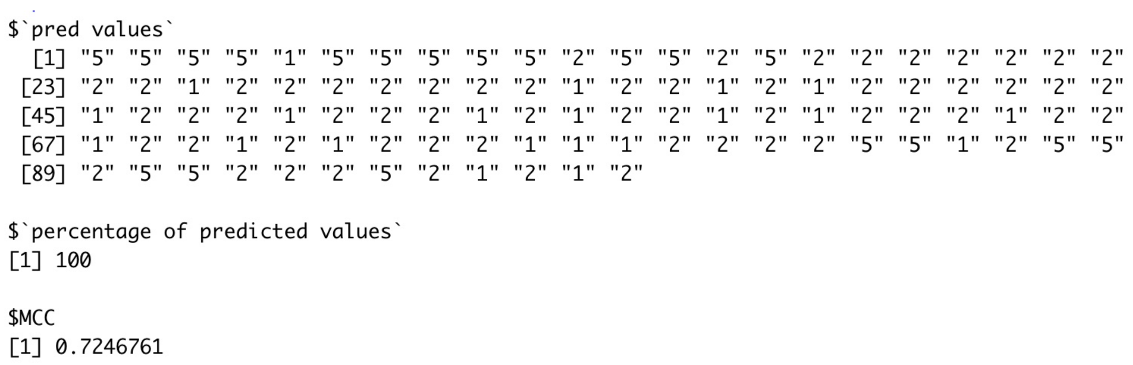
Two functions are developed, one for regression [7] and one for classification [8], they slightly differ in the code because for the classification the predicted values vector is treated as a factor. The function *suppressWarnings* is used because since there is one test observation each time, R will produce automatically a warning. Once the seed is set, the algotithm splits the data into train and test using a method that allows for data stratification. Then the training distance matrix (*train_dist_matrix*) and the test distance matrix (*test_dist_matrix*) are computed using the function *dist()* from the package proxy that computes the Euclidian distance, along with the average distance of the train distance matrix (mean_dist). Subsequently a variable - *pred_values* - is created. *pred_values* must be a factor for the classification algorithm (there is no need to specify anything for regression) that will be the vector of predicted values. The function *sapply* substitutes a for cycle, optimizing the cost complexity of the algorithm, for each column of the test distance matrix. The code is wrapped in *tryCatch* to handle exceptions. The algorithm creates a data frame (*distance_matrix*) with distances and the original index, then it calculates the number of elements to keep within the specified range, meaning the minimum between the maximum size and the number or rows of the distance matrix. If the number of elements to keep is less than the minimum size, it returns *NA*, otherwise, it filters the distance matrix (*filtered_df*): first selecting the *n* closest distances, then removing the distances greater than the maximum distance. Consequently, if *filtered_df* has less rows than the minimum size, it returns NA, otherwise it firsts check if there is just one class in the data frame, and if there is, return that class as a predicted value. If there are at least two classes it creates a test matrix and a train matrix using the closest distances stored in *filtered_df*. For the regression, there is no need to do the check for classes. Then it runs model that can be either SVM or RF, and then it predicts the values. The algorithm then returns a list containing three objects:

1. the predicted values.

2. the percentage of the test data used.

3. the MCC (Matthew's Correlation Coefficient) value for classification or the MSE (Mean Square Error) for regression.

A standard output (output for the dataset Covertype, parameters min_size = 3, max_size = 6, max_dist = 0.5) will look like this:



```
$`pred values`
  [1] "5" "5" "5" "5" "1" "5" "5" "5" "5" "5" "2" "5" "5" "2" "5" "2" "2" "2" "2" "2" "2" "2"
 [23] "2" "2" "1" "2" "2" "2" "2" "2" "2" "2" "1" "2" "2" "1" "2" "1" "2" "2" "2" "2" "2" "2"
 [45] "1" "2" "2" "2" "1" "2" "2" "2" "1" "2" "1" "2" "2" "1" "2" "1" "2" "2" "2" "1" "2" "2"
 [67] "1" "2" "2" "1" "2" "1" "2" "2" "2" "1" "1" "1" "2" "2" "2" "2" "5" "5" "1" "2" "5" "5"
 [89] "2" "5" "5" "2" "2" "2" "5" "2" "1" "2" "1" "2"

$`percentage of predicted values`
[1] 100

$MCC
[1] 0.7246761
```

Figure 4.1: Standard Output of a Classification task

As shown in Fig. 4.1, the predicted values are considered characters since it is a classification problem, and it shows the values of the predictions that in this case are classes from 1 to 7, for a regression task, instead of classes there are numbers. The second element is the percentage of the predicted values: not always the algorithm can predict for each observation in the test set, due to the tuning of the model parameters. Usually this percentage is good when it is high. The last element is the value of the MCC, for regression tasks the MSE is displayed instead.

Alongside the description, a pseudo code has been written to have a generalization of the algorithm.

- Documentation: the pseudo code serves as a form of documentation for the algorithm. It will help future developers understand the intended logic and functionality of code.

- Problem solving: it allows a better understanding of the code, breaking down complex problems into smaller, more manageable steps, making it easier to identify and correct issues in your logic.
- Cross-Platform Development: when designing software that needs to run on multiple platforms, pseudo code can be used to outline the common logic that can be adapted for specific platforms. It will be easier to reproduce the algorithm on other programs.

It allows to break down complex problems into smaller, more manageable steps, making it easier to identify and correct issues in your logic.

4.1.2 Pseudo-Code

MS-QSAR Regression

```
Function MSQSAR_reg:
    Input: matrix of independent variables, vector of dependent
     variable, percentage of training data, minimum size, maximum size,
     maximum distance, seed, model type, kernel, cost, ntrees
Set the seed
Split into train and test
Create train distance and test distance matrices using distances
Calculate mean distance of train distance matrix and call it mean_dist
for i in test distance matrix columns do
    function (i)
        Append in results using tryCatch
        Create a data frame for each column of the test
        distance matrix and the original indices
        Do the minimum between maximum size and number of rows
        of the data frame and call it min
        if the minimum is less then the minimum size then return NA
        end if
        Filter the dataset according to min and and a distance
        that is calculated as maximum distance * mean_dist
        if the number of rows of the filtered data frame
        is less than the minimum size then return NA
        end if
        Extract the indices from the data frame text
        Create a test matrix using the classes in the test set and the values
        extracted with the indices in the test distance matrix
        Create a train matrix using the classes in the train set and the
        values extracted with the indices in the train distance matrix
        Choose a model type (either SVM or RF) with the respective
        parameters (kernel, cost and ntrees)
        Predict the values based on the model
        Create a warning if an error appears
        return the predictions
```

MS-QSAR Regression (continued)

```
33:     end function
34:     Calculate the percentage of the predictions
35:     Calculate the MCC value
36:     return a list made of the predictions,
37:      the percentage of the predictions and the MCC value
38: end for
```

MS-QSAR Classification

```
1: Function MSQSAR_cl:
2:     Input: matrix of independent variables, vector of dependent
3:      variable, percentage of training data, minimum size, maximum size,
4:      maximum distance, seed, model type, kernel, cost, ntrees
5: Set the seed
6: Split into train and test
7: Create train distance and test distance matrices using distances
8: Calculate mean distance of train distance matrix and call it mean_dist
9: for i in test distance matrix columns do
10:     function (i)
11:         Append in results using tryCatch
12:         Create a data frame for each column of the test
13:         distance matrix and the original indices
14:         Do the minimum between maximum size and number of rows
15:         of the data frame and call it min
16:         if the minimum is less then the minimum size then return NA
17:         end if
18:         Filter the dataset according to min and and a distance
19:         that is calculated as maximum distance * mean_dist
20:         if the number of rows of the filtered data frame
21:         is less than the minimum size then return NA
22:         end if
23:         if the filtered data frame has only one unique class then return the class
24:         and assign it to a vector unique_class
```

MS-QSAR Classification (continued)

```
        else
            *Extract* the indices from the data frame text
            *Create* a test matrix using the classes in the test set and the values
            extracted with the indices in the test distance matrix
            *Create* a train matrix using the classes in the train set and the
            values extracted with the indices in the train distance matrix
            *Choose* a model type (either SVM or RF) with the respective
            parameters (kernel, cost and ntrees)
            *Predict* the values based on the model
        **end if**
        *Create* a warning if an error appears
        **return** the predictions
    **end function**
    *Calculate* the percentage of the predictions
    *Calculate* the MCC value
    **return** a list made of the predictions,
     the percentage of the predictions and the MCC value
**end for**
```

4.1.3 Metrics

The evaluation of the models' performance encompasses two primary dimensions: accuracy and efficiency. In the context of classification, we rely on Matthew's Correlation Coefficient (MCC) to gauge accuracy, while for regression tasks, Mean Square Error (MSE) is employed. Assessing efficiency involves considering training and prediction times, which provide straightforward measures. However, it is essential to delve into a more detailed examination of the accuracy dimension.

MCC MCC has been chosen over Accuracy as a measure of accuracy for different reasons:

1. Imbalanced Datasets: in imbalanced datasets, where one class is much more prevalent than the other, accuracy can be misleading. For example, if 95% of the samples belong to Class A and only 5% to Class B, a model that predicts all samples as Class A will still have an accuracy of 95%, even though it is not providing meaningful predictions. MCC takes into account both true positives and true negatives, providing a more balanced evaluation.

2. Unequal Costs: in some applications, the cost of making a false positive (Type I error) and the cost of making a false negative (Type II error) can be substantially different. MCC considers both types of errors, making it more suitable for scenarios where the cost of mistakes varies.

3. Handling Class Imbalance: MCC is better at assessing models' performance when there's a class imbalance, as it provides a more comprehensive view of the model's ability to correctly classify both classes.

4. Robustness: MCC is more robust when working with small sample sizes or noisy data because it considers all four components (true positives, true negatives, false positives, and false negatives) in its calculation.

5. Accounting for Chance: accuracy does not account for random or chance predictions. In contrast, MCC takes into account the possibility of random guessing, providing a more informative metric.

6. Multi-class: it can be extended for use in multi-class classification problems. When applied to multi-class cases, MCC provides a single metric that takes into account the entire confusion matrix, making it a valuable tool for assessing the quality of multi-class classification models. Formula 4.1 shows how MCC is adapted for

multi-class scenarios: in a multi-class problem, there are typically more than two classes, denoted as Class 1, Class 2, Class 3, and so on.

MCC is defined by Formula 4.1:

$$MCC = \frac{\sum_k^N TP_k \times TN_k - FP_k \times FN_k}{\sqrt{\sum_k^N (TP_k + FP_k) \times (TP_k + FN_k) \times (TN_k + FP_k) \times (TN_k + FN_k)}} \tag{4.1}$$

where:

- N is the number of classes.
- TP_k represents true positives for class k.
- TN_k represents true negatives for class k.
- FP_k represents false positives for class k.
- FN_k represents false negatives for class k.

For a binary classification task, the MCC is defined by Formula 4.1 can be simplified into MCC is defined by Formula 4.2.

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{4.2}$$

where TP is the number of true positives, TN is the number of true negatives, FP is the number of false positives and FN is the number of false negatives.

The MCC score ranges between -1 and +1, a value of:

- 1 represents perfect prediction;
- 0 means that the model is no better than random guessing;
- -1 indicating complete disagreement between predictions and actual outcomes.

MSE Mean Squared Error (MSE) is commonly used as a measure of accuracy in regression tasks for several reasons:

1. Quantitative Assessment: MSE provides a quantitative measure of the accuracy of a regression model. It calculates the average of the squared differences between the predicted values and the actual target values. A lower MSE indicates a better fit of the model to the data.

2. Sensitivity to Errors: squaring the errors in MSE gives more weight to larger errors, which can be important in regression tasks where large errors might be more detrimental. This sensitivity to errors helps in identifying and addressing significant discrepancies between predictions and actual values.

3. Differentiability: MSE is a differentiable function, which is essential for gradient-based optimization algorithms. Many machine learning algorithms, including linear regression and neural networks, use gradient descent to optimize model parameters. MSE's differentiability makes it suitable for these optimization techniques.

4. Statistical Properties: when the errors in a regression model are normally distributed and independent, minimizing MSE corresponds to maximum likelihood estimation. This statistical property makes MSE a suitable choice for linear regression when underlying assumptions are met.

5. Interpretability: MSE is expressed in the same units as the target variable, making it interpretable in the context of the problem. This allows you to quantify the average squared error in the original units, which can be more meaningful to stakeholders.

6. Consistency: under certain conditions, minimizing MSE provides consistent estimates of the model parameters, meaning that as the sample size increases, the estimates converge to the true values. This consistency property is essential for ensuring that the model parameters become increasingly accurate with more data.

MSE is defined by Formula 4.3:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{4.3}$$

where

- n is the number of data points.
- y_i denotes the actual target value for the i-th data point.

- $\hat{y}_i$ represents the predicted value for the i-th data point.

However, it is important to note that MSE also has some limitations:

1. Sensitivity to Outliers: MSE is highly sensitive to outliers, which means that a single extreme value can significantly impact the measure. In situations where outliers are common, other metrics like Mean Absolute Error (MAE) or Huber loss might be preferred.
2. Units of Measurement: The units of MSE are squared units of the target variable, which might not always be intuitive for interpretation.

To sum up, MSE is a widely used measure of accuracy in regression tasks due to its quantitative and differentiable nature, statistical properties, and interpretability. However, its suitability depends on the specific characteristics of the problem and the importance of sensitivity to outliers. It is often valuable to consider multiple metrics to gain a comprehensive understanding of model performance.

4.2 Results of the implementation with common benchmarks

4.2.1 Datasets

The algorithm was tested on a total of six datasets: 3 for classification and 3 for regression. The datasets used for classification were:

– Breast Cancer Dataset [1] Wolberg & Street (1995)

– Covertype Dataset[1] [2] Blackard (1998)

– Diabetes Dataset [3] Efron et al. (2004)

The datasets used for regression:

– Auto Dataset [4] [2] Quinlan (1993)

[1]only firsts 10 features and firsts 500 observations taken into account
[2]8th and 9th features not included

– Boston Housing Dataset [5] Harrison & Rubinfeld (1978)

– Forest Fires Dataset[3] [6] Cortez & Morais (2008)

The description of the datasets can be found in the appendix.

4.2.2 Model Prediction and Comparison with the State of Art

The algorithm was run on the datasets mentioned previously and it was compared with the state of art. The analysis of the MSQSAR algorithm was performed using the SVM as an argument. Firstly, the parameters were tuned to find the best ones to use, choosing them according to the highest MCC or MSE value. A grid search was used to choose the parameters since no method to select the parameters has been developed yet.

The parameters were initialized as:

- Minimum size: 3, 4, 5
- Maximum size: 6, 8, 10
- Maximum distance: 0.5, 1, 1.5

The tuning of the kernel and cost was not performed, because tuning is computationally expensive and for many tasks the best kernel is radial, whereas changing the cost does not affect much the accuracy of the results. The k-value for the KNN was chosen according to the minimum size of the MSQSAR algorithm, meaning that when the minimum size is 3, k is set equal to 3.

The best parameters to use in the algorithm for each dataset are stored in Table 4.1.

Table 4.1: Datasets and Parameters for Classification

Dataset	Parameters		
	Min Size	Max Size	Max Distance
Breast Cancer	3	8	1.5
Covertype	3	6	0.5
Diabetes	3	10	1

Although the algorithm is thought to work for dataset in which the values are in between 0 and 1, the data were not scaled. In the algorithm the distance for which it filters

[3] 3rd and 4th features not included

the data is the product of the average distance of the matrix and the "maximum distance" parameter. Since the MSQSAR is a model that can be thought as something in between KNN and SVM (since I used SVM as an argument of the MSQSAR algorithm), in the results the comparison among those three methods will have a higher focus. The methods chosen for comparison are: Logistic Regression Cox (1958), KNN, SVM and RF. The results and comparison are stored in Table 4.2.

Table 4.2: MCC Comparison of Methods for Classification

Dataset	Methods				
	Logistic Regression	MSQSAR	KNN	RF	SVM
Breast Cancer	0.850	0.870	0.873	0.925	0.907
Covertype	0.260	0.725	0.699	0.796	0.701
Diabetes	-0.512	0.367	0.325	0.544	0.397

1. Breast Cancer Dataset:
    - In the Breast Cancer dataset, all the methods perform very well, with Random Forest being the best model.
    - MSQSAR achieved a high MCC of 0.870, indicating its effectiveness in accurately classifying breast cancer cases.
    - Notably, MSQSAR has a similar measure as KNN, which had an MCC of 0.873, and came close to the performance of SVM with an MCC of 0.907. This suggests that MSQSAR is a competitive method for this dataset and is particularly effective in breast cancer classification.
2. Covertype Dataset:
    - When applied to the Covertype dataset, MSQSAR demonstrated strong classification performance with an MCC of 0.725. This result underscores the method's ability to accurately categorize forest cover types.
    - MSQSAR's performance outperformed both KNN and SVM, and approached the level of RF with an MCC of 0.796. This indicates that MSQSAR is a valuable choice for classifying forest cover types in this dataset.
3. Diabetes Dataset:
    - In the context of the Diabetes dataset, MSQSAR showcased a significant improvement in classification accuracy with an MCC of 0.367. This demonstrates the method's effectiveness in accurately predicting diabetes outcomes.

- MSQSAR's performance was notably better than KNN, which had an MCC of 0.325, and it approached the performance level of SVM with an MCC of 0.397, but the best method is RF with a value of 0.544. This highlights the ability of MSQSAR to enhance the accuracy of diabetes prediction in this dataset.

In summary, the results show that MSQSAR is a powerful classification method that performs well across different datasets. It consistently outperforms or closely matches the performance of other methods like KNN and SVM, making it a promising choice for various classification tasks.

Consequently, a regression analysis on three datasets for regression was carried out, Table 4.3 shows the parameters used for the MSQSAR algorithm.

Table 4.3: Datasets and Parameters for Regression

Dataset	Parameters		
	Min Size	Max Size	Max Distance
Auto	3	8	0.5
Boston Housing	3	6	0.5
Forest Fires	3	6	0.5

The result and comparison are stored in Table 4.4.

Table 4.4: MSE Comparison of Methods for Regression

Dataset	Methods				
	Logistic Regression	MSQSAR	KNN	RF	SVM
Auto	9.603	19.078	13.890	5.200	4.323
Boston Housing	17.014	28.206	23.349	11.330	12.280
Forest Fires	1495.859	1883.524	1866.376	1876.375	1606.62

1. Auto Dataset:
    - In the Auto dataset MSQSAR achieved an MSE of 19.078. While this value indicates a certain degree of error in the predictions, it is important to consider it in the context of the dataset's characteristics and the other methods.
    - Support Vector Machine outperformed all other methods with the lowest MSE of 4.323. MSQSAR is the worst performer in this dataset. Moreover, KNN does not perform well compared to the other methods.

2. Boston Housing Dataset:

    - MSQSAR's performance was the worst among those methods also in this case. with an MSE of 28.206, while RF and SVM being the best models with MSE values of 11.330 and 12.280.
    - Also here KNN performs slighly better than MSQSAR but worse than the other techniques.

3. Forest Fires Dataset:

    - MSQSAR's MSE is similar to that of KNN and RF, while the best method is the Logistic Regression.
    - The dataset's nature and complexity may contribute to the higher MSE values across all methods. MSQSAR remains a viable choice for regression tasks in this context.

The MSE values provide insights into the performance of different regression methods across various datasets. MSQSAR, while not always the best performer, consistently delivers competitive results and can be a valuable choice in regression applications. Compared to the classification tasks, in regression the MSQSAR performs worse, this may be due to the choice of the datasets used. As shown in the regression task, MSQSAR performs good when KNN has good results. It must be also taken into account that, before performing the MSQSAR using the SVM, the parameters for the SVM were not tuned, not knowning if actually the choice of the radial kernel and the cost equal to one are good or bad choices. Tuning the parameters will probably lead to better results for the MSQSAR, but tuning leads to a higher time complexity.

4.2.3 System Time

The complexity of an algorithm plays a fundamental role on its use and future development. It was evaluated the time the MSQSAR algorithm takes using the covertype dataset (Fig. 4.2), performing the analysis with 500, 1000 and 1500 observations, then being compared with the time the SVM takes (Fig. 4.3).

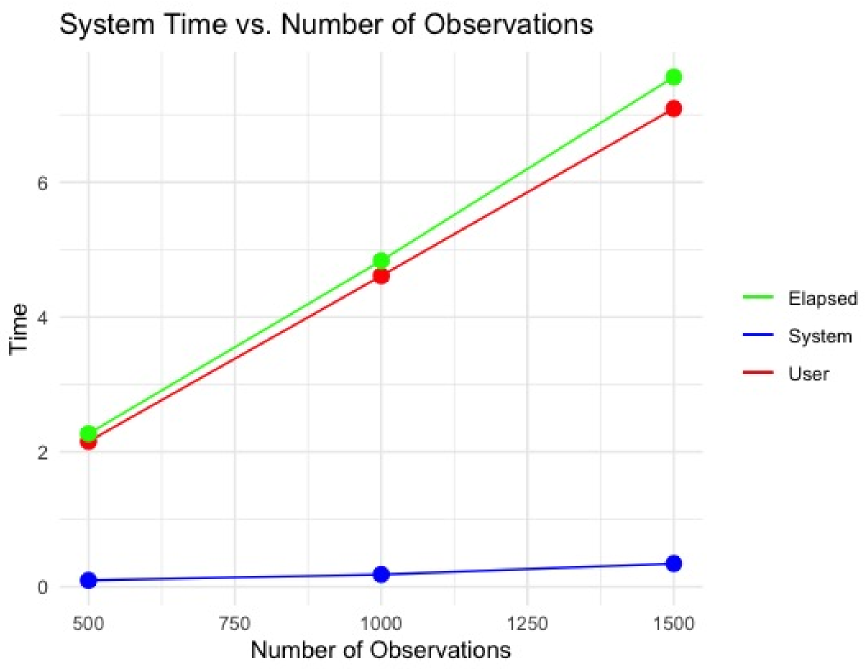


Figure 4.2: System Time of MSQSAR for the dataset Covertype

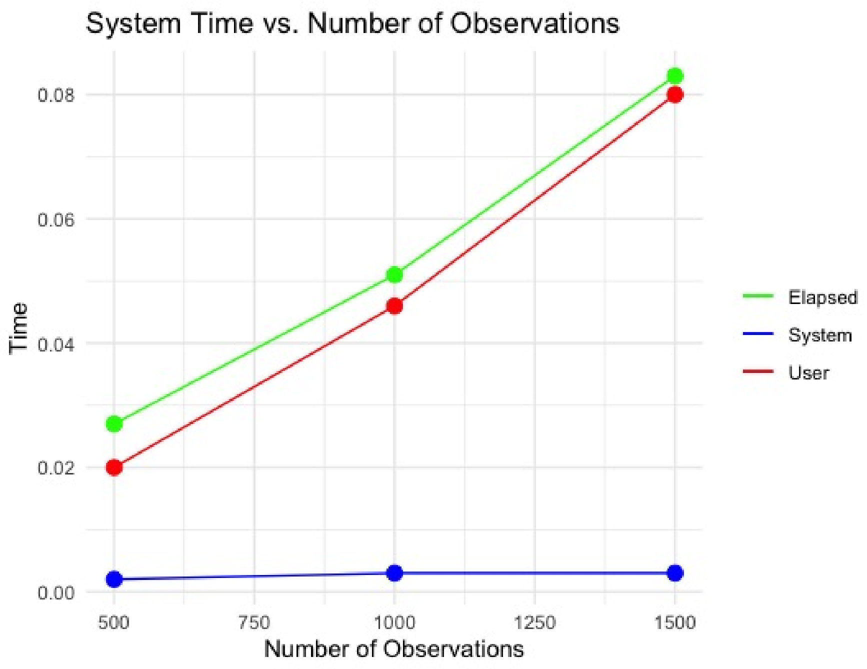


Figure 4.3: System Time of SVM for the dataset Covertype

As displayed in Fig. 4.2, the more observations, the longer the algorithm will take. For 1000 observations it will take almost five seconds, that is incredibly slower than the SVM (Fig. 4.3). The algorithm must be improved under this aspect because it is too slow to be used with bigger datasets. MSQSAR is slow because the algorithm stores a training distance matrix $n\ x\ n$, and a test matrix $n\ x\ p$ where n is the number of observations and p is the number of features. Hence, the algorithm is very dependent on the number of observations. To avoid the problem of the storage, the algorithm can be updated using the "minimum hashing" algorithm.

4.3 Minimum Hashing Algorithm

The minimum hashing algorithm (MinHash) is a technique primarily used for estimating the similarity between sets, especially when dealing with large datasets.

MinHash is a probabilistic algorithm that allows for the efficient approximation of the Jaccard similarity coefficient between two sets. The Jaccard similarity coefficient is a measure of the proportion of shared elements between two sets. Given the computational intensity of directly computing the Jaccard similarity for large sets, MinHash offers a more scalable solution.

The MinHash works in the following way:

1. Hashing: Each element in a set is passed through multiple hash functions, producing a hash value for each element.
2. Minimum Hash Value: For each hash function, the minimum hash value is selected, resulting in a "signature" for the set.
3. Comparison: The signatures of two sets are compared to estimate their Jaccard similarity. The proportion of hash functions for which the two sets have the same minimum hash value gives an estimate of their similarity.

Among its application we can name the document similarity: clustering similar articles and useful for plagiarism detection; recommendation systems: by identifying similar items through probabilities; bioinformatics: looking for similar genetic patterns. Although it is used mainly to find similar items, and it can be very useful in the MS-QSAR algorithm since we look for the smallest distances, the MinHash has another great advantage: the storage efficiency. Instead of storing entire sets, thus in our case the train and test matrices, that can be hard to compute when dealing with many observations, one can store their compact MinHash signatures, saving space. The limitation of the MinHash is providing an approximation of the Jaccard similarity and not the exact value.

In conclusion, MinHash is a milestone technique in the domain of data similarity, offering a balance between accuracy and computational efficiency. Its ability to approximate set similarities in large datasets makes it very useful when treating big datasets.

Chapter 5

Discussion and Limitations

The MSQSAR version for R developed has good results in terms of accuracy of the model, and, considering the dataset used, it works better for classification than for regression, but this is possibly due to the choice of the datasets. The biggest issue is the time: further work must be done in order to make the algorithm faster.

There are few improvements that can be made:

– The distance used in the algorithm to calculate the training distance matrix and the test distance matrix is the Euclidian distance. Other types of distance exist and may be more effective than the Euclidian, for instance the Jaccard distance or the Mahalanobis distance can be included in the algorithm as an argument to have the possibility to check if, calculating the distance in another way, the results will change.

– A formula to find the best parameters - minimum size, maximum size and maximum distance - has not yet been developed. To find the best parameters I used a grid search, but this is not the most effective way to find them. Other iterative methods can be used, although I believe a formula can be developed to choose the number of elements we must consider.

– More metrics can be added to check for accuracy. Having multiple metrics for model evaluation is essential for obtaining a comprehensive and balanced assessment of model performance. It allows to consider various aspects of performance, trade-offs, and domain-specific requirements, making the evaluation more robust and informative.

In the future, working further on the algorithm, the main idea is to create an R package where the algorithm can be stored. The structure of the package will be similar to the other packages for machine learning in R, like “randomForest” (for RF), or “e1017” (for SVM). Furthermore, other techniques can be developed within the algorithm, as the variable importance (used also in randomForest package) or graphs (when possible, using the ggplot2 package).

Chapter 6

Conclusion

The MSQSAR algorithm can be seen as a middle ground between the KNN and the SVM. The KNN analyses the local effects, while the SVM analyses the global effect. This algorithm is thought to have a similar performance to KNN for the datasets where the KNN has good results, thus where the local effects have more weight, and to be like the SVM for datasets where the SVM performs better, thus where the global effects are more significant. The R version of the algorithm can be found in the Appendix [7], [8], and on my github, alongside the analysis: https://github.com/EmanueleTartaglione . In conclusion, the MSQSAR algorithm, initially thought to work on datasets that had values between 0 and 1, can be used with every kind of datasets. The results, in term of accuracy, are not far away from the results used with the known methods, and further work should be done to improve this version: a way to select the parameters should be found, different types of distances should be tested, the speed of the algorithm must decrease, and the algorithm must be tested on more datasets. It can be also developed a unique algorithm, without the need to divide it for regression and classification problems.

Translating the MSQSAR algorithm from Python to R was a big challenge. I encountered difficulties mostly because it was not a direct translation since I had to develop an algorithm that could work with every kind of datasets. During this work, I learned a lot about both Python and R because when translating the algorithm, I had to understand the logic behind the functions used. For instance, one big challenge was to avoid the "for" cycles in R, in favor of the apply family and the pipelines that are computationally much faster than the cycles. To sum up, I enjoyed working on this project that Professor Falcão assigned me because I have seen it as a perfect experience to conclude my Master's Degree in Statistical Sciences. I thank him for the support and help he had given me through this process and I hope our paths will cross again in the future.

Chapter 7

Appendix

Datasets Description:

[1] Breast Cancer Wisconsin (Diagnostic) Dataset:

Features are computed from a digitized image of a fine needle aspirate (FNA) of a breast mass. They describe characteristics of the cell nuclei present in the image. A few of the images can be found at http://www.cs.wisc.edu/street/images/ . Separating plane described above was obtained using Multisurface Method-Tree (MSM-T) [K. P. Bennett, "Decision Tree Construction Via Linear Programming." Proceedings of the 4th Midwest Artificial Intelligence and Cognitive Science Society, pp. 97-101, 1992], a classification method which uses linear programming to construct a decision tree. Relevant features were selected using an exhaustive search in the space of 1-4 features and 1-3 separating planes. The actual linear program used to obtain the separating plane in the 3-dimensional space is that described in: [K. P. Bennett and O. L. Mangasarian: "Robust Linear Programming Discrimination of Two Linearly Inseparable Sets", Optimization Methods and Software 1, 1992, 23-34].

1. ID number
2. Diagnosis (M = malignant, B = benign)
3. Ten real-valued features are computed for each cell nucleus:
    - Radius (mean of distances from center to points on the perimeter)
    - Texture (standard deviation of gray-scale values)
    - Perimeter
    - Area

- Smoothness (local variation in radius lengths)
- Compactness $\frac{\text{perimeter}^2}{\text{area}} - 1.00$
- Concavity (severity of concave portions of the contour)
- Concave points (number of concave portions of the contour)
- Symmetry
- Fractal dimension (”coastline approximation” - 1)

[2] Covertype Dataset

The samples in this dataset correspond to 30×30m patches of forest in the US, collected for the task of predicting each patch’s cover type, i.e. the dominant species of tree. There are seven covertypes, making this a multiclass classification problem. Each sample has 54 features, described in the website https://archive.ics.uci.edu/dataset/31/covertype. Some of the features are boolean indicators, while others are discrete or continuous measurements.

[3] Diabetes Dataset

Information about dataset attributes:

1. Pregnancies: To express the number of pregnancies
2. Glucose: To express the glucose level in blood
3. BloodPressure: To express the blood pressure measurement
4. SkinThickness: To express the thickness of the skin
5. Insulin: To express the insulin level in blood
6. BMI: To express the body mass index
7. DiabetesPedigreeFunction: To express the diabetes percentage
8. Age: To express the age
9. Outcome: To express the final result (1 is ”Yes” and 0 is ”No”)

[4] Auto Dataset

Auto: This dataset is a slightly modified version of the dataset provided in the StatLib library. In line with the use by Ross Quinlan (1993) in predicting the attribute ”mpg”, 8 of the original instances were removed because they had unknown values for the ”mpg” attribute. ”The data concerns city-cycle fuel consumption in miles per gallon, to be predicted in terms of 3 multivalued discrete and 5 continuous attributes.” (Quinlan, 1993) I used mpg as independent variable

1. mpg: Continuous
2. cylinders: Multi-valued discrete
3. displacement: Continuous
4. horsepower: Continuous
5. weight: Continuous
6. acceleration: Continuous
7. model year: Multi-valued discrete
8. origin: Multi-valued discrete
9. car name: String (unique for each instance)

[5] The Boston Housing Dataset

The Boston Housing Dataset is a derived from information collected by the U.S. Census Service concerning housing in the area of Boston, MA. The following describes the dataset columns:

1. CRIM - Per capita crime rate by town
2. ZN - Proportion of residential land zoned for lots over 25,000 sq.ft.
3. INDUS - Proportion of non-retail business acres per town.
4. CHAS - Charles River dummy variable (1 if tract bounds river; 0 otherwise)
5. NOX - Nitric oxides concentration (parts per 10 million)
6. RM - Average number of rooms per dwelling

7. AGE - Proportion of owner-occupied units built prior to 1940
8. DIS - Weighted distances to five Boston employment centers
9. RAD - Index of accessibility to radial highways
10. TAX - Full-value property-tax rate per $\$10000$
11. PTRATIO - Pupil-teacher ratio by town
12. $B - 1000(B_k - 0.63)^2$ where Bk is the proportion of blacks by town
13. LSTAT - $\%$ Lower status of the population
14. MEDV - Median value of owner-occupied homes in $\$1000$'s

[6] Forest Fires Dataset

For more information, read [Cortez and Morais, 2007].

1. X - x-axis spatial coordinate within the Montesinho park map: 1 to 9
2. Y - y-axis spatial coordinate within the Montesinho park map: 2 to 9
3. month - month of the year: 'jan' to 'dec'
4. day - day of the week: 'mon' to 'sun'
5. FFMC - FFMC index from the FWI system: 18.7 to 96.20
6. DMC - DMC index from the FWI system: 1.1 to 291.3
7. DC - DC index from the FWI system: 7.9 to 860.6
8. ISI - ISI index from the FWI system: 0.0 to 56.10
9. temp - temperature in Celsius degrees: 2.2 to 33.30
10. RH - relative humidity in
11. wind - wind speed in km/h: 0.40 to 9.40
12. rain - outside rain in mm/m²: 0.0 to 6.4
13. area - the burned area of the forest (in ha): 0.00 to 1090.84

[7]

```
msqsar_reg <- function(x, y, p = 0.8,
                        min_size = 3, max_size = 7, max_dist
                            = 4, seed = 1,
                        model_type = "svm", k = "radial",
                           cost = 1, ntrees = 500) {
  suppressWarnings({
    set.seed(seed)

    # Split x and y into training and test
    index <- sort(sample(nrow(x), nrow(x) * 0.8, replace =
       FALSE), decreasing = F)
    x.train <- x[index,]
    y.train <- y[index]
    x.test <- x[-index,]
    y.test <- y[-index]

    # Create training matrix and test matrix
    train_dist_matrix <- as.matrix(dist(x.train))
    test_dist_matrix <- as.data.frame.matrix(dist(x.train,
       x.test))

    # Calculate the mean distance
    mean_dist <- mean(train_dist_matrix)

    # Calculate the predictions
    pred_values <- sapply(1:ncol(test_dist_matrix),
       function(i) {
      # Wrapping the code in tryCatch to handle exceptions
      result <- tryCatch({
        # Creating a data frame with distances and original
            indices
        distance_matrix <- data.frame(distances = test_dist
           _matrix[, i],
                                      original_index =
                                         rownames(test_dist
                                         _matrix))
```

```r

        # Calculating the number of elements to keep within
            the specified range
        num_elements_to_keep <- min(max_size, nrow(distance
           _matrix))

        # If the number of elements to keep is less than
           the minimum size, return NA
        if (num_elements_to_keep < min_size) {
          return(NA)
        }

        # Keep the smallest elements within the range
        # and filter the selected observations based on max
           _dist
        filtered_df <- distance_matrix %>%
          top_n(-num_elements_to_keep, wt = distances) %>%
          arrange(distances) %>%
          slice_head(n = num_elements_to_keep) %>%
          filter(distances <= max_dist*mean_dist)

        # If after filtering by max_dist,
        # the number of observations is less than min_size,
            return NA
        if (nrow(filtered_df) < min_size) {
          return(NA)
        }

        # Get the indices to keep
        indices <- as.integer(filtered_df$original_index)

        # Create the test matrix
        test_matrix <- data.frame(
          y.test[i],
          data.frame(
            t(test_dist_matrix[rownames(test_dist_matrix) %
               in% indices, ]))[i,])

```

```r
      # Create the train matrix
      train_matrix <- data.frame(
        y[indices],
        train_dist_matrix[rownames(train_dist_matrix) %in% indices,
                          colnames(train_dist_matrix) %in% indices])
      colnames(train_matrix)[1] <- "y.train"

      # Choose model type based on model_type argument
      if (model_type == "svm") {
        model <- svm(y.train ~ ., train_matrix,
                     kernel = k, cost = cost, type = "eps-regression")
      } else if (model_type == "rf") {
        model <- randomForest(y.train ~ ., train_matrix, ntree = ntrees)
      } else {
        stop("Invalid model_type. Please choose 'svm' or 'rf'.")
      }

      return(predict(model, newdata = test_matrix))

    }, error = function(e) {
      # Handle errors and return NA
      warning("Error occurred for column ", i, ": ", conditionMessage(e))
      return(NA)
    })
  })

  # Calculate the percentage of predictions done
  perc_pred_values <- length(
    pred_values[!is.na(pred_values)]) / length(pred_values) * 100

  # Calculate the Mean Square Error
```

```
    mse <- mean((pred_values - y.test)^2, na.rm = T)

    # Return the predictions, the percentage and the MSE
    return(list("pred values" = round(pred_values,3),
                "percentage of predicted values" = perc_
                    pred_values,
                "MSE" = mse))
  })
}
```

[8]

```
msqsar_cl <- function(x, y, p = 0.8,
                      min_size = 3, max_size = 7, max_dist
                        = 1, seed = 1,
                      model_type = "svm", k = "radial",
                        cost = 1, ntrees = 500) {
    set.seed(seed)

    # Split x and y into training and test
    index <- sort(sample(nrow(x), nrow(x) * p, replace =
       FALSE), decreasing = F)
    x.train <- x[index,]
    y.train <- y[index]
    x.test <- x[-index,]
    y.test <- y[-index]

    # Create training matrix and test matrix
    train_dist_matrix <- as.matrix(dist(x.train))
    test_dist_matrix <- as.data.frame.matrix(dist(x.train,
       x.test))

    # Calculate the mean distance
    mean_dist <- mean(train_dist_matrix)

    # Calculate the predictions
    pred_values <- sapply(1:ncol(test_dist_matrix),
       function(i) {
```

```r

    # Wrapping the code in tryCatch to handle exceptions
    result <- tryCatch({

      # Creating a data frame with distances and original indices
      distance_matrix <- data.frame(distances = test_dist_matrix[, i],
                                 original_index = rownames(test_dist_matrix))

      # Calculating the number of elements to keep within the specified range
      num_elements_to_keep <- min(max_size, nrow(distance_matrix))

      # If the number of elements to keep is less than the minimum size, return NA
      if (num_elements_to_keep < min_size) {
        return(NA_character_)
      }

      # Keep the smallest elements within the range
      # and filter the selected observations based on max_dist
      filtered_df <- distance_matrix %>%
        top_n(-num_elements_to_keep, wt = distances) %>%
        arrange(distances) %>%
        slice_head(n = num_elements_to_keep) %>%
        filter(distances <= max_dist*mean_dist)

      # If after filtering by max_dist,
      # the number of observations is less than min_size, return NA
      if (nrow(filtered_df) < min_size) {
        return(NA_character_)
      }

    # If filtered_df has only one unique class, return
       that species
    unique_class <- unique(y[as.integer(filtered_df$
       original_index)])
    if (length(unique_class) == 1) {
      return(as.character(unique_class))
    } else {

      # For cases where there are fewer classes than
         original but more than one,
      # proceed to build the model with the available
         classes

      # Get the indices to keep
      indices <- as.integer(filtered_df$original_index)

      # Create the test matrix
      test_matrix <- data.frame(
        y.test[i],
        data.frame(
          t(test_dist_matrix[rownames(test_dist_matrix)
              %in% indices, ]))[i,])

      # Create the train matrix
      train_matrix <- data.frame(
        y[indices],
        train_dist_matrix[rownames(train_dist_matrix) %
           in% indices,
                          colnames(train_dist_matrix) %
                             in% indices])
      colnames(train_matrix)[1] <- "y.train"

      # Choose model type based on model_type argument
      if (model_type == "svm") {
        model <- svm(y.train ~ ., train_matrix,
                     kernel = k, cost = cost, type = "C
                        -classification")
      } else if (model_type == "rf") {
```

```r
      train_matrix$y.train <- factor(train_matrix$y.train)
      model <- randomForest(train_matrix$y.train ~ .,
                 train_matrix, ntree = ntrees)
     } else {
      stop("Invalid model_type. Please choose 'svm' or 'rf'.")
     }

     return(as.character(predict(model, newdata = test_matrix)))
    }

   }, error = function(e) {
    # Handle errors and return NA
    warning("Error occurred for column ", i, ": ", conditionMessage(e))
    return(NA_character_)
   })
  })

  # Calculate the percentage of predictions done
  perc_pred_values <- length(
   pred_values[!is.na(pred_values)]) / length(pred_values) * 100

  # Calculate Matthew's Correlation Coefficient
  MCC <- mcc(pred_values, as.character(y.test))

  # Return the predictions, the percentage and the MCC
  return(list("pred values" = pred_values,
        "percentage of predicted values" = perc_pred_values,
        "MCC" = MCC))
}
```

Bibliography

Altman N. S., 1992, *An introduction to kernel and nearest-neighbor nonparametric regression*, The American Statistician, 46, 175

Blackard J., 1998, *Covertype*, UCI Machine Learning Repository

Christiani T., Pagh R., 2016, *Set Similarity Search Beyond MinHash*, CoRR, abs/1612.07710

Cortes C., Vapnik V., 1995, *Support-vector networks*, Machine learning, 20, 273

Cortez P., Morais A., 2008, *Forest Fires*, UCI Machine Learning Repository

Cox D. R., 1958, *The regression analysis of binary sequences*, Journal of the Royal Statistical Society: Series B (Methodological), 20, 215

Efron B., Hastie T., Johnstone I., Tibshirani R., 2004, *Least Angle Regression*, Annals of Statistics (with discussion), pp 407–499

Gareth James Daniela Witten T. H. R. T., 2013, *An introduction to statistical learning : with applications in R*. Springer

Han J., Kamper M., Pei J., 2012, *Data Mining Concepts and Techniques*. Morgan Kaufmann, Waltham, MA, United States of America

Harrison D., Rubinfeld D., 1978, *Hedonic prices and the demand for clean air*, J. Environ. Economics & Management, 5, 81

Hastie T., Tibshirani R., Friedman J., 2017, *The Elements of Statistical Learning - Data Mining, Inference, and Prediction*. Springer

Ho T. K., 1995, in Proceedings of 3rd international conference on document analysis and recognition. pp 278–282

Murphy K. P., 2012, *Machine Learning - A Probabilistic Perspective*. The MIT Press, Cambridge, MA, United States of America

Quinlan R., 1993, *Auto MPG*, UCI Machine Learning Repository

Wolberg William M. O. S. N., Street W., 1995, *Breast Cancer Wisconsin (Diagnostic)*, UCI Machine Learning Repository