Universitat Politècnica de Catalunya

Data Mining

Bachelor Degree in Computer Science

Study over diabetic inpatients readmission rates

Authors:
Marc Badia Romero, Aleix Balletbó Gregorio,
Bernat Gené Skrabec, Víctor Giménez Ábalos,
Guillem Ferrer Nicolás, Daniel Tarrés Amselem

Professor:
Mario Martin

Q1 Course 2018/2019

Contents

1 Data description

Our dataset comes from Kaggle, and it is named:

Kaggle's Diabetes 130 US hospitals for years 1999-2008

(https://www.kaggle.com/brandao/diabetes/home)

It is a dataset over the admission of 'diabetic' inpatients, with variables over demographics, diagnoses, administered drugs, procedures, insurance, weight, medical measurements, speciality etc. Specifically, we will focus on the readmission variable, which has values: "No", "<30" and ">30" depending on whether, after this encounter, he was readmitted into a hospital and when.

This variable is of great interest, since hospital readmissions, specifically in the case of diabetic inpatients, could be caused by dangerous treatment choices during its prior admission. Since we know most of the other data before, with this we can extrapolate whether the treatment will be effective during his first admission, thus avoiding readmission.

Our original dataset is constituted by 101766 individual encounters, which are characterised by:

- It is a hospital admission
- Of diabetic inpatients (meaning any kind of diabetes was entered to the system as a diagnosis)
- Whose stay lasted between 1 and 14 days
- With laboratory tests performed during the encounter
- With drugs administered during the encounter

Each data row contains information relating to that encounter. There are a total of 54 variables:

encounter_id, patient_nbr, race,gender, age, weight, admission_type_id, discharge_disposition_id, admission_source_id, time_in_hospital, payer_code, medical_specialty, num_lab_procedures, num_procedures, num_medications, number_outpatient, number_emergency, number_inpatient, diag_1, diag_2, diag_3, number_diagnoses, max_glu_serum, A1Cresult, metformin, repaglinide, nateglinide, chlorpropamide, glimepiride, acetohexamide, glipizide, glyburide, tolbutamide, pioglitazone, rosiglitazone, acarbose, miglitol, troglitazone, tolazamide, examide, citoglipton, insulin, glyburide-metformin, glipizide-metformin, glimepiride-pioglitazone, metformin-rosiglitazone, metformin-pioglitazone, change, diabetesMed, readmitted.

Most of these are self-explanatory. To clarify the least clear ones, number_outpatient, number_inpatient, number_emergency reference the number of visits of that kind (with no admission, with admission and in the emergency ward) of the same person previously during the year. Change is a variable that compiles whether a change in diabetic medications was done (either administering new drugs, stopping a treatment or changing the dose). Also, diagnoses are a alphanumeric code, with 1000 modalities.

The following is a snippet of the head of the dataset, with some medicines removed:

```
encounter_id 2278392 149190 64410 500364 16680
```

```
patient_nbr 8222157 55629189 86047875 82442376 42519267
race Caucasian Caucasian AfricanAmerican Caucasian Caucasian
gender Female Female Female Male Male
age [0-10) [10-20) [20-30) [30-40) [40-50)
weight NaN NaN NaN NaN NaN
admission_type_id 6 1 1 1 1
discharge_disposition_id 25 1 1 1 1
admission_source_id 1 7 7 7 7
time_in_hospital 1 3 2 2 1
payer_code NaN NaN NaN NaN NaN
medical_specialty Pediatrics-Endocrinology NaN NaN NaN NaN
num_lab_procedures 41 59 11 44 51
num_procedures 0 0 5 1 0
num_medications 1 18 13 16 8
number_outpatient 0 0 2 0 0
number_emergency 0 0 0 0 0
number_inpatient 0 0 1 0 0
diag_1 250.83 276 648 8 197
diag_2 NaN 250.01 250 250.43 157
diag_3 NaN 255 V27 403 250
number_diagnoses 1 9 6 7 5
max_glu_serum None None None None None
A1Cresult None None None None None
metformin No No No No No
repaglinide No No No No No
nateglinide No No No No No
chlorpropamide No No No No No
glimepiride No No No No No
acetohexamide No No No No No
...
change No Ch No Ch Ch
diabetesMed No Yes Yes Yes Yes
readmitted NO >30 NO NO NO
```

Amongst all of these variables, we originally had 7 with missing values, ranging from less than 1% to some useless variables with almost all NAs. Specifically, we are quite surprised at the missing values of the weight variable, which had 96.86% missing values.

2 Preprocessing procedure and output data

The preprocessing performed to this dataset was extensive and quite complicated, since the dataset itself is not a trivial one. We will give the final preprocessing applied to the data, which was improved over several iterations, in which we measured the performance of a simple Random Forest with some modifications to see the impact of the preprocessing. These tests were done on a subset of the training data so as not to skew the final test result. We later on moved the splitting of the data to the end of the document for simplicity.

Given the complexity of the whole process, we will focus more on the modelling decisions rather than code snippets, which are available in the 'Preprocessing' notebook.

Decisions taken, ordered and with justification:

- **Decision 1: We will classify between earlier readmissions and the rest:** This is a decision regarding the problem itself. We observed in a previous study that >30 and NO classes behave almost the same, since most of the >30 encounters are not as likely to be a 'reincidence' rather than a 'readmission' due to other problems.
- **Decision 2: We will drop variables with a massive number of NAs, but we will conserve a weight variable which indicates whether weight was measured or not, as it might be interesting for study:** This also comes from the previous study, that patients with weight recorded behave slightly different than those that don't have it. This was also observed upon iterations of preprocessing.
- **Decision 3: Drop citoglipton and examide:** They were constant over the whole dataset, therefore useless.
- **Decision 4: Drop unknown-gender records:** There are 3 cases out of 100K, so we can actually drop them without affecting our results.
- **Decision 5: Reduction of admission_type_id, discharge_disposition_id, admission_source_id:** This was the only thing we could do in order to avoid a massive curse of dimensionality later on. The collapse follows an external study, that points to the fact that most of these ids are quite similar in meaning. For example, admission types of 1, 2 and 7 all correspond to urgent care, emergency and trauma, which make more sense together. Also, discharge_disposition_id = 11 is death during hospitalisation. In this case, prediction is useless, since we know beforehand.
- **Decision 6: One-hot-encode A1Cresult and max_glu_serum into Normal and Abnormal:** We encode the results into normal and abnormal results for the test instead of having 4 levels. This also works for reducing dimensionality, but also to avoid bias induced by using numerical values for them. If the test was not performed, both variables are left at value 0. Changing this to 4 new variables (one for each level) decreased the performance on several preprocessing iterations.
- **Decision 7: Age to numerical:** It is the best way to encode an ordered categorical variable in this case, since we could always impute the median. Since we will normalise later, instead we use values 1 to 10, which is equivalent.
- **Decision 8: One-hot encoding of gender:** Since we only have 2, we can encode it in a single 'Female?' binary variable.
- **Decision 9: Medications will be re-mapped to 0 if he does not take it, 1 otherwise (independently of dosage change):** With the amount of medications in the dataset and the distribution of levels (most of them are quite constant at steady or no), this induces the least bias. It is also confirmed by the preprocessing iterations.
- **Decision 10: We collapse diagnosis 1 into a higher granularity and one-hot encode it:** Having over 1000 levels, this was the only sensible choice. Label encoding

proved detrimental to the ML methods, both pre and post collapse, which makes a lot of sense. We also dropped diagnosis 2 and 3, since they worsen the results (no matter whether we create new vars or apply an or with the ones from diagnosis 1).

- **Decision 11: One hot encode change and diabetesMed:** for the sake of having an all-numeric dataset. These transformations are trivial.
- **Decision 12: One hot encode Ids:** We one-hot encode them after their collapse (@ decision 5).
- **Decision 13: Drop all NAs from race and one-hot-encode it:** This is a rather unorthodox decision with the amount of missings we have (2000), but with the amount of non-missings, we will not try to impute them, and instead will drop them, since the final dataset size is permissible and there are not many variables in the dataset to impute it from (not relevantly).
- **Decision 14: Applying logarithmic transformations to these variables POST SPLIT:** It is important to know that for detecting whether the variables required a log-transform we did not look at the test data, however we applied it in both if our training set was indeed in need of it. In the end, this affected 3 variables: outpatiency, emergency and inpatiency.
- **Decision 15: Standardisation of numerical variables:** Since all methods can work with a set of normalised variables, but not all of them work well with non-normalised, we decided to normalise all of the numerical variables (not counting binary ones). To do so, we also avoided looking at the test set, but we applied it in it too after fitting it with the train data. (Note: when we speak of standardisation we mean (x - u) / s; where u is the training mean of the variable and s its standard deviation, applied to each value x).

After doing these steps, we outputted the two datasets, the training and testing ones. Some of the techniques may require further individual preprocessing, however the common part is done.

3 Data splitting into train and test sets

Having a fairly straightforward dataset, which is not really time-dependant nor ordered, we decided to use a simple train-test split, with 30% going to the test (validation) split.

We perform this split exactly before we need to look at the data for more complex operations such as logarithmic transformations, standardisation or upsampling to avoid introducing bias into our results (which is quite common and we have seen in some studies over this dataset).

We would also like to put emphasis in the fact that we only look at the testing dataset when giving a final result for a set (or subset) of methods and never for cherry-picking hyper-parameters nor making decisions prior to the final test. This also ensures our fairness over the methods. In order to pick parameters and other decisions, we will use crossvalidation when applicable, or if it cannot be done (see data upsampling), we will create another split in the training data to pick them and only when all decisions are taken will we test the 'test' dataset that comes out of preprocessing.

4 ML Methods employed

4.1 Naive Bayes

Even before starting to work with *Naive Bayes* it was easy to see that it would not be the best method given the characteristics of our dataset. As a matter of fact, it has turned out to be the worse performing method.

For Naive Bayes to work, we first we have to make a naive assumption of independence, hence the name of method. This means that we assume that the values of each variable are independent from the others, but in reality, this may not be the case, and more so in our dataset.

In this dataset we have clear examples of correlation between different columns, such as *time in hospital* and columns related to number diagnosis or number of procedures, since more time in hospital probably means more diagnosis and procedures. Like this one, there are some more obvious and not so obvious cases of correlation between variables that tamper with the performance of the *Naive Bayes* method. Even worse, the method cannot detect whether two variables working together raise the probability of a class (it cannot classify a XOR function, for example, which probably is implicit in the behaviour of our dataset).

We do have enough number of elements to obtain reliable probabilities, but this is not enough to achieve satisfactory results for this study, therefore it is expected that the overall results will be worse than others methods.

In our first attempt with *Naive Bayes* we tried to emulate the method as it was explained to us, using *StratifiedKFold* with 10 splits and predicting the results using cross validation over a *GaussianNB*.

Unsurprisingly the results were more than underwhelming, so we decided to set aside this approach.

Next, we tried to adjust the probability threshold to obtain the best possible results. For that, we split our dataset in train and test data, and iterated over the train data in order to find which threshold gave us the best f-score in each iteration.

Once we had these thresholds, we computed the mean, and used it. We obtain the following results:

```
Naive Bayes final report
threshold: 6.040727552857502e-24
accuracy: 0.8806198910081744
              precision    recall  f1-score   support

           0       0.22      0.01      0.02      3409
           1       0.88      0.99      0.94     25951

   micro avg       0.88      0.88      0.88     29360
   macro avg       0.55      0.50      0.48     29360
weighted avg       0.81      0.88      0.83     29360
```

As we can see, even if the accuracy is good, the recall from class 0 is terribly low, meaning that it's predicting virtually every patient to not be readmitted. As conclusion, we believe that we can't better the results due to the problems mentioned at the beginning of this section.

4.2 KNN

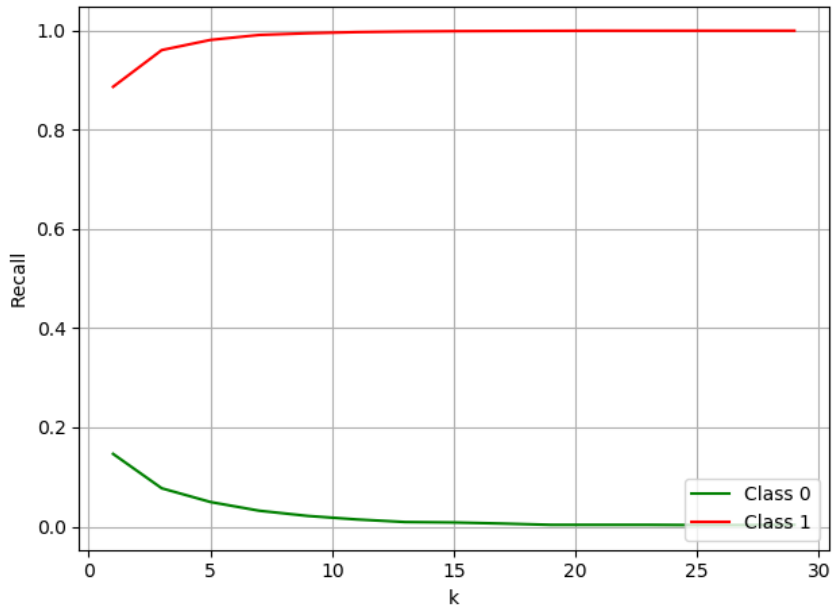
For this technique we needed to downsample our training dataset, since the code takes too long to execute.

We split the training set into a second training and validation data (0.7 and 0.3 respectively). We downsampled the second training data, eliminating 75% of the rows (without balancing).

```
Rows before downsample: 47953
Rows after downsample: 11988
```

Then, we executed our script to determine the best k-parameter using the downsampled training data and the validation data. This script is basically a manual implementation of GridSearchCV modified so it chooses the best k-parameter based on the recall of class 0 instead of measuring the accuracy. The reason behind this is because our dataset is imbalanced, and so we encountered that a higher accuracy meant that the recall of class 0 (re-admitted to the hospital) was rounding 0.00 and class 1 (not re-admitted, or late re-admission probably because of unrelated causes) was rounding 1.00, meaning that it was predicting the same outcome for every row. Since we also have a lot more patients who are in class 1, this results in a high accuracy overall, but it is a bad result.

The script generates the following plot:



As we can see, there is just too much of a difference between the two classes' recall. The best k-parameter is 1 and then class 0 just gets worse every iteration.

The classification report for k = 1 is:

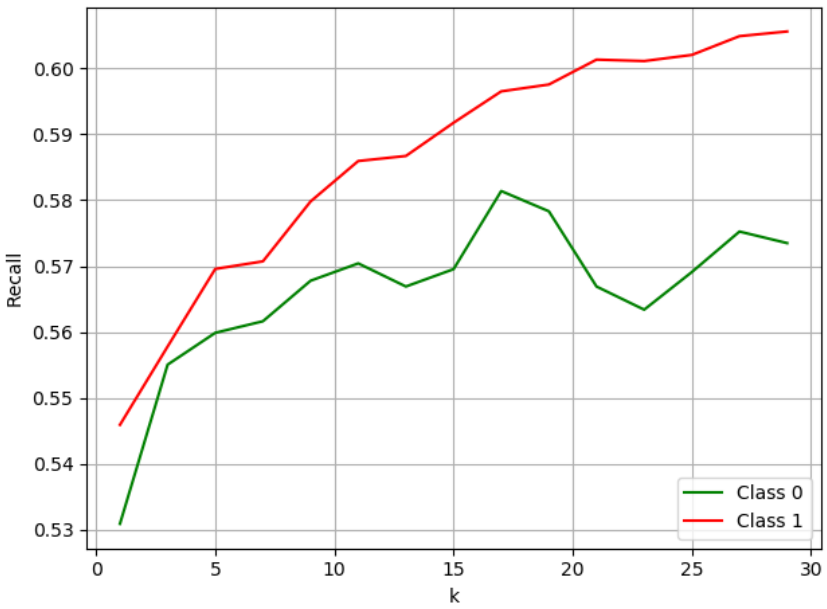
```
KNN with downsampled data final report
accuracy with 1 neighbors: 0.8046905410665629
[[  334  1945]
 [ 2069 16204]]
              precision    recall  f1-score   support

           0       0.14      0.15      0.14      2279
           1       0.89      0.89      0.89     18273

   micro avg       0.80      0.80      0.80     20552
   macro avg       0.52      0.52      0.52     20552
weighted avg       0.81      0.80      0.81     20552
```

Again, even if the accuracy is 0.8, we have to focus on the recall of class 0, which is 0.15, so it's pretty bad.

Now, we will try the same but balancing our training dataset. We downsampled only the elements of class 1, so both classes would have the same number of rows.

class	rows before balancing	rows after balancing
0	5476	5476
1	42477	5476

If we run again the script with the new training data and validation data, we get the following plot:



This looks much better than the other plot, with the k-parameter having some ups and downs but peaking at 13. The recall of class 1 is still greater than class 0, but it was to be expected.

Now we have to downsample the original training data with the same proportion used:

class	rows before balancing	rows after balancing
0	7755	7755
1	60750	7832

The classification report running the original training (downsampled with the same proportion) and the original test data shows as:

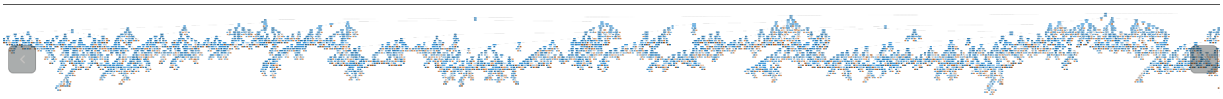
```
KNN final report
accuracy with 17 neighbors: 0.6035762942779291
[[ 1972  1437]
[10202 15749]]
              precision    recall  f1-score   support

           0       0.16      0.58      0.25      3409
           1       0.92      0.61      0.73     25951

   micro avg       0.60      0.60      0.60     29360
   macro avg       0.54      0.59      0.49     29360
weighted avg       0.83      0.60      0.67     29360
```

Obviously both the accuracy and the recall of class 1 have gone down, but now the recall of class 0 is good enough, so we can say that it's best to balance the data and then use 17 as the k-parameter.

4.3 Decision Trees

Even before we started working on implementing the Decision Trees method, we can acknowledge that our data-set is not particularly suited for it. We have 71 numeric variables, which is going to make it difficult to obtain a comprehensible model. Also, we might have issues with over-fitting.

In our first attempt, we used the technique without changes. We are splitting our Train data-set into a sub-Train and sub-Test, using the first subset to train the model and the second for validation. As expected, we got a gargantuan tree, so much so, that it was impossible to read due to resolution problems. We decided to include a picture of it just to illustrate the massive scale:



And that is only a quarter of the whole tree, the actual one is four times wider. However, the numerical results were not terrible:

```
Decision Tree - First attempt
Accuracy on test set: 0.7928133514986376
precision    recall  f1-score   support
```

```
              0       0.15      0.17      0.16      3409
              1       0.89      0.87      0.88     25951

   micro avg          0.79      0.79      0.79     29360
   macro avg          0.52      0.52      0.52     29360
weighted avg          0.80      0.79      0.80     29360
```

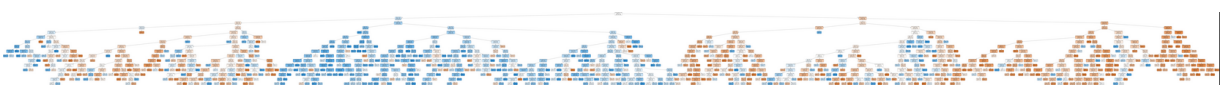
The first thing we tried to change if we wanted to reduce the size of the tree is to have a stricter split criteria or other changes to limit its growth. In *sklearn*, we have several pruning hyperparameters. We can change the minimum number of samples that will trigger a split, the minimum number of samples that may constitute a leaf and the maximum depth of the tree. We decided to increase the minimum number of samples to split and reduce the maximum depth. We achieved a somewhat more comprehensible tree, but to the expense of recall in the zero class.

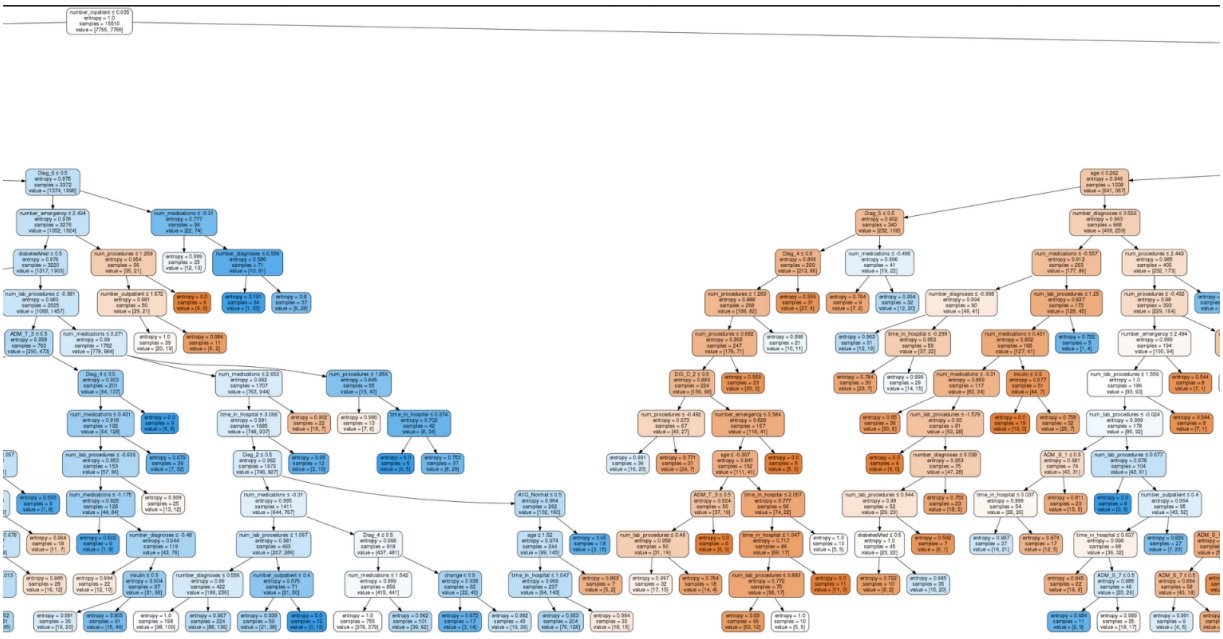
The next thing we wanted to try was to balance the training data, because decision trees are highly sensible to this. Also, we wanted to try to reduce the number of variables, as several of them may be irrelevant and they contribute to noise and also to the size of the tree. We will do that by feature selection.

What we did was, first, do a feature selection using *sklearn's SelectKBest* to select the best 30 features using the *f-classif* function. Then we used a *RandomUnderSampler* to under-sample the over-represented class and therefore balance our data-set. We maintained the same changes regarding the leaf size, the minimum samples to split and the maximum depth. We obtained the satisfactory results. We then used the model to predict the original Test subset of our data-set and obtained the following results:

```
              Decision Trees Final report
              precision    recall  f1-score   support

              0       0.16      0.54      0.24      3409
              1       0.91      0.62      0.74     25951

   micro avg          0.61      0.61      0.61     29360
   macro avg          0.53      0.58      0.49     29360
weighted avg          0.82      0.61      0.68     29360

Confusion matrix on test set:
 [[ 1844  1565]
 [ 9846 16105]]
```

As we can see, we obtained much better results, very similar to those obtained with other methods. The shape of the tree also changed quite a lot, and although still quite big, it is now at least readable. The following two figure show the whole shape of the tree and a zoom in of one of its branches:





However, the tree is still a bit too big to be actually useful at gathering an understanding of the classification visually. Therefore, this method may not be the best approach for solving our problem.

4.4 SVM

Given that our dataset is entirely numerical and normalized with a Gaussian, but in order to start working with this classifier we had restrict the values to [0,1] to work with them. After that, we could start our process. However, since we would want to compare approaches, we also applied a split to get a validation and training from the training set (with the same ratios as with others).

First of all, we noticed that execution time for SVMs is quite high, so training the classifiers with the whole training dataset was unfeasible. Therefore, we followed a different approach: instead of working with a single SVM, we worked with five different SVMs that cooperate to form a voting classifier, each of them trained with a portion of the training data. To acquire those portions, we performed a random under sample of the data, but preserving the ratio of the classes, given that our dataset is highly unbalanced. Our portions had 500 instances of class 0 and 3917 of class 1. With this system, we could work with a representative amount of data from the original dataset while the execution time was still reasonable.

Once we had our portions ready, we started with linear SVMs, which means that no kernel is used to move the data to a higher dimensional space. Our first results were the following:

```
Confusion matrix on validation set:
 [[    0  2327]
 [    0 18225]]
              precision    recall  f1-score   support
```

```
              0       0.00      0.00      0.00      2327
              1       0.89      1.00      0.94     18225

      micro avg       0.89      0.89      0.89     20552
      macro avg       0.44      0.50      0.47     20552
   weighted avg       0.79      0.89      0.83     20552
```

As we can see, the classifier is unable to identify instances of class 0. This is problematic, since the we are interested in the proportion of positive cases detected for this class (Recall). This problem is produced by our dataset being too unbalanced. However, we could fix that with the help of one parameter of the SVMs, the class weight. It is used to give more importance to one of the classes, depending on the given weights. When set to 'balanced', it automatically adjusts weights inversely proportional to class frequencies in the input data. We proceeded with the balanced option, and the results were:

```
Confusion matrix on validation set:
 [[ 1219  1108]
 [ 5482 12743]]
              precision    recall  f1-score   support

           0       0.18      0.52      0.27      2327
           1       0.92      0.70      0.79     18225

   micro avg       0.68      0.68      0.68     20552
   macro avg       0.55      0.61      0.53     20552
weighted avg       0.84      0.68      0.74     20552
```

Now we have a good starting point to work with, so we will keep the balanced class weight parameter for all the experiments with SVMs. Next, we had to optimize the linear SVMs by adjusting the C parameter. Since we are working with five different SVMs, we had to repeat the same process five times, each one with different data portions. For every SVM, we report the best value found, the mean of the recall for the training dataset, and the number and proportion of supports. Finally, we obtain the confusion matrix and the classification report, showing precision, recall and f1-score. This format is being used through the document every time we need to adjust some parameters. After using GridSearchCV to find the optimal value, the results were the following:

```
Split 0
Best value of parameter C found:  {'C': 0.01}
Recall 10-fold cross mean on train data = 0.508
Number of supports:  3970 ( 0 of them have slacks)
Prop. of supports:  0.8988000905592031

Split 1
Best value of parameter C found:  {'C': 0.01}
Recall 10-fold cross mean on train data = 0.526
```

```
Number of supports:  3880 ( 0 of them have slacks)
Prop. of supports:  0.8784242698664252

Split 2
Best value of parameter C found:  {'C': 0.01}
Recall 10-fold cross mean on train data = 0.502
Number of supports:  3874 ( 0 of them have slacks)
Prop. of supports:  0.87706588182024

Split 3
Best value of parameter C found:  {'C': 0.01}
Recall 10-fold cross mean on train data = 0.5720000000000001
Number of supports:  3740 ( 0 of them have slacks)
Prop. of supports:  0.8467285487887707

Split 4
Best value of parameter C found:  {'C': 10.0}
Recall 10-fold cross mean on train data = 0.508
Number of supports:  3520 ( 0 of them have slacks)
Prop. of supports:  0.7969209870953136

Confusion matrix on validation set:
 [[ 1233  1094]
 [ 5741 12484]]
              precision    recall  f1-score   support

           0       0.18      0.53      0.27      2327
           1       0.92      0.68      0.79     18225

   micro avg       0.67      0.67      0.67     20552
   macro avg       0.55      0.61      0.53     20552
weighted avg       0.84      0.67      0.73     20552
```

After trying with the linear kernel, the next step consisted in trying with polynomial kernels. We decided to perform two different experiments with this kernel, changing its degree. The first experiment was performed using a polynomial kernel of degree 2. The preliminary results, without adjusting parameters, were the following:

```
Confusion matrix on validation set:
 [[ 1233  1094]
 [ 5664 12561]]
              precision    recall  f1-score   support

           0       0.18      0.53      0.27      2327
           1       0.92      0.69      0.79     18225

   micro avg       0.67      0.67      0.67     20552
```

```
   macro avg       0.55      0.61      0.53     20552
weighted avg       0.84      0.67      0.73     20552
```

After adjusting the C parameter using GridSearchCV, we obtained the following results:

```
Split 0
Best value of parameter C found:  {'C': 0.1}
Recall 10-fold cross mean on train data = 0.43600000000000005
Number of supports:  4211 ( 0 of them have slacks)
Prop. of supports:  0.9533620104143083

Split 1
Best value of parameter C found:  {'C': 1.0}
Recall 10-fold cross mean on train data = 0.53
Number of supports:  3756 ( 0 of them have slacks)
Prop. of supports:  0.8503509169119312

Split 2
Best value of parameter C found:  {'C': 1.0}
Recall 10-fold cross mean on train data = 0.48599999999999993
Number of supports:  3765 ( 0 of them have slacks)
Prop. of supports:  0.8523884989812089

Split 3
Best value of parameter C found:  {'C': 1.0}
Recall 10-fold cross mean on train data = 0.5519999999999999
Number of supports:  3629 ( 0 of them have slacks)
Prop. of supports:  0.8215983699343445

Split 4
Best value of parameter C found:  {'C': 1.0}
Recall 10-fold cross mean on train data = 0.502
Number of supports:  3694 ( 0 of them have slacks)
Prop. of supports:  0.8363142404346842

Confusion matrix on test set:
 [[ 1242  1085]
 [ 5745 12480]]
              precision    recall  f1-score   support

           0       0.18      0.53      0.27      2327
           1       0.92      0.68      0.79     18225

   micro avg       0.67      0.67      0.67     20552
   macro avg       0.55      0.61      0.53     20552
weighted avg       0.84      0.67      0.73     20552
```

As we can see, the results are really similar to the ones obtained with the linear kernel. After that, we repeated this process but this time using a polynomial kernel of degree 3. Again, the preliminary results were:

```
Confusion matrix on test set:
 [[ 1198  1129]
 [ 5586 12639]]
              precision    recall  f1-score   support

           0       0.18      0.51      0.26      2327
           1       0.92      0.69      0.79     18225

   micro avg       0.67      0.67      0.67     20552
   macro avg       0.55      0.60      0.53     20552
weighted avg       0.83      0.67      0.73     20552
```

And the results obtained after adjusting the C parameter with GridSearchCV were:

```
Split 0
Best value of parameter C found:  {'C': 0.1}
Recall 10-fold cross mean on train data = 0.44000000000000006
Number of supports:  4205 ( 0 of them have slacks)
Prop. of supports:  0.9520036223681232

Split 1
Best value of parameter C found:  {'C': 1.0}
Recall 10-fold cross mean on train data = 0.514
Number of supports:  3700 ( 0 of them have slacks)
Prop. of supports:  0.8376726284808693

Split 2
Best value of parameter C found:  {'C': 1.0}
Recall 10-fold cross mean on train data = 0.47000000000000003
Number of supports:  3712 ( 0 of them have slacks)
Prop. of supports:  0.8403894045732397

Split 3
Best value of parameter C found:  {'C': 1.0}
Recall 10-fold cross mean on train data = 0.5280000000000001
Number of supports:  3591 ( 0 of them have slacks)
Prop. of supports:  0.8129952456418383

Split 4
Best value of parameter C found:  {'C': 1.0}
Recall 10-fold cross mean on train data = 0.506
Number of supports:  3653 ( 0 of them have slacks)
```

```
Prop. of supports:  0.8270319221190854

Confusion matrix on validation set:
 [[ 1203  1124]
 [ 5615 12610]]
              precision    recall  f1-score   support

           0       0.18      0.52      0.26      2327
           1       0.92      0.69      0.79     18225

   micro avg       0.67      0.67      0.67     20552
   macro avg       0.55      0.60      0.53     20552
weighted avg       0.83      0.67      0.73     20552
```

In this case, we observe a slightly decrease in recall for class 0, so for polynomial kernels the best values are obtained using a kernel of degree 2.

The last type of kernel that we tried was the RBF kernel. As we did with the previous experiments, we began with an execution with the default parameters. The results were the following:

```
Confusion matrix on validation set:
 [[ 1216  1111]
 [ 5554 12671]]
              precision    recall  f1-score   support

           0       0.18      0.52      0.27      2327
           1       0.92      0.70      0.79     18225

   micro avg       0.68      0.68      0.68     20552
   macro avg       0.55      0.61      0.53     20552
weighted avg       0.84      0.68      0.73     20552
```

For this type of kernel, the parameters that we had to adjust were C and gamma. Again, we used GridSearchCV for this purpose, trying to find the best combination of both. The results that we obtained were the following:

```
Split 0
Best combination of parameters found:  {'C': 10.0, 'gamma': 0.001}
Recall 10-fold cross on train data = 0.502
Number of supports:  3893 ( 0 of them have slacks)
Prop. of supports:  0.8813674439664931

Split 1
Best combination of parameters found:  {'C': 100.0, 'gamma': 0.0001}
Recall 10-fold cross on train data = 0.5279999999999999
Number of supports:  3782 ( 0 of them have slacks)
Prop. of supports:  0.856237265112067
```

```
Split 2
Best combination of parameters found:  {'C': 100.0, 'gamma': 0.0001}
Recall 10-fold cross on train data = 0.5039999999999999
Number of supports:  3783 ( 0 of them have slacks)
Prop. of supports:  0.8564636631197645

Split 3
Best combination of parameters found:  {'C': 10.0, 'gamma': 0.001}
Recall 10-fold cross on train data = 0.5640000000000001
Number of supports:  3652 ( 0 of them have slacks)
Prop. of supports:  0.8268055241113879

Split 4
Best combination of parameters found:  {'C': 10.0, 'gamma': 0.001}
Recall 10-fold cross on train data = 0.5
Number of supports:  3719 ( 0 of them have slacks)
Prop. of supports:  0.8419741906271225

Confusion matrix on validation set:
 [[ 1232  1095]
 [ 5601 12624]]
              precision    recall  f1-score   support

           0       0.18      0.53      0.27      2327
           1       0.92      0.69      0.79     18225

   micro avg       0.67      0.67      0.67     20552
   macro avg       0.55      0.61      0.53     20552
weighted avg       0.84      0.67      0.73     20552
```

The results obtained with the RBF kernel were almost identical to the ones obtained with the polynomial kernel of degree 2 and the linear kernel. Having found the options that gave us the maximum recall, we needed to decide which one is the best. The rationale for this purpose is to look at the performance of each classifier and the generalisation. Since the performance has been maximised and the values are the same, we need to focus on generalisation. To do so, we will look at the proportion of supports, knowing that the lower the proportion the better the generalisation. For each experiment with the selected kernels, we obtained the average of the proportion of supports for the five SVMs. Those were the results:

```
Linear kernel average prop. of supports: 0.8595879556259906
Polynomial kernel average prop. of supports: 0.8628028073352955
RBF kernel average prop. of supports: 0.8525696173873669
```

After observing the results, we concluded that the best option was the RBF kernel. However, the value is still high, so this classifier will not generalise well.

The last step was to give the final results of our classifier. For this purpose, we repeated the process done with the RBF kernel, using the best values found for each SVM. This time we began the process with the whole training dataset, and used the test dataset that came out from the preprocessing step for validation purposes, which was unused until this point. The final results were the following:

```
SVM final report
 [[ 1788  1621]
 [ 8039 17912]]
              precision    recall  f1-score   support

           0       0.18      0.52      0.27      3409
           1       0.92      0.69      0.79     25951

   micro avg       0.67      0.67      0.67     29360
   macro avg       0.55      0.61      0.53     29360
weighted avg       0.83      0.67      0.73     29360

Average prop. of supports: 0.8501811594202898
```

4.5 Metamethods

4.5.1 Tree bagging, Random Forest, Extra Trees

All these three techniques follow the same philosophy and we expected the same patterns, hence we group them here. However, we consider each of these techniques a different one, so we are going to 'test' each of them once. This means we will use the test three times, since our objective isn't comparing and choosing one out of the three.

We started using a traditional Random Forest. However, preliminary results over a 10-fold cross-validation (which is a reasonable number of folds given that our training dataset is quite massive), showed that a naive approach would not cut it.

class	precision	recall	f1-score
0	0.53	0.01	0.02
1	0.89	1.00	0.94

This kind of classifier is quite useless in itself, since it predicts class 1 (no-readmission) most of the times. This is obvious given the blatant imbalance of the dataset (about 60K of class 1 and 7.8K of class 0). However, the nature of our problem implies a much greater cost if we don't catch a true class 0 (which is a patient with potential risk of readmission). Therefore, we sought to tip the balance of the technique toward predicting class 0 more, even if we reduce its precision.

After the first try, we thought of two possible strategies to improve the recall. First, to shift the threshold at which we predict class 0 to make it trigger more easily. Second, to upsample the training set, adding more instances of class 0.

To move the threshold, we once again used the 10-fold cross-validation, and empirically found the best results at threshold 0.12 (predict class 0 if the probability of being class 0 exceeds 0.12).

This number is not insignificant, as with several other tests it also came out. To be precise, 0.12 is the ratio between class 0 and 1 (7.8/60), so shifting the probabilities this way 'tips' the balance to correct the difference of instances. The CV result:

class	precision	recall	f1-score
0	0.16	0.56	0.25
1	0.92	0.62	0.74

To upsample the dataset, we could not use cross-validation. Other studies do use cross-validation with an upsampled dataset, but this will only give extremely inaccurate results (see example in the notebook). Instead, we did a split of the training data, and we upsampled one of the subsets, leaving the other for validation. The results were better than in the original approach, but much worse than with the threshold:

class	precision	recall	f1-score
0	0.34	0.03	0.05
1	0.89	0.99	0.94

We also tried both techniques simultaneously, using another split. The results were not much better than the first one:

class	precision	recall	f1-score
0	0.16	0.52	0.24
1	0.92	0.65	0.76

Therefore, for the final test, we decided to go with a threshold .12 and no upsampling. We trained it with the whole training set and predicted the test set from preprocessing. We achieved the following results:

```
Random forest final report
              precision    recall  f1-score   support

           0       0.17      0.57      0.26      3409
           1       0.92      0.63      0.75     25951

   micro avg       0.62      0.62      0.62     29360
   macro avg       0.54      0.60      0.50     29360
weighted avg       0.83      0.62      0.69     29360

Random forest final report
 [[ 1927  1482]
 [ 9661 16290]]
```

For Extra trees, we saw a similar tendency, and in the end we also went for the same threshold and no upsample (further intermediate results in the notebook):

```
Extra Trees final report
              precision    recall  f1-score   support

           0       0.17      0.52      0.25      3409
           1       0.91      0.65      0.76     25951
```

```
   micro avg       0.64      0.64      0.64     29360
   macro avg       0.54      0.59      0.51     29360
weighted avg       0.83      0.64      0.70     29360

Extra Trees final CM
 [[ 1780  1629]
 [ 8981 16970]]
```

Finally, tree bagging, which should be the same as a Random Forest:

```
Tree Bagging final report
              precision    recall  f1-score   support

           0       0.16      0.53      0.25      3409
           1       0.91      0.64      0.75     25951

   micro avg       0.62      0.62      0.62     29360
   macro avg       0.54      0.58      0.50     29360
weighted avg       0.82      0.62      0.69     29360

Tree Bagging final CM
 [[ 1802  1607]
 [ 9451 16500]]
```

Even though our objective wasn't to compare the methods, we consider that all three perform mostly the same (probably a McNemar test should show they are equally good).

4.5.2 Adaboost, GradientBoost

Here we will look at the performance of Adaboost using decision stumps, Adaboost using decision trees with a max depth of 5 and Gradient Boost.

We used a similar process to the one used in the previous section, making a preliminary prediction over a 10-fold-cross-valiation and then trying to move the prediction threshold, trying to upsample the training set and finally doing both.

In all three types of boosting the preliminary results were very bad. This is the Gradient Boost one, for example:

class	precision	recall	f1-score
0	0.56	0.01	0.03
1	0.89	1.00	0.94

Afterwards we compared the performance of the three modifications (applying a threshold, upsampling and doing both).

Firstly, we found that while just upsampling improved the performance in all three boosting types, it did so minimally in comparison to the other two modifications.

Secondly, we found that for both types of Adaboost the two other modifications were better than just upsampling but they didn't present a significant difference between eachother. The final results for the Adaboosts, using just a modified prediction threshold of 0.495 when using decision stumps and 0.494 when using decision trees, were as follows:

```
Adaboost with decision stumps final report
              precision    recall  f1-score   support

          0       0.18      0.51      0.27      3409
          1       0.92      0.70      0.80     25951

avg / total       0.83      0.68      0.74     29360

Adaboost with decision stumps final CM
 [[ 1724  1685]
 [ 7657 18294]]


Adaboost with decision trees final report
              precision    recall  f1-score   support

          0       0.15      0.55      0.23      3409
          1       0.91      0.59      0.71     25951

avg / total       0.82      0.58      0.66     29360

Adaboost with decision trees final CM
 [[ 1862  1547]
 [10768 15183]]
```

Finally in Gradient Boost we found that while the other two modifications presented a better performance than just upsampling, amongst them a modified threshold of 0.11 with no upsampling presented the best performance:

```
Gradient boosting final report
              precision    recall  f1-score   support

          0       0.18      0.61      0.27      3409
          1       0.92      0.63      0.75     25951

avg / total       0.84      0.62      0.69     29360

Gradient boosting final CM
 [[ 2075  1334]
 [ 9702 16249]]
```

5 Comparisons and conclusions

After having completed all of the different methods to try to predict if a patient should be readmitted or not, now is time to do a global comparison. Given the nature of the problem, the cost of each type of miss-classification is different. We had a special focus on the recall of Class 0, which is a patient with potential risk of readmission. The following table compares the different methods according to our criteria.

Method	f1-scr-c0	f1-scr-c1	recall-c0
NB	0.02	0.94	0.01
KNN	0.25	0.73	0.58
DT	0.24	0.73	0.54
SVM	0.27	0.79	0.52
MM-RF	0.26	0.75	0.57
MM-ET	0.25	0.76	0.52
MM-TB	0.25	0.75	0.53
MM-ABDS	0.27	0.80	0.51
MM-ABDT	0.23	0.71	0.55
MM-GB	0.27	0.75	0.61

We can observe that, overall, the results are quite similar, with an exception of Naive Bayes, which is by far the worst performing one. The rest of the methods achieve somewhat similar results, probably close to the best that can be done with this data-set. KNN performs quite good and if we consider its structural simplicity, it has a lot of credit. Regarding the meta-methods, we see that tree-like methods are satisfactory with thresholding, but the boosting methods seem better. Gradient Boosting seems to be the method that gave us the best results.

Naive Bayes may perform horribly because there is a lot of interaction between variables. For example, when seeing the Decision Trees, we could see that administration of a medication was good for readmission when several other factors were there, and bad when they were not. This method's naivety would not detect it, as it simply increases the class' probability based on the value of a single variable. The rest of methods are capable of seeing this, thus the abysmal diference.

KNN worked well once we balanced the dataset, which makes sense since this method is quite sensible to unbalanced data-sets.

We discourage using decision trees for obtaining a comprehensible model for this problem. It is very hard to achieve a comprehensible model that could explain the classification visually, which is one of the main points of using decision trees. Later on, we will explain our theory so as to why it is not possible to obtain a comprehensible model on this problem.

SVM gives one of the best results, however it is quite slow and required a down-sample to achieve desirable execution times. The Kernel method and increased dimensionality work better than most of these methods by the same reason the rest of methods work better than Naive Bayes.

Regarding the meta-methods, they achieved the best results in general, comparable only to SVMs. Both Ada-boost with decision stumps and Gradient Boosting gave pretty satisfactory results compared to the rest of the methods. Specifically Gradient Boosting appears to be the best of all.

All of these facts made us think about the problem itself, and the reason both classes exist. Class 0, the readmitted patients, are **generally** readmitted because of an ineffective or harmful treatment.

From this affirmation, we extract two things: not all of them come because of their previous visit, some readmissions might be because of external factors we simply cannot get. For example, a person who came because of a liver infection is readmitted shortly after because of a car crash. In our dataset, this would be a class 0 (readmitted), however there is absolutely no way to predict this, and also it is not really interesting for our problem, since we want to predict readmissions due to ineffective or harmful treatments.

Furthermore, from the affirmation we also extract that hospital personnel fight against this. Doctors and experts are trying to give the best treatment and pushing the patients toward the class 1 (not readmitted in a short timespan). These experts are highly formed and have more data than our dataset can get (since they might ask the patient about things we don't have recorded, such as family history or several other things). This means that if a model as simple as a small decision tree existed, these experts should have already found it, and thus prevented the class 0 instances (making the model useless).

However, experts do make mistakes sometimes, and this is what we wanted to prevent with our analysis. Using any of our best methods (SVM or a Metamethod), we can detect one out of two cases of readmission (which given the fact that not all of the readmissions are explainable seems like a good ratio), at the cost of diagnosing as possible readmitting patients four out of five non-readmitting ones. However, the nature of our problem tells us that our model would 'warn' an expert of a possible re-admittance, which should then be investigated by the expert, and so the cost of this miss-classification is not grave. The keypoint here is that the use of the model should be a warning system to an expert to further investigate, since one out of five cases could be a possible readmittance due to bad treatment.

As a final note, between all of the classifiers trained, we consider that the best one to be Gradient Boosting because of its metrics, with some other methods close behind. Given the similarity in metrics, to give a definite answer, we would need to run the McNemar test between the top ranking ones, which was not done because of the scope of the project and the time limitations.