Automating document error detection from reimbursement requests

Business objective
Process Optimization
Sector
Bank and Insurance

Document error detection to reduce errors and manual processes

If you ask your customers to fill in a template document, there might be errors in the information provided by the customers. It means that you must manually check if there are errors before validating the document. This way of working is a **costly process** that requires a lot of employees to verify each document.

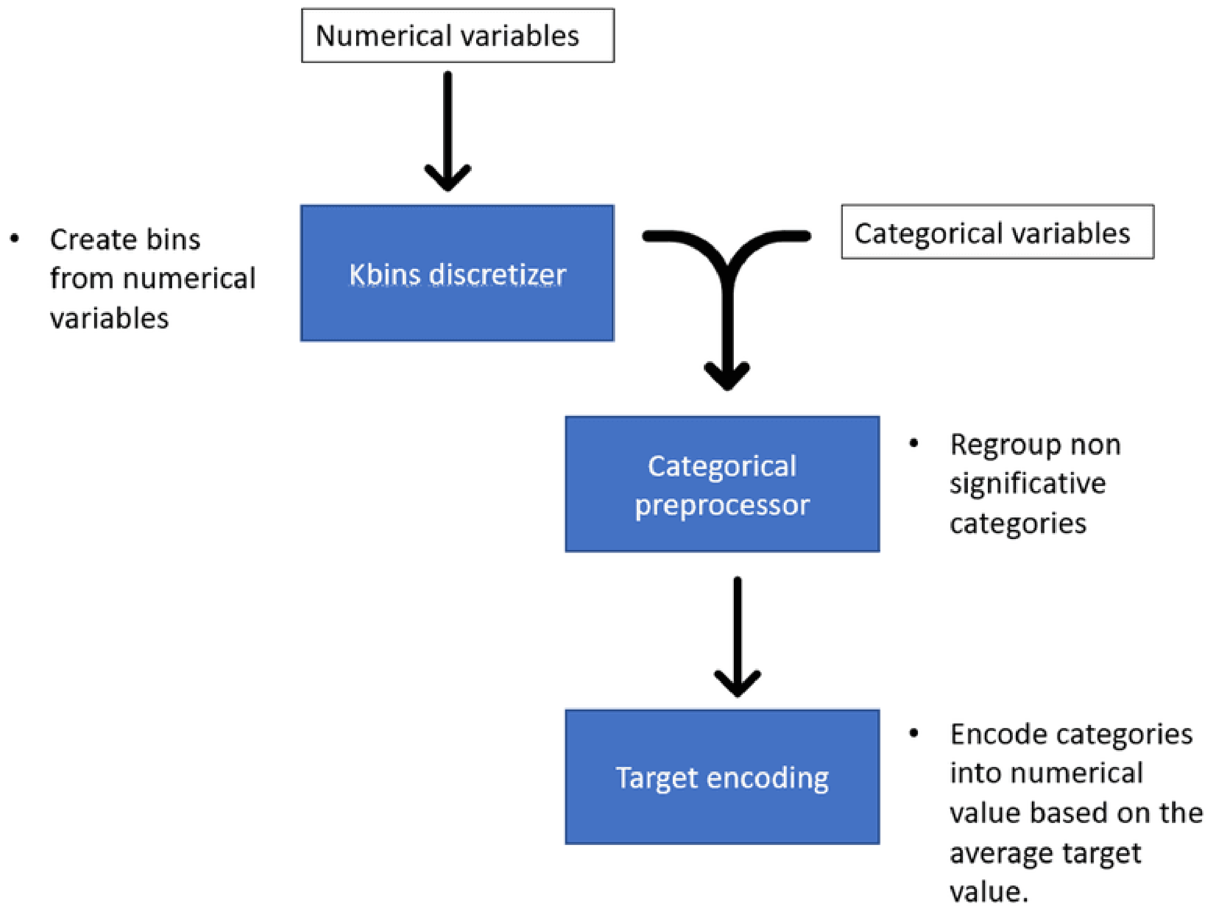
An insurance company contacted us because they receive more than 100,000 bills from their customers for medical reimbursement per year. The following document error detection project consisted of classifying bills into two categories: 1) the document requires modification, or 2) it does not. The input data has 117,000 bills and 312 features. We can break it down into the features:

- Flag features about the presence of a particular code line in the bill
- Amounts displayed on the bill (reimbursement amount, supplementary amount, total amount)
- Type of hospital, rooms, and invoices
- Metadata about the bill (number of entries and dates)

Most of the errors come from using the wrong code for services received by the hospital. Then it comes from the date used in the bill that is not the correct one, or the amount specified is not correct either.

Preprocessing the data for document error detection

Before sending the data into the model for our document error detection, it is better to preprocess it first. We can create new variables to yield better performance, which is called **feature engineering**. We must categorize continuous variables into bins. Then we must transform categorical variables into numerical variables so the model can process them.



Numerical variables
Create bins from numerical variables
Kbins discretizer
Categorical variables
Categorical preprocessor
Regroup non significative categories
Target encoding
Encode categories into numerical value based on the average target value.

Feature engineering for document error detection

To achieve this, we use the cobra library, which implements several functions to preprocess the data. **Cobra** is handy because you can apply and fine-tune different preprocessing functions using only a preprocessor object.

Feature engineering

We can try to help the model **discover new patterns in the data by creating new variables**. The interaction between different variables can make these variables. For example, we create the proportion of reimbursement amount by dividing the reimbursement amount by the total amount.

We can also find patterns in data through time. So, if you have a date variable, you can use it to create a month, weekday, and season variable. The model might find hidden cycles appearing during months, weekdays, or seasons.

Kbins Discretizer

Kbins Discretizer will **bin continuous data into intervals of a predefined size**. It is interesting because it can introduce nonlinearity to the model. You can set a number of bins and the strategy used to create the bins. There are three strategies in KBinsDiscretizer:

- Uniform: The bin widths are constant in each dimension.
- Quantile: Each bin has the same number of samples.
- Kmeans: The discretization is based on the centroids of a KMeans clustering procedure.

Categorical data preprocessor

The Categorial data preprocessor implemented by Cobra will ***regroup the categories of categorical variables*** *based on significance with the target variable*. A chi-squared test determines if a category *is significantly different from the rest of the categories for the classification model, given the target variable*.

We can set the p-value threshold to consider at which p-value the categories are significant or not. **If the category is not significant**, it will be **regrouped** under the "Others" category.

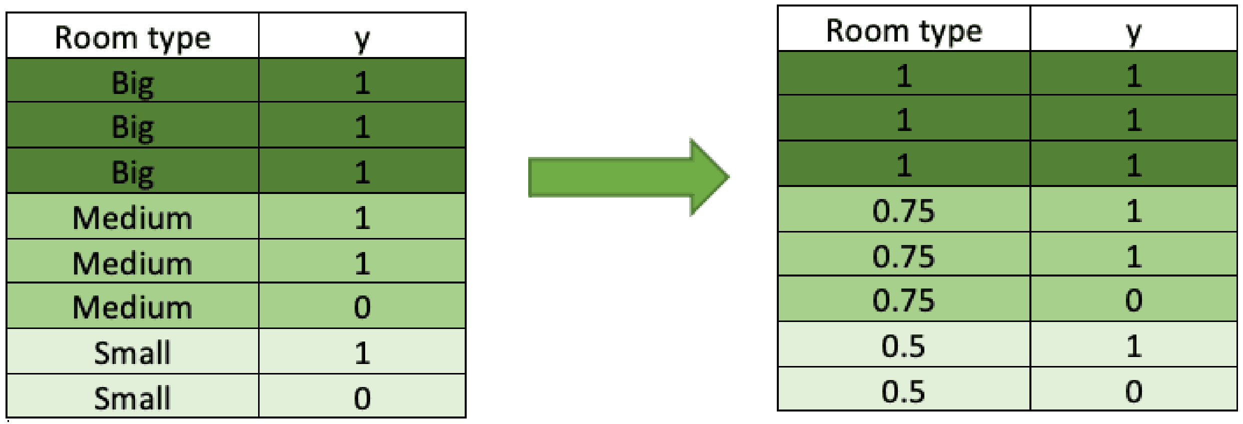
We can also set the minimal size of a category to keep it as a separate category. **It** will be **regrouped** under the "Others" category if the category size is below this minimal size.

There is also an option to replace the missing values with the additional category "Missing."

This module enables us only to keep relevant categories and diminish the number of isolated data samples in the dimension space.

Target encoding

Target encoding (or Mean encoding) computes the **ratio of positive occurrences** in the target variable for each feature. It is a way **to transform categorical values into numerical values**.



Room type	y
Big	1
Big	1
Big	1
Medium	1
Medium	1
Medium	0
Small	1
Small	0

Room type	y
1	1
1	1
1	1
0.75	1
0.75	1
0.75	0
0.5	1
0.5	0

The target encoding method for document error detection

Target encoding is the preferred method over the traditional one-hot encoding because it does not add additional columns. One-hot encoding would add many dimensions to our dimensional space that is already composed of 300 features. A high-dimensional space has two problems:

- It increases the usage of memory and computation.
- It increases the capacity of the model to overfit the data.

So target encoding enables to reduce the memory and computation usage and the overfitting of the model.

However, target encoding might still introduce **overfitting** because it computes the ratios based on the training samples we give. The ratios might not be the same in the test sample. For example, in our test sample, 'Small rooms' might have a ratio of 0.75 instead of 0.5. We can reduce this problem by using **additive smoothing**. Cobra chooses to implement it for its target encoding class. When computing the ratio of positive occurrences for each categorical value, it adds the ratio of positive occurrences of the target variable proportionally to a weight. Max Halford explains it in a detailed manner in his article.

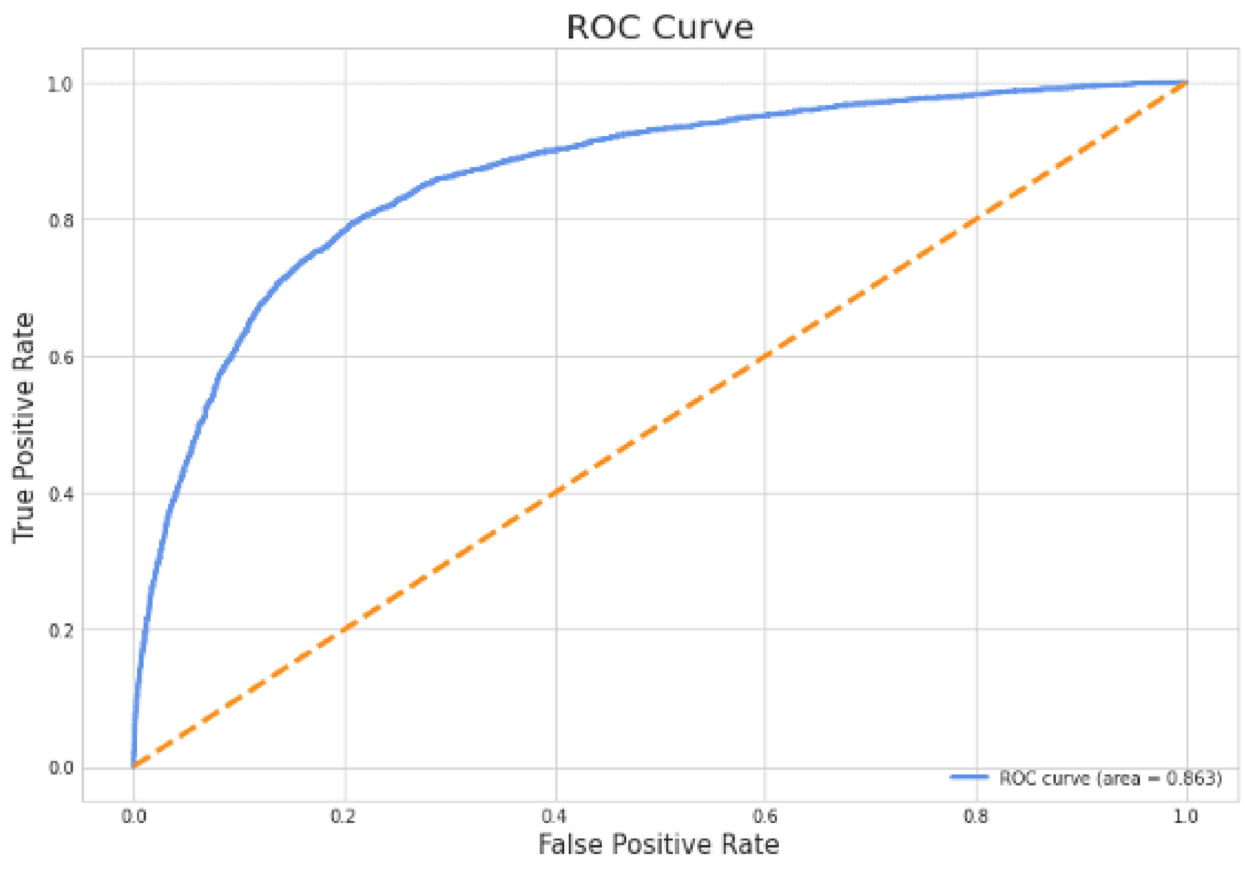
Model and results for document error detection

Based on the client's need, the appropriate metric is the Area Under the Curve (AUC).

We select the features that increase the AUC substantially. To avoid having too many dimensions and reduce the possibility of overfitting, we only selected the features that increased the AUC by a substantial margin. In other words, we performed a **forward selection**.

The best model is the **Gradient boosting** tree. We use the **XGBoost library,** an optimized distributed gradient boosting library designed to be highly efficient, flexible, and portable. The library implements early stopping to avoid overfitting the training data. It is also possible to specify other parameters, such as the maximum depth of each tree or the minimum sum of instance weight needed for a child. You can check all the parameters available here.

We achieved an AUC of 86.3%.



ROC Curve as a result of the Document Error Detection project

Conclusion

We were able to automate 47% of the bills with an error rate of 3.06%. It means that around 55,000 documents did not need manual review, saving countless hours for the client.

About Agilytic

Since 2015, Agilytic helps innovative leaders solve their biggest challenges through the smarter use of data. With over 150 successful projects to date, we have perfected a pragmatic approach to putting data at the service of business goals, be they commercial, operational, financial, or human. Reach out today for a quick introduction, we'd love to hear from you.