# ML Algorithms for reserving

## A case study from Kaggle

## Introduction

Machine learning advances at an unprecedented pace these days. ML algorithms can read traffic lights and are expected to drive cars soon. From an actuarial perspective, the next step is necessarily that they adequately reserve for accidents they might cause.

Jokes aside, the question of how and how much such algorithms can be integrated into the reserving processes of insurance companies is very real and urgent. Companies making use of these new solutions might gain an edge over their competitors, on the other laying themselves open to their critical aspects, for example lack of transparency.

The “Actuarial Loss Prediction” (ALP) competition aimed to further this discussion by creating a reserving challenge that goes beyond conventional methods. The challenge was hosted on Kaggle, the leading platform for data science competitions with a community of over 8 million data scientists, and was presented as follows (see [1]):

The Actuaries Institute of Australia, Institute and Faculty of Actuaries and the Singapore Actuarial Society are delighted to host the Actuarial loss prediction competition 2020/21 to promote development of data analytics talent especially among actuaries. The challenge is to predict Workers Compensation claims using highly realistic synthetic data.

The data is fully synthetic and not specific to any legal jurisdiction or country. We are grateful to Colin Priest for building and supplying the dataset.

We invite the competitors to take claims inflation into account.

The ALP ran from December 2020 to April 2021, keeping 140 teams of actuaries and data scientists busy. At the end we retained the second place and some good insights about how machine learning in reserving might look like in the future, which we would like to share in this article.

## Aim of the competition

The introduction of the ALP already provides a succint description of the aim: “The challenge is to predict Workers Compensation claims using highly realistic synthetic data.”. In this section we provide a bit more detail on the aim of the competition and the scoring metric.

The 90.000 rows dataset was divided into a train set (54.000 rows) and a test set (36.000 rows). Approximately 50% of the test data was retained by the competition host and only used once for the final evaluation of the models.

The typical ML workflow is to train the algorithm on the train set and check its predictions on a validation set, to make sure it is not too tightly tailored to the data it was trained upon. Based on the results on the validation set the model is fine-tuned, and in the last step - to check whether the model was indirectly tailored upon the validation dataset - the model is evaluated on the unseen test set. An algorithm that is able to perform on the test set has generalized well and successfully neglected the idiosyncracies of the training data.

In this competition the predictions were scored according to the root mean squared error (RMSE).

## Data Exploration

The train dataset contained the fields below. The test dataset contained the same fields except for the explained variable *UltimateIncurredClaimCost*.

* **ClaimNumber**: Unique policy identifier
* **DateTimeOfAccident**: Date and time of accident
* **DateReported**: Date the accident was reported
* **Age**: Age of the worker
* **Gender**: Gender of the worker
* **MaritalStatus**: Martial status of the worker: (M)arried, (S)ingle, (U)nknown
* **DependentChildren**: The number of dependent children
* **DependentsOther**: The number of dependants excluding children
* **WeeklyWages**: Total weekly wage
* **PartTimeFullTime**: Whether the worker was employed (P)art time or (F)ull time
* **HoursWorkedPerWeek**: Total hours worked per week
* **DaysWorkedPerWeek**: Number of days worked per week
* **ClaimDescription**: Free text description of the claim
* **InitialIncurredClaimCost**: Initial estimate by the insurer of the claim cost
* **UltimateIncurredClaimCost**: Total claims payments by the insurance company

Outliers were removed or recoded based on expert judgment. Some new features were added, e.g. whether the accident happened within core working hours. Date(time) features were also used to create new variables, such as reporting delay. The explanatory variables were partly transformed to alleviate or reinforce the effect of extreme data points. For instance, polynomial features were created for the variables *DependentChildren*, *DependentsOther*.

Different large claim thresholds were tested for different kinds of models. Moreover, several models were calibrated for large claims. However, explicit large claim modelling did not lead to significant model improvements. Therefore, the large claims were accounted for implicitly in the final model.

## Natural Language Processing

The chapter is titled Natural Language Processing (NLP) because the *ClaimDescription* was analyzed in part with NLP techniques. NLP techniques range from overarching sentiment analysis, which describes the overall emotional content of a piece of text, to syntactic analysis, which breaks the text in sentences and tokens to be analyzed singularly (e.g. Google Cloud’s natural language API, see [2]).

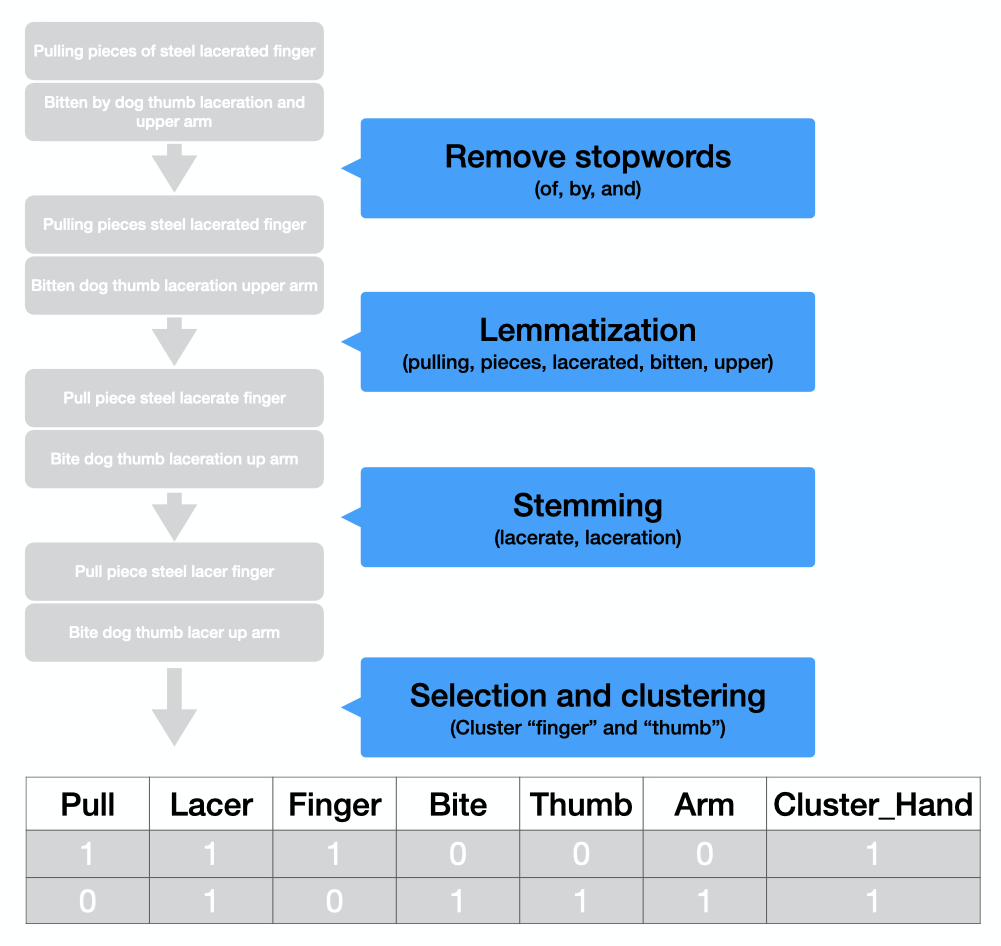
Unfortunately, the full potential of NLP techniques could not be exploited because of two reasons. First, the descriptions were sometimes too short, ruling out e.g. sentiment analysis. Second, for data protection reasons some descriptions had been distorted to a point where they became hardly intelligible, e.g. “TO RIGHT LEG RIGHT KNEE”. Therefore, NLP was combined with simple text analysis to extract information from the claim descriptions.

The descriptions were preprocessed as follows: First, stopwords were removed to reduce the noise in the data. The set of stopwords were taken from the nltk corpus ([3]). Second, the words were lemmatized using SpaCy ([4]), as well as stemmed using the SnowBall stemmer of nltk. Lemmatization reduces the word to its form that appears in the dictionary, stemming reduces the word to its root form. The former produces semantically meaningful word forms, whereas the latter may not. For example, the lemma of lacerated is lacerate, the stem form of lacerated, laceration or lacerate is lacer.

Next, we introduced 3 categories: body part, type of wound, and accident cause. For each category we defined a list of words based on the dataset as well as our own judgment. The lists contained words like: ankle, knee, eye (body parts), strain, bruise, fracture (types of wound), slip, explosion, spider (accident cause). Moreover, we grouped these words based on semantic similarity in order to allow for potential inconsistencies in the labeling. For instance, face, cheek and jaw were clustered together, the rationale behind being that a face injury may be very similar to an injury of the cheek or jaw.

For each word in the lists, a new column was created and added to the dataset. The columns contained the number of occurences of the word in the claim descriptions. For each cluster, a new column was created as well. The value of these columns were 1 if any of the words in the cluster appeared at least once in the claim description, otherwise 0. All these numeric columns representing the claim descriptions were given as inputs to the model.

The workflow is summarised in the picture for two claim descriptions:



NLP Workflow

Clustering and stemming make sure that the model can detect the common aspect of the two description, i.e. a laceration on the claimant’s hand, while differentiating its cause (respectively pulling something and a bite) and retaining more specific information.

## Modelling with xgboost

We implemented our model in Python making use of the xgboost package. In the words of its own authors this library “implements machine learning algorithms under the Gradient Boosting framework”, and we applied it to boost a random forest algorithm.

In this section we shortly refresh the concepts of random forest and gradient boosting and then describe the way xgboost introduces probability distributions in its algorithms.

### Random Forests and Gradient Boosting

Here is a short recap of the concepts of random forests and gradient boosting:

* **Random forests** are made up of decision trees - hence the “forest” -, each of which is only allowed to train on a randomly selected subset of the training set - hence the “random”. Each of the decision trees populating the forest predicts a result, and a unique result is then provided by taking the average of all results (for regression problems) or the most common one (for classification problems).
* **Gradient boosting** is a machine learning technique in which the same algorithm is applied iteratively. After a first prediction has been provided, we fit a new model on its residuals - this is the first boosting round. We can then combine the first two models and model their residuals, which would be the second boosting round, and so forth.

In combining random forests and gradient boosting we refine the classic decision tree algorithm in two different ways. The predictions improve at the cost of interpretability: building up a forest we lose the possibility of looking at the individual splits of our decision tree, and each boosting round adds a new forest. A simple way of visualizing feature importance is the F-score, which counts how many times a variable was split on.

### Regression to distributions in xgboost

Suppose that we want to predict a numerical outcome Y based on a set of n dependent , and that we suspect to be gamma distributed with mean : a common way to approach this regression problem would be to set up a generalized linear model (GLM) around the equation

where is a vector of regression parameters to be determined and we used as link function. We can determine the s by maximizing likelihood.

This approach cannot be translated directly to decision trees, which describe the outcome in terms of piecewise constant functions rather than polynomials. However, it is possible to introduce the likelihood in the computation via the loss function.

The learning process of a machine learning algorithm is guided by minimizing a loss function. Given a vector of true values and of model outcomes we can, for instance, ask for the that minimizes the squared error or the squared log error . One typical issue of the squared error is that it is driven by large outcomes - a small percentual error in predicting a large value, possibly due to noise, has a great impact on the squared error. This effect is tamed by the squared log error. So we can choose different loss functions to target different peculiarities in the data and guide the learning process of the algorithm.

Loss functions are in principle fully customizable, but xgboost already provides a wide selection, including the negative of the log likelihood to logistic, gamma and tweedie distributions with the link function. Minimizing the negative of the likelihood obviously maximizes the likelihood; this is, however, not achieved by finding the optimal betas (as in GLMs), but by finding the best splits for each decision tree.

In practice, for a Gamma distribution with mean and shape parameter we have the density function

We can set the *objective* parameter of xgboost to *reg:gamma* to obtain the negative log likelihood as loss function:

Here is interpreted as a weight parameter and defaults to 1. Notice also that in this computation we took for simplicity the identity as link function. For details, please see the source code of xgboost ([5]) at and the discussion at [6].

## Model analysis

An xgboost model can be quite complex. These are some of the parameters that need to be set, with some sample values:

'objective':'reg:gamma,  
'eval\_metric':'rmse',  
'eta': 5e-2,  
'max\_depth': 4,  
'min\_child\_weight': 6,  
'subsample': 0.7,  
'num\_parallel\_trees': 50  
'alpha' : 2e+2

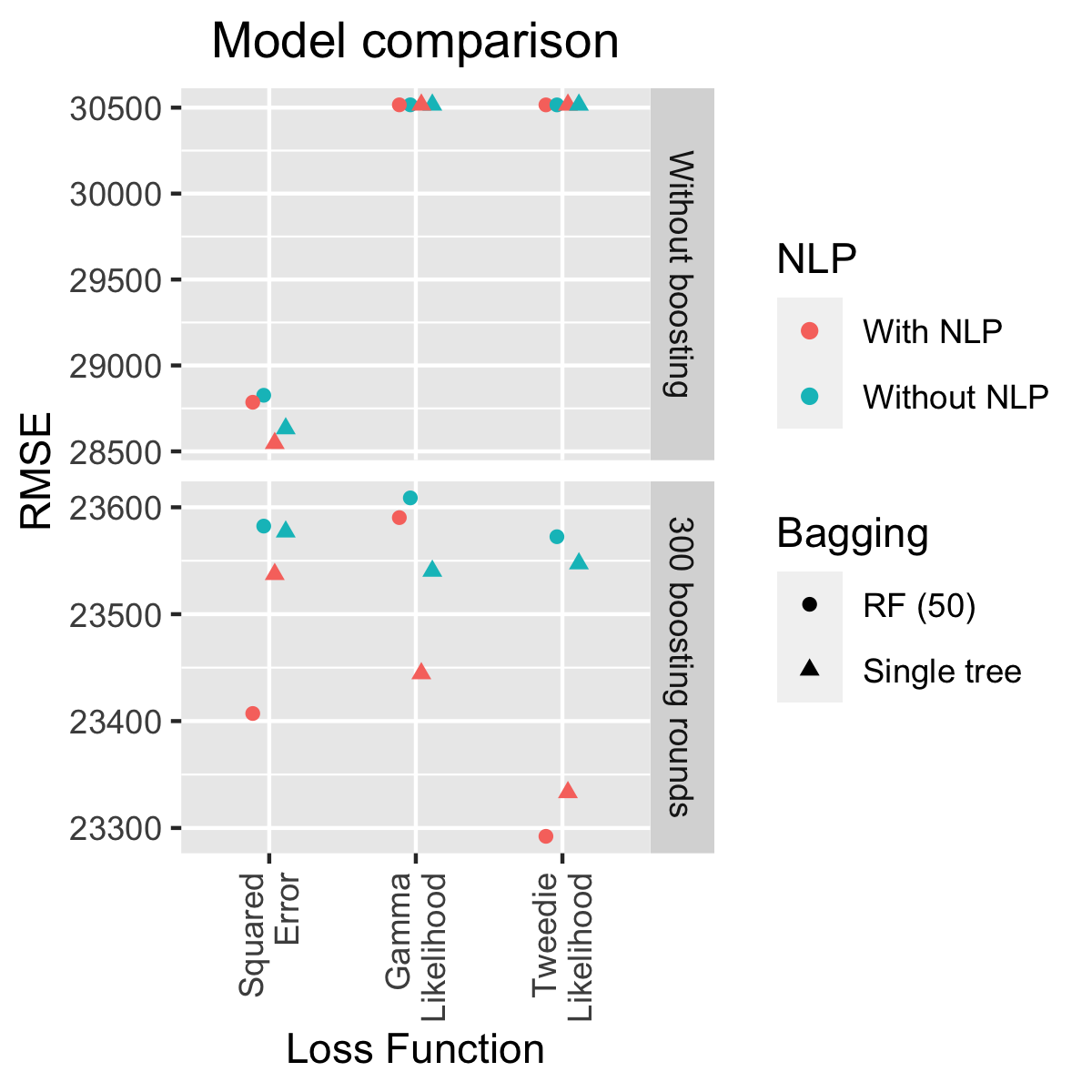
For a description and complete list of parameters, please see the official xgboost documentation ([7]). Once the parameters are set, we can train a model for a given number of boosting rounds as follows (Python):

import xgboost as xgb  
xgb.train(params=parameters, dtrain=xgb.DMatrix(input\_data, label=y), num\_boost\_round=300)

### Model comparison

It is important to understand that the parameters interact with each other, and several optimal configurations might be possible. Moreover, the data is usually preprocessed to make it easier for the model to interpret: the preprocessing and feature engineering performed on the data interacts with the hyperparameter as well, so that the final model is much more than a simple sum of input data and hyperparameters.

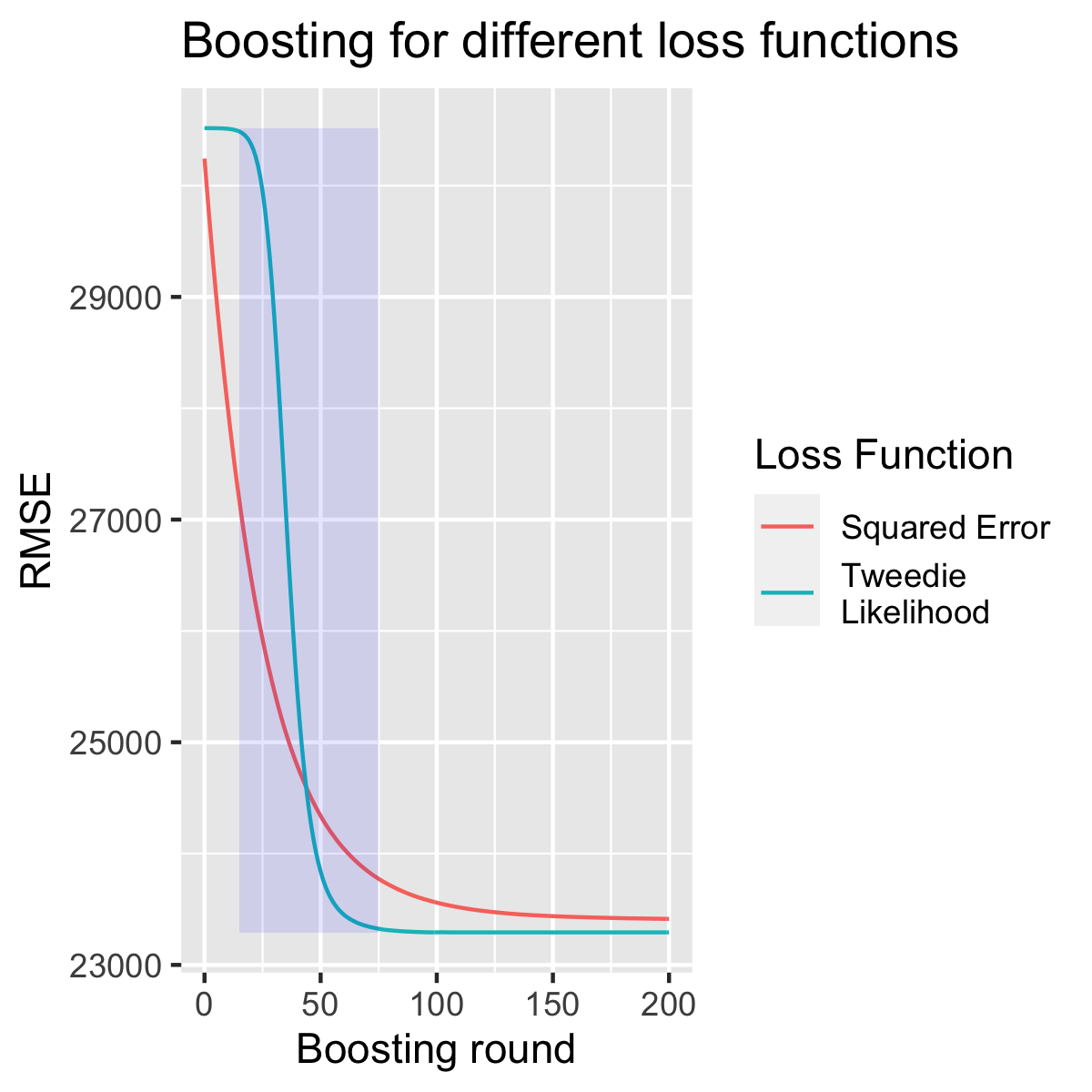
With this remark in mind, we would like to proceed to an analysis of the model to try to describe which aspects played the most important role in producing good predictions. In the figure below, we display the RMSE of the predictions on the test dataset for different choices of hyperparameters and feature engineering: we considered all possible combinations of the three loss functions squared error, gamma distribution and tweedie distributions, NLP analysis (with vs without), boosting (300 boosting rounds vs no boosting), bagging (random forest with 50 trees vs single tree).



Model Comparison

From the plot we can see that the major improvement is given by inserting boosting. It should also be noted that unboosted models essentially do not make use of the NLP features and are not powerful enough to detect the distribution behind the data: for unboosted models the squared error is by far the best loss function, for boosted models it is outperformed by tweedie likelihood.

We also see this in the figure below, which compares the drop in RMSE on the test set after each boosting round for a model with squared error and tweedie likelihood loss function: this drop is much more regular for the squared error loss function, which is directly related to RMSE, and more of a side effect for the tweedie likelihood loss function. In the latter case the RMSE abruptly drops in the blue shaded area (boosting rounds 15 to 75) and then stabilizes again.

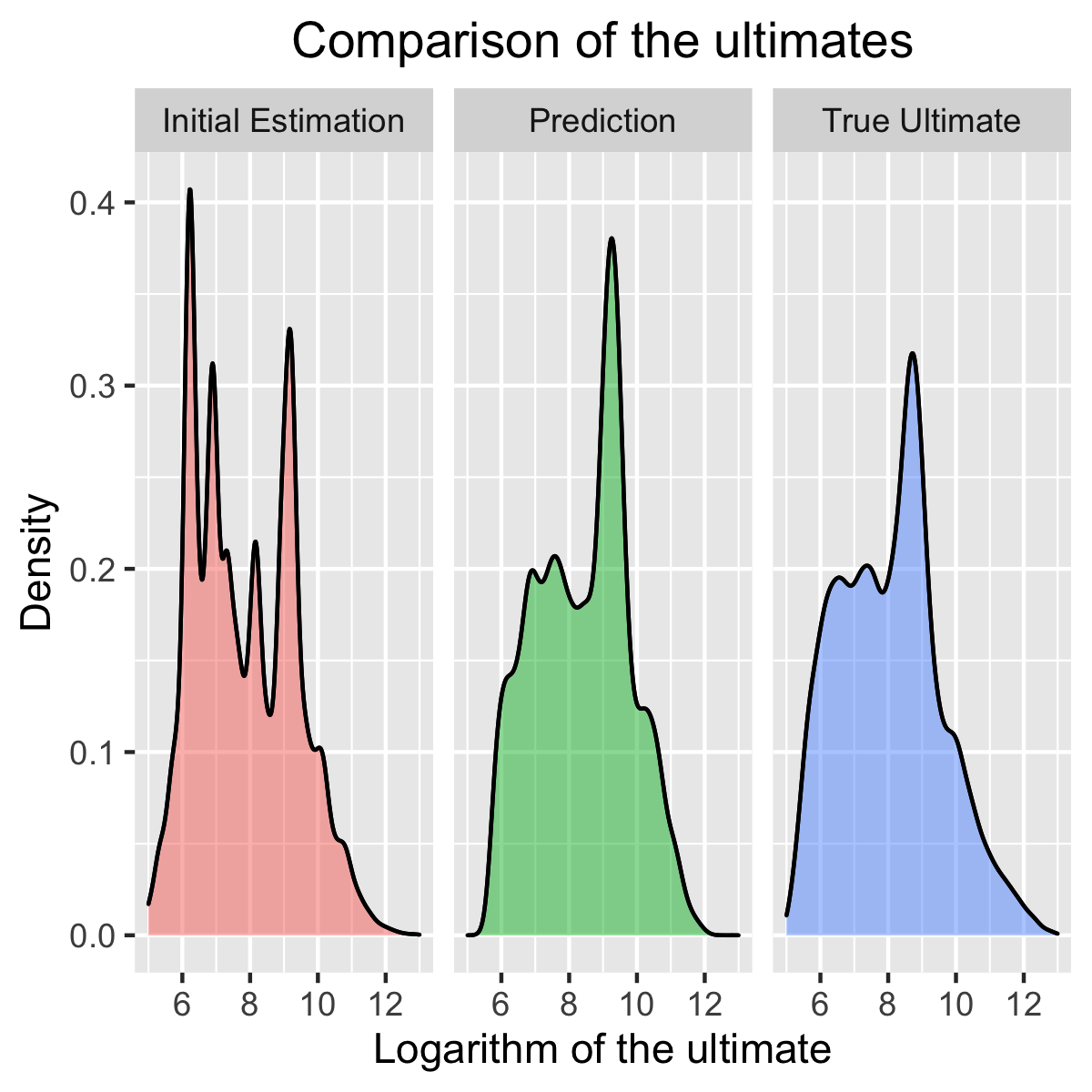


Boosting for different loss functions

### Backtesting

The input data for the prediction contained an initial estimation for the ultimate cost. The model essentially took and boosted this initial estimation using the remaining predictors.

The next plot shows the distribution of the logarithm of the ultimate for the initial estimation, the prediction and the true ultimate in the train data:



Ultimate Comparison

One of the main banes of this challenge was the prediction of large losses, which repeatedly led the model astray: the fat tail of the real distribution shifts the core of the prediction distribution to the right, overestimating small values, but the fitted tail is still too thin to correctly predict large values. For our final submission we followed a hybrid approach, choosing a large loss threshold and combining our models differently for large and small losses.

## Conclusion

This competition was a useful exercise in machine learning and in discovering its potential in the insurance industry. While experimenting we realised how important it was to have a good grasp of the theory behind the models: choosing to use random forests only leads to more questions, for example how many trees the random forest should be made up of. Hyperparameters can be fine-tuned automatically in Python, e.g. using the Ray package, but already choosing a reasonable range of hyperparameters has been at times challenging.

We also tried several different machine learning algorithms, including neural networks. While boosting on neural network is a known concept (see e.g. [8], thanks to M. Kiermayer for the hint), a ready-made package equivalent to xgboost is not readily available, to the best of our knowledge. Therefore, powerful models may remain unused only because they are hard to implement with the available packages.

Interpretability of the results remained a problem throughout. We used feature importance as a guideline and applied error analysis to improve our results. A straightforward way of interpreting the model is not available and is perhaps just not possible, given the way ML algorithms are constructed. This remains an open challenge for the future.

## References

[1] https://www.kaggle.com/c/actuarial-loss-estimation

[2] https://cloud.google.com/natural-language/docs/basics

[3] https://www.nltk.org

[4] https://spacy.io

[5] https://github.com/dmlc/xgboost/tree/master/src

[6] https://stats.stackexchange.com/questions/484555

[7] https://xgboost.readthedocs.io/en/latest/parameter.html

[8] https://arxiv.org/abs/2002.07971v2