Tryg Machine Learning

Simon Bruun-Simonsen
simbr@itu.dk

Nikolaj Pahus Pedersen
nppe@itu.dk

Michal Grega
migre@itu.dk

22/12/25

Course: Machine Learning
BSMALEA1KU

1 Introduction

In this report, we are exploring and discussing different ways of applying machine learning methods to a well-known problem in the insurance business- estimating risk. We are given a publicly available dataset from a large nordic insurance company, Tryg. It is an ambiguous task that could be approached in different ways and we will try to dive into some of them, specifically, we will produce different regression models.

2 Target variables and evaluation metrics

We take *ClaimNb* [2] as the target variable and define the claim risk as

$$E[\text{ClaimNb} \mid \mathbf{X}], \quad \mathbf{X} = (X_1, \ldots, X_p),$$

where p denotes the number of features.

We estimate claims using regression models. Model performance is evaluated using the mean squared error (MSE) as the loss function, and the R2 metric to assess the proportion of variance in the target variable explained by the individual models.

3 Data Cleaning

In order to get the most meaningful data, given our dataset, firstly we clean the dataset from meaningless or illogical information that it contains. The first column removed from the dataset is IDPol. This column contains a unique identifier for each policy, which does not carry any meaningful information for analysis or modeling. Since it cannot contribute to understanding patterns or predicting outcomes, it is safe to drop it from the dataset.
By exploring the dataset in Python, we observed that the exposure variable occasionally exceeds its intended range of 1.0. These out-of-range values are removed from the dataset, as they could adversely affect model performance. We also examined the Area and Density variables. According to the project description [2], they appear to describe the same underlying concept, with Area labeled categorically and Density as a continuous numerical variable. Upon inspection, we confirmed that as Density increases, Area also increases. To retain the more informative representation, we removed the Area column, keeping Density for modeling purposes. There also being alot of categorical values also poses a problem, when modelling later on. So we one-hot-encoded these features so that the models could read them aswell.

4 Exploratory Data Analysis

Moving onto the exploratory data analysis, we examined the distribution of claim counts in the training data. As seen in table 1 we find that approximately 95% of the observations have zero claims, while the remaining 5% have one or more claims. This highlights a strong zero inflation in the target variable, which is an important characteristic to keep in mind for subsequent feature exploration and modeling.
Since 95% of our observations have zero claims and the remaining 5% have one or more claims, we examined how the feature distributions differ across these two groups. Specifically, we wanted to assess whether policies with at least one claim exhibit different characteristics compared to those with no claims. These plots can be seen in the file DataAnalysis.ipynb.

Claims	**%**
0	94.981678
1	4.742933
2	0.261167
3	0.012375
4	0.000739
5	0.000369
6	0.000185
9	0.000185
11	0.000369

Table 1: Distribution of claims

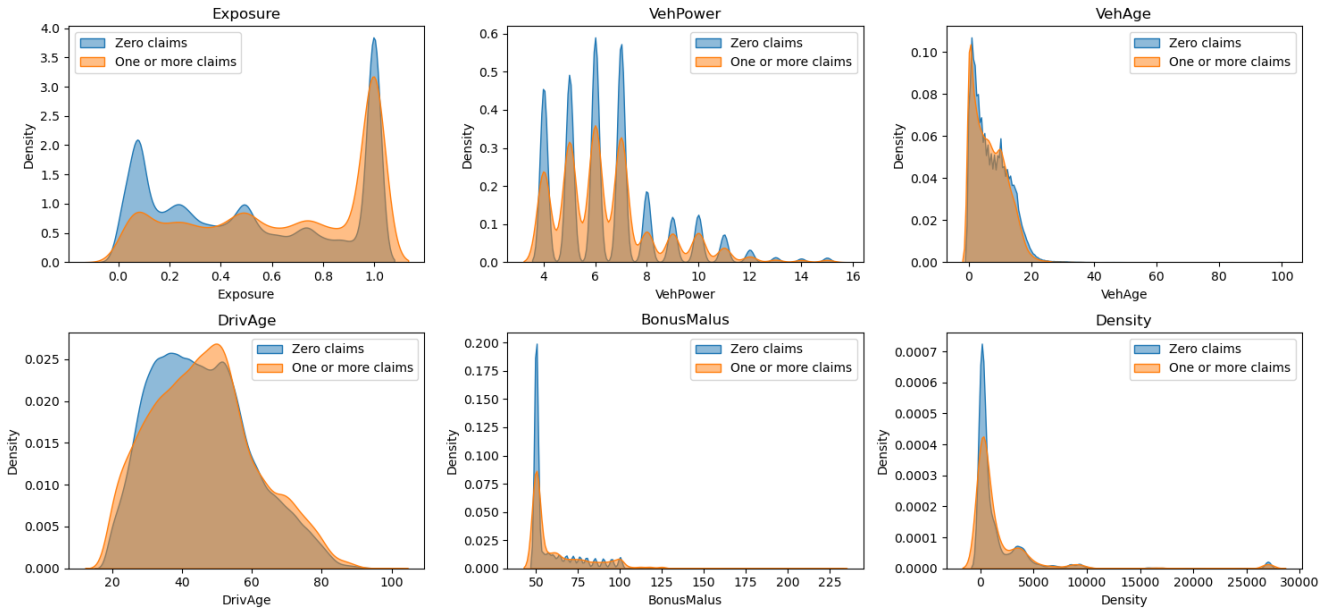

Figure 1: Distribution of features

From the density plots, we observed substantial overlap in the distributions of all numerical features between the zero-claim and one-or-more-claim groups. This overlap suggests that there are no clear feature values that can easily distinguish between the two groups. Consequently, predicting the expected number of claims based solely on individual feature values may be challenging in our regression task, and effective modeling will likely rely on combining multiple features and capturing more complex relationships.

5 Feature Extraction and Selection

5.1 Feature Selection

Our feature selection was guided by our version of the decision tree regressor, which would deem what features to keep and which not to. These features would be used for all the other models. The reason for this is that we wanted to minimize the amount of features needed to make a simpler model. If we could get a model that performed better or similarly with a couple of features rather than all the features then the alternative with fewer features would be better. Our method of measuring features importance was via permutation importance. This measurement looks at the contribution of each feature by shuffling a features data randomly and seeing it's effect on the model's performance. Here is the formula for computing a single feature importantance [9]:

$$i_j = s - 1/K \sum_{k=1}^{K} s_{k,j}$$

- j: Is the feature/column
- K: Is the amount of repetitions. A repetition is one shuffling of the that features data, so the data then becomes corrupted.
- s: Is the reference score on the uncorrupted data
- $s_{k,j}$: Is the score of model on corrupted data for repetition k

- i_j: The importance score for j'th feature

The number of repetitions was 10 to make sure of some variation, while not making it too computationally heavy. The chosen error metric was MSE. Eventually, this process indicated that BonusMalus, Exposure, Vehicle Age, Vehicle Brand 12 and Drivers Age are among the most important predictors contributing to the model's performance.
Keep in mind that we use stratified K-fold validation when testing the performance to get reliable results. I did this by dividing the data in two classes, where one was claims number above zero and the other was claim number equaling zero. This way I could make sure that both classes appeared in each fold.
In our next step we choose to shrink the amount of features to avoid overfitting and making the future models simpler. All the features having an importance score above 0 we kept and then iterated removing the least important feature each time, calculating the models performance and then doing the same until we have results from all the different models. The result is the combination of the first eight most important features, which gives the best results with an R^2 score of 0.0319. A better performance than the model with all features used, which got an R^2 of 0.0304, suggesting that it contains noise.
This mean that the 8 base features are BonusMalus, Exposure, Vehicle Age, Vehicle Brand B12, Drivers Age, Vehicle Power, Vehicle Gas Regular and Density.

5.2 Feature Engineering

With our now 8 base features chosen we wanted to do some feature engineering to see if some combination could explain more variance. We tried every possible combination of two features by either adding, multiplying or dividing. None of the combinations of features improved the model, despite pairplots showing some vaguely promising non-linear relationships between the features. Therefore, we moved on making the models with only the original 8 features, without any engineered ones.

The final feature dataset contains BonusMalus, Exposure, Vehicle Age, Vehicle Brand B12, Drivers Age, Vehicle Power, Vehicle Gas Regular and Density.

6 Principal Component Analysis

As observed in the EDA, the data are zero-inflated and show complex, non-linear relationships, making interpretation challenging. To explore potential structure, we apply PCA to the six numerical features. Four components are required to explain 90% of the variance, indicating that dimensionality reduction is possible but limited due to the small initial feature set.(Figure 2).

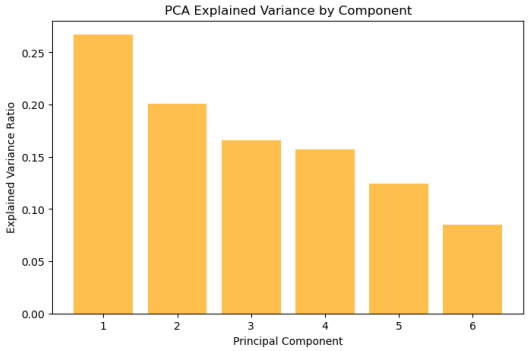

Figure 2: Component explainability

Projecting the data onto the first two principal components, which together explain 45% of the variance,

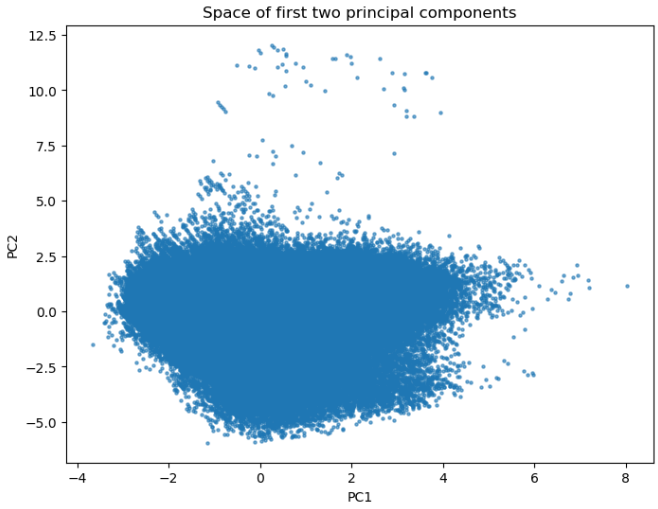

Figure 3: Two Principal components space

reveals a largely homogeneous cloud with a few outliers (Figure 3). The absence of clear separation suggests that the numerical features do not exhibit strong low-dimensional structure or clustering, reinforcing the presence of overlapping risk profiles rather than distinct groups.

7 Clustering

For clustering, we chose to use k-means. Although k-means may not be the optimal clustering method due to overlap in the feature space of individual claims, it is the most intuitive approach. Moreover, given the large number of observations (approximately 500,000), hierarchical clustering would be too computationally expensive.
Before fitting k-means to the full dataset, we evaluated the appropriate number of clusters by plotting the within-cluster sum of squares (also known as inertia)[8] and the silhouette score[6], which were computed on random samples of the data to reduce computational cost.
When plotting inertia, we look for an "elbow," where the rate of decrease sharply changes. The elbow in our analysis appears to occur somewhere between 5 and 10 clusters, suggesting that a value in this range could provide a reasonable trade-off between model complexity and fit.

However, the silhouette analysis indicates that cluster quality for values between 5 and 10 is relatively poor compared to 2 and 4 clusters. Based on this combined evidence, we chose k=4 for the final k-means model, which was then fit on numerical features. When visualizing the clusters against the numerical features, we observe substantial overlap among clusters in most features. The exception is the density feature (highlighted in green), where high-density regions form relatively well-separated clusters compared to the others.

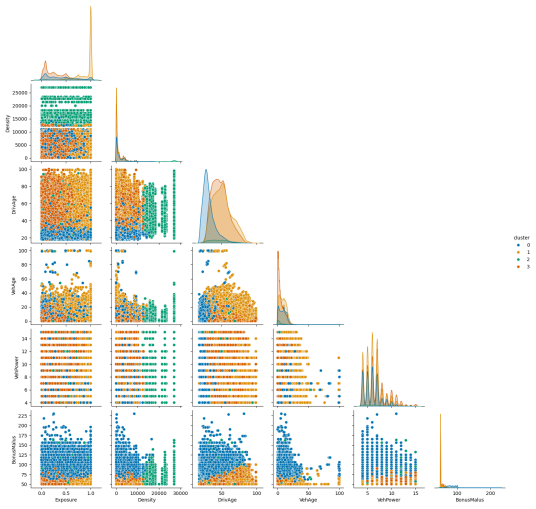

Figure 4: Clustering pairplot

8 Implementations

We will go into details about how we implemented our regression models, how and why we chose our hyperparameters and how we assert the corretness of our models.

8.1 M1

Decision Tree Regressor (DTR)

8.1.1 Own implementation

First we called our model for CDTR (Custom Decision Tree Regressor). In making our own implementation of the decision tree regressor we wanted to make a general class that has all the necessary functions to do evaluation, so that we could perform fit and predict. We choose only two parameters (max_depth and min_samples_leafs) as not to over complicate the model, but still being able tune some parameters too avoid overfitting and underfitting.
We utilize a build_tree function as a recursive function to be able to construct the tree with all it's nodes. [1]. To make the our implementation a little faster we used another way of calculating RSS:

$$\text{RSS} = \sum_{i=1}^{n} y_i^2 - \frac{(\sum_{i=1}^{n} y_i)^2}{n}$$

So rather than calculating each RSS for each region for each data point each time, we could just store $\sum_{i=1}^{n} y_i^2$ in a variable aswell with $\sum_{i=1}^{n} y_i^2$.

When finding the optimal parameters we looked at traning vs test error to find where we could get the best test score without overfitting or underfitting. This can be seen the file FeatureSelectionAndOptimizingDTR's.ipynb and also the use of stratifiedKfold for both our implementation and sklearns. The analysis resulted in a max depth of 8 and minimum samples in leaf is 40. This resulted in an R^2 of 0.0322.

8.1.2 Library implementation

For the library implementation we used scikit-learn implementation of the decision tree regressor. We ran the same parameters for the sklearn implementation as we did for our own to make the models comparable. Though using these parameters for this model is still very reasonable as the difference in the training vs test error was close to none. The resulting R^2 score was 0.03828.

8.1.3 Comparison

When using the final test data on the M1 using both our own implementation and sklearn we got the following results seen in table 2 with the same hyperparameters (Max depth = 8 and minimum samples in leaf = 40). The results are fairly close though the sklearn implementation is doing slightly better. It's 1.189% better.

	Own	**sklearn**
R^2 **score**	0.03282	0.03319

Table 2: R^2 score for our own implmentation and sklearn implementation

8.2 M2

Feed-forward neural network (FFNN)
We implemented M2 from scratch using Python's NumPy library, utilizing linear algebra principles such as matrix operations and vectorization for efficiency. Our implementation strategy follows the principle of over-parametrization, also known as "stretch pants"[7]. Rather than to manually engineer and optimize for the exact number of hidden layers and neurons required to fit the data perfectly, we initialized a wide and deep network. The input layer size is defined by the amount of features we use and the hidden layers are drastically larger than what we theoretically need. Implementing the backward pass, we used the reverse feature from python, which was provided by LLM.[3]

To prevent overfitting, we employ early stopping during training[7]. Model performance is monitored on a validation set using a patience criterion. If the validation loss does not improve for a specified number of epochs, training is stopped and the model parameters corresponding to the lowest validation loss are restored. This strategy aims to identify the point of optimal generalization rather than minimizing training loss indefinitely. Since the loss landscape is non-convex, convergence to a global minimum is not guaranteed. Instead, early stopping selects a solution that achieves strong validation performance before overfitting occurs.
For optimization we use mini batch gradient descent to optimize the weight and biases. This approach balances the computational efficiency of stochastic gradient descent with the stability of batch gradient descent. To mitigate the risk of dying neurons, we use the Leaky ReLU activation function[7]. Additionally, HE initialization is applied to the network weights to improve training stability by maintaining appropriate variance in activations and gradients during the initial stages of training[7].

8.2.1 Hyperparameters

The neural network model contains a considerable number of hyperparameters to tune, including the network architecture, number of training epochs, learning rate, mini-batch size, early stopping patience, and the regularization parameter λ [7]. Due to time constraints and the scope of the project, we focused primarily on tuning the architecture, number of epochs, learning rate, and mini-batch size. 5 k-fold cross-validation was applied to the latter three hyperparameters, after which a grid search was conducted that also included several alternative network architectures. This helped us navigate this complex optimization landscape, as it performed a 3-fold cross validation over 384 candidates, resulting in 1152 fits. The grid search was conducted on a previous dataset using a feature set that was initially believed to be the most explanatory and to yield the best performance; however, subsequent feature importance analysis led to modifications of this feature set. Due to time constraints, the grid search was not repeated. This was deemed acceptable, as the original and revised feature sets are highly similar and produce comparable training and validation performance when using the same hyperparameters as can be seen in the M2.ipynb. After getting the best combinations of hyperparamters, we trained our model on final_train_data.csv and tested on processed_test_data.csv, which resulted in R2 of 0.030 and MSE of 0.058. To replicate this use the following hyperparameters in train_batch method mini_batch_size=42,epochs=175, learning_rate=0.009, patience=20 and define the regressor with hidden _layers=[100,50,20],seed=42

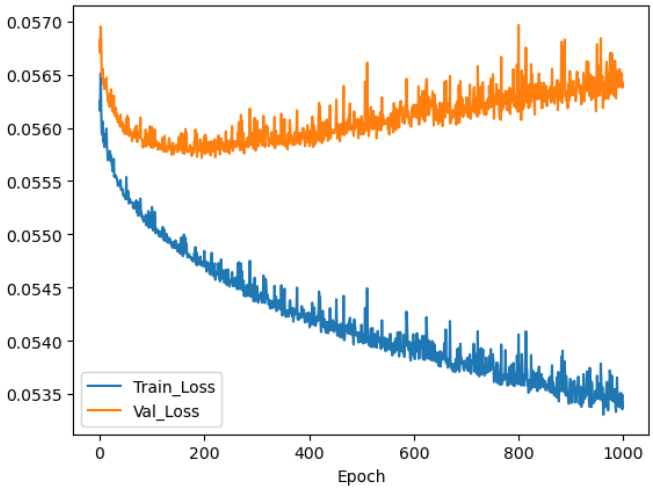

Figure 5: MSE Loss over Epoch

8.2.2 Correctness

To assert the correctness of our M2 implementation, we developed a reference model using the Keras library. The reference model employs the same design choices, including He initialization, Leaky ReLU activation functions, and early stopping, allowing for a direct and fair comparison. Both models were trained on the same dataset to evaluate whether they exhibit comparable learning behavior and performance. The loss curves for both implementations show highly similar trends. Although the Keras model produces a smoother training loss curve—likely due to differences in numerical optimization and internal implementation—the overall behavior is consistent with our model. In particular, both models achieve comparable mean squared error and similar R^2scores, indicating equivalent predictive performance. To further validate the implementation, we examined the models' behavior under overfitting conditions. When trained for a large number of epochs or on small subsets of the training data, both models exhibit clear signs of overfitting: the training loss continues to decrease while the validation loss begins to increase. This behavior is illustrated in Figure 5, where the divergence between training and validation loss occurs after a certain number of epochs. This confirms that the model transitions from learning general patterns to memorizing the training data.
Additional experiments conducted on reduced dataset sizes demonstrate that our implementation can overfit small samples when trained for many epochs, consistent with the expected behavior of neural networks. Overall, the close agreement between our implementation and the Keras reference model provides confidence in the correctness and stability of our training procedure.

8.3 M3

Random Forest Regressor (RFR)
For the third model in our project we choose to go with a Random Forest Regressor. We would like to get some better performance metrics as from our decision tree in M1, therefore we choose this ensemble method, that aggregates the predictions from all the decision trees, only that each tree is trained on a random set of features to avoid the most important features doing all the decisions [4]. We again take the advantage of the scikit-learn implementation of a RFR. Here we need to think about the running time of our code as well as in M2, since a lot of computations are taking place in the cross-validation processes. The first thing that we do is cross-validation on a selected set of hyperparameters. We run this computationally expensive code on only 50 decision trees and 6 of the base features, as anything more would result in excessively long running times for very little error improvements. Based on the ordinary

regression error metrics, we choose the hyperparameters with the best rates.

Validation

Afterwards, we check the model with these exact configurations we found and make the predictions with both training and validation set. Findings are this time a little "disturbing", as we notice a significant difference in MSE as well as a double higher R^2 score for the training set as opposed to the validation set. The way we tackle this issue, is again, by doing cross-validation. Only that now we check the hyperparameter configurations for both training and validation splits. When multiple hyperparameter configurations achieved similar validation R^2, the configuration with the smaller train–validation gap was selected *(min_samples_leaf=12, max_depth=8)*, as it indicates more stable generalization and lower variance, which we then expect to be more robust and perform better for the real test data. This was done by plotting heatmaps on the two splits that iterated through different values for hyperpamareters that were chosen arbitrarily, but in reasonable ranges.

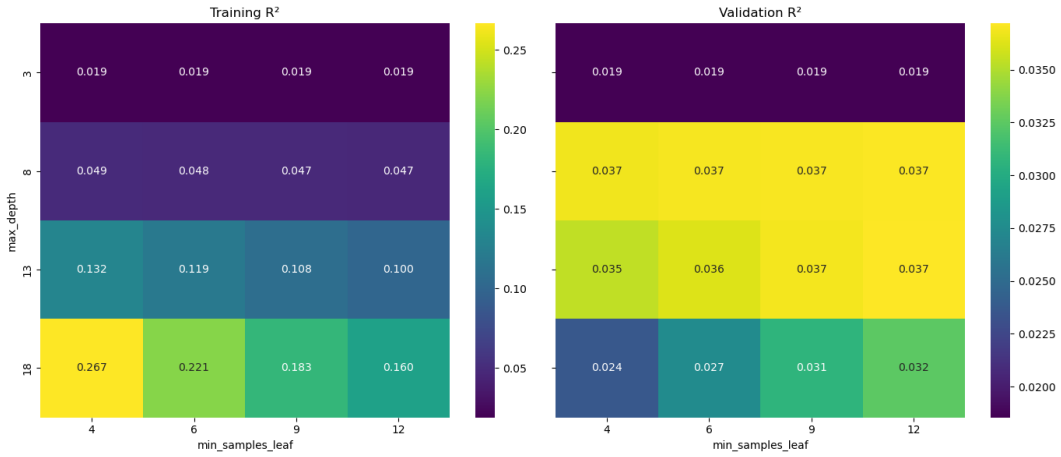

Figure 6: Hyperparameters selection for RFR

Finally, we have the best performing hyperparameters which are also overfitting-proof. We can therefore

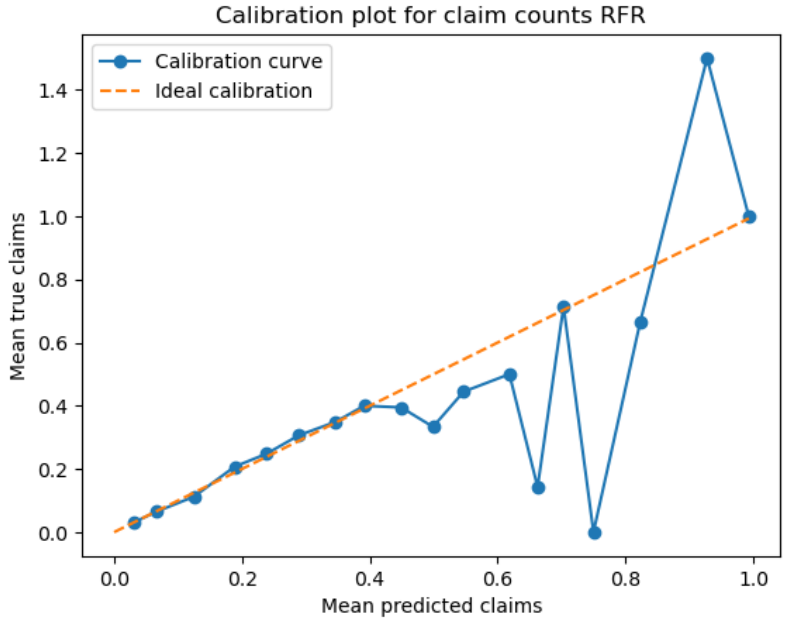

Figure 7: Calibration plot of RFR

make the final model with all the features included, as well as much more decision tree estimators and run it on standard 5 different folds to get a good estimate range on what error we can expect.

On the calibration plot on Figure 7, we can see that our models mean predictions were following the true mean somewhere until 0.4 mean claims, afterwards the calibration curve was way out of proportion, mainly because the amount of predictions that were above around 0.4 was very low.

As the very last step, the trained model was finally tested on the yet unseen test data, and produced a single result of :

	R^2	**MSE**
test scores	0.0386	0.0579

9 Discussion & Evaluation

Feeding our models the right data, as well as fine-tuning their hyper-parameters is what cost us a considerable amount of effort. We put a lot of thought into following the machine learning principles and practices.

While being aware of adverse effects of using the same set of features on DTR as well as FFNN because of how fundamentally different they exploit the structure in data [5], for the sake of comparison of all the models on the same feature dataset, we have not made any changes in the features across all three models.

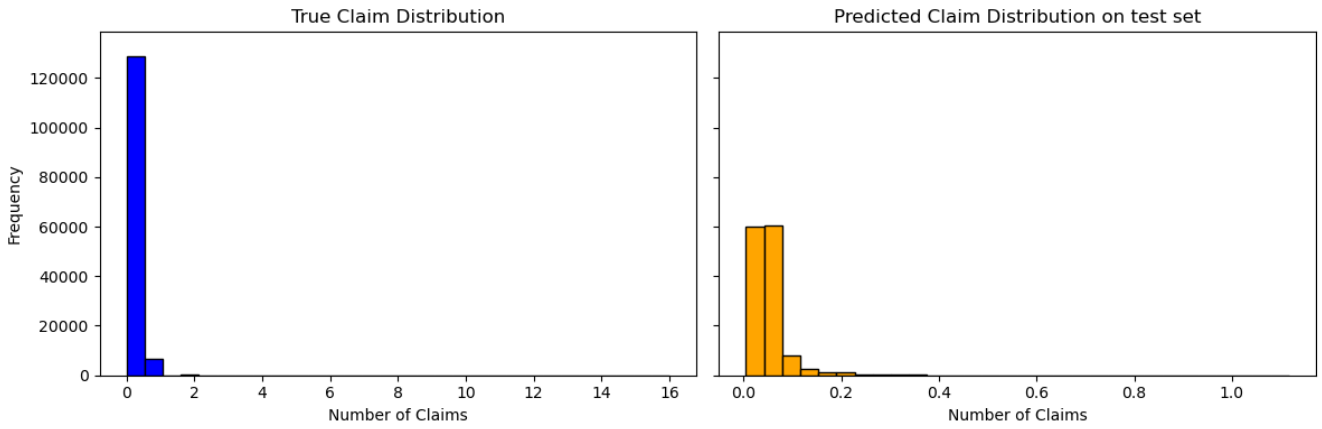

Figure 8: Test data claims distribution vs RFR predictions

Figure 8 shows what our predictions look like for the test set using RFR. We note the majority of predictions are around the mean true claims, which is around 0.05. That is the result of the heavily skewed dataset towards 0 claims.

All three the models are trained on the same data their results are still different. A reason for the RandomForest to perform better may be because it is an ensemble method and is good at generalizing and has therefore less variance in the output. This tendency is good since decision tree tend to overfit too much and there having too much variance, which is probably the case for decision tree regressor. The neural network might not have enough neurons and hidden layers to generalize from and the hyperparameters may not be optimally tuned, given the size and complexity of the search space of different combinations. While for now, the Random forest outperforms the M1 and M2 models, perfoming a gridsearch over a broader range of hyperparameters for M2 could potentially improve its performance.

10 Conclusion

All three models showed that while there is a lot of freedom in whether it be feature selection, engineering or tweaking the hyperparameters, there are certain limitations to what we can achieve. And that heavily depends on the explainability of the data that is used for training the models. Insurance claims risk prediction belongs to one of those tasks where even a small improvement in correctness is valuable. By applying our knowledge of machine learning practices, we managed to make improvements to our models, however, the overall performance of our models fell short of +4% of the R^2 score.

References

[1] Chatgpt. Used AI for the implementation of making the outline for the best split and build tree functions.

[2] Ml proposal.

[3] Gemini, 2025. Used large language model assistance for backpropagation implementation, including reversed Python feature in the loop. Additional comments available in the notebook code.

[4] Gareth James et. al. *An Introduction to Statistical Learning with Applications in python.* Springer, 1st edition, 2023.

[5] Trevor Hastie et. al. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction.* Springer, 2nd edition, 2009.

[6] geeksforgeeks. Silhouette score. https://www.geeksforgeeks.org/machine-learning/what-is-silhouette-score/, 2025.

[7] Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems.* O'Reilly Media, 2nd edition, 2019.

[8] Scikit-learn. Clustering. https://scikit-learn.org/stable/modules/clustering.html, 2025.

[9] scikit learn. Permutation feature importance. https://scikit-learn.org/stable/modules/permutation_importance.html, 2025.