FRISS Data Scientist case report

1 Introduction

During the week that was afforded to me for this assignment, I was not able to train a well-performing model to classify the data that was provided.
Given the extremely high class imbalance, I ventured to try and find solutions to avoid training naive classifiers, as most basic approaches I tested (which can be found in the `Sklearn_tests` notebook) tended to predict only one of the classes.

2 Metrics

Given the nature of the data, simply relying on accuracy would be misleading, as naive classifiers easily achieved accuracy scores of over 99%. As such, all tested classifiers were evaluated on **Precision**, **Recall**, **F1 score** and **AUROC** in addition to accuracy. With the F1 score calculated on the test-set used as the main evaluation metric.

3 Inbalanced-learn

One of the fist libraries I found to try and alleviate the issues stemming from such an unbalanced dataset was **Imbalanced-Learn**, which provides numerous utilities for over and under-sampling which were tested on several of the approaches. As a general rule, I found that the over-sampling techniques such as SMOTE and ADASYN did not provide satisfactory results. The same can be said for the combined over and under-sampling techniques SMOTETomek and SMOTEENN. Given this, only the random under-sampler provided by the library was used, which simply selects a random sample of the majority class of the dataset of the same size as the minority class. Imbalanced-learn also provides a number of classifiers that aim to avoid the pitfalls generally common in highly imbalanced classification problems. These classifiers were trained and evaluated (the results can be found in the `Imbalanced_learn` notebook) leading to generally very poor results.

4 Python Outlier Detection

Considering the poor performance of the classification methods contained in the `Sklearn_tests` and `Imbalanced_learn` notebooks, I explored the possibility of re-framing the problem from a classification problem to an outlier detection one. For this purpose I found **PyOD**, a library containing implementations of numerous outlier detection techniques. The tests that were conducted can be found in the `PyOD_tests` notebook. Once again, these techniques did not provide satisfactory results.

5 PyTorch Classifier

I dedicated the largest amount of effort to a simple multi-layer perceptron approach which can be found in the `claim_classifier.py` file and the `PyTorch_Classifier` notebook.
The network used is composed of three linear layers with 23, 32 and 16 neurons respectively. The activation function used between layers was ReLU. The model was trained using the training data under-sampled using Imbalanced-learn's random under-sampler and validated using a tenth of the under-sampled training data. After each epoch, the above-mentioned metrics were calculated on the validation set and the epoch's accuracy was compared to previous epoch's scores. After a given number of epochs showing no improvement on accuracy, training would stop and the best-performing model would be loaded. Note that while F1 score was the main performance metric used to evaluate performance on the test-set, as the training and validation sets contained an equal number of entries relative to each class, accuracy was preferred to determine when to stop training the model.
While validation metrics were usually mediocre at best, even these low results could not be transferred to the test-set, in which performance was abysmal. For instance, the model which was used to determine the predictions contained in the Docker API, which can be found in the GitHub repo in the file `least_bad_model.mdl` delivered the following performace metrics:

Metric	Validation	Test
Accuracy	0.6000	0.7052
Precision	0.4308	0.4706
Recall	0.6512	0.0109
F1	0.5185	0.0213
AUROC	0.6454	0.5902