Fraud detection with cost-sensitive machine learning

The concept of example-dependent cost-sensitive classification algorithms

Roman Moser Follow
Mar 29, 2019 · 11 min read ★

In traditional two-class classification problems we aim to minimize misclassifications and measure the model performance with metrics like Accuracy, F-score or the AUC-ROC Curve. In certain problems, however, it is for the best to allow more misclassifications at the benefit of lower total costs. If costs associated with misclassifications vary among samples, we should apply an example-dependent cost-sensitive learning approach. But let's start from the beginning…

In this article, I will be explaining the concept of example dependent cost-sensitive

set. Please note that I chose the models for this task for the purpose of illustrating the concept rather than achieving optimal prediction results. Snippets of the code are presented in this article and the full code is available on my GitHub.

What is Cost-Sensitive learning?

Whereas traditional classification models assume that all misclassification errors carry the same cost, cost-sensitive models consider costs that vary by type of classification and across samples.

Let's take a look at the case of credit card transactions. Transactions that are not authorized by the true holder are considered fraudulent (usually a very small portion of all transactions). A credit card fraud detection system should automatically identify and block such fraudulent transactions and at the same time avoid blocking legitimate transactions.

What are the costs associated with each type of classification? Let's assume the following scenario. If a fraudulent transaction is not recognized by the system, the money is lost and the card holder needs to be reimbursed for the whole transaction amount. If the system labels a transaction as fraudulent, the transaction is blocked. In that case administrative costs occur because the card holder needs to be contacted and the card needs to be replaced (if the transaction was correctly labeled fraudulent) or reactivated (if the transaction was actually legitimate). Let's also make the simplified assumption that the administrative cost are always identical. If the system correctly labels a transaction as legitimate, the transaction is automatically approved and no costs occur. This results in the following costs associated with each prediction scenario:

	Actual Fraud $y_{true} = 1$	**Actual Legitimate** $y_{true} = 0$
Predicted Fraud $y_{pred} = 1$	*True Positive* $cost_{TP} = Admin$	*False Positive* $cost_{FP} = Admin$
Predicted Legitimate	*False Negative*	*True Negative*

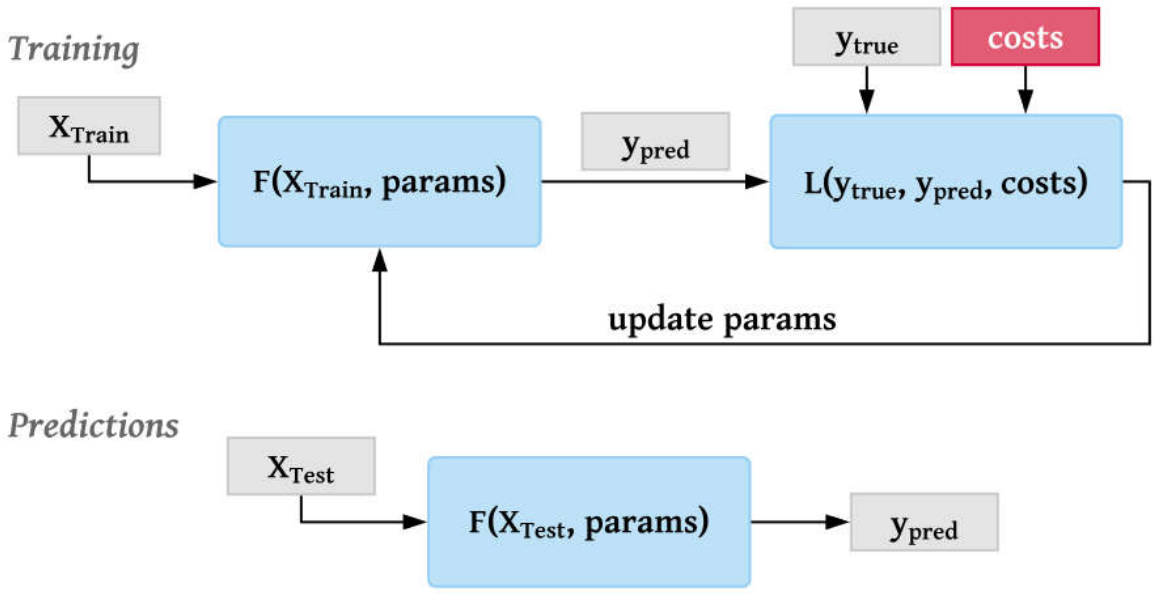
Note that "Positives" are transactions predicted as fraudulent and "Negatives" are transactions predicted as legitimate. "True" and "False" refer to correct and incorrect predictions, respectively.

Because the transaction cost depends on the sample, the cost of a False Negative can be negligibly low (e.g. for a transaction of $0.10), in which case the administrative costs of a positive prediction would outweigh the reimbursement costs, or very high (e.g. for a transaction of $10,000).

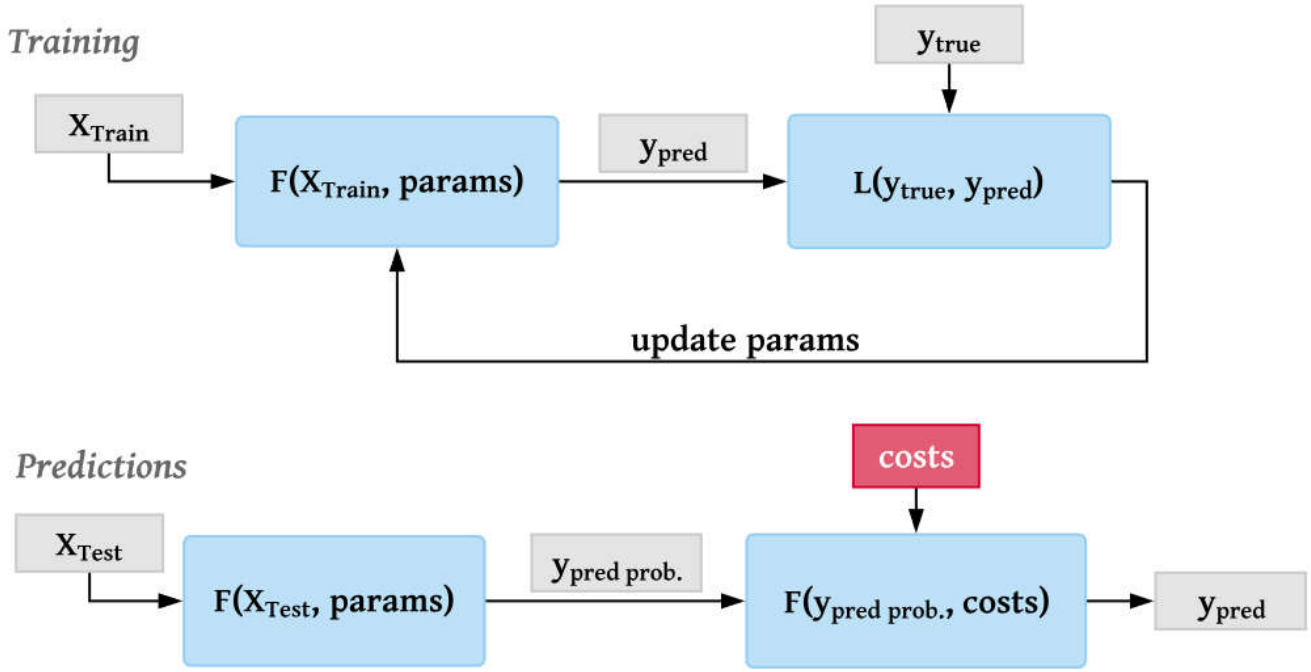
> *The idea behind cost-sensitive learning is to take these example dependent costs into account and make predictions that aim to minimize the overall costs instead of minimizing misclassifications.*

Cost sensitive training vs cost dependent classification

Let's consider two different approaches. The first one is to train a model with a loss function that minimizes the actual costs ($) instead of misclassification errors. In this case we need to provide the loss function with the costs associated with each of the four cases (False Positives, False Negatives, True Positives and True Negatives) so that the model can learn to make optimal predictions accordingly.



The second approach is to train a regular model, but classify each sample when making predictions according to the lowest expected costs. In this case the costs on the training set are not needed. However, this approach only works for models that predict a probability which can then be used to calculate the expected costs.



> *In the following, I will refer to models that use a cost-sensitive loss function as* ***"Cost-sensitive models"*** *and to models that minimize the expected costs when making predictions as* ***"Cost classification models"***

Implementing and evaluating models

For this case study, I used a credit card fraud data set (available on Kaggle) with 284,000 samples and 30 features. The target variable indicates whether a transaction is legitimate (0) or fraudulent (1). The data is highly imbalanced with only 0.17% fraudulent transactions. I trained and evaluated the following five models.

1. **Regular Logistic Regression** (from scikit-learn)
2. **Regular Artificial Neural Network** (built in Keras)
3. **Cost-sensitive Artificial Neural Network** (Keras)

5. **Cost classification Artificial Neural Network**

In practice, artificial neural networks ("ANNs") might not be the first choice for fraud detection. Tree based models such as Random Forests and Gradient Boosting Machines have the advantage of interpretability and often perform better. For the purpose of this illustration I used ANNs because of the relatively straightforward implementation of a cost-sensitive loss function. Also, as I will be showing, a simple ANN delivers quite strong results.

To evaluate the results, I used two different metrics. The first one is the traditional F1-score which weighs precision and recall but does not consider the example dependent cost of misclassifications.

$$F1\ score = \frac{2 * precision * recall}{precision + recall}$$

To evaluate a model's performance in terms of costs, I first calculated the sum of all costs resulting from the predictions based on whether the model predicted a False Positive, False Negative, True Positive or True Negative and the costs associated with each case.

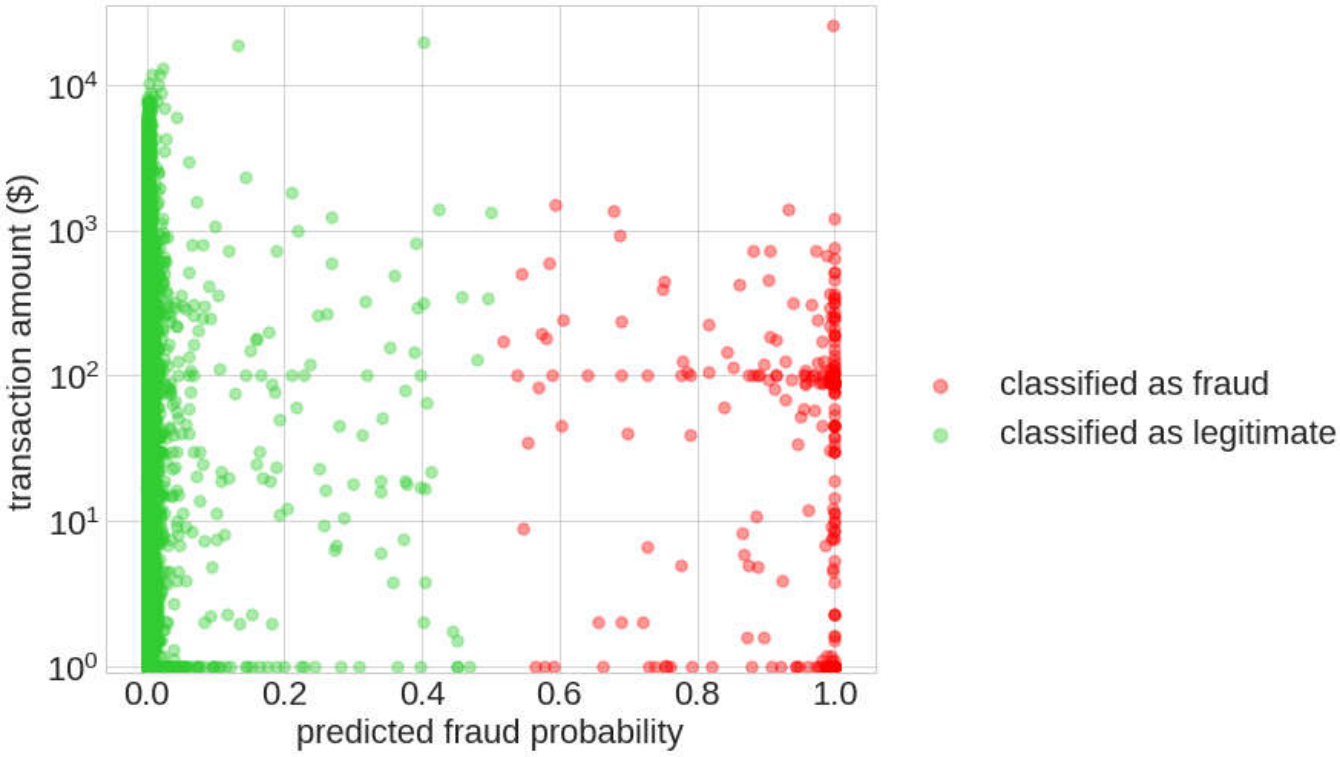
$$costs = \sum_i (c_{FP} * FP_i + c_{FN(i)} * FN_i + c_{TP} * TP_i + c_{TN} * TN_i)$$

I then calculated the sum of the costs that would occur if all cases were predicted negative ("cost_max"), and define the cost savings as the fraction by which the actual predictions reduce the costs.

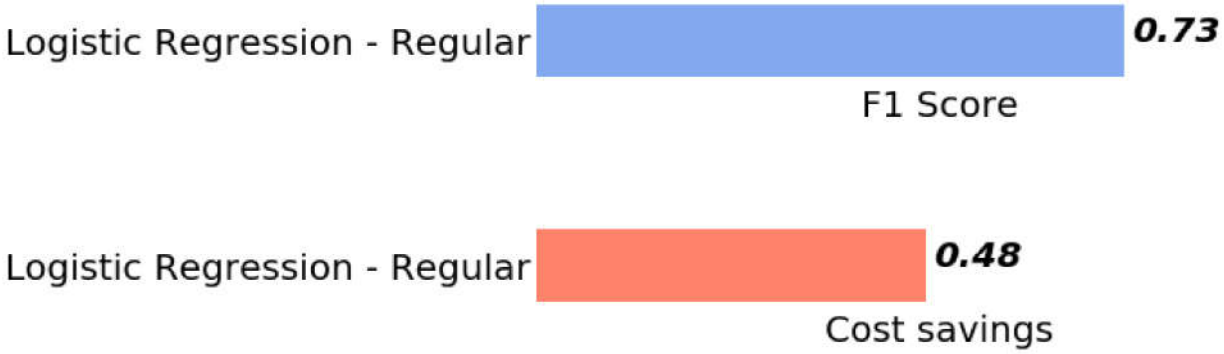
$$Cost\ savings = 1 - \frac{costs}{costs_{max}}$$

To evaluate the models I used 5-fold cross-validation and split the data into five different training (80%) and test sets (20%). The results presented in the subsequent section refer to the average result on the five test sets.

As base model serves a regular Logistic Regression model from the scikit-learn library. The plot below visualizes the distribution between predicted probabilities and transaction amounts. Without cost-sensitive classification there is no visible association between fraud probability and transaction amount.



The logistic regression performs reasonably well with an average test set F1-score of 0.73 and cost savings of 0.48.



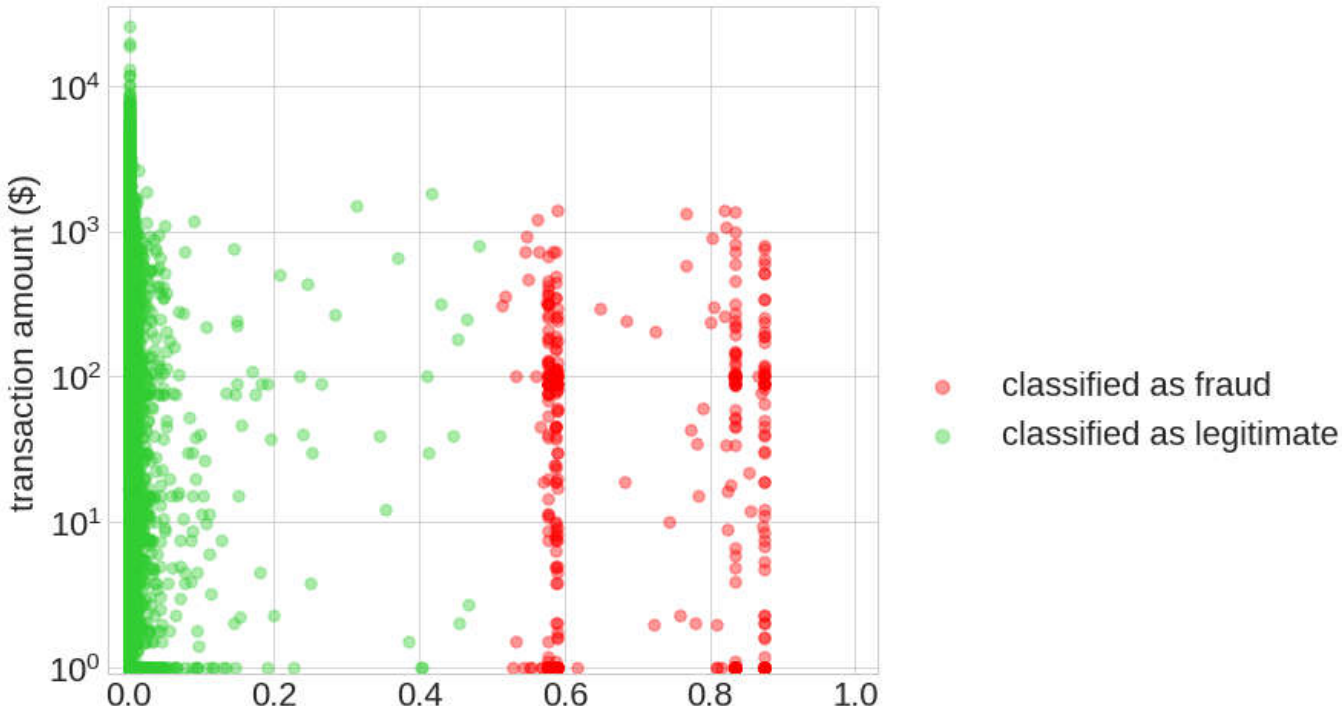
Artificial Neural Network

Next, I built an ANN in Keras with three fully connected layers (50, 25 and 15 neurons) and two dropout layers. I ran the model for two epochs and used a batch size of 50.

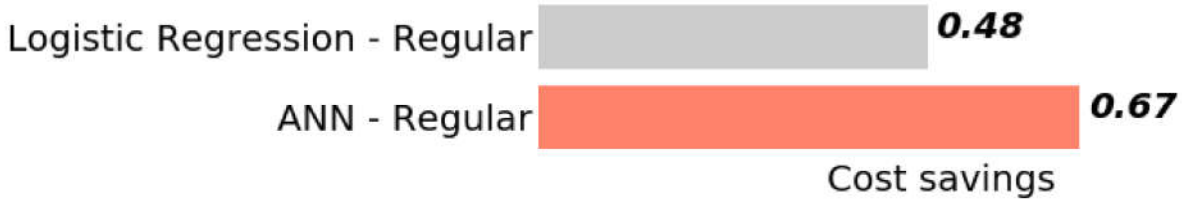
Using the Sequential model API from Keras, the implementation in Python looks like this:

```
from keras.models import Sequential
from keras.layers import Dense, Dropout

def ann(indput_dim, dropout=0.2):
    model = Sequential([
    Dense(units=50, input_dim=indput_dim, activation='relu'),
    Dropout(dropout),
    Dense(units=25, activation='relu'),
    Dropout(dropout),
    Dense(15, activation='relu'),
    Dense(1, activation='sigmoid')])
    return model

clf = ann(indput_dim=X_train.shape[1], dropout=0.2)
clf.compile(optimizer='adam', loss='binary_crossentropy')
clf.fit(X_train, y_train, batch_size=50, epochs=2, verbose=1)
clf.predict(X_test, verbose=1)
```

Below is the distribution of predicted fraud probabilities with the ANN. Similar to the logistic regression model there is no visible relationship between fraud probabilities and transaction amount.



The ANN outperformed the logistic regression model in terms of both, F1-score and cost savings.



Cost-sensitive Artificial Neural Network

Now it gets a bit more interesting. The cost-sensitive ANN is identical to the regular ANN with the difference of a cost-sensitive loss function. Both of the previous models used the logarithmic loss ("binary cross entropy") as loss function:

$$Loss = y_{true} * log(y_{pred}) + (1 - y_{true}) * log(1 - y_{pred})$$

This loss function punishes false negatives and false positives equally. Let's now take a look at a cost-sensitive loss function. Here, all four possible outcomes (False Positives, False Negatives, True Positives and True Negatives) are being considered and each of the outcomes carries a specified cost. The cost-sensitive loss function looks like this:

$$Loss_{CS} = y_{true} * (c_{FN} * log(y_{pred}) + c_{TP} * log(1 - y_{pred})) + (1 - y_{true}) * (c_{FP} * log(1 - y_{pred}) + c_{TN} * log(y_{pred}))$$

Remember from the first section that True Positives and False Positives are being considered as equally expensive (fixed administrative cost for blocking a transaction). The cost for True negatives is $0 (no action) and the cost for False Negatives is the

these four costs, only the cost for false negatives is example-dependent. This has the effect that with a higher transaction amount, the punishment for a not identified fraudulent transaction increases relative to the administrative cost of a positive prediction. The loss function should therefore train a model that is likelier to reject suspicious transaction when the transaction amount is higher. The transaction amounts range anywhere from $0 to $25,691 with a mean of $88 and I assumed a fixed administrative cost of $3.

In Python we define the costs for False Positives, False Negatives, True Positives and True Negatives accordingly. Since the costs for False Negatives are example-dependent they are represented in a vector of length equal to number of samples.

```
cost_FP = 3
cost_FN = data['Amount']
cost_TP = 3
cost_TN = 0
```

Implementing an example dependent loss function in Keras is tricky because Keras does not allow arguments other than y_true and y_pred to be passed to the loss function. Constant variables can be passed to the loss function by wrapping the loss function into another function. However, the costs for False Negatives are example-dependent. I therefore used the trick of adding the costs of False Negatives as digits after the comma to y_true and extracting them inside the custom loss function while rounding y_true to the original integer value. The implementation of the functions to transform y_true and the custom loss function in Keras look like this:

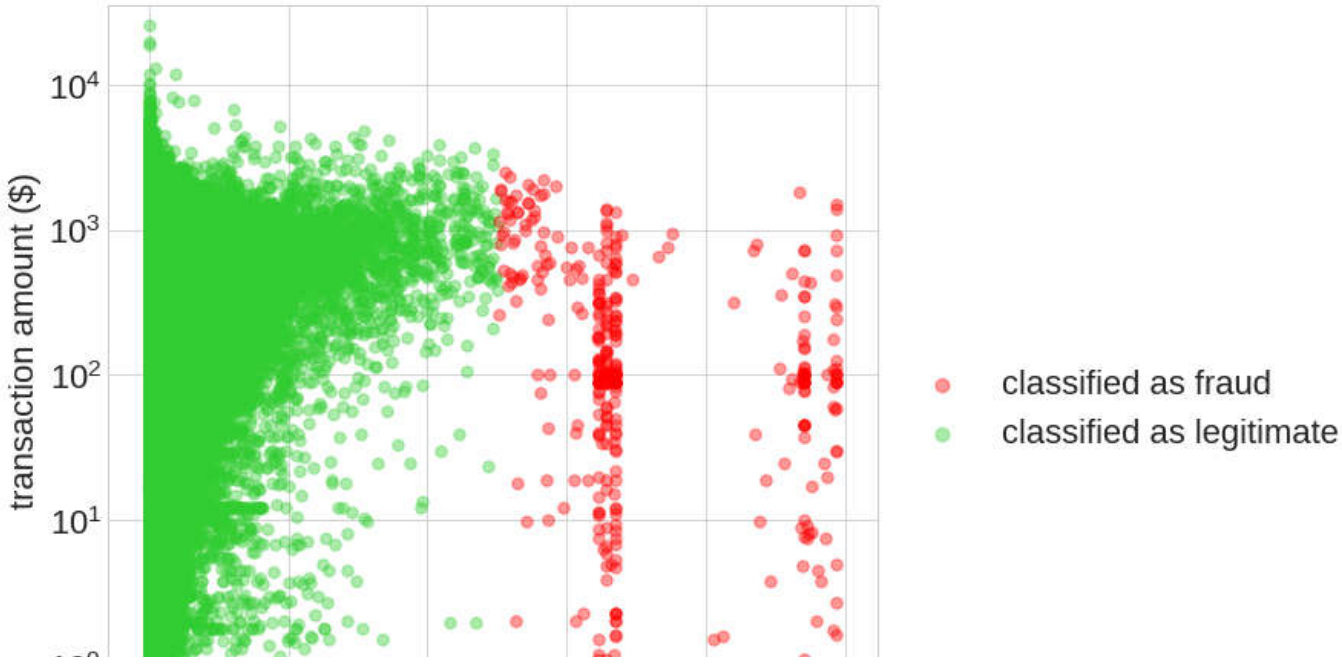
```
import keras.backend as K

def create_y_input(y_train, c_FN):
    y_str = pd.Series(y_train).reset_index(drop=True).\
            apply(lambda x: str(int(x)))
    c_FN_str = pd.Series(c_FN).reset_index(drop=True).\
            apply(lambda x: '0'*(5-len(str(int(x)))) + str(int(x))
    return y_str + '.' + c_FN_str

def custom_loss(c_FP, c_TP, c_TN):
```

```
        c_FN = (y_input - y_true) * 1e5
        cost = y_true * K.log(y_pred) * c_FN +
               y_true * K.log(1 - y_pred) * c_TP) +
               (1 - y_true) * K.log(1 - y_pred) * c_FP +
               (1 - y_true) * K.log(y_pred) * c_TN)
        return - K.mean(cost, axis=-1)
    return loss_function
```

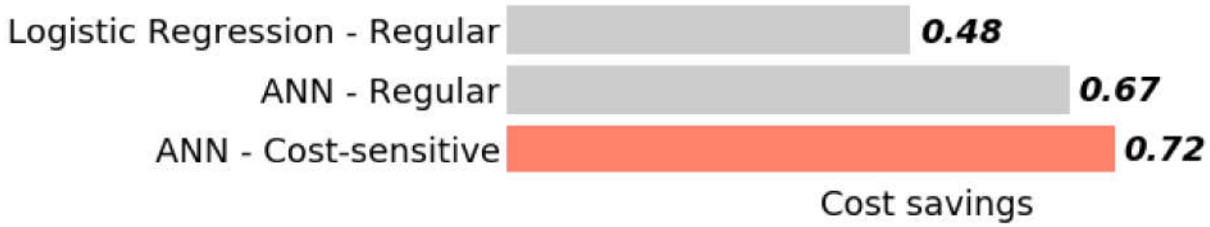
I then called the defined functions to create the y_input vector, train the cost-sensitive ANN and make predictions:

```
y_input = create_y_input(y_train, cost_FN_train).apply(float)
clf = ann(indput_dim=X_train.shape[1], dropout=0.2)
clf.compile(optimizer='adam', loss=custom_loss(cost_FP, cost_TP,
            cost_TN))
clf.fit(X_train, y_input, batch_size=50, epochs=2, verbose=1)
clf.predict(X_test, verbose=1)
```

In the distribution plot below we can see the effect of cost-sensitive learning. With an increasing transaction amount, the general distribution of predictions expands to the right (higher fraud probabilities). Note that in this case, due to the nature of the problem and definition of the loss function, "predicted fraud probability" means "should we identify the transaction as fraudulent?" rather than "is the transaction fraudulent".



The evaluation shows the expected effect of cost-sensitive learning. The cost savings increased by 5% and the F1-score decreased by a similar margin. The consequence of cost-sensitive classification is a higher number of misclassifications at the benefit of lower total misclassification costs.



Cost classification models

As opposed to a cost-sensitive model that trains with a customized loss function, cost classification models calculate the expected costs based on predicted probabilities. The expected costs for a predicting a legitimate and a fraudulent transaction are calculated as follows:

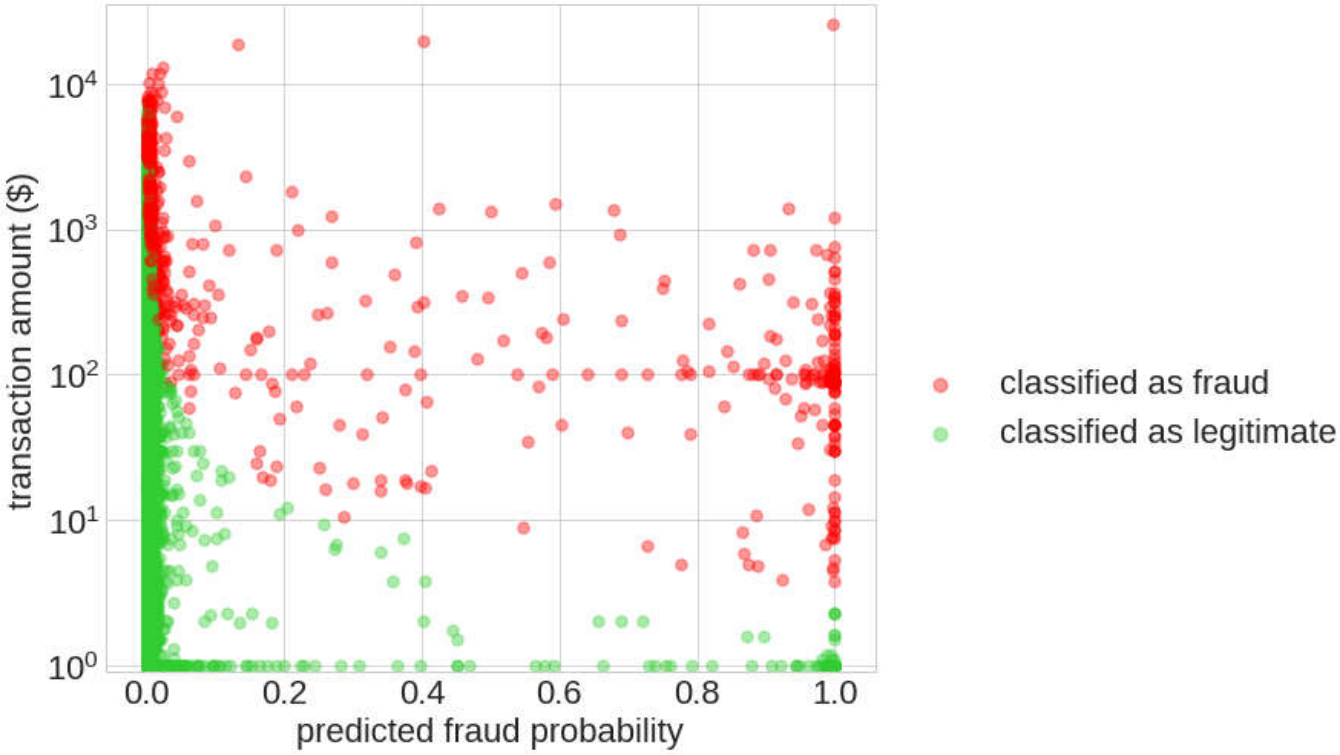
$$E_{classify\ fraud} = c_{TP} * y_{proba} + c_{FP} * (1 - y_{proba})$$

$$E_{classify\ legitimate} = c_{TN} * (1 - y_{proba}) + c_{FN} * y_{proba}$$

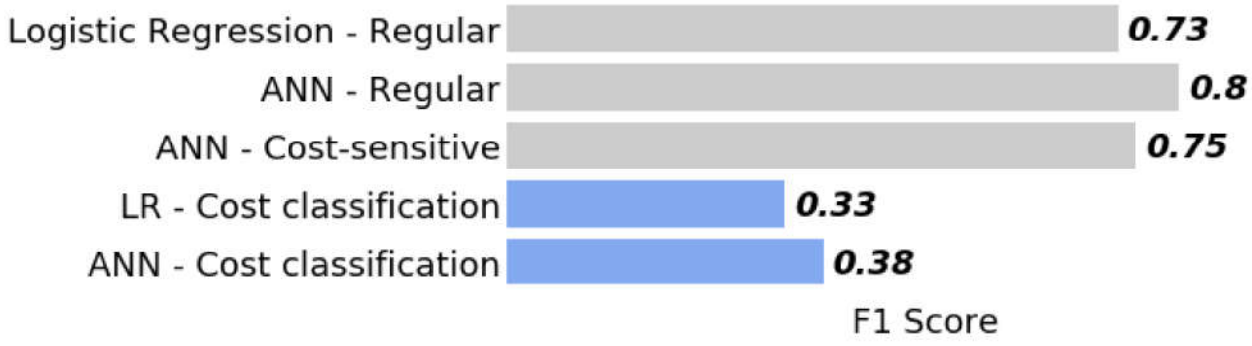
The classifier then chooses whichever prediction is expected to result in lower costs.

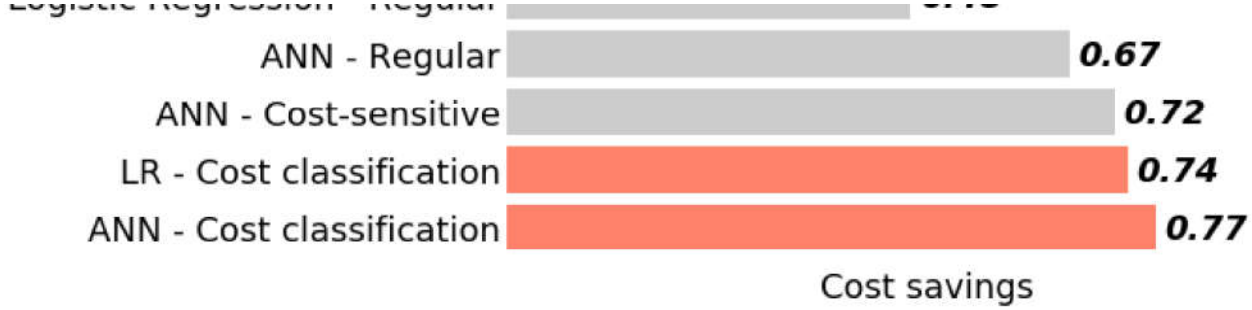
I therefore used the probability prediction results from the regular logistic regression and ANN and reclassified the predictions based on the expected costs. The plot below visualizes the effect of the cost-dependent classification for the example of the logistic regression model. Note that the distribution of predicted probabilities did not change

dependent classification, the model tends to identify transactions with a small fraud probability as fraudulent as the transaction amount increases. On the right side of the plot we see that transactions with a very small amount are predicted as legitimate even as the fraud probabilities approach 1. This is due to the assumption that True Positives carry administrative costs of $3.



Classifying the predictions based on expected costs leads to even better results in terms of cost savings (and significantly worse results in terms of F1-score). While implementing a cost-sensitive loss function for the ANN reduced the costs by 5%, the cost classification ANN was able to reduce costs by 10%.





Conclusion

This article illustrates two fundamentally different approaches for example based cost-sensitive classification on credit card fraud prediction. While cost-sensitive training models require a custom loss function, cost classification models only require the probabilities for each class and the costs associated with each outcome to classify a transaction. In my sample case cost classification models achieved slightly better cost savings at the expense of a high number of misclassifications. Additionally, a cost-classification model is easier to implement as it does not require a custom loss function for training. However, the cost classification method is only applicable for models that predict probabilities, which a logistic regression and ANN conveniently do. Tree based models, adopted more widely for fraud detection, however, generally separate predictions directly into classes, making the cost classification approach infeasible. A cost-sensitive approach for tree based models is, while conceptually similar to the one presented in this article, more complicated in implementation. If you are interested in this topic I would suggest to take a look at the paper referenced below.

. . .

Thanks for reading this article. Please feel free to comment or ask questions via the comment section below or to connect with me on LinkedIn.

The code created for this illustration can be accessed on my GitHub

The credit card fraud data set is available on Kaggle

If you are interested in learning more about cost-sensitive learning with tree based

the costcla GitHub repository

Machine Learning Fraud Fraud Detection Neural Networks Credit Cards