Bachelor Thesis

„ An approach to Out-of-Model feature explanations in the application to credit fraud detection “

Supervisor:
Prof. Dr. Joerg Schaefer
OOP, Distr. Systems and DBs, Machine Learning
Frankfurt University of Applied Sciences

Co-Supervisor:
Fatima Butt, M.Sc. Computer Science, Doctorate-Candidate

Supervisor at Commerzbank:
Dr. Alexander Fischer

Bachelor Thesis Author:
Rainer Gogel
B.Sc. Computer Science - Student

Summer Semester 2021

Monday, September 6, 2021

Statutory declaration

I hereby declare that I am the sole author of this bachelor thesis and the program code presented therein. Thoughts taken directly or indirectly from external sources are marked as such. The work has not yet been submitted to any other examining authority and has not yet been published.

Use of customer-related data

Bank data in general and fraud data and the investigation process of Commerzbank AG in particular, need to be kept strictly confidential. This requires that the data and the feature names used in this thesis are anonymized to the extent that no conclusion about the origin and the type of the variables can be drawn. As such, all customer-related data used in this thesis is anonymized and transformed. It is therefore not possible to draw any conclusions about the Commerzbank AG portfolio.

Acknowledgment

I hereby would like to express my gratitude towards my supervisor Dr. Alexander Fischer at Commerzbank AG. Dr. Fischer spent a significant part of his time and energy to guide me in the field of data science and to impart the knowledge to me this thesis builds on. He did this with great dedication, perseverance, patience, and a lot of understanding.

I would also like to thank my supervisor at Frankfurt University of Applied Sciences Prof. Dr. Joerg Schaefer for his willingness to supervise this thesis and for his support and guidance.

Table of Contents

Executive Summary

The topic of this thesis is to derive insight from features not existent in a trained Machine Learning ("ML") model. The idea is to transfer information from an original ML model A ("Original Model") and its feature set to a new ML model B ("Transfer Model") and a new feature set by using the predictions of the Original Model as input/dependent variable in the Transfer Model. The feature set of the new Transfer Model is not accessible for the training of the Original Model but might reveal information in the newly trained Transfer Model through the application of explanation techniques such as Shapley values. I conclude that the newly trained Transfer Model with its new "out-of-model" features is stable and accept the hypothesis that it is possible to explain the predictions of the Original Model with the new features of the Transfer Model.

1. Introduction

A. Commerzbank AG and its Big Data & Advanced Analytics division

From March to April 2020, I worked as an intern at Commerzbank AG in the "Big Data & Advanced Analytics" division ("BDAA") and since have been working as a working student there. Commerzbank AG is the fourth largest bank in Germany[1] and has around 30,000 corporate clients and around 11 million private and business clients.[2] Lending to corporate customers is a core business of Commerzbank AG. There, in very rare cases, loan frauds occur. Loan fraud is defined as the fraudulent use of credit by providing false information in the loan application by the applicant and is aimed at not repaying all or part of the funds received. A typical use case for the responsible BDAA department ("Predictive Analytics") therefore is the analysis of corporate customer data with the help of ML models to detect such loan frauds early.

B. Reasoning for a Transfer Model

The ML model used by the "Predictive Analytics" department to predict corporate fraud scores ("Fraud-Score") is well established and has yielded good results in the past. At the time the model was developed, data was collected for the preceding 10 years to have enough model training data as the number of corporate fraud cases is rare.

Once every month, the ML model calculates an ordinal Fraud-Score for each client. Fraud-Scores above a certain threshold are reported and forwarded to an internal investigation department. The actual forensic analysis and decision on the existence of credit fraud is then carried out there by specially trained investigators, often with the involvement of other units of the bank. Various additional data sources are used during this investigation process, among them payment transaction ("Zahlungsverkehr") data.

To integrate additional data and features into an existing ML model, a common approach would be to develop a new ML model that includes these new features. But there are use cases where this is not possible. The reasons for that may vary. For instance, an existing ML model might be deeply integrated in a structured process to the extent that reengineering this process would be almost impossible from a business perspective. In the case of BDAA, the need to develop a Transfer Model derives from the fact, that the payment transaction data, although important, could not be used directly in the original ML model as its data history was not long enough.

The desire of the investigators is to obtain hints and explanations from this payment transaction data in a more structured, direct, automated, and informative manner. In particular, the investigation department wants to know how the payment transaction data relates to the model's Fraud-Score and which of the payment transaction data features contribute most to it.

The goal of this thesis is to elaborate on the hypothesis that it is possible to perform model explanations on "out-of-model" features (in our case, the features from the payment transaction data). By enabling global views on the most important out-of-model features related to fraud as well as applying local feature analysis for a specific case under investigation, this approach aims at further supporting the investigators during their analysis.

[1] (Wikipedia: Banken in Deutschland, 2021)

[2] (Commerzbank.de, 2021)

C. Original Model and Transfer Model

In this thesis, the attempt to link the predictions of an existing ML model to a new feature set and a newly developed model will be outlined and described. Going forward, I will refer to the existent ML model as the "Original Model" and to the newly yet to be developed ML model as the "Transfer Model". Accordingly, the features used in the existing ML model will be named "Original Features" and the new feature set coming from the payment transaction data will be referred to as "Transfer Features".

The Transfer Model uses the Original Model's predicted Fraud-Scores as the dependent or target vector ("y-vector") and the Transfer Features as explanatory or independent variables ("X-matrix"). The Transfer Model will be developed by using different ML classification algorithms. The modelling process also includes preprocessing steps and feature reduction techniques, but these are not discussed in this thesis.

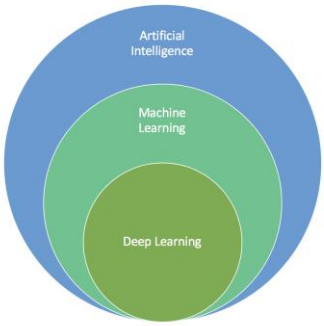
By applying global and local model explanation techniques to the Transfer Model instead of the Original Model, interpretations on the payment transaction features and their contribution to the predictions of the Original Model should become possible.

2. The Process to a Transfer Model

A. Machine Learning

Machine learning belongs to the field of "Artificial Intelligence" ("AI") and stands for the attempt to generate or extract knowledge from data:

Figure 1: AI, Machine Learning and Deep Learning



ML methods typically are further subdivided into Supervised and Unsupervised Learning.

In Supervised Learning, the goal is to find a function that best describes the unknown relationship between a known input data set (a vector or matrix of independent variables) and a known output data set (a vector of the dependent or "target" variable) to later make predictions for data whose output data is unknown. Supervised Learning methods are further divided into two groups: classification and regression tasks. If the output data is categorical, it is a classification task. If the output data is of concrete and often continuous numerical values, it is a regression task.

In contrast to Supervised Learning, Unsupervised Learning does not require any output data (no "label"), but only input data or independent variables. The aim is to present the most informative structure that best describes the provided data set. "Clustering" is the attempt to separate the data set into several groups or to form "clusters" along some common feature characteristics. "Anomaly Detection" has the goal to identify those outliers that are so far away from the rest of the data, that they can be classified as "abnormal". A third common use case in Unsupervised Learning is to reduce the dimension or number of features to enable visualizations in a lower-dimensional space that is perceptible to the human eye.

Original- and Transfer-Model

As both, the Original Model and the Transfer Model aim to predict target classes, they clearly belong to the Supervised Learning type and there to the classification category.

B. The typical Machine Learning Process

The typical ML pipeline in a classification task looks like this:

1. ETL

The first step is the ETL-phase, an abbreviation that stands for "Extract, Transform, Load". The data first needs to be collected and might be gathered from different sources (the "Extract" in ETL). The data format then must be converted into a format that ML algorithms and database management systems ("DBMS") accept (the "Transform" in ETL), typically a

tabular format with sole numbers in columns and rows. The "Load" in ETL defines the process where the data is loaded from a database, often with SQL-commands.

Original- and Transfer-Model
The data for both, the Original Model and the Transfer Model has already gone through that process and thus is also available in structured tabular form as "Comma-Separated-Values" ("CSV" format). As the ETL-phase is also not part of the modelling process described in this thesis, I will not elaborate on it further.

2. Preprocessing and Feature Engineering

The second phase deals with handling erroneous and missing data and with converting the data into an appropriate number format. The general aim is to transform the independent variables in such a way that they can be sensibly used by the ML algorithms, or in other words to engineer the features out of the unready variables.

Typically, raw data in a ML process cannot be used immediately but must first be corrected for systematic problems and errors. The reasons for incorrect data can be manifold: Typing errors in data collection, data damage during transmission or storage, erroneous duplication, improper handling, etc.

Missing or erroneous data entries can either be replaced or removed. As a removal of data generally goes along with a loss of information, using an appropriate replacement strategy such as to replace those entries with the mean, median, the most frequent value or a constant is generally preferred.

Some ML algorithms, such as the logistic regression estimator, require that the independent variables do not have high correlations with each other, that they do not have any extreme outliers or that they have a standard normal distribution.

Most ML programs offer a wide range of transformer functions, be it to convert the data into a standard distribution ("standardize"), to clip outliers ("winsorize") or to transform two or more highly correlated variables to a new feature vector.

Original- and Transfer-Model
The independent variables have already been transformed and feature-engineered for both, the Original Model, and the Transfer Model. Both feature sets are available in structured tabular form as CSV-files.

One motivation to engineer new features is that nominal values in general might be less informative than the ratio of two variables. This particularly applies to the payment transaction data as it contains payment amounts that range from comparably low to very high numbers for the biggest corporate clients. For Transfer Features in the same business domain, ratios of two already existent features were calculated and added to the initial feature set in cases this made sense. Altogether, the Transfer Model in its training phase consists of close to 300 features. As the business logic of the feature sets needs to be strictly confidential, the preprocessing and feature engineering phase is not discussed, neither for the Original Model nor for the Transfer Model.

3. Train and Test data and Cross-Validation

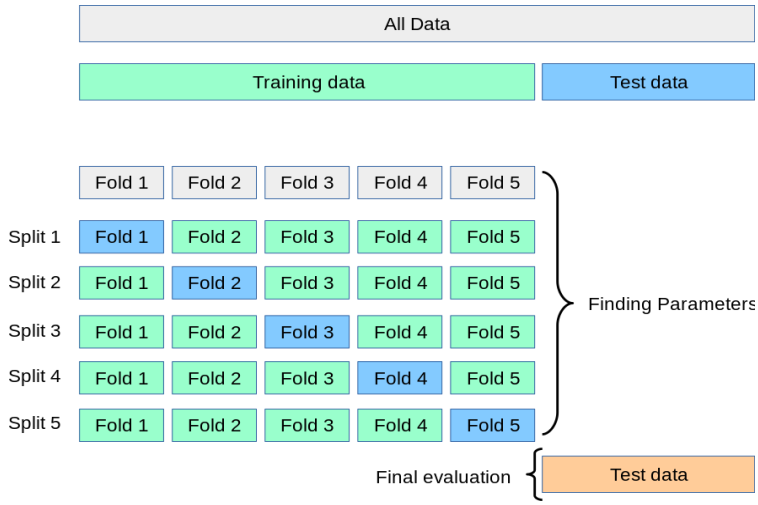
As already outlined above, ML attempts to "learn" from sample data. The actual learning phase in Supervised Learning is generally divided into two parts: In a first "training" step, a ML algorithm calculates a complex function that best describes the relationship between the feature set and the target vector. In a second "testing" step, the best model found during the first step is evaluated on an "unknown" data set from the same domain and with the same features to check the validity and stability of the new model.

Since usually only one data set is available for both, training and testing, the sample data set must first be split into a test and a training set. If this separation was not made and the training data was also used for testing, there would no longer be any "unknown" data. The training data would then already contain the information that one is trying to predict. In data science, this is referred to as "data leakage", as information from a training set "leaks" into a test set. This makes models unusable and must be avoided.

To select from several trained models, evaluation and testing is already desirable during the training phase. However, the test data should not be used for this purpose, as it is only used for the final evaluation of the model eventually selected.

For the training set to be used in both, training and testing, the training data must be split up again. The validation of models within the training set in data science is also known as "cross validation" or "CV".

Figure 2: Cross- and Final Validation



In the figure above, for example, a training set is split up five times and thus each time in a ratio of 4 to 1. The data contained in each split is also known as "fold". With five model training runs, each time four of the five folds are aggregated and used for training, and one fold is used for validation. The quality and stability of a model can be read out from the results of the five validation folds and their variance.

Original- and Transfer-Model

As outlined above, the Original Model has already been created by BDAA and is currently used and run in the programming language R. As it is planned to switch from R to Python in the long term, the Original Model was re-developed in Python and its affiliated libraries using the train-ready Original Feature set.

4. Model Selection

A ML algorithm that calculates a complex function to best describe the relationship between the feature set and the target vector is also known as "estimator". When estimators are trained or "fitted", some parameters are passed to them which are usually referred to as "hyperparameters". Estimators together with its hyperparameters form a ML "model".

The goal in the modelling process is not only to minimize the difference between the predicted and the measured values, but also to increase the stability of the model by setting the hyperparameters accordingly. Some of these parameters aim at "regularize" the ML model to avoid such "overfitting" (see below).

Many different models are available for classification tasks. Three of the most widespread are the Logistic Regression model ("Logit"), the random Forest Classifier model ("RFC") and the XGBoost Classifier model ("XGB").

Despite its name, the Logit model is a linear model for classifying discrete dependent variables using regression methods. Logistic regression aims at calculating the probability of a binary event such as true and false. It uses a logistic function such as the Sigmoid function to map the regression output to a probability value between `0` and `1`.

The RFC model is an "ensemble" method because the algorithm combines the predictions of several decision trees by calculating an average from the individual tree predictions. The RFC is a non-linear model since the decision trees themselves are also non-linear in nature. The probability of an instance to belong to one of the two target values is simply calculated by dividing the number of data instances in the leaves of the aggregated trees.

The XGB model is like the RFC model in that it is a non-linear ensemble method that combines the predictions of several decision trees. It differs from the RFC as it belongs to the class of boosting algorithms that for the aggregate prediction dynamically weigh and re-weigh the results of the individual decision trees depending on their prediction quality.

Predictions made with these classification models usually yield a predicted probability for the sample data to belong to one of the two target classes. Often, a predicted probability of `0.5` is considered the border at and above which a sample is classified to belong to target class `1`.

As the focus of this thesis is not on model selection but on information transfer, I will not elaborate further on the different ML models.

Original- and Transfer-Model

The Original Model written in the programming language R is a composition or "ensemble" of different individual models such as Logit, RFC, XGB and others. The final model result is calculated by weighing and aggregating the results or the "voting" of these individual models. The exact composition of this ensemble model used by BDAA cannot be disclosed due to confidentiality issues.

Nevertheless, it can be disclosed that the Logit model and so called "ensemble" models such as the RFC model and the XGB model were used in the Original Model by BDAA. Their respective feature sets can also be disclosed in an anonymized form.

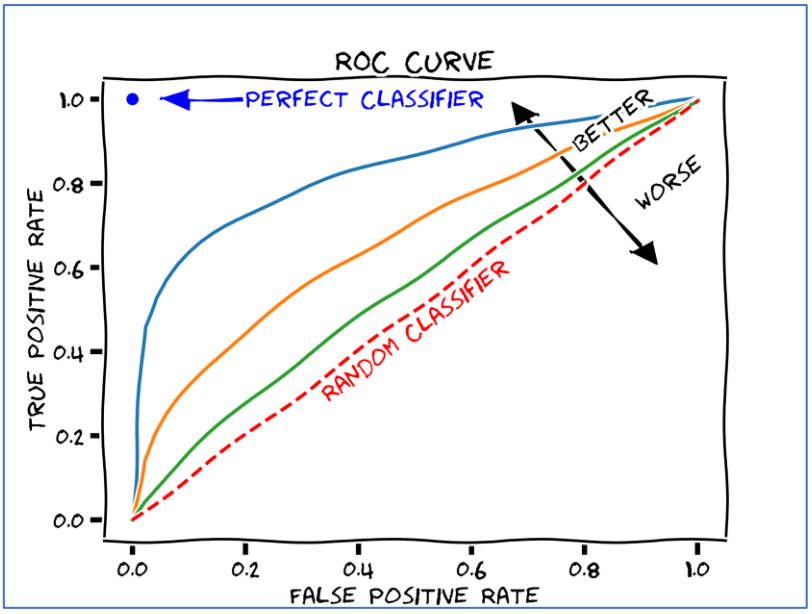
I will therefore use three models (Logit, RFC and XGB) in the application to both, the provided (anonymized) Original Features and the Transfer Features. As a result, there will be three Original "models" and three Transfer "models".

5. Evaluation

The quality measure of models is derived from certain key figures that are calculated upon the answer to the question of whether the forecast for an individual data point was correct or not. False positive ("FP") are forecasts that incorrectly categorize negative events as positive. True positive ("TP") are forecasts that correctly categorize positive events as positive. False negative ("FN") are forecasts that incorrectly categorize positive events as negative. True negative ("TN") are forecasts that correctly categorize negative events as negative. The number of data points in each of these four categories divided by the total number of data points yields their respective "rate".

Typically, no model can predict all data points with complete accuracy. When trying to minimize the number of false predictions, often there is a "trade-off": improvements in one of the above categories will only come at the expense of a deterioration in another category. A well-known measure of accuracy in data science is the measure "ROC-AUC", which stands for the "area under the ROC curve". The ROC curve is a diagram in which the trade-off between the improvement in the true positive rate ("TP") versus the deterioration of the false positive rate ("FP") is shown:

Figure 3: Wikipedia: Receiver operating characteristic



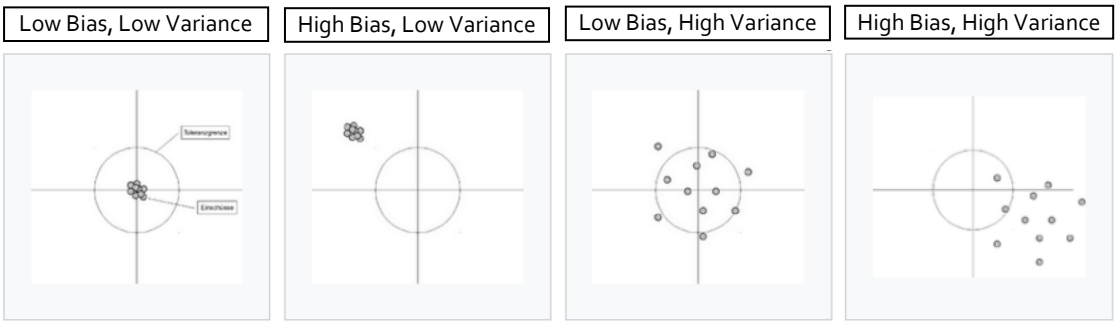
The ROC-AUC score in the above figure quantifies the area within the square that lies to the right or below the colored lines. The smaller the area on the left or above the colored lines, the better the model quality that can be achieved with a given data set. The ROC-AUC score has its minimum value at `0.5` where the model predictions are purely random and its maximum value at `1.0` where all data points are predicted correctly.

If data sets have roughly the same number of target values in each of the two binary classes ("balanced datasets"), the quality of a model can be measured with the so called "accuracy" score.

If data sets have a very unequal number of target values in each of the two binary classes ("unbalanced datasets"), high values of the accuracy score lose their meaningfulness. Quality measures such as the ROC-AUC score should then be used.

An important concept in measuring the quality of ML models is expressed in the terms "Bias" and "Variance". Generally, a "Low Bias, Low Variance"-model is preferred:

Figure 4: Bias and Variance



Source:Wikipedia

"Bias" in Machine Learning can be defined as the phenomena of observing results that are systematically prejudiced due to faulty assumptions.[3] One reason for that could be missing important features that are not yet included in the data set but potentially contained some additive prediction power.
Another source of "Bias" is if we dropped data in the preprocessing stage that itself has a tendency or "Bias" in one or the other prediction direction and thus might lead to a "Bias" in the opposite direction for the remaining data. For such "High Bias" or "underfitted" models, it is recommended to go back and do some more research on the domain topic, to find or engineer more features or to add additional data if possible.
"High Bias" cases can usually be recognized by low values in their quality metrics such as ROC-AUC or "accuracy". I consider models with a ROC-AUC score of between `0.80` and `0.90` as having "Medium Bias" and those with a ROC-AUC score above `0.90` as having "Low Bias".

"Variance" in the context of ML models can be defined as the amount that the estimate of the model will change if different training data was used.[4] "High Variance" suggests large changes to the estimate of the model with changes to the training data. This also includes large changes in the estimates of the different training folds within the cross validation. It is said that the model in this case is "overfitted". Solutions to this problem include to "regularize" the model by setting the regularization hyperparameters accordingly, to manually drop features with little contribution to the model quality metric or to add additional data if possible.
"High Variance" cases can usually be recognized by a high divergence in the ROC-AUC scores of the test sets for different training sets or cross-validation folds. I consider models whose ROC-AUC scores for different test sets do not diverge by more than `0.05` to have a "Low Variance".

<u>Original- and Transfer-Model</u>
As stated previously, loan fraud cases at Commerzbank AG are very rare. Although the data for the Original Features was collected for a time span of over ten years, the data set was still very unbalanced with a low number of positive ("fraud") and an overwhelming number of negative ("no fraud") target values. The appropriate measure for the model quality of the Original Model thus is the ROC-AUC score.

[3] (Jaspret, 2019)

[4] (Brownlee, 2019)

The number of positive target values for the Transfer Model depends on how the target values are calculated as explained in later chapters. Generally, the number of positive target values is also low in the learning data of the Transfer Model. Also, to compare the Transfer Models with the Original Models, the ROC-AUC score was also chosen there as the appropriate model quality metric.

The terms "Bias" and "Variance" will not be discussed for the Original Model but only for the Transfer Model in later chapters.

C. Software

I used Python as the programming language. Python is an open source interpreted high-level general-purpose programming language that comes with a comprehensive standard library. Third-party libraries can easily be integrated.[5]

One of those libraries is Scikit-learn, an open-source machine learning package that supports Supervised and Unsupervised learning. It also provides various tools for model fitting, data preprocessing, model selection and evaluation, and many other utilities.[6] Scikit-learn was used as the main tool for the ML process.

In addition, various open-source libraries were also used for data manipulation and conversion, two of the most important are "pandas" and "NumPy".

Pandas is a fast, powerful, flexible, and easy to use data analysis and manipulation tool built on top of the Python programming language. pandas "DataFrames", a two-dimensional, size-mutable, potentially heterogeneous tabular data structure[7] was used quite frequently.

NumPy is the fundamental package for scientific computing in Python. It provides routines for fast operations on arrays.[8]

In addition to various Python libraries, Jupyter Notebooks were used for interactive analysis and quick Python program runs. Project Jupyter is a non-profit, open-source project to support interactive data science and scientific computing across the three main programming languages Julia, Python and R from which its name derives. [9]

As integrated development environment ("IDE"), I used Jetbrains Intellij with a Python plugin.[10]

D. Differences to the typical ML process

As a result of the previous discussion, the specific process to create a Transfer Model looks like this:

[5] (Wikipedia: Python, 2021)
[6] (Scikit-Learn, 2021)
[7] (Pandas, 2021)
[8] (numpy.org, 2021)
[9] (Wikipedia: Jupyter, 2021)
[10] https://www.jetbrains.com/de-de/idea/

1. **Original Model Training**

The initial feature set for the Original Model contained 664 preprocessed and pre-engineered features and around 14,000 rows of borrowers that were either corporate or business clients. The binary target vector was defined for each borrower to be either fraudulent (target class 1) or not (target class 0) in the context of the BDAA definition of "fraud".

For their ensemble model, BDAA selected 99 features for the Logit model component and 32 features for the XGB and RFC model component, respectively. The final selection was the result of comparing the model performances for different feature sets.

To rerun the three model components from the BDAA ensemble model, I rerun the Logit, XGB and RFC estimators in Scikit-learn on the 99 and 32 Original Features and 14,000 borrowers and repeatedly optimized the hyperparameters. The learning phase of the Original Model ends with obtaining the three best fitted Logit, XGB and RFC model.

2. **Predict Probabilities and calculate Fraud-Scores**

With each of the three best models for Logit, XGB and RFC, predictions are made for current borrowers on monthly loan data to calculate the probability that a current borrower is fraudulent. The monthly data contains the Original Features and around 520,000 borrowers that are either corporate or business clients.

Each of the three Scikit-learn estimators offers a "predict_proba"-method that is supposed to return the probability of the samples to belong to one of the two target classes. It ranges from 0 to 1 indicating very low to high probabilities of "fraud", respectively. Sometimes, results from the predict_proba-method are "distorted in a specific and consistent way"[11] and are not perfectly calibrated. This means that the mean of the predicted probabilities ("predict_probas") might differ from the mean of the real target values for different target bins or segments of the target value distribution.

To overcome this and to enable inter-monthly comparison of results, each month the predict_probas are calibrated by BDAA. This is done by calculating 1000-quantiles or "permilles" of these probabilities for a specific and representative sub-portfolio of borrowers called the "quantile_portfolio".

The desired characteristic of the "quantile_portfolio" is that the derived Fraud-Scores later are uniformly distributed in between the extremes of theoretically 0 and 100. The "quantile_portfolio" should actually be named "permille_portfolio" as theoretically there are 1000 bins or intervals. In practice however, the number of bins might be less than 1000 due to minor rounding differences of pandas "qcut"-function. But for generalization purposes, in the program it is still referred to as the "quantile_portfolio".

The predicted probabilities for all borrowers are then allocated into these 1000 (range differing) permille bins. The permille of the 1000 bins ("permille interval") in which the predicted probability is contained then serves as the Fraud-Score and thus enables the desired inter-monthly comparison and calibration. The Fraud-Scores range from 0 to 100 in 0.1-steps.

[11] (Caruana, et al., 2006)

3. Convert Fraud-Scores to Transfer Model Target Values

The intended Transfer Model shall transport information from one feature set into another and carry this information in the values of the target vector. The target values in the Original Model are the predict_probas and the derived Fraud-Scores. Thus, the target values of the Transfer Model must also be the predict_probas or the Fraud-Scores in one form or another.

The first possibility to create the Transfer Model target vector is to just take the Fraud-Scores itself. But as the Fraud-Score is a continuous variable ranging from `0` to `100` in `0.1`-steps, the task to predict these values would be a regression task and the Transfer Model would be a regression model.

As the Transfer Model aims at supporting the investigators during their analysis, the investigators would be overwhelmed by a big number of borrowers and would rather focus only on those that have the highest Fraud-Scores. This necessitates a threshold value for the Fraud-Score above which borrowers are further analyzed and a distinction between "suspicious" and "not-so-suspicious" borrowers based on model results. If such a distinction must be made, a classification task is also an appropriate approach.

The second possibility to create the Transfer Model target vector therefore is that the Fraud-Score values will be converted to binary target values. Fraud-Scores at or above a certain threshold will be classified to belong to class `1` ("fraud"), else to class `0` ("no fraud").

The threshold can be set in different ways. Each of the fitted Logit, XGB and RFC models outputs different predictions and thus different Fraud-Scores for each of the (currently) around `520,000` borrowers in the monthly data. A threshold level could be applied to all, any two or only one of the three model's Fraud-Scores. For this approach, each of the different combinations would need to be investigated. It would require in-depth research on if and why the three models differ in their results and how they are correlated. As this goes beyond the scope of this thesis, the chosen approach was to require all three models to have a Fraud-Score above a certain threshold to classify the respective borrower as "fraudulent". The Transfer Model so could also be seen as an "ensemble" model that requires the aggregated results of three different models.

But at which level shall this threshold be set? A threshold level set very low, would yield a more balanced data set as the number of borrowers in each of the target classes (class `0` = "no fraud" and class `1` = "fraud") were similar. But this would also result in more false positive predictions.

A threshold level set very high, would include only the most suspicious borrowers and would result in a lower false positive rate. But this would also come at the disadvantage to have a very unbalanced data set with many samples in class `0` ("no fraud") and only very few samples or borrowers in class `1` ("fraud"). It would also increase the false negative rate.

The chosen approach was a classification task in that the Fraud-Score values were converted to binary target values and the threshold was set to a minimum Fraud-Score of `90` required to be passed by all three models. This was the result of a repetitive process to maximize model performance, have a balanced data set and a lower number of false positive predictions. In later chapters, I will elaborate on this.

This concludes the overview of the specific ML process to create a Transfer Model.

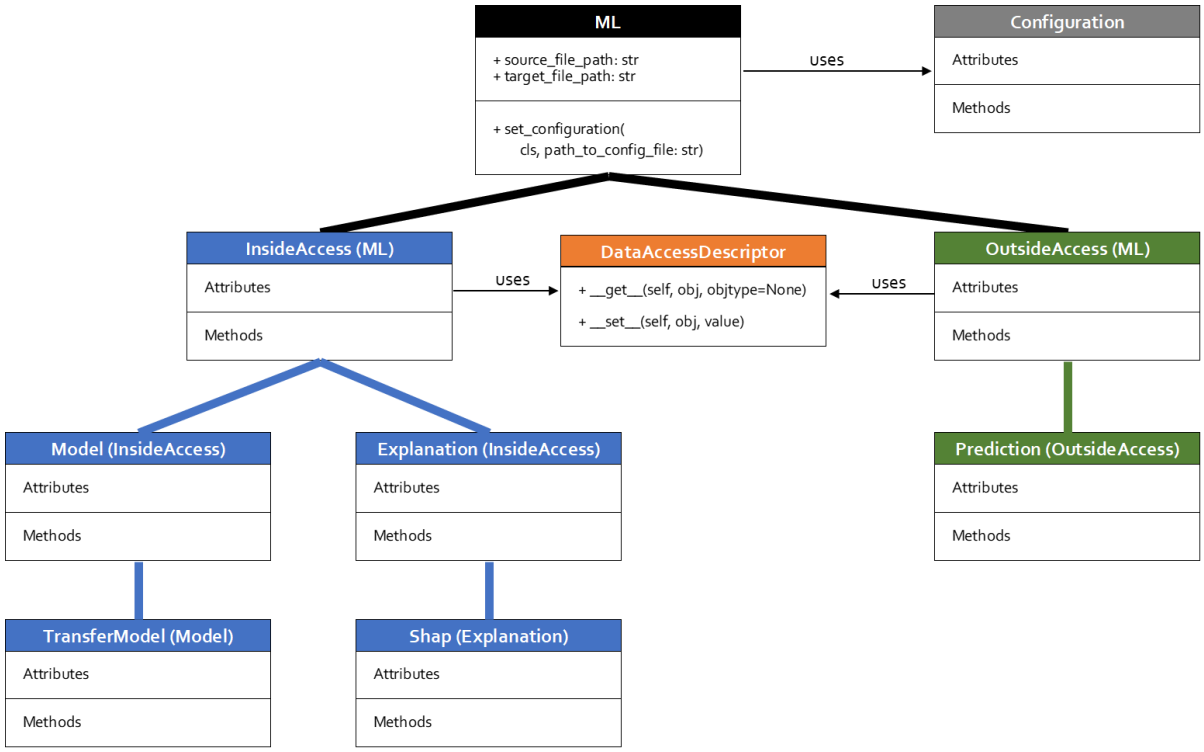
E. Program Architecture

ML program code in the modelling, prediction and explanation phase is often long and contains many similarities or even redundancies. This particularly applies to a Transfer Model as the modelling, prediction and explanation must be done for both, the Original Model, and the Transfer Model. It thus is a typical use case for the Object-Oriented programming ("OOP") software architecture paradigm. What follows is the description of the program architecture and the classes used.

1. Class Diagram

The following class diagram, that excludes the signature of the methods and the attributes, provides a rough overview over the architecture of the program:

Figure 5: Class diagram



The architecture style adapts to the modelling technique of "Domain-driven design"[12], a concept where class names usually refer to the business logic they implement.

A. Class ML

The class "ML" is contained in the Python file "ml.py" and is the most senior class in the inheritance structure. Its only (class) method "*set_configuration()*" instantiates an object of class "Configuration" and to it passes the file path to a configuration file ("config.ini"). The instance of class "Configuration" is saved and later used by all sub-classes to get access to the config.ini-file. The config.ini-file contains all the parameters, settings and directories for the modelling, prediction, and explanation phase. Without an instance of class "ML", the sub-classes cannot be instantiated. The class (in a modified form) so could serve as the security gate that protects access to the entire ML process.

[12] (Evans, 2003)

B. Class Configuration

The class "Configuration" is also contained in the Python file "ml.py" and makes use of Python's "ConfigParser", a class contained in the "configparser" library that creates and reads configuration files. Its main task is to read data from the config.ini-file.

C. Class DataAccessDescriptor

Python Descriptors let objects customize attribute lookup, storage, and deletion.[13] The class "DataAccessDescriptor" serves as an accessor to data, here to data saved in files. But it could also be used to access data stored in DBMS or any other form of persistent data storage.
Descriptors implement and overwrite any or all of the three Python "dunder"-methods ("double underscore methods") "*get*()", "*set*()" or "*delete*()". DataAccessDescriptor instances are set as class attributes in the classes "InsideAccess" and "OutsideAccess". If values are assigned TO these instances, the "*set*()"-method of the Descriptor is called and so saves the data to the referenced directory and file. If values are assigned to any other variable FROM these instances, the "*get*()"-method of the Descriptor is called and so loads the data from the referenced directory and file.
The implemented "*get*()" and "*set*()" methods load and save either csv-files or joblib-files. The csv-format was chosen for the persistent storage of tabular data and the joblib-format for the persistent storage of the fitted Scikit-learn model estimators.

D. Classes InsideAccess and OutsideAccess

To separate use cases where model internals shall be available only to some objects and users but not to others, the two different access rights to these model internals are incorporated in the two different classes "InsideAccess" and "OutsideAccess".
Both classes are contained in the "data_access.py"-file and inherit from the class "ML".
Instances of class "InsideAccess" have full access to model internals as they have different instance attributes and can also use all class instances of the DataAccessDescriptor-class as described above. With their respective "load_..."- and "save_..."-methods, "InsideAccess"-instances so have access to all persistently stored data.
On the other hand, "OutsideAccess"-instances with their own attributes and own "load_..."- and "save_..."-methods contain less and different DataAccessDescriptor-class instances and thus have only access to general output- or generic model-information. Sensitive data, such as the training and test data or the model parameters are so hidden from those objects and users. The class "OusideAccess" currently only has one sub-class named "Prediction" but is forecast to get other children in a production mode for purposes such as monthly "Reports" to be produced for the investigators.

E. Class Model

The class "Model" implements the ML modelling logic and is one of the biggest classes in the program. It is contained in the "model.py"-file and is a sub-class of "InsideAccess" from which it inherits the instances, attributes and full data access rights.
Its methods make use of the Scikit-learn toolset and the data structures and manipulation tools provided by pandas and NumPy.

i) *set_dataframe(...)*

The learning data can either be loaded with the "InsideAccess"-method "*load_dataframe*()" or by passing as parameter to this method a pandas "DataFrame" that contains the feature set and the target vector.

[13] (Hettinger, 2021)

ii) *set_key_target_features(...)*

This method receives the following strings and lists as parameters and sets them as instance attributes: the name of the target, the name of the features, the name of the features excluded in the training and the names of the key and the second key. The two "key" parameters are the columns in the "dataframe" that identify a particular data row and represent the individual corporate or business client. The first key is the Commerzbank internal client identification number ("ID") and the second key the months and year the data was recorded. As the Original Model data contained duplicated client ID's for different months, the second key was necessary to make the data row unique. The name of the (first) key in the anonymized dataframe is "orig_key_1" and the name of the second key is "orig_key_2".

The "features_excluded_in_training"-parameter lists those column names in the "dataframe", that might be key values to identify individual rows, but anyway are not desired for the training of the model. These names must be carried for reasons related to the avoidance of data leakage as explained below.

iii) *set_estimator_and_parameters(...)*

This method is passed a Scikit-learn estimator and a reference to the hyperparameters in the config.ini-file. The retrieved hyperparameters and the passed estimator are then set as "Model" instance attributes.

If the estimator is of type "Logit", which might require either imputation, standardization or winsorization of the learning data in the cross-validation phase, a Scikit-learn "Pipeline" instance is created by making a call to the private method "make_pipeline(...)". All the parameters needed for the "Pipeline" are retrieved from the config.ini-file and set in this private method. The "Pipeline" instance is then set as estimator attribute instead of the "Logit"-estimator and later is treated differently compared to estimators that are not a "Pipeline". To keep track of that, the Boolean flag "estimator_is_pipeline" is set to "True".

iv) *under_sample_dataframe(...)*

This method allows to rebalance an unbalanced dataset, i.e., a data set whose target values are heavily dominated by just one of the two target classes. As even the Transfer Model data contains much more "non-fraud" than "fraud"-cases, this method allows to resample and resize the data set so that the "majority"-class is reduced according to the desired strategy. This method is later used in the Transfer Model. Allowed parameters are float-type numbers that express the desired ratio of the minority class (`1` = "fraud") to the majority class (`0` = "no fraud") or any string allowed as "sampling_strategy" as defined by the "RandomUnderSampler"-method of the Scikit-learn "imbalanced-learn" library.

v) *train_test_split_dataframe(...)*

This method splits the "dataframe"-rows into a training set and a test set. If the parameter "group" is provided (default is: None), it must be the name of a feature in the "dataframe" with duplicate entries that shall either be in the train and or in the test set but not in both to avoid data leakage. The duplicate entries form so called "groups" and Scikit-learn's tool "GroupShuffleSplit" can create an iterator that separates these "groups".

The Original Model contained borrowers with different company ID's but some of them belong to the same conglomerate or concern and thus had the same concern ID. As the company data has some overlap with the concern data, the concern ID's form "groups" that need to be accounted for in the train-test-split. After retrieving the split ratio from the config.ini-file, it is passed to the "GroupShuffleSplit"-function along with the name of the "groups" column in the "dataframe". The name of the "groups"-column in the anonymized "dataframe" is "orig_key_3". The function then splits the "dataframe"-rows into a train and a

test set, accounting for the mentioned "groups" by ensuring that rows with equal concern ID's are EITHER in the test set OR in the training set but not in both. The function then stores the row indexes of the respective sets in the "Model" instance. The private method "*set_train_test_set_rows*()" eventually splits the rows of the feature set and the target vector into "X_train", "X_test", "y_train" and "y_test" sets along these indexes and saves them as "Model" instance attributes.

vi) *train_model_and_cross_validate(...)*

This method does all the training and cross-validation. As data leakage also lures in this phase, it must be ensured that the individual cross-validation folds are also separated along the just discussed "groups". This is accomplished by using yet another Scikit-learn function named "GroupKFold" which does exactly what the "GroupShuffleSplit"-function does, except that it does this only for the "X_train" dataset rather than the entire learning data. Its parameters are not only the "group" column name, but also the number of folds ("num_cv_folds"). The "num_cv_folds" is retrieved from the config.ini-file as is the "scoring" method.

Besides the created "GroupShuffleSplit"-iterator and the scoring-method, the previously retrieved hyperparameters and the provided estimator also go into a newly created Scikit-learn "GridSearchCV"-instance as parameters. Finally, the "*fit(...)*"-method is run on that instance and on the feature set that excludes the "features_excluded_in_training" to do the cross-validation. The best estimator with the best hyperparameters is then saved as instance attribute and can be persistently stored with the "IndsideAccess"-method "*save_fitted_estimator(...)*".

vii) *get_cv_results()* and *get_test_set_roc_auc_score()*

To get the cross-validation results in the form of an aggregated pandas "DataFrame", the method *get_cv_results()* can be run. To get the "ROC-AUC score" for the previously set aside test data set, the method *get_test_set_roc-auc_score()* can be used.

viii) *generate_prediction_input()* and *generate_explanation_input()*

The *generate_prediction_input()*-method can generate a "namedtuple" data structure from the Python collections library. This namedtuple "PredictionInput" object carries all the required data that is needed for instances of class "Prediction". The *generate_explanation_input()*-method also creates a namedtuple "ExplanationInput" object also carrying all the required data, that is later needed for instances of class "Explanation". This object can be persistently stored to interrupt and later again resume the ML process.

F. Class Prediction

The class "Prediction" is contained in the Python file "prediction.py" and its task is to perform monthly predictions. It inherits from class "OutsideAccess" and so has only limited access to the model internals. At instantiation, it must receive a "PredictionInput" object, a "namedtuple" data structure that contains the fitted estimator, the feature names in the training phase, the name of the target vector, the key name ("orig_key_1" in the anonymized "dataframe") and whether the fitted estimator is an instance of Scikit-learns "Pipeline". The "PredictionInput" object does not contain the learning data, nor does it reveal the hyperparameters or any results from the model's learning phase. As described in the last chapters, there are several steps to arrive at the Fraud-Score used at BDAA:

i) *set_quantile_keys(...)*

Firstly, the probabilities for the two target classes are predicted for a representative monthly sub-portfolio of borrowers (the "quantile_portfolio").

The meaning of the "quantile_portfolio" is described in detail in the "Predict Probabilities and calculate Fraud-Scores" chapter above.
To set this "quantile_portfolio", this method *set_quantile_keys(...)* receives as parameter a pandas DataFrame containing the key values (key: "orig_key_1") for the representative "quantile_portfolio". It then stores as an instance attribute the key values as list.

ii) *get_pred_proba_for_key()* and *predict_probas()*
Next, the *predict_probas()*-method predicts the probabilities for either target class by using the Scikit-learn-function *predict_proba(...)*. For that, a "prediction_dataframe" containing the monthly data for which the probabilities shall be predicted, must have been loaded before with the "OutsideAccess"-method *load_prediction_dataframe(...)*.
The method *get_pred_proba_for_key()* can also predict probabilities for an individual borrower if a value for the key "orig_key_1" is provided.

iii) *calculate_quantiles()*
This method is the most comprehensive in the class. Its task is to eventually convert the just calculated probabilities for the two target classes to Fraud-Scores.
This process is described in detail in the "Predict Probabilities and calculate Fraud-Scores" chapter above.
In the first step, only those borrowers that are in the "quantile_portfolio" list - together with their predicted probabilities – go into the instance attribute "instance_quantile_portfolio", a pandas DataFrame.
Next, the "permilles" and the "permille-bins" ("permille intervals") are calculated for the predicted probabilities of the "quantile_portfolio". Both are added to the "instance_quantile_portfolio" as new columns.
Into the "instance_result_dataframe", which already contains the key values and the predicted probabilities, the "permille-bins" from the "instance_quantile_portfolio", are inserted according to the predicted probabilities therein with the pandas "cut"-function.
In the final step, the permilles from the "quantile_portfolio" are also inserted or "merged" into the "instance_result_dataframe". As already described, the permilles also serve as Fraud-Scores.

iv) *add_results_to_aggregate()*
Because the "instance_result_dataframe" and the "instance_quantile_portfolio" are instance attributes, they must be aggregated at a higher level if the Fraud-Scores threshold shall be calculated on the aggregated Fraud-Scores for the Logit, XGB and RFC models. The method *add_results_to_aggregate()* aggregates the individual instance predictions to the "result_dataframe" and the "quantile_portfolio", which are class attributes of the "OutsideAccess"-class. These aggregated class attributes can also be loaded and persistently stored with the "OutsideAccess"-methods *load_quantile_portfolio(...)*, *save_quantile_portfolio(...)*, *load_result_dataframe(...)* and *save_result_dataframe(...)*.

v) *generate_transfer_model_inputs()*
The *generate_transfer_model_inputs*()-method can generate a "namedtuple" data structure. This namedtuple "TransferModelInput" object carries all the required data that is needed for instances of class "TransferModel".

G. Class TransferModel
The class "TransferModel" is a sub-class of class "Model" and is also contained in the "model.py"-file. It has one proprietary method and overwrites one of its parent's methods to account for "TransferModel" specifics.

i) *set_transfer_model_inputs(...)*

This method is passed a namedtuple "TransferModelInput" that contains the Original Model's target name, the key name of the Original Model and most importantly the three "permilles" (one for each model) from the "quantile_portfolio".

The Original Model's target name is converted to the Transfer Model's target name by concatenating a "TRANS_TARGET_of_"-prefix.

The name of the two key variables ("orig_key_1" and "orig_key_2" for the anonymized data set) of the Transfer Model are taken over from the Original Model.

The three "permilles" from the "quantile_portfolio", as stated previously, represent the Fraud-Scores for each of the three models. They are used to calculate a Fraud-Score-threshold at or above which the Transfer Model target class is equal to one (1 = "fraud") and below that equal to zero (0 = "no fraud"). This is done by first receiving the settings for this calculation from the config.ini file, which are then used by the two "helper"-methods *create_target_column()* and *combine_targets_with_features()* to arrive at the Fraud-Score-threshold. The calculated values represent the target vector of the Transfer Model.

ii) *set_key_target_features(...)*

This method overwrites three of the instance parameters passed to this method and sets the "key", "second_key" and "target" attributes according to the data received in the namedtuple "TransferModelInput" that is processed by the previous method.

As the "TransferModel" is a sub-class of class "Model", a Transfer Model instance could also generate a namedtuple "PredictionInput" and use it for another "Prediction"-instance to predict monthly data based on the new Transfer Features.

H. Class Explanation

The class "Explanation" is contained in the "explanation.py"-file and inherits from class "InsideAccess". It thus has full access to the model internals as most Python libraries used to do feature explanations require this.

The class currently has only one sub-class "Shap" but will get more children if explanation techniques other than Shapley-values shall be used in production mode.

Its only method is the init(...)-method that receives a namedtuple "ExplanationInput" object that contains the fitted model estimator, the learning data set "X", the "X_train" and "X_test" set and the key names. If features for monthly data shall be explained, the "shap_data_for_explanation"-variable in the config.ini-file must be set to "monthly" and a "prediction_dataframe" loaded first.

I. Class Shap

The class "Shap" inherits from class "Explanation" and is also contained in the "explanation.py"-file. It uses the Python library "shap" that offers insights into the contribution of features to model results which will be explained later in more detail.

The class offers methods that do not only calculate Shapley-values but also provide data around this topic that can be extracted and analyzed further.

In addition to that, the class offers methods to plot graphs that contain all the relevant information but do not reveal model internals. The latter could be used to create informational reports.

i) create_shap_objects()

This method sets some required variables and creates an "explanation"- and an "explainer"-object that are saved as instance attributes. The "data"-parameter passed to the explainer object must be a shap "masker" object instead of the actual data, because otherwise the

standard masker is set in the background whose "max_samples"-attribute is set to `100` rows only.

ii) *calc_global_explanation()*
After invoking the *create_shap_objects()*-method, the Shapley-data is imported into a pandas DataFrame and saved as "global_shap_values"-attribute with this method.

iii) *get_local_shap_values_from_global(...)*
There are two ways to get Shapley-values for an individual data row or individual borrower: The first is to get them from the "global_shap_values"-attribute. Then the Shapley-values are just extracted from that global Shapley-values data set. The method *get_local_shap_values_from_global()* does exactly that.

iv) *get_local_shap_values_from_method(...)*
The second possibility to get Shapley-values for an individual data row or individual borrower is to let the method "shap_values" from the shap library calculate the Shapley values from the feature data that are passed as parameters. To do that, the method *get_local_shap_values_from_method(...)* first extracts the feature data from the "shap_explanation_data" and then applies the "shap_values"-method. As of the time this thesis is written, there is a (minor) difference for the individual Shapley-values dependent on the method that is used to calculate them. The results from this method add up to the difference between the calculated Shapley-values and the respective base value and is therefore considered to be "correct".

v) *compare_local_shap_from_global_with_local_shap_from_method(...)*
This method shows the difference between the local Shapley-values for the two different approaches.

vi) *plot_global_bars(...), plot_global_beeswarm(...)* and *plot_global_heatmap(...)*
These methods for the respective type of plot show Shapley-values for all the data that was referenced in the config.ini-file as "shap_data_for_explanation"-variable.

vii) *plot_global_scatter_for_features(...)*
This scatter plot shows the Shapley values for one or two features. Dependent on the settings for the parameters, it can also show an interaction effect.

viii) *plot_local_force(...)*
This is the most important plot to show Shapley-values for an individual borrower. It is based on the method *get_local_shap_values_from_method(...)* that is considered to show the "correct" values as stated above. It is passed one key value ("orig_key_1"), or if the Original Features shall be explained, two key values ("orig_key_1" and "orig_key_2") as parameters.

ix) *reduce_shap_explanation_data_rows()*
Not all "helper"-methods are mentioned here, but this one deserves some attention. The time to calculate Shapley-values increases exponentially by the number of rows or borrowers in the data set. To limit the calculation time and to reduce the size of the "shap_explanation_data"-attribute, this method receives the "max_num_explanation_data_rows"-variable from the config.ini-file and randomly selects the provided number of rows from the "shap_explanation_data" to reduce its size.

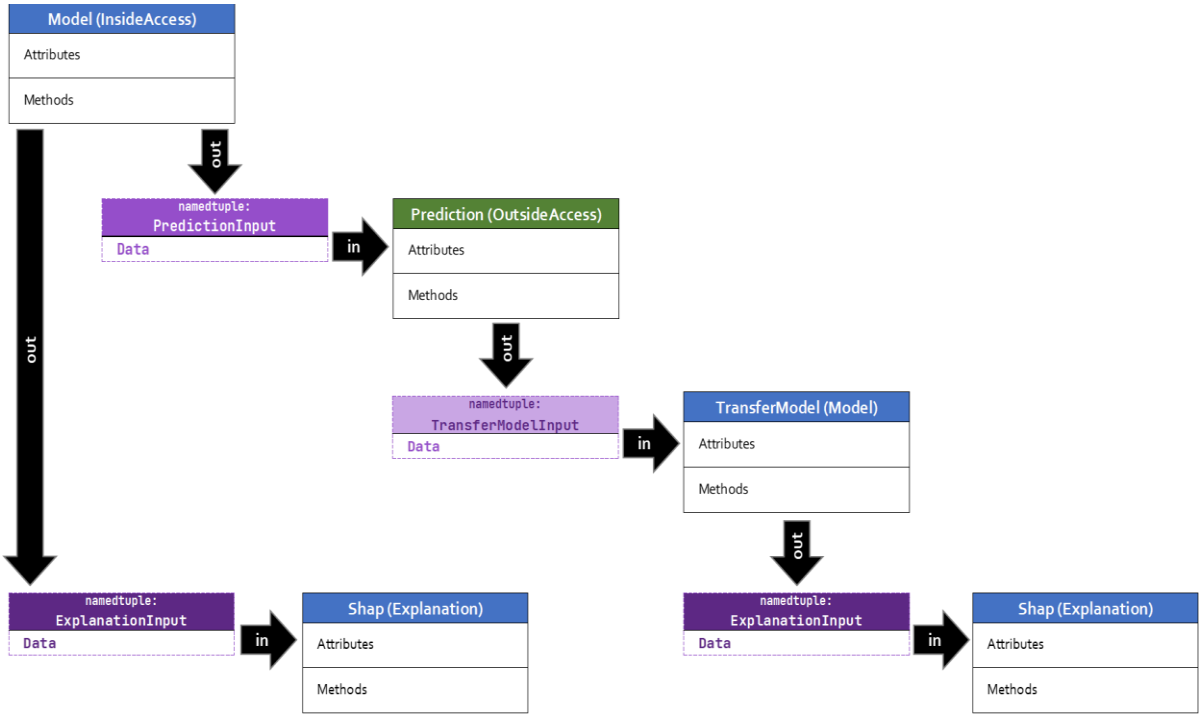
J. Others

There are some other Python-files, classes, and helper methods in the program. These are not explained as they are outside the focus of this thesis. Nevertheless, they are all shown in the appendix.

2. The ML Process with Class Instances

As a result of the previous chapter's discussion, the specific ML process to create a Transfer Model in the OOP-paradigm looks like this:

Figure 6: namedtuples as input objects



Firstly, an instance of class "Model" withs its methods and attributes, does all the ML modelling for the Original Model as described previously. It concludes with generating a "PredictionInput" object, a "namedtuple" data structure from the Python collection library. The "PredictionInput" object carries all the required data that is passed to a newly created instance of class "Prediction".

Secondly, this instance of class "Prediction" then with its methods and attributes does all the desired predictions and at the end generates a "TransferModelInput" object. This again, is a "namedtuple" data structure, that contains all the required data for a newly created instance of class "TransferModel" to which it is passed.

Thirdly, the instance of class "TransferModel" then does all the desired ML modelling for the Transfer Model and concludes with generating an "ExplanationInput" object, also a "namedtuple" data structure.
Fourthly, this "ExplanationInput" object carries all the required data that a newly created instance of class "Shap" needs to do SHAP explanations on the Transfer Features (on which I will elaborate later).

Finally, the instance of class "Model" created in the first step could also generate an "ExplanationInput" object that would serve as input to another instance of class "Shap" to explain the Original Features.

The namedtuple data structures in this way encapsulate sensitive information and ensure that data is revealed only to privileged users and objects. These data structures can also be stored persistently with the help of the Python joblib library. This allows to interrupt the ML model at any stage and later resume it.

3. Program Code

To show the ML process from an Original Model to a Transfer Model in program code, some important steps are shown as extracts of a Jupyter Notebook for just one of the three models, here the XGB model:

(1) Set the configuration with the class method of class "ML":

```
ML.set_configuration(path_to_config_file='../Configuration/config.ini')
```

(2) Create an instance of class "Model" for the Original Model:

```
xgb_model = Model()      # For: XGBoost or XGB
```

(3) Load the learning data for the Original Model by passing in references to the paths of the file in the config.ini-file:

```
xgb_model.load_dataframe(config_section='Original.Model.Data.Training',
                         config_name='learn_data_preprocessed')
```

(4) Set key, target and features of the Original Model. The respective features (32 features for XGB and RFC ("ensemble_index"), 99 features for Logit ("logit_index")) are referenced to two lists in a Python-file named "Feature_Indices_Anonymized.py" (see appendix: "feature indexes"):

```
xgb_model.set_key_target_features(key='orig_key_1',target='orig_target',
                                  features=ensemble_index,
                                  features_excluded_in_training=
                                 ['orig_key_1','orig_key_3', 'orig_key_2'],
                                  second_key='orig_key_2')
```

(5) Split the Original Model learning data into a train and a test set. The "orig_key_3" feature vector must be "grouped" to avoid data leakage:

```
xgb_model.train_test_split_dataframe(group='orig_key_3')
```

(6) Set estimators and hyperparameters for the Original Model:

```
estimator = XGBClassifier()
xgb_params = 'xgb'
xgb_model.set_estimator_and_parameters(unfitted_estimator=estimator,
                                       hyper_params_dict_name_in_config=
                                       xgb_params)
```

(7) Train the Original XGB model and cross-validate them:

```
xgb_model.train_model_and_cross_validate()
```

(8) Get the cross-validation results for the Original Model.

```
xgb_model.get_cv_results()
```

Here, extracts from the "results_dataframe" are shown for the (Original) XGB-model ("xgb_model"):

split0_test_ROC-AUC	0.957058
split1_test_ROC-AUC	0.939859
split2_test_ROC-AUC	0.942978
split3_test_ROC-AUC	0.924194
split4_test_ROC-AUC	0.956416
mean_test_ROC-AUC	0.944101
std_test_ROC-AUC	0.013556

The ROC-AUC scores for the five individual test folds and their mean of 0.9441 indicate a good model fit.

(9) Get the ROC-AUC score for the test set:

```
xgb_model.get_test_set_roc_auc_score()
```

ROC-AUC score for the XGB model: **0.943084.** The good model fit is confirmed by the ROC-AUC score of the test set.

(10) Generate a "PredictionInput" (namedtuple) object from the XGB model and pass it to a newly created instance of the class "Prediction":

```
xgb_model.generate_prediction_input()
predict_xgb = Prediction(xgb_model.prediction_input_object)
```

(11) Load monthly data (that contains the Original Features) for which to make predictions ("predict_probas"):

```
predict_xgb.load_prediction_dataframe(config_section=
                                      'Original.Model.Data.Monthly',
                                      config_name='data_for_prediction')
```

(12) Load the representative "quantile_portfolio" for which later "permilles" and "permille_bins" are calculated:

```
predict_xgb.load_quantile_portfolio(config_section=
                                    'Original.Model.Data.Monthly',
                                    config_name=
                                    'quantile_portfolio_key_values')
```

(13) Set the quantile keys from the "quantile_portfolio":

```
predict_xgb.set_quantile_keys()
```

(14) Predict the probabilities ("predict_probas") for the monthly data of the Original Model:

```
predict_xgb.predict_probas()
```

(15) Now calculate the "permilles" and "permille_bins" for the "quantile_portfolio" based on the predicted probabilities ("predict_probas"):

```
predict_xgb.calculate_quantiles()
```

(16) Aggregate the "permilles" and "permille_bins" for all three models at the class level to later use them for the Transfer Model:

```
predict_xgb.add_results_to_aggregate()
```

(17) The "result_dataframe" and the "quantile_portfolio" with the aggregated values can be shown from any instance:

```
predict_xgb.result_dataframe

predict_xgb.quantile_portfolio
```

Figure 7: Result dataframe

Index	orig_key_1	predict_probas_RandomForestClassifier	predict_probas_RandomForestClassifier_bin	predict_probas_RandomForestClassifier_quantile	predict_probas_Pipeline_LogisticRegression	predict_probas_Pipeline_LogisticRegression_bin	predict_probas_Pipeline_LogisticRegression_quantile	predict_probas_XGBClassifier	predict_probas_XGBClassifier_bin	predict_probas_XGBClassifier_quantile
0	k1_val1	0	(-1e-05, 0.00016949]	0	0.006589	(0.0065653, 0.0065965]	38.3	0.00011	(0.00010966, 0.00011064]	15.4
1	k1_val2	0	(-1e-05, 0.00016949]	0	0.006589	(0.0065653, 0.0065965]	38.3	0.00011	(0.00010966, 0.00011064]	15.4
2	k1_val3	0.000714	(0.00068966, 0.00076923]	0.5	0.001356	(0.0013462, 0.0013603]	13.6	0.000028	(2.7941e-05, 2.8382e-05]	4.5
3	k1_val4	0.015979	(0.015927, 0.015994]	13.7	0.003072	(0.0030657, 0.0030855]	24.6	0.000234	(0.00023234, 0.00023457]	24.9
4	k1_val5	0.005791	(0.0056465, 0.0057957]	3.9	0.004864	(0.0048469, 0.0048723]	32.4	0.000327	(0.00032504, 0.00032739]	29.5
...	...	...	...	...	...	...	...	...	...	...
518881	k1_val523644	0.045523	(0.04544, 0.045584]	44.2	0.120684	(0.1197, 0.12086]	88.5	0.062695	(0.06209, 0.063324]	90.6
518882	k1_val523645	0.125684	(0.12522, 0.1258]	77.5	0.046415	(0.046406, 0.046677]	76.8	0.032723	(0.032498, 0.033002]	86.8
518883	k1_val523646	0.028621	(0.028615, 0.028728]	27.8	0.065322	(0.065019, 0.065498]	81.7	0.025532	(0.025338, 0.025661]	85
518884	k1_val523647	0.125684	(0.12522, 0.1258]	77.5	0.046415	(0.046406, 0.046677]	76.8	0.032723	(0.032498, 0.033002]	86.8
518885	k1_val523648	0.028621	(0.028615, 0.028728]	27.8	0.179754	(0.17898, 0.18175]	91.8	0.025532	(0.025338, 0.025661]	85

Figure 8: The quantile_portfolio

Index	orig_key_1	predict_probas_RandomForestClassifier	predict_probas_RandomForestClassifier_bin	predict_probas_RandomForestClassifier_quantile	predict_probas_Pipeline_LogisticRegression	predict_probas_Pipeline_LogisticRegression_bin	predict_probas_Pipeline_LogisticRegression_quantile	predict_probas_XGBClassifier	predict_probas_XGBClassifier_bin	predict_probas_XGBClassifier_quantile
0	k1_val21	0.054195	(0.054097, 0.05424]	51.4	0.106531	(0.10609, 0.10714]	87.3	0.001809	(0.0017974, 0.0018119]	55.2
1	k1_val23	0.109139	(0.1088, 0.10928]	74.3	0.003593	(0.0035886, 0.0036062]	27.1	0.000724	(0.0007204, 0.00072604]	41.3
2	k1_val32	0.331059	(0.32872, 0.33205]	91.9	0.053792	(0.053786, 0.054107]	79	0.043525	(0.043376, 0.044101]	88.6
3	k1_val42	0.074961	(0.07496, 0.075193]	63.5	0.109817	(0.10902, 0.11021]	87.6	0.003759	(0.0037393, 0.0037683]	65.3
4	k1_val44	0.032109	(0.032056, 0.032183]	31.3	0.004745	(0.0047424, 0.0047664]	32	0.000393	(0.00039089, 0.00039325]	32.2
...	...	...	...	...	...	...	...	...	...	...
87310	k1_val523592	0.042161	(0.042135, 0.042237]	41.3	0.034866	(0.034824, 0.03499]	72	0.002085	(0.0020696, 0.0020859]	57.2
87311	k1_val523594	0.054038	(0.05395, 0.054097]	51.3	0.015847	(0.015804, 0.015877]	56.6	0.002519	(0.0025028, 0.002522]	59.9
87312	k1_val523626	0.02251	(0.022489, 0.022577]	21.3	0.056758	(0.056583, 0.056998]	79.7	0.032392	(0.032015, 0.032498]	86.7
87313	k1_val523627	0.069936	(0.069765, 0.069975]	61	0.144304	(0.14317, 0.14485]	90.1	0.002134	(0.002128, 0.0021434]	57.6
87314	k1_val523631	0.040824	(0.04078, 0.040872]	40	0.085798	(0.084973, 0.085822]	85	0.019194	(0.019006, 0.019213]	82.8

(18) Generate a "TransferModelInput" (namedtuple) object:

```
predict_xgb.generate_transfer_model_inputs()
```

(19) Create a new instance of the class "TransferModel" for a new XGB model:

```
trans_model_xgb = TransferModel()
```

(20) Load the learning data for the Transfer Model:

```
trans_model_xgb.load_dataframe(config_section=
                               'Transfer.Model.Data.Training',
                               config_name='learn_data_feat_engineered')
```

(21) Pass the "TransferModelInput" (namedtuple) object to the "TransferModel" instance with the method *set_transfer_model_inputs(...)*. This method also creates the target vector column with class `0` and class `1` values based on the Fraud-Score-threshold from the config.ini-file:

```
trans_model_xgb.set_transfer_model_inputs(transfer_model_input=
                                  predict_xgb.transfer_model_input_object)
```

(22) Set the key, target and features for the Transfer Model, which are mostly predetermined by the data contained in the "TransferModelInput" (namedtuple) objects. Only the "features_excluded_in_training"-parameter needs to be set:

```
trans_model_xgb.set_key_target_features(features_excluded_in_training =
                                ['orig_key_1', 'orig_key_2', 'orig_key_3'])
```

(23) Set estimators and hyperparameters for the Transfer XGB model:

```
estimator = XGBClassifier()
params_name_xgb = 'xgb'
trans_model_xgb.set_estimator_and_parameters(unfitted_estimator=estimator,

hyper_params_dict_name_in_config=params_name3)
```

(24) Under-sample the learning data so that the majority class `0` ("no fraud") is not heavily overrepresented. In this case, the ratio of minority class `1` ("fraud") to majority class `0` ("no fraud") is `0.10`:

```
trans_model_xgb.under_sample_dataframe(sampling_strategy=0.10)
```

(25) Split the Transfer Features learning data into a train and a test set. The "orig_key_3"-vector must be "grouped" to avoid data leakage:

```
trans_model_xgb.train_test_split_dataframe(group='orig_key_3')
```

(26) Train the XGB model of the Transfer Model and do cross-validation:

```
trans_model_xgb.train_model_and_cross_validate()
```

(27) Get the cross-validation results for the Transfer Model:

```
trans_model_xgb.get_cv_results()
```

For the Transfer Model cross-validations, all three model results for the test fold of the training set are shown:

Figure 9: Cross-Validation results

Cross-Validation					
roc_auc for RFC		**roc_auc for Logit**		**roc_auc for XGB**	
split0_test	0.8593	split0_test	0.8634	split0_test	0.9122
split1_test	0.8742	split1_test	0.8475	split1_test	0.8883
split2_test	0.8842	split2_test	0.8806	split2_test	0.8875
split3_test	0.8497	split3_test	0.8689	split3_test	0.8846
split4_test	0.8741	split4_test	0.8494	split4_test	0.8732
mean_test	**0.8683**	**mean_test**	**0.8620**	**mean_test**	**0.8892**
std_test	0.0122	std_test	0.0124	std_test	0.0128

(28) In addition to that, the ROC-AUC score for the test set, that was set aside at the beginning of the modelling process, can also be shown:

```
trans_model_xgb.get_test_set_roc_auc_score()
```

The results for the respective model are:

Figure 10: Test set results

Test Set	
roc_auc RFC	**0.8728**
roc_auc Logit	**0.8668**
roc_auc XGB	**0.8879**

(29) The XGB model creates a (namedtuple) "ExplanationInput" object:

```
trans_model_xgb.generate_explanation_input()
```

(30) This "ExplanationInput" object is then passed to a newly created instance of class "Shap":

```
shap_xgb = Shap(explanation_input=trans_model_xgb.explanation_input_object)
```

(31) From the Original Model, we could also generate a (namedtuple) "ExplanationInput" object and also pass it into new instances of class "Shap" to explain the Original Features:

```
# XGB of the Original Model
xgb_model.generate_explanation_input()
shap_orig_xgb = Shap(explanation_input=xgb_model.explanation_input_object)
```

But we do not pursue this further as we only want to explain the Transfer Model.

(32) We first create the Shap-Objects (Explainer- and Explanation-objects):

```
shap_xgb.create_shap_objects()

shap_xgb.calc_global_explanation()
```

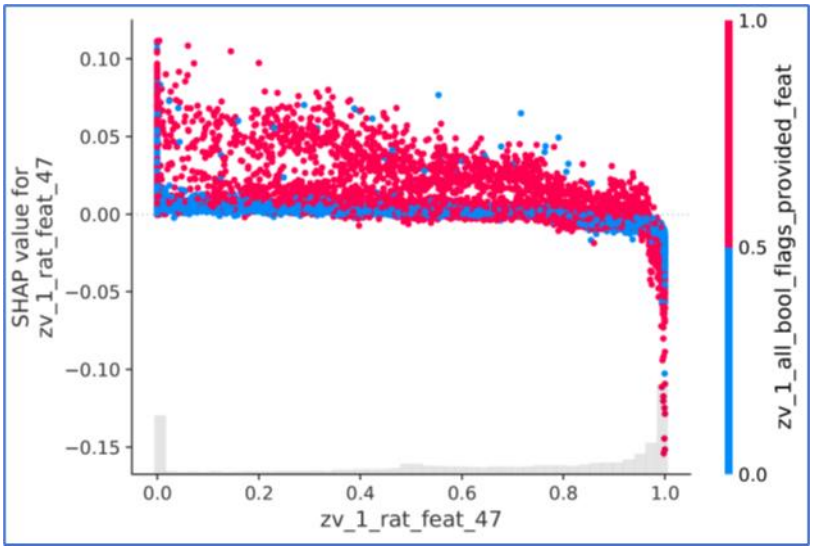
(33) We then can create the global plots (only scatter plot is shown):

```
# XGB - Bar Plot
shap_xgb.plot_global_bars(num_feat_shown=20)

# XGB - Beeswarm plot
shap_xgb.plot_global_beeswarm(num_feat_shown=20)

# XGB - Scatter plot for feature 'zv_1_rat_feat_47'
shap_xgb.plot_global_scatter_for_features(feaure_name_one=
                                          'zv_1_rat_feat_47',
                                          display_only_feat_one = False,
                                          feature_name_two =
                                        'zv_1_all_bool_flags_provided_feat',
                                          save_plot_as_pdf=True)
```

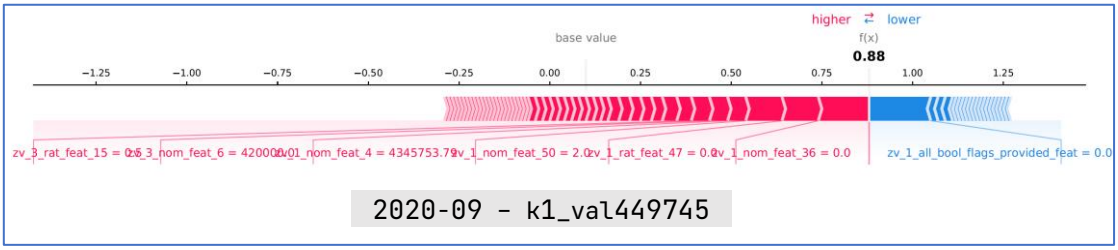
Figure 11: SHAP global scatter plot



(34) And also create a local force plot for a particular borrower:

```
# XGB
shap_xgb.plot_local_force(key_value='k1_val449745',
                          save_plot_as_pdf=True,
                          contribution_threshold=0.0371)
```

Figure 12: SHAP local force plot



In this chapter, I have shown the entire process from an Original Model to a Transfer Model. What follows is an analysis of the Transfer Model and a discussion about explaining model results with feature explanation techniques such as Shapley-values.

3. The Transfer Model

This chapter will be about an analysis of the Transfer Model.

A. The Transfer Model target values

The Transfer Model learning set initially has around 80,000 rows. It is thus much smaller than the monthly data for the Original Features that contain around 520,000 rows. This is because the payment data is not available for all borrowers.
The actual size of the learning data in the training phase depends on two crucial inputs:
Firstly, the number of class 1 ("fraud") samples in the target vector of the Transfer Model. To calculate the target values for the Transfer Model, the "permilles" or the Fraud-Scores of the Original Model (RFC, Logit, XGB) were converted to binary target values. Borrowers with predicted Fraud-Scores in all three Original models at or above a threshold of 90 were classified to belong to class 1, else to class 0. But what are the effects if we chose a Fraud-Score threshold that is different from 90? From the discussion in the first chapter, we know that a threshold that is too low would increase the number of "suspicious" borrowers and raise the false positive rate. If we chose a threshold that is too high, we would have less "suspicious" borrowers but at the expense of a higher false negative rate.

Secondly, the actual size of the training data is then determined by the "resampling" done for the Transfer Model learning data. The learning data must be resampled to reduce the size of the heavily overweighted majority class 0 ("no fraud") relative to the minority class 1 ("fraud"). The desired ratio ("minority-majority-ratio") between the minority and majority class is chosen and passed to the "under_sample_dataframe"-method. Upon execution, this resampling method leaves the samples of the minority class untouched but throws out some majority class samples to get to the desired ratio.
If this minority-majority-ratio is too small, then the training set is heavily skewed towards the majority class. If the minority-majority-ratio is too big, the size of the training data becomes very small.
The following tables with different settings for these parameters illustrate the trade-off:

Figure 13: Transfer Model ROC-AUC scores for minority-majority-ratio: 0.10

ROC_AUC Scores (Ratio: 0.10)								
Original Model	Transfer Model							
	CV-Mean			Test Set			Minority Class to Majority Class: 0.10	Number of class 1 ("fraud") instances in monthly data at or above threshold (total of around 80000)
Fraud-Score Threshold for all 3 models (RFC, Logit, XGB)	RFC	Logit	XGB	RFC	Logit	XGB	Size of Train Set (average of 3 models)	
95.0	0.9077	0.8741	0.9227	0.8910	0.8932	0.9185	6310 rows	816 cases
90.0	0.8683	0.8620	0.8892	0.8728	0.8668	0.8879	16045 rows	2097 cases
85.0	0.8545	0.8672	0.8722	0.8651	0.8574	0.8693	27831 rows	3626 cases
80.0	0.8193	0.8213	0.8265	0.8301	0.8148	0.8240	43659 rows	5676 cases

Figure 14: Transfer Model ROC-AUC score for minority-majority-ratio: 0.25

ROC_AUC Scores (Ratio: 0.25)								
Original Model	Transfer Model							
	CV-Mean			Test Set			Minority Class to Majority Class: 0.25	Number of class 1 ("fraud") instances in monthly data at or above threshold (total of around 80000)
Fraud-Score Threshold for all 3 models (RFC, Logit, XGB)	RFC	Logit	XGB	RFC	Logit	XGB	Size of Train Set (average of 3 models)	
95.0	0.9032	0.8978	0.9007	0.9051	0.8677	0.9190	2192 rows	628 cases
90.0	0.8785	0.8731	0.8917	0.8911	0.8850	0.9047	6731 rows	1929 cases
85.0	0.8615	0.8632	0.8815	0.8685	0.8702	0.8774	11333 rows	3234 cases
80.0	0.8326	0.8320	0.8361	0.8320	0.8308	0.8401	15171 rows	4338 cases

The results in the first table (figure 11) correspond to a minority-majority-ratio that is set to 0.10 whereas in the second table (figure 12) they correspond to a minority-majority-ratio set to 0.25.

In case the Fraud-Score-threshold is set to 90, at a minority-majority-ratio of 0.10 the "under_sample_dataframe"-method leaves 16,045 rows in the training set but only 6,731 rows if the minority-majority-ratio is set to 0.25. The size of the training set though would go up from 6,731 rows to 11,333 rows if the Fraud-Score threshold of the latter would be lowered from 90.0 to 85.0. At this level however, the number of monthly "fraud" cases that appear to be suspicious would be 3,234.

To select an appropriate Fraud-Score threshold and an appropriate minority-majority-ratio, it is tempting to just pick the best ROC-AUC score. But the number of features in the Transfer Model data must also be considered. As stated previously, the Transfer Model consists of close to 300 features. Thus, the "curse of dimensionality" needs to be addressed: as the number of features or dimensions grows, the amount of data that is required to generalize accurately grows exponentially.[14] With the number of features close to 300, I would require the data set to have at least 15,000 rows.

Given the requirements of a low number of "fraud" cases, a high ROC-AUC score and at least 15,000 rows of data for the training set, I would choose a minority-majority-ratio of 0.10 and a Fraud-Score-threshold of 90. This choice is marked in green in the first table.

B. Variance and Bias

The cross-validation results of the Transfer Model for the chosen minority-majority-ratio of 0.10 and a Fraud-Score threshold of 90.0 have been shown previously as have been the ROC-AUC scores for the test sets, that were set aside at the beginning of the modelling process:

Figure 15: Cross-Validation and Test set results

Cross-Validation					
roc_auc for RFC		roc_auc for Logit		roc_auc for XGB	
split0_test	0.8593	split0_test	0.8634	split0_test	0.9122
split1_test	0.8742	split1_test	0.8475	split1_test	0.8883
split2_test	0.8842	split2_test	0.8806	split2_test	0.8875
split3_test	0.8497	split3_test	0.8689	split3_test	0.8846
split4_test	0.8741	split4_test	0.8494	split4_test	0.8732
mean_test	**0.8683**	**mean_test**	**0.8620**	**mean_test**	**0.8892**
std_test	0.0122	std_test	0.0124	std_test	0.0128

Test Set	
roc_auc RFC	**0.8728**
roc_auc Logit	**0.8668**
roc_auc XGB	**0.8879**

[14] (Swapnil-Vishwakarma, 2021)

As discussed previously, "Bias" in Machine Learning is defined as the phenomena of observing results that are systematically prejudiced due to faulty assumptions. The degree of "Bias" can usually be directly read from the ROC-AUC score.
The ROC-AUC scores of the five cross-validation folds for the RFC model range from 0.8497 to 0.8842 and the ROC-AUC score for the RFC test set is 0.8728. All values thus lie close but below the threshold of 0.90 above which I would consider a model to have a "Low Bias".
The same applies to the XGB and Logit models as their values also lie below this threshold.

To improve or to lower the "Bias" of these "Medium Bias" models, in the future it is intended to add more features coming from the balance sheet data of the borrowers.
Not only could they contain additional information directly, but also indirectly in ratio features that are engineered by combining balance sheet and payment data.

Another possibility to lower the "Bias" is to increase the amount of learning data. As mentioned above, the Transfer Model learning data was resampled with the "*under_sample_dataframe*"-method so that the majority class (0 = "no fraud") did not dominate the minority class (1 = "fraud") too much, but with a minority-majority-ratio of 0.10 just enough to address the "curse of dimensionality". But even with this more unbalanced data set, the model's ROC-AUC scores for several program runs, and even different minority-majority-ratios, only changed very minimally and to a negligible extent. The size of the data set thus had almost no effect on the "Bias" of the three models.

A third possibility is to adjust the hyperparameters to improve the "Bias", but this has already been exhausted in the training phase. I would therefore classify the "Bias" of the transfer model to be of "Medium Bias".

As discussed previously, the "Variance" is the amount that the estimate of the model will change if different training data was used. "Low Variance", as I defined it above, allows only for up to 0.05 in difference between the ROC-AUC scores for the different test sets of the training data. The difference between the highest and lowest ROC-AUC scores for the different models in the cross-validation are as follows:

CV-Difference max – min ROC-AUC for RFC: 0.8842 – 0.8497 = **0.0345**
CV-Difference max – min ROC-AUC for Logit: 0.8806 – 0.8474 = **0.0332**
CV-Difference max – min ROC-AUC for XGB: 0.9122 – 0.8732 = **0.0390**
The difference between the mean in the cross-validation and the final test set ROC-AUC scores are:

Difference ROC-AUC CV-mean – test set for RFC: absolute (0.8683 – 0.8728) = **0.0045**
Difference ROC-AUC CV-mean – test set for Logit: absolute (0.8620 – 0.8668) = **0.0048**
Difference ROC-AUC CV-mean – test set for XGB: absolute (0.8892 – 0.8879) = **0.0013**

Both kind of differences are very low and safely within the 0.05-threshold. I would therefore classify the "Variance" of the transfer model to be of "Low Variance".

C. Model Stability

The learning data of the Transfer Model stems from the monthly payment data for September 2020. To check whether the Transfer Model is viable and valid, we not only need to check its "Bias" and "Variance", but also its stability over time. What if the learning data comes from another month? Does the Transfer Model then have similar characteristics and model parameters or are the latter just dependent on the month from which we take the data? To check that, several months would have to be compared with each other. But as this goes beyond the scope of this thesis, I just re-run the entire modelling process for one other month (April 2021) and did the comparison to the month of September 2020.

Here are the general results:

Figure 16: Results for September 2020 and April 2021

September 2020	
Cross-Validation	
roc_auc for RFC	
split0_test	0.8593
split1_test	0.8742
split2_test	0.8842
split3_test	0.8497
split4_test	0.8741
mean_test	**0.8683**
std_test	0.0122
roc_auc for Logit	
split0_test	0.8634
split1_test	0.8475
split2_test	0.8806
split3_test	0.8689
split4_test	0.8494
mean_test	**0.8620**
std_test	0.0124
roc_auc for XGB	
split0_test	0.9122
split1_test	0.8883
split2_test	0.8875
split3_test	0.8846
split4_test	0.8732
mean_test	**0.8892**
std_test	0.0128
Test Set	
roc_auc RFC	**0.8728**
roc_auc Logit	**0.8668**
roc_auc XGB	**0.8879**

April 2021	
Cross-Validation	
roc_auc for RFC	
split0_test	0.8824
split1_test	0.8748
split2_test	0.8640
split3_test	0.8704
split4_test	0.8554
mean_test	**0.8694**
std_test	0.0092
roc_auc for Logit	
split0_test	0.8563
split1_test	0.8527
split2_test	0.8752
split3_test	0.8700
split4_test	0.8751
mean_test	**0.8658**
std_test	0.0095
roc_auc for XGB	
split0_test	0.8902
split1_test	0.8810
split2_test	0.8823
split3_test	0.8802
split4_test	0.8624
mean_test	**0.8792**
std_test	0.0091
Test Set	
roc_auc RFC	**0.8691**
roc_auc Logit	**0.8724**
roc_auc XGB	**0.8826**

The general model results for both data sets, the cross-validation and the tests sets are very similar for both months. The ROC-AUC scores only differ slightly, and their "Bias" and "Variances" are comparable. If we assume that the results are very similar for other months too, it could be argued that the model is stable over time, at least based on these general results.

What is missing is the analysis of the result sources or the feature contributions. Only if similar model features contribute to the model results for the two different months in a similar way, we can claim that the model is indeed stable over time. This question will be discussed in the next chapter which deals with feature explanation.

So far, we could classify our Transfer Model as a "Low Variance, Medium Bias" model with an expected improvement in "Bias" if we added new features from the balance sheet data. This could possibly deteriorate the "Variance" of the model again as there usually is a trade-off between "Variance" and "Bias". The ML process would then need to adjust for that by fine-tuning the hyperparameters or by dropping some features with low explanation power.

4. Feature Explanation

A. Introduction

The goal in our classification task was to find a model that best describes the unknown relationship between a known input feature matrix and a known output target vector. At the end of the modelling process, we get the model results that tell us how well the model can do predictions. But because the results do not tell us how these predictions are made in detail, ML models are often considered black boxes.

To overcome this and to make models more interpretable, there are several approaches that all fall under the terms "model interpretability" or "feature explanation".

1. Choosing interpretable models

The first approach to make ML models interpretable is to only choose those ML models that can be interpreted on basis of its model parameters. In a linear regression model, for instance, the estimated feature coefficients reveal how much a particular feature vector contributed to the final prediction. But the easy interpretability comes at the cost that the model must meet some requirements: it is only applicable if the relationship between its feature matrix and the target vector is linear and is only interpretable if the data is normalized, homoscedastic, not multicollinear, etc.

Other models do not have such requirements. Decision Trees, for instance, can also map non-linear relationships. The feature importances can be expressed as ratios of the number of instances in the different leaves of the Decision Tree and the data may be non-normalized, non-homoscedastic and multicollinear. But the model is unstable in the sense that a few changes in the training set can create completely different trees. If a different feature is selected as the root node, the entire tree structure changes. This instability does not create confidence in the interpretability of the model.

But the biggest disadvantage of such models is that their interpretability is model-specific and that the parameters of different models cannot be compared with each other. The feature coefficients of a Linear Regression model, for instance, must be interpreted very differently than the feature coefficients of a Logistic Regression model.

What is needed is a model-agnostic approach to overcome this. The great advantage of model-agnostic methods over model-specific ones is their flexibility. Any ML model can be used when such a generic interpretation method can be applied to this model.[15]

2. Partial Dependence Plots (PDP)

PDP is such a model-agnostic feature importance technique. The partial dependence plot shows the marginal effect that one or two features have on the predicted outcome of a ML model.[16] A partial dependence plot can show whether the relationship between the target and a feature is linear, monotonic, or more complex. For example, when applied to a linear regression model, partial dependence plots always show a linear relationship. The PDP also show how strong the relationship between the respective feature and the target vector is. This can be read from the steepness/ascent of the line in the PDP assuming the relationship between the feature and the target vector is linear.

[15] (Molnar, 2019)

[16] (Hastie, et al., 2009)

One disadvantage of PDP is that it only calculates the partial dependence of a feature based on all feature values (global) but not for a single (local) data instance individually. Fraud investigators would thus not be able to get the feature contributions for a particular borrower. Another disadvantage of PDP is its assumption that features are not correlated with each other whereas in practice they often are.

3. Permutation Importance

Another approach to measure feature importance is the "Permutation Importance" method. The idea is quite simple: After a model is fitted, the values in a feature vector or feature column are randomly shuffled. The expectation is that the shuffling will cause less accurate predictions and thus worse model results. The shuffling is done for every feature individually and the most important features are those for which the model results deteriorate the most.

But the disadvantages of this method are the same as with PDP: Its global scale and its assumption that features are independent of each other. The shuffling of the features data produces unlikely data rows when two or more features are correlated. For instance: If the data row contains physical measures such as a person's height and weight, a data row with a (shuffled) child's height of 2,10 meter and a (unshuffled) child's weight of 10 kg is physically highly unlikely. Nevertheless, the unreal data row contributes to the decreased accuracy and feature importance measurement. Moreover, correlation can wrongly decrease the importance of an important feature if this feature is highly correlated with another feature as the importance is then "shared" between the two. In a Decision Tree, for instance, the algorithm might switch back and forth between two important but correlated features when selecting the top nodes which decreases the feature importance of both. And as the permutation is done for the entire feature column, a permutation importance measure and an analysis for a single data row is not possible.

4. Feature Interaction

As discussed above, correlation between features is a concern when trying to calculate feature importances. But even more important to it is another measure: Feature interaction. Whereas feature correlation only measures the co-movement of feature values independent of the target vector, feature interaction measures the co-movement-effect of two or more features on the target vector.

When features interact with each other in a prediction model, the prediction result is not just the sum of the individual feature contributions but must also account for the interaction effect between the features. The interaction between two features so defined is the rise or fall of the prediction value if feature values are changed after accounting for the individual feature effects. To demonstrate this, I here cite the example by Christoph Molnar[17]:

Suppose we have the following extract of house data that contains the features "Location" and "Size" (of the house) and try to explain the predictions of house prices:

[17] (Molnar, 2019)

Figure 17: Interaction effect

NO Interaction Effect		
Location	Size	Prediction
good	big	300,000
good	small	200,000
bad	big	250,000
bad	small	150,000

Interaction Effect		
Location	Size	Prediction
good	big	400,000
good	small	200,000
bad	big	250,000
bad	small	150,000

We can decompose the predictions into a constant term, an effect for the "Location" feature, an effect for the "Size" feature and an interaction effect. We here define the constant term as the prediction value if the values for the features "Location" and "Size" are at the low level of "bad" and "small" respectively. In both tables the constant term is 150,000.

Considerations for the left table ("NO Interaction Effect"):
If one chooses a big house instead of a small one, one must pay an additional 100,000 which is the "Size" effect. If one chooses a good location instead of a bad location, one must pay an additional 50,000. If one chooses a good location and a big house, the prediction is just the sum of the constant term and the two effects ("Size", "Location"). There is no interaction effect in the left table.

Considerations for the right table ("Interaction Effect"):
If one chooses a big house instead of a small one in a bad location, the "Size" effect is the same as before (100,000). If one chooses a good location instead of a bad one and a small house, the "Location" effect is also the same as before (50,000). But if one chooses a big house in a good location in the right table, the prediction is NOT just the sum of the constant term and the two effects ("Size", "Location") but also contains an additional interaction effect of 100,000. This appears reasonable, as one would expect unproportionally higher prices for more living space if the house is in a better living area. And the logic might apply to other business domains as well.

5. Feature Contribution

From the right table of the last example, we have seen that there is an interaction effect between the features "Size" and "Location". But what if we are forced to allocate this interaction effect just on the two features? Let us assume that the initial state in the right table ("Interaction Effect") of the last example is bad location and small size. The predicted price for that combination is 150,000. Now we change our mind and first change bad location to good location and afterwards also change from a small house to a big house: The predicted price first jumps from 150,000 to 200,000 and we would attribute the difference of 50,000 to the "Location" feature. Afterwards, it jumps from 200,000 to 400,000 and we would attribute the increase of 200,000 to the "Size" feature.

In an alternative trial and also starting from an initial state of bad location and small size, we would now first change from a small house to a big house and afterwards from a bad location to a good location. The predicted price now first jumps from 150,000 to 250,000 and we would attribute the difference of 100,000 to the "Size" feature. Afterwards, it jumps from 250,000 to 400,000 and we would attribute the increase of 150,000 to the "Location" feature.

If we compare the marginal contributions from the two trials for each of the two features "Size" and "Location", we realize that the effect is different and dependent on the order of

the change. In the first trial, the marginal contribution of the "Size" feature was 200,000 but was only 100,000 in the second trial. In the first trial, the marginal contribution of the "Location" feature was only 50,000 but was 150,000 in the second trial.

One could now calculate the average marginal contribution for both features by taking the mean of the two trials: The average marginal contribution for the feature "Size" is (200,000 + 100,000)/2 = 150,000 and it varies from 100,000 to 200,000. The average marginal contribution for the feature "Location" is (50,000 + 150,000)/2 = 100,000 and it varies from 50,000 to 150,000.

In contrast to that, in our "NO Interaction Effect"-table, the respective marginal contributions for the features "Size" and "Location" are 100,000 and 50,000 respectively and they do not vary. We can say that, ceteris paribus, marginal contributions have a higher variance if there is an interaction effect between the feature values.

In our example, we divided the interaction effect of 100,000 equally and added the dividends to both features. In our "NO Interaction Effect"-table, the respective marginal contributions for the features "Size" and "Location" were 100,000 and 50,000 respectively and this changed to 150,000 and 100,000 after accounting for the interaction effect.

From this example, two things become clear: First, the allocation order of feature contributions is crucial. Second, the calculated marginal contribution value for any feature has a higher variance if its interaction (effect) with the other features is larger.

Both aspects must be taken into consideration, and this is where Shapley values come into play.

B. Shapley Values and SHAP

Before going into Shapley values, I first want to define two terms that I use. "Feature value" is the numerical or categorical value of a feature and instance. "Shapley value" is the feature contribution to a prediction of a ML model.

1. Shapley Values

Shapley values initially were named after Lloyd Shapley, an American mathematician, and Nobel Prize-winning economist.[18] Shapley tried to answer a core question in game theory: what is the fairest way to share the collective payoff in a coalition or group with multiple players of different skill sets?
One way to approach this is to imagine that each player joins the group one after another and so sequentially adds to the collective payout. His or her marginal contribution would then just be his or her added value to the collective sum. For instance: if player A was the first to join the group, with a payoff of 5, and player B joined to bring the payoff to 9, and later player C joined to bring the payoff to 11, then the players' respective marginal contributions would be 5, 4 and 2.

But there is a problem with this approach. If player C and B have very similar skill sets, it might be that player C would have a higher marginal contribution if he or she joined the group before player B, because she or he would be the first one to provide his or her overlapping skill set. When player B then joined after player C, his or her marginal contribution would be

[18] (Shapley, 1953)

lower than before, and we could imagine different marginal contributions for players A, B and C of 5,1 and 5 for instance.

This is a similar problem to the one we had in our discussion in the prediction of house prices. There we realized that the order of feature contributions is crucial if there is an interaction effect between the features. Here we realize that the order the players join the group is crucial for their marginal contribution if their skill set is similar.

This problem is what lead to the formulation of Shapley values, which can be understood as "finding each player's (or feature's) marginal contribution, averaged over every possible sequence in which the players (or features) could have been added to the group (or prediction result)". So, for the example above, one would simulate all possible arrival sequences of the players to the game:

Figure 18: Possible arrival sequences of 3 players

A -> B -> C
A -> C -> B
B -> C -> A
B -> A -> C
C -> A -> B
C -> B -> A

For each of the six sequences, the marginal contributions for each player would then be written down. The average of all six contributions for each player is then their respective Shapley value.

This is like the example in our house price prediction case from the last chapter. There we calculated the marginal contribution average for the two trials because they differed for each trial because of the interaction effect between the two features "Size" and "Location". And there, we had two possible arrival sequences:

Figure 19: Possible arrival sequences for 2 features

Size -> Location
Location -> Size

The number of possible arrival sequences or orders how the players (features) can arrive to the game (prediction result) is dependent on the number of players (features) and can be calculated as "factorial of number of players (features)". The number of possible sequences was six (3! = 6) for our players example and two (2! = 2) for our house price example.

Shapley values, as applied here to feature importance, are defined as the sequential impact on the model's output of observing each input feature's value, averaged over all possible feature orderings.[19]

[19] (Lundberg, et al., 11 May 2019)

The calculation of Shapley values always starts with a concrete question. For instance:

Figure 20: Prediction for one instance in a 3-feature model

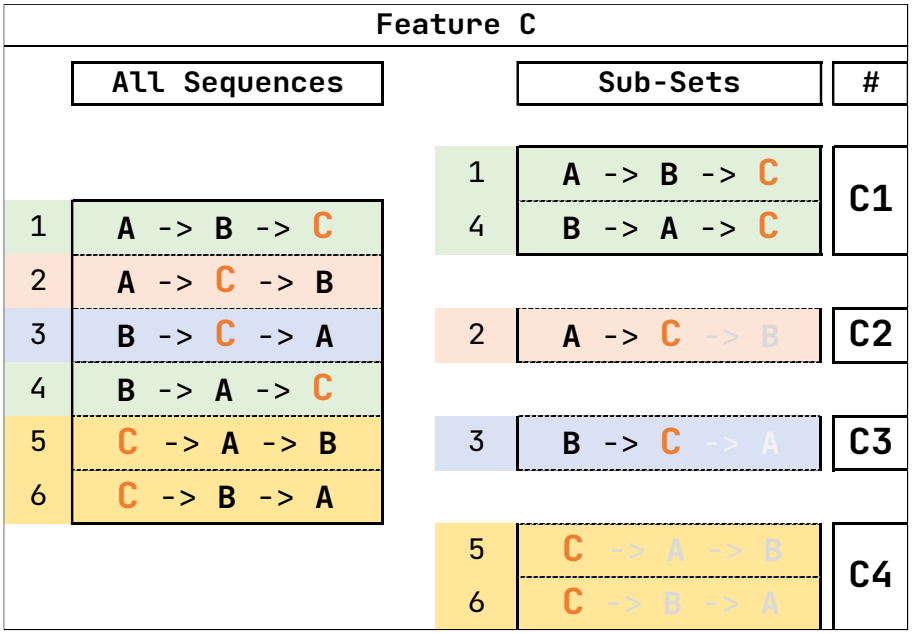
	Existing (Trained) 3-Feature-Model			
	Features			Model
	A	B	C	Predicts
single data instance	6	5	8	10

A ML model with the features A, B and C was trained and a prediction was made for a single data instance. The feature values for this data instance are 6, 5 and 8 for the respective features A, B and C and the ML model predicts a value of 10. How can this value of 10 be attributed to each feature and how can we measure their successive contributions?

In the classic "Shapley values"-approach[20] this is done by dividing up all arrival sequences into subsets (or "coalitions" as they are called in Game Theory[21]). As these subsets contain redundant information that is also contained in other subsets, the calculation procedure can be simplified. To demonstrate this, let us go back to our player's example and let us assume that the previous players are now the three features of the single data instance for which we want to calculate the respective feature contributions to the prediction value of 10.

To calculate the contribution of feature C for all its possible positions in all of the six possible arrival sequences, four subsets are built from the "All Sequences"-set:

Figure 21: Arrival sequences of 3 features and subsets from it: Feature C



In subset #C1, feature C arrives at the last/third position, in subset #C2 and #C3 at the second position and in subset #C4, feature C arrives at the first position.

[20] (Lundberg, et al., 25 Nov 2017)

[21] https://en.wikipedia.org/wiki/Cooperative_game_theory

For subset #C1, we would be required to build one new ML model that must first be trained and has only the two features A and B. Beside that new 2-feature-model, we would also use our existing 3-feature-model. We would then put in the values 6 and 5 for feature A and B into the new 2-feature-model and the values 6, 5 and 8 into our existing 3-feature model and repeatedly make several predictions for both ML models to account for their variance (in making predictions). Each time we would calculate the prediction difference of the two models and then take the average of all the prediction differences:[22]

Figure 22: Prediction results of a 3-feature-model (A, B, C) and a 2-feature-model (A, B)

#C1	Existing (Trained) 3-Feature-Model					NEW (Trained) 2-Feature-Model		
	Features			Model		Features		Model
	A	B	C	Predicts		A	B	Predicts
single data instance	6	5	8	10	single data instance	6	5	7
subset #C1 Contribution of feature C = 10 - 7 =				3				

=> DO this several times and take the average of the differences

This (average) difference (in the above example, the (average) difference is 3) represents the marginal contribution of feature C to enter the model at the THIRD/LAST position. I will call it the "**subset #C1 contribution**".

For subset #C2 (and #C3), this process must be repeated, but here we would need to train two new models (for each of the two subsets): One new model with feature A only (B only) and another new model with the features A and C (B and C). We would then again put the feature values of the single data instance into the two models respectively and (repeatedly) make predictions. Afterwards we calculate the (average) difference in the predictions between the new 2-feature model and the new 1-feature-model. This (average) difference represents the contribution of feature C to enter the model at the SECOND position. I will call it the "**subset #C2 contribution**" (and "**subset #3 contribution**"):

Figure 23: Prediction results of a 2-feature-models (A, C) and (B, C) and 1-feature-models (A) and (B)

#C2	New (Trained) 2-Feature-Model				NEW (Trained) 1-Feature-Model		
	Features		Model		Features	Model
	A	C	Predicts		A	Predicts
single data instance	6	8	5	single data instance	6	8
subset #C2 Contribution of feature C = 5 - 8 =						-3.0

=> DO this several times and take the average of the differences

[22] Note: In this and all subsequent tables and graphs, I use purely random numbers for the feature values that have no meaning whatsoever and are not part of a real ML model. The focus here is only on the prediction values and their differences.

#C3	New (Trained) 2-Feature-Model				NEW (Trained) 1-Feature-Model	
	Features		Model Predicts		Features	Model Predicts
	B	C			B	
single data instance	5	8	7	single data instance	5	5.5
subset #C3 Contribution of feature C = 7 - 5.5 =						1.5
=> DO this several times and take the average of the differences						

For subset #C4, this process must also be repeated but is simplified by the fact, that we only need to run one new model with the feature vector of feature C against the target vector. The contribution of feature C to enter the model at the FIRST position is then just the (average) prediction. I will call this average "**subset #C4 contribution**":

Figure 24: Prediction result of a 1-feature-model (C)

#C4	New (Trained) 1-Feature-Model	
	Features	Model Predicts
	C	
single data instance	8	4
subset #C4 Contribution of feature C =		4
=> DO this several times and take the average		

These respective differences of averages in the paper of Lundberg & Lee[23] were expressed as:

$$\left[f_{S\cup\{i\}}(x_{S\cup\{i\}}) - f_S(x_S)\right]$$

But how is the average of all four marginal contributions for our four subsets (i.e., the Shapley Values) calculated? How are these four subsets weighted to arrive at the Shapley value? In Lundberg & Lee[24], this is done by weighing the four individual averages with the following formula:

Figure 25: Shapley value weights

$$\frac{|S|!(|F| - |S| - 1)!}{|F|!}$$

For the four subsets in our example, S! corresponds to the number of ways that all previous features could have been added leading up to the position of feature C in the sequence. Since |F| is the total number of features in the coalition, |F| — |S| — 1 corresponds to the number of features left to be added after the position of feature C.

[23] (Lundberg, et al., 25 Nov 2017)

[24] (Lundberg, et al., 25 Nov 2017)

Applied to the subset #C1, S! is the number of ways that features A and B could have been added leading up to the last position of feature C in the sequence. This is: 2! = 2. The total number of features in our example is 3 and thus the above formula for the weight of the "subset #C1 contribution" becomes:

$$\frac{|S|!\,(|F|-|S|-1)!}{|F|!} = \frac{2!\,(3-2-1)!}{3!} = \frac{2!\,0!}{3!} = \frac{2}{6}$$

Accordingly, the respective weights of the "subset #C2 contribution" and "subset #C3 contribution" are:

$$\frac{|S|!\,(|F|-|S|-1)!}{|F|!} = \frac{1!\,(3-1-1)!}{3!} = \frac{1!\,1!}{3!} = \frac{1}{6}$$

And the weight for "subset #C4 contribution" is:

$$\frac{|S|!\,(|F|-|S|-1)!}{|F|!} = \frac{0!\,(3-0-1)!}{3!} = \frac{0!\,2!}{3!} = \frac{2}{6}$$

The weights of our four subsets correspond to the number of rows in each subset in the above picture.

The four weights are then multiplied with the four contributions of the subsets to arrive at the weighted average of all marginal contributions to the prediction for feature C, i.e., the Shapley value. This is expressed in the following formula for the Shapley values by Lundberg & Lee[25]:

Figure 26: Shapley value formula

$$\phi_i = \sum_{S \subseteq F \setminus \{i\}} \frac{|S|!(|F|-|S|-1)!}{|F|!} \left[f_{S\cup\{i\}}(x_{S\cup\{i\}}) - f_S(x_S) \right].$$

In dividing up the entire set of 6 sequences into 4 subsets, we reduce the number of calculations from six (rows) to four (subsets). But why can we package rows 1 and 4 in our "Feature C" example into a subset and do the contribution calculation for that subset only once, for instance?

Let us assume that we only do the calculations for the sequence in row 4 of the above table "Feature C". To calculate the marginal contribution of feature C for row 4 only, we would first let feature B enter the model, afterwards feature A and then take the (average of the) difference in their predictions. But to arrive at the SAME average, we could also have taken row 1. From the perspective of feature C, it does not matter which of the two features A and B first enters the model. The marginal contribution of feature C in row 1 is the same as in row 4. The same logic applies to row 5 and 6 that we have grouped into subset #4: From the perspective of feature C, it does not matter if feature A or B joins next after feature C has already entered the model. For this reason, we must do the calculation for the contribution only once (in our case for row 4) and then weight this contribution by the number of occurrences in the set. What is important to note for later considerations is that with an

[25] (Lundberg, et al., 25 Nov 2017)

increasing number of features, the number of such "equal" subsets or "coalitions" also increase which later simplifies the calculation process. So now we have calculated the 4 marginal contributions and the respective weights for these contributions. We thus could calculate the Shapley value for feature C:

Figure 27: Shapley value of feature C

Shapley value of feature C				
Subset	subset #C1	subset #C2	subset #C3	subset #C4
Marginal Contribution	3.0	-3.0	1.5	4.0
Weight	1/3	1/6	1/6	1/3
Marginal Contribution x Weight	1.00	-0.50	0.25	1.33
	Sum of weighted contributions:			2.08

For our single data instance, the feature C contribution to the prediction of 10 is 2.08. If we did the calculations for the features A and B accordingly, we would have all our Shapley values and thus could calculate the contribution of each feature for our single data instance to the model prediction of 10.

But there are several practical problems with this approach: In our little example above, we would have to train six (!) new ML models in addition to the existing ML model. But usually, we do not want to run additional ML models but use the existing ML model that we have already trained before.

The solution to this problem is that we take the existing model and only "PRETEND" that features are successively added (or omitted).

In this solution, a 0/1-coalition-vector is introduced, where 1 means that the feature is "**present**" and 0 means that the feature is "**absent**". Applied to our example from above, the coalition-vector [1,1,0], for instance, would mean that feature A and B are present, but feature C is absent. This coalition-vector corresponds to our new 2-feature-model in subset #C1.

Figure 28: Feature C sub-set #C1

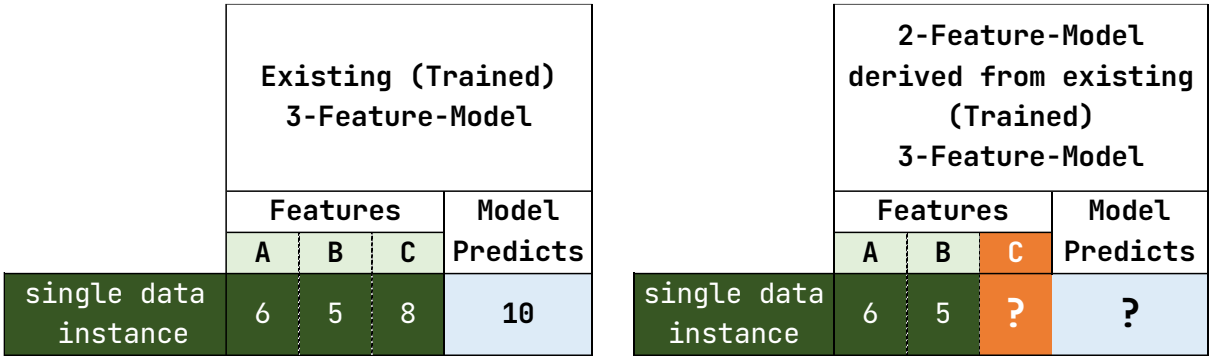
Feature C		
	Sub-Sets	#
1	A -> B -> C	C1
4	B -> A -> C	

To recall: This new 2-feature-model was required to make a prediction and calculate the difference to the prediction of the 3-feature-model. The difference would be the prediction contribution of feature C in the 3-feature-model.

But how can we simulate the new 2-feature-model (features A and B) within our existing 3-feature-model? We cannot just put "Absent" or "Nothing" into our existing model as

placeholder for feature C values because the existing 3-feature-model expects 3 feature vectors with real numbers to make predictions:

Figure 29: From a 3-feature-model to a 2-feature-model?



	Existing (Trained) 3-Feature-Model			
	Features			Model Predicts
	A	B	C	
single data instance	6	5	8	10

	2-Feature-Model derived from existing (Trained) 3-Feature-Model			
	Features			Model Predicts
	A	B	C	
single data instance	6	5	?	?

The solution is to put in random feature values from the training set for the respective feature(s) that is (are) missing.

For our required 2-feature-model in subset #C1, for instance, the placeholder for our feature C value would just be any random feature C value from the training set. If we put a random feature C value into the single data row as "placeholder" for the feature C (value) to be "absent", the "pretended" 2-feature-model now can make a prediction for this single data row. We stick with this data row and repeat the sampling process for the feature C value to be "inserted" as placeholder several times.

The expected result of this sampling exercise is to get a (single) number that is the average model prediction for this data row/single data instance:

Figure 30: Sampling of missing or absent feature values

	2-Feature-Model derived from existing (Trained) 3-Feature-Model			
	Features			Model Predicts
	A	B	C	
single data instance	6	5	Sampling of random feature C values from training set	Average prediction for all feature C values put in

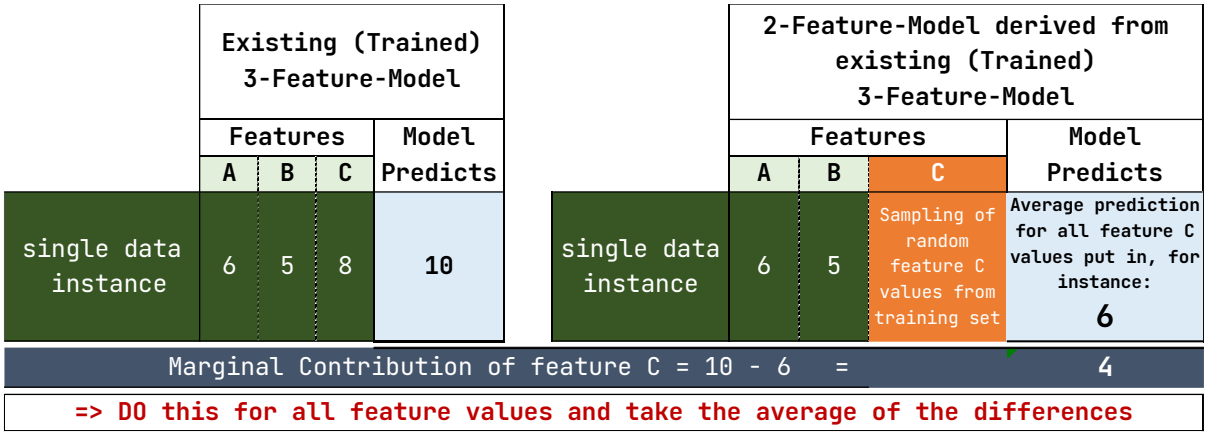
If we repeated the sampling process until all feature C values from all the rows of the training set have been put in and all predictions have been made, we could finally calculate an exact average of all those predictions for this particular data row. Done for all other subsets, this would later also get us the exact Shapley values. But as this process is computationally very expensive, in practice the sampling here is "approximated" by sampling only some but not all rows from the training set. The part of the training set that is used for sampling is called "background dataset" in the SHAP Python library (see below).

But can't we just take a shortcut here and take the average of all feature C values in the training set as a single entry and then make a prediction from it? Shouldn't we then also get the same exact prediction as with the process of repeatedly sampling the entire training set and take the predictions average? No! Because of the (potential) non-linearity of our ML model and the interaction effects, the frequency distribution of the predictions is not known. To enter just the average of all feature C values as "placeholder" of the "absent" feature C

value thus will not (necessarily) get us the same and exact prediction as before. This is the reason why we here would need to do the sampling for all the data rows in the training set if we later wanted to calculate the "exact" Shapley values.

After we have done this, we can then calculate the difference between our 3-feature-model prediction and our "derived" 2-feature-model prediction (i.e., the average of all predictions as just described) to arrive at the "subset #C1 contribution":

Figure 31: Sampling of missing values for feature C



At first glance, it seems counterintuitive to put in sampled feature C values to simulate that feature C is "absent". But what are we trying to do here? For our particular data instance, we want to know the specific effect to have a feature C value of 8 instead of a feature C value of "absent". But what is the best value guess or numerical approximation to "feature C is absent"?

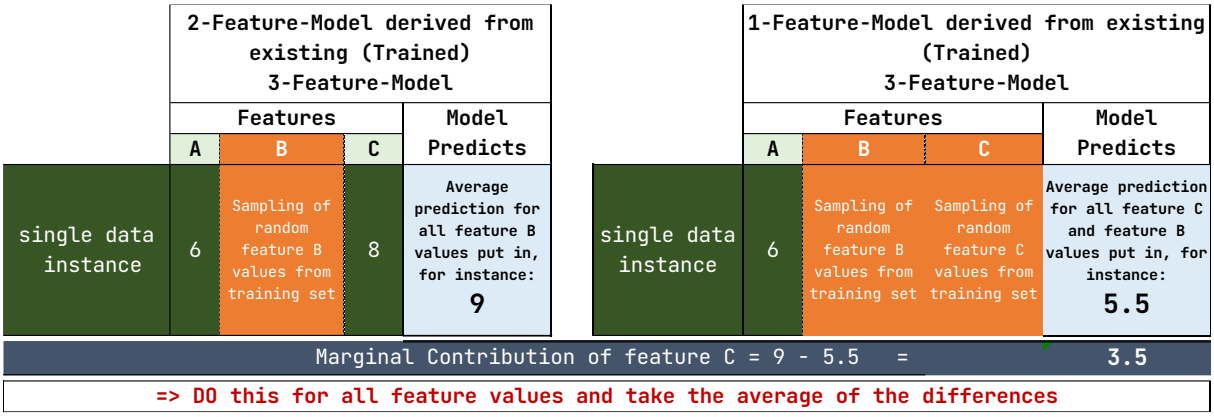
Isn't the 2-feature-model all about simulating the state of a "neutral base" (in respect to feature C) from which individual data instances can deviate? If we see it in this context, we could re-formulate our goal from above: we want to know the specific effect to have a feature C value of 8 instead of a "feature C base value for the entire data population". The "entire data population" in the SHAP library would just be the "background data set", a sample set of all data instances.

In this context, we can use the approach described above and this is also the approach used by Lundberg and Lee.[26]

To calculate the "subset #C2 contribution", we would need to create and train two new models: a 1-feature-model with feature A (coalition-vector: [1,0,0]) and a 2-feature-model with features A and C (coalition-vector: [1,0,1]).

[26] (Lundberg, et al., 25 Nov 2017)

Figure 32: Sampling of missing values for features B, C



Here we would not only need to sample feature C values but also feature B values from the training set.

But independent whether we sample all feature values from the training set or only part of it, here we must distinguish between "conditional sampling" and "marginal sampling". "Conditional sampling" takes the dependency structure of the two features - in our example the features B and C - into account whereas "Marginal Sampling" does not.

The difference in the predictions of our 2-feature-model and our 1-feature-model here would be a what Lundberg and Lee call "Conditional Expectation" if the sampling method was "conditional" and a "Marginal Expectation" if the sampling method was "marginal". A "Conditional Expectation" is described by Lundberg and Lee as the "expected value of the function conditioned on a subset S of the input features"[27]:

$$f_x(z') = f(h_x(z')) = E[f(z) \mid z_S]$$

where z' is the (reduced) input feature set.
In this definition, $\mathbf{h_x(z')}$ represents the mapping function, that for our example from above, would map samples of feature B values to samples of feature C values.
In the SHAP Python library later used, "Conditional Expectation" is used in the TreeSHAP method whereas "Marginal Expectation" is used in the KernelSHAP method.
KernelSHAP thus ignores the dependency structure between present and absent features and thus suffers from the same problem as all permutation-based interpretation methods: The estimation puts too much weight on unlikely instances.[28]

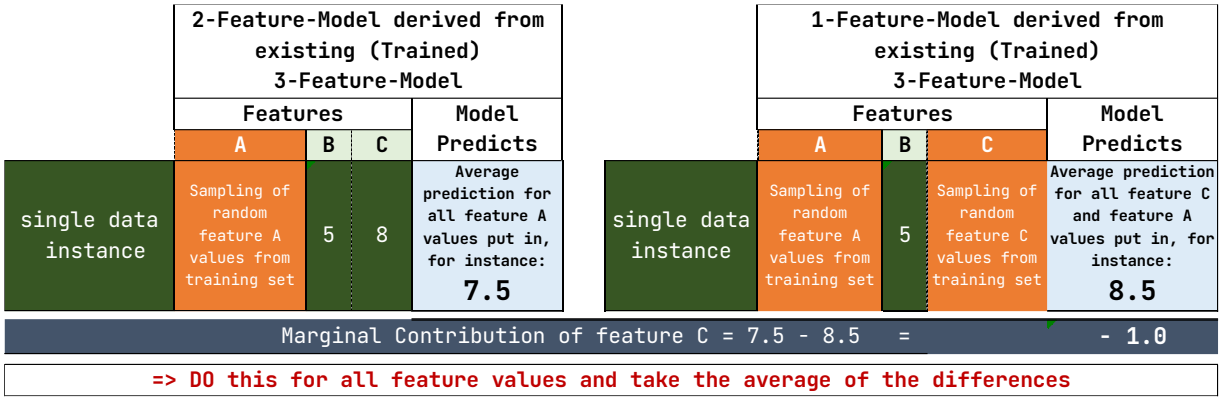
Back to our 2-feature-model and 1-feature-model to calculate the "subset #C2 contribution": For each feature B sample, we would need to sample all feature C samples as we intend to calculate the feature C Shapley value here. The number of all samplings for our single data instance would be the product of the number of feature C samples times the number of feature B samples. Sampling here, again, means inserting the respective sample value(s) and do a model prediction. The average of all prediction differences would be the "subset #C2 contribution".

The process to calculate the "subset #C3 contribution" with a coalition vector of [0,1,1]is very similar, but instead of features B and C, we would sample features A and C:

[27] (Lundberg, et al., 7 Mar 2019)

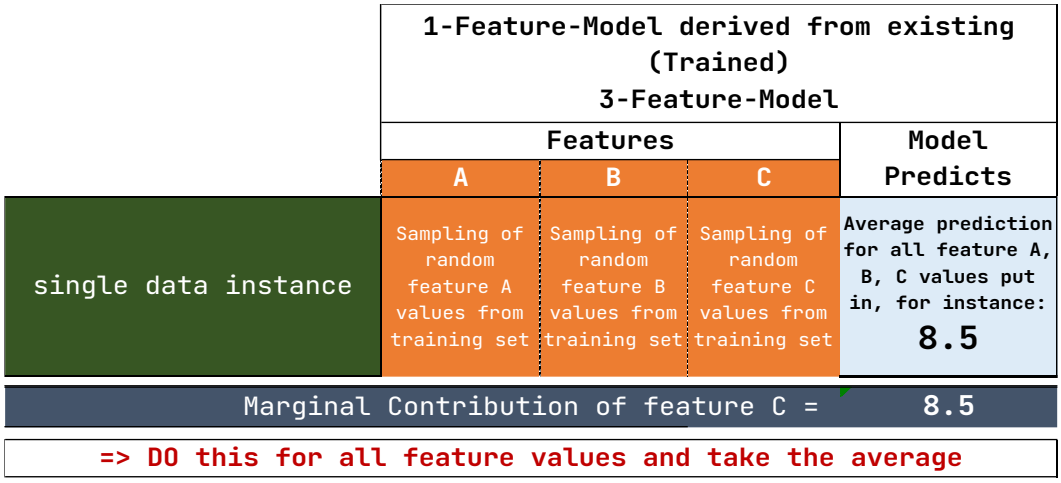
[28] (Molnar, 2019)

Figure 33: Sampling of missing values for features A, C



For the calculation of the "subset #C4 contribution", theoretically, we would deduct the predictions of 1-feature-model from the predictions of a 0-feature-model. But a 0-feature-model does not exist (or we can say its prediction is 0). So, the calculation of the "subset #C4 contribution" with a coalition vector of [0,0,1]is simplified by the fact that we only need one 1-factor-model. But now we need to sample the values for all three features A, B, C:

Figure 34: Sampling of missing values for features A, B, C



What do we get when we calculate this 1-feature-model for feature C? We get the average prediction if all features are sampled. This is the so called "base value" of feature C or "base value #C4". But is it just the base value for feature C?

The weight for each of the three "subset contributions #4" is:

$$\frac{|S|!\,(|F|-|S|-1)!}{|F|!} = \frac{0!\,(number\ of\ features-0-1)!}{number\ of\ features!}$$

which is equal to:

$$\frac{1}{number\ of\ features}$$

So, the "base value" for our single data instance would be:

$$base\ value = \frac{base\ value\ \#A4}{3} + \frac{base\ value\ \#B4}{3} + \frac{base\ value\ \#C4}{3}$$

But the process to calculate the "base value #A4", "base value #B4" and "base value #C4" is the same for all three features because the values for all three features are sampled each time. Therefore, it must be that:

$$base\ value = base\ value\ \#A4 = base\ value\ \#B4 = base\ value\ \#C4$$

So, the just calculated base value for feature C is the base value for our single data instance. This "base value" later is used in SHAP "force plots" to mark the average SHAP values of the background data set.

The number of samplings for our single data instance would be the product of the number of feature B samples **times** the number of feature A samples **times** the number of <u>all</u> feature C samples. And again, sampling here means inserting the respective sample values, do the model predictions and calculate the average of all predictions. This average would be the "subset #C4 contribution".

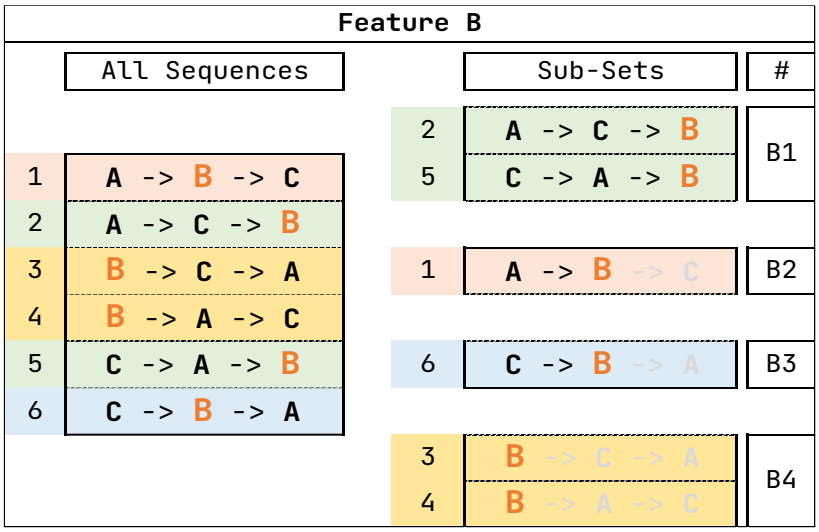
With the coalition-vectors, our Shapley value formula to calculate the Shapley value from above would turn into[29]:

$$\phi_i(f,x) = \sum_{z' \subseteq x'} \frac{|z'|!(M-|z'|-1)!}{M!}\left[f_x(z') - f_x(z' \setminus i)\right]$$

where the last term is the difference of the "expected value of the function" and the "expected value of the function without the feature vector for which we want to calculate the contribution".

This process must then be repeated to calculate the Shapley values for the features A and B. In our example though, this is simplified by the fact that some required 2-feature-model results and 1-feature-model results have already been calculated in the process to calculate the Shapley value for feature C. For instance, if we wanted to calculate the Shapley value for feature B, ...

Figure 35: Arrival sequences of 3 features and subsets from it: Feature B



..., for the "subset #B1", we would need to calculate a 2-feature-model with features A and C.

[29] (Lundberg, et al., 25 Nov 2017)

But the results of this 2-feature-model already exist because it was calculated for the subset #C2 in the process to calculate the Shapley value for feature C (see above). Important to note is that we need to use the same data rows or instances as before if we want to make use of this "redundancy" of already existent results. We would also need to use the same sampling values for the respective feature(s) that is (are) missing. This useful "redundancy" would increase with the number of features and can later be used to reduce the computation complexity.

Shapley values, proved by Lundberg & Lee, is the only explanation method available that meets the requirements of "Local Accuracy", "Missingness" and "Consistency" [30].

Once the Shapley values in our example for all three features A, B and C are calculated, they translate our ML model to an additive linear explanation model, which is an interpretable approximation of the original ML model:

$$f(x) = g(x') = \phi_0 + \sum_{i=1}^{M} \phi_i x'_i$$

In our 3-feature-model example, the formula concretizes to:

$$\text{Prediction} = g(x') = \phi_0 + \phi_A x'_A + \phi_B x'_B + \phi_C x'_C$$

where $\phi x'$ are the three Shapley values for the respective features A, B and C for our single data instance plus an "approximation error"-term $\phi 0$.
The ML model prediction of our single data instance can so be approximated by adding up the respective Shapley values.

2. SHAP (SHapley Additive exPlanation)

The major problem of Shapley values is its computational complexity. This complexity increases exponentially with an increasing number of features. If not only three features are used as in our example above, but hundreds or thousands as is usually the case, then the computation task becomes infeasible. In a tree-based ML model such as Random Forest, for instance, the complexity to calculate Shapley values in big-O-notation is of $O(TLM2^M)$, where M is the number of input features, L the number of leaves and T the number of trees.[31] Lundberg et al[32] addressed this problem and developed several algorithms and methods that reduce the complexity of calculating the Shapley values for such models. KernelSHAP and TreeSHAP are two of those.

[30] (Lundberg, et al., 25 Nov 2017)
[31] (Lundberg, et al., 11 May 2019)
[32] (Lundberg, et al., 11 May 2019)

a) KernelSHAP

KernelSHAP is a combination of linear LIME and Shapley values. LIME stands for Local Interpretable Model-agnostic Explanations and belongs to the class of surrogate models. Surrogate models are trained to approximate the predictions of an underlying ML model.[33]

The calculation of KernelSHAP values in a simplified form, follows a 5-step process:

1) Get the required feature coalitions for the features in the ML model
2) Sample the feature values of "absent" features with some random values from the background data set (which is a sampled set of the training set)
3) Make predictions for the selected data rows. Afterwards, you have tabular data with a matrix of feature values (**X**) and a result vector with the predictions ($\mathbf{y}_{\mathbf{pred}}$)
4) Train a new linear (and thus explainable) regression model (in this case linear LIME) with **X** as the dependent variables and $\mathbf{y}_{\mathbf{pred}}$ as the target variable
5) The regression coefficients of your new linear regression model are your SHAP values!

To demonstrate this, I use our 3-feature-model from above:

Figure 36: Kernel SHAP procedure 1

				Features			Model (Average) Predicts
Coaltion	M	z'	Number of rows	A	B	C	
[0,0,0]	3	0	#feat A samples * #feat B samples	sampled value(s)	sampled value(s)	sampled value(s)	prediction values
[0,0,1]	3	1	#feat A samples * #feat B samples	sampled value(s)	sampled value(s)	8	prediction values
[0,1,0]	3	1	#feat A samples * #feat C samples	sampled value(s)	5	sampled value(s)	prediction values
[0,1,1]	3	2	#feat A samples	sampled value(s)	5	8	prediction values
[1,0,0]	3	1	#feat B samples * #feat C samples	6	sampled value(s)	sampled value(s)	prediction values
[1,0,1]	3	2	#feat B samples	6	sampled value(s)	8	prediction values
[1,1,0]	3	2	#feat C samples	6	5	sampled value(s)	prediction values
[1,1,1]	3	3	1	6	5	8	prediction values

In Step 1, we just get the possible coalitions for our 3-feature-model which has 8 coalitions. In Step 2, we need to sample values for the absent features from the background data set as "placeholders" for the missing feature values. The sampling is done not just once, but several times. **We so get not only 8 data rows as in our picture above, but as many as we have sampled our missing feature values (above: column "Number of rows").**

In Step 3, we make predictions with our existing ML model for all coalitions. In the process described before, we would now calculate the (average) differences between our n-feature-models and the n-1-feature-models and weigh these (average) differences with the Shapley weights accordingly.

[33] (Molnar, 2019)

But not in KernelSHAP:

Figure 37: KernelSHAP procedure 2

				Features			Model (Average) Predicts	Weights $\pi_{x'}(z') =$
Coaltion	M	z'	Number of rows	A	B	C		
[0,0,0]	3	0	#feat A samples * #feat B samples	sampled value(s)	sampled value(s)	sampled value(s)	prediction values	$\frac{(M-1)}{(M \text{ choose } \lvert z' \rvert)\lvert z' \rvert(M-\lvert z' \rvert)}$
[0,0,1]	3	1	#feat A samples * #feat B samples	sampled value(s)	sampled value(s)	8	prediction values	
[0,1,0]	3	1	#feat A samples * #feat C samples	sampled value(s)	5	sampled value(s)	prediction values	
[0,1,1]	3	2	#feat A samples	sampled value(s)	5	8	prediction values	
[1,0,0]	3	1	#feat B samples * #feat C samples	6	sampled value(s)	sampled value(s)	prediction values	
[1,0,1]	3	2	#feat B samples	6	sampled value(s)	8	prediction values	
[1,1,0]	3	2	#feat C samples	6	5	sampled value(s)	prediction values	
[1,1,1]	3	3	1	6	5	8	prediction values	
				X			$\mathbf{y_{pred}}$	

KernelSHAP just considers the feature values of the rows as matrix of independent variables (**X**) and the model predictions as a target vector ($\mathbf{y_{pred}}$).

In Step 5, KernelSHAP trains a new linear regression ML model, in this case a linear LIME model[34], which regresses the **X**-Matrix against the $\mathbf{y_{pred}}$-vector. **The regression coefficients for each of the three features A, B and C in this new linear regression model are then the SHAP values.**

The weights for each of the 8 "subset contributions" though would not be calculated with the formula used in the Shapley values. Instead, Lundberg and Lee use a new formula that they call "SHAP kernel":

$$\pi_{x'}(z') = \frac{(M-1)}{(M \text{ choose } |z'|)|z'|(M-|z'|)}$$

The SHAP kernel weights are applied to each coalition. Lundberg and Lee show that a linear regression with this kernel weights yield the Shapley values.[35]

KernelSHAP, like the Shapley values, is also an additive linear explanation model, which is an interpretable approximation of the original ML model:

$$f(x) = g(x') = \phi_0 + \sum_{i=1}^{M} \phi_i x'_i$$

As already noticed, KernelSHAP uses a "Marginal Expectation"-approach, as described above, and ignores the dependency structure between present and absent features. KernelSHAP therefore suffers from the same problem as all permutation-based interpretation methods.[36]

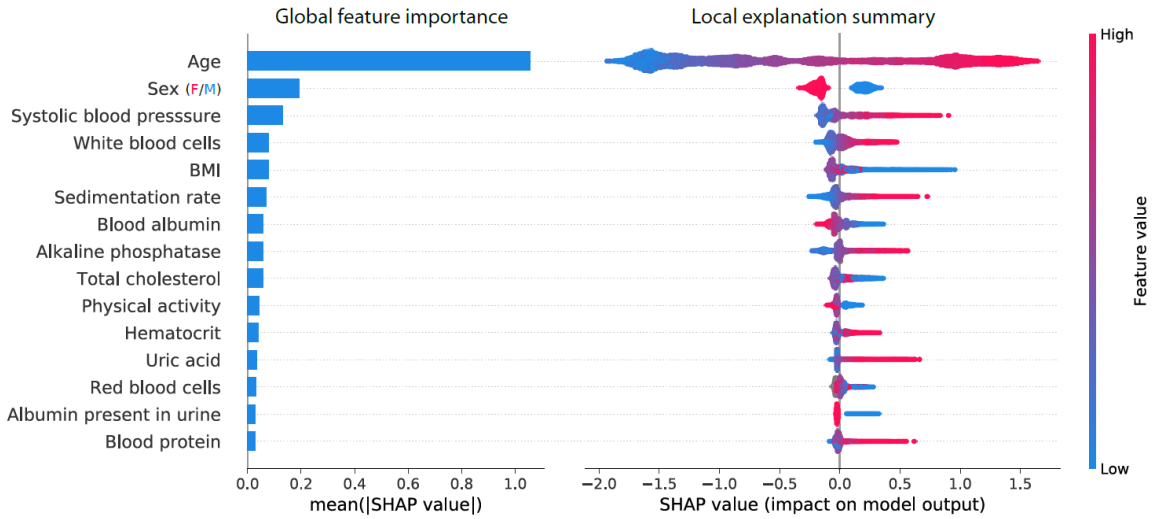
[34] (Lundberg, et al., 25 Nov 2017)
[35] (Lundberg, et al., 25 Nov 2017)
[36] (Molnar, 2019)

b) TreeSHAP

TreeSHAP is a model-agnostic method but is specific in the sense that it can only be applied to tree-based ML models such as decision trees, random forests and gradient boosted trees. TreeSHAP was introduced as a fast "Conditional Expectation"-algorithm. It reduces the computational complexity from $O(TLM2^M)$ to $O(TLD^2)$, by exploiting the characteristics of the tree structure of the ML models.[37] M is the number of input features, L the number of leaves, T the number of trees and D the maximum depth of any tree.

In the following, I show what we can do with the Python TreeSHAP methods in general. For this, I use plots from Lundberg et al[38] from their (gradient boosted decision tree) Mortality Model that identifies influential features on mortality rates:

Figure 38: SHAP global plots: example 1



When running the TreeSHAP methods, the SHAP calculations usually are not just done for a single data instance, but for several.

The left bar plot shows the "Global feature importance", which for each feature, is the average of the (absolute) individual SHAP values for all the single data rows. The individual SHAP values are clustered in the right beeswarm plot "Local explanation summary". The color indicates if the feature value within its feature vector was high (red) or low (blue). The individual SHAP values of the feature "Age", for instance, show a strong positive correlation with mortality rates whereas for the feature "Blood albumin" they show a negative correlation. The Shapley values of the feature "Sex" clearly show that the mortality rate in the data set is lower for all women than for men if the other feature values are equal.

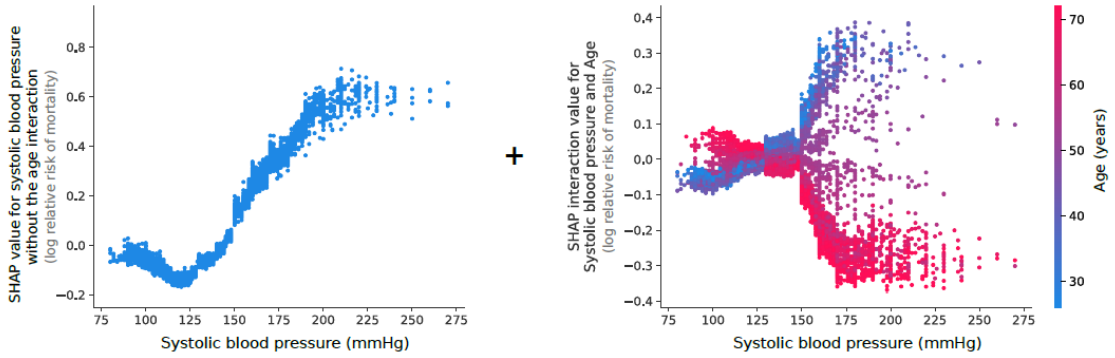
The TreeSHAP library thus allows us not only to break down feature contributions for individual or local instances, but also an aggregation of these results into a global view.

Next is a TreeSHAP Scatter Plot that shows the individual SHAP value dependency on the values for the feature "Systolic blood pressure" (left graph). Beside it (right graph) the same graph is plotted against a second feature "Age" where the value magnitude for the latter is represented by the colors red (high value for "Age") and blue (low values for "Age"):

[37] (Lundberg, et al., 7 Mar 2019)

[38] (Lundberg, et al., 11 May 2019)

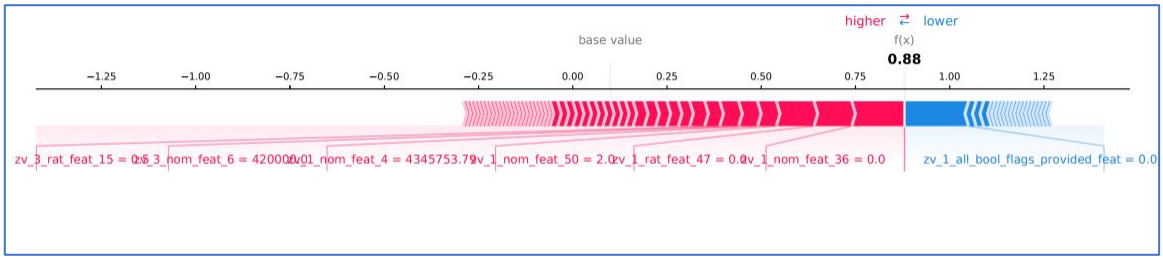
Figure 39: SHAP global plots: example 2



This allows us not only to reveal feature dependency structures, but also to distinguish between a main and an interaction effect. Interaction effects can be identified in these scatter plots if there is a strong vertical dispersion of red and blue dots at the same horizontal location. In the above example (right plot) for instance, there is a strong interaction affect between the two features "Age" and "Systolic blood pressure" as at a value of 175 for "Systolic blood pressure", the SHAP values strongly diverge dependent on the values for "Age". TreeSHAP methods thus can display the variance in marginal contributions that are caused by an interaction effect.

Next is a so called "force plot" that shows the feature contributions to a prediction value for a single (or local) data instance in our Transfer Model. The prediction value here is the prediction received from the Scikit-learn "predict_proba"-method that is later converted to Fraud-Scores as explained above. The "base value" of 0.10 shown in the plot is the average prediction for all instances in the background data set. The individual contribution of each feature moves the individual prediction away from this "base value". All individual feature contributions plus the "base value" add up to the prediction of the single data instance (in this case 0.88).

Figure 40: SHAP local plot: Force plot



To simplify, I will elaborate on the contributions to the predictions only for the XGB model of the Transfer Model and leave out the results for the RFC and Logit model. For the Logit model, there is a comparable LinearSHAP method in the Python SHAP package.

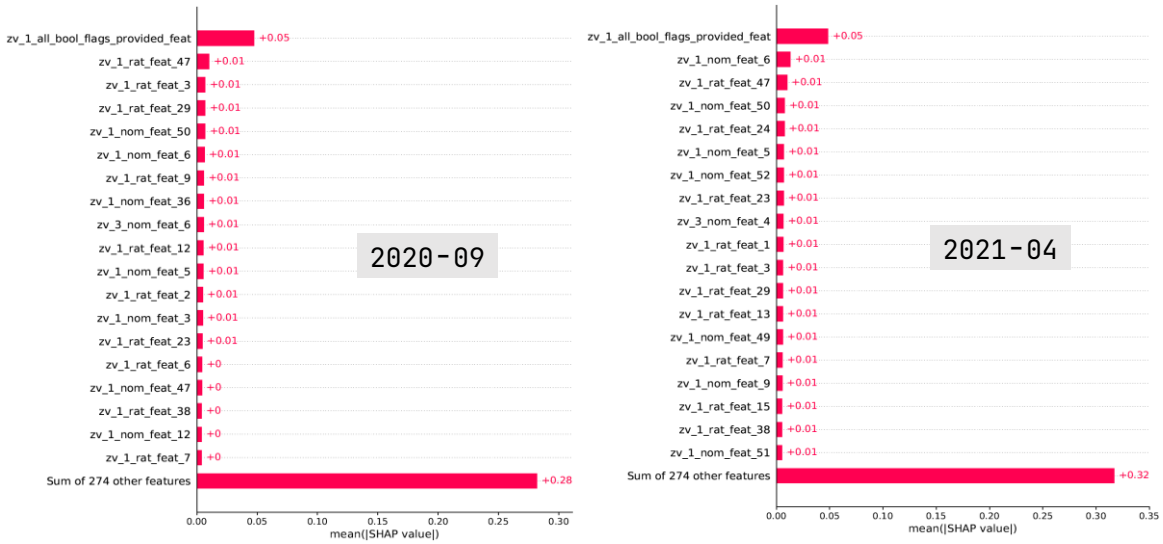
C. Feature stability

In our discussion about the stability of the Transfer Model, we concluded that the Transfer Model is stable over time if we base our judgement on the general model results. What was missing was the confirmation that the stability also applies to the features of the Transfer Model. Only if similar model features contribute to the model results similarly for different months, we can claim that the model is indeed stable over time.

I limit the stability analysis to the comparison between the months of April 2021 and September 2020. But this would need to be done for more than two months to claim that the feature contributions are indeed stable over time.

From looking at the TreeSHAP bar plots for the September 2020 (left side of figure 39) and April 2021 (right side of figure 39), it becomes clear that the global feature contributions to the prediction are very similar for both months:

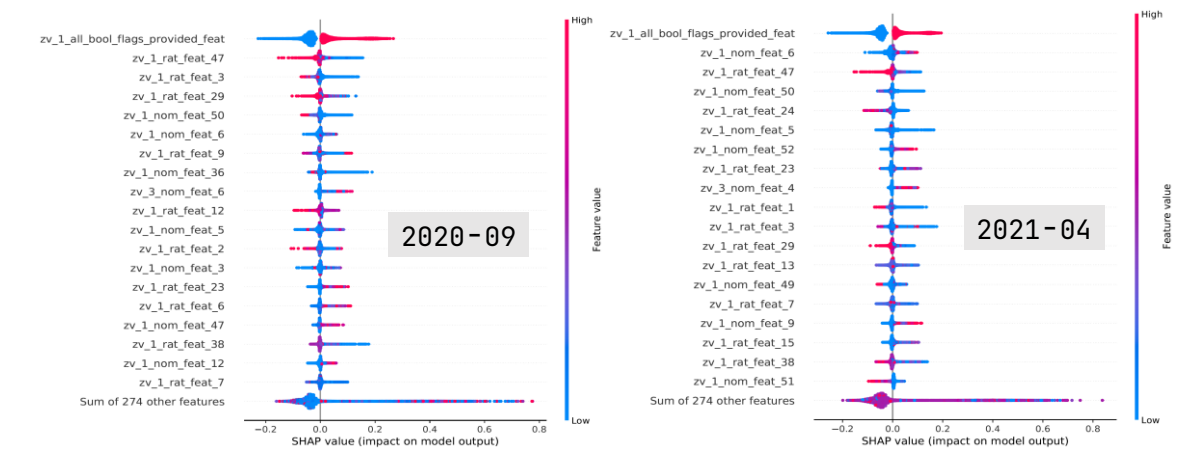
Figure 41: SHAP Bar plots for 2020-09 and 2021-04



The feature "zv_1_all_bool_flags_provided_feat" on average has the highest (absolute) "predict_proba"-contribution of 0.05 in both months. From the following 18 most important features in September 2020, 10 of them are also the most important features in April 2021 and their SHAP values are comparable.

From looking at the SHAP beeswarm plots of both months, it also becomes clear that the direction of their contribution to the predictions is also comparable:

Figure 42: SHAP Beeswarm plots for 2020-09 and 2021-04



For both months, the feature "zv_1_rat_feat_29" for instance, has higher SHAP values (horizontal scale) for lower feature values (blue colored). The same applies to the features "zv_1_rat_feat_47", "zv_1_rat_feat_24", "zv_1_rat_feat_1", "zv_1_rat_feat_49", "zv_1_rat_feat_38" and "zv_1_nom_feat_51". In the opposite direction, namely higher SHAP values for higher feature values (red colored), go the features "zv_1_nom_feat_6", "zv_1_nom_feat_52", "zv_1_nom_feat_4" and "zv_1_nom_feat_9", also for both months.

If we assume that this pattern is inherent in other months as well, we indeed now can claim that the Transfer Model not only is stable over time in respect to general model results (see above) but also in respect to the contribution of its features to model predictions.

D. Tools for analysis

As stated above, the SHAP tool not only allows for the analysis of dependency structures, but also for the analysis of interaction effects:

Figure 43: Interaction 1

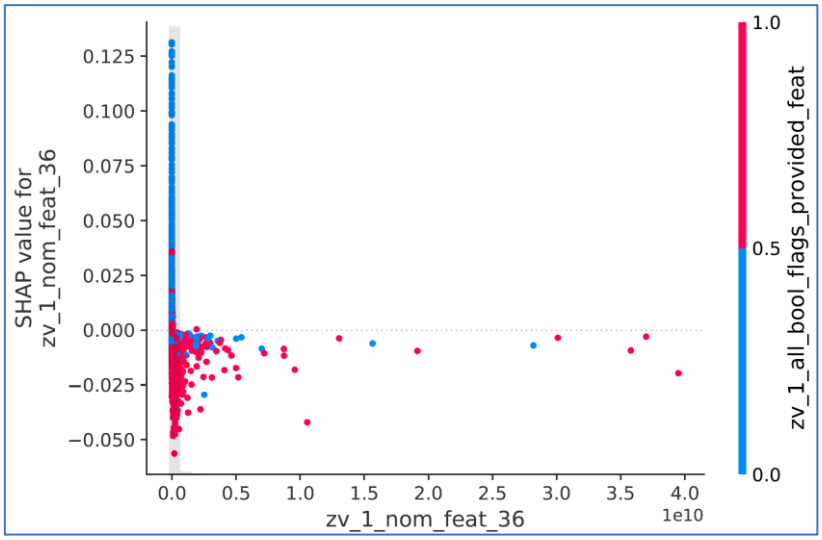
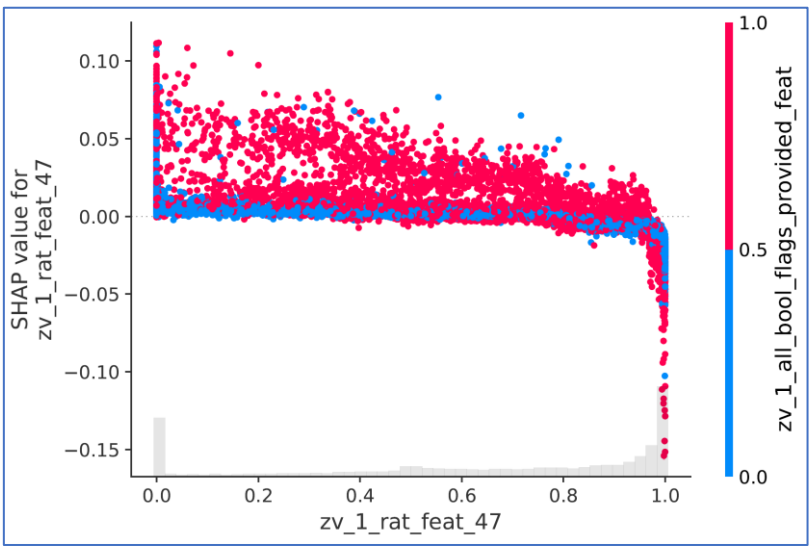


Figure 44: Interaction 2



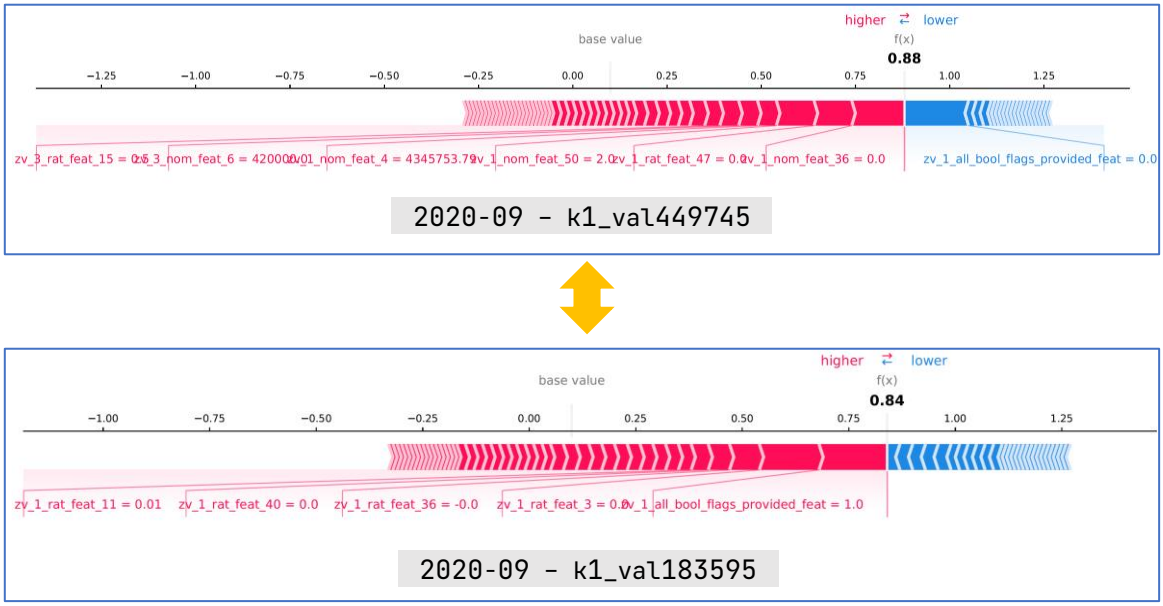
The features "zv_1_rat_feat_47" and "zv_1_nom_feat_36" ("feature 1") in the plots above are vertically dispersed (y-axis) at comparable locations for their feature values (x-axis) dependent on the values of the feature "zv_1_all_bool_flags_provided_feat" ("feature 2").

In the first plot, in general the SHAP values are higher for lower values of feature 2 whereas in the second plot they are lower for lower values of feature 2. At the second plot, at extremely high values for feature 1 this interaction effect reverses.

This kind of analysis could, for example, help fraud investigators to analyze general interaction patterns and try to spot individual borrowers whose values for such interacted features change.

In the next figure, I display the September 2020 predictions for two different borrowers, whose high "predict_proba"-values are attributed to different features:

Figure 45: SHAP Force plots for the same month but two different borrowers with comparable predict_probas



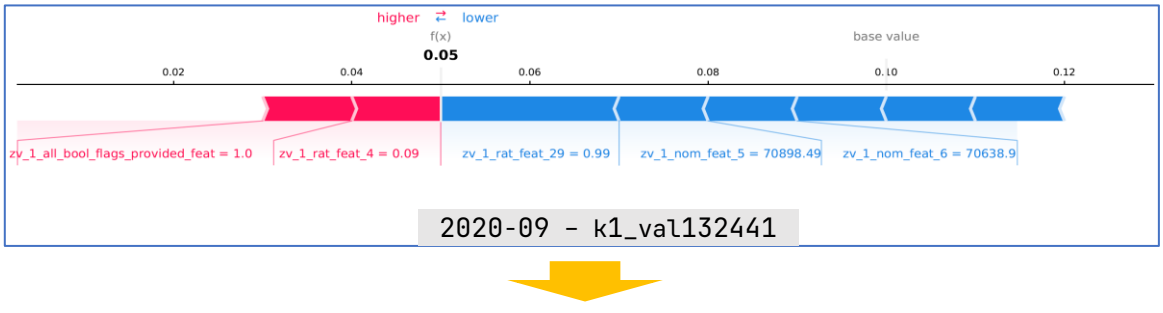
The "predict_proba"-value of 0.88 for the borrower with the ID "k1_val449745" is mainly attributed to the values of the features "zv_1_nom_feat_36", "zv_1_rat_feat_47", "zv_1_nom_feat_50", "zv_1_nom_feat_4", "zv_1_nom_feat_6" and "zv_1_rat_feat_15".

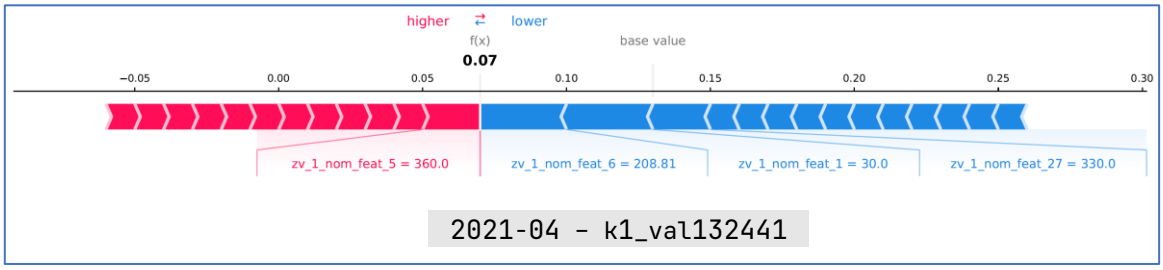
Whereas for the borrower with the ID "k1_val183595", the "predict_proba"-value of 0.84 is mainly driven by the values of the features "zv_1_all_bool_flag_provided_feat", "zv_1_rat_feat_3", "zv_1_rat_feat_36", "zv_1_rat_feat_40" and "zv_1_rat_feat_11".

This approach in general provides detailed arguments in the discussion of why a certain decision was taken for a particular case, or applied to applicants, why a particular application was rejected or accepted.

Next is a plot for just one borrower whose "predict_proba"-value remained almost constant in both months:

Figure 46: No change: SHAP Force plots for borrower k1_val132441 for 2020-09 and 2021-04

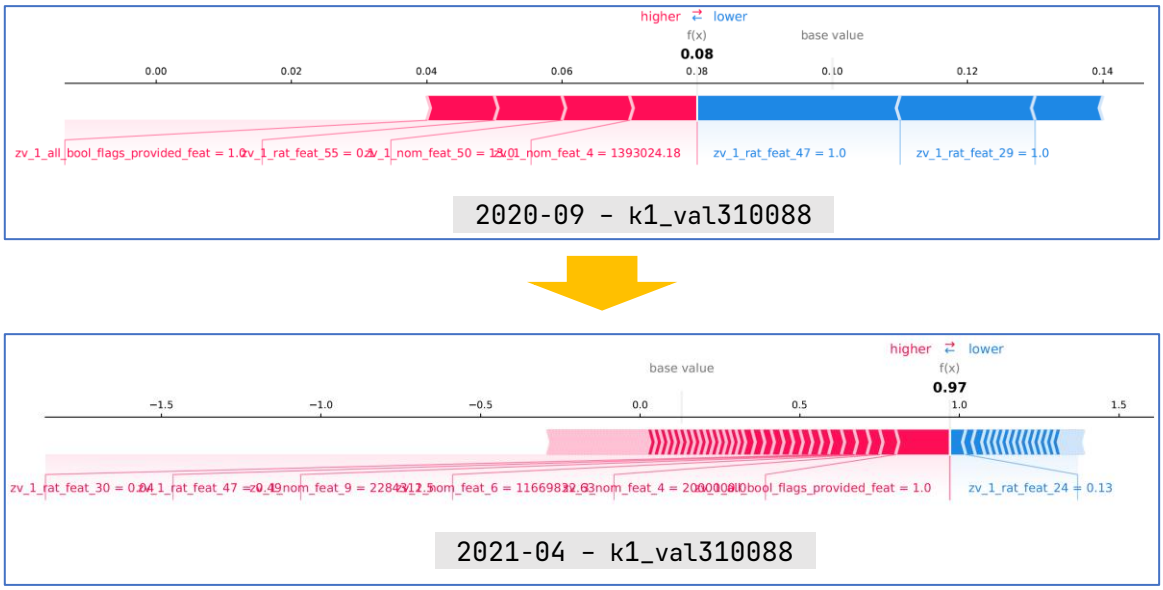




The base values for both months are only slightly different (0.10 and 0.13). But the feature contributions that explain the divergence from this base value for the two months do differ.

In the following plot, the "predict_proba"-value for one borrower jumped from a low "predict_proba"-value of 0.08 in September 2020 to a high value of 0.97 in April 2021:

Figure 47: Big Jump: SHAP Force plot for borrower k1_val310088 for 2020-09 and 2021-04



The features "zv_1_rat_feat_47" and "zv_1_rat_feat_47" whose values lowered the SHAP value in September 2020 were not present in April 2021. As the two important features "zv_1_all_bool_flag_provided_feat" and "zv_1_nom_feat_47" were present in both months, it was the many small contributions of many other features that raised the "predict_proba"-value dramatically. Here the investigators are not given a few significant features that they can elaborate on but are shown that there are cases where ML intelligence can identify suspicious borrowers.

5. Conclusion

In this thesis, I have shown the process to extract additional information from new features that are not part of an existing ML model ("Original Model") that is used to detect credit fraud. The new features are the independent variables of a new ML model ("Transfer Model") whose dependent variable or target values are the predictions of the Original Model. Through these predictions, the Original Model and the Transfer Model are connected. The new Transfer Model features reveal their information when the contributions of the new features to the predictions of the Original Model are analyzed. This was done by using feature explanation techniques.

The need to develop a Transfer Model arises when new features cannot be integrated into an existing ML model. In the case discussed in this thesis, the new features (the payment transaction data) could not be integrated into the Original Model because the history of this data was not long enough.

There are several feature explanation techniques available, but the focus was laid on Shapley values and the Python SHAP package because of the desire of fraud investigators to analyze feature contributions for single data instances or in their case individual borrowers.

I started with describing the process to develop a typical ML pipeline and the differences to it when establishing a Transfer Model. I laid out the software program architecture for this pipeline, justified the application of Object-Oriented Programming principles and explained in detail how the implemented classes, attributes, methods and data structures work. I also showed extracts of this pipeline as program code run in Jupyter Notebooks and its outputs.

Next, I reasoned why the Transfer Model should do a classification instead of a regression modelling and explained the methodology to convert discrete Fraud-Scores of the Original Model to binary target values for the Transfer Model. The distinction criterion to qualify as "fraud" instead of "no fraud" was set to a Fraud-Score of 90 and the reasons for that decision were discussed. The training set was reduced so that the "no fraud" samples did not dominate the rare "fraud" samples too much. The Transfer Model in terms of ROC-AUC scores was evaluated as having "medium bias" and "low variance". It was prognosticated that the Transfer Model could be improved if additional balance sheet data was used. When the Transfer Model was trained with data from two different months, it was found that the two sets of ROC-AUC scores were comparable concluding that the Transfer Model was stable over time. But this judgement would need to be confirmed by analyzing it over a longer time horizon.

I argued that the assessment of a "stable ML model" also requires that feature contributions are stable over time and thus similar for different months. To measure feature contributions, I used the TreeSHAP methods of the Python shap package developed by Lundgren et al, but first explained the concept of feature importances in general and Shapley values in particular. I also showed how fraud investigators could use the TreeSHAP methods and plots to analyze individual borrowers.

By comparing global SHAP values or contributions for the most significant features for the two months of April 2021 and September 2020, I concluded that not only the ROC-AUC scores but also the feature contributions were similar and thus stable over time.

Assuming the required analysis over a longer time horizon would not change this conclusion, I would affirm the hypothesis that it is possible to perform model explanations on "out-of-model" features.

By using a Transfer Model, fraud investigators could gain additional insights and access to additional information sources to which they did not have access before.

6. Appendix

A. List of Abbreviations

B

BDAA ..... *Big Data & Advanced Analytics - Commerzbank department*

D

DBMS..... *Database Management Systems*

F

FN ..... *False neagtive (rate)*
FP ..... *False-Positive rate*

L

Logit..... *Logistic Regression Model*

M

ML ..... *Machine Learning*

O

Original Features ..... *The features used in the Original Model*
Original Model ..... *The existent ML model without payment transaction data*

P

predict_probas ..... *The predicted probabilities from the Scikit-Learn "predict_proba"-method*

R

RFC ..... *Random Forest Classifier*
ROC curve..... *Plot for trade-off between True-Positive and False-Positive rate*
ROC-AUC ..... *Area under the ROC curve*

T

TN ..... *True-Negative rate*
TP ..... *True-Positive rate*
Transfer Features ..... *The features of the Transfer Model (payment transaction data)*
Transfer Model..... *The new transfer ML model with payment transaction data features*

X

XGB..... *XGBoost Classfier - Extreme Gradient Boosting Classifier*

B. List of Figures

C. References

Brownlee, Jason. 2019. https://machinelearningmastery.com. *machinelearningmastery.* [Online] Brownlee, Jason, October 25, 2019. [Cited: September 01, 2021.] https://machinelearningmastery.com/gentle-introduction-to-the-bias-variance-trade-off-in-machine-learning/.

Caruana, Rich and Niculescu-Mizil, Alexandru. 2006. *Predicting good probabilities with supervised learning.* [ed.] Cornell University, Ithaca, New York Department of Computer Science. Proceedings of the 23rd International Conference on Machine Learning, Pittsburgh, PA, 2006 : s.n., 2006. pp. 1-8.

Commerzbank.de. 2021. www.commerzbank.de. [Online] 2021. [Cited: September 01, 2021.] https://www.commerzbank.de/de/hauptnavigation/konzern/konzern.html.

Evans, Eric. 2003. *Domain-Driven Design: Tackling Complexity in the Heart of Software.* [ed.] Addison-Wesley. 1. A Edition (20. August 2003). s.l. : Addison-Wesley Professional, 2003. ISBN 978-0321125217.

Hastie, Trevor, Tibshirani, Robert and Friedman, Jerome. 2009. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction.* Second Edition (2009). s.l. : Springer, 2009. Vol. Springer Series in Statistics. ISBN 978-0-387-84857-0.

Hettinger, Raymond. 2021. Descriptor HowTo Guide. *Python HOWTOs, English, Python 3.9.6.* [Online] 2021. [Cited: September 01, 2021.] https://docs.python.org/3/howto/descriptor.html.

Jaspret. 2019. https://towardsdatascience.com. *towardsdatascience.* [Online] 2019. [Cited: September 01, 2021.] https://towardsdatascience.com/understanding-and-reducing-bias-in-machine-learning-6565e23900ac.

Lundberg, Scott and Lee, Su-In. 25 Nov 2017. *A Unified Approach to Interpreting Model Predictions.* Conference Presentation: 31st Conference on Neural Information Processing Systems (NIPS 2017). Long Beach, CA, USA : s.n., 25 Nov 2017.

Lundberg, Scott, Erion, Gabriel and Lee, Su-In. 7 Mar 2019. *Consistent Individualized Feature Attribution for Tree Ensembles.* University of Washington. s.l. : arXiv preprint arXiv:1802.03888 (2018), 7 Mar 2019.

Lundberg, Scott, et al. 11 May 2019. Explainable AI for Trees. From Local Explanations to Global Understanding. s.l. : arXiv preprint arXiv:1905.04610 (2019), 11 May 2019.

Molnar, Christoph. 2019. *Interpretable Machine Learning. A Guide for Making Black Box Models Explainable.* s.l. : lulu.com, 2019. ISBN 9780244768522.

numpy.org. 2021. numpy.org. [Online] 2021. [Cited: September 01, 2021.] https://numpy.org/about/.

Pandas. 2021. Pandas. [Online] 2021. [Cited: September 01, 2021.] https://pandas.pydata.org/.

Scikit-Learn. 2021. https://scikit-learn.org/stable/. [Online] 2021. [Cited: September 01, 2021.] https://scikit-learn.org/stable/.

Shapley, Lloyd. 1953. *Contributions to the Theory of Games: A value for n-person games.* [ed.] H.W. Kuhn and A.W. Tucker. Annals of Mathematics Studies v. 28. s.l. : Princeton University Press, 1953. pp. 307-317. Vol. Volume II. ISBN 0-691-07935-8.

Swapnil-Vishwakarma. 2021. https://www.analyticsvidhya.com. [Online] 2021. [Cited: September 01, 2021.] https://www.analyticsvidhya.com/blog/2021/04/the-curse-of-dimensionality-in-machine-learning/.

Wikipedia: Banken in Deutschland. 2021. Wikipedia. *Wikipedia.* [Online] 2021. [Cited: September 01, 2021.] https://de.wikipedia.org/wiki/Liste_der_gr%C3%B6%C3%9Ften_Banken_in_Deutschland.

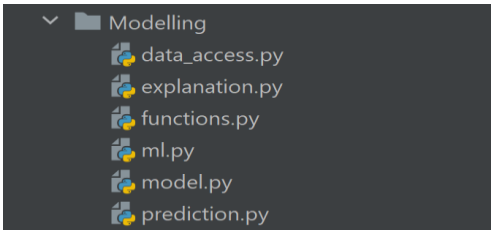
Wikipedia: Jupyter. 2021. Wikipedia: Jupyter. [Online] 2021. [Cited: September 01, 2021.] https://de.wikipedia.org/wiki/Project_Jupyter.

Wikipedia: Python. 2021. Wikipedia: Python. [Online] 2021. [Cited: September 01, 2021.] https://en.wikipedia.org/wiki/Python_(programming_language).

D. Program Code

Please be aware that the following code might contain incorrect line breaks or improper indentation because of word formatting conversion errors.

1. Program Structure



2. Classes ML and Configuration

```
class ML:
    configuration = None

    def __init__(self):
        self.source_file_path = None
        self.target_file_path = None

    @classmethod
    def set_configuration(cls, path_to_config_file: str):
                                cls.configuration = Configuration(path_to_config_file)


class Configuration:
    def __init__(self, path_to_config_file: str):
        self._config = ConfigParser()
        self._config.read(path_to_config_file)

    def get_config_setting(self, config_section: str, config_name: str, model_type: str = None):
        try:
            if model_type is not None:
                return  eval(self._config[config_section][config_name]).get(model_type)
            else:
                return self._config[config_section][config_name]
        except KeyError:
            return f'Either section {config_section!r} or var_name {config_name!r} not found. Please check config file!'

    def get_path(self, config_section: str, config_name: str, model_type: str = None) -> str:
        return join(self.get_config_setting(config_section=config_section, config_name='file_path'),
                    self.get_config_setting(config_section=config_section, config_name=config_name,model_type=model_type))
```

3. Class DataAccessDescriptor

```
class DataAccessDescriptor:
    def __get__(self, obj, objtype=None):
        file_path = obj.source_file_path.lower()
        try:
            if file_path.endswith('.csv'):
                return pd.read_csv(obj.source_file_path)
            elif file_path.endswith('.joblib'):
                return jl.load(obj.source_file_path)
            else:
                raise ValueError('FILE NOT RETRIEVED: File extension unknown. Extension must
                                  be ".csv" or ".joblib" !')
        except OSError:
            print(f'File could not be read from file path "{obj.source_file_path}" !')

    def __set__(self, obj, value):
        file_path = obj.target_file_path.lower()
        try:
            if file_path.endswith('.csv'):
                value.to_csv(obj.target_file_path)
            elif file_path.endswith('.joblib'):
                jl.dump(value, obj.target_file_path)
            else:
                raise ValueError('FILE NOT SAVED: File extension unknown. Extension must be
                                  ".csv" or ".joblib" !')
        except OSError:
            print(f'File could not be written to file path "{obj.target_file_path}" !')

        print(f'The file was successfully saved as/to: {obj.target_file_path} !')
```

4. Class InsideAccess

```
class InsideAccess(ML):
    _stored_dataframe = DataAccessDescriptor()
    _stored_fitted_estimator = DataAccessDescriptor()
    _stored_prediction_input_object = DataAccessDescriptor()
    _stored_explanation_input_object = DataAccessDescriptor()

    def __init__(self):
        if super().configuration is not None:
            super().__init__()
            self.dataframe = None
            self.fitted_estimator = None
            self.data_for_prediction = None
            self.prediction_input_object = None
            self.explanation_input_object = None
        else:
            raise PermissionError('Configuration and access to configuration file not set yet !')

    def load_dataframe(self, config_section: str, config_name: str, model_type: str = None):
        self.source_file_path = self.configuration.get_path(config_section, config_name, model_type)
        self.dataframe = self._stored_dataframe

    def save_dataframe(self, config_section: str, config_name: str, model_type: str = None):
        self.target_file_path = self.configuration.get_path(config_section, config_name, model_type)
        self._stored_dataframe = self.dataframe

    def load_fitted_estimator(self, config_section: str, config_name: str, model_type: str = None):
        self.source_file_path = self.configuration.get_path(config_section, config_name, model_type)
        self.fitted_estimator = self._stored_fitted_estimator
```

```
    def save_fitted_estimator(self, config_section: str, config_name: str, model_type: str = None):
        self.target_file_path = self.configuration.get_path(config_section, config_name, model_type)
        self._stored_fitted_estimator = self.fitted_estimator

    def load_data_for_prediction(self, config_section: str, config_name: str, model_type: str=None):
        self.source_file_path = self.configuration.get_path(config_section, config_name, model_type)
        self.dataframe = self._stored_dataframe

    def save_data_for_prediction(self, config_section: str, config_name: str, model_type: str=None):
        self.target_file_path = self.configuration.get_path(config_section, config_name, model_type)
        self._stored_dataframe = self.dataframe

    def load_prediction_input_object(self, config_section: str, config_name: str, model_type: str = None):
        self.source_file_path = self.configuration.get_path(config_section, config_name, model_type)
        self.prediction_input_object = self._stored_prediction_input_object

    def save_prediction_input_object(self, config_section: str, config_name: str, model_type: str = None):
        self.target_file_path = self.configuration.get_path(config_section, config_name, model_type)
        self._stored_prediction_input_object = self.prediction_input_object

    def load_explanation_input_object(self, config_section: str, config_name: str, model_type: str = None):
        self.source_file_path = self.configuration.get_path(config_section, config_name, model_type)
        self.explanation_input_object = self._stored_explanation_input_object

    def save_explanation_input_object(self, config_section: str, config_name: str, model_type: str = None):
        self.target_file_path = self.configuration.get_path(config_section, config_name, model_type)
        self._stored_explanation_input_object = self.explanation_input_object
```

5. Class OutsideAccess

```
class OutsideAccess(ML):
    _stored_prediction_dataframe = DataAccessDescriptor()
    _stored_result_dataframe = DataAccessDescriptor()
    _stored_quantile_portfolio = DataAccessDescriptor()
    _stored_transfer_model_input_object = DataAccessDescriptor()

    # These are the aggregated results (ONE concatenated pd.DataFrame for each) from the
    individual instances:
    result_dataframe = None
    quantile_portfolio = None

    def __init__(self):
        if super().configuration is not None:
            super().__init__()
            self.prediction_dataframe = None
            self.transfer_model_input_object = None
        else:
            raise PermissionError('Configuration and access to configuration file not set yet !')

    # Must be at the class level as attribute is class attribute
    def load_result_dataframe(self, config_section: str, config_name: str, model_type: str=None):
        self.source_file_path = self.configuration.get_path(config_section, config_name, model_type)
        OutsideAccess.result_dataframe = self._stored_result_dataframe

    # Must be at the class level as attribute is class attribute
    def save_result_dataframe(self, config_section: str, config_name: str, model_type: str=None):
        self.target_file_path = self.configuration.get_path(config_section, config_name, model_type)
        self._stored_result_dataframe = OutsideAccess.result_dataframe
```

```
    # Must be at the class level as attribute is class attribute
    def load_quantile_portfolio(self, config_section: str, config_name: str, model_type: str=None):
        self.source_file_path = self.configuration.get_path(config_section, config_name, model_type)
        OutsideAccess.quantile_portfolio = self._stored_quantile_portfolio

    # Must be at the class level as attribute is class attribute
    def save_quantile_portfolio(self, config_section: str, config_name: str, model_type: str=None):
        self.target_file_path = self.configuration.get_path(config_section, config_name, model_type)
        self._stored_quantile_portfolio = OutsideAccess.quantile_portfolio

    def load_prediction_dataframe(self, config_section: str, config_name: str, model_type: str=None):
        self.source_file_path = self.configuration.get_path(config_section, config_name, model_type)
        self.prediction_dataframe = self._stored_prediction_dataframe

    def save_prediction_dataframe(self, config_section: str, config_name: str, model_type: str=None):
        self.target_file_path = self.configuration.get_path(config_section, config_name, model_type)
        self._stored_prediction_dataframe = self.prediction_dataframe

    def load_transfer_model_input_object(self, config_section: str, config_name: str, model_type: str = None):
        self.source_file_path = self.configuration.get_path(config_section, config_name, model_type)
        self.transfer_model_input_object = self._stored_transfer_model_input_object

    def save_transfer_model_input_object(self, config_section: str, config_name: str, model_type: str = None):
        self.target_file_path = self.configuration.get_path(config_section, config_name, model_type)
        self._stored_transfer_model_input_object = self.transfer_model_input_object
```

6. Class Model

```
class Model(InsideAccess):

  def __init__(self):
    super().__init__()
    self.key_name_in_dataframes = None
    self.second_key_name_in_dataframes = None
    self.name_of_target = None
    self.train_set_row_index = None
    self.test_set_row_index = None
    self.X = None
    self.X_train = None
    self.X_test = None
    self.y = None
    self.y_train = None
    self.y_test = None
    self.model = None
    self.unfitted_estimator = None
    self.estimator_is_pipeline = False
    self.hyper_params = None
    self.group = None
    self.are_features_excluded = False
    self.features_in_training = None
    self._set_key_target_features_has_run = False
    self._set_estimators_and_parameters_has_run = False
    self._train_test_split_has_run = False

  def set_dataframe(self, dataframe: pd.DataFrame):
    check_parameter_is_of_type(parameter=dataframe, parameter_type=pd.DataFrame)
    self.dataframe = dataframe

  def set_key_target_features(self, key: str, target: str, features: list = None,
                              features_excluded_in_training: List[str] = None, second_key: str = None):
    self._check_dataframe_is_set()
    self._check_key_is_in_dataframe(key)
    check_parameter_is_of_type(parameter=key, parameter_type=str)
    self.key_name_in_dataframes = key
    check_parameter_is_of_type(parameter=target, parameter_type=str)
    self.name_of_target = target
    if features is None:
      features = [feat for feat in self.dataframe.columns if feat is not self.name_of_target]
    else:
      check_parameter_is_of_type(parameter=features, parameter_type=list)
      features = [feat for feat in features if feat is not self.name_of_target]
    self.X = self.dataframe.loc[:, features]
    self.y = self.dataframe.loc[:, target]
    if features_excluded_in_training is not None:
      check_parameter_is_of_type(parameter=features_excluded_in_training, parameter_type=list)
      self.are_features_excluded = True
      self.features_in_training = [feat for feat in features if feat not in
                                   features_excluded_in_training]
    else:
      self.features_in_training = features
    if second_key is not None:
      check_parameter_is_of_type(parameter=second_key, parameter_type=str)
      self.second_key_name_in_dataframes = second_key
    self._set_key_target_features_has_run = True

  def set_estimator_and_parameters(self, unfitted_estimator, hyper_params_dict_name_in_config: str):
    hyp_params = eval(self.configuration.get_config_setting(config_section='Model.Hyperparams',
                  config_name=hyper_params_dict_name_in_config))
    unfit_est = unfitted_estimator
    if unfit_est.__class__.__name__ == 'LogisticRegression':
      hyp_params = make_param_grid_for_pipe_gridsearch('estimator', hyp_params)
      unfit_est = self._make_pipeline(unfitted_estimator=unfit_est)
      self.estimator_is_pipeline = True
    self.hyper_params = hyp_params
    self.unfitted_estimator = unfit_est
    self._set_estimators_and_parameters_has_run = True
```

Class Model continued

```
def train_test_split_dataframe(self, group: list or str = None):
    self._check_set_key_target_features_has_run()
    if self.X is not None and self.y is not None:
        test_set_size = float(
            self.configuration.get_config_setting(config_section='Model.General',
                                                  config_name='test_set_size'))
        if group is not None:
            self.group = group
            X_group = self.X.loc[:, self.group]
        else:
            X_group = None
        self.train_set_row_index, self.test_set_row_index = next(
            GroupShuffleSplit(n_splits=1, test_size=test_set_size, random_state=None).split(X=self.X,
                                                                                            y=self.y, groups=X_group))
        self._set_train_test_set_rows()
    else:
        raise ValueError('Features and target not set yet. Set them first before splitting into train-
                         /test-sets !')
    self._train_test_split_has_run = True

def train_model_and_cross_validate(self):
    self._check_dataframe_is_set()
    self._check_set_key_target_features_has_run()
    self._check_set_estimators_and_parameters_has_run()
    self._check_train_test_split_has_run()
    if all(attr is not None for attr in
           [self.X_train, self.y_train, self.unfitted_estimator, self.train_set_row_index]):
        if self.group is not None:
            X_train_group = self.X_train.loc[:, self.group]
        else:
            X_train_group = None
        num_cv_folds = int(
            self.configuration.get_config_setting(config_section='Model.General',
                                                  config_name='num_cv_folds'))
        grid_search_iterator = GroupKFold(n_splits=num_cv_folds).split(X=self.X_train, y=self.y_train,
                                                                       groups=X_train_group)
        scoring = eval(self.configuration.get_config_setting(config_section='Model.General',
                                                             config_name='scoring'))
        grid_search = GridSearchCV(estimator=self.unfitted_estimator, param_grid=self.hyper_params,
                                   scoring=scoring,
                                   n_jobs=None,
                                   cv=grid_search_iterator, refit=scoring[0], return_train_score=True,
                                   verbose=0)
        self._exclude_features_in_train_test_set()
        self.model = grid_search.fit(self.X_train, self.y_train)
        self.fitted_estimator = self.model.best_estimator_
    else:
        raise ValueError('Missing input for either X, y, train_set_row_index and/or estimator. Please
                         set those first !')

def under_sample_dataframe(self, sampling_strategy: str or float):
    if not self._train_test_split_has_run:
        under_sampler = RandomUnderSampler(sampling_strategy=sampling_strategy)
        self.X, self.y = under_sampler.fit_resample(self.X, self.y)
    else:
        raise TypeError('train_test_split_dataframe()-method has already run. sampling not possible
                        anymore !')

def save_test_set(self, config_section: str, model_type: str = None):
    self.target_file_path = self.configuration.get_path(config_section, 'X_test', model_type)
    self._stored_dataframe = self.X_test
    self.target_file_path = self.configuration.get_path(config_section, 'y_test', model_type)
    self._stored_dataframe = self.y_test

def get_cv_results(self) -> pd.DataFrame:
    return pd.DataFrame(data=self.model.cv_results_).T

def get_test_set_roc_auc_score(self) -> float:
    return roc_auc_score(self.y_test, self.fitted_estimator.predict_proba(self.X_test)[:, 1])

def generate_prediction_input(self):
    self.prediction_input_object = PredictionInput(key_name_in_dataframes=self.key_name_in_dataframes,
                                                   model_best_estimator=self.fitted_estimator,
```

Class Model continued

```
def generate_explanation_input(self):
    self.explanation_input_object = ExplanationInput(model_best_estimator=self.fitted_estimator,
                                        X=self.X, X_train=self.X_train, X_test=self.X_test,
                                        key_name_in_dataframes=self.key_name_in_dataframes,
                                        second_key_name_in_dataframes=self.second_key_name_in_dataframes)

def _set_train_test_set_rows(self):
    self.X_train = self.X.iloc[self.train_set_row_index, :]
    self.y_train = self.y.iloc[self.train_set_row_index]
    self.X_test = self.X.iloc[self.test_set_row_index, :]
    self.y_test = self.y.iloc[self.test_set_row_index]

def _exclude_features_in_train_test_set(self):
    if self.are_features_excluded:
        self.X_train = self.X_train.loc[:, self.features_in_training]
        self.X_test = self.X_test.loc[:, self.features_in_training]

def _check_dataframe_is_set(self):
    if self.dataframe is None:
        return FileNotFoundError('dataframe is not set yet !')

def _check_train_test_split_has_run(self):
    if self._train_test_split_has_run is False:
        raise TypeError('Run train_test_split_dataframe() first !')

def _check_set_key_target_features_has_run(self):
    if self._set_key_target_features_has_run is False:
        raise TypeError('Run set_key_target_features() method first !')

def _check_set_estimators_and_parameters_has_run(self):
    if self._set_estimators_and_parameters_has_run is False:
        raise TypeError('Run set_estimators_and_parameters_has_run() method first !')

def _check_key_is_in_dataframe(self, key: str):
    if key not in self.dataframe.columns:
        return ValueError('key is not in dataframe !')

def _make_pipeline(self, unfitted_estimator):
    all_columns = self.features_in_training
    impute_dict = eval(self.configuration.get_config_setting(config_section='Model.CV.Preprocessing',
                                            config_name='impute'))
    winsorize_dict = eval(self.configuration.get_config_setting(config_section='Model.CV.Preprocessing',
                                             config_name='winsorize'))
    scale_dict = eval(self.configuration.get_config_setting(config_section='Model.CV.Preprocessing',
                                           config_name='scale'))
    imp_cols = all_columns if impute_dict.get('impute_columns') == 'All' else
                                             impute_dict.get('impute_columns')

    imp_str = impute_dict.get('impute_strategy')
    imp_knn_num_neigh = impute_dict.get('impute_knn_num_neighbors')
    win_cols = all_columns if winsorize_dict.get('winsorize_columns') == 'All' else winsorize_dict.get(
                                                    'winsorize_columns')
    win_lower = winsorize_dict.get('winsorize_lower_bound')
    win_upper = winsorize_dict.get('winsorize_upper_bound')
    scale_cols = all_columns if scale_dict.get('scale_columns') == 'All' else
    scale_dict.get('scale_columns')
    scale_str = scale_dict.get('scale_strategy')
    scale_min_max = scale_dict.get('scale_min_max_range')
    scale_rob_range = scale_dict.get('scale_robust_quantile_range')
    pipe = make_preprocessing_pipeline_for_cv_with_function(
        unfitted_estimator=unfitted_estimator,
        impute_columns=imp_cols,
        winsorize_columns=win_cols,
        scale_columns=scale_cols,
        impute_strategy=imp_str,
        impute_knn_num_neighbors=imp_knn_num_neigh,
        scale_strategy=scale_str,
        scale_min_max_range=scale_min_max,
        scale_robust_quantile_range=scale_rob_range,
        winsorize_lower_bound=win_lower,
        winsorize_upper_bound=win_upper)
    return pipe
```

7. Class TransferModel

```
class TransferModel(Model):

  def __init__(self):
    super().__init__()
    self.transfer_model_targets = None
    self.cutoff_columns = None
    self._set_transfer_model_inputs_has_run = False

  def set_transfer_model_inputs(self, transfer_model_input: TransferModelInput):
    check_parameter_is_of_type(parameter=transfer_model_input, parameter_type=TransferModelInput)
    check_parameter_is_of_type(parameter=transfer_model_input.name_of_target, parameter_type=str)
    check_parameter_is_of_type(parameter=transfer_model_input.transfer_model_targets,
                               parameter_type=pd.DataFrame)
    logging.warning(
      f" Please be aware that the dataframe of the TransferModel MUST NOT CONTAIN the target column as
        the"
      f" target column is generated within the transfer model itself. Please check !")
    # transfer_model_targets contain the columns: key (i.e. "orig_key_1") and quantiles from all models
    self.name_of_target = 'TRANS_TARGET_of_' + transfer_model_input.name_of_target
    self.transfer_model_targets = transfer_model_input.transfer_model_targets  # contains "key" at first
                                                                                column
    self.key_name_in_dataframes = transfer_model_input.key_name_in_dataframes
    self._create_target_column()
    self._combine_targets_with_features()
    self._set_transfer_model_inputs_has_run = True

  def set_key_target_features(self, key: str = None, target: str = None, features: list = None,
                              features_excluded_in_training: List[str] = None, second_key: str = None):
    # We must invalidate the parameters "key", "second_key" and "target" as they are already set with
    # the set_transfer_model_inputs()-method
    self._check_set_transfer_model_inputs_has_run()
    self._check_dataframe_is_set()
    super().set_key_target_features(key=self.key_name_in_dataframes,
                                    target=self.name_of_target,
                                    features=features,
                                    features_excluded_in_training=features_excluded_in_training,
                                    second_key=self.second_key_name_in_dataframes)

  def _create_target_column(self):
    estimators_for_cutoff =
              eval(self.configuration.get_config_setting(config_section='Transfer.Model.Cutoff',
                                                         config_name='estimators_for_cutoff'))
    cutoff_threshold =
              float(self.configuration.get_config_setting(config_section='Transfer.Model.Cutoff',
                                                          config_name='cutoff_threshold'))
    self.cutoff_columns = [name for name in self.transfer_model_targets.columns if
                           any(name_part in name for name_part in estimators_for_cutoff)]
    self.transfer_model_targets[self.name_of_target] = np.where(
      (self.transfer_model_targets[self.cutoff_columns] >= cutoff_threshold).all(axis='columns'), 1, 0)

  def _combine_targets_with_features(self):
    num_rows_before_merge = len(self.dataframe.index)
    key_and_target_columns = self.transfer_model_targets.loc[:, [self.key_name_in_dataframes,
                                                                 self.name_of_target]]
    self.dataframe = pd.merge(self.dataframe, key_and_target_columns, on=self.key_name_in_dataframes,
                              how='inner')
    num_rows_after_merge = len(self.dataframe.index)
    if num_rows_before_merge != num_rows_after_merge:
      logging.warning(
        f" Please be aware that the number of dataframe rows was reduced from
          {num_rows_before_merge!s}"
        f" rows to {num_rows_after_merge!s} rows as the targets did not contain all "
        f" {self.key_name_in_dataframes}'s that were in the dataframe!")

  def _check_set_transfer_model_inputs_has_run(self):
    if not self._set_transfer_model_inputs_has_run:
      raise ValueError('Run set_transfer_model_inputs() method first !')
```

8. Class Prediction

```
class Prediction(OutsideAccess):
    quantile_columns = list()

    def __init__(self, prediction_input: PredictionInput):
        check_parameter_is_of_type(parameter=prediction_input, parameter_type=PredictionInput)
        super().__init__()
        self.model_estimator = prediction_input.model_best_estimator
        self.estimator_is_pipeline = prediction_input.estimator_is_pipeline
        self.X_train_test_columns = prediction_input.features_in_training
        self.key_name_in_dataframes = prediction_input.key_name_in_dataframes
        self.name_of_target = prediction_input.name_of_target
        self.instance_result_dataframe = None
        self.quantile_portfolio_keys = None
        self.instance_quantile_portfolio = None
        self.unique_quantiles = None
        self._calculate_quantiles_has_run = False
        self._predict_probas_has_run = False
        self._add_results_to_aggregate_has_run = False

    def predict_probas(self):
        self._check_prediction_dataframe_is_set()
        if self.instance_result_dataframe is None:
            self._instantiate_instance_result_dataframe()
        adjusted_data_for_prediction = self.prediction_dataframe.loc[:, self.X_train_test_columns]
        predict_probas = self.model_estimator.predict_proba(adjusted_data_for_prediction)[:, 1]
        pred_proba_column_name = self._get_pred_proba_column_name()
        self.instance_result_dataframe[pred_proba_column_name] = predict_probas
        self._predict_probas_has_run = True

    def set_quantile_keys(self, quantile_portfolio: pd.DataFrame = None):
        if quantile_portfolio is None:
            self._check_quantile_portfolio_is_set()
            self.quantile_portfolio_keys = self.quantile_portfolio[self.key_name_in_dataframes].tolist()
        else:
            check_parameter_is_of_type(parameter=quantile_portfolio, parameter_type=pd.DataFrame)
            self.quantile_portfolio_keys = quantile_portfolio[self.key_name_in_dataframes].tolist()

    def _check_quantile_portfolio_is_set(self):
        # This is the "Prediction" class attribute "quantile_portfolio"
        if self.quantile_portfolio is None:
            return FileNotFoundError('quantile_portfolio is not set yet !')

    def calculate_quantiles(self):
        self._check_instance_result_dataframe_is_set()
        self._check_predict_probas_has_run()
        pred_proba_column_name = self._get_pred_proba_column_name()
        # Transfer predict_probas() from monthly prediction data into the instance_quantile_portfolio
        self.instance_quantile_portfolio = \
            self.instance_result_dataframe[
            self.instance_result_dataframe[self.key_name_in_dataframes].isin(self.quantile_portfolio_keys)]
        num_bins = 1000
        # Calculate quantile bins in the instance_quantile_portfolio
        quantile_bin = pd.qcut(self.instance_quantile_portfolio[pred_proba_column_name], num_bins,
                               labels=None, duplicates='drop', precision=5)
        # Calculate quantiles in the instance_quantile_portfolio
        quantile = pd.qcut(self.instance_quantile_portfolio[pred_proba_column_name], num_bins, labels=False,
                           duplicates='drop', precision=5) * 0.10
        bin_col_name = pred_proba_column_name + '_bin'
        quantile_col_name = pred_proba_column_name + '_quantile'
        # Store quantile_col_name for later usage in the transfer model. Storage must be at the
        # Prediction class level (not the instance level),  otherwise individual instances cannot be
        # aggregated consistently
        Prediction.quantile_columns.append(quantile_col_name)
        # Add bin and quantile values to the quantile_portfolio
        self.instance_quantile_portfolio[bin_col_name] = quantile_bin
        self.instance_quantile_portfolio[quantile_col_name] = quantile
        # Pack both (bin and quantile values) into a DataFrame and remove duplicates
        self.unique_quantiles = pd.DataFrame(data={bin_col_name: quantile_bin,
                                                   quantile_col_name: quantile}).drop_duplicates(inplace=False)
        # Allocate quantile bins from the instance_quantile_portfolio into the instance_result_dataframe
        self.instance_result_dataframe[bin_col_name] =
                                        pd.cut(self.instance_result_dataframe[pred_proba_column_name],
                                               bins=quantile_bin.cat.categories,
                                               duplicates='drop', include_lowest=True,
                                               right=True,
                                               precision=5)
        # Now also allocate the quantiles corresponding to the bins into the instance_result_dataframe
        self.instance_result_dataframe = pd.merge(self.instance_result_dataframe, self.unique_quantiles,
                                                  on=bin_col_name,
                                                  how='left')
        self._calculate_quantiles_has_run = True
```

Class Prediction continued

```
def get_pred_proba_for_key(self, key_value: str) -> str or KeyError:
    self._check_instance_result_dataframe_is_set()
    self._check_predict_probas_has_run()
    try:
        pred_proba_column_name = self._get_pred_proba_column_name()
        return self.instance_result_dataframe[
            self.instance_result_dataframe[self.key_name_in_dataframes] == key_value][
            pred_proba_column_name].values[0]
    except KeyError:
        return f'"predict_proba" not found for {self.key_name_in_dataframes}: {key_value!s} !'

def add_results_to_aggregate(self):
    # Here we have to aggregate at the OutsideAccess class level (not the Prediction instance level),
    # otherwise individual instances cannot be aggregated consistently and the results cannot be saved
    self._check_predict_probas_has_run()
    self._check_calculate_quantiles_has_run()
    try:
        if OutsideAccess.result_dataframe is None:
            OutsideAccess.result_dataframe = self.instance_result_dataframe
        else:
            OutsideAccess.result_dataframe = pd.merge(OutsideAccess.result_dataframe,
                                      self.instance_result_dataframe,
                                      on=self.key_name_in_dataframes,
                                      how='left')
        if OutsideAccess.quantile_portfolio is None:
            OutsideAccess.quantile_portfolio = self.instance_quantile_portfolio
        else:
            OutsideAccess.quantile_portfolio = pd.merge(OutsideAccess.quantile_portfolio,
                                       self.instance_quantile_portfolio,
                                       on=self.key_name_in_dataframes,
                                       how='left')
    except ValueError:
        print('Could not aggregate results. Check if instance and class dataframes have same shape !')
    self._add_results_to_aggregate_has_run = True

def generate_transfer_model_inputs(self):
    # Here we get the (aggregated) OutsideAccess class attributes, as the instance attributes only
    # contain individual (not aggregated) instance results and quantiles
    self._check_add_results_to_aggregate_has_run()
    transfer_model_target_columns = self.quantile_columns.copy()
    transfer_model_target_columns.insert(0, self.key_name_in_dataframes)
    targets = self.result_dataframe.loc[:, transfer_model_target_columns]
    self.transfer_model_input_object = TransferModelInput(transfer_model_targets=targets,
                                        name_of_target=self.name_of_target,
                                      key_name_in_dataframes=self.key_name_in_dataframes)

def _check_instance_result_dataframe_is_set(self):
    if self.instance_result_dataframe is None or \
            self.key_name_in_dataframes not in self.instance_result_dataframe.columns:
        return FileNotFoundError('instance_result_dataframe is not set yet !')

def _check_prediction_dataframe_is_set(self):
    if self.prediction_dataframe is None:
        return FileNotFoundError('prediction_dataframe is not set yet !')

def _check_features_in_prediction_dataframe(self):
    feat_not_in = [feat for feat in self.X_train_test_columns if feat not in
                   self.prediction_dataframe.columns]
    if len(feat_not_in) > 0 or self.key_name_in_dataframes not in self.prediction_dataframe.columns:
        raise TypeError(
            f'Either the required features {feat_not_in!r} or the key {self.key_name_in_dataframes}'
            f' are missing in the prediction_dataframe !')

def _instantiate_instance_result_dataframe(self):
    self._check_prediction_dataframe_is_set()
    self._check_features_in_prediction_dataframe()
    self.instance_result_dataframe = pd.DataFrame(self.prediction_dataframe.loc[:,
                                      self.key_name_in_dataframes])

def _check_calculate_quantiles_has_run(self):
    if not self._calculate_quantiles_has_run:
        raise ValueError('Run calculate_quantiles() method first !')

def _check_predict_probas_has_run(self):
    if not self._predict_probas_has_run:
        raise ValueError('Run predict_probas() method first !')

def _check_add_results_to_aggregate_has_run(self):
    if not self._add_results_to_aggregate_has_run:
        raise ValueError('Run add_results_to_aggregate() method first !')
```

```
def _get_pred_proba_column_name(self):
    if self.estimator_is_pipeline:
        estimator_name = 'Pipeline_' + self.model_estimator.named_steps['estimator'].__class__.__name__
    else:
        estimator_name = self.model_estimator.__class__.__name__
    return 'predict_probas_' + estimator_name
```

9. Classes Explanation and Shap

```
class Explanation(InsideAccess):

  def __init__(self, explanation_input: ExplanationInput):
    check_parameter_is_of_type(parameter=explanation_input, parameter_type=ExplanationInput)
    super().__init__()
    self.model_estimator = explanation_input.model_best_estimator
    self.X = explanation_input.X
    self.X_train = explanation_input.X_train
    self.X_test = explanation_input.X_test
    self.key_name_in_dataframes = explanation_input.key_name_in_dataframes
    self.second_key_name_in_dataframes = explanation_input.second_key_name_in_dataframes
    self.keys = None
    self.key_values = None


class Shap(Explanation):
  def __init__(self, explanation_input: ExplanationInput):
    super().__init__ (explanation_input=explanation_input)
    self.shap_explainer_object = None
    self.shap_explanation_object = None
    self.shap_explanation_data = None
    self.shap_explanation_data_all = None
    self.global_explanation_object = None
    self.global_base_value = None
    self.global_shap_values = None
    self.shap_data_set = None
    self._create_shap_objects_has_run = False
    self._calc_global_explanation_has_run = False

  def create_shap_objects(self):
    self._get_shap_explanation_data()
    self._calc_keys()
    self._calc_key_values()
    self._reduce_shap_explanation_data_rows()
    # Create SHAP-Explainer and SHAP-Explanation objects
    """ As of May 2021, the scikit-learn-method "predict_proba" is not yet supported for estimators
    other than XGBClassifier. The following error message occurs if "model_output" is set to
    "predict_proba" (Quote):
    "Exception: Unrecognized model_output parameter value: predict_proba! If model.predict_proba is a
    valid function open a github issue to ask that this method be supported. If you want 'predict_proba'
    just use 'probability' for now " (Quote end). Thus: Instead of "predict_proba" I use "probability"
    In case the estimator is not XGBClassifier. """
    if self.model_estimator.__class__.__name__ == 'XGBClassifier':
      model_output = 'predict_proba'
    else:
      model_output = 'probability'
    shap_background_data_num_samples =
          int(self.configuration.get_config_setting(config_section='Model.Explanation',
                                    config_name='shap_background_data_num_samples'))
    masker = shap.maskers.Independent(data=self.X_train,
                          max_samples=shap_background_data_num_samples)
    self.shap_explainer_object = shap.explainers.Tree(model=self.model_estimator,
                                  data=masker,
                                  feature_names=self.X_train.columns.to_list(),
                                  feature_perturbation="interventional",
                                  model_output=model_output)
    self.shap_explanation_object = self.shap_explainer_object(self.shap_explanation_data)
    self._create_shap_objects_has_run = True

  def calc_global_explanation(self, target_class: int = 1):
    self._check_create_shap_objects_has_run()
    check_parameter_is_of_type(parameter=target_class, parameter_type=int)
    # Define global variables for class = target class (usually the target class is 1 (or in seldom
    # cases 0))
    self.global_explanation_object = self.shap_explanation_object[:, :, target_class]
    self.global_base_value = self.shap_explainer_object.expected_value[target_class]
    self.global_shap_values = pd.DataFrame(data=self.shap_explanation_object.values[:, :, target_class],
                            index=self.shap_explanation_data.index,
                            columns=self.shap_explanation_data.columns)
    self._calc_global_explanation_has_run = True

  def _check_index_is_in_shap_explanation_data(self, key_value, second_key_value=None):
    key_value_index = self._get_index_of_key_value(key_value=key_value,
                            second_key_value=second_key_value)
    if key_value_index not in self.shap_explanation_data.index:
      raise ValueError(f'KEY ERROR: The {self.key_name_in_dataframes} = {key_value!s} cannot be found
          f'in the shap_explanation_data and thus also not in the global_shap_values. The
          f'{self.key_name_in_dataframes} = {key_value!s} was probably excluded when the
          data size was reduced to the size given in the config-file as
          "shap_max_num_explanation_data_rows" !')
```

Class Shap continued

```
def get_local_shap_values_from_global(self, key_value, second_key_value=None) -> np.ndarray:
    self._check_create_shap_objects_has_run()
    self._check_calc_global_explanation_has_run()
    self._check_index_is_in_shap_explanation_data(key_value=key_value,
                                                  second_key_value=second_key_value)
    key_value_index = self._get_index_of_key_value(key_value=key_value,
                                                    second_key_value=second_key_value)
    return self.global_shap_values[self.global_shap_values.index == key_value_index]

def get_local_shap_values_from_method(self, key_value, second_key_value=None,
                                      target_class: int = 1) -> pd.DataFrame:
    self._check_create_shap_objects_has_run()
    self._check_calc_global_explanation_has_run()
    self._check_index_is_in_shap_explanation_data_all(key_value=key_value,
                                                      second_key_value=second_key_value)
    key_value_index = self._get_index_of_key_value(key_value=key_value,
                                                    second_key_value=second_key_value)
    data_row_to_calc_shap_for = self.shap_explanation_data_all[
                                self.shap_explanation_data_all.index == key_value_index].to_numpy()
    data_row_shap_values =
                    self.shap_explainer_object.shap_values(data_row_to_calc_shap_for)[target_class]
    return pd.DataFrame(data=data_row_shap_values,
                        index=self.shap_explanation_data_all.index[
                            self.shap_explanation_data_all.index == key_value_index],
                        columns=self.shap_explanation_data_all.columns)

def compare_local_shap_from_global_with_local_shap_from_method(self, key_value, second_key_value=None,
                                                               target_class: int = 1) -> pd.Series or
                                                               pd.DataFrame:
    """ As also noted in the Notebook files in "FRAUD" (old), there is an issue with the local shap
    values as there is a difference between local shap values coming from global and local shap values
    coming from the method. With this method, the difference can be shown. The error should be corrected
    in newer versions of shap. """
    local_from_global = self.get_local_shap_values_from_global(key_value=key_value,
                                                               second_key_value=second_key_value)
    local_from_method = self.get_local_shap_values_from_method(key_value=key_value,
                                                               second_key_value=second_key_value,
                                                               target_class=target_class)
    if local_from_global.shape == local_from_method.shape:
        print('The difference in shap-values between local_from_method and local_from_global is shown
              here.'
              'The SHAP-value for the following feature(s) should be higher(+)/lower(-) by this amount
              in the '
              'local_from_global-Plots: ')
        diff_method_global = (local_from_method - local_from_global).T
        return diff_method_global[diff_method_global.values != 0]
    else:
        return f'The DataFrames are different in their shape: Shape of local_from_global is ' \
               f'{local_from_global.shape} and shape of local_from_method is {local_from_method.shape}.
               ' \
               f'They thus cannot be compared !'

def plot_global_bars(self, num_feat_shown: int = 12, save_plot_as_pdf: bool = False):
    self._check_create_shap_objects_has_run()
    self._check_calc_global_explanation_has_run()
    shap.plots.bar(shap_values=self.global_explanation_object, max_display=num_feat_shown, show=False)
    if save_plot_as_pdf:
        plot_name = 'global_bar_plot_' + self.model_estimator.__class__.__name__ + '.pdf'
        plt.savefig(plot_name, format='pdf', dpi=1200, bbox_inches='tight')

def plot_global_beeswarm(self, num_feat_shown: int = 12, save_plot_as_pdf: bool = False):
    self._check_create_shap_objects_has_run()
    self._check_calc_global_explanation_has_run()
    shap.plots.beeswarm(shap_values=self.global_explanation_object, max_display=num_feat_shown,
                        show=False)
    if save_plot_as_pdf:
        plot_name = 'global_beeswarm_plot_' + self.model_estimator.__class__.__name__ + '.pdf'
        plt.savefig(plot_name, format='pdf', dpi=1200, bbox_inches='tight')

def plot_global_heatmap(self, num_feat_shown: int = 12, save_plot_as_pdf: bool = False):
    self._check_create_shap_objects_has_run()
    self._check_calc_global_explanation_has_run()
    shap.plots.heatmap(shap_values=self.global_explanation_object, max_display=num_feat_shown)
    if save_plot_as_pdf:
        plot_name = 'global_heatmap_plot_' + self.model_estimator.__class__.__name__ + '.pdf'
        plt.savefig(plot_name, format='pdf', dpi=1200, bbox_inches='tight')
```

Class Shap continued

```
def plot_global_scatter_for_features(self, feature_name_one: str,
                                     display_only_feat_one: bool = True,
                                     feature_name_two: str = None,
                                     save_plot_as_pdf: bool = False):
    """ Adjusted quote from the SHAP website: "If 'feature_name_two' is 'None' and
    'display_only_feat_one' is 'True', this plot only scatters the SHAP values for 'feature_name_one'.
     If 'feature_name_two' is 'None' and 'display_only_feat_one' is 'False', then the scatter plot
    points are colored by the feature that seems to have the strongest interaction effect with the first
    feature. If 'feature_name_two' is another feature name,then the scatter plot points are colored by
    this second feature." """
    self._check_create_shap_objects_has_run()
    self._check_calc_global_explanation_has_run()
    check_parameter_is_of_type(parameter=feature_name_one, parameter_type=str)
    if feature_name_two is None and display_only_feat_one:
        color = None
    elif feature_name_two is None and not display_only_feat_one:
        color = self.global_explanation_object
    else:
        check_parameter_is_of_type(parameter=feature_name_two, parameter_type=str)
        color = self.global_explanation_object[:, feature_name_two]
    shap.plots.scatter(shap_values=self.global_explanation_object[:, feature_name_one],
                       color=color,
                       show=False)
    if save_plot_as_pdf:
        plot_name = 'global_scatter_plot_' + self.model_estimator.__class__.__name__ + '.pdf'
        plt.savefig(plot_name, format='pdf', dpi=1200, bbox_inches='tight')

def plot_local_force(self, key_value, second_key_value=None, target_class: int = 1,
                     contribution_threshold: float = 0.02,
                     save_plot_as_pdf: bool = False):
    local_shap_values = self.get_local_shap_values_from_method(key_value=key_value,
                                                               target_class=target_class,
                                                               second_key_value=second_key_value)
    key_value_index = self._get_index_of_key_value(key_value=key_value,
                                                    second_key_value=second_key_value)
    shap.plots.force(base_value=np.around(self.global_base_value, decimals=2),
                     shap_values=np.around(local_shap_values.to_numpy(),decimals=2),
                     feature_names=self.shap_explanation_data_all.columns.to_list(),
                     features=np.around(self.shap_explanation_data_all[
                         self.shap_explanation_data_all.index == key_value_index], decimals=2),
                     matplotlib=True, show=False, contribution_threshold=contribution_threshold)
    if save_plot_as_pdf:
        plot_name = 'local_force_plot_' + self.model_estimator.__class__.__name__ + key_value + '.pdf'
        plt.savefig(plot_name, format='pdf', dpi=1200, bbox_inches='tight')

def plot_local_bars_from_global(self, key_value, second_key_value=None,
                                num_feat_shown: int = 12,
                                save_plot_as_pdf: bool = False):
    local_explanation_object = self._get_local_explanation_object(key_value=key_value,
                                                                   second_key_value=second_key_value)
    shap.plots.bar(local_explanation_object, max_display=num_feat_shown, show=False)
    if save_plot_as_pdf:
        plot_name = 'local_bar_plot_' + self.model_estimator.__class__.__name__ + key_value + '.pdf'
        plt.savefig(plot_name, format='pdf', dpi=1200, bbox_inches='tight')

def plot_local_waterfall_from_global(self, key_value, second_key_value=None,
                                     num_feat_shown: int = 12,
                                     save_plot_as_pdf: bool = False):
    local_explanation_object = self._get_local_explanation_object(key_value=key_value,
                                                                   second_key_value=second_key_value)
    shap.plots.waterfall(local_explanation_object, max_display=num_feat_shown)
    if save_plot_as_pdf:
        plot_name = 'local_waterfall_plot_' + self.model_estimator.__class__.__name__ + key_value +
                    '.pdf'
        plt.savefig(plot_name, format='pdf', dpi=1200, bbox_inches='tight')

def _get_local_explanation_object(self, key_value, second_key_value=None):
    self._check_create_shap_objects_has_run()
    self._check_calc_global_explanation_has_run()
    self._check_index_is_in_shap_explanation_data(key_value=key_value,
                                                  second_key_value=second_key_value)
    key_value_index = int(self._get_index_of_key_value(key_value=key_value,
                                                        second_key_value=second_key_value))
    row_number = self._get_row_number_of_global_shap_values(index=key_value_index)
    local_explanation_object = self.global_explanation_object[row_number, :]
    return local_explanation_object
```

Class Shap continued

```
def get_info_about_key_vals(self, key_value, second_key_value=None):
    key_value_index = self._get_index_of_key_value(key_value=key_value,
                                                    second_key_value=second_key_value)
    if second_key_value is not None:
        info_text_0 = f'INFO: The {self.key_name_in_dataframes} = {key_value} with ' \
                      f'{self.second_key_name_in_dataframes} = {second_key_value} '
    else:
        info_text_0 = f'INFO: The {self.key_name_in_dataframes} = {key_value} '
    if key_value_index in self.shap_explanation_data.index:
        info_text_1 = f'IS in the shap_explanation_data (reduced data set) '
    else:
        info_text_1 = f'IS NOT in the shap_explanation_data (reduced data set) '
    if key_value_index in self.shap_explanation_data_all.index:
        info_text_2 = f'and IS in the shap_explanation_data_all !'
    else:
        info_text_2 = f'and IS NOT in the shap_explanation_data_all !'
    if key_value_index not in self.shap_explanation_data.index and key_value_index not in
                                                    self.shap_explanation_data_all.index:
        info_text_3 = f' Thus, the searched data is NOT in the provided "{self.shap_data_set}" data set
                      !'
    else:
        info_text_3 = ''
    print(info_text_0 + info_text_1 + info_text_2 + info_text_3)

def _get_index_of_key_value(self, key_value, second_key_value) -> int or ValueError:
    if self.second_key_name_in_dataframes is not None and second_key_value is None:
        raise FileNotFoundError(
            f'Please provide a parameter value for the second_key "{self.second_key_name_in_dataframes}"
            !')
    if self.second_key_name_in_dataframes is None and second_key_value is not None:
        raise TypeError(
            f'The parameter value for the second_key cannot be processed! '
            f'Please omit the parameter value {second_key_value} !')
    if second_key_value is None:
        condition = (self.key_values == key_value)
    else:
        condition = (self.key_values[self.key_name_in_dataframes] == key_value) & \
                    (self.key_values[self.second_key_name_in_dataframes] == second_key_value)
    if len(self.key_values[condition]) == 1:
        return self.key_values[condition].index[0]
    else:
        raise ValueError(
            f'Either the row values for {self.key_name_in_dataframes} = {key_value} and '
            f'{self.second_key_name_in_dataframes} = {second_key_value} were not found or the row index
            for the '
            f'first is different from the row index of the latter. Please check your entries !')

def _get_row_number_of_global_shap_values(self, index: int) -> int:
    check_parameter_is_of_type(parameter=index, parameter_type=int)
    return np.where(self.global_shap_values.index == index)[0][0]

def _get_shap_explanation_data(self):
    self.shap_data_set = self.configuration.get_config_setting(config_section='Model.Explanation',
                                                    config_name='shap_data_for_explanation')
    if self.shap_data_set == 'train':
        self.shap_explanation_data = self.shap_explanation_data_all = self.X_train
    elif self.shap_data_set == 'test':
        self.shap_explanation_data = self.shap_explanation_data_all = self.X_test
    elif self.shap_data_set == 'monthly':
        self._check_data_for_prediction_is_set()
        self.shap_explanation_data = self.shap_explanation_data_all = self.data_for_prediction.loc[:,
                                                    self.X_train.columns.tolist()]
    else:
        raise FileNotFoundError(f'ERROR: {self.shap_data_set} not found or it is not a valid
                                                    shap_data_set name!')

def _calc_keys(self):
    if self.second_key_name_in_dataframes is None:
        self.keys = self.key_name_in_dataframes
    else:
        self.keys = list((self.key_name_in_dataframes, self.second_key_name_in_dataframes))

def _calc_key_values(self):
    if self.shap_data_set == 'monthly':
        self._check_data_for_prediction_is_set()
        self.key_values = self.data_for_prediction.loc[:, self.keys]
    else:
        self.key_values = self.X.loc[:, self.keys]
```

Class Shap continued

```
    def _reduce_shap_explanation_data_rows(self):
        max_num_data_rows_config = \
            int(self.configuration.get_config_setting(config_section='Model.Explanation',
                                                      config_name='shap_max_num_explanation_data_rows'))
        num_data_rows_in_shap_explanation_data = len(self.shap_explanation_data.index)
        num_data_rows = min(max_num_data_rows_config, num_data_rows_in_shap_explanation_data)
        if num_data_rows != num_data_rows_in_shap_explanation_data:
            print(f'The size of the shap_explanation_data is now reduced from
                {num_data_rows_in_shap_explanation_data}'
                f' rows to {num_data_rows} rows !')
        randomized_rows_list = random.sample(self.shap_explanation_data.index.to_list(), num_data_rows)
        randomized_index = self.shap_explanation_data.index.isin(randomized_rows_list)
        self.shap_explanation_data = self.shap_explanation_data[randomized_index]

    def _check_data_for_prediction_is_set(self):
        if self.data_for_prediction is None:
            raise TypeError('"data_for_prediction" not set yet. Please load it first !')

    def _check_shap_dataset_is_set(self):
        if self.shap_explanation_data is None:
            raise TypeError('"shap_explanation_data" not set yet. Please run create_shap_objects()-method
                    first !')

    def _check_create_shap_objects_has_run(self):
        if not self._create_shap_objects_has_run:
            raise TypeError('Run create_shap_objects() method first !')

    def _check_calc_global_explanation_has_run(self):
        if not self._calc_global_explanation_has_run:
            raise TypeError('Run calc_global_explanation() method first !')
```

10. functions.py

```
PredictionInput = namedtuple('PredictionInput', (
    'key_name_in_dataframes', 'model_best_estimator', 'estimator_is_pipeline', 'features_in_training',
    'name_of_target'))
TransferModelInput = namedtuple('TransferModelInput',
                                ('transfer_model_targets', 'name_of_target', 'key_name_in_dataframes'))
ExplanationInput = namedtuple('ExplanationInput', (
    'model_best_estimator', 'X', 'X_train', 'X_test', 'key_name_in_dataframes',
    'second_key_name_in_dataframes'))


def check_parameter_is_of_type(parameter: any, parameter_type: object):
    error_message = f'Provided parameter {parameter!s} is not of required type {parameter_type!s}. Please
                    check !'
    if isinstance(parameter_type, list):
        if not type(parameter) in parameter_type:
            raise TypeError(error_message)
    else:
        if not type(parameter) is parameter_type:
            raise TypeError(error_message)


def dataframe_col_name_to_num_list(dataframe: pd.DataFrame, col_names: list) -> list:
    check_parameter_is_of_type(parameter=dataframe, parameter_type=pd.DataFrame)
    check_parameter_is_of_type(parameter=col_names, parameter_type=list)
    return [dataframe.columns.get_loc(col) if (type(col) == str) else col for col in
            col_names]


def check_list_has_elements(parameter_list: list):
    if len(parameter_list) == 0:
        raise IndexError(f'Provided parameter "{parameter_list!s}" has no elements. Please check !')


def get_key_values_where_index_is_in_both_dataframes(df_one: pd.DataFrame, df_two: pd.DataFrame,
                                                     key_in_df_one: str,
                                                     second_key_in_df_one: str = None) -> list:
    check_parameter_is_of_type(parameter=df_one, parameter_type=pd.DataFrame)
    check_parameter_is_of_type(parameter=df_two, parameter_type=pd.DataFrame)
    check_parameter_is_of_type(parameter=key_in_df_one, parameter_type=str)
    if second_key_in_df_one is None:
        return df_one[df_one.index.isin(df_two.index)][key_in_df_one].values.tolist()
    else:
        check_parameter_is_of_type(parameter=second_key_in_df_one, parameter_type=str)
        return df_one[df_one.index.isin(df_two.index)].loc[:, [key_in_df_one,
                                                               second_key_in_df_one]].values.tolist()


def make_param_grid_for_pipe_gridsearch(name_of_step_in_pipe: str, est_params: dict,
                                        existent_pipe_param_grid: dict = None):
    # Parameters of pipelines can be set using '__' plus the name of the parameter:
    parameter_grid = {}
    new_grid = {name_of_step_in_pipe + '__' + k: v for k, v in est_params.items()}
    if existent_pipe_param_grid:
        parameter_grid.update(existent_pipe_param_grid)
    parameter_grid.update(new_grid)
    return parameter_grid


def winsorize_for_function_transformer(np_array, lower_bound: float = 0.00, upper_bound: float = 1.00):
    # Columns are not passed here as they are passed in the transformer tuple.
    # As the transformer currently (March 2021) only works on numpy arrays and cannot iterate over columns,
    # we must adjust the winsorize-function here.
    lower = np.quantile(np_array, lower_bound, axis=0, interpolation='higher')
    upper = np.quantile(np_array, upper_bound, axis=0, interpolation='lower')
    np_array = np.clip(np_array, lower, upper)
    return np_array
```

functions.py continued

```
def make_preprocessing_pipeline_for_cv_with_function(unfitted_estimator,
                                    impute_columns: list = None,
                                    winsorize_columns: list = None,
                                    scale_columns: list = None,
                                    impute_strategy: str = 'median',
                                                                    impute_knn_num_neighbors: int = 3,
                                    scale_strategy: str = 'standard',
                                    scale_min_max_range: tuple = (0, 1),
                                    scale_robust_quantile_range: tuple = (25, 75),
                                    winsorize_lower_bound: float = 0.00,
                                    winsorize_upper_bound: float = 1.00) -> Pipeline:
  # (integers) Possible impute_strategies = ['median', 'mean', 'most_frequent', 'constant', 'knn']
  impute_set = set(impute_columns) if impute_columns is not None else set()
  winsorize_set = set(winsorize_columns) if winsorize_columns is not None else set()
  scale_set = set(scale_columns) if scale_columns is not None else set()
  impute_winsorize_scale_set = impute_set & winsorize_set & scale_set
  impute_winsorize_scale_list = list(impute_winsorize_scale_set)
  impute_only_list = list(impute_set - winsorize_set - scale_set)
  scale_only_list = list(scale_set - impute_set - winsorize_set)
  winsorize_only_list = list(winsorize_set - impute_set - scale_set)
  impute_scale_list = list(impute_set.intersection(impute_set, scale_set) - impute_winsorize_scale_set)
  impute_winsorize_list = list(impute_set.intersection(winsorize_set) - impute_winsorize_scale_set)
  winsorize_scale_list = list(winsorize_set.intersection(scale_set) - impute_winsorize_scale_set)

  # No1: Imputers
  if impute_strategy == 'knn':
     impute_function = KNNImputer(n_neighbors=impute_knn_num_neighbors, missing_values=np.nan)
  else:
     impute_function = SimpleImputer(strategy=impute_strategy, copy=False, missing_values=np.nan)

  # No2: Winsorizer
  # Takes function, in this case the function winsorize_for_function_transformer() from above
  winsorize_function = FunctionTransformer(func=winsorize_for_function_transformer,
                                  kw_args={'lower_bound': winsorize_lower_bound,
                                           'upper_bound': winsorize_upper_bound})

  # No3: Scalers
  scalers = {'standard': StandardScaler(copy=False, with_mean=True, with_std=True),
          'robust': RobustScaler(with_centering=True, with_scaling=True,
                                 quantile_range=scale_robust_quantile_range, copy=False,
                                 unit_variance=False),
          'maxabs': MaxAbsScaler(copy=False),
          'minmax': MinMaxScaler(feature_range=scale_min_max_range, copy=False, clip=False),
          'power': PowerTransformer(copy=False, method='yeo-johnson', standardize=True)}
  scale_function = scalers.get(scale_strategy, 'Error: The scale_function could not be found')
  preprocess_pipeline_steps = []
  if len(impute_winsorize_scale_list) > 0:
     impute_winsorize_scale = Pipeline(
        steps=[('impute', impute_function),
               ('winsorize', winsorize_function),
               ('scale', scale_function)])
     impute_winsorize_scale_tuple = ('impute_winsorize_scale', impute_winsorize_scale,
                                     impute_winsorize_scale_list)
     preprocess_pipeline_steps.append(impute_winsorize_scale_tuple)

  if len(impute_winsorize_list) > 0:
     impute_winsorize = Pipeline(
        steps=[('impute', impute_function),
               ('winsorize', winsorize_function)])
     impute_winsorize_tuple = ('impute_winsorize', impute_winsorize, impute_winsorize_list)
     preprocess_pipeline_steps.append(impute_winsorize_tuple)

  if len(impute_scale_list) > 0:
     impute_scale = Pipeline(
        steps=[('impute', impute_function),
               ('scale', scale_function)])
     impute_scale_tuple = ('impute_scale', impute_scale, impute_scale_list)
     preprocess_pipeline_steps.append(impute_scale_tuple)

  if len(winsorize_scale_list) > 0:
     winsorize_scale = Pipeline(
        steps=[('winsorize', winsorize_function),
               ('scale', scale_function)])
     winsorize_scale_tuple = ('winsorize_scale', winsorize_scale, winsorize_scale_list)
     preprocess_pipeline_steps.append(winsorize_scale_tuple)

  if len(impute_only_list) > 0:
     impute_only_tuple = ('impute_only', impute_function, impute_only_list)
     preprocess_pipeline_steps.append(impute_only_tuple)

  if len(winsorize_only_list) > 0:
     winsorize_only_tuple = ('winsorize_only', winsorize_function, winsorize_only_list)
     preprocess_pipeline_steps.append(winsorize_only_tuple)

  if len(scale_only_list) > 0:
     scale_only_tuple = ('scale_only', scale_function, scale_only_list)
     preprocess_pipeline_steps.append(scale_only_tuple)

  preprocess_transformer = ColumnTransformer(transformers=preprocess_pipeline_steps, remainder='passthrough')

  if preprocess_transformer:
     return Pipeline(steps=[('preprocess_transformer', preprocess_transformer), ('estimator', unfitted_estimator)])
  else:
     raise ValueError(
        'The preprocess_transformer contains no data. preprocess_transformer is: '.format(preprocess_transformer))
```

11. config.ini

```
[DEFAULT]
base_path = not disclosed for confidentiality reasons

[Model.General]
test_set_size = 0.3
num_cv_folds = 5
scoring = ['roc_auc']

[Model.CV.Preprocessing]
# In the respective columns, fill in the feature names in parenthesis ('') separated by comma (,)
# If all columns shall be processed, replace empty list bracket by 'All'
impute = {'impute_columns': [], 'impute_strategy': 'median', 'impute_knn_num_neighbors': 3}
winsorize = {'winsorize_columns': [], 'winsorize_lower_bound': 0.00, 'winsorize_upper_bound': 1.00 }
scale = {'scale_columns': 'All', 'scale_strategy': 'standard', 'scale_min_max_range': (0, 1), 'scale_robust_quantile_range':
(25, 75)}

[Model.Hyperparams]
logit = {'C': [0.1], 'class_weight': [None], 'dual': [False], 'fit_intercept': [True], 'intercept_scaling': [1], 'l1_ratio':
[None], 'max_iter': [100], 'multi_class': ['auto'], 'n_jobs': [None], 'penalty': ['l2'], 'random_state': [None], 'solver':
['lbfgs'], 'tol': [0.0001], 'verbose': [0], 'warm_start': [False]}
rfc = {'bootstrap': [True], 'ccp_alpha': [0.0], 'class_weight': [None], 'criterion': ['gini'], 'max_depth': [None],
'max_features': ['log2'], 'max_leaf_nodes': [None], 'max_samples': [None], 'min_impurity_decrease': [0.0],
'min_impurity_split': [None], 'min_samples_leaf': [10], 'min_samples_split': [20], 'min_weight_fraction_leaf': [0.0],
'n_estimators': [100], 'n_jobs': [None], 'oob_score': [False], 'random_state': [None], 'verbose': [0], 'warm_start': [False]}
xgb = {'objective': ['binary:logistic'], 'base_score': [None], 'booster': [None], 'colsample_bylevel': [None],
'colsample_bynode': [None], 'colsample_bytree': [None], 'gamma': [None], 'gpu_id': [None], 'importance_type': ['gain'],
'interaction_constraints': [None], 'learning_rate': [None], 'max_delta_step': [None], 'max_depth': [None], 'min_child_weight':
[10], 'missing': [np.nan], 'monotone_constraints': [None], 'n_estimators': [200], 'n_jobs': [None], 'num_parallel_tree':
[None], 'random_state': [None], 'reg_alpha': [None], 'reg_lambda': [None], 'scale_pos_weight': [None], 'subsample': [None],
'tree_method': [None], 'validate_parameters': [None], 'verbosity': [None]}

[Model.Explanation]
shap_background_data_num_samples = 100
# For what set shall SHAP-values be calculated? test set ("test"), train set ("train") or monthly ("monthly") data?:
shap_data_for_explanation = train
shap_max_num_explanation_data_rows = 20000

[Original.Model.Training]
file_path = %(base_path)s/OriginalModel/Training/
learn_data_preprocessed = orig_anonym_learn_data_preprocessed.csv

[Original.Model.Monthly]
file_path =  %(base_path)s/OriginalModel/Monthly/
data_for_prediction = prediction_data_anonym_202009.csv
result_dataframe = result_dataframe_anonym.csv
quantile_portfolio = quantile_portfolio_anonym_202009.csv
prediction_input_object =  {'RFC': 'prediction_input_object_rfc.joblib', 'Logit': 'prediction_input_object_logit.joblib',
'XGB': 'prediction_input_object_xgb.joblib'}

[Original.Model.Models]
file_path =  %(base_path)s/OriginalModel/Models/
logit = logit_anonym.joblib
rfc = rfc_anonym.joblib
xgb = xgb_anonym.joblib

[Original.Model.Explanation]
file_path = %(base_path)s/OriginalModel/Explanation/
explanation_input_object =  {'RFC': 'explanation_input_object_rfc.joblib', 'Logit': 'explanation_input_object_logit.joblib',
'XGB': 'explanation_input_object_xgb.joblib'}

[Transfer.Model.Cutoff]
estimators_for_cutoff = ['RandomForestClassifier', 'LogisticRegression', 'XGBClassifier']
cutoff_threshold = 90.0

[Transfer.Model.Training]
file_path =  %(base_path)s/TransferModel/Training/
learn_data_raw_part_1 = zv_1_anonym_202009.csv
learn_data_raw_part_3 = zv_3_anonym_202009.csv
learn_data_preprocessed = zv_1_3_anonym_preprocessed_202009.csv
learn_data_feat_engineered = zv_1_3_anonym_feat_engineered_202009.csv
transfer_model_input_object = {'RFC': 'transfer_model_input_object_rfc.joblib', 'Logit':
'transfer_model_input_object_logit.joblib', 'XGB': 'transfer_model_input_object_xgb.joblib'}

[Transfer.Model.Models]
file_path =  %(base_path)s/TransferModel/Models/
logit = logit_anonym.joblib
rfc = rfc_anonym.joblib
xgb = xgb_anonym.joblib

[Transfer.Model.Explanation]
file_path = %(base_path)s/TransferModel/Explanation/
explanation_input_object =  {'RFC': 'explanation_input_object_rfc_202009.joblib', 'Logit':
'explanation_input_object_logit_202009.joblib', 'XGB': 'explanation_input_object_xgb_202009.joblib'}
```

12. feature indexes

```
ensemble_index = ['orig_key_1', 'orig_key_3', 'orig_key_2', 'orig_feat_133', 'orig_feat_36',
                  'orig_feat_99', 'orig_feat_3', 'orig_feat_74', 'orig_feat_97', 'orig_feat_5',
                  'orig_feat_109', 'orig_feat_21', 'orig_feat_104', 'orig_feat_22', 'orig_feat_78',
                  'orig_feat_147', 'orig_feat_187', 'orig_feat_7', 'orig_feat_4', 'orig_feat_2',
                  'orig_feat_145', 'orig_feat_75', 'orig_feat_82', 'orig_feat_60', 'orig_feat_88',
                  'orig_feat_1', 'orig_feat_29', 'orig_feat_98', 'orig_feat_76', 'orig_feat_124',
                  'orig_feat_47', 'orig_feat_25', 'orig_feat_144', 'orig_feat_155', 'orig_feat_31']

logit_index = ['orig_key_1', 'orig_key_3', 'orig_key_2', 'orig_feat_36', 'orig_feat_187',
               'orig_feat_239', 'orig_feat_526', 'orig_feat_602', 'orig_feat_39', 'orig_feat_590',
               'orig_feat_530', 'orig_feat_155', 'orig_feat_527', 'orig_feat_528', 'orig_feat_579',
               'orig_feat_29', 'orig_feat_60', 'orig_feat_50', 'orig_feat_237', 'orig_feat_392',
               'orig_feat_546', 'orig_feat_569', 'orig_feat_511', 'orig_feat_354', 'orig_feat_47',
               'orig_feat_16', 'orig_feat_25', 'orig_feat_28', 'orig_feat_591', 'orig_feat_345',
               'orig_feat_488', 'orig_feat_72', 'orig_feat_474', 'orig_feat_348', 'orig_feat_62',
               'orig_feat_346', 'orig_feat_603', 'orig_feat_363', 'orig_feat_4', 'orig_feat_303',
               'orig_feat_552', 'orig_feat_572', 'orig_feat_7', 'orig_feat_485', 'orig_feat_323',
               'orig_feat_107', 'orig_feat_631', 'orig_feat_547', 'orig_feat_409', 'orig_feat_466',
               'orig_feat_643', 'orig_feat_571', 'orig_feat_182', 'orig_feat_356', 'orig_feat_604',
               'orig_feat_31', 'orig_feat_573', 'orig_feat_421', 'orig_feat_26', 'orig_feat_338',
               'orig_feat_465', 'orig_feat_171', 'orig_feat_109', 'orig_feat_594', 'orig_feat_640',
               'orig_feat_252', 'orig_feat_621', 'orig_feat_534', 'orig_feat_623', 'orig_feat_203',
               'orig_feat_646', 'orig_feat_543', 'orig_feat_124', 'orig_feat_42', 'orig_feat_248',
               'orig_feat_164', 'orig_feat_365', 'orig_feat_58', 'orig_feat_126', 'orig_feat_24',
               'orig_feat_599', 'orig_feat_5', 'orig_feat_128', 'orig_feat_322', 'orig_feat_377',
               'orig_feat_328', 'orig_feat_184', 'orig_feat_629', 'orig_feat_176', 'orig_feat_650',
               'orig_feat_567', 'orig_feat_144', 'orig_feat_378', 'orig_feat_468', 'orig_feat_98',
               'orig_feat_376', 'orig_feat_563', 'orig_feat_371', 'orig_feat_514', 'orig_feat_440',
               'orig_feat_544', 'orig_feat_472']
```