From Data to Decisions Explainable AI in Credit Approval

February 14, 2025

1 From Data to Decisions: Explainable AI in Credit Approval

1.1 Introduction

In the era of AI-driven decision-making, understanding how a model arrives at a specific decision is crucial, especially in sensitive areas such as finance. Explainable AI (XAI) provides tools and methods that allow both developers and end-users to understand, trust, and effectively use AI models. XAI methods aim to make the decision-making process of machine learning models more transparent and interpretable for different stakeholders.

Refer to the following links for detailed documentation and resources:

- IBM AI Explainability 360 (AIX360)
- IBM AIX360 Github
- IBM AIX360 Documentation
- AIX360 Credit Approval Tutorial.

1.2 Objectives

By the end of this project, you will be able to:

1. **Set up the environment**: For working with explainable AI (XAI) methods on the HELOC dataset.
2. **Load and explore the HELOC dataset**: Gain insights into the data attributes related to credit applications.
3. **Apply XAI techniques for different perspectives**: Use a range of explainability methods tailored to three perspectives — data scientists (BRCG and LogRR), loan officers (ProtoDash), and customers (CEM) — to make model predictions more interpretable.
4. **Evaluate model interpretability**: Analyze the effectiveness of each XAI method in enhancing transparency for credit decision-making.

This project will equip you with practical skills in applying XAI tools to meet diverse user requirements in credit risk assessment.

1.3 Setup

- `pandas` for data manipulation and analysis, especially to manage and preprocess the HELOC dataset.
- `numpy` for numerical computations and handling array-based data.
- `matplotlib` and `seaborn` for visualizing data distributions and model explanations.

- `aix360` for explainability techniques.
- `sklearn` for machine learning utilities, including model evaluation and preprocessing tools.

1.3.1 Importing required libraries

[1]:
```
!pip install aix360==0.3.0 | tail -n 1
!pip install openpyxl==3.1.5 | tail -n 1
!pip install cvxpy==1.5.3 | tail -n 1
!pip install tensorflow==2.18.0 | tail -n 1
!pip install --no-deps xport==3.6.1 | tail -n 1
!pip install pandas==2.2.3 | tail -n 1
```

```
Successfully installed aix360-0.3.0 contourpy-1.3.1 cycler-0.12.1
fonttools-4.56.0 joblib-1.4.2 kiwisolver-1.4.8 matplotlib-3.10.0 numpy-2.2.3
pandas-2.2.3 pillow-11.1.0 pyparsing-3.2.1 scikit-learn-1.6.1 scipy-1.15.1
threadpoolctl-3.5.0 tzdata-2025.1
Successfully installed et-xmlfile-2.0.0 openpyxl-3.1.5
Successfully installed clarabel-0.10.0 cvxpy-1.5.3 ecos-2.0.14 osqp-0.6.7.post3
qdldl-0.1.7.post5 scs-3.2.7.post2
Successfully installed absl-py-2.1.0 astunparse-1.6.3 flatbuffers-25.2.10
gast-0.6.0 google-pasta-0.2.0 grpcio-1.70.0 h5py-3.12.1 keras-3.8.0
libclang-18.1.1 markdown-3.7 markdown-it-py-3.0.0 mdurl-0.1.2 ml-dtypes-0.4.1
namex-0.0.8 numpy-2.0.2 opt-einsum-3.4.0 optree-0.14.0 protobuf-5.29.3
rich-13.9.4 tensorboard-2.18.0 tensorboard-data-server-0.7.2 tensorflow-2.18.0
termcolor-2.5.0 werkzeug-3.1.3 wrapt-1.17.2
Successfully installed xport-3.6.1
Requirement already satisfied: six>=1.5 in /opt/conda/lib/python3.12/site-
packages (from python-dateutil>=2.8.2->pandas==2.2.3) (1.17.0)
```

[14]:
```
import os
import logging
import warnings

# Set environment variables to suppress TensorFlow warnings
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'  # Suppress all but errors
os.environ['TF_ENABLE_ONEDNN_OPTS'] = '0'  # Disable oneDNN optimizations if␣
 ↪desired
os.environ['CUDA_VISIBLE_DEVICES'] = '-1'  # Disable GPU usage if CUDA is not␣
 ↪available

logging.getLogger('tensorflow').setLevel(logging.ERROR)

warnings.filterwarnings("ignore")

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import tensorflow as tf
```

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
```

1.4 Background

This tutorial uses the **Home Equity Line of Credit (HELOC)** dataset. The HELOC dataset consists of anonymized credit applications made to banks, where each application is assessed based on various financial attributes. The goal is to predict the risk level associated with each applicant, which ultimately helps loan officers decide whether to approve or deny the application.

1.4.1 Three perspectives: Data scientist, loan officer, and customer

To make AI insights accessible, we break down explanations from the point of view of three key stakeholders in the credit approval process:

1. Data scientist perspective

- For the data scientist, the focus is on evaluating and understanding the model before deployment. They need to ensure the model's transparency, fairness, and compliance with industry standards.
- Interpretable rule-based models help the data scientist verify the model's decision-making process, making sure it aligns with ethical and regulatory guidelines.

2. Loan officer perspective

- The loan officer uses the model's predictions to make final credit decisions. Their priority is to understand each prediction in a way that's relatable and actionable.
- Example-based explanations allow the loan officer to see comparable cases, making the model's output more interpretable and practical for real-world decision-making.

3. Customer perspective

- For the customer, clarity on why their application was approved or denied is essential. As non-technical users, they require straightforward explanations.
- Contrastive explanations offer insights into the features that contributed to their result and what could have led to a different outcome, providing them with a clear understanding of the decision.

In the following sections, we will explore these three perspectives, leveraging different XAI techniques to meet each stakeholder's needs.

1.5 Data loading and preprocessing

We use the pandas library to load the HELOC dataset. The data is stored in a CSV file named `heloc-dataset-v1.csv`.

We will now load the dataset and inspect the first few rows of the data using pandas `.head()` function.

[15]:
```
# Load HELOC dataset
csv_url = 'https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/
 ↪TpQl93NfuzpDVAPaFzs8qg/heloc-dataset-v1.csv'

dataset = pd.read_csv(csv_url)
dataset.head()
```

[15]:

	RiskPerformance	…	PercentTradesWBalance
0	Bad	…	69
1	Bad	…	0
2	Bad	…	86
3	Bad	…	91
4	Bad	…	80

[5 rows x 24 columns]

1.5.1 Dataset description

Each row in the dataset represents an individual credit application, with various features related to credit history, inquiries, and account balances.

The `info()` function provides a concise summary of the dataset, including the number of non-null entries, data types, and memory usage. A breakdown of the HELOC dataset follows:

Column	Type	Description
RiskPerformance	Object	Target variable indicating the risk performance of the applicant (Good/Bad).
ExternalRiskEstimate	Integer	Estimate of the external risk level associated with the applicant's credit history.
MSinceOldestTradeOpen	Integer	Months since the applicant's oldest trade (credit account) was opened.
MSinceMostRecentTradeOpen	Integer	Months since the applicant's most recent trade was opened.
AverageMInFile	Integer	Average months in file for all trades.
NumSatisfactoryTrades	Integer	Number of satisfactory (non-delinquent) trades in the applicant's history.
NumTrades60Ever2DerogPubRec	Integer	Number of trades ever 60+ days delinquent or derogatory public records.
NumTrades90Ever2DerogPubRec	Integer	Number of trades ever 90+ days delinquent or derogatory public records.
PercentTradesNeverDelq	Integer	Percentage of trades that have never been delinquent.
MSinceMostRecentDelq	Integer	Months since the applicant's most recent delinquency.
MaxDelq2PublicRecLast12M	Integer	Maximum delinquency in the last 12 months.
MaxDelqEver	Integer	Maximum delinquency ever reported.
NumTotalTrades	Integer	Total number of trades reported for the applicant.
NumTradesOpeninLast12M	Integer	Number of trades opened in the last 12 months.
PercentInstallTrades	Integer	Percentage of installment trades in the applicant's credit file.
MSinceMostRecentInqexcl7days	Integer	Months since the most recent inquiry (excluding last 7 days).
NumInqLast6M	Integer	Number of credit inquiries in the last 6 months.

Column	Type	Description
NumInqLast6Mexcl7days	Integer	Number of credit inquiries in the last 6 months (excluding last 7 days).
NetFractionRevolvingBurden	Integer	Fraction of revolving credit balance relative to total credit limit.
NetFractionInstallBurden	Integer	Fraction of installment credit balance relative to total credit limit.
NumRevolvingTradesWBalance	Integer	Number of revolving trades with a current balance.
NumInstallTradesWBalance	Integer	Number of installment trades with a current balance.
NumBank2NatlTradesWHighUtilization	Integer	Number of bank/national trades with high utilization rate.
PercentTradesWBalance	Integer	Percentage of trades that currently have a balance.

1.5.2 Checking for missing values (if any)

Missing values are common in time series datasets. We can use the `isna()` function to detect missing entries, and `sum()` to count how many missing values exist in each column.

[16]:
```
dataset.isna().sum()
```

[16]:
```
RiskPerformance                      0
ExternalRiskEstimate                 0
MSinceOldestTradeOpen                0
MSinceMostRecentTradeOpen            0
AverageMInFile                       0
NumSatisfactoryTrades                0
NumTrades60Ever2DerogPubRec          0
NumTrades90Ever2DerogPubRec          0
PercentTradesNeverDelq               0
MSinceMostRecentDelq                 0
MaxDelq2PublicRecLast12M             0
MaxDelqEver                          0
NumTotalTrades                       0
NumTradesOpeninLast12M               0
PercentInstallTrades                 0
MSinceMostRecentInqexcl7days         0
NumInqLast6M                         0
NumInqLast6Mexcl7days                0
NetFractionRevolvingBurden           0
NetFractionInstallBurden             0
NumRevolvingTradesWBalance           0
NumInstallTradesWBalance             0
NumBank2NatlTradesWHighUtilization   0
PercentTradesWBalance                0
dtype: int64
```

It looks like there are no missing values.

1.6 Data scientist perspective

The data scientist's goal in this project is to thoroughly understand and validate the model's behavior before it's deployed, ensuring it aligns with business needs and regulatory requirements. This is especially critical in industries such as banking, where model transparency and explainability are essential.

In this section, we will: 1. Encode and separate the target variable, **RiskPerformance**, to classify applicants as "Good" or "Bad" credit risks. 2. Preprocess features to make them interpretable and suitable for rule-based modeling. 3. Train interpretable models such as Boolean Rule Column Generation (BRCG) and Logistic Rule Regression (LRR) to generate clear rules for credit risk assessment.

The target column, **RiskPerformance**, is used as it directly indicates whether an applicant is a good or bad credit risk.

[17]:
```
# Separate the target variable
target = dataset.pop('RiskPerformance')
```

To prepare the target variable, for model training, we use `LabelEncoder` to convert the categorical labels (Good/Bad) into numerical values (0 and 1).

[18]:
```
from sklearn.preprocessing import LabelEncoder

# Instantiate the LabelEncoder
le = LabelEncoder()

# Encode the target variable
target_encoded = le.fit_transform(target)
```

1.6.1 Split data into training and testing sets

We split our dataset into training and testing sets using `train_test_split`. This allows us to train our model on one portion of the data (training set) and evaluate its performance on unseen data (testing set).

[19]:
```
from sklearn.model_selection import train_test_split

dfTrain, dfTest, yTrain, yTest = train_test_split(dataset, target_encoded,␣
 ↪random_state=0, stratify=target_encoded)
```

1.6.2 Data binarization and standardization

To make our data suitable for rule-based modeling, we use `FeatureBinarizer` from AIX360. This tool converts features into binary indicators and standardizes ordinal features (features with a natural order) so they can be easily compared.

What is interpretable rule-based modeling? Interpretable rule-based modeling is a machine learning approach that creates simple, easy-to-understand rules, such as "if-then" statements, to

make predictions. For example, a model might predict an applicant as a "Good" credit risk if they meet certain criteria, such as "IF ExternalRiskEstimate > 75 THEN Good Risk."

What is FeatureBinarizer?

- `FeatureBinarizer` is a tool that transforms continuous and categorical data into binary features, making it easier for rule-based models to interpret. It also standardizes ordinal features, providing both binary and standardized versions.
- Using binary features enables rule-based models, such as Boolean Rule Column Generation (BRCG), to work with simplified conditions, making their outputs easy to interpret. Additionally, standardized ordinal features allow models that rely on numerical relationships, such as Logistic Rule Regression (LRR), to perform better.

Let's apply `FeatureBinarizer` to prepare our training and testing datasets.

[20]:
```
# Import FeatureBinarizer for data preprocessing
from aix360.algorithms.rbm import FeatureBinarizer

# Instantiate FeatureBinarizer
fb = FeatureBinarizer(negations=True, returnOrd=True)
```

Apply the transformations to train and test data.

[21]:
```
# Apply FeatureBinarizer to the training data
# This returns a binarized version of the training data and a standardized␣
 ↪version of ordinal features
train_data_bin, train_data_std = fb.fit_transform(dfTrain)

# Apply the same transformation to the test data
# Ensures that the binarized and standardized features match those in the␣
 ↪training data
test_data_bin, test_data_std = fb.transform(dfTest)
```

[22]:
```
train_data_bin
```

[22]:

	ExternalRiskEstimate				…	PercentTradesWBalance		
	<=				…	>		
	56.0	61.0	65.0	68.0	…	71.0	79.0	86.0
100.0								
4322	0	0	0	0	…	0	0	0
0								
8920	1	1	1	1	…	1	1	1
0								
3288	0	0	0	0	…	0	0	0
0								
928	0	0	0	0	…	0	0	0
0								
7855	0	0	1	1	…	1	1	0
0								

```
…                    …  …  …  …  …                    …  …  …
…
1170                    1    1    1    1  …                    0    0    0
0
9420                    0    0    1    1  …                    1    1    1
0
8826                    0    0    1    1  …                    1    1    1
0
3277                    0    0    1    1  …                    1    0    0
0
8379                    0    0    0    0  …                    1    1    0
0

[7844 rows x 312 columns]
```

The preceding output of `FeatureBinarizer` (train_data_bin) provides a transformed dataset where each original feature has been split into binary indicators, making the data interpretable for rule-based models. Here's a breakdown of the transformation and how it works:

- **Binarization**: Continuous features such as `ExternalRiskEstimate` and `PercentTradesWBalance` are split into binary intervals (e.g., `ExternalRiskEstimate <= 56.0` and `ExternalRiskEstimate > 56.0`). Each interval is a separate binary feature, with `1` indicating that the value is within that range, and `0` indicating it is not.
- **Negations**: With `negations=True`, complementary features are added. For example, if a feature `ExternalRiskEstimate > 56.0` is created, a negated version, `ExternalRiskEstimate <= 56.0`, is also generated, allowing models to use both conditions in their rules.
- **Standardization**: The binarizer also returns standardized ordinal features (in `train_data_std` and `test_data_std`), scaling each to have zero mean and unit variance, which helps models that rely on numerical relationships.

This format supports interpretable rule-based modeling by simplifying conditions into binary indicators while also providing standardized features for models such as Logistic Rule Regression.

Try to explore the transformed dataset to get more information.

1.6.3 BooleanRuleCG (BRCG) model

The `BooleanRuleCG` (BRCG) model is a rule-based model, producing simple "if-then" rules that categorize applicants into groups, such as "Good" or "Bad" credit risk. This model uses a **conjunctive normal form (CNF)** or **disjunctive normal form (DNF)**, creating rules that are easy to understand and evaluate.

What is BooleanRuleCG?

- The `BooleanRuleCG` model in AIX360 is an algorithm that constructs rules in either CNF (AND-of-ORs) or DNF (OR-of-ANDs), both common logical forms. For example, CNF creates conditions where multiple OR statements are combined with AND, making it intuitive to read and interpret.

How does BooleanRuleCG work?

- **Rule Formulation**: BRCG generates rules in CNF or DNF, where each form has a specific use. CNF rules for "Good" classifications (Y = 1) represent combinations of conditions for acceptance, while DNF rules for "Bad" classifications (Y = 0) represent rejection conditions.
- **Column Generation Optimization**: BRCG uses an advanced optimization technique, column generation, to efficiently search the vast space of possible rule combinations. This technique ensures that the model finds the most predictive and concise set of rules.
- **Complexity Control**: Parameters such as `lambda0` and `lambda1` limit rule length by penalizing extra clauses and conditions, keeping the model simple and interpretable.

In this project, BRCG with CNF rules (enabled by `CNF=True`) provides slightly better accuracy for "Good" classifications. Default values for `lambda0` and `lambda1` are used to balance rule simplicity and accuracy.

If you want to learn more about the model, read the original paper on Arxiv.

[24]:
```
# Import BooleanRuleCG for rule-based modeling
from aix360.algorithms.rbm import BooleanRuleCG

# Instantiate BooleanRuleCG with a small complexity penalty and CNF structure
brcg_model = BooleanRuleCG(lambda0=1e-4, lambda1=1e-4, CNF=True)
```

Model training will take ~**4** minutes. So please be patient.

[25]:
```
# Train the BRCG model on the binarized training data
brcg_model.fit(train_data_bin, yTrain)
```

```
Learning CNF rule with complexity parameters lambda0=0.0001, lambda1=0.0001
Initial LP solved
Iteration: 1, Objective: 0.3008
Iteration: 2, Objective: 0.3008
Iteration: 3, Objective: 0.3008
Iteration: 4, Objective: 0.2984
Iteration: 5, Objective: 0.2984
Iteration: 6, Objective: 0.2984
Iteration: 7, Objective: 0.2984
```

Now, let's evaulate the model and see what rule the model came up with.

[26]:
```
from sklearn.metrics import accuracy_score

# Display the generated rules in CNF format
print('Predict Y=0 if ANY of the following rules are satisfied, otherwise Y=1:')
print(brcg_model.explain()['rules'], "\n")
print('Model is predicting Y = 0 (Bad credit risk) if the ExternalRiskEstimate␣
  ↪<= 74')
```

```
Predict Y=0 if ANY of the following rules are satisfied, otherwise Y=1:
['ExternalRiskEstimate <= 74.00']
```

```
Model is predicting Y = 0 (Bad credit risk) if the ExternalRiskEstimate <= 74
```

1.6.4 Interpreting BRCG model's output

The `BooleanRuleCG` (BRCG) model has generated a simple rule in conjunctive normal form (CNF) to classify applicants:

- **Prediction**: The model predicts **Y=0** (Bad credit risk) if the following rule is met:
  - `ExternalRiskEstimate <= 74.00`
- **Otherwise**: If this rule is not satisfied, the model predicts **Y=1** (Good credit risk).

Explanation

- This single rule indicates that an applicant is classified as a "Bad" credit risk if their **ExternalRiskEstimate** is 74 or below. This makes sense, as a lower ExternalRiskEstimate suggests a higher likelihood of credit risk.
- This model achieves simplicity by focusing on a single feature threshold, making the rule easy to interpret. Although this rule alone is straightforward, it may not capture all nuances in the data but provides a clear and interpretable decision boundary.

Note: This simple rule reflects a heuristic approach in AIX360's implementation of BRCG, using beam search. A more complex version of BRCG using integer programming (not available in AIX360) could produce slightly more accurate but complex rules.

1.6.5 LogisticRuleRegression (LRR) model

The `LogisticRuleRegression` (LRR) model combines rule-based conditions with logistic regression to provide interpretable, probabilistic predictions for classification tasks. This approach handles both binary and numerical features, providing a detailed view of prediction probabilities.

What is LogisticRuleRegression?

- `LogisticRuleRegression` (LRR) is an algorithm in AIX360 that builds interpretable classification rules with logistic regression, producing a probability score for each prediction. LRR combines "if-then" rules (created from binary features) with ordinal features to predict outcomes, such as whether an applicant is a "Good" or "Bad" credit risk.

How does LogisticRuleRegression work?

- **Combining rules with logistic regression**: LRR uses binary features (from BRCG) and standardized ordinal features to create a logistic regression model, providing clear rules and probability-based predictions.
- **Controlling complexity**: Parameters such as `lambda0` and `lambda1` keep the model simple and easy to interpret.
- **Building on BRCG**: LRR improves on BRCG by adding numerical insights from ordinal data, making it more effective for credit risk modeling.
- LRR balances simplicity and accuracy, making it suitable for datasets with both binary and continuous features such as credit scores or trade counts.

Comparison with BooleanRuleCG (BRCG)

- **Probabilistic predictions**: LRR offers probability scores instead of just binary outcomes, giving deeper insights into "Good" or "Bad" classifications.
- **Handles numerical data**: LRR works with both binary and continuous features by integrating standardized ordinal data, making it more flexible than BRCG, which only handles binary rules.

If you want to learn more about the model, read the original paper on Arxiv.

[27]:
```
# Import LogisticRuleRegression for interpretable rule-based modeling with␣
 ↪probabilistic output
from aix360.algorithms.rbm import LogisticRuleRegression

# Instantiate LogisticRuleRegression with appropriate complexity penalties and␣
 ↪ordinal features
lrr_model = LogisticRuleRegression(lambda0=0.005, lambda1=0.001, useOrd=True)
```

[28]:
```
# Train the LRR model on the binarized and standardized training data
lrr_model.fit(train_data_bin, yTrain, train_data_std)
```

Evaluate and interpret the LRR model

[29]:
```
print('Probability of Y=1 is predicted as logistic(z) = 1 / (1 + exp(-z))')
print('where z is a linear combination of the following rules/numerical␣
 ↪features:')
lrr_model.explain()
```

```
Probability of Y=1 is predicted as logistic(z) = 1 / (1 + exp(-z))
where z is a linear combination of the following rules/numerical features:
```

[29]:

	rule/numerical feature	coefficient
0	(intercept)	0.509205
1	MSinceMostRecentInqexcl7days <= -8.00	1.320125
2	MSinceMostRecentInqexcl7days <= 0.00	-0.702629
3	NumInqLast6M	-0.567451
4	PercentTradesNeverDelq <= 100.00 AND PercentTr…	0.5583
5	NumSatisfactoryTrades	0.557561
6	NetFractionRevolvingBurden	-0.449196
7	ExternalRiskEstimate <= 86.00 AND PercentTrade…	-0.429603
8	NumRevolvingTradesWBalance <= 5.00	0.423579
9	MaxDelq2PublicRecLast12M <= 5.00	-0.421211
10	ExternalRiskEstimate <= 86.00 AND NumSatisfact…	-0.416672
11	NumSatisfactoryTrades <= 13.00	-0.389988
12	ExternalRiskEstimate <= 74.00	-0.375404
13	ExternalRiskEstimate <= 65.00	-0.355879
14	PercentTradesNeverDelq <= 96.00	-0.31246
15	AverageMInFile <= 58.00	-0.286102
16	PercentInstallTrades <= 48.40	0.271917

17	NetFractionRevolvingBurden <= 47.00	0.268279
18	PercentInstallTrades <= 42.00	0.25665
19	ExternalRiskEstimate <= 81.00	-0.241072
20	ExternalRiskEstimate <= 71.00	-0.187279
21	NumSatisfactoryTrades <= 22.00	0.166199
22	AverageMInFile <= 74.00	-0.155649
23	ExternalRiskEstimate <= 78.00	-0.146636
24	AverageMInFile <= 66.00	-0.146161
25	NetFractionRevolvingBurden <= 36.00	0.145946
26	MSinceOldestTradeOpen	0.134481
27	ExternalRiskEstimate <= 68.00	-0.133075
28	MSinceOldestTradeOpen <= 132.00	-0.122618
29	AverageMInFile <= 81.00	-0.110301
30	PercentTradesWBalance	0.104299
31	PercentTradesNeverDelq <= 94.00	0.077382
32	PercentTradesWBalance <= 71.00	0.032209
33	MSinceOldestTradeOpen <= 157.00	-0.025492
34	NumSatisfactoryTrades <= 16.00	0.005992

1.6.6 Interpreting LRR model's output

The `LogisticRuleRegression` (LRR) model output shows a list of rules and numerical features, each paired with a coefficient. These rules and features are combined to make probabilistic predictions about credit risk in a way that is both interpretable.

Each row represents a condition or feature that affects the prediction:

- **Rule/Numerical Feature**: Specifies a rule (e.g., "ExternalRiskEstimate <= 86.00") or a continuous feature (e.g., "NumSatisfactoryTrades") that contributes to the model's decision.
- **Coefficient**: Indicates the influence of each rule or feature on the likelihood of a "Good" credit risk classification (Y=1). Positive coefficients increase the probability of a "Good" classification, while negative coefficients decrease it.

For example, a positive coefficient for `MSinceMostRecentInqexcl7days <= -8.00` indicates that if this condition is met, the probability of a "Good" classification increases. Conversely, a negative coefficient for `ExternalRiskEstimate <= 86.00` implies that meeting this condition lowers the likelihood of a "Good" classification.

Why this output matters For data scientists, this output is critical as it provides both interpretability and predictive accuracy. By analyzing coefficients, data scientists can understand key factors influencing the model's decisions, ensure fairness, and meet regulatory requirements. The transparency of LRR makes it particularly suited for high-stakes applications, such as credit risk assessment, where understanding model decisions is essential.

In summary, LRR combines simple rules with continuous features to create a balanced, interpretable, and effective model for predicting credit risk.

1.6.7 Model evaluation

To evaluate the effectiveness of each model, we compare their training and test accuracies.

[30]:
```
# BRCG Predictions
brcg_train_pred = brcg_model.predict(train_data_bin)
brcg_test_pred = brcg_model.predict(test_data_bin)

# LRR Predictions
lrr_train_pred = lrr_model.predict(train_data_bin, train_data_std)
lrr_test_pred = lrr_model.predict(test_data_bin, test_data_std)

# Accuracy comparison
print('BRCG Training accuracy:', accuracy_score(yTrain, brcg_train_pred))
print('BRCG Test accuracy:', accuracy_score(yTest, brcg_test_pred))
print("\b")
print('LRR Training accuracy:', accuracy_score(yTrain, lrr_train_pred))
print('LRR Test accuracy:', accuracy_score(yTest, lrr_test_pred))
```

```
BRCG Training accuracy: 0.6993880673125956
BRCG Test accuracy: 0.6982791586998088

LRR Training accuracy: 0.7329168791432943
LRR Test accuracy: 0.7315487571701721
```

[31]:
```
from sklearn.metrics import classification_report

# BRCG Classification report
print("BRCG Classification Report (Test Data):")
print(classification_report(yTest, brcg_test_pred))

# LRR Classification report
print("LRR Classification Report (Test Data):")
print(classification_report(yTest, lrr_test_pred))
```

```
BRCG Classification Report (Test Data):
              precision    recall  f1-score   support

           0       0.68      0.80      0.73      1365
           1       0.73      0.59      0.65      1250

    accuracy                           0.70      2615
   macro avg       0.70      0.69      0.69      2615
weighted avg       0.70      0.70      0.69      2615

LRR Classification Report (Test Data):
              precision    recall  f1-score   support

           0       0.73      0.78      0.75      1365
           1       0.74      0.68      0.71      1250

    accuracy                           0.73      2615
```

```
   macro avg       0.73      0.73      0.73      2615
weighted avg       0.73      0.73      0.73      2615
```

Classification report summary

Metric	**BRCG**	**LRR**
Accuracy	Lower overall accuracy compared to LRR.	Higher accuracy, indicating better generalization.
Precision	Acceptable precision but slightly lower for "Good" credit risk (Class 1).	Improved precision, especially for "Good" credit risk (Class 1).
Recall	Lower recall for "Good" credit risk (Class 1), missing some true positives.	Higher recall, reducing missed "Good" credit risks.
F1-Score	Lower f1-score, reflecting a less balanced trade-off between precision and recall.	Higher f1-score, showcasing a better balance between precision and recall.

1.6.8 Visualize LRR model as a generalized additive model (GAM)

The `LogisticRuleRegression` (LRR) model can be interpreted as a generalized additive model (GAM), which breaks down the prediction into contributions from individual features. By plotting **partial dependence plots (PDPs)** for each feature, we can understand how changes in a specific feature impact the predicted probability, holding all other features constant.

[32]:
```
dfx = lrr_model.explain()
dfx.head()
```

[32]:

	rule/numerical feature	coefficient
0	(intercept)	0.509205
1	MSinceMostRecentInqexcl7days <= -8.00	1.320125
2	MSinceMostRecentInqexcl7days <= 0.00	-0.702629
3	NumInqLast6M	-0.567451
4	PercentTradesNeverDelq <= 100.00 AND PercentTr…	0.5583

1.6.9 Partial dependence plots

To understand the effect of individual features on the LRR model's predictions, we will visualize **partial dependence plots (PDPs)** for selected features. PDPs allow us to see how changes in a single feature impact the predicted probability of an applicant being classified as a "Good" credit risk, while keeping other features constant.

The `plot_partial_dependence` function generates a PDP for a specified feature by calculating probabilities over a range of values. This range is derived from the original (non-standardized) values of the feature, allowing us to interpret results on a natural scale.

- **Input**:

  - The trained LRR model (`lrr`) to extract feature coefficients.
  - The standardized training data (`train_data_std`) to obtain the feature range.
  - The feature name (`feature_name`) to specify which feature's effect to plot.
  - The LRR model's explanation DataFrame (`dfx`) to extract the coefficient of the specified feature.

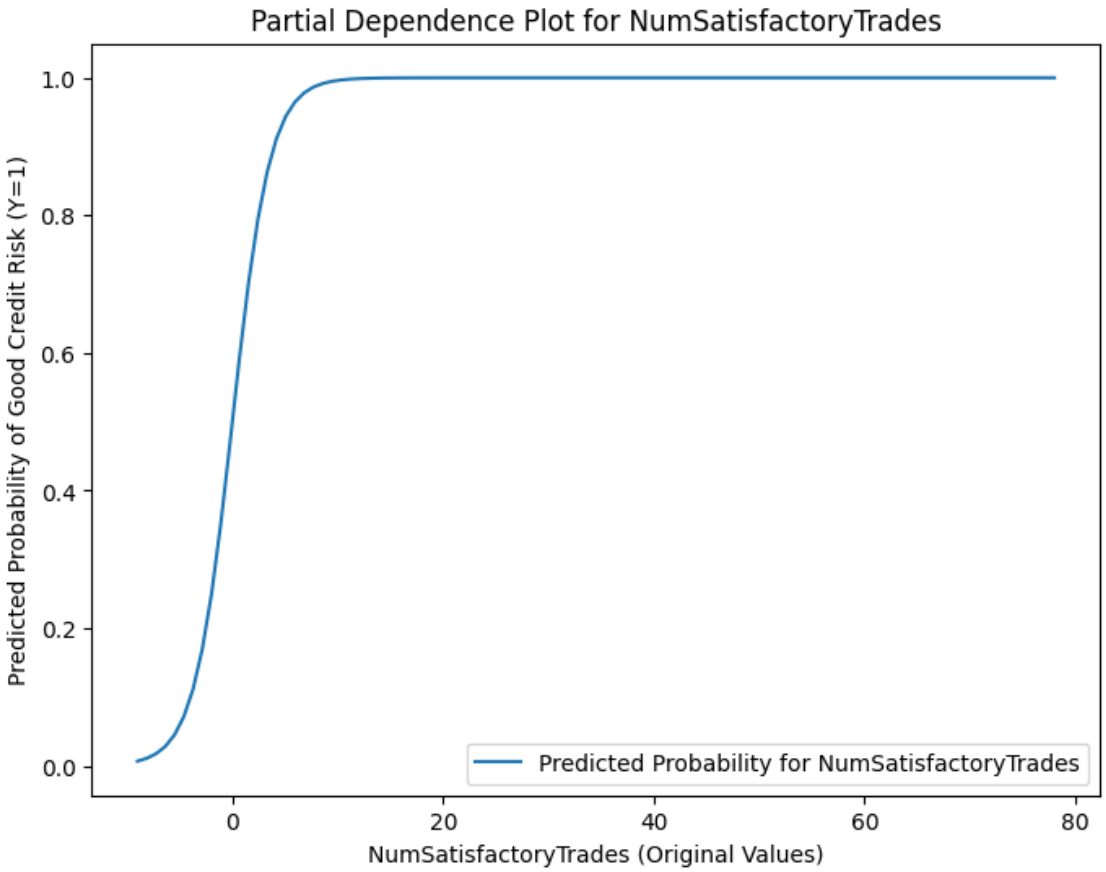
- **Output**: A plot of the predicted probability of a "Good" credit risk classification as a function of the specified feature's values.

This step is valuable for assessing the influence of specific features, as it highlights the relationship between each feature and the outcome, helping to interpret the LRR model as a generalized additive model (GAM).

[44]:
```
from scipy.special import expit  # Sigmoid function

def plot_partial_dependence(lrr, dfTrainStd, feature_name, dfx,␣
↪y_label='Predicted Probability of Good Credit Risk (Y=1)'):

    # Filter for the numerical feature specified
    feature_coef = dfx[dfx['rule/numerical feature'].str.contains(feature_name)]

    # Ensure there is a coefficient available for the feature
    if not feature_coef.empty:
        coef_feature = feature_coef['coefficient'].values[0]
    else:
        raise ValueError(f"No coefficient found for '{feature_name}' in the LRR␣
↪model explanation")

    # Create a range of values for the specified feature using original␣
↪(non-standardized) values
    feature_range = np.linspace(dfTrain[feature_name].min(),␣
↪dfTrain[feature_name].max(), 100)

    # Compute logits manually without an intercept, assuming no intercept is␣
↪available
    logits = coef_feature * feature_range
    probs = expit(logits)  # Apply sigmoid to logits to get probabilities

    # Plot the predicted probabilities as a function of the original feature␣
↪values
    plt.figure(figsize=(8, 6))
    plt.plot(feature_range, probs, label=f'Predicted Probability for␣
↪{feature_name}')
    plt.xlabel(f'{feature_name} (Original Values)')
    plt.ylabel(y_label)
    plt.title(f'Partial Dependence Plot for {feature_name}')
    plt.legend()
    plt.show()
```

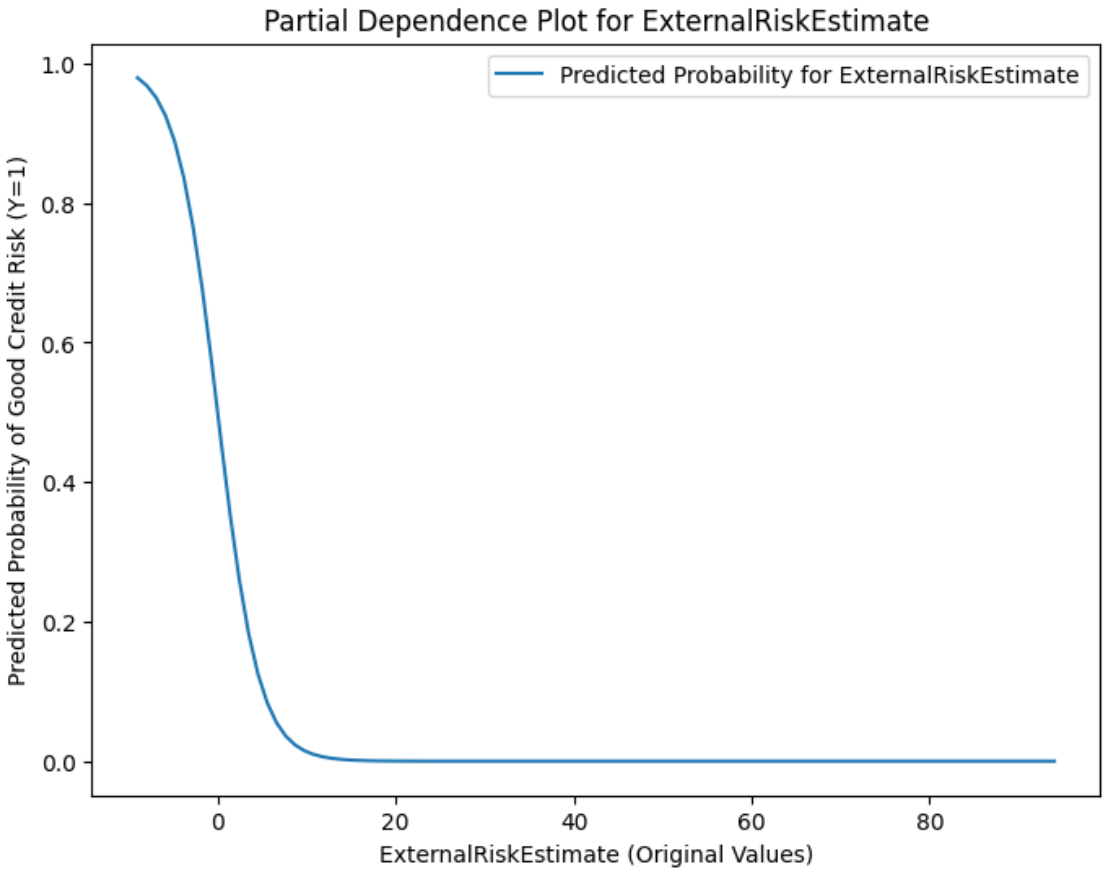
NumSatisfactoryTrades The PDP shows that **'NumSatisfactoryTrades'** is an important factor that increases the chance of being classified as a "Good" credit risk. The steep rise in the plot means that applicants with more satisfactory trades (trades completed without issues) are much more likely to be seen as low risk. However, after a certain number of satisfactory trades, this positive effect levels off, meaning that additional satisfactory trades do not significantly increase the probability further.

[35]:
```
plot_partial_dependence(lrr_model, train_data_std, 'NumSatisfactoryTrades', dfx)
```



ExternalRiskEstimate The PDP for **'ExternalRiskEstimate'** shows that this feature greatly affects the chance of being classified as a "Good" credit risk. As the 'ExternalRiskEstimate' value gets lower, the likelihood of a good credit risk classification quickly rises, almost reaching 100% for very low values. This means that lower 'ExternalRiskEstimate' values are strongly linked to applicants being seen as low-risk. After a certain point, this effect flattens out, meaning that further decreases in 'ExternalRiskEstimate' don't make much difference.

[36]:
```
plot_partial_dependence(lrr_model, train_data_std, 'ExternalRiskEstimate', dfx)
```



Try plotting the PDPs for other features.

1.7 Loan officer perspective

For loan officers, understanding how credit risk decisions are made is crucial for assessing applications. Beyond receiving a simple "Good" or "Bad" label, loan officers can benefit from seeing similar applicant profiles, helping them understand decisions in the context of comparable cases.

1.7.1 What this section covers:

1. **Review applicant profiles**: Analyze the distribution of "Good" applicants for an overview of risk patterns.
2. **Train a neural network (NN)**: Prepare the HELOC dataset and build a classification model for credit risk.
3. **Protodash for explanations**: Identify similar profiles from the training data to provide interpretable insights into decisions.

1.7.2 Why Protodash?

- Protodash finds **prototypical profiles**, highlighting training examples most similar to an applicant. This gives loan officers relatable comparisons that justify classifications such as

"Good" or "Bad."

- It avoids repetitive reasoning (e.g., focusing only on low satisfactory trades) and provides diverse explanations, covering a range of factors influencing decisions.
- Protodash works well even with non-standard data distributions, such as time series, making it more versatile than traditional nearest-neighbor methods.

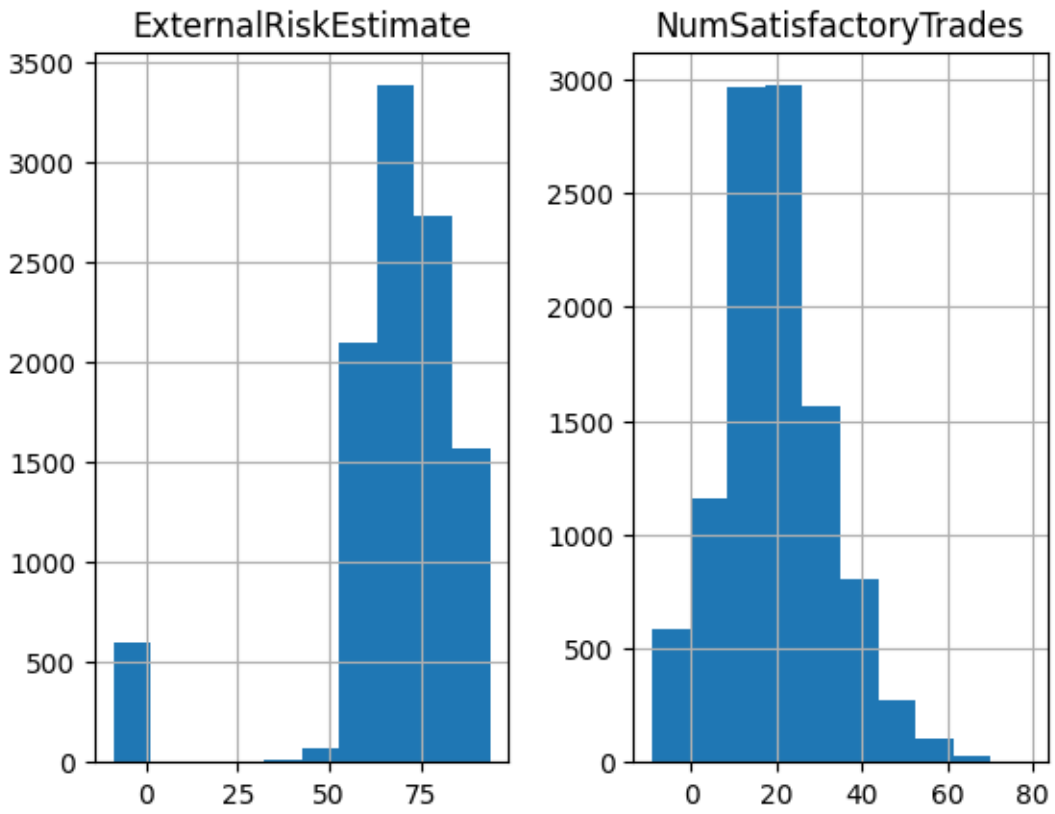
In this section, you'll explore how Protodash explains approvals for "Good" applicants by finding similar profiles and identifying key features that drive similarity, offering actionable insights for real-world decision-making.

[37]:
```
print("Size of HELOC dataset:", dataset.shape)
print("Number of \"Good\" applicants:", np.sum(target=='Good'))
print("Number of \"Bad\" applicants:", np.sum(target=='Bad'))
```

```
Size of HELOC dataset: (10459, 23)
Number of "Good" applicants: 5000
Number of "Bad" applicants: 5459
```

[38]:
```
# Plot (example) distributions for two features
print("Distribution of ExternalRiskEstimate and NumSatisfactoryTrades columns:")
hist = dataset.hist(column=['ExternalRiskEstimate', 'NumSatisfactoryTrades'],␣
↪bins=10)
```

```
Distribution of ExternalRiskEstimate and NumSatisfactoryTrades columns:
```



1.7.3 Process and normalize HELOC dataset for training

Split the data into training and testing sets:

[42]:
```
from sklearn.model_selection import train_test_split

x_train, x_test, y_train_b, y_test_b = train_test_split(dataset, target,␣
↪random_state=0, stratify=target, shuffle=True)
```

Initialize and apply `LabelEncoder` to encode target labels:

[43]:
```
from sklearn.preprocessing import LabelEncoder
from tensorflow.keras.utils import to_categorical

# Initialize and apply LabelEncoder
label_encoder = LabelEncoder()
y_train_b = label_encoder.fit_transform(y_train_b)
y_test_b = label_encoder.transform(y_test_b)

# Convert integer labels to one-hot encoding
y_train_b = to_categorical(y_train_b, num_classes=2)
y_test_b = to_categorical(y_test_b, num_classes=2)
```

To prepare the data for training, we normalize the feature values to a range of `[-0.5, 0.5]`. Normalizing the data ensures that features are on a similar scale, which can improve model performance. After training, we may need to rescale the data back to its original values for interpretation.

[46]:
```
# Combine training and testing data to calculate normalization bounds
Z = np.vstack((x_train, x_test))
Z_max = np.max(Z, axis=0)
Z_min = np.min(Z, axis=0)

# Normalize samples to range [-0.5, 0.5]
def normalize(data):
    normalized_data = (data - Z_min) / (Z_max - Z_min)
    return normalized_data - 0.5

# Rescale normalized data to original values
def rescale(normalized_data):
    return (normalized_data + 0.5) * (Z_max - Z_min) + Z_min

# Apply normalization
X_normalized = normalize(Z)
X_train_norm = X_normalized[:x_train.shape[0], :]
X_test_norm = X_normalized[x_train.shape[0]:, :]
```

1.7.4 Define and train a NN classifier

Define NN architecture `create_nn`, defines a simple feed-forward neural network model without a softmax activation in the final layer, allowing flexibility in the loss function used later.

The network consists of two dense layers:

- An input layer with 10 nodes
- A hidden layer with 2 output nodes

These layers matching the binary classification task.

[47]:
```
# nn with no softmax
def create_nn():
    model = Sequential()
    model.add(Dense(10, input_dim=23, kernel_initializer='normal',␣
 ↪activation='relu'))
    model.add(Dense(2, kernel_initializer='normal'))
    return model
```

Next, we define, compile, and train a neural network (NN) model to classify applicants as "Good" or "Bad" credit risk. Using **EarlyStopping** allows us to prevent overfitting by stopping training when the model's performance on the validation set stops improving.

[48]:
```
from tensorflow.keras.callbacks import EarlyStopping

# Set random seeds for repeatability
np.random.seed(1)
tf.random.set_seed(2)

class_names = ['Bad', 'Good']

# Custom loss function using softmax cross-entropy
def loss_function(correct, predicted):
    return tf.nn.softmax_cross_entropy_with_logits(labels=correct,␣
 ↪logits=predicted)

# EarlyStopping callback to stop training when validation loss does not improve
early_stopping = EarlyStopping(
    monitor='val_loss',  # Monitor validation loss
    patience=10,         # Number of epochs with no improvement before stopping
    restore_best_weights=True  # Restore weights from the best epoch
)

# compile and print model summary
nn_model = create_nn()
nn_model.compile(loss=loss_function, optimizer='adam', metrics=['accuracy'])
nn_model.summary()

# Train the neural network with early stopping on the normalized training data
```

```
nn_model.fit(X_train_norm, y_train_b, batch_size=128, epochs=50, verbose=1,␣
 ↪shuffle=False, validation_split=0.2, callbacks=[early_stopping])

# Evaluate model accuracy on training data
train_score = nn_model.evaluate(X_train_norm, y_train_b, verbose=0)
print('Train accuracy:', train_score[1])

# Evaluate model accuracy on test data
test_score = nn_model.evaluate(X_test_norm, y_test_b, verbose=0)
print('Test accuracy:', test_score[1])
```

```
2025-02-14 00:22:23.290251: E
external/local_xla/xla/stream_executor/cuda/cuda_driver.cc:152] failed call to
cuInit: INTERNAL: CUDA error: Failed call to cuInit: UNKNOWN ERROR (303)
```

Model: "sequential"

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 10)	240
dense_1 (Dense)	(None, 2)	22

Total params: 262 (1.02 KB)

Trainable params: 262 (1.02 KB)

Non-trainable params: 0 (0.00 B)

```
Epoch 1/50
50/50             1s 9ms/step -
accuracy: 0.5175 - loss: 0.6924 - val_accuracy: 0.5743 - val_loss: 0.6870
Epoch 2/50
50/50             0s 5ms/step -
accuracy: 0.5662 - loss: 0.6829 - val_accuracy: 0.6520 - val_loss: 0.6717
Epoch 3/50
50/50             0s 5ms/step -
accuracy: 0.6609 - loss: 0.6627 - val_accuracy: 0.6571 - val_loss: 0.6515
Epoch 4/50
50/50             0s 5ms/step -
accuracy: 0.6834 - loss: 0.6381 - val_accuracy: 0.6571 - val_loss: 0.6342
Epoch 5/50
50/50             0s 4ms/step -
accuracy: 0.6900 - loss: 0.6169 - val_accuracy: 0.6616 - val_loss: 0.6235
```

```
Epoch 6/50
50/50             0s 4ms/step -
accuracy: 0.6920 - loss: 0.6025 - val_accuracy: 0.6648 - val_loss: 0.6181
Epoch 7/50
50/50             0s 4ms/step -
accuracy: 0.6973 - loss: 0.5936 - val_accuracy: 0.6673 - val_loss: 0.6155
Epoch 8/50
50/50             0s 4ms/step -
accuracy: 0.6986 - loss: 0.5882 - val_accuracy: 0.6667 - val_loss: 0.6143
Epoch 9/50
50/50             0s 4ms/step -
accuracy: 0.6990 - loss: 0.5847 - val_accuracy: 0.6679 - val_loss: 0.6135
Epoch 10/50
50/50             0s 4ms/step -
accuracy: 0.6999 - loss: 0.5823 - val_accuracy: 0.6667 - val_loss: 0.6129
Epoch 11/50
50/50             0s 4ms/step -
accuracy: 0.7007 - loss: 0.5805 - val_accuracy: 0.6679 - val_loss: 0.6124
Epoch 12/50
50/50             0s 4ms/step -
accuracy: 0.7028 - loss: 0.5790 - val_accuracy: 0.6692 - val_loss: 0.6119
Epoch 13/50
50/50             0s 4ms/step -
accuracy: 0.7024 - loss: 0.5778 - val_accuracy: 0.6686 - val_loss: 0.6114
Epoch 14/50
50/50             0s 4ms/step -
accuracy: 0.7037 - loss: 0.5768 - val_accuracy: 0.6711 - val_loss: 0.6109
Epoch 15/50
50/50             0s 4ms/step -
accuracy: 0.7040 - loss: 0.5759 - val_accuracy: 0.6730 - val_loss: 0.6104
Epoch 16/50
50/50             0s 4ms/step -
accuracy: 0.7047 - loss: 0.5752 - val_accuracy: 0.6750 - val_loss: 0.6100
Epoch 17/50
50/50             0s 5ms/step -
accuracy: 0.7043 - loss: 0.5745 - val_accuracy: 0.6762 - val_loss: 0.6095
Epoch 18/50
50/50             0s 4ms/step -
accuracy: 0.7044 - loss: 0.5738 - val_accuracy: 0.6756 - val_loss: 0.6091
Epoch 19/50
50/50             0s 4ms/step -
accuracy: 0.7046 - loss: 0.5731 - val_accuracy: 0.6794 - val_loss: 0.6089
Epoch 20/50
50/50             0s 4ms/step -
accuracy: 0.7027 - loss: 0.5729 - val_accuracy: 0.6750 - val_loss: 0.6081
Epoch 21/50
50/50             0s 5ms/step -
accuracy: 0.7066 - loss: 0.5720 - val_accuracy: 0.6756 - val_loss: 0.6076
```

```
Epoch 22/50
50/50             0s 4ms/step -
accuracy: 0.7076 - loss: 0.5716 - val_accuracy: 0.6756 - val_loss: 0.6070
Epoch 23/50
50/50             0s 4ms/step -
accuracy: 0.7079 - loss: 0.5710 - val_accuracy: 0.6769 - val_loss: 0.6065
Epoch 24/50
50/50             0s 4ms/step -
accuracy: 0.7083 - loss: 0.5705 - val_accuracy: 0.6769 - val_loss: 0.6058
Epoch 25/50
50/50             0s 4ms/step -
accuracy: 0.7102 - loss: 0.5698 - val_accuracy: 0.6788 - val_loss: 0.6057
Epoch 26/50
50/50             0s 4ms/step -
accuracy: 0.7106 - loss: 0.5695 - val_accuracy: 0.6794 - val_loss: 0.6050
Epoch 27/50
50/50             0s 4ms/step -
accuracy: 0.7108 - loss: 0.5688 - val_accuracy: 0.6801 - val_loss: 0.6044
Epoch 28/50
50/50             0s 4ms/step -
accuracy: 0.7116 - loss: 0.5683 - val_accuracy: 0.6788 - val_loss: 0.6039
Epoch 29/50
50/50             0s 4ms/step -
accuracy: 0.7129 - loss: 0.5677 - val_accuracy: 0.6807 - val_loss: 0.6034
Epoch 30/50
50/50             0s 4ms/step -
accuracy: 0.7130 - loss: 0.5672 - val_accuracy: 0.6813 - val_loss: 0.6029
Epoch 31/50
50/50             0s 4ms/step -
accuracy: 0.7128 - loss: 0.5668 - val_accuracy: 0.6832 - val_loss: 0.6025
Epoch 32/50
50/50             0s 4ms/step -
accuracy: 0.7132 - loss: 0.5664 - val_accuracy: 0.6839 - val_loss: 0.6020
Epoch 33/50
50/50             0s 4ms/step -
accuracy: 0.7131 - loss: 0.5659 - val_accuracy: 0.6871 - val_loss: 0.6015
Epoch 34/50
50/50             0s 4ms/step -
accuracy: 0.7136 - loss: 0.5655 - val_accuracy: 0.6858 - val_loss: 0.6011
Epoch 35/50
50/50             0s 4ms/step -
accuracy: 0.7130 - loss: 0.5651 - val_accuracy: 0.6890 - val_loss: 0.6007
Epoch 36/50
50/50             0s 5ms/step -
accuracy: 0.7125 - loss: 0.5648 - val_accuracy: 0.6902 - val_loss: 0.6003
Epoch 37/50
50/50             0s 4ms/step -
accuracy: 0.7137 - loss: 0.5644 - val_accuracy: 0.6902 - val_loss: 0.6000
```

```
Epoch 38/50
50/50             0s 4ms/step -
accuracy: 0.7140 - loss: 0.5641 - val_accuracy: 0.6915 - val_loss: 0.5996
Epoch 39/50
50/50             0s 4ms/step -
accuracy: 0.7141 - loss: 0.5638 - val_accuracy: 0.6922 - val_loss: 0.5993
Epoch 40/50
50/50             0s 5ms/step -
accuracy: 0.7146 - loss: 0.5635 - val_accuracy: 0.6928 - val_loss: 0.5989
Epoch 41/50
50/50             0s 4ms/step -
accuracy: 0.7146 - loss: 0.5631 - val_accuracy: 0.6934 - val_loss: 0.5986
Epoch 42/50
50/50             0s 4ms/step -
accuracy: 0.7141 - loss: 0.5629 - val_accuracy: 0.6934 - val_loss: 0.5982
Epoch 43/50
50/50             0s 4ms/step -
accuracy: 0.7143 - loss: 0.5625 - val_accuracy: 0.6947 - val_loss: 0.5978
Epoch 44/50
50/50             0s 4ms/step -
accuracy: 0.7141 - loss: 0.5623 - val_accuracy: 0.6947 - val_loss: 0.5975
Epoch 45/50
50/50             0s 4ms/step -
accuracy: 0.7144 - loss: 0.5620 - val_accuracy: 0.6960 - val_loss: 0.5972
Epoch 46/50
50/50             0s 4ms/step -
accuracy: 0.7151 - loss: 0.5617 - val_accuracy: 0.6960 - val_loss: 0.5969
Epoch 47/50
50/50             0s 4ms/step -
accuracy: 0.7146 - loss: 0.5615 - val_accuracy: 0.6966 - val_loss: 0.5966
Epoch 48/50
50/50             0s 4ms/step -
accuracy: 0.7146 - loss: 0.5613 - val_accuracy: 0.6973 - val_loss: 0.5963
Epoch 49/50
50/50             0s 4ms/step -
accuracy: 0.7149 - loss: 0.5610 - val_accuracy: 0.6979 - val_loss: 0.5958
Epoch 50/50
50/50             0s 4ms/step -
accuracy: 0.7152 - loss: 0.5607 - val_accuracy: 0.6973 - val_loss: 0.5954
Train accuracy: 0.7144314050674438
Test accuracy: 0.7177820205688477
```

1.7.5 Generating prototypical explanations

Protodash provides loan officers with examples of applicants whose profiles closely resemble the applicant in question. By examining these **prototypical examples**, loan officers can better understand the rationale behind a particular credit risk classification. These prototypes are real profiles from the training data that the model used to learn patterns and classify applicants as "Good" or

"Bad." We will go through an example of an applicant predicted as "Good".

1.7.6 Obtain similar samples as explanations for a HELOC applicant predicted as Good

To provide the loan officer with meaningful examples, we need to identify applicants in the training set who were classified as "Good" by the neural network model. By isolating these instances, we can use them as a reference for finding similar profiles for any given applicant.

[49]:
```
# Get class predictions as either 0 ("Bad") or 1 ("Good")
train_pred = np.argmax(nn_model.predict(X_train_norm), axis=1) # Get class␣
 ↪predictions as 0 or 1
train_pred = train_pred.reshape((train_pred.shape[0], 1)) # Reshape to match␣
 ↪original format

# Append the predictions to the normalized training data
# This creates a combined dataset with the normalized features and their␣
 ↪predicted class labels
norm_train_pred = np.hstack((X_train_norm, train_pred))
# Filter the combined dataset to keep only the instances predicted as "Good"␣
 ↪(class 1)
good_norm_train = norm_train_pred[norm_train_pred[:, -1] == 1, :]

# Perfrom the same thing for unnormalized data
unnorm_train_pred = np.hstack((x_train, train_pred))
good_unnorm_train = unnorm_train_pred[unnorm_train_pred[:, -1] == 1, :]
```

```
246/246             0s 1ms/step
```

Prediction and interpretation for a specific test sample using neural network Here, we select a specific test sample, make a prediction using the neural network model, and interpret the result by:

1. **Applying Softmax** to convert logits to probability scores, helping us understand the model's confidence in its classification.
2. **Retrieving original feature values** for the chosen sample, providing interpretable results that are easy for a loan officer to understand.

By displaying both the predicted class and original feature values, we make the model's decision more transparent and interpretable.

[50]:
```
from tensorflow.keras.activations import softmax

# Index of the chosen test sample for prediction
index = 8

# Reshape the chosen sample for model prediction
idx_norm = X_test_norm[index].reshape((1,) + X_test_norm[index].shape)
```

```
# Make prediction using the model and apply softmax to get probabilities
logits = nn_model.predict(idx_norm)
probs = softmax(logits).numpy()
predicted_class = np.argmax(probs)

# Print the chosen sample and model predictions
print("Chosen Sample:", index)
print("Prediction made by the model:", class_names[predicted_class])
print("Prediction probabilities:", probs)
print("")

# Attach the predicted class to the chosen sample's normalized values for␣
 ↪further analysis
idx_pred = np.hstack((idx_norm, [[predicted_class]]))

# Retrieve and display the original (unnormalized) feature values for the␣
 ↪chosen sample
idx_orig = X_test_norm[index].reshape((1,) + X_test_norm[index].shape)
df_sample = pd.DataFrame.from_records(idx_orig.astype('double'))  # Create␣
 ↪DataFrame with original feature values
df_sample[23] = class_names[predicted_class]  # Add prediction class to the␣
 ↪DataFrame
df_sample.columns = list(dataset.columns) + ["Prediction"] # Set columns with␣
 ↪feature names and prediction
df_sample.transpose()
```

```
1/1             0s 39ms/step
Chosen Sample: 8
Prediction made by the model: Good
Prediction probabilities: [[0.19043615 0.8095638 ]]
```

[50]:

	0
ExternalRiskEstimate	0.364078
MSinceOldestTradeOpen	0.060345
MSinceMostRecentTradeOpen	-0.369898
AverageMInFile	-0.07398
NumSatisfactoryTrades	-0.215909
NumTrades60Ever2DerogPubRec	-0.142857
NumTrades90Ever2DerogPubRec	-0.142857
PercentTradesNeverDelq	0.5
MSinceMostRecentDelq	-0.478261
MaxDelq2PublicRecLast12M	0.333333
MaxDelqEver	0.5
NumTotalTrades	-0.269912
NumTradesOpeninLast12M	-0.178571
PercentInstallTrades	-0.197248

```
MSinceMostRecentInqexcl7days       -0.469697
NumInqLast6M                           -0.38
NumInqLast6Mexcl7days                  -0.38
NetFractionRevolvingBurden         -0.462656
NetFractionInstallBurden           -0.497917
NumRevolvingTradesWBalance         -0.256098
NumInstallTradesWBalance             -0.1875
NumBank2NatlTradesWHighUtilization -0.166667
PercentTradesWBalance               0.041284
Prediction                              Good
```

The preceding output shows the prediction details for **Sample 8**, a specific test case passed through the trained neural network model to determine its credit risk classification.

- **Prediction**: The model classified the applicant as **"Good"**, indicating they are likely to qualify for the loan.
- **Prediction probabilities**: The model shows a high confidence level with approximately **80% probability** for "Good" and **20%** for bad.
- **Feature values**: Each feature is listed with its **normalized value** for this applicant, highlighting key details of the applicant's profile.
- **Prediction outcome**: The final column confirms the prediction of "Good," providing clear reasoning behind the decision.

This concise breakdown gives a clear view of the applicant's feature values, the model's confidence, and the final decision, helping stakeholders understand the reasoning behind the classification.

1.7.7 ProtodashExplainer

The `ProtodashExplainer` is an algorithm in AIX360 designed to provide explanations by selecting **prototypical examples** or similar user profiles from a dataset. This approach helps in understanding a specific data point or group of data points by comparing them with similar instances from the training data, making it particularly useful for applications such as loan approvals.

What is ProtodashExplainer?

- `ProtodashExplainer` is an algorithm in AIX360 that selects representative or prototypical examples from a dataset to explain a given data point. It identifies a set of instances that are most similar to the target instance based on feature values, helping users see patterns and characteristics in the data that support the model's decision.

How does ProtodashExplainer work?

- **Prototype selection**: `ProtodashExplainer` selects a specified number of prototypes from the training set by minimizing the **maximum mean discrepancy (MMD)** between the target instance and the prototypes. This ensures the prototypes are as representative as possible of the target.
- **Greedy optimization with weights**: The algorithm uses a greedy approach to find the best matching prototypes and assigns a weight to each prototype, reflecting its similarity to the target instance. Higher weights indicate closer similarity.

If you want to learn more about the model, read the original paper on Arxiv.

[51]:
```
from aix360.algorithms.protodash.PDASH import ProtodashExplainer
import xport

# Remove the last column (predicted class) from z_train_good
good_applicant_features = good_norm_train[:, :-1]  # Exclude the last column to␣
↪keep only features

explainer = ProtodashExplainer()
(W, S, setValues) = explainer.explain(idx_norm, good_applicant_features, m=5) ␣
↪# Use the corrected feature set
```

[52]:
```
print(f"W: {W}\n")
print(f"S: {S}\n")
print(f"setValues: {setValues}")
```

```
W: [7.82083295e-01 3.94430791e-02 2.16829753e-22 7.36073415e-02
 1.52218773e-01]

S: [1848 1569  262 3080  702]

setValues: [1.21483828 1.21837639 1.22078248 1.22118194 1.22259731]
```

The output of the `ProtodashExplainer` provides three key components that help us understand the similarity between a target applicant and selected prototypes:

- **Weights (W)**: Higher weights indicate greater similarity between the target and the selected prototype.
- **Indices (S)**: Each index maps to a training data profile, allowing us to examine feature values that drive similarity.
- **MMD values (setValues)**: These scores measure the overall difference between the target and prototypes; lower values are a closer match to the target.

This output helps identify the most representative profiles from the training data and assess their relative similarity to the target applicant, providing interpretable context for the model's decision.

1.7.8 Displaying similar applicant profiles

This code creates a DataFrame (`prototypes_df`) to show the most similar applicant profiles (prototypes) to a target applicant, based on selected features.

1. **Extract feature values**: Shows feature values of selected prototypes from the training data.
2. **Add predicted class**: Appends the predicted class for each prototype based on the original training data.
3. **Add similarity weights**: Adds normalized weights to indicate how similar each prototype is to the target applicant (higher weight = greater similarity).

The table provides a clear comparison of feature values, predictions, and similarity scores, helping to interpret the model's decision.

[53]:
```
# Create DataFrame for selected prototypes (feature values only, excluding the␣
 ↪prediction column)
prototypes_df = pd.DataFrame.from_records(good_unnorm_train[S, 0:-1].
 ↪astype('double'))

# Append the predicted class names ("Good" or "Bad") to the 24th column
predicted_labels=[]

for i in range(S.shape[0]):
    predicted_labels.append(class_names[int(good_norm_train[S[i], -1])]) #␣
 ↪Append class names
prototypes_df[23] = predicted_labels # Add prediction column

# Add the normalized weights as a separate column to indicate similarity
prototypes_df["Weight"] = np.around(W, 5) / np.sum(np.around(W, 5))  #␣
 ↪Calculate normalized weights

# Define column names including "Prediction" and "Weight" for clarity
column_names = list(dataset.columns) + ["Prediction", "Weight"]

# Set the final column names for the DataFrame
prototypes_df.columns = column_names

prototypes_df.transpose()
```

[53]:

	0	1	…	3	4
ExternalRiskEstimate	87.0	-9.0	…	57.0	67.0
MSinceOldestTradeOpen	430.0	383.0	…	168.0	586.0
MSinceMostRecentTradeOpen	35.0	383.0	…	73.0	7.0
AverageMInFile	154.0	383.0	…	107.0	173.0
NumSatisfactoryTrades	14.0	1.0	…	8.0	45.0
NumTrades60Ever2DerogPubRec	0.0	1.0	…	2.0	3.0
NumTrades90Ever2DerogPubRec	0.0	1.0	…	2.0	2.0
PercentTradesNeverDelq	100.0	100.0	…	89.0	89.0
MSinceMostRecentDelq	-7.0	-7.0	…	8.0	6.0
MaxDelq2PublicRecLast12M	7.0	6.0	…	0.0	4.0
MaxDelqEver	8.0	8.0	…	6.0	4.0
NumTotalTrades	14.0	1.0	…	9.0	47.0
NumTradesOpeninLast12M	0.0	0.0	…	0.0	2.0
PercentInstallTrades	21.0	100.0	…	44.0	17.0
MSinceMostRecentInqexcl7days	-8.0	-7.0	…	-7.0	0.0
NumInqLast6M	0.0	1.0	…	0.0	3.0
NumInqLast6Mexcl7days	0.0	1.0	…	0.0	3.0
NetFractionRevolvingBurden	10.0	-8.0	…	-8.0	37.0
NetFractionInstallBurden	-8.0	-8.0	…	-8.0	63.0
NumRevolvingTradesWBalance	2.0	-8.0	…	-8.0	6.0
NumInstallTradesWBalance	1.0	-8.0	…	-8.0	4.0

```
NumBank2NatlTradesWHighUtilization       0.0      -8.0  …      -8.0       2.0
PercentTradesWBalance                   60.0      -8.0  …      -8.0      53.0
Prediction                              Good      Good  …      Good      Good
Weight                              0.746723  0.037657  …  0.070282  0.145338

[25 rows x 5 columns]
```

The table displays five prototypical applicant profiles selected as the most similar to a target applicant. Each row represents a feature, and each column (0–4) corresponds to one of the selected prototypes. Here's a breakdown of the key elements:

- **Feature values**: Each cell contains the original (unnormalized) value for a specific feature, allowing us to compare each prototype's profile directly.
- **Prediction**: All prototypes have been classified as "Good" by the model, indicating that these applicants were approved.
- **Weight**: The last row shows the similarity weight of each prototype relative to the target applicant. A higher weight indicates a closer match. Weights are normalized to sum to 1.

This table helps in interpreting the model's decision by comparing the target applicant with similar profiles, offering insights into the characteristics of other applicants who were also classified as "Good".

As a part of learning, obtain similar samples as explanations for a HELOC applicant predicted as "Bad" to get more understanding.

1.8 Customer perspective: Contrastive explanations for HELOC use case

For homeowners applying for a HELOC, understanding the reasons behind approval or rejection is crucial. Contrastive explanations provide insights by focusing on:

1. **Pertinent negatives (PNs)**: Identify minimal changes that, if made to the applicant's profile, would have changed the model's decision (e.g., from "reject" to "accept").
2. **Pertinent positives (PPs)**: Show the minimal set of features and values required to maintain the original decision. For example, if an application is approved, the explanation may indicate that even if certain values were lower, the loan would still be accepted.

In this section, you will:

1. **Select a target applicant**
2. **Compute pertinent negatives**
3. **Compute pertinent positives**

These explanations enable applicants with actionable insights to improve their profile and help financial institutions understand key trends influencing decisions. For more details, check out the original paper on Arxiv.

We select a specific applicant from the test data and use the model to predict their HELOC application outcome. We also retrieve the probability scores for each class ("Good" or "Bad"), providing insights into the model's confidence in its prediction.

[54]:
```
# Some interesting user samples to try: 2344 449 1168 1272
index = 1272
```

```
# Reshape the chosen sample for model prediction
target_sample = X_test_norm[index].reshape((1,) + X_test_norm[index].shape)

# Make prediction using the model and apply softmax to get probabilities
logits = nn_model.predict(target_sample)
probs = softmax(logits).numpy()
predicted_class = np.argmax(probs)

print("Computing PN for Sample:", index)
print("Prediction made by the model:", class_names[predicted_class])
print("Prediction probabilities:", probs)
print("")
```

```
1/1             0s 36ms/step
Computing PN for Sample: 1272
Prediction made by the model: Bad
Prediction probabilities: [[0.8866199  0.11338005]]
```

1.8.1 Compute pertinent negative (PN)

The function identifies a **pertinent negative (PN)**, showing the smallest change needed to flip the model's prediction (e.g., from "reject" to "approve"). This helps applicants understand the minimal adjustments required to alter their outcome.

Function flow:

1. **Prepare data**: Converts applicant data into a TensorFlow variable for optimization.
2. **Set up optimization**: Uses the Adam optimizer to iteratively adjust the data.
3. **Adjust features**:
   - Calculates the model's score for the target class (e.g., "approve").
   - Defines a loss function to maximize the target class score.
   - Updates the data using gradients, making small, targeted changes.
4. **Early stopping**: Stops when adjustments are minimal, as defined by the tolerance.

The result is `perturbed_data`, the adjusted profile that flips the prediction. This provides actionable insights into how an applicant can improve their profile for a desired outcome.

[55]:
```
def find_pertinent_negative(model, applicant_data, target_class=1,␣
↪max_steps=1000, learning_rate=0.01, tolerance=1e-5):
    # Convert applicant data to a TensorFlow Variable
    applicant_data = tf.Variable(applicant_data.reshape(1, -1), dtype=tf.
↪float32)

    # Define an optimizer
    optimizer = tf.optimizers.Adam(learning_rate=learning_rate)

    # Optimization loop
```

```
    for step in range(max_steps):
        with tf.GradientTape() as tape:
            # Get the model's output for the current data
            logits = model(applicant_data)
            target_score = logits[0, target_class]  # Target class score for␣
↪class flipping

            # Define a loss function encouraging a high score for the target␣
↪class
            loss = -target_score  # We minimize negative target score to␣
↪maximize it

        # Compute gradients
        grads = tape.gradient(loss, applicant_data)

        # Apply the gradient updates
        optimizer.apply_gradients([(grads, applicant_data)])

        # Check if change is small enough
        if tf.reduce_max(tf.abs(grads)) < tolerance:
            break

    # Get the final perturbed data
    perturbed_data = applicant_data.numpy()
    return perturbed_data
```

The following code generates and compares a **pertinent negative (PN)** for a selected applicant (1168).

Flow

1. **Select target applicant**: Choose a sample applicant and generate their PN using the `find_pertinent_negative` function.
2. **Compare predictions**: Display the original and PN predictions, showing how minimal changes can flip the outcome.
3. **Calculate Differences**: Compute and display the feature-wise differences between the original and PN profiles.
4. **Create Display Table**: Present the original, PN, and difference values in a DataFrame with conditional formatting to highlight significant changes.

[56]:
```
# Sample index
index = 1168
applicant = x_test.iloc[index].values.reshape(1, -1)  # Original applicant data␣
↪(normalized)
pertinent_negative = find_pertinent_negative(nn_model, applicant[0])[0]  #␣
↪Generate Pertinent Negative

# Display predictions for original and perturbed instances
```

```
print("Sample:", index)
og_pred = nn_model.predict(applicant)
pn_pred = nn_model.predict(pertinent_negative.reshape(1, -1))
print("Prediction for original applicant (X):", og_pred, class_names[np.
 ↪argmax(og_pred)])
print("Prediction for pertinent negative (X_PN):", pn_pred, class_names[np.
 ↪argmax(pn_pred)])

# Calculate delta (difference) between original and pertinent negative without␣
 ↪rescaling
delta = pertinent_negative - applicant
delta = np.around(delta.astype(np.double), 2)
delta[np.absolute(delta) < 1e-4] = 0  # Remove very small changes for clarity

# Prepare DataFrame for display
comparison_data = np.vstack((applicant, pertinent_negative, delta))  # Stack␣
 ↪original, PN, and delta

# Calculate the total difference as the sum of absolute changes for each feature
total_difference = np.sum(np.abs(delta))
```

```
Sample: 1168
1/1             0s 58ms/step
1/1             0s 33ms/step
Prediction for original applicant (X): [[-299.61374  279.79987]] Good
Prediction for pertinent negative (X_PN): [[-378.12744  352.579  ]] Good
```

[57]:
```
# Add a more meaningful label for the Outcome row in the Difference column
out_labels = [
    class_names[np.argmax(og_pred)],
    class_names[np.argmax(pn_pred)],
    f"Total Difference: {total_difference:.2f}"
]

# Create DataFrame with features, predictions, and changes
comparison_df = pd.DataFrame(comparison_data, columns=dataset.columns)  #␣
 ↪Assuming column names come from dataset
comparison_df['Outcome'] = out_labels  # Add an outcome column
comparison_df.rename(index={0: 'Original (X)', 1: 'Pertinent Negative (X_PN)',␣
 ↪2: 'Difference (X_PN - X)'}, inplace=True)

# Transpose and display DataFrame with conditional formatting
comparison_df_t = comparison_df.transpose()
```

[58]:
```
# Highlight function to color positive differences
def highlight_changes(s, col, ncols):
    if isinstance(s[col], (int, float)):
```

```
        return ['background-color: green' if s[col] > 0 else 'background-color:␣
↪red'] * ncols
    return ['background-color: black'] * ncols

comparison_df_t.style.apply(highlight_changes, col='Difference (X_PN - X)',␣
↪ncols=3, axis=1)
```

[58]: <pandas.io.formats.style.Styler at 0x7fb781cdd220>

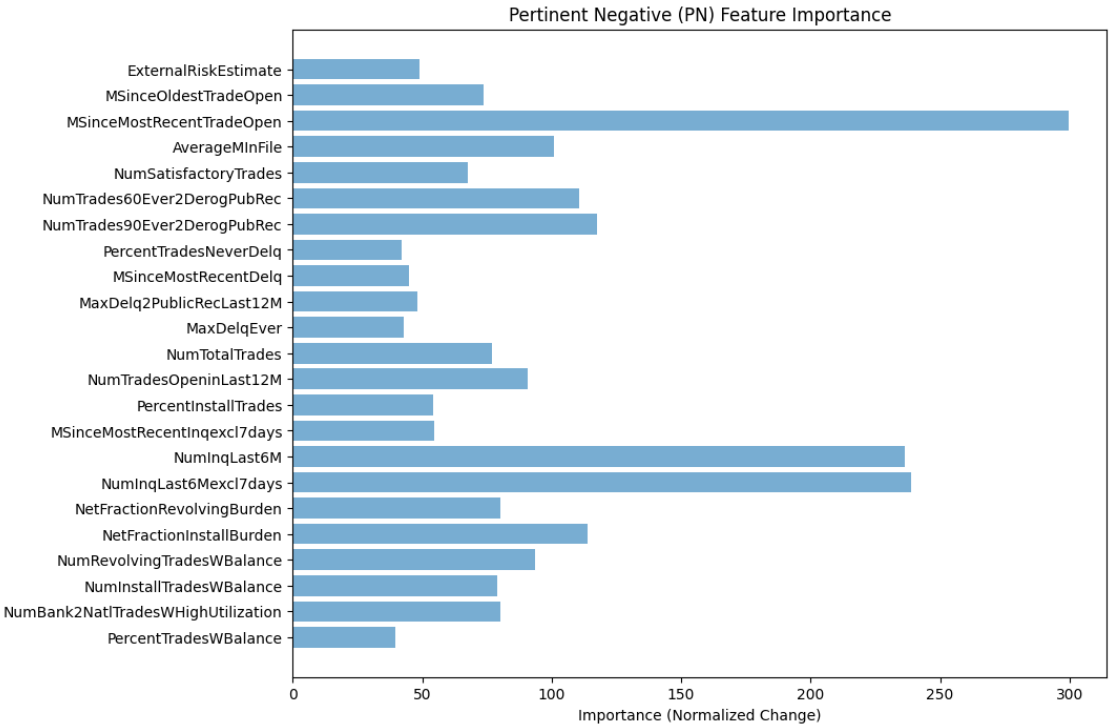
The table compares an applicant's **Original Profile (X)** and their **Pertinent Negative (X_PN)**—a minimally adjusted profile that maintains the same "Good" outcome. Key insights from the table are as follows:

1. **Minimal adjustments**: Small feature changes in **X_PN** ensure the positive outcome is preserved.
   - Example: **ExternalRiskEstimate** increases by +10 (89 → 99).
   - Example: **MSinceOldestTradeOpen** decreases by -10 (299 → 289).
2. **Positive and Negative Changes**:
   - **Increases**: Features such as **NumSatisfactoryTrades** and **PercentTradesNeverDelq** need slight improvements.
   - **Decreases**: Features such as **PercentInstallTrades** and **NetFractionRevolvingBurden** are reduced.
3. **Outcome consistency**: Both profiles lead to the "Good" classification, confirming that small adjustments can help maintain approval.
4. **Total adjustment**: The overall change across features totals ~**230**, representing the cumulative effort to achieve the PN.

Feature Importance Plot Below generates a **feature importance plot** to show which features have the greatest impact in differentiating the applicant's original profile from their Pertinent Negative (PN). The features with the highest bars represent those that require the most significant change to achieve the PN, highlighting their importance in maintaining the "Good" classification.

[59]:
```
# Compute the scaled absolute difference between original and PN (Pertinent␣
↪Negative)
feature_imp = np.abs(applicant - pertinent_negative) / np.std(X_train_norm.
↪astype('double'), axis=0)

# Define feature labels for the y-axis
feature_labels = dataset.columns  # Assuming column names from the dataset

# Reverse the order for plotting so the highest feature importance is at the top
performance = feature_imp.flatten()[::-1]
feature_labels = feature_labels[::-1]

# Plot the feature importance
```

```
plt.figure(figsize=(10, 8))
plt.barh(range(len(feature_labels)), performance, align='center', alpha=0.6)
plt.yticks(range(len(feature_labels)), feature_labels)
plt.xlabel('Importance (Normalized Change)')
plt.title('Pertinent Negative (PN) Feature Importance')
plt.show()
```



1.8.2 Compute pertinent positive (PP)

The function identifies a **pertinent positive (PP)**, representing the smallest changes required to keep the model's favorable prediction intact (e.g., ensuring "Good" classification). This helps applicants understand which features are critical to maintaining approval.

Function flow:

1. **Prepare data**: Converts the applicant's profile into a TensorFlow variable for optimization.
2. **Set up optimization**: Uses the Adam optimizer to make iterative adjustments.
3. **Preserve Key Features**:
   - Calculates the model's score for the current favorable class (e.g., "Good").
   - Defines a loss function to maintain this score while penalizing unnecessary feature changes.
   - Updates the data using gradients, ensuring minimal modifications.
4. **Early Stopping**: Stops when adjustments are minor, based on the tolerance threshold.

The result is `perturbed_data`, a minimally adjusted profile that retains the favorable prediction, providing insights into the most influential features needed to secure approval.

[60]:
```
# Function to find a pertinent positive
def find_pertinent_positive(model, applicant_data, original_class=1,␣
↪max_steps=1000, learning_rate=0.01, tolerance=1e-5):
    # Convert applicant data to a TensorFlow Variable
    applicant_data = tf.Variable(applicant_data.reshape(1, -1), dtype=tf.
↪float32)

    # Define an optimizer
    optimizer = tf.optimizers.Adam(learning_rate=learning_rate)

    # Optimization loop
    for step in range(max_steps):
        with tf.GradientTape() as tape:
            # Get the model's output for the current data
            logits = model(applicant_data)
            original_score = logits[0, original_class]  # Original class score␣
↪to retain

            # Define a loss function to maximize the original class score while␣
↪minimizing feature values
            loss = -original_score + tf.reduce_sum(tf.square(applicant_data)) *␣
↪0.01

        # Compute gradients
        grads = tape.gradient(loss, applicant_data)

        # Apply the gradient updates
        optimizer.apply_gradients([(grads, applicant_data)])

        # Check if change is small enough
        if tf.reduce_max(tf.abs(grads)) < tolerance:
            break

    # Get the final perturbed data
    perturbed_data = applicant_data.numpy()
    return perturbed_data
```

The following code generates and compares a pertinent positive (PP) for a selected applicant.

1.8.3 Flow

1. **Select target applicant**: Choose a sample applicant and generate their PP using the `find_pertinent_positive` function.
2. **Compare predictions**: Display the original and PP predictions, illustrating how small changes can help retain the positive outcome.

3. **Calculate differences**: Compute and display the feature-wise differences between the original and PP profiles to understand key factors.
4. **Create display table**: Present the original, PP, and difference values in a DataFrame with conditional formatting to highlight impactful changes.

[61]:
```
# Sample index and initial setup
index = 1168
applicant = x_test.iloc[index].values.reshape(1, -1)  # Original applicant data␣
↪(normalized)
pertinent_positive = find_pertinent_positive(nn_model, applicant[0])[0]  #␣
↪Generate Pertinent Positive

# Display predictions for original and perturbed instances
print("Sample:", index)
og_pred = nn_model.predict(applicant)
pp_pred = nn_model.predict(pertinent_positive.reshape(1, -1))
print("Prediction for original applicant (X):", og_pred, class_names[np.
↪argmax(og_pred)])
print("Prediction for pertinent positive (X_PP):", pp_pred, class_names[np.
↪argmax(pp_pred)])

# Calculate delta (difference) between original and pertinent positive without␣
↪rescaling
delta = pertinent_positive - applicant
delta = np.around(delta.astype(np.double), 2)
delta[np.absolute(delta) < 1e-4] = 0  # Remove very small changes for clarity

# Prepare DataFrame for display
comparison_data = np.vstack((applicant, pertinent_positive, delta))  # Stack␣
↪original, PP, and delta

# Calculate the total difference as the sum of absolute changes for each feature
total_difference = np.sum(np.abs(delta))
```

```
Sample: 1168
1/1             0s 32ms/step
1/1             0s 31ms/step
Prediction for original applicant (X): [[-299.61374  279.79987]] Good
Prediction for pertinent positive (X_PP): [[-310.68573  290.4525 ]] Good
```

[64]:
```
# Add a more meaningful label for the Outcome row in the Difference column
classes = [
    class_names[np.argmax(og_pred)],
    class_names[np.argmax(pp_pred)],
    f"Total Difference: {total_difference:.2f}"
]
```

```
comparison_df = pd.DataFrame(comparison_data, columns=dataset.columns)  #␣
↪Assuming column names come from dataset
comparison_df['Outcome'] = classes  # Add an outcome column
comparison_df.rename(index={0: 'Original (X)', 1: 'Pertinent Positive (X_PP)',␣
↪2: 'Difference (X_PP - X)'}, inplace=True)

# Transpose and display DataFrame with conditional formatting
comparison_df_t = comparison_df.transpose()
```

[63]:
```
comparison_df_t.style.apply(highlight_changes, col='Difference (X_PP - X)',␣
↪ncols=3, axis=1)
```

[63]:
```
<pandas.io.formats.style.Styler at 0x7fb7cebaf1a0>
```

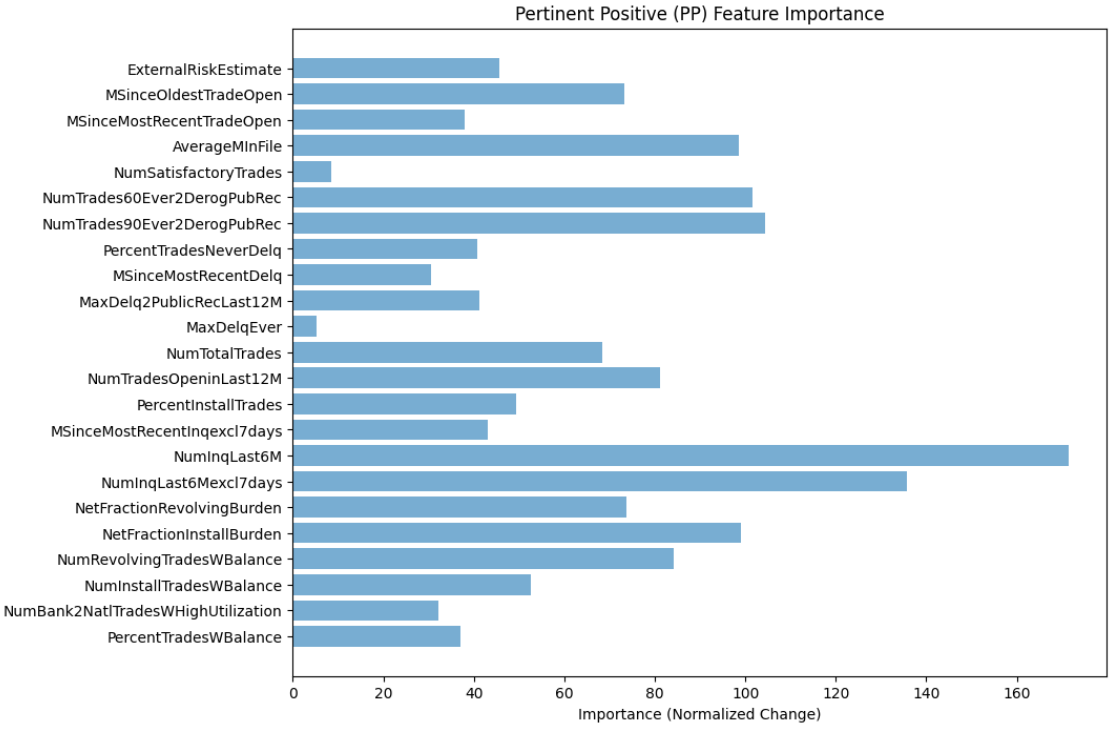
The following table compares an applicant's **Original Profile (X)** and their **Pertinent Positive (X_PP)**—a minimally adjusted profile that retains the same favorable "Good" classification. Key insights from the table follow:

1. **Feature adjustments**: Small modifications to features in **X_PP** ensure the "Good" outcome is preserved.
   - Example: **ExternalRiskEstimate** slightly decreases from 89 to ~79.3.
   - Example: **MSinceMostRecentTradeOpen** increases from 6 to ~15.
2. **Positive and negative changes**:
   - **Increases**: Features such as **MSinceMostRecentDelq** and **MaxDelq2PublicRecLast12M** show positive changes to maintain approval.
   - **Decreases**: Features such as **PercentTradesNeverDelq** and **NetFractionRevolvingBurden** reduce slightly while keeping the favorable outcome.
3. **Outcome consistency**: Both profiles result in a "Good" classification, confirming that small adjustments can help secure the positive result.
4. **Total Adjustment**: The cumulative change across all features totals ~**161**, reflecting the effort needed to retain the favorable classification.

Feature importance plot The following plot generates a **feature importance plot** to illustrate which features have the most substantial impact in differentiating the applicant's original profile from their pertinent positive (PP). The features with the tallest bars indicate those that require the most significant adjustment to retain the "Good" classification, highlighting their importance in preserving the favorable outcome. By focusing on these key features, the applicant gains insight into the characteristics that are essential to maintaining their approval status.

[65]:
```
# Compute the scaled absolute difference between original and PP (Pertinent␣
↪Positive)
feature_imp = np.abs(applicant - pertinent_positive) / np.std(X_train_norm.
↪astype('double'), axis=0)

# Define feature labels for the y-axis
```

```
feature_labels = dataset.columns  # Assuming column names from the dataset

# Reverse the order for plotting so the highest feature importance is at the top
performance = feature_imp.flatten()[::-1]
feature_labels = feature_labels[::-1]

# Plot the feature importance for Pertinent Positive
plt.figure(figsize=(10, 8))
plt.barh(range(len(feature_labels)), performance, align='center', alpha=0.6)
plt.yticks(range(len(feature_labels)), feature_labels)
plt.xlabel('Importance (Normalized Change)')
plt.title('Pertinent Positive (PP) Feature Importance')
plt.show()
```



Applicant 1168's loan application would likely remain in the "Good" status with a few key adjustments:

- **Increasing the time since the most recent trade (MSinceMostRecentTradeOpen)** has the most positive effect, suggesting that a longer gap since the last trade aligns with a stable credit profile.
- **Reducing PercentInstallTrades** and **NetFractionInstallBurden** also support the "Good" classification, indicating that lower reliance on installment trades and a reduced installment loan burden strengthen the profile.
- **Raising ExternalRiskEstimate** (risk score) further improves creditworthiness.

These adjustments align with a stable financial profile by reducing debt burden and enhancing credit history. This guidance is specific to Applicant 1168 and shows the model's perspective. If any adjustments seem unusual, it may signal areas for model refinement.