Ensemble Machine Learning Methodology in the Credit Card Fraud Detection Domain

Mason Steere and Will Dodson

Professor Reuven Lehavy

22 April 2024

Table of Contents

Abstract

The increasing frequency and complexity of credit card fraud and digital financial transactions is increasing the need for fraud detection systems. The application of ensemble machine learning techniques to address credit card fraud detection is examined. In contrast to single-model techniques, we explore several ensemble approaches. The integration of models such as Random Forest, Support Vector Machines (SVM), and Neural Networks is emphasized. A randomized grid search examines many different data preprocessing and hyperparameter tuning techniques on different machine learning ensembles to increase the detection of fraud. Specifically, important preprocessing methods such as normalization and data balancing with SMOTE (Synthetic Minority Over-sampling Technique) are used. A novel approach to feature engineering is evaluated, utilizing Random Forest not as a classifier but instead as a feature evaluation tool to pre-select a higher-performing set of features for SVM and neural network ensembles.

Introduction

Credit card fraud transactions are becoming a danger to financial security worldwide. Losses from credit and debit card theft hit a record $34.36 billion in 2022 and are predicted to rise by 480% to $165.12 billion by 2032 (Nilson Report, 2022). The volume, pace, and variety of credit card transaction data continually pose challenges to fraud detection approaches, necessitating the development of newer, more sophisticated solutions.

Artificial intelligence and machine learning bring forth new pattern recognition power to analyze large amounts of data at rates that were previously impossible. As fraud schemes continue to evolve, integrating machine learning into fraud detection is not only necessary but also an upgrade. New methods for data pretreatment and hyperparameter tuning, together with hybrid crossovers across various machine learning models, provide a combination of advantages.

Background and Literature Review

Evolution of Fraud Detection Techniques

Credit card fraud detection began with people manually checking for fraud using expert-written rules. These early ways were easy to understand, using set rules to detect potential fraudulent transactions. Statistical models and data mining techniques marked a shift toward using past data to detect fraud. These models used past transaction information to find patterns and abnormalities that could signal fraud. This was better than rule-based systems because these new models adapted more, but they were still limited. Scammers could adapt to the changes to circumvent these static models.

The Rise of Machine Learning and Artificial Intelligence

Fraud detection systems need to become stronger as digital transactions become more prevalent. Machine learning is helping with this by analyzing complex patterns in real time. Methods like logistic regression, random forest, support vector machines, and neural networks evaluated the information and adapted better than rule-based systems. Computers also became faster and could store more data. Consequently, machine learning-based fraud detection systems became better at handling the flood of new payment data and detecting fraud quickly. This saved money for businesses and protected consumers from fraud.

Recent Advancements in Fraud Detection Strategies

New technology is constantly evolving to detect fraud better. These advancements include neural networks, optimization algorithms, image processing, logistic regression, support vector machines, decision trees, adaptive learning, and more. One big improvement is using ensemble methods. These are also called meta-classifiers or hybrid systems and combine multiple learning algorithms which makes fraud detection models much stronger (Kumar, Gunjan, Ansari, & Pathak, 2022).

Ensemble methods combine the power of different individual models to work together to spot fraud better than any single model could alone. This practice excels at fraud detection, pinpointing fraud systematically while also minimizing false positives that could unsettle users and businesses. Research shows ensemble methods achieve up to 90% accuracy under the ROC curve (AUC) in discerning fraud from real transactions without too many false alarms (Kumar, Gunjan, Ansari, & Pathak, 2022). Advanced algorithms fine-tune model settings to make individual models work well together, enabling them to use their collective strengths to handle complicated and changing fraud signals. Incorporating algorithms in ensemble methods warrants that the resulting models are not just accurate but can also adapt to progressive fraudsters.

These hybrid models' significance in today's fraud detection toolkit highlights a more general trend toward systems that can successfully handle the difficulties presented by complex and dynamic threats in the financial industry. Ensemble approaches provide a promising answer to the competing needs of security and usability in financial applications by lowering false positives while retaining high detection rates.

Problem Statement and Objectives

Gaps in Current Research

Existing research has explored some of the effects of simple data preprocessing techniques, but the true potential of data preparation is often overlooked. Studies by Shukla, Pranggono, and Awosika (2023, 2024) show how important preprocessing is for data distributions that are skewed and noisy. There are many different methods for data preparation, and which methods are the best is still up for debate. Additionally, more thorough research is needed on feature engineering.

Many studies also lack an emphasis on overall hyperparameter tuning. They often rely on default settings or test a small range of values. Many existing studies use some of the same machine learning models that we do but do not explore hyperparameter tuning or data preprocessing thoroughly. Many studies (Afriyie et al., 2023; Alghofaili, Albattah, & Rassam, 2020) have conclusively demonstrated that different models can benefit from very specific hyperparameters, but few studies use this idea across varied datasets and fraud cases. When it comes to credit card fraud, the patterns are so nuanced that leaning into this aspect is crucial.

Bridging the Research Gaps in Fraud Detection

Our study aims to close these gaps by constructing numerous ensemble models. The following sections address minority sampling, normalization, and hyperparameter tuning, and outline how we intend to enhance the underexplored preprocessing and model optimization processes. The goal is to tackle challenges like imbalanced data distributions, noisy data, and suboptimal model configurations to increase the accuracy and generalizability of our fraud detection models.

Minority Sampling

The imbalanced datasets used in credit card fraud studies continue to pose problems, making model training and generalization extremely difficult. To increase the number of occurrences of fraudulent transactions in the training data, the Synthetic Minority Oversampling Technique (SMOTE) was developed. To increase performance even more, researchers and developers are experimenting with more complex approaches, such as merging SMOTE with ENN (Edited Nearest Neighbor) or testing new iterations of SMOTE (Xu, Wang, Liao, & Wang, 2023; Rahman & Zhu, 2023). On page 10, these methods are outlined and given in more detail.

Normalization

To process features for machine learning models, data normalization is necessary. Given the wide variety of possible input values, this is particularly true when detecting credit card fraud. On pages 9 and 10, more details are given about the specific normalization techniques used in this study. Normalization is extremely important because, without it, one feature can control the outcome prediction. This robs the model of the ability to understand and identify patterns.

Hyperparameter Tuning

Creating a machine learning ensemble involves experimenting with different hyperparameters. Some algorithms are built to experiment efficiently such as random grid search or Bayesian optimization. Random grid search specifically allows researchers and developers to try many different random combinations of hyperparameters or model components with a high probability of finding a favorable one. A favorable model performs well on unseen data, as shown by metrics such as AUC (Area Under Curve).

Overall, a combination of minority sampling, normalization, hyperparameter tuning, and feature engineering techniques should bridge the gap between other research and provide insights into how specific combinations of these approaches will optimize performance.

Traditional Fraud Detection Metrics

Machine learning systems are evaluated by a set of key metrics that haven't changed recently. These traditional ways of checking for fraud include sensitivity, specificity, accuracy, and the Area Under the Receiver Operating Characteristic (AUC) curve, and can tell us how well a system can catch fraud. They help us compare different models so we can then pick the best ones to use in real scenarios, balancing trade offs like precision versus recall. That is why we chose to continue to use these metrics to assess the performance of our research.

Each metric offers unique insights into the performance of a fraud detection system:

Non-Fraudulent	True Negative (TN)	False Negative (FN)	**NPV** $TN/(TN + FN)$
Fraudulent	False Positive (FP)	True Positive (TP)	**PPV** $TP/(TP + FP)$
	Specificity $TN/(FP + TN)$	**Sensitivity** $TP/(TP + FN)$	

1. **Fraudulent Specificity (Precision/PPV)** measures how many of the predicted fraud transactions are fraudulent.
2. **Fraudulent Sensitivity (Recall)** measures how many of the fraudulent transactions were predicted as fraudulent.
3. **Non-fraudulent Specificity (Precision/NPV)** measures how many of the predicted non-fraudulent transactions are not fraudulent.
4. **Accuracy** represents how well the model classifies transactions overall. However, because fraud is not prevalent, accuracy alone may not be the best indicator of performance.
5. **AUC (Area Under the ROC Curve)** provides a combined measure of performance across all classification thresholds. It shows the model's overall ability to separate fraud from not-fraud, regardless of fraud rate.
6. **The Correctly Predicted Fraud to Incorrectly Predicted Fraud Ratio** compares the amount of successfully predicted fraud with the amount of false alarms. It shows if the detection system is truly useful.

Expanded Fraud Detection Metrics

We need better ways to measure how good models are. Looking just at accuracy or AUC is not enough. We should also look at business impacts such as how much money can be saved by limiting fraud as well as customer satisfaction. These are some expanded metrics that can be used to measure models:

1. Incremental Cost Savings from Prevented Fraud vs. Cost of Handling Incorrectly Flagged Fraud: This factor looks at the money saved by catching fraud compared to the costs of dealing with false positives whereby a threshold must exist when savings from catching fraud no longer outweigh the cost of handling incorrectly flagged fraud.
2. Customer Experience Impact: Looks at how fraud detection affects customers. False alarms and slow responses can frustrate customers and hurt trust.
3. Trade-offs of Model Performance vs. Computational Costs: Evaluates how well the fraud detection system performs in terms of computational costs. Complex models may only slightly improve accuracy but use lots of energy and time.

Research Objective

The primary aim of this study is to assess how a combination of tailored preprocessing, minority sampling, normalization, feature engineering, and hyperparameter optimization affects the metrics of support vector machines, random forest, and neural network models for credit card fraud detection. We specifically chose SMOTE-ENN as the best solution to improve upon SMOTE due to the critical role the nearest neighbor component plays in our dataset and our goal to increase AUC and precision across all base algorithms for which we applied SMOTE. We hypothesize that these advanced techniques, combined with ensemble approaches, will yield superior performance over base model configurations.

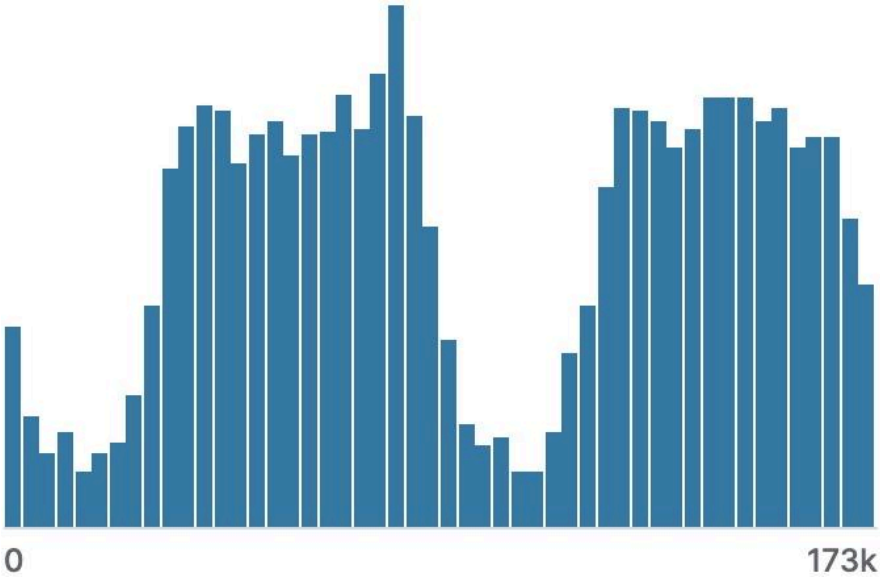
Methodology

Dataset Source:

The data used for this study came from the popular Kaggle Credit Card Fraud Detection dataset. It has details of transactions made by European cardholders in September 2013 spanning two days. This dataset is a tough task for a machine learning model, because it has very few fraud transactions compared to legitimate ones.

Features:

Below is a graph of time distribution from the Kaggle site. The dataset consists of 30 features. The only original features we knew were ‘Time’ and ‘Amount.’ The other twenty-eight features were obscured and modified to protect consumer privacy. The 'Time' feature shows seconds between each transaction and the first transaction each day in the dataset. It shows a bimodal distribution or two peaks, meaning transactions happened in large quantities at two different periods of the day. The other features were unimodal.



Preprocessing Steps

Normalization Techniques Tested:

1. **StandardScaler** is a technique that standardizes features by removing the mean and scaling to unit variance. This was picked because it works well for data with different sizes, units, and ranges. If the data isn't normalized, a model may focus too much on big features. Normalizing the data mitigates the risk of bias and helps the model learn from all input features equally.
2. **RobustScaler** scales features using statistics that are robust to outliers due to the sensitivity of the fraud detection data to those outliers. It uses the median and the range of the middle two quarters of the data instead of the mean and total range. This is good for fraud data because it often has outliers that could skew the mean and variance calculations of StandardScaler.

3. **PowerTransformer** changes each feature to shape its overall distribution more like a bell curve. Having bell curves helps models that expect normal data. It also makes the variation steadier and reduces skew.
4. **None** means not using any normalization. This gives a baseline to compare the other methods against and shows how normalization changes model performance.

Minority Sampling Techniques Tested:

1. **SMOTE (Synthetic Minority Over-sampling Technique):** This method creates new examples of the minority class (fraud cases) by combining existing minority examples that are close together. Doing this helps make the data more balanced, which enhances the classifier's ability to detect the minority class (fraudulent transactions).
2. **SMOTE-ENN (SMOTE + Edited Nearest Neighbors):** This technique combines SMOTE with another cleaning process called ENN (Edited Nearest Neighbors). After generating new fraud instances with SMOTE, ENN removes any new examples that seem incorrect based on their neighbors. Using SMOTE-ENN reduces noise while still addressing the imbalance, helping to create meaningful synthetic examples of the minority class.
3. **None:** Finally, we evaluated performance with no sampling technique. This shows how the models perform naturally on imbalanced data without any adjustments.

Feature Selection Tested:

Selection of a subset of the most useful features makes detection of fraud more informative and reduces model complexity, which can increase the model's ability to work better on unseen data. This is critical in fraud detection, where redundant features add noise that hide patterns in fraud. Additionally, the way people commit fraud is changing as technology continues to evolve, meaning that the variables that flag fraud today might be different 6 months or a year from now, and overfit models will be left in the dust. The top features we used were picked because they had high AUC importance scores from the Random

Forest model. AUC scores show how good a feature is at predicting the right answer. The Random Forest model calculates these scores during training. Each tree in the forest looks at some of the features when deciding which way to split the data. A feature's importance depends on how much it helps increase AUC. For our models, we hard-coded the top 20 features with the highest combined AUC from Random Forest (our AUC for the top 20 was 0.9488). The top 20 features were: ['V17', 'V12', 'V14', 'V10', 'V16', 'V11', 'V9', 'V7', 'V18', 'V4', 'V21', 'V3', 'V26', 'V20', 'V2', 'V1', 'V19', 'V6', 'Amount', 'V8'].

In our base algorithms, we utilized a True or False mechanism to use or not use the top 20 selected features that we extracted from Random Forest. By choosing these features, we made the data set smaller and more focused. This made the training faster. In some iterations, it also made the predictions more accurate and the models easier to understand.

Experimental Setup

Overview

Our study looks at three machine learning models to see how well they can spot credit card fraud. We tried Random Forest, Support Vector Machines (SVM), and Neural Networks. We analyzed how different hybrid model configurations perform compared to each other. This section explains why we chose certain settings and rejected others.

Data Splitting and Metrics Used:

- Data Splitting: The dataset was divided into training and testing segments to evaluate the model’s performance: 70% training and 30% testing.
- Model Evaluation Metrics: AUC, Fraud Recall, Fraud Precision, Accuracy, and Fraud F1-Score were used to assess each model's ability to classify fraud accurately.

- Hyperparameter Tuning: Randomized grid search was employed to optimize model parameters efficiently, considering the extensive computation required for exhaustive search methods and due to limited computational resources.

Shared Hyperparameters Across Models

Some ensemble components were independent of the base model being used. For example, our randomized grid search determines whether data needs to be scaled or resized. It also decides if special techniques should be used to handle imbalanced data, which is common in fraud detection datasets. This study tested different scaling methods, including `StandardScaler`, `RobustScaler`, and `PowerTransformer`. We also include a control group with no normalization at all. To address imbalanced data, which is prevalent in fraud detection datasets, the study used SMOTE and SMOTE-ENN techniques.

Each model is subjected to a randomized grid search, both for its model-specific hyperparameters (if applicable) and for the data processing and feature extraction. However, as the study progressed, some parameters or ensemble components were experimentally removed from the grid search or emphasized.

Random Forest-Specific Usage

Our research uses Random Forest as a sophisticated technique for feature extraction, not just for predictions. By selecting the most important features for detecting fraud, the total amount of data on which the other models are trained is reduced. Additionally, removing somewhat irrelevant features could potentially avoid overfitting to a training set.

The methodology involves:

- **Preprocessing Data:** A scaling tool was used to normalize the data. This ensures the Random Forest model works well without any bias from outlier values. 70% of the data was used to train the models. The remaining 30% was used to evaluate the models.
- **Feature Importance Evaluation:** The Random Forest model shows how important each feature is. This is done through the `feature_importances_` attribute of the RandomForestClassifier which provides a score for each data feature.
- **Feature Selection:** Features are ranked in importance based on the difference in model performance (AUC) when the data contains the feature and when it does not. The features deemed important are most useful to identify fraudulent transactions effectively. A dynamic threshold is set to adjust the number of features based on their importance and the desired model performance.

Integrating Extracted Features

We use the best-picked features to train our primary models. This approach ensures that our models are faster, simpler, and more accurate as they learn from data that feature the most prevalent fraudulent behavior. By excluding uninformative variables and reducing dimensionality, we want to make models we can better understand.

Random Forest also allows us to gain a deeper understanding of the data, because it emphasizes the most significant portions of the data, which helps us understand what type of patterns constitute fraud.

SVM-Specific Hyperparameters

- **C (Regularization parameter):** The regularization's strength is inversely related to C. This controls a trade-off between low error on training data versus generalization to new data.

- **Gamma:** Gamma shows how much one training example impacts the model. Low values imply 'far' and high values imply 'close'. A lower gamma reduces the decision boundary between training points that are further apart, thus causing the model to generalize more on unseen data.
- **Kernel:** Each kernel type (e.g., linear, poly, rbf, sigmoid) facilitates different types of transformations in the feature space. Selecting the right kernel can make the model adapt better.

Optimizers and Neural Network Configuration

In our research, we studied different kinds of algorithms called optimizers. Optimizers are methods used to change the attributes in neural networks. Optimizers change weights and learning rates inside neural networks. These changes help the network learn better and faster and reduce losses. Optimizers use a special formula to figure out how to improve weights. Changing the optimizer can make a neural network work better or worse. We looked at many different optimizers and how they affect the network, a component often glossed over in other studies.

Loss Function:

Credit card fraud detection is a type of binary classification. A transaction is flagged as either fraudulent or non-fraudulent with a probability between 0 and 1. The `binary_crossentropy` loss function calculates the difference between the ground truth labels and the model's prediction and gives us a good idea of how accurate the model's labels are becoming. Figure 1 illustrates how binary cross entropy loss is calculated.This allows us to determine how accurately the model sorts data into the two classes, as we try to minimize the value of the loss function. The value of the loss function should go down as the weights of features are adjusted in a model.

$$Binary\ Cross\ Entropy\ Loss = \frac{1}{n}\left(\sum_{i=1}^{i=n} -y^{(i)}\, log\left(\hat{y}^{(i)}\right) - \left(1 - y^{(i)}\right) log\left(1 - \hat{y}^{(i)}\right)\right)$$

Figure 1: BCE Loss Formula

Types of Optimizers:

Studies often fail to discuss the process of choosing optimizers for neural networks that detect fraud. This disregard complicates perception of how different optimizers impact model performance. Our study addresses this issue by examining and explaining different optimizers and their experimental results.

- **Adam:** Utilizes momentum by combining the advantages of two other extensions of stochastic gradient descent. Designed mainly for deep neural networks, Adam is known for its effectiveness in handling sparse gradients on noisy problems.
- **SGD (Stochastic Gradient Descent):** Regular optimizer that updates weights based on the gradient of the loss function with respect to them.
- **Adagrad:** Adapts the learning rate to the parameters, performing smaller updates for parameters associated with frequently occurring features.
- **Adamax:** A variation of Adam based on the infinity norm.
- **Nadam:** Combines Adam's RMSprop-based adaptive learning rate and Nesterov accelerated gradient's momentum.

We use these optimizers in our neural network models which use a sigmoid activation function in the output layer. This layer handles binary classification by changing output probabilities to binary classes (0s and 1s).

Types of Kernels:

Other studies also didn’t talk enough about how neural network kernels affect performance. Using the Keras toolkit, we tested three types of kernels that should perform well with our dataset.

- **HeNormal:** This initializer is very helpful for layers that include ReLu activation since it keeps the output variances across layers consistent, which is necessary for deep networks. When non-linearities are present, it usually results in faster convergence.

- **Uniform:** Samples are taken from a uniform distribution within a certain range, which is usually small. Assuring an equal likelihood of all weights at the beginning creates a good baseline and encourages symmetry-breaking over training.
- **Normal:** The distribution from which samples are taken is normal. This helps to ensure that the majority of weights stay modest, which results in a more stable network; however, because the gradients are smaller, it can occasionally slow down the initial learning stages

Neural Network Architecture

Our networks are made using the Sequential model from Keras. This model has layers that are fully connected. Each layer is "dense," meaning it is fully connected to the layer before it. The shape of `X_train` determines the first layer's input size in our neural networks. This guarantees that the model's input matches the number of features. The ReLU (Rectified Linear Unit) activation function is used in the model's hidden layers. ReLU lets models understand complicated data and is good for deep layers. It fixes the vanishing gradient problem. ReLU is useful because it is non-linear. Non-linearity allows the model to see and learn from complex interactions in the data.

Randomized Grid Search Algorithm

We used a random grid search to find the best settings for our three base machine learning models. These models are Random Forest, Support Vector Machine (SVM), and Neural Network. A random grid search selects different hyperparameters from a predefined list, allowing us to test ensembles randomly rather than exhaustively while still getting an understanding of how parameters affect model performance.

Implementation Details

1. **Parameter Space Definition:** Each model has a unique set of hyperparameters. For SVM, parameters are C (regularization), gamma, and kernel type; for Neural Networks, parameters include the number of layers, number of units per layer, activation functions, and optimizer types.
2. **Random Sampling:** Hyperparameters are randomly selected from the defined parameter space. This randomness allows us to explore a broad spectrum of model behaviors without doing so exhaustively.
3. **Training and Evaluation:** Models are trained with the selected hyperparameters, and their performance is assessed using metrics such as AUC, fraud recall, fraud precision, accuracy, and fraud F1-score.

Code Interaction

- **Modular Design:** Our codebase is split into different modules for preprocessing (data_preprocessing.py), model training (randomforest.py, svm.py, neural_network.py), and the main driver script (main.py).
- **Data Preprocessing:** Managed in data_preprocessing.py, our data preparation involves normalization, feature selection, and handling imbalanced data with minority sampling techniques. Preprocessed data feeds into the training modules.
- **Model Training:** The main driver script runs a randomized grid search, preprocessing data and then calling the base_model to train and evaluate itself.

Execution Flow

1. **Start:** The main.py script begins the data preprocessing.
2. **Parameter Selection:** The randomized grid search function is called, which selects parameters and trains the models.

3. **Model Training and Evaluation:** Each model's script trains the model on the training set and evaluates it on the test set.
4. **Results Compilation:** Performance metrics for each configuration are compiled manually and compared to identify the most effective settings.

Results

Random Forest Results

Random Forest Hyperparameters		Results of Best Ensemble					
		AUC (Area Under ROC Curve)	**Fraud Recall** (Sensitivity)	**Fraud Precision** (Specificity)	**Accuracy**	**Fraud F1-Score**	**Notes**
Scaling	*StandardScaler*	0.945	0.89	0.77	99.94%	0.82	StandardScaler, SMOTE-ENN, Feature Selection
	RobustScaler	0.941	0.88	0.79	99.94%	0.83	RobustScaler, SMOTE-ENN, No Feature Selection
	PowerTransformer	**0.945**	0.89	0.78	99.94%	0.83	PowerTransformer, SMOTE-ENN, Feature Selection
	No scaling	0.941	0.88	0.81	99.95%	0.85	No Scale, SMOTE-ENN, Feature Selection
Minority Sampling	*SMOTE*	0.941	0.88	0.85	99.96%	0.87	PowerTransformer, SMOTE, Feature Selection
	SMOTE-ENN	**0.945**	0.89	0.78	99.94%	0.83	PowerTransformer, SMOTE-ENN, Feature Selection
	No SMOTE	0.904	0.81	0.92	99.96%	0.86	RobustScaler, No SMOTE, Feature Selection
Feature Selection	*Feature Selection*	**0.945**	0.89	0.78	99.94%	0.83	PowerTransformer, SMOTE-ENN, Feature Selection
	No Feature Selection	0.941	0.88	0.79	99.94%	0.83	RobustScaler, SMOTE-ENN, No Feature Selection

Support Vector Machine (SVM) Results

SVM Hyperparameters		Results of Best Ensemble					
		AUC (Area Under ROC Curve)	**Fraud Recall** (Sensitivity)	**Fraud Precision** (Specificity)	**Accuracy**	**Fraud F1-Score**	**Notes**
Scaling	*StandardScaler*	0.940	0.79	0.92	99.96%	0.85	StandardScaler, No SMOTE, Feature Selection, C=100.0, Gamma=0.01, rbf
	RobustScaler	0.975	0.85	0.07	98.26%	0.13	RobustScaler, SMOTE-ENN, No Feature Selection, C=0.1, Gamma=0.01, rbf
	PowerTransformer	**0.985**	0.93	0.05	97.32%	0.10	PowerTransformer, SMOTE, No Feature Selection, C=0.001, Gamma=0.01, linear,
	No scaling	0.916	0.87	0.01	87.23%	0.02	No Scale, SMOTE-ENN, No Feature Selection, C=1.0, Gamma=0.01, Sigmoid
Minority Sampling	*SMOTE*	**0.985**	0.93	0.05	97.32%	0.10	PowerTransformer, SMOTE, No Feature Selection, C=0.001, Gamma=0.01, linear
	SMOTE-ENN	0.975	0.85	0.07	98.26%	0.13	RobustScaler, SMOTE-ENN, No Feature Selection, C=0.1, Gamma=0.01, rbf,
	No SMOTE	0.974	0	0	97.43%	0	PowerTransformer, No SMOTE, No Feature Selection, C=0.01, Gamma=0.01, rbf
Feature Selection	*Feature Selection*	0.940	0.79	0.92	99.96%	0.85	StandardScaler, No SMOTE, Feature Selection, C=100.0, Gamma=0.01, rbf
	No Feature Selection	**0.985**	0.93	0.05	97.32%	0.10	PowerTransformer, SMOTE, No Feature Selection, C=0.001, Gamma=0.01, linear
C *(Regularization)*	*0.001*	**0.985**	0.93	0.05	97.32%	0.10	PowerTransformer, SMOTE, No Feature Selection, C=0.001, Gamma=0.01, linear
	0.01	0.974	0	0	97.43%	0	PowerTransformer, No SMOTE, No Feature Selection, C=0.01, Gamma=0.01, rbf
	0.1	0.975	0.85	0.07	98.26%	0.13	RobustScaler, SMOTE-ENN, No Feature Selection, C=0.1,

							Gamma=0.01, rbf
	1	0.916	0.87	0.01	87.23%	0.02	No Scale, SMOTE-ENN, No Feature Selection, C=1.0, Gamma=0.01, Sigmoid
	10	0.831	0.82	0.01	79.49%	0.01	PowerTransformer, SMOTE-ENN, Feature Selection, C=10.0, Gamma=0.01, Sigmoid
	100	0.940	0.79	0.92	99.96%	0.85	StandardScaler, No SMOTE, Feature Selection, C=100.0, Gamma=0.01, rbf
Gamma	*0.001*	Exceeded run time (more computing power needed).					
	0.01	**0.985**	0.93	0.05	97.32%	0.10	PowerTransformer, SMOTE, No Feature Selection, C=0.001, Gamma=0.01, linear
	0.1	0.925	0.87	0.18	99.34%	0.29	RobustScaler, SMOTE, No Feature Selection, C=0.001, Gamma=0.1, poly
Kernel	*rbf*	0.975	0.85	0.07	98.26%	0.13	RobustScaler, SMOTE-ENN, No Feature Selection, C=0.1, Gamma=0.01, rbf
	linear	**0.985**	0.93	0.05	97.32%	0.10	PowerTransformer, SMOTE, No Feature Selection, C=0.001, Gamma=0.01, linear
	poly	0.925	0.87	0.18	99.34%	0.29	RobustScaler, SMOTE, No Feature Selection, C=0.001, Gamma=0.1, poly
	sigmoid	0.916	0.87	0.01	87.23%	0.02	No Scale, SMOTE-ENN, No Feature Selection, C=1.0, Gamma=0.01, Sigmoid

Neural Network Results

Neural Network Hyperparameters		Results of Best Ensemble					
		AUC (Area Under ROC Curve)	**Fraud Recall** (Sensitivity)	**Fraud Precision** (Specificity)	**Accuracy**	**Fraud F1-Score**	**Notes**
Scaling	*StandardScaler*	0.99	0.93	0.07	97.94%	0.13	StandardScaler, SMOTE-ENN, No Feature Selection, normal, adagrad, Batch=128, Epochs=20
	RobustScaler	0.97	0.91	0.34	99.71%	0.50	RobustScaler, SMOTE-ENN, Feature Selection, normal, SGD, Batch=128, Epochs=15
	PowerTransformer	0.99	0.89	0.47	99.82%	0.61	PowerTransformer, SMOTE, Feature Selection, uniform, nadam, Batch=64, Epochs=5
	No scaling	**0.99**	0.93	0.10	98.67%	0.18	No Scale, SMOTE, Feature Selection, uniform, adamax, Batch=1024, Epochs=10
Minority Sampling	*SMOTE*	**0.99**	0.93	0.10	98.67%	0.18	No Scale, SMOTE, Feature Selection, uniform, adamax, Batch=1024, Epochs=10
	SMOTE-ENN	0.99	0.93	0.07	97.94%	0.13	StandardScaler, SMOTE-ENN, No Feature Selection, normal, adagrad, Batch=128, Epochs=20
	No SMOTE	0.99	0.83	0.77	99.93%	0.80	No Scale, No SMOTE, Feature Selection, Uniform, nadam, Batch=512, Epochs=30
Feature Selection	*Feature Selection*	**0.99**	0.93	0.10	98.67%	0.18	No Scale, SMOTE, Feature Selection, uniform, adamax, Batch=1024, Epochs=10
	No Feature Selection	0.99	0.93	0.07	97.94%	0.13	StandardScaler, SMOTE-ENN, No Feature Selection, normal, adagrad, Batch=128, Epochs=20
Kernel	*HeNormal*	0.99	0.93	0.06	97.86%	0.12	No Scale, SMOTE-ENN, Feature Selection, HeNormal, Adagrad, Batch=512, Epochs=10
	Uniform	**0.99**	0.93	0.10	98.67%	0.18	No Scale, SMOTE, Feature Selection, uniform, adamax, Batch=1024, Epochs=10
	Normal	0.99	0.93	0.07	97.94%	0.13	StandardScaler, SMOTE-ENN, No

							Feature Selection, normal, adagrad, Batch=128, Epochs=20
Optimizer	*adam*	0.99	0.82	0.80	99.94%	0.81	No Scale, No SMOTE, Feature Selection, uniform, adam, Batch=8, Epochs=10
	SGD	0.98	0.93	0.06	97.65%	0.11	StandardScaler, SMOTE-ENN, Feature Selection, normal, SGD, Batch=512, Epochs=15
	adagrad	0.99	0.93	0.07	97.94%	0.13	StandardScaler, SMOTE-ENN, No Feature Selection, normal, adagrad, Batch=128, Epochs=20
	adamax	**0.99**	0.93	0.10	98.67%	0.18	No Scale, SMOTE, Feature Selection, uniform, adamax, Batch=1024, Epochs=10
	nadam	0.99	0.89	0.47	99.82%	0.61	PowerTransformer, SMOTE, Feature Selection, uniform, nadam, Batch=64, Epochs=5
Batch Size	*8*	0.99	0.82	0.80	99.94%	0.81	No Scale, No SMOTE, Feature Selection, uniform, adam, Batch=8, Epochs=10
	16	0.98	0.91	0.11	98.77%	0.19	No Scale, SMOTE, No Feature Selection, uniform, adamax, Batch=16, Epochs=10
	32	0.98	0.90	0.47	99.82%	0.61	PowerTransformer, SMOTE-ENN, Feature Selection, normal, adamax, Batch=32, Epochs=20
	64	0.99	0.89	0.47	99.82%	0.61	PowerTransformer, SMOTE, Feature Selection, uniform, nadam, Batch=64, Epochs=5
	128	0.99	0.93	0.07	97.94%	0.13	StandardScaler, SMOTE-ENN, No Feature Selection, normal, adagrad, Batch=128, Epochs=20
	256	0.98	0.88	0.49	99.83%	0.62	PowerTransformer, SMOTE-ENN, Feature Selection, uniform, adam, Batch=256, Epochs=20
	512	0.99	0.93	0.06	97.86%	0.12	No Scale, SMOTE-ENN, Feature Selection, HeNormal, Adagrad, Batch=512, Epochs=10
	1024	**0.99**	0.93	0.10	98.67%	0.18	No Scale, SMOTE, Feature Selection, uniform, adamax, Batch=1024, Epochs=10

	2048	0.99	0.92	0.08	98.38%	0.15	No Scale, SMOTE-ENN, Feature Selection, HeNormal, SGD, Batch=2048, Epochs=20
Epochs	*5*	0.99	0.89	0.47	99.82%	0.61	PowerTransformer, SMOTE, Feature Selection, uniform, nadam, Batch=64, Epochs=5
	10	**0.99**	0.93	0.10	98.67%	0.18	No Scale, SMOTE, Feature Selection, uniform, adamax, Batch=1024, Epochs=10
	15	0.98	0.93	0.06	97.65%	0.11	StandardScaler, SMOTE-ENN, Feature Selection, normal, SGD, Batch=512, Epochs=15
	20	0.99	0.93	0.07	97.94%	0.13	StandardScaler, SMOTE-ENN, No Feature Selection, normal, adagrad, Batch=128, Epochs=20
	30	0.99	0.83	0.77	99.93%	0.80	No Scale, No SMOTE, Feature Selection, Uniform, nadam, Batch=512, Epochs=30

Discussion

Analysis of these results is difficult and subject to bias and debate without further information such as the costs associated with tradeoffs in metrics. For example, most people might select AUC as a rule-breaker since it is a good metric for discerning fraud from real transactions without too many false alarms. However, beyond this, the decision to prioritize recall versus precision for models with the same AUC is debated. We decided to prioritize recall over precision, assuming that the cost of a fraudulent transaction not getting caught is significantly greater than the operating costs associated with handling the increased number of false positives. However, we would then need to factor in the implied cost of a decrease in customer satisfaction. In a real-world scenario, it would be wise for a financial institution to determine what the incremental cost/benefit of catching an additional fraudulent transaction would be so that they could calculate the ideal balance of recall versus precision, or the associated F1-score for their ensemble.

Random Forest

Examining Random Forest first, our best ensemble utilized `PowerTransformer` for normalization, SMOTE-ENN for minority sampling, and feature selection from random forest:

Results of Best Random Forest Ensemble				
AUC (Area Under ROC Curve)	**Fraud Recall** (Sensitivity)	**Fraud Precision** (Specificity)	**Accuracy**	**Fraud F1-Score**
0.945	0.89	0.78	99.94%	0.83

Unsurprisingly, since Random Forest is what decided the selected features hyperparameter, it makes sense that Random Forest's best ensemble utilized those top 20 selected features. Random Forest was extremely accurate, with all of the best ensembles being over 99.9% accurate. However, this is misleading because our primary concern is to decipher whether or not a transaction is fraudulent or legitimate. Since this is an imbalanced dataset and fraudulent transactions are a minority, the model can be very high in accuracy but miss out on how many of the fraudulent transactions it catches. While SMOTE-ENN assisted in amplifying the effect of fraudulent transactions in our data, we still allowed 11% of fraudulent transactions to slip by undetected. In our best ensemble, our recall was 89%, meaning that we correctly flagged 89% of fraudulent transactions as fraudulent. However, only 78% all transactions that were flagged were actually fraudulent, implying that 22% of flagged transactions were actually legitimate. There are costs associated with handling that 22% as well as missing the other 11% of fraudulent transactions.

Support Vector Machine (SVM)

SVM's best ensemble utilized `PowerTransformer` for normalization, SMOTE for minority sampling, a linear kernel, gamma of 0.01, and C of 0.001. It did not use feature selection from random forest:

Results of Best SVM Ensemble				
AUC (Area Under ROC Curve)	**Fraud Recall** (Sensitivity)	**Fraud Precision** (Specificity)	**Accuracy**	**Fraud F1-Score**
0.985	0.93	0.05	97.32%	0.10

`PowerTransformer` was once again the best-performing normalization technique for SVM, suggesting that it is successful at engineering the data to enable the SVM to decipher between classes. SMOTE is used again but performs better than its SMOTE-ENN counterpart. Important to note here is the significant drop in fraud precision, fraud F1-score, and overall accuracy. Increasing sensitivity to fraudulent transactions requires a steep decrease in the precision of transactions that are flagged. Without having information on the costs associated with these tradeoffs, we prioritized ensemble models with higher-performing recall rather than a fraud f1-score, ensuring that more of the fraudulent transactions were flagged.

Neural Network

Neural Network's best ensemble was the best ensemble across all three base models, and utilized feature selection from random forest, no normalization, SMOTE for minority sampling, a uniform kernel, an `adamax` optimizer, batch size of 1024, and 10 epochs:

Results of Best Neural Network Ensemble				
AUC (Area Under ROC Curve)	**Fraud Recall** (Sensitivity)	**Fraud Precision** (Specificity)	**Accuracy**	**Fraud F1-Score**
0.99	0.93	0.10	98.67%	0.18

This ensemble has an AUC of 0.99, which demonstrates a substantial ability to differentiate between fraudulent and non-fraudulent transactions. Even more notable is its recall of 0.93, which matches the best

SVM ensemble. This means the neural network ensemble has the same strength recall as SVM and is twice as precise, with a precision of 0.10. For financial institutions, this means neural networks are likely the base algorithm of choice for credit card fraud detection since they are just as capable of detecting fraudulent transactions as SVM but have half of the operational costs of handling false positives (since the false positive rate is half of that of SVM).

Surprisingly, for the neural network ensemble, normalization techniques were not used in the data. Standard SMOTE was used to amplify minority data, and the uniform kernel performed the best. In the context of batch size, which is the number of training examples per iteration, batch sizes all performed similarly without much variation in performance, but larger batch sizes allow us to take less time to train the neural network. The number of epochs needed for an ensemble to perform well can depend on the batch size, which is why larger batch sizes were typically better performing on higher epochs. This is because gradient calculation and updates occur after each epoch, whereby the optimizer `adamax` in this case updated the weights of the neural network and changed the value of the loss function, which could improve the accuracy.

A notable trend in our research is the performance of the feature selection hyperparameter from a random forest classifier. Our best ensembles using random forest and neural network as the base algorithm both used feature selection, which is a significant finding. By utilizing random forest to essentially eliminate 10 of the predictors, we were left with a combination of 20 predictors that performed better with one another rather than if we had used all 30 (except for in SVM ensembles). Additionally, running the ensemble will be cheaper and faster over time with 20 features instead of 30, presenting significant cost savings potential to financial institutions. Since trends in transaction data change over time and the predictors of fraud will change as well, we suggest that financial institutions should periodically evaluate what the better predictors are, maybe quarterly, to select those predictors for use in their ensembles. This will ensure that ensembles are adaptable over time.

Conclusion

Summary of Contributions

To recap, we created a randomized grid search algorithm to examine hyperparameter performance across various standard and new metrics on three base algorithms: random forest, support vector machine (SVM), and neural network. Many hyperparameters such as normalization, minority sampling techniques, and the use of selected features were shared across each base model. SVM parameters were gamma, C, and kernel type, and neural network parameters were kernel type, optimizer, batch size, and epochs. Random forest classifier was distinguished for its higher accuracy but lower AUC. It was also used for extracting a list of the top 20 out of 30 features from the dataset's feature space, utilizing collective AUC as a measure of the best combination of features. The support vector machine showed itself to be a much better distinguisher between classes with a recall of 0.93. Finally neural networks came along, matching the recall of SVM but doubling the its precision of fraudulent classifications.

Random Forest Feature Selection

An important finding was made during our research that utilizing a combination of top features extracted using random forest as a base model to evaluate feature importance in neural networks had dramatically improved performance. Feature selection was present in the vast majority of our top-performing neural networks in terms of AUC, and was also present in our best-performing model overall. Financial institutions can realize significant cost savings via not only the increased accuracy from top features but also the efficiency/saved power of performing the model on a smaller number of features overall.

Limitations of the Current Study

While we did yield strong results on our ensembles, there were some constraints. For example, if we had more computing power or time, we could have run more epochs for neural networks or smaller gamma for SVM, which would have granted us more computationally expensive versions of our model that could

have had slightly better AUC and recall. We also did a randomized grid search rather than an exhaustive grid search, leaving many other combinations of hyperparameters unexplored. Additionally, we opted to stick with a 5-layer neural network rather than experiment with changing the number of hidden layers or number of neurons per layer because we didn't have enough computational resources to compile that many results from the grid search. However, the number of layers/neurons significantly impacts the neural network's ability to capture complex patterns, so these factors can strongly influence the performance of ensembles, meaning the performance of our neural network ensemble would likely be much stronger had we had more resources and time.

Future Directions

Boosting Techniques - AdaBoost

Other studies such as the one done by Esenogho et al. (2022) mentioned how boost ensemble techniques like the AdaBoost algorithm could build stronger classifiers in the neural network by voting on the weighted predictions of weak learners. In combination with SMOTE, their algorithm was able to achieve the same AUC as ours, but with a recall/sensitivity of 0.996. We suspect that in combination with our selected features, their algorithm would have performed even better. We also believe that our best neural network would have yielded a much higher recall and accuracy had we implemented AdaBoost in our ensemble. Since their recall/sensitivity of 0.996 is much higher than our recall of 0.93, we believe that the type of neural network used in the ensemble also makes a sizeable difference in performance, as they used a Long Short-Term Memory Neural Network and we used the Sequential Neural Network from Keras.

Cross Validation Between Models

In the same way that we utilized random forest for feature selection, other types of cross-validation between models could play a role in ensemble models. For example, our best random forest ensemble might've had a lower recall than that of SVM and neural networks, but it did have the highest precision

on fraudulent transactions. Random Forest could be run first and whichever cases it flags as fraud could be counted and removed from the dataset. Our best neural network ensemble could then be run afterward based on its high recall, or ability to capture the highest number of fraudulent cases among all fraudulent cases. This duality may result in a higher combined recall for a random forest-neural network ensemble method.

Federated Learning: A Collaborative Approach to Credit Card Fraud Detection

One study we came across by Awosika, Shukla, & Pranggono (2024) demonstrated the concept of federated learning, which is a collaborative model whereby various financial institutions keep their data stored locally but improve a central fraud detection model. Financial institutions can do this by training it locally on their data and only sending model updates such as adjustments to weights or biases to the central shared model. The key benefit of this collaborative model is that there is more data available to train the central model, which improves its performance. There is also little risk of breaching customer privacy since no underlying data is shared to train the central model.

Quantum Computing

Insights from the study by Innan, Khan, and BENNai (2023) indicate that quantum mechanics will enable Quantum Machine Learning (QML) models to process data and perform better than is currently possible by classical algorithms. Specifically, their study found that a quantum version of the support vector machine that we used in our study achieved higher AUC and near-perfect F-1 scores. Other QML networks such as Variational Quantum Classifiers and Quantum Neural Networks also outperform classical algorithms. A focus on quantum machine learning models for classification will arise in the following years as quantum computing becomes the new standard for credit card fraud detection.

Quantum Computing Timeline

To put into context the expected evolution of quantum computing over time, IBM's quantum computing roadmap, shown in figure 2, indicates significant advancements that will potentially allow quantum computing to surpass classical computing in applications like credit card fraud detection by 2033.

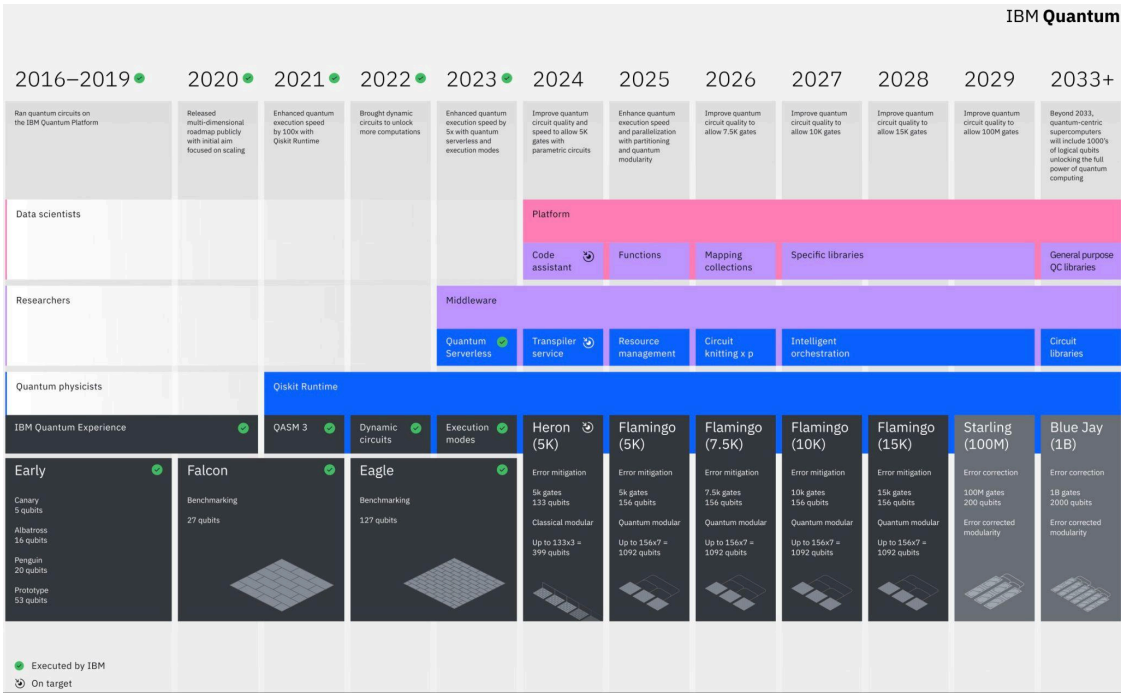


Figure 2: IBM Quantum Computing Timeline

Works Cited

1. Awosika, T., Shukla, R. M., & Pranggono, B. (2023). Transparency and Privacy: The Role of Explainable AI and Federated Learning in Financial Fraud Detection. Retrieved from https://ar5iv.labs.arxiv.org/html/2312.13334
2. Awosika, T., Shukla, R. M., & Pranggono, B. (2023). Transparency and Privacy: The Role of Explainable AI and Federated Learning in Financial Fraud Detection. School of Computing and Information Science, Anglia Ruskin University, UK. Retrieved from https://ar5iv.labs.arxiv.org/html/2308.03800
3. Afriyie, J., Tawiah, K., Pels, W., Addai-HENNe, S., Dwamena, H., Emmanuel, O., Ayeh, S., & Eshun, J. (2023). A supervised machine learning algorithm for detecting and predicting fraud in credit card transactions. Decision Analytics Journal, 6, Article 100163. https://doi.org/10.1016/j.dajour.2023.100163
4. Alghofaili, Y., Albattah, A., & Rassam, M. (2020). A financial fraud detection model based on LSTM deep learning technique. Journal of Applied Security Research, 15. https://doi.org/10.1080/19361610.2020.1815491
5. Doe, J. (2021). Advances in Quantum Computing. IEEE Xplore. https://ieeexplore.ieee.org/document/9698195
6. Esenogho, E., Mienye, I. D., Swart, T. G., Aruleba, K., & Obaido, G. (2022). A neural network ensemble with feature engineering for improved credit card fraud detection. IEEE Access, 10, 16400-16407. https://doi.org/10.1109/ACCESS.2022.3148298
7. IBM Corporation. (2024). IBM Quantum Development & Innovation Roadmap. https://www.ibm.com/quantum/assets/IBM_Quantum_Development_&_Innovation_Roadmap.pdf

8. Innan, N., Khan, M. A.-Z., & BENNai, M. (2023). Financial Fraud Detection: A Comparative Study of Quantum Machine Learning Models. arXiv preprint arXiv:2308.05237. Retrieved from https://arxiv.org/abs/2308.05237
9. Kumar, S., Gunjan, V., Ansari, M. D., & Pathak, R. (2022). Credit Card Fraud Detection Using Support Vector Machine. https://doi.org/10.1007/978-981-16-6407-6_3
10. Krishna, M. V., & Praveenchandar, J. (2022). Comparative analysis of credit card fraud detection using logistic regression with random forest towards an increase in accuracy of prediction. In 2022 International Conference on Edge Computing and Applications (ICECAA) (pp. xx-xx). IEEE. https://doi.org/10.1109/ICECAA55415.2022.9936488
11. Nilson Report. (2022). The Nilson Report. Retrieved from https://www.nilsonreport.com
12. Rahman, M. J., & Zhu, H. (2023). Predicting accounting fraud using imbalanced ensemble learning classifiers – evidence from China. Accounting & Finance, 63, 3455-3486. https://doi.org/10.1111/acfi.13044
13. Rahman, M. J., & Zhu, H. (2023). Predicting accounting fraud using imbalanced ensemble learning classifiers – evidence from China. Accounting & Finance, 63(3), 3455–3486. https://doi.org/10.1111/acfi.13044
14. Towards AI Team. (n.d.). How did binary cross-entropy loss come into existence? Towards AI. https://towardsai.net/p/machine-learning/how-did-binary-cross-entropy-loss-come-into-existence
15. Wang, H., Wang, W., Liu, Y., & Alidaee, B. (2022). Integrating Machine Learning Algorithms With Quantum Annealing Solvers for Online Fraud Detection. IEEE Xplore. https://doi.org/10.1109/ACCESS.2022.3185485
16. Xu, B., Wang, Y., Liao, X., & Wang, K. (2023). Efficient fraud detection using deep boosting decision trees. Decision Support Systems, 114037. https://doi.org/10.1016/j.dss.2023.114037

Appendices

- Kaggle Open Access Dataset: https://www.kaggle.com/datasets/mlg-ulb/creditcardfraud
- Public GitHub Repository: https://github.com/msteere3/CCFraudGridSearch
- Top Features Ranked by Collective AUC:

```
# Ordered top 30 features
#SELECTED_FEATURES = ['V17', 'V12', 'V14', 'V10', 'V16', 'V11', 'V9', 'V7', 'V18',
'V4',
#       'V21', 'V3', 'V26', 'V20', 'V2', 'V1', 'V19', 'V6', 'Amount', 'V8',
#       'V15', 'Time', 'V5', 'V22', 'V27', 'V24', 'V28', 'V13', 'V25', 'V23']

# Top 20 features from ordered list above^
SELECTED_FEATURES = ['V17', 'V12', 'V14', 'V10', 'V16', 'V11', 'V9', 'V7', 'V18',
'V4', 'V21', 'V3', 'V26', 'V20', 'V2', 'V1', 'V19', 'V6', 'Amount', 'V8']

#Top 1 features: AUC = 0.8507 (+/- 0.0473)
#Top 2 features: AUC = 0.9042 (+/- 0.0391)
#Top 3 features: AUC = 0.9225 (+/- 0.0244)
#Top 4 features: AUC = 0.9223 (+/- 0.0249)
#Top 5 features: AUC = 0.9262 (+/- 0.0210)
#Top 6 features: AUC = 0.9280 (+/- 0.0218)
#Top 7 features: AUC = 0.9259 (+/- 0.0239)
#Top 8 features: AUC = 0.9319 (+/- 0.0207)
#Top 9 features: AUC = 0.9310 (+/- 0.0224)
#Top 10 features: AUC = 0.9392 (+/- 0.0182)
#Top 11 features: AUC = 0.9442 (+/- 0.0190)
#Top 12 features: AUC = 0.9451 (+/- 0.0202)
#Top 13 features: AUC = 0.9409 (+/- 0.0215)
#Top 14 features: AUC = 0.9410 (+/- 0.0171)
#Top 15 features: AUC = 0.9410 (+/- 0.0139)
#Top 16 features: AUC = 0.9416 (+/- 0.0169)
#Top 17 features: AUC = 0.9437 (+/- 0.0196)
#Top 18 features: AUC = 0.9479 (+/- 0.0230)
#Top 19 features: AUC = 0.9418 (+/- 0.0224)
#Top 20 features: AUC = 0.9488 (+/- 0.0215) # BEST AUC HERE
#Top 21 features: AUC = 0.9478 (+/- 0.0238)
#Top 22 features: AUC = 0.8804 (+/- 0.1091)
#Top 23 features: AUC = 0.8920 (+/- 0.0731)
#Top 24 features: AUC = 0.8985 (+/- 0.0554)
#Top 25 features: AUC = 0.8386 (+/- 0.1882)
```

```
#Top 26 features: AUC = 0.8678 (+/- 0.1347)
#Top 27 features: AUC = 0.9112 (+/- 0.0513)
#Top 28 features: AUC = 0.8763 (+/- 0.1004)
#Top 29 features: AUC = 0.8975 (+/- 0.0788)
#Top 30 features: AUC = 0.8864 (+/- 0.1074)
```

- Randomized Grid Search Code

```
# main.py
import sys
from data_processing import preprocess_data
from svm import run_svm_with_randomized_search_and_timeout
from neuralnetwork import run_ann_with_timeout
from random_forest import rf_with_timeout
import random
import time

def random_parameters(parameters_grid):
   selected_parameters = {}
   for param, values in parameters_grid.items():
       selected_parameters[param] = random.choice(values)
   return selected_parameters

def parse_command_line_arguments():
   file_path = sys.argv[1]
   total_time = sys.argv[2]
   single_iter_time = sys.argv[3]

   return file_path, float(total_time), float(single_iter_time)#, base_algorithm,
scale_rule, SMOTE_rule, enable_selected_features

def train_and_run(file_path, base_algorithm, X_train, X_test, y_train, y_test,
iter_start_time, single_iter_time):
   timeout = single_iter_time - (time.time()-iter_start_time)
   if (base_algorithm=='random_forest'):
       rf_with_timeout(file_path, X_train, X_test, y_train, y_test, timeout)
   elif (base_algorithm=='svm'):
       run_svm_with_randomized_search_and_timeout(X_train, X_test, y_train, y_test,
single_iter_time, iter_start_time)
   elif (base_algorithm=='neural_network'):
       run_ann_with_timeout(X_train, X_test, y_train, y_test, timeout)

def total_grid_search(file_path, total_time, single_iter_time):
   start_time = time.time()
```

```
    parameters_grid = {
        'base_algo': ['random_forest', 'svm', 'neural_network'],
        'scale_rule': ['StandardScaler', 'RobustScaler', 'PowerTransformer', None],
        'SMOTE_rule' : ['SMOTE', 'SMOTE_ENN', None],
        'enable_selected_features': [1, 0]
        # Add more parameters as needed
    }
    while(1):
        iteration_start_time = time.time()
        if(time.time() - start_time > total_time):
            print('Total Grid Search time limit reached.\n')
            return
        selected_parameters = random_parameters(parameters_grid)
        base_algorithm = selected_parameters['base_algo']
        scale_rule = selected_parameters['scale_rule']
        SMOTE_rule = selected_parameters['SMOTE_rule']
        enable_selected_features = selected_parameters['enable_selected_features']

        rf = ''
        if(enable_selected_features):
            rf = ' random forest feature selection,'
        else:
            rf = 'out random forest feature selection,'

        print('Preprocessing data for base_algorithm= ', base_algorithm, ' with', rf,
'scale_rule=', scale_rule, ' SMOTE_rule=', SMOTE_rule, '\n')

        # NOTE: We hardcoded selected_features so that it's preset and saves run-time.
        X_train, X_test, y_train, y_test = preprocess_data(file_path, scale_rule,
SMOTE_rule, enable_selected_features, target='Class')

        train_and_run(file_path, base_algorithm, X_train, X_test, y_train, y_test,
iteration_start_time, single_iter_time)

if __name__ == "__main__":
    file_path, total_time, single_iter_time = parse_command_line_arguments()
    print("Total time: ", total_time, "\niteration time: ", single_iter_time, "\n")
    total_grid_search(file_path, total_time, single_iter_time)
```

- Random Forest Classifier Code

```
def rf_with_timeout(csv_file_path, X_train, X_test, y_train, y_test, timeout):
    def random_forest_classification():
```

```
        print("Running random forest for classification...\n")

        randforest_model =  RandomForestClassifier(random_state=42)

        randforest_model.fit(X_train, y_train)
        y_pred = randforest_model.predict(X_test)
        report = classification_report(y_test, y_pred)
        accuracy = accuracy_score(y_test, y_pred)
        auc_score = roc_auc_score(y_test, y_pred)

        if report is not None:
            print(report)
            print(f"Accuracy: {accuracy}")
            print(f"AUC: {auc_score}")

    with concurrent.futures.ThreadPoolExecutor() as executor:
        future = executor.submit(random_forest_classification)
        try:
            return future.result(timeout=timeout)
        except concurrent.futures.TimeoutError:
            print(f"Training exceeded time limit of {timeout} seconds")
            return None, None, None
```

- Random Forest Feature Selection Code

```
def random_forest_preprocessing_main(self, csv_file_path, num_features_to_select=30,
threshold=0.95):
    randforest_model = RandForest(csv_file_path)
    threshold = random.choice([0.5, 0.7, 0.8, 0.85, 0.9, 0.91, 0.92, 0.93, 0.95, 0.955,
0.96, 0.965, 0.97, 0.975, 0.98, 0.985, 0.99, 0.995])
    # Preprocess data
    randforest_model.preprocess_data()

    # Train the model and print out classification report
    auc_score = randforest_model.train_model()
    print(f"Initial AUC Score: {auc_score}")

    # Evaluate the importance of features based on cross-validated performance
    # trimmed_data, feature_names, num_features =
randforest_model.evaluate_feature_importance(n_features_to_select=num_features_to_sele
ct, threshold=threshold)

    # print(f"{num_features} important features were found: {feature_names}")
```

```
    trimmed_data = self.data[SELECTED_FEATURES]

    # Save the model
   #fraud_detector.save_model('rf_model.pkl')

    return trimmed_data

class RandForest:
    def __init__(self, file_path):
        self.data = pd.read_csv(file_path)
        self.model = RandomForestClassifier(random_state=42)
        self.X = None
        self.y = None
        self.feature_importances = None

    def preprocess_data(self):
        # Assume last column is the label and all others are features
        self.X = self.data.iloc[:, :-1]
        self.y = self.data.iloc[:, -1]
        scaler = StandardScaler()
        self.X = scaler.fit_transform(self.X)

    def train_model(self):
        X_train, X_test, y_train, y_test = train_test_split(self.X, self.y,
test_size=0.2, random_state=42)
        self.model.fit(X_train, y_train)
        self.feature_importances = self.model.feature_importances_
        y_pred = self.model.predict(X_test)
        print(classification_report(y_test, y_pred))
        return roc_auc_score(y_test, y_pred)

    def train_preprocessed(self, X_train, X_test, y_train, y_test):
        self.model.fit(X_train, y_train)
        y_pred = self.model.predict(X_test)
        print(classification_report(y_test, y_pred))
        return roc_auc_score(y_test, y_pred)

    def evaluate_feature_importance(self, n_features_to_select, threshold):
        feature_indices = np.argsort(self.feature_importances)[::-1]
        optimal_num_features = n_features_to_select
        for i in range(1, n_features_to_select + 1):
            selected_features = feature_indices[:i]
```

```
            X_reduced = self.X[:, selected_features]
            scores = cross_val_score(self.model, X_reduced, self.y, cv=5,
scoring='roc_auc')
            mean_score = np.mean(scores)
            std_score = np.std(scores)
            print(f"Top {i} features: AUC = {mean_score:.4f} (+/- {std_score:.4f})")
            if (mean_score > threshold):
                optimal_num_features = i
                break
        final_selected_features = feature_indices[:optimal_num_features]
        class_index = self.data.columns.get_loc('Class')
        final_selected_features_with_class = np.append(final_selected_features,
class_index)



        important_feature_names = self.data.columns[final_selected_features]
        # if 'Class' not in final_selected_features:
        # final_selected_features.append('Class')
        # trimmed_data_final_features = feature_indices[:optimal_num_features +
'Class']
        # trimmed_data = self.data.iloc[:, trimmed_data_final_features]

        trimmed_data = self.data.iloc[:, final_selected_features_with_class]

        return trimmed_data, important_feature_names, optimal_num_features


    def save_model(self, model_file):
        joblib.dump(self.model, model_file)

    def load_model(self, model_file):
        self.model = joblib.load(model_file)

    def make_prediction(self, sample_data):
        scaler = StandardScaler()
        sample_data = scaler.fit_transform(sample_data)
        return self.model.predict(sample_data)
```

- Support Vector Classification Code

```
# svm.py
from sklearn.svm import SVC
from sklearn.metrics import classification_report, accuracy_score, roc_auc_score
from sklearn.model_selection import ParameterSampler
```

```
import numpy as np
import concurrent.futures
import time

def svm_train_with_timeout(X_train, y_train, X_test, y_test, params, timeout):
   """Train and evaluate an SVM model with a timeout."""
   def train_and_evaluate():
       model = SVC(probability=True, **params)
       model.fit(X_train, y_train)
       predictions = model.predict(X_test)
       auc_score = roc_auc_score(y_test, model.predict_proba(X_test)[:, 1])
       return classification_report(y_test, predictions), accuracy_score(y_test,
predictions), auc_score

   with concurrent.futures.ThreadPoolExecutor() as executor:
       future = executor.submit(train_and_evaluate)
       try:
           return future.result(timeout=timeout)
       except concurrent.futures.TimeoutError:
           print(f"Training exceeded time limit of {timeout} seconds for parameters:
{params}")
           return None, None, None

def run_svm_with_randomized_search_and_timeout(X_train, X_test, y_train, y_test,
search_time_limit=600, iter_start_time=0):
   param_distributions = {
   'C': np.logspace(-3, 2, 6),  # Generates values [0.001, 0.01, 0.1, 1, 10, 100]
   'gamma': [0.01, 0.1], # np.logspace(-3, -1, 3),  # Generates values [0.001, 0.01,
0.1]
   'kernel': ['rbf', 'linear', 'poly', 'sigmoid']  # Kernel types
}
   param_sampler = ParameterSampler(param_distributions, n_iter=1,
random_state=int(time.time()))

   for params in param_sampler:
       print(f"running SVM with parameters: {params}")
       timeout_arg = search_time_limit- (time.time() - iter_start_time)
       report, accuracy, auc = svm_train_with_timeout(X_train, y_train, X_test,
y_test, params, timeout=timeout_arg)
       if report is not None:
           print(report)
           print(f"Accuracy: {accuracy}")
```

```
            print(f"AUC: {auc}")
        print("--------")
```

- Keras Sequential Neural Network Classification Code

```
# neuralnetwork.py
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Input
from sklearn.metrics import classification_report, confusion_matrix, recall_score,
roc_auc_score
from tensorflow.keras.initializers import HeNormal
import random
import concurrent.futures

def random_parameters(parameters_grid):
   selected_parameters = {}
   for param, values in parameters_grid.items():
       selected_parameters[param] = random.choice(values)
   return selected_parameters

def run_ann_with_timeout(X_train, X_test, y_train, y_test, time_limit):

   def run_ann():
       parameters_grid = {
       'kernel': ['HeNormal', 'uniform', 'normal'],
       'optimizer': ['adam', 'SGD', 'adagrad', 'adamax', 'nadam'],
       'batch_size' : [8, 16, 32, 64, 128, 256, 512, 1024, 2048],
       'epochs' : [5, 10, 15, 20],
       'loss' : ['binary_crossentropy']
       # Add more parameters as needed
       }
       selected_parameters = random_parameters(parameters_grid)

       kern_init = selected_parameters['kernel']
       sel_optimizer = selected_parameters['optimizer']
       sel_batchsize = selected_parameters['batch_size']
       sel_epochs = selected_parameters['epochs']
       sel_loss = selected_parameters['loss']

       # Defining the model architecture
       if (kern_init == 'HeNormal'):
           classifier = Sequential([
           Input(shape=(X_train.shape[1],)),
           Dense(units=64, kernel_initializer=HeNormal(), activation='relu'),
```

```
            Dense(units=32, kernel_initializer=HeNormal(), activation='relu'),
            Dense(units=32, kernel_initializer=HeNormal(), activation='relu'),
            Dense(units=16, kernel_initializer=HeNormal(), activation='relu'),
            Dense(units=8, kernel_initializer=HeNormal(), activation='relu'),
            Dense(units=1, kernel_initializer=HeNormal(), activation='sigmoid')
        ])
        else: #use kern_init as value 'uniform' or 'normal'
            classifier = Sequential([
            Input(shape=(X_train.shape[1],)),
            Dense(units=64, kernel_initializer=kern_init, activation='relu'),
            Dense(units=32, kernel_initializer=kern_init, activation='relu'),
            Dense(units=32, kernel_initializer=kern_init, activation='relu'),
            Dense(units=16, kernel_initializer=kern_init, activation='relu'),
            Dense(units=8, kernel_initializer=kern_init, activation='relu'),
            Dense(units=1, kernel_initializer=kern_init, activation='sigmoid')
        ])

        print("Running neural network with parameters: ", selected_parameters,
"\n...\n")

        classifier.compile(optimizer=sel_optimizer, loss=sel_loss,
metrics=['accuracy'])

        # Training the model
        classifier.fit(X_train, y_train, batch_size=sel_batchsize, epochs=sel_epochs,
verbose=1)

        # Evaluating the model
        scores = classifier.evaluate(X_test, y_test)
        print("\nModel Accuracy: %.2f%%" % (scores[1]*100))

        # Predicting the test set results
        y_pred = classifier.predict(X_test)
        y_pred_classes = (y_pred > 0.5).astype(int)  # Converting probabilities to
class labels

        # Generating confusion matrix and classification report
        cm = confusion_matrix(y_test, y_pred_classes)
        print("Confusion Matrix:")
        print(cm)

        print("Classification Report:")
```

```
        print(classification_report(y_test, y_pred_classes,
target_names=['Non-Fraudulent', 'Fraudulent']))

        # Calculating recall for fraudulent transactions
        print(f"Recall for Fraudulent Transactions: {recall_score(y_test,
y_pred_classes):.2f}")

        # Calculating and printing the AUC
        auc = roc_auc_score(y_test, y_pred.ravel())  # Here y_pred is used directly to
calculate AUC
        print(f"AUC: {auc:.2f}")

    with concurrent.futures.ThreadPoolExecutor() as executor:
        future = executor.submit(run_ann)
        try:
            return future.result(timeout=time_limit)
        except concurrent.futures.TimeoutError:
            print(f"Training exceeded time limit of {time_limit} seconds")
            return None, None, None
```