Alternative Credit Scoring for SMEs Using Transaction Forecasting

Matthew Newman

September 2025

School of Mathematics,
Cardiff University

A dissertation submitted in partial fulfilment of the requirements for **MSc Data Science and Analytics** by taught programme, supervised by **Liam Turner**.

Executive Summary

Credit scoring has become prolific within the financial sector and has become critical in allowing businesses to access finances. However, many small-to-medium enterprises lack credit history, meaning that traditional approaches assessing credit history and demographic variables frequently lock them out of the system due to no credit history frequently being equated with poor credit history when this is not necessarily the case.

To address this, alternative data sources and modelling techniques have increasingly been explored. Many existing approaches have utilised transaction and cash flow data, and one approach successfully applied transaction forecasting to behavioural scoring for personal consumers. However, forecasting has been untested in the application scoring domain for small-to-medium enterprises and has not been tested in tandem with other data. Therefore, this study tested and compared numerous forecasting methods to find the best forecasting approach and examine whether transaction forecasting could improve credit scoring accuracy.

Both cumulative and month-by-month transaction forecasting were tested, and in both cases, the more complex forecasting approaches often struggled, and a simpler baseline forecast frequently came out on top. This is thought to be because small-to-medium enterprises frequently have volatile finances and greater exposure to economic shocks, alongside relatively little prior financial history. This will make any extrapolated trends unreliable, meaning that the more complex methods may struggle to make accurate predictions. Month-by-month forecasting was found to be less accurate than cumulative forecasting, likely due to volatility being captured instead of overall trends. Further examination of the results unearthed recent volatility as a significant factor in poor model performance, with poor performers generally having greater volatility in their recent transaction patterns.

When forecasting data was integrated into a credit scoring classification model, it was found that neither cumulative nor month-by-month forecasting data meaningfully improved credit scoring efficacy compared to traditional variables, even when forecasting data was used in tandem with traditional variables. This is thought to be

due to volatility and limited transaction history increasing the difficulty of accurately forecasting small-to-medium enterprise transactions. Month-by-month forecasting was found to offer slightly more predictive power than cumulative, which is likely due to month-by-month forecasting offering a clearer view of volatility as opposed to only offering a net trend, but neither outperformed traditional variables. It should also be noted that the synthetic data and links, used due to data availability constraints, were a limitation, with no classifier attaining overly strong results. This would imply that real-world data cannot truly be replaced.

Resultantly, it is recommended that real-world data is used to test machine learning approaches in credit scoring for the most accurate insights. While the linking assumptions made were informed, assumptions are inherently blunt and cannot capture edge cases, so real-world data with known links is arguably irreplaceable. Also, exploration of volatility forecasting is recommended due to the results unearthing the importance of volatility in small-to-medium enterprise credit risk. Forecasting of raw values has limitations in the small-to-medium enterprise space, particularly where recent volatility is present, but volatility is itself a highly influential factor in small-to-medium enterprise credit risk, so the use of volatility forecasting may unearth interesting insights.

Acknowledgements

I would firstly like to extend my sincerest thanks to my academic supervisor, Dr Liam Turner. I could not have completed the project without you, and your support and advice has been invaluable throughout the process.

I would also like to thank my corporate supervisor Daniel Goldney, as well as Jeremy Bowles and the rest of the team at Menna. Working alongside you has been a fantastic experience, and your support and advice has been invaluable throughout the project.

Furthermore, I would like to thank my MSc course colleague Edward Watson. Our biweekly meetups in the Abacws building made the dissertation period so much more interesting and enjoyable, and talking to you gave me some much-needed inspiration at points.

As ever, I would finally like to thank my parents. Their support from the sidelines has kept me going throughout this process and the entire MSc programme.

Contents

Figures

Tables

1 Introduction

Credit scoring is an important branch of data analytics. While relatively young, it has become prolific within finance (Abdou and Pointon 2011). Traditionally, approaches such as logistic regression (LR) have been used to classify default risk based on data such as demographic information and credit history (Li and Zhong 2012).

However, traditional credit scoring approaches impede those with limited credit history. VantageScore (2023) accuse "outdated and unnecessarily restrictive" traditional approaches of preventing those with limited credit history from accessing credit. Limited credit history is not inherently indicative of poor financial acumen (Brevoort et al. 2016), yet traditional approaches equate no credit history with poor credit history regardless of other factors (Avery et al. 2004).

This poses issues for small-to-medium enterprises (SMEs). While it is estimated that the advent of credit scoring increased SME lending (Frame and Woosley 2004), insufficient credit history remains the most common reason for SME credit rejection (Federal Reserve Banks 2017), suggesting that traditional approaches suit SMEs lacking credit history poorly. With many SMEs resultantly accessing insufficient credit (Bams et al. 2021), this is a considerable issue that impedes both SMEs and lenders. SMEs lack access to credit, while lenders bypass potential good customers.

In a world of big data, alternative data sources could arguably be harnessed for credit scoring where credit history is limited, as is often the case with SMEs. Djeundje et al. (2021) found, for example, that psychometric data could enhance credit scoring efficacy, and sources such as social network and shopping data have also been explored (Hurley and Adebayo 2016). The potential for complex relationships within alternative data exists and merits investigation.

This thesis proposes an alternative SME credit scoring approach utilising transaction-based cash flow prediction. While works exist that utilise cash flow and transaction variables, these simply utilise past data rather than predicting the future. Kotios et al. (2022) found that transaction patterns can be mapped to SME cash flow, and that cash flow is an important financial fundamental. While Ala'raj et al. (2022) successfully

used Long Short-Term Memory (LSTM) neural networks to predict personal transactions in behavioural scoring, this approach has never been applied to SME transactions or application scoring. Furthermore, this study proposes testing different forecasting approaches. Neural networks (NNs) can require plentiful training data (Zhang 2007), and SMEs may lack transaction history to draw upon.

1.1 Research Questions

1. What data is currently used in credit scoring, including for SMEs?
2. What modelling techniques are currently used in credit scoring, including for SMEs?
3. Can other time series forecasting approaches forecast transactions more effectively than LSTMs?
4. Can transaction forecasting improve credit scoring efficacy compared to traditional methods?

1.2 Research Objectives

- Investigate existing methods of credit scoring and existing data used in SMEs.
- Develop a credit scoring model that utilises transaction forecasting.
- Evaluate the model against existing techniques using synthetic data.

1.3 Scope

This study covers SME credit scoring, the use of alternative data sources in credit scoring and the use of alternative modelling techniques in credit scoring. While adjacent fields such as fraud detection may use some similar techniques, these fields are not to be covered here.

2 Literature Review

To satisfy RQ1, RQ2 and RO1, a literature review was conducted. Four key areas were reviewed: SME credit scoring, alternative data in credit scoring, credit scoring with multiple data sources, and alternative modelling approaches in credit scoring.

2.1 SME Credit Scoring

SME lending is risky for credit lenders due to SMEs often having less available financial information and greater instability against economic challenges (Tsaih et al. 2004). Therefore, credit scoring is imperative to ensure that both SMEs and lenders can benefit from loans. It is estimated that the advent of credit scoring increased SME lending (Frame and Woosley 2004), so it has overall benefitted SMEs.

Wang et al. (2020) proposed an XGBoost approach to SME credit scoring, with Shapley scores integrated for explainability. The approach used financial transaction data, with variables representing aspects such as account balance patterns, spending patterns and cash flow. When deployed at QuickBooks Capital, the method reduced loan default rates by 20%, and it was also found that the Kolmogorov-Smirnov (KS) statistic of the model was 7% higher than that of a traditional scorecard. While the approach was successful, the transaction variables were largely derived from prior data as opposed to making a prediction of future spending patterns. The use of time series forecasting to predict transactions could have been employed to predict the future picture rather than relying on the past.

Bitetto et al. (2024) employed a random forest (RF) approach and tested its efficacy against a traditional LR approach. Invoice and financial statement variables were used, with financial variables including current liabilities and turnover and invoice variables including credit history metrics such as delinquency. It was found that the RF improved upon the traditional LR approach, and that financial statement variables were more effective than invoice variables. While the approach proved effective, it did not utilise any alternative data sources. Using sources such as banking transactions may have enhanced the approach.

Li et al. (2021) tested a NN-based approach for SME credit scoring. The NN was compared to two LR approaches, and qualitative information was used alongside

traditional quantitative financial information. The NN improved upon the LR scorecards, but while the qualitative information was found to increase credit scoring efficacy, it was not effective on its own, with quantitative information proving irreplaceable. The approach could have been improved by integrating alternative financial data sources, as the authors noted themselves.

Abedin et al. (2023) tested a RF, combined with weighted SMOTE oversampling to combat class imbalance. A mix of financial variables, non-financial variables and macroeconomic variables were employed, with financial variables including debt-to-asset ratio and liquidity ratio, non-financial variables including operation period and audit status, and macroeconomic variables including GDP growth rate and inflation. The approach was effective, yielding a 99.16% true positive rate, and it also improved on alternatives such as a support vector machine (SVM) and a K-nearest neighbours (KNN) classifier. Despite the approach's efficacy, however, the authors noted the dataset's limited scope, and recommended using a larger, more complex dataset.

Gu et al. (2024) tested an XGBoost approach in tandem with various data resampling methods. The approach utilised a combination of financial data, enterprise behaviour data, public credit data and third-party information. The approach was more effective than various alternatives, obtaining a 96-100% true positive rate and a 94-100% true negative rate. SMOTE was the most effective resampling approach, obtaining a 99% true positive rate. Despite the approach's efficacy, however, it only considered 20 different variables. While numerous data sources were utilised, the range of variables considered could have been wider, with data sources such as transactions being untapped.

Ahmed (2019) tested a naïve-bayes classifier (NBC), a decision tree (DT) and a RF for SME credit scoring. Various financial variables were used, including revenue and age of business. The RF and DT both surpassed 99% classification accuracy, and loan interest rate was deemed influential in predicting default, with higher interest rates resulting in higher chances of default. The approach was effective, but it did not consider any alternative data sources. Other data sources such as transactions could have offered a more holistic view.

Yan and Li (2023) proposed a hybrid approach that fused a DT classifier and a linear programming (LP) model. The DT was used to classify loans by risk level, and the LP model was used to generate an optimal loan amount, interest rate and customer churn rate for each category. Financial variables such as profitability, average amount per sale and number of units were used as the underlying data, and the approach was effective, even where credit history was limited. However, the approach was only trained on businesses with credit history, so the inclusion of businesses with little or no credit history in training may have enhanced efficacy further.

Gulsoy and Kulluk (2019) proposed an objective method of assessing SME credit worthiness. 12 different variables, encompassing both subjective traits such as morality and partnership structure and objective financial traits such as liquidity and profitability were used. A Multi Objective Evolutionary Fuzzy Classifier (MOEFC) and various DT types were tested for modelling. Tests found that the MOEFC was most accurate, attaining 78% accuracy, and that it only required 4 fuzzy rules. While the approach was successful, subjective metrics may be harder to empirically assess, as subjective category definitions may not be universally agreed (Annett 2002). Resultantly, more concrete rules for subjective attribute definitions could have been introduced.

Lee and Sohn (2017) proposed a LR scorecard utilising accounting ethics variables. The model utilised variables relating to internal control structure, financial transparency, related parties, business ethics and “other unethical conduct”. The ethics-based approach achieved 70.59% accuracy, beating the 67.65% accuracy attained by a traditional financial-based model, and it also boasted increased sensitivity at detecting default risk. However, despite the approach’s efficacy, ethics variable values may lack universally agreed definitions, so the introduction of more concrete rules for definitions of these subjective attributes may have improved the approach.

Karimova (2024) proposed a RF approach to SME credit scoring. Variables such as revenue growth, cash flow variance and debt-to-equity ratio were used. It was found that the RF boasted improved accuracy, precision, recall and F1 score compared to a

traditional LR scorecard, but it should be noted that model explainability was not researched.

Roy and Shaw (2021) proposed a multicriteria decision making (MCDM) model using the Best Worst Method and TOPSIS. A Likert scale was used for each criterion to formulate the credit score, and a combination of subjective and financial variables was used. The approach was successful, with the scores generated by the MCDM model having a correlation coefficient of 0.95 with the credit scores generated by industry professionals. However, the approach could have been enhanced by using alternative data sources, such as transactions or social media data.

2.2 Alternative Data in Credit Scoring

The financial data sources traditionally used in credit scoring assume that the customer has assessable credit history (Djeundje et al. 2021). However, many SMEs lack credit history (Bams et al. 2021), so traditional methods suit them poorly. Therefore, alternative data sources are important to enhance credit access for SMEs.

Djeundje et al. (2021) proposed an approach utilising psychometric and email usage data. Demographic variables were used alongside psychometric variables such as "choice between smaller amount now or larger amount in 3-6 months" and email usage variables such as "time taken to send last 2,000 emails". The approach enhanced efficacy when used in conjunction with traditional variables and was also effective when employed in isolation. However, this approach raises privacy concerns. GDPR dictates that irrelevant personal data should not be used (Li et al. 2019), so people may not be receptive to lenders accessing their email data.

Verkade (2018) proposed using transaction data for assessing credit worthiness. Variables including positive income shocks, balance returns, and relative expenditure were used, and these were tested both in conjunction with traditional data and in isolation. A LR model combining traditional and transaction history data was developed, and an AUC of 0.850 was attained. When tested with transactional data alone, an AUC of 0.804 was attained, but precision and recall were poor, indicating that transactional data alone may be insufficient. While the approach was successful when the variables were combined, it did not consider alternatives to LR. LR makes

restrictive assumptions, such as linearity of relationships between predictor and target (Stoltzfus 2011), that non-traditional data sources may violate. The consideration of alternatives may have uncovered more complex relationships within the transaction data.

Lohokare et al. (2017) proposed a NN-based approach, underpinned by financial transaction and social media data, to determine credit worthiness. Bank transaction data was collected from SMS messages detailing them, collecting information such as transaction amount and remaining balance. Social media data such as number of followers, sentiment of content and educational details was also collected to create comprehensive customer profiles. An artificial neural network (ANN) was used for modelling, and when employed on a small dataset, the approach attained 72.1% classification accuracy. While the approach seems promising, it should be tested on a larger dataset to confirm efficacy. It also relies purely on past data and does not aim to predict the future.

Dushimimana et al. (2020) proposed utilising a blend of loan details and customer phone behaviour variables to credit score for airtime loans. Three models were tested: a LR model, a DT classifier and a RF classifier. All three models boasted high accuracy but attained weak specificity. While the approach showed potential, the use of additional alternative data sources, including alternative financial data, could have improved specificity.

Guo et al. (2016) proposed a gradient-boosted decision tree (GBDT) approach to credit scoring, underpinned by social media data. Demographic features, tweet features and network features were derived from Weibo and used as predictive variables. A two-tier modelling framework for personal credit scoring was also assessed, which utilised heterogeneous data extraction, low level feature extraction, high level feature extraction and ensemble learning prediction using the GBDT. The framework worked effectively, with the GBDT attaining an Area Under the Curve (AUC) of 0.6251, but only social media data was used. Utilising other data sources may have enhanced efficacy.

Ots et al. (2020) proposed using mobile phone data to assess credit worthiness. Data on calls, SMS messages, images and number of contacts was used, and five modelling approaches were tested. These were LR, RF, DT, SVM, and NN. It was found that the RF was most effective, attaining the highest AUC, and that the use of mobile phone data outperformed a random study, inferring its predictive power. While the approach was effective, financial data could have enhanced the approach further, with credit scoring primarily focusing on user finances.

Chen and Chen (2022) proposed using public sentiment regarding corporations to determine credit worthiness. Numerous corporations had sentiment regarding their latest financial results scraped from various sources, including social media and forums. KNN classification was used for modelling, and 86% accuracy was attained, which improved upon regular financial variables. While the approach was successful, it could have also combined sentiment variables with financial variables to see if pairing the two can improve performance further.

Masyutin (2015) proposed using social network data for determining credit worthiness. Data such as subscriptions, communities, friends, and wall posts was used alongside demographic data such as age and sex. LR was used for modelling, and it was found that while moderate predictive power was attained, the Gini coefficient decreased significantly when age and sex were removed, indicating that social network data alone may lack predictive power. Therefore, social network data could have been combined with traditional financial variables to see if performance improved by pairing the two.

2.3 Credit Scoring with Multiple Data Sources

While credit scoring has traditionally relied on traditional financial data sources, alternative data is becoming increasingly popular. With alternative data around loan applicants being plentiful, there is great potential in using it to score applicants. For SMEs and applicants lacking credit history, it would be prudent to use as many sources as possible to enhance scoring efficacy (Gao and Lau 2024).

Shu et al. (2024) proposed extracting risk factors from loan documents using NLP for determining credit worthiness. Various loan application types were used, from SME

loans to corporate loans, and risk factors included cash flow volatility, supply chain disruption and accounting charges. For modelling, a hybrid pipeline was used to preprocess loan documents and pass them through a transformer-CNN hybrid to interpret the temporal progression of risk factors. SHAP scores were also used to make the approach explainable, and results showed that the approach predicted default effectively. Despite its efficacy, however, the approach could have integrated other sources, such as social media sentiment. Third-party sentiment may provide a more refined viewpoint than first-person sentiment (To and Panadero 2019), so its use may have enhanced efficacy.

Putra et al. (2020) proposed combining social media data with traditional financial data to assess credit worthiness. A mixture of general company information, financial data from the last 5 years and social media data regarding posts and engagement was used. LR and XGBoost were employed for modelling, and it was found that while social media data was not effective in isolation, it did improve accuracy when combined with financial data compared to using financial data alone. While the approach was affective, forecasting future statistics could have been tested to better understand future trends.

Yu et al. (2023) proposed combining ChatGPT with a LightGBM model to assess credit worthiness when both textual and structured data are available. Textual data was taken from the applicant and fed to ChatGPT, which gave scores for each big 5 personality trait. These scores were then fed to the LightGBM model alongside structured data to assess credit worthiness. The use of ChatGPT alone was found to be significantly weaker than regular methods, but its use within an integrated framework alongside the LightGBM model improved results. While the integrated framework was effective, it should be noted that Krumm et al. (2024) found that ChatGPT was a potentially unreliable judge of human behaviour. Therefore, its psychological judgements may be questionable.

Fathy (2024) proposed a unified framework for credit scoring in retail banking. The dataset consisted of structured, semi-structured and unstructured attributes, and a hybrid approach unifying Bayesian inference with spatio-temporal attention

mechanisms was used for modelling. As stated by the author, additional digital channels could have been employed.

Kriebel and Stitz (2022) proposed deriving credit variables from user-generated text to determine credit worthiness. Data from Lending Club combining financial information with a written justification of loan necessity from the applicant was used. Six deep learning (DL) approaches were tested: a CNN, a recurrent neural network (RNN), a convolutional recurrent neural network (CRNN), an average embedding neural network (AENN), BERT and RoBERTa. It was found that RoBERTa had the strongest performance when purely text-based data was utilised, but other networks were stronger when structured data or a combination of the two was used. The strongest results were attained when both were combined. The approach was effective, but it could have been enhanced by integrating third-party sentiment alongside, or instead of, first-person sentiment. Third-party sentiment may provide a more accurate view (To and Panadero 2019).

Óskarsdóttir et al. (2019) proposed using telecommunications data alongside traditional financial data. The telecommunications data consisted of over 90 million phone numbers and was used to derive social networks, while the financial data consisted of both demographic information and credit information. LR, DT and RF were tested for modelling, and it was found that RF was the most effective approach. It was also found that there were fewer social connections between defaulters and non-defaulters, and that social network data could match or outperform financial data in default prediction efficacy. Furthermore, it was found that default can propagate among communities. However, the models combining both social and financial data performed better than models using either alone. Despite the approach's efficacy, it social influence can arguably only go so far. People's private stances and actions are only influenced by social networks to a certain extent (Christoff et al. 2016), so with a private matter like finances, the effect of peers is potentially debatable.

Nguyen et al. (2021) proposed a multimodal machine learning (ML) approach utilising text data from SEC filings alongside tabular data. The tabular data contains 25 financial variables including liquidity and profitability ratios, while the text data was converted into 11 sentiment-based attributes, including positivity, negativity and risk.

Four modelling approaches were tested: KNN, RF, boosted DT and NN. Stack ensembling and repeated k-fold bagging were also tested. Tests found that the inclusion of language data alongside traditional financial data enhanced accuracy, and the ensemble also frequently attained the highest accuracy. While the approach was successful, it should be noted that no effort was made to make it explainable.

Gao and Lau (2024) proposed a DL-based credit scoring model utilising multimodal signals from social media alongside traditional financial data. The traditional data consisted of variables such as mortgage amount and loan amount, while the social media data consisted of variables such as number of words per post and relative frequency of certain topics. EfficientNet-B7 and Latent Dirichlet Allocation were used to process the social media data, while LR, DT, NBC, RF and AdaBoost were used for scoring. The inclusion of social media data was found to boost accuracy by ~20%, and AdaBoost was the most effective of the scoring approaches. While the approach was successful, additional social networks could have been scraped aside from Facebook. Using more social networks could provide a more holistic view.

Lu et al. (2025) proposed a multimodal deep belief network (DBN) for corporate credit risk prediction. Numerical and categorical data from two corporate credit datasets was combined with text data from SEC filings. A DBN combined with a ResNet layer for numerical feature extraction, an embedding layer for categorical feature extraction and a BERT layer for textual feature extraction was used for modelling. The approach outperformed various traditional alternatives, attaining 88-97% accuracy, and it was also found that combining all three modalities returned the best results. While the approach was successful, it could have employed third-party sentiment alongside, or instead of, first-person sentiment for a more holistic view.

Sawhney et al. (2020) proposed a multimodal financial risk forecasting model integrating text and audio data. A dataset of conference calls, containing sentence sequences matched with corresponding audio clips, was used alongside historical stock price data. For modelling, a text-audio aligned encoder, an ensemble method combining a SVM for traditional financial metrics with a biLSTM for text and audio handling, was used. Tests found that the use of multiple modalities significantly improved performance compared to using financial data alone, but that gains

diminished the longer the prediction period became. The approach was successful, but other data sources such as transaction data could also have been used.

Huang et al. (2024) proposed a suite of financial language models that aimed to address the limitations of LLMs within finance. Traditional financial data taken from the German credit dataset, a Lending Club dataset and an Australian credit dataset was used for credit scoring, and the LLM was trained on a wide corpus of financial texts. The LLM LLaMA was specialised into Fin-LLaMA, which handled financial use cases including credit scoring and fraud detection. It was found that Fin-LLaMA performed inconsistently at credit scoring, with some datasets performing better than others. The approach could have utilised other financial data sources, such as transaction data, for stronger credit scoring performance.

El-Qadi et al. (2023) proposed using feature extraction from text data and merging with financial data for predicting default risk within one year. The dataset contained 34 variables, including company information, financial information and comments from a risk analyst. Word embedding and sentiment analysis were used to manipulate the text data using FinancialBERT (a finance-adapted BERT model), and LR, RF, SVM, XGBoost, LightGBM and a NN were used to score. The use of textual data improved scoring efficacy, and sentiment analysis generally outperformed word embedding. While the approach was successful, it could have used third-party sentiment for a more holistic view.

2.4 Alternative Modelling Approaches in Credit Scoring

While LR is commonly used for modelling in credit scoring, it struggles with non-linear and codependent variables (Hooman et al. 2016). Linear inter-variable relationships are not guaranteed, so exploring alternative methods is worthwhile to ensure that relationships are fully exploited. Methods such as NNs can deal with complex relationships (Livingstone et al. 1997).

Feng et al. (2022) proposed a clustering approach to personal credit scoring. Gender, income, age, historical credit limit and historical default number were used as variables. K-means clustering was used to group the consumers, and a DT was used to classify each user group. The clustering approach improved performance

significantly, increasing classifier accuracy by 11% compared to a standalone model. However, the approach used very few variables and data sources, and considerably more could have been tested.

Ala'raj et al. (2022) proposed a NN approach with LSTM layers to model personal finances for behavioural scoring. Information on user transactions was used to classify default risk and predict number of transactions. LSTM layers were employed to learn about and predict user transaction patterns over time. The default prediction model attained 84-96% accuracy, substantially improving on other methods such as a random forest or SVM, and the transaction forecasting model was also effective. While the approach was successful, it could have combined transaction data with other variables. Other approaches reported optimal results when merging multiple data sources.

Luo et al. (2017) proposed an approach involving a DBN. Data on credit default swaps and business recovery rate after an emergency was used. The DBN was tested alongside LR, a SVM and a multi-layer perceptron (MLP), and it was found to be the most accurate method. While the approach was successful, not all businesses will have encountered an emergency, so the use of recovery rate after an emergency may not be universally applicable.

Gicić et al. (2023) proposed the use of LSTMs with multi-variable data to predict credit worthiness. The approach used traditional credit variables from the German credit dataset, and LSTMs were used with stationary multi-variable data in this approach. A stacked LSTM was tested alongside a stacked bi-directional LSTM, and the two models attained 93-98% accuracy, outperforming other methods. The approach was effective, but it could have used other data sources beyond traditional credit data.

Zhu et al. (2018) proposed a hybrid approach encompassing a CNN and the feature selection algorithm Relief. Variables such as daily payments, credit card payments and bank account balance were used, and Relief was used to select the 50 most important features before transforming and classifying records using the CNN. The approach attained superior accuracy to benchmark methods like LR and SVM, but no attempt was made to forecast future variable values.

Trivedi (2020) tested various feature selection and ML approaches for credit scoring. Traditional credit data from the German credit dataset was used, and a Bayesian classifier, NBC, DT, RF and SVM were tested. Information gain, gain ratio and chi-square were also tested for feature selection. The RF with chi-squared feature selection yielded the highest accuracy. However, no alternative data sources were tested, and no DL-based approaches were tested either.

Muhammad et al. (2024) compared the efficacy of six different ML approaches for credit scoring. Traditional credit data such as age, gender, marital status and loan delinquency variables were used, and LR, a NN, a SVM, a DT, a KNN classifier and linear discriminant analysis were tested. Tests found that while all techniques were effective, the NN and KNN classifier were particularly effective. However, no alternative data sources aside from traditional credit data were tested.

Talaat et al. (2024) proposed a DL approach encompassing explainable AI techniques. The dataset used 25 variables, including demographic information, credit limits and payment history. Recursive feature elimination and LASSO were used for feature selection, while a NN was used for prediction and SHAP scores were used for explainability. The approach attained 84% accuracy, 88% sensitivity and 98% specificity. While the approach was effective, more sophisticated NN architectures could have been tested.

De Lange et al. (2022) proposed an explainable AI approach for credit assessment. Two different data subsets were used. One contained behavioural and application assessment from the customer's application, while the other contained account records. A LightGBM model with SHAP scores was used and tested alongside a LR model. Tests found that the LightGBM model beat the LR model in terms of performance. While the approach was successful, it could have forecasted future variable values instead of relying purely on past values.

Wang and Xiao (2022) proposed a feature embedded transformer for credit scoring. The dataset used contained financial feature data alongside social media behavioural data. A transformer was used to interpret behavioural data and convert it into variables,

and this was merged with the financial data to make a prediction. The transformer outperformed various benchmark methods including LR and XGBoost, although no other alternative data sources were explored beyond social media data.

2.5 Literature Gap

One notable literature gap is that few works forecast future transactions in SME application scoring. While numerous works use transaction and cash flow variables, these are simply past values rather than a forecast of the future. Ala'raj et al. (2022) used an LSTM for transaction forecasting to inform a behavioural scoring model in personal credit scoring, but the approach is untested within application scoring and in SME credit scoring. It has also not been combined with other variables. This approach only tested the forecast, and other approaches found that combining multiple data sources improved efficacy.

Also, no other time series forecasting approaches aside from LSTM have been tested. While LSTMs are powerful, NNs can require a lot of training data (Zhang 2007), and many SMEs may lack sufficient prior data to make an accurate prediction. Therefore, other approaches merit exploration.

3 Methodology

3.1 Dataset

3.1.1 Description

Three synthetic datasets, sourced from the FCA Sandbox, were used in this study. The datasets related to SME current accounts (NayaOne's Digital Twin 2025), SME loans (NayaOne 2025b) and SME transaction data (NayaOne 2025a). Real-world data could not be sourced due to privacy concerns.

3.1.1.1 SME Current Accounts Dataset

The SME current accounts dataset contained data for 50,000 accounts. The dataset variables used are shown in Table 1.

Variable	Data Type	Use	Description
Primary Sector	Categorical	Predictor	The primary sector of operation.
Company Type	Categorical	Predictor	The company type the business operates as.
Incorporation Date	Date	Manipulated to provide "years in business" value for one assumption set.	The date the business was established on.
Revenue	Numerical	Predictor	The amount of revenue the business made in the prior year.
Costs	Numerical	Predictor	The business' costs in the prior year.
Accounts Receivable	Numerical	Predictor	The anticipated upcoming amount from unpaid delivered services.

Capital and Reserves	Numerical	Predictor	The composite of the business' profit and share capital minus shareholder dividends.
Current Assets	Numerical	Predictor	The total asset value that can be converted to cash within 12 months.
Current Liabilities	Numerical	Predictor	The total debt to pay off within 12 months.
Fixed Assets	Numerical	Predictor	The total long-term asset value.
Long-Term Liabilities	Numerical	Predictor	The total value of debts that are expected to be paid off in more than 12 months.
Provisions for Liabilities	Numerical	Predictor	The amount of money set aside to weather unexpected shocks.

Table 1: A table showcasing the used variables from the SME current accounts dataset

3.1.1.2 SME Transactions Dataset

The transactions dataset contained 99,960 transactions from 79 accounts. The dataset variables used are shown in Table 2.

Variable	Data Type	Description
TransactionDate	Date	The date on which the transaction occurred. Originally a date and timestamp, but was converted to a month value for grouping.
TransactionType	Categorical	The type of transaction that occurred. Can be "Credit" or "Debit".
Amount	Numerical	The amount transferred during the transaction, in pounds. For any "Debit" transactions, this value is converted to a negative.
AccountId	Categorical	An identifier column denoting the account the transaction occurred on.

Table 2: A table showcasing the used variables from the SME transactions dataset

The distribution of transaction numbers per account is shown in Figure 1.

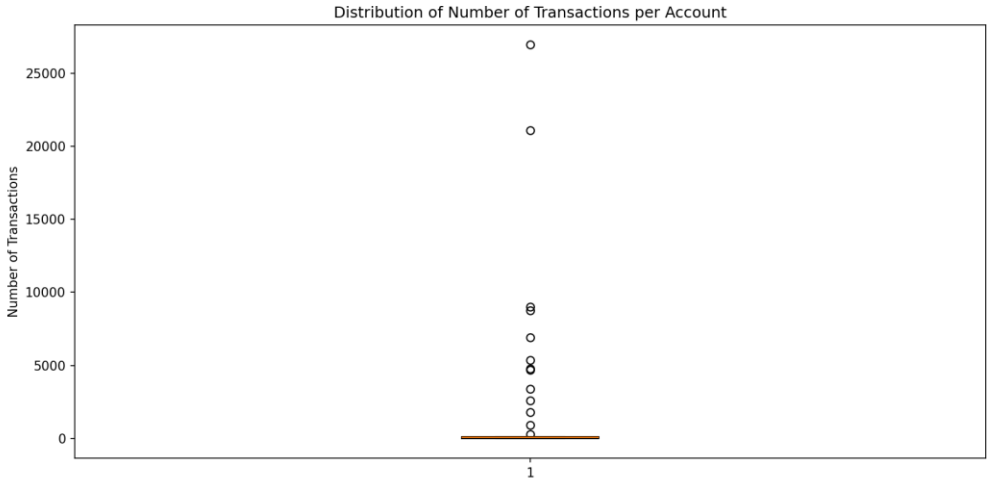

Figure 1: A boxplot showing the distribution of transaction amounts per account

Most accounts have a small number of transactions, but a select few have far more, with the highest number for one account being over 25,000.The distribution of number of months per account is shown in Figure 2.

Figure 2: A boxplot showing the distribution of number of months per account

Most accounts have between 9 and 15 months of data, with the median being 14, but some have as little as 1. However, this does not consider the imputation of empty months in between, which provided more months in some cases.

3.1.1.3 SME Loans Dataset

The SME loans dataset contained data on 3,000 loans. After filtering to completed loans that either repaid or defaulted, this fell to 554. The dataset variables used are shown in Table 3.

Variable	**Data Type**	**Use**	**Description**
Loan Amount	Numerical	Predictor	The loan amount being requested.
Years of Credit History	Numerical	Predictor	The number of years of credit history the business has.
Total Credit Lines	Numerical	Predictor	The total number of credit lines the business has had.

Open Credit Lines	Numerical	Predictor	The number of open credit lines the business has.
Bankruptcies	Numerical	Predictor	The prior number of bankruptcies the business has had.
Delinquencies	Numerical	Predictor	The prior number of delinquencies (e.g. missing a payment) the business has had.
Net Value	Numerical	Predictor	The net value of the business.
Status	Categorical	Target	The loan's ending status, which is either "Paid" or "Defaulted".

Table 3: A table showcasing the used variables in the SME loans dataset

The target class distribution is shown in Figure 3.

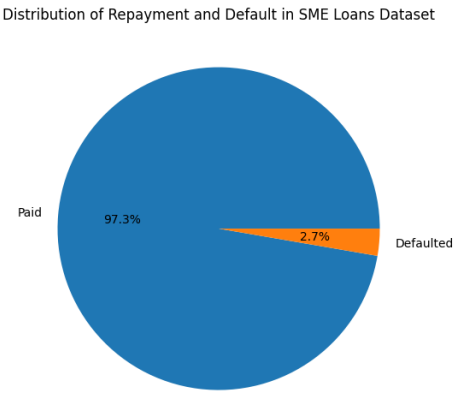

Figure 3: A pie chart showing the SME loans dataset's target class distribution

There is a considerable imbalance between repayers and defaulters, with defaulters only comprising 2.7% of the data. To ensure that repayment was not over-predicted, this imbalance needed to be addressed. To address this, the classifier's class weighting was adjusted. Class reweighting was chosen because while it can have a high learning cost, applying greater weight to certain examples is flexible and maintains data integrity (Zhang et al. 2022).

Alternatively, resampling could have been conducted. While resampling does address imbalance, undersampling can remove useful data, while oversampling can induce overfitting (Barandela et al. 2003). Neither would be useful here given the scarcity of data on defaulters.

3.1.2 Pre-Processing

3.1.2.1 Assumption-Based Dataset Linking

One pre-processing issue encountered was that the disparate datasets were not explicitly linked. For experimentation purposes, it was decided to assume linking was possible and link the datasets using assumptions. While dataset records were not explicitly linked, feasible links were forged.

To link the current account and transaction datasets, the account IDs in the two datasets were paired based on percentiles of cash flow and average monthly transaction amount, with the closest percentiles being paired. Cash flow and transactions can be closely equated (Kotios et al. 2022), and while correct record links are not explicitly known, the dataset's statistical utility is maintained as the links are feasible (Guepin et al. 2024).

The current accounts dataset and loans dataset were also linked using assumptions. The assumption sets tested are outlined and justified in Table 4.

Assumption Set	**Linking Assumption Made**	**Justification**
1	Random linking	While not informed, random linking makes no assumptions and

		implementation is simple. Assumptions cannot represent every case, so making none may arguably have merit.
2	Percentile-based linking - Older businesses will have a longer credit history	Youth is a key reason for lack of credit history, so older businesses typically having longer credit history is a logical assumption.
3	Percentile-based linking - Businesses with worse cash flow (Revenue-Costs) will seek bigger loans	SMEs frequently use loans as a means of servicing debts and providing long-term financial support, so businesses with weaker cash flow may seek larger loans to service this.

Table 4: A table describing and justifying the linking assumptions tested for the current account and loan datasets

(Jones 1992; Kotey 1999; Zhang et al. 2013)

3.1.2.2 Synthetic Data Generation

Only 79 unique accounts were present in the transaction dataset, and some accounts had only 1 or 2 months of transaction history. Resultantly, accounts with only 1 or 2 months were firstly removed from the dataset to ensure that all accounts could provide training data with a usable trend. After this, Python's Synthetic Data Vault (SDV) library was used to generate additional accounts and transactions from the remaining data. Using SDV, 800 additional account IDs were generated and paired with newly generated transactions such that the distribution of account transaction numbers paralleled that of the original data. The correlation coefficient between the two distributions was 0.93 and the cosine similarity was 0.99, so the synthetic data broadly paralleled the original data.

After these changes were made and newly generated accounts not meeting length requirements were removed, the dataset contained 969,057 transactions from 872 unique accounts.

3.1.2.3 Time Series Granularity

For each account's transaction time series, the net sum of transaction amounts was aggregated by month. Monthly granularity was chosen because loan repayments are most commonly made monthly, and in finance, any deeper granularity level lacks value (Mester et al. 2007). While monthly granularity shortens the time series, which raises the difficulty of forecasting accurately (Wang et al. 2008), extracting meaningful insights from a time series is important (Van Goethem et al. 2017). Sample size is irrelevant if these are diluted.

3.2 Modelling Approach

3.2.1 Forecasting Model

For transaction forecasting, Singular Spectrum Analysis (SSA), Prophet and LSTM were tested. SSA and Prophet were tested because they are unused in existing literature and require little training data. For many SMEs, plentiful prior data may not exist. 55% of UK SMEs do not last for 5 years after creation (Global Banking and Finance Review 2014), so as seeking a loan will mostly predate insolvency, many SMEs may lack prior transaction data for training. LSTM was tested because LSTMs have been employed successfully in existing literature, most notably by Ala'raj et al. (2022) for personal transaction forecasting in behavioural scoring.

The SSA models were developed using a Python class, mySSA, sourced from GitHub (aj-cloete 2017), the Prophet models were developed using the Prophet Python library, and the LSTM models were developed in Python using Keras and TensorFlow.

The chosen hyperparameters for each approach are shown in Table 5.

Approach	Hyperparameters
SSA	Without information about an optimal value, the window length L was universally set to N/2, where N is the training dataset length. This resultantly informed the chosen embedding dimension K, derived from the formula N – L + 1. The suspected seasonality was set to the lag with the highest autocorrelation. If no autocorrelations were statistically significant, this was set as 0. For each account, the use of all potential signal numbers in reconstruction was tested, with the number of signals returning the lowest RMSE becoming the "chosen" model for that account.
Prophet	No tuning was required for the Prophet model, as it infers many parameters automatically.
LSTM	The LSTM architecture was built to parallel that used by Ala'raj et al. (2022). They used 5 LSTM layers, each with 16 neurons and a hyperbolic tangent activation function, followed by 2 densely connected layers with a sigmoid activation function and an output layer. 20 epochs and an Adam optimiser were used for training.

Table 5: A table outlining the chosen hyperparameters for each model

Various considered forecasting approaches are compared in Table 6.

Model	**Advantages**	**Disadvantages**	**Chosen? (Yes/No)**	**Justification**
SSA	SSA makes limited assumptions and requires little prior data.	SSA requires careful parameter choices and can overfit or underfit if these are chosen poorly.	Yes	With SSA's minimal assumptions and low training data requirements, it should work well.
Prophet	Prophet requires little training data and can infer many parameters easily.	Prophet can exaggerate seasonality, even if it often predicts it correctly.	Yes	While Prophet can exaggerate seasonality, its lack of assumptions and low training data requirement make it a strong choice.
LSTM	LSTMs make limited assumptions and can derive complex relationship dynamics such as non-linearity.	LSTMs can require large amounts of training data.	Yes	While LSTMs can require plentiful training data, they have performed strongly in literature, so LSTM is a strong choice.

ARIMA	ARIMA is simple to implement and is a common forecasting technique.	ARIMA assumes linearity in the data, so can struggle with complex predicted trajectories.	No	With the time series form potentially varying by business, ARIMA's linearity assumption made it suboptimal.
Holt Winter's Exponential Smoothing (HW)	HW is commonly used and has proved effective.	HW requires a linear trend and known seasonality, which SMEs may lack.	No	With its restrictive assumptions, HW was a weak choice.

Table 6: A table comparing considered forecasting approaches

(Hassani and Thomakos 2010; Khashei and Bijari 2011; Siami-Namini et al. 2018; Almazrouee et al. 2020; Golyandina 2020; Ma 2020; Kumar Jha and Pande 2021; Ning et al. 2022)

3.2.2 Classification Model

For the classifier, XGBoost was chosen because it has commonly been employed effectively for SME credit scoring in literature. Grid searches were used to select the learning rates, maximum depths and estimator numbers because while they scale poorly on large search spaces (Liashchynskyi and Liashchynskyi 2019), they provide a guaranteed optimum (Huang et al. 2012). Due to the small search space here (only three hyperparameters were tested), it was decided that a guaranteed optimum warranted increased computational complexity.

Stratified k-fold cross validation was also employed to prevent biased evaluation and ensure equal class representation (Jung 2018). While k-fold can be computationally intensive (Yadav and Shukla 2016), a moderate k value of 10 was chosen to mitigate

this. Marcot and Hanea (2021) found this to be optimal for balancing complexity with robustness.

Various considered classifiers are compared in Table 7.

Approach	**Advantages**	**Disadvantages**	**Chosen? (Yes/No)**	**Justification**
XGBoost	XGBoost performs strongly, integrates regularisation to prevent overfitting and is simple to implement.	Interpretability is lost compared to a regular tree-based method.	Yes	With its strong performance, integrated regularisation, simple implementation and successful uses in literature, XGBoost was ideal.
LR	LR is simple to implement and can handle multiple continuous variables simultaneously.	Assumes linear relationships, so struggles with non-linearity and co-dependency.	No	With its restrictive assumptions, LR was a weak choice.
NN	NNs assume little and can model complex relationships.	NNs are computationally intensive and can overfit if tuned poorly.	No	While NNs are powerful, their propensity to overfitting made them a weak choice.
KNN	KNN is simple to implement	KNN is inefficient and dependent on	No	With KNN being inefficient and

	and often effective.	the k value chosen.		dependent on the k value, it was suboptimal here.
RF	RFs perform highly, make limited assumptions and accommodate many variables.	RFs can be computationally complex.	No	While RFs perform highly, their complexity made them suboptimal.

Table 7: A table comparing various considered classifiers

(Livingstone et al. 1997; Guo et al. 2003; Sperandei 2014; Hooman et al. 2016; Nielsen 2016; Aydin and Ozturk 2021; Wu et al. 2021; Zhang 2022; Langsetmo et al. 2023; Salman et al. 2024)

3.3 Experimental Approach

To address RQ3 and RO2, the forecasting model was tested. SSA and Prophet were tested on all accounts in the transaction dataset and compared to LSTM, with separate models being developed for each account, trained on its transactions and tested. A naïve forecast, which purely forecasts based on the prior value, was also developed for baseline comparison. The models were evaluated by training on the first 70% of the account's transaction history, using the last 30% to test and calculating the Root Mean Squared Error (RMSE) and Scatter Index (SI) between the test data and the model forecast. Both cumulative transaction forecasting and month-by-month transaction forecasting were tested. For assessment of both cumulative and month-by-month forecasting, the error metrics attained across all accounts were averaged, and the most effective of the approaches was used in classification.

RMSE was used to evaluate because while other measures such as mean absolute error (MAE) are arguably clearer (Willmott and Matsuura 2005), RMSE penalises larger errors more and better exhibits performance variation (Chai and Draxler 2014). SI was used because showing RMSE relative to the mean test observation puts it into

perspective more (Karunasingha 2022). SI was primarily used to compare approaches because it is scale-agnostic, simplifying direct comparison.

To address RQ4 and RO2, the optimal forecasting approach was used to perform 12-month cumulative and month-by-month forecasts for each account. Variables were derived from these transaction forecasts and the pairing with the current accounts dataset was performed as previously outlined. These accounts were also paired with loan dataset accounts using three different assumption sets as outlined in Table 4. An XGBoost classifier was trained using the paired data and used to classify an account as a repayer or a defaulter. Seven variable subsets were tested and compared, as shown in Table 8.

Variable Subset	**Subset Name**	**Variables Used**
1	Cumulative Forecast	Starting Amount, Ending Amount, Net Change over Forecast Period
2	Month-by-Month Forecast	Highest Amount, Lowest Amount, Difference over Forecast Period
3	Cumulative + Month-by-Month	Combines 1 and 2
4	Traditional Variables	Primary Sector, Company Type, Revenue, Costs, Accounts Receivable, Capital and Reserves, Current Assets, Current Liabilities, Fixed Assets, Long-Term Liabilities, Provisions for Liabilities, Loan Amount, Years of Credit History, Total Credit Lines, Open Credit Lines, Bankruptcies, Delinquencies, Net Value

5	Traditional + Forecasting	Combines 4 with the most effective of 1, 2 and 3.
6	Uniform Random Dummy Classifier (Baseline)	N/A
7	Most Frequent Dummy Classifier (Baseline)	N/A

Table 8: A table describing the variable subsets tested

70% of records were used for training, and 30% were used for testing.

The macro averages of precision and recall across both classes were chosen to evaluate because they focus on metrics such as false positive rate and false negative rate, which provide greater insight where imbalance exists (Saito and Rehmsmeier 2015). Accuracy was not chosen because it can be unrepresentative on imbalanced datasets (Thabtah et al. 2020), and an imbalance of approximately 36:1 existed here.

4 Results and Evaluation

4.1 Transaction Forecasting

To address RQ3 and RO2, transaction forecasting models were developed. Cumulative and month-by-month transaction forecasting were both attempted, and for each, SSA, Prophet, LSTM and a naïve forecast were tested. SI was used to assess the approaches' efficacy, and both the mean and median SI were recorded to assess the performance distribution.

4.1.1 Cumulative Forecasting

The cumulative forecasting results are shown in Table 9 and Figure 4.

Model	Mean RMSE	Mean SI	Median RMSE	Median SI
SSA	317,311.2	89.6%	10,585.7	61.5%
Prophet	106,535.9	60.3%	8,636.1	35.4%
LSTM	131,782.2	65.0%	8,719.9	40.3%
Naive	60,743	36.0%	4,654	17.4%

Table 9: A table showing the cumulative forecasting results

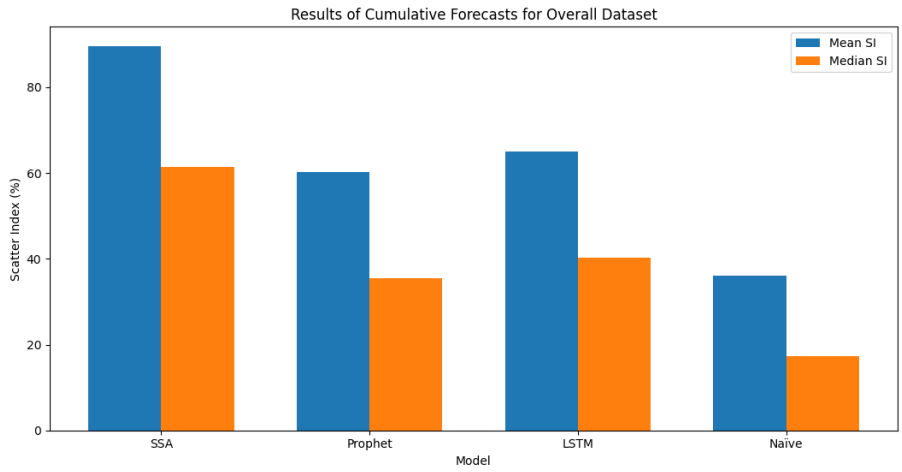

Figure 4: A graph comparing model performances for cumulative forecasting

The naïve forecast performed most strongly for cumulative forecasting, with a mean SI of 36% and a median SI of 17%. Prophet also outperformed LSTM, with mean and median SIs of 60% and 35% respectively compared to mean and median SIs of 65% and 40% for LSTM. SSA performed most weakly, with mean and median SIs of 90% and 62% respectively.

Despite its simplicity, the results imply that a naïve forecast may be the optimal approach for cumulative transaction forecasting.

4.1.2 Month-by-Month Forecasting

The month-by-month forecasting results are shown in Table 10 and Figure 5.

Model	Mean RMSE	Mean SI	Median RMSE	Median SI
SSA	21,288.2	437.3%	3,637.5	145.0%
Prophet	25,731.8	481.3%	4,760.8	182.0%
LSTM	54,843.3	862.3%	6,941.0	295.2%
Naive	14,612.5	491.3%	3,957.4	153.4%

Table 10: A table showing the month-by-month forecasting results.

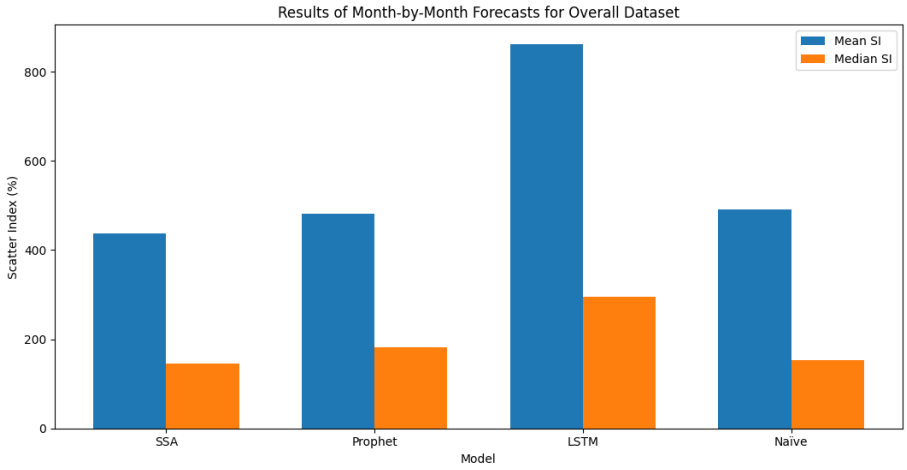

Figure 5: A graph comparing model performances for month-by-month forecasting

SSA performed most strongly for month-by-month forecasting, attaining mean and median SIs of 437% and 145% respectively. Naïve forecasting also performed strongly once again, attaining mean and median SIs of 491% and 153% respectively. Prophet was comparable or slightly weaker, attaining mean and median SIs of 481% and 182% respectively, and LSTM was a notably weaker performer, attaining mean and median SIs of 862% and 295% respectively.

These results suggest that SSA is the strongest choice for month-by-month transaction forecasting, but that naïve forecasting remains relatively strong. Notably, month-by-month forecasting accuracy was also universally lower than that for cumulative forecasting.

4.1.3 Longer Subset

One interesting commonality between the cumulative and month-by-month forecasting results is that the mean SI was universally significantly higher than the median SI, implying a right-tailed performance distribution and a select few notably weaker performing accounts. With a select few accounts also lacking training data and a good sample size being important for reliable forecasting (Hyndman 2007), the forecasting approaches were resultantly tested on a subset of the data containing only accounts with 14 or more months of transactions (the median amount or higher), to see if results improved and the right tail effect lessened.

The cumulative forecasting results for the longer subset are shown in Table 11 and Figure 6.

Model	Mean RMSE	Mean SI	Median RMSE	Median SI
SSA	525,789.4	71.7%	23,368.3	56.6%
Prophet	175,260.7	44.4%	14,466.6	27.9%
LSTM	209,508.5	49.2%	16,872.8	41.2%
Naive	100,331.8	28.2%	8,066.5	15.7%

Table 11: A table showing the longer subset's cumulative forecasting results

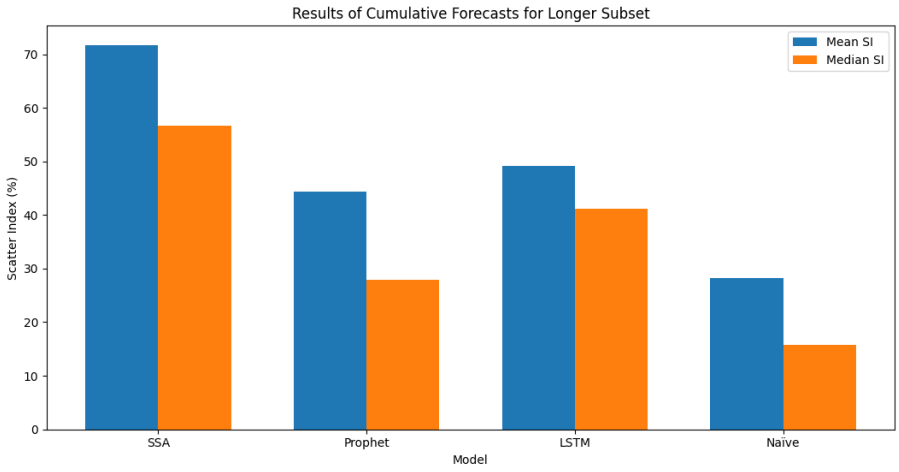

Figure 6: A graph comparing the longer subset's model performances for cumulative forecasting

On the longer subset, the right-tailed nature of the performance distribution lessened, and performance improved. The broad picture persisted, with naïve forecasting being the strongest performer and SSA being the weakest, but both mean and median SIs improved by varying amounts in all cases when the longer subset was used.

The month-by-month forecasting results for the longer subset are shown in Table 12 and Figure 7.

Model	Mean RMSE	Mean SI	Median RMSE	Median SI
SSA	33,644.4	181.6%	5,478.4	126.9%
Prophet	40,963.0	211.7%	6,850.3	156.1%
LSTM	77,482.8	386.6%	11,183.4	231.9%
Naive	22,773.9	173.8%	6,098.0	134.8%

Table 12: A table showing the longer subset's month-by-month forecasting results

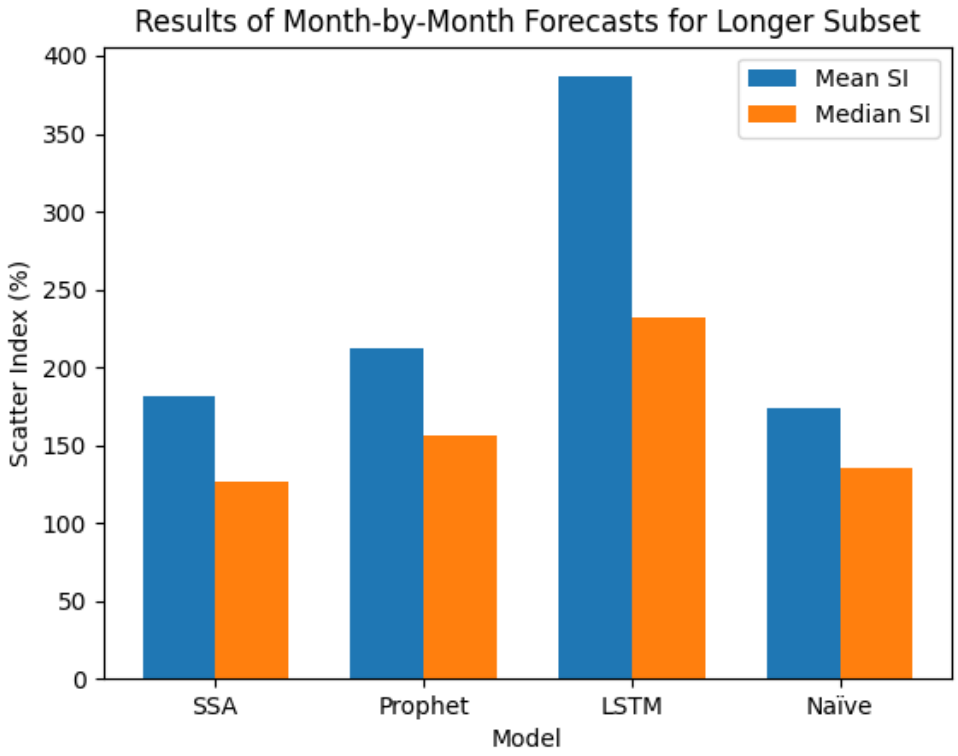

Figure 7: A graph comparing the longer subset's model performances for month-by-month forecasting

As with cumulative forecasting, using the longer subset improved performance and lessened the right tail of the performance distribution across the board. The broad picture of model performances persisted, with LSTM performing notably more weakly, but SSA and naïve forecasting were more closely tied on the longer subset. Naïve forecasting had the lowest mean SI and a lesser right tail in its performance distribution, but SSA had the lowest median SI.

4.1.4 Analysis of Top and Bottom Performers

While the right-tailed nature of performance distributions decreased and performance improved with the longer subset compared to the whole dataset, a considerable right tail was still universally present, indicating that insufficient training data may not solely cause poor predictive performance. Therefore, the strongest and weakest performing accounts in the longer subset were compared, using SSA as an example case, to see if there were any key differences.

Notably, there appeared to be greater relative volatility in the most recent transaction patterns of the weaker performing accounts for both cumulative and month-by-month forecasting. To test this hypothesis, the final 3 percentage changes in each account among the top and bottom 10 performing accounts were averaged and put into a list. A one-tailed, two-sample t-test was used to test for statistically significant evidence of a larger average percentage change among the weakest performers than the strongest performers.

On a cumulative basis, the p-value for the one-tailed t-test was 0.002, indicating statistically significant evidence of larger recent volatility at the 1% significance level among weaker performing accounts at the cumulative level. On a month-by-month basis, the p-value for the one-tailed t-test was 0.029, indicating statistically significant evidence of larger recent volatility at the 5% significance level among weaker performing accounts at the month-by-month level.

These findings would indicate that forecasting is less accurate when recent transaction patterns have been volatile or there has been a recent notable shock. Resultantly, transaction forecasting may not be ideal where evidence of recent relative volatility exists.

4.2 Classification

To address RQ4, RO2 and RO3, a classification model was developed to discriminate between defaulters and repayers. XGBoost was used to classify in this instance, and five different permutations of variables were tested alongside two dummy classifiers for baseline comparison, as outlined in Table 8.

For dataset pairing, three assumption sets were tested, as outlined in Table 4.

The classification results from assumption set 1 are shown in Table 13 and Figure 8.

Variable Subset	Default Precision	Default Recall	Repaid Precision	Repaid Recall	Average Precision	Average Recall
1	4%	40%	97%	67%	50%	53%
2	5%	60%	98%	66%	52%	63%
3	5%	20%	97%	88%	51%	54%
4	14%	100%	100%	81%	57%	90%
5	15%	80%	99%	86%	57%	83%
6 (Baseline)	3%	60%	97%	44%	50%	52%
7 (Baseline)	0%	0%	97%	100%	49%	50%

Table 13: A table showing the classification results for assumption set 1

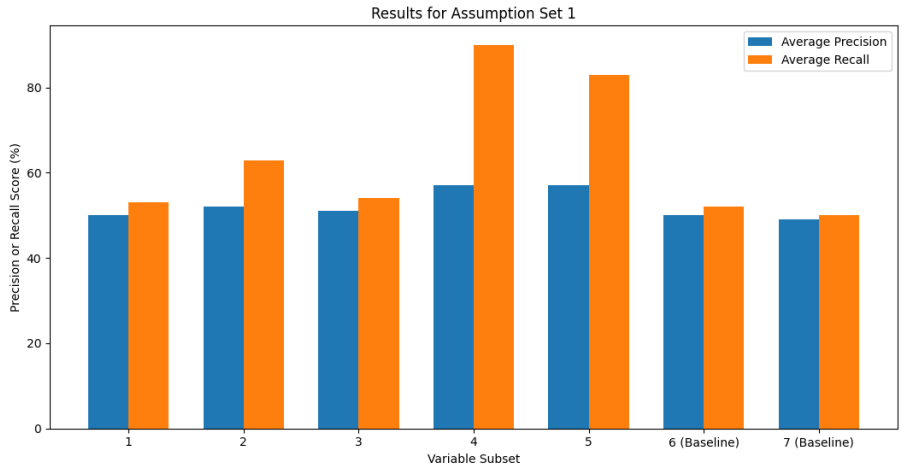

Figure 8: A graph comparing variable subset performances for assumption set 1

When assumption set 1 was used, month-by-month forecasting was more effective, particularly in terms of recall, than cumulative forecasting or both combined, but neither matched traditional variables in isolation. When month-by-month forecasting variables were combined with traditional variables, average recall lowered by 7% and

average precision remained identical, indicating that forecasting did not add meaningful value to the classification.

However, month-by-month forecasting did beat both baseline measures in terms of average recall by over 10%, even if its performance did not match that of traditional variables. Therefore, it did provide some value in isolation, albeit less than traditional variables.

The classification results with assumption set 2 are shown in Table 14 and Figure 9.

Variable Subset	Default Precision	Default Recall	Repaid Precision	Repaid Recall	Average Precision	Average Recall
1	5%	40%	98%	75%	51%	57%
2	4%	80%	99%	44%	51%	62%
3	4%	80%	99%	44%	51%	62%
4	30%	60%	99%	96%	64%	78%
5	27%	60%	99%	95%	63%	78%
6 (Baseline)	4%	60%	98%	57%	51%	59%
7 (Baseline)	0%	0%	97%	100%	49%	50%

Table 14: A table showing the classification results for assumption set 2

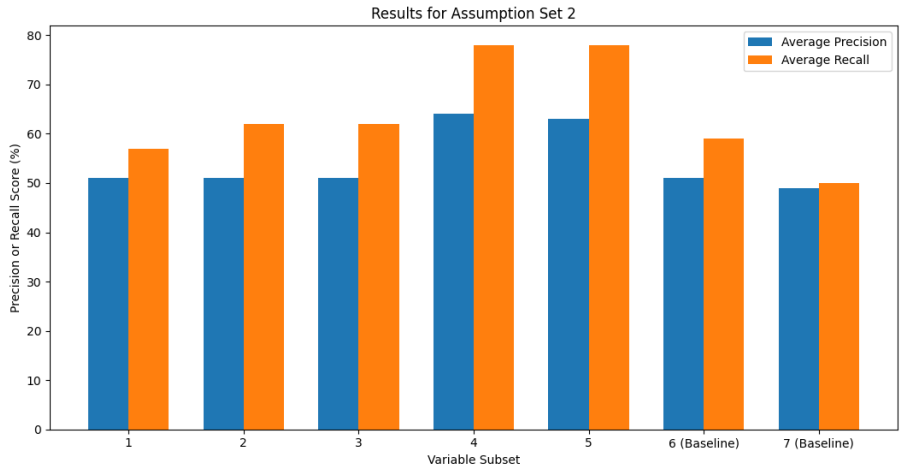

Figure 9: A graph comparing variable subset performances on assumption set 2

Once again, it was found that month-by-month forecasting was more effective in isolation than cumulative forecasting, and the combination of both did not improve performance. Traditional variables also notably outperformed forecasting variables once again, with the addition of forecasting variables alongside traditional variables not meaningfully improving performance.

Cumulative forecasting had weaker performance than a random baseline classifier, and month-by-month and combined forecasting only improved in recall by 3% over the random classifier, compared with a 19% improvement when traditional variables were used.

The classification results with assumption set 3 are shown in Table 15 and Figure 10.

Variable Subset	Default Precision	Default Recall	Repaid Precision	Repaid Recall	Average Precision	Average Recall
1	8%	100%	100%	65%	54%	83%
2	14%	100%	100%	80%	57%	90%
3	19%	80%	99%	90%	59%	85%
4	19%	80%	99%	90%	59%	85%
5	7%	80%	99%	67%	53%	73%
6 (Baseline)	3%	60%	97%	45%	50%	53%
7 (Baseline)	0%	0%	97%	100%	49%	50%

Table 15: A table showing the classification results for assumption set 3

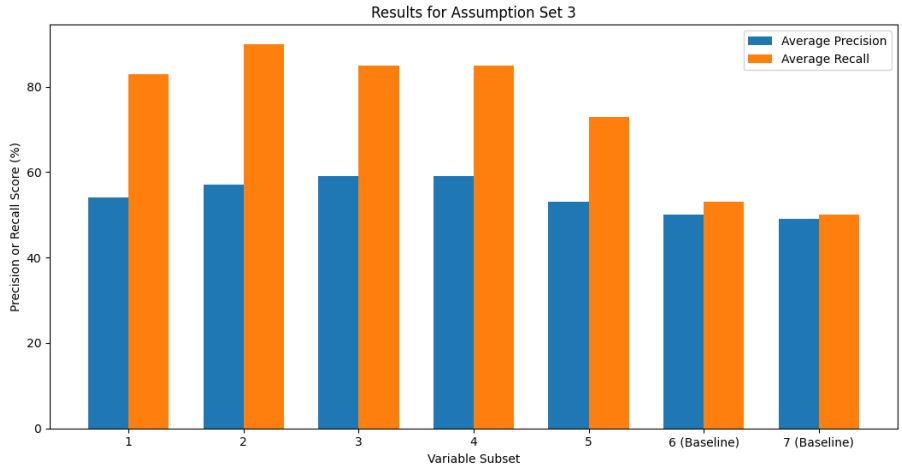

Figure 10: A graph comparing variable subset performances on assumption set 3

Yet again, month-by-month forecasting was more effective than cumulative forecasting in isolation, with 3% higher precision and 7% higher recall. The combination of both attained 2% more precision, but 5% less recall than month-by-month, so as month-by-month also used fewer variables, it was still deemed to be superior on balance. Interestingly, the gains of using traditional variables also lessened on this assumption set. Traditional variables alone attained the same results as the

combination of forecasts did, and the combination of forecasting and traditional variables performed most weakly of the five main classifiers.

All models outperformed the baseline measures, however, with even the weakest performer boasting 3% greater average precision and 20% greater average recall.

4.3 Evaluation

Research of existing literature in credit scoring found that data on past transactions had been used successfully in SME credit scoring, and that transaction forecasting, while successful in behavioural scoring for personal consumers, had not yet been attempted in application scoring for SMEs. Time series forecasting methods aside from LSTM had also not been tested for transaction prediction, despite LSTM's shortcomings. This study aimed to plug these gaps by comparing SSA and Prophet, alongside a baseline naïve forecast, with LSTM for transaction forecasting. The inclusion of both cumulative and month-by-month forecasting variables in an XGBoost-based credit scoring model was also tested, and the approaches' efficacy was tested using synthetic data.

The study firstly aimed to examine whether other forecasting methods could outperform LSTM for transaction prediction. It was found that overall, other methods could outperformed LSTM, but that performance was context dependent.

For cumulative forecasting, Prophet outperformed LSTM, but the strongest performer was the baseline naïve forecast, which attained a mean SI of 36.1% and a median SI of 17.4% when tested on all accounts. Despite its simplicity, a naïve forecast has often been found to be the strongest choice where volatility is high and trend and seasonality are weak, particularly where training data is relatively lacking (Beck et al. 2025). With SMEs often having greater financial volatility and exposure to economic shocks than larger corporations (Behera et al. 2024), naïve forecasting may match or better some more complex methods for SME transaction prediction due to the unreliability of any extrapolated patterns. These may be particularly unreliable in this dataset as all accounts, relatively speaking, lacked training data. Some had only 2 months of training data, and even the most populated only had 10-11 months after a train/test split.

For month-by-month forecasting, LSTM was a notably weaker performer, with all other approaches outperforming it. SSA was the strongest performer, with mean and median SIs of 437% and 145% respectively being attained, but naïve forecasting once again performed relatively strongly, with it being the 2nd strongest performer, and it closely matched SSA on the longer subset. However, month-by-month forecasting accuracy was universally much weaker than that of cumulative. This could be because trend and seasonality are important aspects of an accurate time series forecast (Kourentzes et al. 2014), and while cumulative transaction tracking at least aims to introduce a trend, month-by-month transaction tracking guarantees neither trend nor seasonality.

Interestingly, the mean SI was considerably higher than the median SI across the board in both cumulative and month-by-month forecasting, indicating a notably right-tailed performance distribution. It was initially felt that this tail may have been caused by insufficient training data for some accounts, but while the tail reduced and performance improved when only longer-than-average accounts were tested, a tail persisted, indicating that a lack of prior data did not solely cause poor predictive performance. A comparison of strongly and weakly performing accounts found that weaker performers had greater relative volatility among their most recent months of transactions on both a cumulative and month-by-month basis. Resultantly, it can be inferred that transaction forecasting may not be ideal where there has been considerable recent relative volatility in transaction history. However, volatility can be an indicator of default risk (Bao et al. 2020), so a business with highly volatile transaction patterns could be inherently riskier than one with more consistent patterns.

The other aim of the study was to examine whether transaction forecasting data could improve credit scoring efficacy compared to the use of traditional variables. Three pairing assumption sets were tested, and overall, transaction data did not add notable value compared to using traditional variables in any case.

Using assumption sets 1 and 2, transaction data weakened classification efficacy, and even using assumption set 3, month-by-month efficacy only narrowly matched traditional variable efficacy (month-by-month forecasting attained higher recall, but traditional variables attained higher precision). The reason for this could be due to

SME transactions being hard to accurately forecast. Many SMEs will be both volatile and lacking in financial history (Chen et al. 2022), thus will lack reliable trend and seasonality to provide an accurate forecast. If forecast accuracy is lower, the forecast will provide less meaningful insight into the relationship between future transactions and default.

It was universally found that despite being harder to forecast accurately, month-by-month forecasting provided greater value than cumulative forecasting. In two assumption sets, it was notably more effective than cumulative or combined, and in the 3rd, it was still slightly higher than combined (albeit the difference was lower). With volatility being an indicator of default risk (Bao et al. 2020), it may be more relevant than any overall trend. Therefore, month-by-month forecasting could provide greater value because while month-by-month amounts are harder to predict accurately, they better showcase a business' financial volatility than cumulative forecasting, which purely tracks net trend.

It should be noted that records were paired using assumptions, and the attained results arguably exhibit the weaknesses of synthetic pairing. No classifier attained particularly strong results, with a maximum of 64% average precision and average recall seldom exceeding 80% in any case. Assumption sets 2 and 3, using logical assumptions, generally yielded higher performance than the random pairings of assumption set 1, but the results still frequently exhibit considerable room for improvement. No matter how informed the assumptions are, synthetic data arguably cannot replace real-world data, as assumptions are blunt and cannot consider edge cases (Van Breugel et al. 2023). Real-world, concretely linked data could not be accessed here, but the results arguably prove that real-world, linked data is necessary to exploit optimal insights from ML methods in credit scoring. However, it could also be argued that real-world data does not guarantee a stronger result and would simply produce a more robust one (Liu and Demosthenes 2022), so with this in mind, transaction forecasting may still not be as effective as traditional variables.

4.4 Limitations and Further Work

While the utmost effort was made to ensure robustness, this study did have limitations.

Firstly, data availability was a significant limitation. Real-world data could not be made, so synthetic data had to be used. With the datasets being synthetic and lacking concrete links, assumptions were also necessary to make them usable. While the assumptions made were informed and aimed to maximise value, synthetically linked data arguably cannot replicate the robustness of real-world data with confirmed links.

Another limitation of this study is that volatility in transaction patterns was not meaningfully considered as a predictive variable. Both the forecasting results, inferring that volatile businesses are harder to predict accurately, and the classification results, inferring that the month-by-month forecast more transparently exhibiting volatility was more effective, would infer that volatility is an important aspect of SME credit risk, and this was not examined meaningfully here.

Further work recommendations can resultantly be made. Firstly, testing the approach with real-world, linked data is recommended, to test its true efficacy on data where known links exist. Secondly, researching the effect of volatility in transaction patterns upon SME credit risk is also recommended. It may be interesting to trial a volatility forecasting approach such as GARCH either instead of or alongside one of the forecasting approaches used here, as considering a business' financial volatility may provide interesting insights. Volatility is likely to need considering if ML methods are used in SME credit scoring, with the volatility of many SMEs.

5 References

Abdou, H.A. and Pointon, J. 2011. Credit Scoring, Statistical Techniques and Evaluation Criteria: A Review of the Literature. *Intelligent Systems in Accounting, Finance and Management* 18(2–3), pp. 59–88. doi: 10.1002/isaf.325.

Abedin, M.Z., Guotai, C., Hajek, P. and Zhang, T. 2023. Combining weighted SMOTE with ensemble learning for the class-imbalanced prediction of small business credit risk. *Complex and Intelligent Systems* 9(4), pp. 3559–3579. doi: 10.1007/s40747-021-00614-4.

Ahmed, A.N. 2019. *Credit-Risk Assessment of Small Business Loans using Naïve Bayes, Decision Tree and Random Forest*. MSc, National College of Ireland.

aj-cloete. 2017. *pssa*. Available at: https://github.com/aj-cloete/pssa [Accessed: 14 July 2025].

Ala'raj, M., Abbod, M.F., Majdalawieh, M. and Jum'a, L. 2022. A deep learning model for behavioural credit scoring in banks. *Neural Computing and Applications* 34(8), pp. 5839–5866. doi: 10.1007/s00521-021-06695-z.

Almazrouee, A.I., Almeshal, A.M., Almutairi, A.S., Alenezi, M.R. and Alhajeri, S.N. 2020. Long-term forecasting of electrical loads in Kuwait using prophet and holt-winters models. *Applied Sciences (Switzerland)* 10(16), p. 5627. doi: 10.3390/app10165627.

Annett, J. 2002. Subjective rating scales: Science or art? *Ergonomics* 45(14), pp. 966–987. doi: 10.1080/00140130210166951.

Avery, R.B., Calem, P.S. and Canner, G.B. 2004. Consumer credit scoring: Do situational circumstances matter? *Journal of Banking & Finance* 28(4), pp. 835–856. doi: 10.1016/j.jbankfin.2003.10.009.

Aydin, Z.E. and Ozturk, Z.K. 2021. Performance Analysis of XGBoost Classifier with Missing Data. *Manchester Journal of Artificial Intelligence and Applied Sciences (MJAIAS* 2(2). Available at: https://www.researchgate.net/publication/350135431 [Accessed: 21 July 2025].

Bams, D., Pisa, M. and Wolff, C.C.P. 2021. Spillovers to small business credit risk. *Small Business Economics* 57(1), pp. 323–352. doi: 10.1007/s11187-019-00308-9.

Bao, M.X., Billett, M.T., Smith, D.B. and Unlu, E. 2020. Does Other Comprehensive Income Volatility Influence Credit Risk and the Cost of Debt? *Contemporary Accounting Research* 37(1), pp. 457–484. doi: 10.1111/1911-3846.12548.

Barandela, R., Sanchez, J.S., Garcia, V. and Ferri, F.J. 2003. Learning from Imbalanced Sets through Resampling and Weighting. In: *Pattern Recognition and Image Analysis*. Mallorca, Spain: Springer, pp. 80–88.

Beck, N., Dovern, J. and Vogl, S. 2025. Mind the naive forecast! a rigorous evaluation of forecasting models for time series with low predictability. *Applied Intelligence* 55(6). doi: 10.1007/s10489-025-06268-w.

Behera, P.K., Sahu, N.C. and Mahanta, A. 2024. Volatility Spillover and Connectedness Between SME and Main Markets of India and China. *Asia-Pacific Financial Markets*. doi: 10.1007/s10690-024-09492-5.

Bitetto, A., Cerchiello, P., Filomeni, S., Tanda, A. and Tarantino, B. 2024. Can we trust machine learning to predict the credit risk of small businesses? *Review of Quantitative Finance and Accounting*. doi: 10.1007/s11156-024-01278-0.

Van Breugel, B., Qian, Z. and Van Der Schaar, M. 2023. Synthetic Data, Real Errors: How (Not) to Publish and Use Synthetic Data. In: *Proceedings of the 40th International Conference on Machine Learning*. Honolulu, Hawaii: PMLR, pp. 34793–34808.

Brevoort, K.P., Grimm, P. and Kambara, M. 2016. Credit Invisibles and the Unscored. *Cityscape* 18(2), pp. 9–34. doi: 10.2307/26328254.

Chai, T. and Draxler, R.R. 2014. Root mean square error (RMSE) or mean absolute error (MAE)? -Arguments against avoiding RMSE in the literature. *Geoscientific Model Development* 7(3), pp. 1247–1250. doi: 10.5194/gmd-7-1247-2014.

Chen, T., Huang, Y., Lin, C. and Sheng, Z. 2022. Finance and Firm Volatility: Evidence from Small Business Lending in China. *Management Science* 68(3), pp. 2226–2249. doi: 10.1287/mnsc.2020.3942.

Chen, Y.J. and Chen, Y.M. 2022. Forecasting corporate credit ratings using big data from social media. *Expert Systems with Applications* 207. doi: 10.1016/j.eswa.2022.118042.

Christoff, Z., Hansen, J.U. and Proietti, C. 2016. Reflecting on Social Influence in Networks. *Journal of Logic, Language and Information* 25(3–4), pp. 299–333. doi: 10.1007/s10849-016-9242-y.

Djeundje, V.B., Crook, J., Calabrese, R. and Hamid, M. 2021. Enhancing credit scoring with alternative data. *Expert Systems with Applications* 163. doi: 10.1016/j.eswa.2020.113766.

Dushimimana, B., Wambui, Y., Lubega, T. and McSharry, P.E. 2020. Use of Machine Learning Techniques to Create a Credit Score Model for Airtime Loans. *Journal of Risk and Financial Management* 13(8). doi: 10.3390/jrfm13080180.

El-Qadi, A., Trocan, M., Conde-Cespedes, P., Frossard, T. and Diaz-Rodriguez, N. 2023. Credit Risk Scoring Using a Data Fusion Approach. In: *Computational Collective Intelligence*. Budapest, Hungary: Springer, pp. 769–781.

Fathy, T. 2024. Artificial Intelligence and Predictive Data Analytics to Enhance Risk Assessment and Credit Scoring Mechanisms in Retail Banking. *Journal of Data Science, Predictive Analytics, and Big Data Applications* 9(9), pp. 1–9.

Federal Reserve Banks. 2017. *Small Business Credit Survey*. Available at: www.fedsmallbusiness.org/partnership.

Feng, Y., Mao, Y. and Hu, Y. 2022. Adaptive iterative clustering and decision tree for multimodal personal credit prediction. In: *Second International Conference on Digital Signal and Computer Communications (DSCC 2022)*. SPIE-Intl Soc Optical Eng, p. 8. doi: 10.1117/12.2641274.

Frame, W.S. and Woosley, L. 2004. Credit scoring and the availability of small business credit in low-and moderate-income areas. *Financial Review* 39(1), pp. 35–54. doi: 10.1111/j.0732-8516.2004.00066.x.

Gao, T. and Lau, R.Y.K. 2024. Leveraging Deep Learning and Multimodal Signals from Social Media to Enhance Credit Risk Prediction. In: *2024 IEEE International Conference on Signal Processing, Communications and Computing, ICSPCC 2024*. Institute of Electrical and Electronics Engineers Inc., pp. 1–6. doi: 10.1109/ICSPCC62635.2024.10770419.

Gicić, A., Đonko, D. and Subasi, A. 2023. Intelligent credit scoring using deep learning methods. *Concurrency and Computation: Practice and Experience* 35(9). doi: 10.1002/cpe.7637.

Global Banking and Finance Review. 2014. *Growing Pains: Majority of SMEs Don't Last Beyond 5 Years*. Available at: https://www.globalbankingandfinance.com/growing-pains-majority-of-smes-dont-survive-five-years [Accessed: 7 July 2025].

Van Goethem, A., Staals, F., Löffler, M., Dykes, J. and Speckmann, B. 2017. Multi-Granular Trend Detection for Time-Series Analysis. *IEEE Transactions on Visualization and Computer Graphics* 23(1), pp. 661–670. doi: 10.1109/TVCG.2016.2598619.

Golyandina, N. 2020. Particularities and commonalities of singular spectrum analysis as a method of time series analysis and signal processing. *Wiley Interdisciplinary Reviews: Computational Statistics* 12(4). doi: 10.1002/wics.1487.

Gu, Z., Lv, J., Wu, B., Hu, Z. and Yu, X. 2024. Credit risk assessment of small and micro enterprise based on machine learning. *Heliyon* 10(5). doi: 10.1016/j.heliyon.2024.e27096.

Guepin, F., Meeus, M., Cretu, A.-M. and De Montjoye, Y.-A. 2024. Synthetic Is All You Need: Removing the Auxiliary Data Assumption for Membership Inference Attacks Against Synthetic Data. In: *Computer Security: ESORICS 2023 International Workshop*. Springer, pp. 182–198. doi: https://doi.org/10.1007/978-3-031-54204-6_10.

Gulsoy, N. and Kulluk, S. 2019. A data mining application in credit scoring processes of small and medium enterprises commercial corporate customers. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 9(3). doi: 10.1002/widm.1299.

Guo, G., Wang, H., Bell, D., Bi, Y. and Greer, K. 2003. KNN Model-Based Approach in Classification. In: *On The Move to Meaningful Internet Systems 2003: CoopIS, DOA and ODBASE*. Springer, pp. 986–996. doi: https://doi.org/10.1007/978-3-540-39964-3_62.

Guo, G., Zhu, F., Chen, E., Liu, Q., Wu, L. and Guan, C. 2016. From footprint to evidence: An exploratory study of mining social data for credit scoring. *ACM Transactions on the Web* 10(4), pp. 1–38. doi: 10.1145/2996465.

Hassani, H. and Thomakos, D. 2010. A review on singular spectrum analysis for economic and financial time series. *Statistics and Its Interface* 3(3), pp. 377–397. doi: 10.4310/sii.2010.v3.n3.a11.

Hooman, A., Marthandan, G., Yusoff, W.F.W., Omid, M. and Karamizadeh, S. 2016. Statistical and data mining methods in credit scoring. *The Journal of Developing Areas* 50(5), pp. 371–381. doi: 10.1353/jda.2016.0057.

Huang, J. et al. 2024. Open-FinLLMs: Open Multimodal Large Language Models for Financial Applications. Available at: http://arxiv.org/abs/2408.11878 [Accessed: 17 June 2025].

Huang, Q., Mao, J. and Liu, Y. 2012. An Improved Grid Search Algorithm of SVR Parameters Optimization. In: *2012 IEEE 14th International Conference on Communication Technology*. Chengdu, China: IEEE, pp. 1022–1026. doi: 10.1109/ICCT.2012.6511415.

Hurley, M. and Adebayo, J. 2016. Credit Scoring in the Era of Big Data. *Yale Journal of Law and Technology* 18, pp. 148–216. Available at: http://abcnews. Hyndman, R.J. 2007. Minimum Sample Size Requirements for Seasonal Forecasting Models. Available at: https://www.researchgate.net/publication/5055549.

Jones, S.L. 1992. How to Obtain a Small Business Loan. *Economic Development Review* 10(2).

Jung, Y. 2018. Multiple predicting K-fold cross-validation for model selection. *Journal of Nonparametric Statistics* 30(1), pp. 197–215. doi: 10.1080/10485252.2017.1404598.

Karimova, N. 2024. Application of AI in Credit Risk Scoring for Small Business Loans: A case study on how AI-based random forest model improves a Delphi model outcome in the case of Azerbaijani SMEs. *arXiv preprint*.

Karunasingha, D.S.K. 2022. Root mean square error or mean absolute error? Use their ratio as well. *Information Sciences* 585, pp. 609–629. doi: 10.1016/j.ins.2021.11.036.

Khashei, M. and Bijari, M. 2011. A novel hybridization of artificial neural networks and ARIMA models for time series forecasting. In: *Applied Soft Computing Journal*. pp. 2664–2675. doi: 10.1016/j.asoc.2010.10.015.

Kotey, B. 1999. Debt Financing and Factors Internal to the Business. *International Small Business Journal* 17(3), pp. 11–29. doi: 10.1177/0266242699173001.

Kotios, D., Makridis, G., Fatouros, G. and Kyriazis, D. 2022. Deep learning enhancing banking services: a hybrid transaction classification and cash flow prediction approach. *Journal of Big Data* 9(1). doi: 10.1186/s40537-022-00651-x.

Kourentzes, N., Petropoulos, F. and Trapero, J.R. 2014. Improving forecasting by estimating time series structural components across multiple frequencies. *International Journal of Forecasting* 30(2), pp. 291–302. doi: 10.1016/j.ijforecast.2013.09.006.

Kriebel, J. and Stitz, L. 2022. Credit default prediction from user-generated text in peer-to-peer lending using deep learning. *European Journal of Operational Research* 302(1), pp. 309–323. doi: 10.1016/j.ejor.2021.12.024.

Krumm, S., Thiel, A.M., Reznik, N., Freudenstein, J.P., Schäpers, P. and Mussel, P. 2024. Creating a Psychological Test in a Few Seconds: Can ChatGPT Develop a Psychometrically Sound Situational Judgment Test? *European Journal of Psychological Assessment*. doi: 10.1027/1015-5759/a000878.

Kumar Jha, B. and Pande, S. 2021. Time Series Forecasting Model for Supermarket Sales using FB-Prophet. In: *Proceedings - 5th International Conference on Computing Methodologies and Communication, ICCMC 2021*. Institute of Electrical and Electronics Engineers Inc., pp. 547–554. doi: 10.1109/ICCMC51019.2021.9418033.

De Lange, P.E., Melsom, B., Vennerød, C.B. and Westgaard, S. 2022. Explainable AI for Credit Assessment in Banks. *Journal of Risk and Financial Management* 15(12). doi: 10.3390/jrfm15120556.

Langsetmo, L. et al. 2023. Advantages and Disadvantages of Random Forest Models for Prediction of Hip Fracture Risk Versus Mortality Risk in the Oldest Old. *JBMR Plus* 7(8). doi: 10.1002/jbm4.10757.

Lee, B.K. and Sohn, S.Y. 2017. A credit scoring model for SMEs based on accounting ethics. *Sustainability (Switzerland)* 9(9). doi: 10.3390/su9091588.

Li, B., Xiao, B. and Yang, Y. 2021. Strengthen credit scoring system of small and micro businesses with soft information: Analysis and comparison based on neural network models. *Journal of Intelligent and Fuzzy Systems* 40(3), pp. 4257–4274. doi: 10.3233/JIFS-200866.

Li, H., Yu, L. and He, W. 2019. The Impact of GDPR on Global Technology Development. *Journal of Global Information Technology Management* 22(1), pp. 1–6. doi: 10.1080/1097198X.2019.1569186.

Li, X.-L. and Zhong, Y. 2012. An Overview of Personal Credit Scoring: Techniques and Future Work. *International Journal of Intelligence Science* 2(4), pp. 181–189. doi: 10.4236/ijis.2012.224024.

Liashchynskyi, P. and Liashchynskyi, P. 2019. Grid Search, Random Search, Genetic Algorithm: A Big Comparison for NAS. *arXiv preprint*. Available at: http://arxiv.org/abs/1912.06059 [Accessed: 6 August 2025].

Liu, F. and Demosthenes, P. 2022. Real-world data: a brief review of the methods, applications, challenges and opportunities. *BMC Medical Research Methodology* 22(1). doi: 10.1186/s12874-022-01768-6.

Livingstone, D.J., Manallack, D.T. and Tetko, I. V. 1997. Data modelling with neural networks: Advantages and limitations. *Journal of Computer-Aided Molecular Design* 11, pp. 135–142.

Lohokare, J., Dani, R. and Sontakke, S. 2017. Automated Data Collection for Credit Score Calculation based on Financial Transactions and Social Media. In: *2017 International Conference on Emerging Trends & Innovation in IT (ICEI)*. Pune, India: IEEE, pp. 134–138. doi: 10.1109/ETIICT.2017.7977024.

Lu, S., Zhang, X., Su, Y., Liu, X. and Yu, L. 2025. Efficient multimodal learning for corporate credit risk prediction with an extended deep belief network. *Annals of Operations Research*, pp. 1–38. doi: 10.1007/s10479-025-06612-w.

Luo, C., Wu, D. and Wu, D. 2017. A deep learning approach for credit scoring using credit default swaps. *Engineering Applications of Artificial Intelligence* 65, pp. 465–470. doi: 10.1016/j.engappai.2016.12.002.

Ma, Q. 2020. Comparison of ARIMA, ANN and LSTM for Stock Price Prediction. *E3S Web of Conferences* 218. doi: 10.1051/e3sconf/202021801026.
Marcot, B.G. and Hanea, A.M. 2021. What is an optimal value of k in k-fold cross-validation in discrete Bayesian network analysis? *Computational Statistics* 36(3), pp. 2009–2031. doi: 10.1007/s00180-020-00999-9.

Masyutin, A.A. 2015. Credit Scoring Based on Social Network Data. *Business Informatics* 3(33), pp. 15–23.

Mester, L.J., Nakamura, L.I. and Renault, M. 2007. Transactions accounts and loan monitoring. *Review of Financial Studies* 20(3), pp. 528–556. doi: 10.1093/rfs/hhl018.

Muhammad, A.B., Olawoyin, I.O., Yahaya, A., Gulumbe, S.U., Muhammad, Abdullahi.A. and Salisu, I.A. 2024. Credit Risk Analysis: An Assessment of the Performance of Six Machine Learning Techniques in Credit Scoring Modelling.

Fudma Journal of Sciences 8(6), pp. 163–173. Available at: https://fjs.fudutsinma.edu.ng/index.php/fjs/article/view/2893.

NayaOne. 2025a. *Synthetic UK Open Banking Transactions*. Available at: https://platform.fcainnovation.co.uk/datasets/357/description [Accessed: 22 July 2025].

NayaOne. 2025b. *Synthetic UK SME Loans*. Available at: https://digitalsandbox.fcainnovation.co.uk/datasets/539/description [Accessed: 16 June 2025].

NayaOne's Digital Twin. 2025. *Synthetic UK Business Current Accounts*. Available at: https://digitalsandbox.fcainnovation.co.uk/datasets/137/description [Accessed: 16 June 2025].

Nguyen, C. V., Das, S.R., He, J., Yue, S., Hanumaiah, V., Ragot, X. and Zhang, L. 2021. Multimodal machine learning for credit modeling. In: *Proceedings - 2021 IEEE 45th Annual Computers, Software, and Applications Conference, COMPSAC 2021*. Institute of Electrical and Electronics Engineers Inc., pp. 1754–1759. doi: 10.1109/COMPSAC51774.2021.00262.

Nielsen, D. 2016. *Tree Boosting With XGBoost Why Does XGBoost Win 'Every' Machine Learning Competition?* Master's, NTNU.

Ning, Y., Kazemi, H. and Tahmasebi, P. 2022. A comparative machine learning study for time series oil production forecasting: ARIMA, LSTM, and Prophet. *Computers and Geosciences* 164. doi: 10.1016/j.cageo.2022.105126.

Óskarsdóttir, M., Bravo, C., Sarraute, C., Vanthienen, J. and Baesens, B. 2019. The value of big data for credit scoring: Enhancing financial inclusion using mobile phone data and social network analytics. *Applied Soft Computing Journal* 74, pp. 26–39. doi: 10.1016/j.asoc.2018.10.004.

Ots, H., Liiv, I. and Tuur, D. 2020. Mobile Phone Usage Data for Credit Scoring. In: *Databases and Information Systems (D&IS) 2020*. Communications in Computer and Information Science. Tallinn, Estonia: Springer International Publishing, pp. 82–95. Available at: https://link.springer.com/10.1007/978-3-030-57672-1 [Accessed: 10 June 2025].

Putra, S.G.P., Joshi, B., Redi, J. and Bozzon, A. 2020. A Credit Scoring Model for SMEs Based on Social Media Data. In: Bielikova, M., Mikkonen, T., and Pautasso, C. eds. *Web Engineering*. Lecture Notes in Computer Science. Helsinki, Finland: Springer International Publishing, pp. 113–129. Available at: http://link.springer.com/10.1007/978-3-030-50578-3.

Roy, P.K. and Shaw, K. 2021. A multicriteria credit scoring model for SMEs using hybrid BWM and TOPSIS. *Financial Innovation* 7(1). doi: 10.1186/s40854-021-00295-5.

Saito, T. and Rehmsmeier, M. 2015. The precision-recall plot is more informative than the ROC plot when evaluating binary classifiers on imbalanced datasets. *PLoS ONE* 10(3). doi: 10.1371/journal.pone.0118432.

Salman, H.A., Kalakech, A. and Steiti, A. 2024. Random Forest Algorithm Overview. *Babylonian Journal of Machine Learning* 2024, pp. 69–79. doi: 10.58496/bjml/2024/007.

Sawhney, R., Mathur, P., Mangal, A., Khanna, P., Shah, R.R. and Zimmermann, R. 2020. Multimodal Multi-Task Financial Risk Forecasting. In: *MM 2020 - Proceedings of the 28th ACM International Conference on Multimedia*. Association for Computing Machinery, Inc, pp. 456–465. doi: 10.1145/3394171.3413752.

Shu, M., Liang, J. and Zhu, C. 2024. Automated Risk Factor Extraction from Unstructured Loan Documents: An NLP Approach to Credit Default Prediction. *Artificial Intelligence and Machine Learning Review* 5, pp. 10–24. doi: 10.69987/AIMLR.2024.50202.

Siami-Namini, S., Tavakoli, N. and Siami Namin, A. 2018. A Comparison of ARIMA and LSTM in Forecasting Time Series. In: *Proceedings - 17th IEEE International Conference on Machine Learning and Applications, ICMLA 2018*. Institute of Electrical and Electronics Engineers Inc., pp. 1394–1401. doi: 10.1109/ICMLA.2018.00227.

Sperandei, S. 2014. Understanding logistic regression analysis. *Biochemia Medica* 24(1), pp. 12–18. doi: 10.11613/BM.2014.003.

Stoltzfus, J.C. 2011. Logistic regression: A brief primer. *Academic Emergency Medicine* 18(10), pp. 1099–1104. doi: 10.1111/j.1553-2712.2011.01185.x.

Talaat, F.M., Aljadani, A., Badawy, M. and Elhosseini, M. 2024. Toward interpretable credit scoring: integrating explainable artificial intelligence with deep learning for credit card default prediction. *Neural Computing and Applications* 36(9), pp. 4847–4865. doi: 10.1007/s00521-023-09232-2.

Thabtah, F., Hammoud, S., Kamalov, F. and Gonsalves, A. 2020. Data imbalance in classification: Experimental evaluation. *Information Sciences* 513, pp. 429–441. doi: 10.1016/j.ins.2019.11.004.

To, J. and Panadero, E. 2019. Peer assessment effects on the self-assessment process of first-year undergraduates. *Assessment and Evaluation in Higher Education* 44(6), pp. 920–932. doi: 10.1080/02602938.2018.1548559.

Trivedi, S.K. 2020. A study on credit scoring modeling with different feature selection and machine learning approaches. *Technology in Society* 63. doi: 10.1016/j.techsoc.2020.101413.

Tsaih, R., Liu, Y.J., Liu, W. and Lien, Y.L. 2004. Credit scoring system for small business loans. *Decision Support Systems* 38(1), pp. 91–99. doi: 10.1016/S0167-9236(03)00079-4.

VantageScore. 2023. *Credit Scoring and Financial Inclusion - Research Study*. Available at: https://www.vantagescore.com/resources/knowledge-center/credit-scoring-and-financial-inclusion-research-study [Accessed: 12 April 2025].

Verkade, W.J. 2018. *Credit scoring for small medium enterprises using transaction data*. Master's, Delft Institute of Applied Mathematics.

Wang, C. and Xiao, Z. 2022. A Deep Learning Approach for Credit Scoring Using Feature Embedded Transformer. *Applied Sciences (Switzerland)* 12(21), p. 10995. doi: 10.3390/app122110995.

Wang, W., Lesner, C., Ran, A., Rukonic, M., Xue, J. and Shiu, E. 2020. Using Small Business Banking Data for Explainable Credit Risk Scoring. *Proceedings of the AAAI Conference on Artificial Intelligence* 34(8), pp. 13396–13401. doi: https://doi.org/10.1609/aaai.v34i08.7055.

Wang, X., Wu, M., Li, Z. and Chan, C. 2008. Short time-series microarray analysis: Methods and challenges. *BMC Systems Biology* 2. doi: 10.1186/1752-0509-2-58.
Willmott, C.J. and Matsuura, K. 2005. Advantages of the mean absolute error (MAE) over the root mean square error (RMSE) in assessing average model performance. *Climate Research* 30, pp. 79–82. doi: https://doi.org/10.3354/cr030079.

Wu, J., Li, Y. and Ma, Y. 2021. Comparison of XGBoost and the Neural Network model on the class-balanced datasets. In: *2021 IEEE 3rd International Conference on Frontiers Technology of Information and Computer, ICFTIC 2021*. Institute of Electrical and Electronics Engineers Inc., pp. 457–461. doi: 10.1109/ICFTIC54370.2021.9647373.

Yadav, S. and Shukla, S. 2016. Analysis of k-Fold Cross-Validation over Hold-Out Validation on Colossal Datasets for Quality Classification. In: *Proceedings - 6th International Advanced Computing Conference, IACC 2016*. Institute of Electrical and Electronics Engineers Inc., pp. 78–83. doi: 10.1109/IACC.2016.25.

Yan, Y. and Li, B. 2023. The Research in Credit Risk of Micro and Small Companies with Linear Regression Model. In: Tan, Y., Shi, Y., and Luo, W. eds. *International Conference in Swarm Intelligence (ICSI) 2023*. Lecture Notes in Computer Science. Springer Nature Switzerland, pp. 490–500. Available at: https://link.springer.com/10.1007/978-3-031-36625-3.

Yu, L., Bai, X. and Chen, Z. 2023. GPT-LGBM: A ChatGPT-based integrated framework for credit scoring with textual and structured data. *SSRN*. Available at: https://ssrn.com/abstract=4671511.

Zhang, C.J., Chen, L., Jagadish, H. V, Cao, C.C. and Kong, H. 2013. Reducing Uncertainty of Schema Matching via Crowdsourcing. *Proceedings of the VLDB Endowment* 6(9), pp. 757–768. doi: 10.14778/2536360.2536374.

Zhang, G.P. 2007. A neural network ensemble method with jittered training data for time series forecasting. *Information Sciences* 177(23), pp. 5329–5346. doi: 10.1016/j.ins.2007.06.015.

Zhang, P., Jia, Y. and Shang, Y. 2022. Research and application of XGBoost in imbalanced data. *International Journal of Distributed Sensor Networks* 18(6). doi: 10.1177/15501329221106935.

Zhang, S. 2022. Challenges in KNN Classification. *IEEE Transactions on Knowledge and Data Engineering* 34(10), pp. 4663–4675. doi: 10.1109/TKDE.2021.3049250.

Zhu, B., Yang, W., Wang, H. and Yuan, Y. 2018. A Hybrid Deep Learning Model for Consumer Credit Scoring. In: *2018 International Conference on Artificial Intelligence and Big Data (ICAIBD)*. Chengdu, China: IEEE Press, pp. 205–208. doi: 10.1109/ICAIBD.2018.8396195.

6 Appendices

6.1 Appendix 1: Code and Dataset Assembly Instructions

The entire code submission, with datasets, can be viewed and downloaded here: https://drive.google.com/drive/folders/1Ukvy99RDb_EWhwUjzaXntHgEKQn765U8?usp=sharing

The original datasets are contained within the folder entitled "Original Datasets". The transaction dataset is entitled "synthetic_ob_transactions.csv", the current accounts dataset is entitled "synthetic_uk_business_current_accounts.csv", and the SME loans dataset entitled in "sme_loans_data.csv". The post-processing datasets are contained in "Post-Processing Datasets", the errors and forecasts CSVs from the forecasting models are in "Errors and Forecasts CSVs", and the results CSVs are in "Results CSVs".

Prior to running any forecasting models, the code for generating the boosted synthetic transaction dataset, entitled "Generating Synthetic Transaction Dataset.py", needs to be executed. This will take the original transaction data, boost it to add 800 new accounts with corresponding transactions, and write the boosted dataset to a CSV, entitled "Boosted Transaction Dataset.csv". This file will be needed to run all forecasting models on the whole dataset.

Prior to running the forecasting models for the longer subset, the code for generating the longer subset, entitled "Generating Subset of Longer Accounts from Synthetic Data (14 Months).py", needs to be executed. This will take the boosted transaction data, filter it to only contain accounts with 14 or more months of transaction data, and write the longer subset to a CSV, entitled "Longer Subset of Transaction Data (14 Months).csv". This file will be needed to run all forecasting models on the longer subset.

Prior to running the SSA models, the Python class "mySSA.py" will be needed within the Google Colab environment. The file can be found in the code folder on Google Drive, or alternatively, it can be downloaded from the original GitHub source: GitHub - aj-cloete/pssa: Singular Spectrum Analysis for time series forecasting in Python

To run the LSTM models, accounts needed to be processed in batches of 150 so as not to violate Google Colab's memory requirements. Each batch is shown as a separate code block in the notebook. Run all cells prior to any of the batches before running the batches themselves but only run the code generating the UniqueAccounts set and corresponding "UniqueAccounts.json" file before running the first batch (this is to maintain the same order of accounts between batches and ensure no duplicate account runs). After one batch has been run, press "Restart session" in Google Colab and re-execute all cells prior to the LSTM batches apart from the code generating the account list set and JSON file, as explained above. Only run the block for the batch you wish to run (e.g. run only accounts 0-150 first before restarting, then restart session and run only accounts 150-300 and so on). This is to prevent the running of all accounts from exceeding the Google Colab free tier's memory limits.

When running each forecasting model's notebook, two CSV files will be generated: "[Model Name] Errors.csv" and "[Model Name] Forecasts.csv". The errors files are used to generate the error rates shown in the final code block of the Google Colab notebook, and the forecasts files should be downloaded, as they are needed later for the data pairing prior to use in classification. Due to them being the strongest performers in each case, the naïve forecasts were used for cumulative forecasting and the SSA forecasts were used for month-by-month forecasting, so these are the two necessary files. Only the whole dataset forecast files were needed for classification, however.

Before running the classification models, the file pairing the three disparate datasets, entitled "Overall Data Pairing [Assumption Set].py" needs to be executed for each assumption set. This performs the pairing according to the outlined assumptions for each assumption set, and generates a file entitled "Paired Dataset [Assumption Set].csv" for each assumption set. These files will be needed to run the classification models.

6.2 Appendix 2: Code Files

6.2.1 Appendix 2.1: Python Classes (.py files)

These files are Python classes (.py files). These can be executed in any native Python environment.

6.2.1.1 Appendix 2.1.1: mySSA.py (sourced from GitHub)

```
import numpy as np
import pandas as pd
from numpy import matrix as m
from pandas import DataFrame as df
from scipy import linalg
try:
   import seaborn
except:
   pass
from matplotlib.pylab import rcParams
rcParams['figure.figsize'] = 11, 4

class mySSA(object):
   '''Singular Spectrum Analysis object'''
   def __init__(self, time_series):

      self.ts = pd.DataFrame(time_series)
      self.ts_name = self.ts.columns.tolist()[0]
      if self.ts_name==0:
         self.ts_name = 'ts'
      self.ts_v = self.ts.values
      self.ts_N = self.ts.shape[0]
      self.freq = self.ts.index.inferred_freq

   @staticmethod
   def _printer(name, *args):
      '''Helper function to print messages neatly'''
      print('-'*40)
      print(name+':')
      for msg in args:
         print(msg)

   @staticmethod
   def _dot(x,y):
      '''Alternative formulation of dot product to allow missing values in
arrays/matrices'''
      pass

   @staticmethod
   def get_contributions(X=None, s=None, plot=True):
      '''Calculate the relative contribution of each of the singular values'''
      lambdas = np.power(s,2)
```

```
        frob_norm = np.linalg.norm(X)
        ret = df(lambdas/(frob_norm**2), columns=['Contribution'])
        ret['Contribution'] = ret.Contribution.round(4)
        if plot:
            ax = ret[ret.Contribution!=0].plot.bar(legend=False)
            ax.set_xlabel("Lambda_i")
            ax.set_title('Non-zero contributions of Lambda_i')
            vals = ax.get_yticks()
            ax.set_yticklabels(['{:3.2f}%'.format(x*100) for x in vals])
            return ax
        return ret[ret.Contribution>0]

    @staticmethod
    def diagonal_averaging(hankel_matrix):
        '''Performs anti-diagonal averaging from given hankel matrix
        Returns: Pandas DataFrame object containing the reconstructed series'''
        mat = m(hankel_matrix)
        L, K = mat.shape
        L_star, K_star = min(L,K), max(L,K)
        new = np.zeros((L,K))
        if L > K:
            mat = mat.T
        ret = []

        #Diagonal Averaging
        for k in range(1-K_star, L_star):
            mask = np.eye(K_star, k=k, dtype='bool')[::-1][:L_star,:]
            mask_n = sum(sum(mask))
            ma = np.ma.masked_array(mat.A, mask=1-mask)
            ret+=[ma.sum()/mask_n]

        return df(ret).rename(columns={0:'Reconstruction'})

    def view_time_series(self):
        '''Plot the time series'''
        self.ts.plot(title='Original Time Series')

    def embed(self, embedding_dimension=None, suspected_frequency=None,
verbose=False, return_df=False):
        '''Embed the time series with embedding_dimension window size.
        Optional: suspected_frequency changes embedding_dimension such that it is
divisible by suspected frequency'''
        if not embedding_dimension:
            self.embedding_dimension = self.ts_N//2
        else:
            self.embedding_dimension = embedding_dimension
        if suspected_frequency:
            self.suspected_frequency = suspected_frequency
            self.embedding_dimension =
(self.embedding_dimension//self.suspected_frequency)*self.suspected_frequency
```

```
        self.K = self.ts_N-self.embedding_dimension+1
        self.X = m(linalg.hankel(self.ts,
np.zeros(self.embedding_dimension))).T[:,:self.K]
        self.X_df = df(self.X)
        self.X_complete = self.X_df.dropna(axis=1)
        self.X_com = m(self.X_complete.values)
        self.X_missing = self.X_df.drop(self.X_complete.columns, axis=1)
        self.X_miss = m(self.X_missing.values)
        self.trajectory_dimentions = self.X_df.shape
        self.complete_dimensions = self.X_complete.shape
        self.missing_dimensions = self.X_missing.shape
        self.no_missing = self.missing_dimensions[1]==0

        if verbose:
            msg1 = 'Embedding dimension\t:  {}\nTrajectory dimensions\t: {}'
            msg2 = 'Complete dimension\t: {}\nMissing dimension     \t: {}'
            msg1 = msg1.format(self.embedding_dimension, self.trajectory_dimentions)
            msg2 = msg2.format(self.complete_dimensions, self.missing_dimensions)
            self._printer('EMBEDDING SUMMARY', msg1, msg2)

        if return_df:
            return self.X_df

    def decompose(self, verbose=False):
        '''Perform the Singular Value Decomposition and identify the rank of the
embedding subspace
        Characteristic of projection: the proportion of variance captured in the
subspace'''
        X = self.X_com
        self.S = X*X.T
        self.U, self.s, self.V = linalg.svd(self.S)
        self.U, self.s, self.V = m(self.U), np.sqrt(self.s), m(self.V)
        self.d = np.linalg.matrix_rank(X)
        Vs, Xs, Ys, Zs = {}, {}, {}, {}
        for i in range(self.d):
            Zs[i] = self.s[i]*self.V[:,i]
            Vs[i] = X.T*(self.U[:,i]/self.s[i])
            Ys[i] = self.s[i]*self.U[:,i]
            Xs[i] = Ys[i]*(m(Vs[i]).T)
        self.Vs, self.Xs = Vs, Xs
        self.s_contributions = self.get_contributions(X, self.s, False)
        self.r = len(self.s_contributions[self.s_contributions>0])
        self.r_characteristic = round((self.s[:self.r]**2).sum()/(self.s**2).sum(),4)
        self.orthonormal_base = {i:self.U[:,i] for i in range(self.r)}

        if verbose:
            msg1 = 'Rank of trajectory\t\t: {}\nDimension of projection space\t: {}'
            msg1 = msg1.format(self.d, self.r)
            msg2 = 'Characteristic of projection\t: {}'.format(self.r_characteristic)
```

```
        self._printer('DECOMPOSITION SUMMARY', msg1, msg2)

    def view_s_contributions(self, adjust_scale=False, cumulative=False,
return_df=False):
        '''View the contribution to variance of each singular value and its corresponding
signal'''
        contribs = self.s_contributions.copy()
        contribs = contribs[contribs.Contribution!=0]
        if cumulative:
            contribs['Contribution'] = contribs.Contribution.cumsum()
        if adjust_scale:
            contribs = (1/contribs).max()*1.1-(1/contribs)
        ax = contribs.plot.bar(legend=False)
        ax.set_xlabel("Singular_i")
        ax.set_title('Non-zero{} contribution of Singular_i {}'.\
                format(' cumulative' if cumulative else '', '(scaled)' if adjust_scale else
''))
        if adjust_scale:
            ax.axes.get_yaxis().set_visible(False)
        vals = ax.get_yticks()
        ax.set_yticklabels(['{:3.0f}%'.format(x*100) for x in vals])
        if return_df:
            return contribs

    @classmethod
    def view_reconstruction(cls, *hankel, names=None, return_df=False, plot=True,
symmetric_plots=False):
        '''Visualise the reconstruction of the hankel matrix/matrices passed to *hankel'''
        hankel_mat = None
        for han in hankel:
            if isinstance(hankel_mat,m):
                hankel_mat = hankel_mat + han
            else:
                hankel_mat = han.copy()
        hankel_full = cls.diagonal_averaging(hankel_mat)
        title = 'Reconstruction of signal'
        if names or names==0:
            title += ' associated with singular value{}: {}'
            title = title.format('' if len(str(names))==1 else 's', names)
        if plot:
            ax = hankel_full.plot(legend=False, title=title)
            if symmetric_plots:
                velocity = hankel_full.abs().max()[0]
                ax.set_ylim(bottom=-velocity, top=velocity)
        if return_df:
            return hankel_full

    def _forecast_prep(self, singular_values=None):
        self.X_com_hat = np.zeros(self.complete_dimensions)
        self.verticality_coefficient = 0
```

```
        self.forecast_orthonormal_base = {}
        if singular_values:
            try:
                for i in singular_values:
                    self.forecast_orthonormal_base[i] = self.orthonormal_base[i]
            except:
                if singular_values==0:
                    self.forecast_orthonormal_base[0] = self.orthonormal_base[0]
                else:
                    raise('Please pass in a list/array of singular value indices to use for
forecast')
        else:
            self.forecast_orthonormal_base = self.orthonormal_base
        self.R = np.zeros(self.forecast_orthonormal_base[0].shape)[:-1]
        for Pi in self.forecast_orthonormal_base.values():
            self.X_com_hat += Pi*Pi.T*self.X_com
            pi = np.ravel(Pi)[-1]
            self.verticality_coefficient += pi**2
            self.R += pi*Pi[:-1]
        self.R = m(self.R/(1-self.verticality_coefficient))
        self.X_com_tilde = self.diagonal_averaging(self.X_com_hat)

    def forecast_recurrent(self, steps_ahead=12, singular_values=None, plot=False,
return_df=False, **plotargs):
        '''Forecast from last point of original time series up to steps_ahead using
recurrent methodology
        This method also fills any missing data from the original time series.'''
        try:
            self.X_com_hat
        except(AttributeError):
            self._forecast_prep(singular_values)
        self.ts_forecast = np.array(self.ts_v[0])
        for i in range(1, self.ts_N+steps_ahead):
            try:
                if np.isnan(self.ts_v[i]):
                    x = self.R.T*m(self.ts_forecast[max(0,i-self.R.shape[0]): i]).T
                    self.ts_forecast = np.append(self.ts_forecast,x[0])
                else:
                    self.ts_forecast = np.append(self.ts_forecast,self.ts_v[i])
            except(IndexError):
                x = self.R.T*m(self.ts_forecast[i-self.R.shape[0]: i]).T
                self.ts_forecast = np.append(self.ts_forecast, x[0])
        self.forecast_N = i+1
        new_index = pd.date_range(start=self.ts.index.min(),periods=self.forecast_N,
freq=self.freq)
        forecast_df = df(self.ts_forecast, columns=['Forecast'], index=new_index)
        forecast_df['Original'] = np.append(self.ts_v, [np.nan]*steps_ahead)
        if plot:
            forecast_df.plot(title='Forecasted vs. original time series', **plotargs)
        if return_df:
```

```
        return forecast_df

if __name__=='__main__':
    from mySSA import mySSA
    from pandas import DataFrame as df
    import pandas as pd
    import numpy as np
    from matplotlib.pylab import rcParams

    # Construct the data with gaps
    ts = pd.read_csv('AirPassengers.csv', parse_dates=True, index_col='Month')
    ts_ = ts.copy()
    ts_.ix[67:79] = np.nan
    ts_ = ts_.set_value('1961-12-01','#Passengers', np.nan).asfreq('MS')
    ssa = mySSA(ts_)

    # Plot original series for reference
    ssa.view_time_series()

    ssa.embed(embedding_dimension=36, suspected_frequency=12, verbose=True)
    ssa.decompose(True)
    ssa.view_s_contributions(adjust_scale=True)

    # Component Signals
    components = [i for i in range(13)]
    rcParams['figure.figsize'] = 11, 2
    for i in range(5):
        ssa.view_reconstruction(ssa.Xs[i], names=i, symmetric_plots=i!=0)
    rcParams['figure.figsize'] = 11, 4

    # RECONSTRUCTION
    ssa.view_reconstruction(*[ssa.Xs[i] for i in components], names=components)

    # FORECASTING
    ssa.forecast_recurrent(steps_ahead=48, plot=True)
```

6.2.1.2 Appendix 2.1.2: Checking Number of Transactions per Account.py

```
#Importing libraries
import pandas as pd
from datetime import datetime
from dateutil import parser
import matplotlib.pyplot as plt

#Reading in the data
Data = pd.read_csv("synthetic_ob_transactions.csv")

#Visualising number of transactions per account in a boxplot
BusinessAccountAmounts = Data["AccountId"].value_counts()
plt.boxplot(BusinessAccountAmounts)
plt.title("Distribution of Number of Transactions per Account")
plt.ylabel("Number of Transactions")
plt.show()
```

6.2.1.3 Appendix 2.1.3: Checking Number of Months per Account.py

```
#Importing libraries
import uuid
from sdv.single_table import GaussianCopulaSynthesizer
from sdv.metadata import Metadata
import numpy as np
import pandas as pd
from datetime import datetime
from dateutil import parser
from sklearn.metrics.pairwise import cosine_similarity
from sklearn.preprocessing import MinMaxScaler
import matplotlib.pyplot as plt

#Reading in the data and selecting relevant columns
TransactionData = pd.read_csv("synthetic_ob_transactions.csv")
TransactionData = TransactionData[["TransactionDate", "TransactionType",
"Amount", "AccountId"]]
print(TransactionData)

#Converting transaction dates into months
TransactionDates = TransactionData["TransactionDate"]

ParsedTransactionDates = [parser.parse(date) for date in TransactionDates]

FormattedTransactionMonths = [date.strftime("%m/%Y") for date in
ParsedTransactionDates]

TransactionData["TransactionDate"] = FormattedTransactionMonths

#Checking the number of months per account
NumberOfMonthsPerAccount =
TransactionData.groupby(by=["AccountId"])["TransactionDate"].nunique().sort_val
ues(ascending=False)
print(NumberOfMonthsPerAccount)
NumberOfMonthsPerAccount = NumberOfMonthsPerAccount.reset_index()

#Visualising the distribution of number of months per account
BusinessMonthAmounts = NumberOfMonthsPerAccount["TransactionDate"]
plt.boxplot(BusinessMonthAmounts)
plt.title("Distribution of Number of Months per Account")
plt.ylabel("Number of Months")
plt.show()
```

6.2.1.4 Appendix 2.1.4: Generating Synthetic Transaction Dataset.py

```
#Importing libraries
import uuid
from sdv.single_table import GaussianCopulaSynthesizer
from sdv.metadata import Metadata
import numpy as np
import pandas as pd
from datetime import datetime
from dateutil import parser
from sklearn.metrics.pairwise import cosine_similarity
from sklearn.preprocessing import MinMaxScaler
import matplotlib.pyplot as plt

#Reading in the data and selecting relevant columns
TransactionData = pd.read_csv("synthetic_ob_transactions.csv")
TransactionData = TransactionData[["TransactionDate", "TransactionType",
"Amount", "AccountId"]]

#Selecting transaction dates for parsing
TransactionDates = TransactionData["TransactionDate"]

#Reformatting transaction dates as months
ParsedTransactionDates = [parser.parse(date) for date in TransactionDates]

FormattedTransactionMonths = [date.strftime("%m/%Y") for date in
ParsedTransactionDates]

TransactionData["TransactionDate"] = FormattedTransactionMonths

#Checking the number of unique accounts
UniqueTransactions =
TransactionData['AccountId'].drop_duplicates().reset_index(drop=True)

print(len(UniqueTransactions))

#Generating the distribution of number of transactions per account
AccountTransactionAmounts = TransactionData["AccountId"].value_counts()
AccountDistribution = AccountTransactionAmounts.value_counts().sort_index()
print(AccountDistribution)

#Checking the distribution of number of months per account
NumberOfMonthsPerAccount =
TransactionData.groupby(by=["AccountId"])["TransactionDate"].nunique().sort_val
ues(ascending=False)
print(NumberOfMonthsPerAccount)
NumberOfMonthsPerAccount = NumberOfMonthsPerAccount.reset_index()

#Visualising the distribution of number of months per account
BusinessMonthAmounts = NumberOfMonthsPerAccount["TransactionDate"]
plt.boxplot(BusinessMonthAmounts)
```

```
plt.title("Distribution of Number of Months per Account")
plt.ylabel("Number of Months")
plt.show()

#Removing all accounts that have less than 3 months of data to ensure that all
models receive an adequate training set
for i in range(0, len(NumberOfMonthsPerAccount)):
    if NumberOfMonthsPerAccount.iloc[i]["TransactionDate"] < 3:
        TransactionData = TransactionData[TransactionData["AccountId"] !=
NumberOfMonthsPerAccount.iloc[i]["AccountId"]]

#Checking the updated number of unique accounts
UpdatedUniqueTransactions =
TransactionData['AccountId'].drop_duplicates().reset_index(drop=True)

print(len(UpdatedUniqueTransactions))

#Checking the updated transaction amount distribution
AccountTransactionAmounts = TransactionData["AccountId"].value_counts()
AccountDistribution = AccountTransactionAmounts.value_counts().sort_index()
print(AccountDistribution)

#Detecting metadata from existing transaction dataset
TransactionsMetadata = Metadata.detect_from_dataframe(data=TransactionData,
table_name="Transactions")
TransactionsMetadata.update_column(column_name='AccountId',
sdtype='categorical')

#Fitting the synthetic data generator to the metadata from the original dataset
SyntheticDataGenerator = GaussianCopulaSynthesizer(TransactionsMetadata)
SyntheticDataGenerator.fit(TransactionData)

#Declaring the number of new accounts to be created and the number of transactions
needed to parallel the original dataset
NumberOfNewAccounts = 800
np.random.seed(271)
TransactionCounts = np.random.choice(
    AccountDistribution.index,
    size=NumberOfNewAccounts,
    p=AccountDistribution.values/AccountDistribution.sum())

#Calculating the required number of rows
TotalRequiredRows = TransactionCounts.sum()
print(TotalRequiredRows)

#Generating synthetic data
Sample = SyntheticDataGenerator.sample(num_rows=int(TotalRequiredRows))

#Assigning account IDs to transactions in the synthetic data
NewAccountIDs = [str(uuid.uuid4()) for _ in range(NumberOfNewAccounts)]
```

```
AccountIDsRepeated = [AccountID for AccountID, Count in zip(NewAccountIDs,
TransactionCounts) for _ in range(Count)]

AccountIDsRepeated = AccountIDsRepeated[:len(Sample)]
Sample['AccountId'] = AccountIDsRepeated

#Ensuring that no transaction misses an account ID and summarising the amount of
new data generated
assert Sample['AccountId'].isna().sum() == 0
print(f"Total synthetic transactions: {len(Sample)}")
print(f"Unique synthetic accounts: {Sample['AccountId'].nunique()}")

#Combining the synthetic data with the original data and calculating the new amount
of unique accounts
BoostedTransactions = pd.concat([TransactionData, Sample])
UniqueBoostedTransactions =
BoostedTransactions['AccountId'].drop_duplicates().reset_index(drop=True)
print(len(UniqueBoostedTransactions))

#Checking the distribution of transaction amounts per account in the new combined
dataset
print(BoostedTransactions)
BoostedTransactionAmounts = BoostedTransactions["AccountId"].value_counts()
BoostedAccountDistribution =
BoostedTransactionAmounts.value_counts().sort_index()

#Converting any transactions labelled "Debit" into negative transactions, as Debit
denotes money leaving the account
BoostedTransactions.loc[
    BoostedTransactions["TransactionType"].str.strip() == "Debit",
    "Amount"
] *= -1

#Ensuring that all transaction dates in the boosted dataset are parsed as months
TransactionDates = BoostedTransactions["TransactionDate"]

ParsedTransactionDates = [parser.parse(date) for date in TransactionDates]

FormattedTransactionMonths = [date.strftime("%m/%Y") for date in
ParsedTransactionDates]

BoostedTransactions["TransactionDate"] = FormattedTransactionMonths

BoostedTransactions["TransactionDate"] =
pd.to_datetime(BoostedTransactions["TransactionDate"])
BoostedTransactions["TransactionDate"] =
BoostedTransactions["TransactionDate"].dt.to_period("M").dt.to_timestamp()

#Checking grouped number of transactions by account and month as a test
```

```
GroupedNumberOfTransactionsByAccountAndMonth =
BoostedTransactions.groupby(by=["AccountId", "TransactionDate"]).count()

GroupedNumberOfTransactionsByAccountAndMonth =
GroupedNumberOfTransactionsByAccountAndMonth.sort_values(by=["AccountId"
, "TransactionDate"])

#Checking that the distributions parallel each other using cosine similarity and
showing the length of the boosted dataset
Similarity = cosine_similarity([AccountDistribution],
[BoostedAccountDistribution])[0][0]
Correlation = np.corrcoef(AccountDistribution, BoostedAccountDistribution)[0, 1]
print(Similarity)
print(Correlation)
print(len(BoostedTransactions))

#Checking the number of months per account distribution in the boosted dataset
NumberOfMonthsPerAccount =
BoostedTransactions.groupby(by=["AccountId"])["TransactionDate"].nunique().sort
_values(ascending=False)
print(NumberOfMonthsPerAccount)
NumberOfMonthsPerAccount = NumberOfMonthsPerAccount.reset_index()

#Removing any synthetically generated accounts with less than 3 months of data to
ensure that every model has sufficient training data
for i in range(0, len(NumberOfMonthsPerAccount)):
    if NumberOfMonthsPerAccount.iloc[i]["TransactionDate"] < 3:
        BoostedTransactions =
BoostedTransactions[BoostedTransactions["AccountId"] !=
NumberOfMonthsPerAccount.iloc[i]["AccountId"]]

#Rechecking the number of months per account distribution after this
NumberOfMonthsPerAccount =
BoostedTransactions.groupby(by=["AccountId"])["TransactionDate"].nunique().sort
_values(ascending=False)
print(NumberOfMonthsPerAccount)

#Showing the final dataset length and number of unique accounts
print(len(BoostedTransactions))
print(BoostedTransactions["AccountId"].nunique())

#Using cosine similarity and correlation to check that the final synthetic and original
distributions parallel each other
BoostedTransactionAmounts = BoostedTransactions["AccountId"].value_counts()
BoostedAccountDistribution =
BoostedTransactionAmounts.value_counts().sort_index()
Similarity = cosine_similarity([AccountDistribution],
[BoostedAccountDistribution])[0][0]
Correlation = np.corrcoef(AccountDistribution, BoostedAccountDistribution)[0, 1]
print(Similarity)
```

```
print(Correlation)

#Writing the boosted dataset to a CSV
BoostedTransactions.to_csv("Boosted Transaction Dataset.csv")
```

6.2.1.5 Appendix 2.1.5: Generating Subset of Longer Accounts from Synthetic Data (14 Months).py

```
#Importing libraries
import pandas as pd
from datetime import datetime
from dateutil import parser

#Reading in the data
TransactionData = pd.read_csv("Boosted Transaction Dataset.csv")
print(TransactionData.head())

#Extracting transaction dates and converting them into months
TransactionDates = TransactionData["TransactionDate"]

ParsedTransactionDates = [parser.parse(date) for date in TransactionDates]

FormattedTransactionMonths = [date.strftime("%m/%Y") for date in
ParsedTransactionDates]

TransactionData["TransactionDate"] = FormattedTransactionMonths

print(TransactionData.head())

#Checking the number of months per account
NumberOfMonthsPerAccount =
TransactionData.groupby(by=["AccountId"])["TransactionDate"].nunique().sort_val
ues(ascending=False)
NumberOfMonthsPerAccount = NumberOfMonthsPerAccount.reset_index()
print(NumberOfMonthsPerAccount)
print(len(NumberOfMonthsPerAccount))

#Dneoting the longer subset by selecting only accounts with 14 or more months of
data
LongerAccounts =
NumberOfMonthsPerAccount[NumberOfMonthsPerAccount["TransactionDate"] >=
14]
print(LongerAccounts)
print(len(LongerAccounts))

#Generating a set of longer accounts and limiting the longer subset to only accounts
that are in that set
LongerAccountsSet = set(LongerAccounts["AccountId"])

LongerSubset =
TransactionData[TransactionData["AccountId"].isin(LongerAccountsSet)]

#Checking the lengths of the transaction dataset and the longer subset
print(len(TransactionData))
print(len(LongerSubset))
```

```
#Writing the longer subset to a CSV
LongerSubset.to_csv("Longer Subset of Transaction Data (14 Months).csv")
```

6.2.1.6 Appendix 2.1.6: Cumulative Forecast Whole Dataset Result Visualisation.py

```
#Importing libraries
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

#Reading in the results
Data = pd.read_csv("Cumulative Forecast Whole Dataset Results.csv")

#Denoting x and y values
x = np.arange(len(Data["Model"]))
y1 = Data["Mean SI"]
y2 = Data["Median SI"]

#Denoting bar width
Width = 0.35

#Visualising the results
plt.bar(x - Width/2, y1, Width, label="Mean SI")
plt.bar(x + Width/2, y2, Width, label="Median SI")
plt.xticks(x, Data["Model"])
plt.xlabel("Model")
plt.ylabel("Scatter Index (%)")
plt.title("Results of Cumulative Forecasts for Overall Dataset")
plt.legend()
plt.show()
```

6.2.1.7 Appendix 2.1.7: Month-by-Month Forecast Whole Dataset Result Visualisation.py

```
#Importing libraries
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

#Reading in the data
Data = pd.read_csv("Month-by-Month Forecast Whole Dataset Results.csv")

#Denoting x and y values
x = np.arange(len(Data["Model"]))
y1 = Data["Mean SI"]
y2 = Data["Median SI"]

#Declaring bar width
Width = 0.35

#Visualising the forecast results
plt.bar(x - Width/2, y1, Width, label="Mean SI")
plt.bar(x + Width/2, y2, Width, label="Median SI")
plt.xticks(x, Data["Model"])
plt.xlabel("Model")
plt.ylabel("Scatter Index (%)")
plt.title("Results of Month-by-Month Forecasts for Overall Dataset")
plt.legend()
plt.show()
```

6.2.1.8 Appendix 2.1.8: Cumulative Forecast Longer Subset Result Visualisation.py

```
#Importing libraries
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

#Reading in the data
Data = pd.read_csv("Cumulative Forecast Longer Subset Results.csv")

#Declaring x and y columns
x = np.arange(len(Data["Model"]))
y1 = Data["Mean SI"]
y2 = Data["Median SI"]

#Declaring bar width
Width = 0.35

#Visualising forecast results
plt.bar(x - Width/2, y1, Width, label="Mean SI")
plt.bar(x + Width/2, y2, Width, label="Median SI")
plt.xticks(x, Data["Model"])
plt.xlabel("Model")
plt.ylabel("Scatter Index (%)")
plt.title("Results of Cumulative Forecasts for Longer Subset")
plt.legend()
plt.show()
```

6.2.1.9 Appendix 2.1.9: Month-by-Month Forecast Longer Subset Result Visualisation.py

```
#Importing libraries
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

#Reading in the data
Data = pd.read_csv("Month-by-Month Forecast Longer Subset Results.csv")

#Declaring x and y columns
x = np.arange(len(Data["Model"]))
y1 = Data["Mean SI"]
y2 = Data["Median SI"]

#Declaring bar width
Width = 0.35

#Visualising the results
plt.bar(x - Width/2, y1, Width, label="Mean SI")
plt.bar(x + Width/2, y2, Width, label="Median SI")
plt.xticks(x, Data["Model"])
plt.xlabel("Model")
plt.ylabel("Scatter Index (%)")
plt.title("Results of Month-by-Month Forecasts for Longer Subset")
plt.legend()
plt.show()
```

6.2.1.10 Appendix 2.1.10: Comparison of Growth Rates between Best and Worst Performers (Month-by-Month).py

```
#Importing libraries
import pandas as pd
import numpy as np
from scipy.stats import ttest_ind

#Reading in the data and extracting account IDs
BoostedTransactions = pd.read_csv("Boosted Transaction Dataset.csv")
AccountIDs = BoostedTransactions["AccountId"].values

#Converting transaction dates into months
BoostedTransactions["TransactionDate"] =
pd.to_datetime(BoostedTransactions["TransactionDate"])
BoostedTransactions["TransactionDate"] =
BoostedTransactions["TransactionDate"].dt.to_period("M").dt.to_timestamp()

#Grouping transactions by account and month and summating the net transaction
amount per grouping
GroupedSumOfTransactionsByAccountAndMonth =
BoostedTransactions.groupby(by=["AccountId",
"TransactionDate"]).agg(["sum"])["Amount"]

#Declaring the 10 poorest account IDs and creating a list to store their recent growth
percentage figures
PoorAccountIDs = ["16443", "16429", "16424", "78db5423-a74c-44ba-b1db-
26f8b3e24014", "d3886a65-04e5-4575-b798-7e217cd7d3c6",
            "2aaef9e1-6293-4f85-97cf-dc9ef8d6ae19", "1186c2fd-7a00-4f21-956f-
8c5a896d9111", "16461", "43aaf6de-65e5-4d44-876d-c9bb99722e59",
            "ef9951b1-7a6c-450b-bbcd-eef377be520d"]
PoorGrowthPercentages = []

GroupedSumOfTransactionsByAccountAndMonth =
GroupedSumOfTransactionsByAccountAndMonth.reset_index()

#For each poor account, the data is split into training and testing sets
#The final 3 percentage growth rates are then taken, averaged and put into the
percentages list
for i in PoorAccountIDs:
    AccountID = i
    GrowthPercentages = []

    ForecastingTestTimeSeries =
GroupedSumOfTransactionsByAccountAndMonth[GroupedSumOfTransactionsBy
AccountAndMonth["AccountId"] == AccountID][["TransactionDate", "sum"]]

    ForecastingTestTimeSeries =
ForecastingTestTimeSeries.set_index("TransactionDate")

    ForecastingTestTimeSeries = ForecastingTestTimeSeries.resample("MS").sum()
```

```
    TrainEnd = int(0.7 * len(ForecastingTestTimeSeries))

    TrainData = ForecastingTestTimeSeries[:TrainEnd]
    TestData = ForecastingTestTimeSeries[TrainEnd:]

    KeyIndices = TrainData[-4:]

    Mean = abs(np.mean(KeyIndices))
    for j in range(0, len(KeyIndices)):
        if j == 0:
            pass
        else:
            PreviousRecord = KeyIndices.iloc[j-1]
            CurrentRecord = KeyIndices.iloc[j]
            Change = CurrentRecord - PreviousRecord
            Percentage = abs((Change/PreviousRecord)*100)
            GrowthPercentages.append(Percentage)

    MeanPercentage = np.mean(GrowthPercentages)
    PoorGrowthPercentages.append(MeanPercentage)

print(PoorGrowthPercentages)

#Declaring the good performers and creating a list to store their recent growth
percentages
GoodAccountIDs = ["4b171653-23c9-4f55-924a-46e51bf11ee4", "196d06cf-e811-
4aab-9649-ab6642551b9b", "eaccaa73-b149-48ae-a224-2f231f46432c",
                  "ee64480e-4c31-4c7c-9858-91cd2ae43e06", "0734342c-6b9a-42f8-834c-
08a98d60e98c", "77303cad-e32f-4568-9482-552625c1bf54",
                  "e8ebaf0f-ea95-46a1-ae7d-9d00d68960c9", "265754d8-dff8-4d8c-a663-
a913ecb071a9", "8f26459d-4da4-4c12-b2a1-56cb3f2559b4",
                  "6d990843-e957-4d5d-8c33-85a8026a07d0"]
GoodGrowthPercentages = []

#For each good account, the data is split into training and testing sets
#The final 3 percentage growth rates in the training set are then taken, averaged and
put into the percentages list
for i in GoodAccountIDs:
    AccountID = i
    GrowthPercentages = []

    ForecastingTestTimeSeries =
GroupedSumOfTransactionsByAccountAndMonth[GroupedSumOfTransactionsBy
AccountAndMonth["AccountId"] == AccountID][["TransactionDate", "sum"]]

    ForecastingTestTimeSeries =
ForecastingTestTimeSeries.set_index("TransactionDate")

    ForecastingTestTimeSeries = ForecastingTestTimeSeries.resample("MS").sum()
```

```
    TrainEnd = int(0.7 * len(ForecastingTestTimeSeries))

    TrainData = ForecastingTestTimeSeries[:TrainEnd]
    TestData = ForecastingTestTimeSeries[TrainEnd:]

    KeyIndices = TrainData[-4:]

    Mean = abs(np.mean(KeyIndices))
    for j in range(0, len(KeyIndices)):
        if j == 0:
            pass
        else:
            PreviousRecord = KeyIndices.iloc[j-1]
            CurrentRecord = KeyIndices.iloc[j]
            Change = CurrentRecord - PreviousRecord
            Percentage = abs((Change/PreviousRecord)*100)
            GrowthPercentages.append(Percentage)

    MeanPercentage = np.mean(GrowthPercentages)
    GoodGrowthPercentages.append(MeanPercentage)

print(GoodGrowthPercentages)

#Performing the one-tailed t-test to determine if the percentage growth rates are
higher on poorer performing accounts
TestStatistic, PValue = ttest_ind(PoorGrowthPercentages, GoodGrowthPercentages)
print("Test statistic:",TestStatistic)
print("P-Value:",(PValue/2))
```

6.2.1.11 Comparison of Growth Rate between Best and Worst Performers (Cumulative).py

```
#Importing libraries
import pandas as pd
import numpy as np
from scipy.stats import ttest_ind

#Reading in the data
BoostedTransactions = pd.read_csv("Boosted Transaction Dataset.csv")
AccountIDs = BoostedTransactions["AccountId"].values

#Converting transaction dates to months
BoostedTransactions["TransactionDate"] =
pd.to_datetime(BoostedTransactions["TransactionDate"])
BoostedTransactions["TransactionDate"] =
BoostedTransactions["TransactionDate"].dt.to_period("M").dt.to_timestamp()

#Grouping transactions by account and month and summating the net transaction
amount per grouping
GroupedSumOfTransactionsByAccountAndMonth =
BoostedTransactions.groupby(by=["AccountId",
"TransactionDate"]).agg(["sum"])["Amount"]

#Declaring a list of poorly performing accounts and a list to store their averaged
growth percentages in
PoorAccountIDs = ["997ef1f2-371c-4607-8758-54ff96ac6e02", "8ce8dbb8-1682-
491c-bd22-455fffefabed", "16424", "0a6729fb-4fe2-4987-921e-8ab6a6f289cb",
            "16429", "c3beeaa8-87ce-455d-9482-e01679b9d1eb", "b1a8e3ff-b811-
4629-b37f-c303b4094e2c", "3a98e91c-b44d-4427-ba16-8b762379db5c",
            "efb8bcab-7a10-45d9-b6b3-5a2f855e2c96", "703f718c-79ac-42e9-a793-
37e1795e7ef8"]
PoorGrowthPercentages = []

ResetDataFrame = GroupedSumOfTransactionsByAccountAndMonth.reset_index()

#For each poor account, the cumulative sum of transactions over time is calculated
#This data is then split into training and testing sets
#The final three percentage growth rates in the training set are averaged and
appended into the growth percentages list
for i in PoorAccountIDs:
    AccountID = i
    GrowthPercentages = []

    TimeSeries = ResetDataFrame[ResetDataFrame["AccountId"] == AccountID]
    TimeSeries = TimeSeries.set_index("TransactionDate")

    TimeSeries = TimeSeries.resample("MS").sum()
    for j in range(0, len(TimeSeries)):
        if TimeSeries.iloc[j]["AccountId"] == 0:
            TimeSeries.at[TimeSeries.index[j], "AccountId"] = AccountID
```

```
    NewGroupedDataset = TimeSeries
    NewGroupedDataset = NewGroupedDataset.reset_index()
    NewGroupedDataset = NewGroupedDataset.set_index(["AccountId",
"TransactionDate"])

    GroupedCumulativeSumOfTransactionsByAccountAndMonth =
NewGroupedDataset.groupby(level=0).cumsum().reset_index()

    ForecastingTestTimeSeries =
GroupedCumulativeSumOfTransactionsByAccountAndMonth[GroupedCumulative
SumOfTransactionsByAccountAndMonth["AccountId"] ==
AccountID][["TransactionDate", "sum"]]

    ForecastingTestTimeSeries =
ForecastingTestTimeSeries.set_index("TransactionDate")

    TrainEnd = int(0.7 * len(ForecastingTestTimeSeries))

    TrainData = ForecastingTestTimeSeries[:TrainEnd]
    TestData = ForecastingTestTimeSeries[TrainEnd:]

    KeyIndices = TrainData[-4:]

    Mean = abs(np.mean(KeyIndices))
    for j in range(0, len(KeyIndices)):
        if j == 0:
            pass
        else:
            PreviousRecord = KeyIndices.iloc[j-1]
            CurrentRecord = KeyIndices.iloc[j]
            Change = CurrentRecord - PreviousRecord
            Percentage = abs((Change/PreviousRecord)*100)
            GrowthPercentages.append(Percentage)

    MeanPercentage = np.mean(GrowthPercentages)
    PoorGrowthPercentages.append(MeanPercentage)

print(PoorGrowthPercentages)

#Declaring a list of well performing accounts and a list to store their averaged
growth percentages in
GoodAccountIDs = ["16434", "16453", "3fbc9c73-8f31-4391-83fd-bb0161aaff53",
"4504a897-bbdb-46c5-932b-e6109adcd825", "e3224f97-0eb1-4f42-963e-
87bb423bdab7",
                  "47255e34-7fcb-457c-aee6-fe0b48bae3c7", "2971bf3e-670d-4e90-b127-
921b7dae0f24", "c9f19b41-8752-401e-b054-12f9d14cdeb7",
                  "16450", "16c0e231-2dd6-42f6-908a-a668c98c9c21"]
GoodGrowthPercentages = []
```

```
#For each poor account, the cumulative sum of transactions over time is calculated
#This data is then split into training and testing sets
#The final three percentage growth rates in the training set are averaged and
appended into the growth percentages list
for i in GoodAccountIDs:
    AccountID = i
    GrowthPercentages = []

    TimeSeries = ResetDataFrame[ResetDataFrame["AccountId"] == AccountID]
    TimeSeries = TimeSeries.set_index("TransactionDate")

    TimeSeries = TimeSeries.resample("MS").sum()
    for j in range(0, len(TimeSeries)):
        if TimeSeries.iloc[j]["AccountId"] == 0:
          TimeSeries.at[TimeSeries.index[j], "AccountId"] = AccountID

    NewGroupedDataset = TimeSeries
    NewGroupedDataset = NewGroupedDataset.reset_index()
    NewGroupedDataset = NewGroupedDataset.set_index(["AccountId",
"TransactionDate"])

    GroupedCumulativeSumOfTransactionsByAccountAndMonth =
NewGroupedDataset.groupby(level=0).cumsum().reset_index()

    ForecastingTestTimeSeries =
GroupedCumulativeSumOfTransactionsByAccountAndMonth[GroupedCumulative
SumOfTransactionsByAccountAndMonth["AccountId"] ==
AccountID][["TransactionDate", "sum"]]

    ForecastingTestTimeSeries =
ForecastingTestTimeSeries.set_index("TransactionDate")

    TrainEnd = int(0.7 * len(ForecastingTestTimeSeries))

    TrainData = ForecastingTestTimeSeries[:TrainEnd]
    TestData = ForecastingTestTimeSeries[TrainEnd:]

    KeyIndices = TrainData[-4:]

    Mean = abs(np.mean(KeyIndices))
    for j in range(0, len(KeyIndices)):
        if j == 0:
            pass
        else:
            PreviousRecord = KeyIndices.iloc[j-1]
            CurrentRecord = KeyIndices.iloc[j]
            Change = CurrentRecord - PreviousRecord
            Percentage = abs((Change/PreviousRecord)*100)
            GrowthPercentages.append(Percentage)
```

```
    MeanPercentage = np.mean(GrowthPercentages)
    GoodGrowthPercentages.append(MeanPercentage)

print(GoodGrowthPercentages)

#Performing the one-tailed t-test to determine whether the growth rate percentages
are higher among the poor performers
TestStatistic, PValue = ttest_ind(PoorGrowthPercentages, GoodGrowthPercentages)
print("Test statistic:",TestStatistic)
print("P-Value:",(PValue/2))
```

6.2.1.12 Appendix 2.1.12: Visualisation of Repayment and Default Distribution in SME Loans Dataset.py

```
#Importing libraries
import pandas as pd
import matplotlib.pyplot as plt

#Reading in the data and splitting it into repayers and defaulters
Data = pd.read_csv("../Menna Dissertation Dataset/MSc
project/open_src_data/sme_loans/sme_loans_data.csv")
RepaidData = Data[Data["status"] == "Paid"]
DefaultedData = Data[Data["status"] == "Defaulted"]
Data = pd.concat([RepaidData, DefaultedData], axis=0)

#Visualising the class distribution in a pie chart
plt.pie(Data["status"].value_counts(), labels=["Paid", "Defaulted"],
autopct="%1.1f%%")
plt.title("Distribution of Repayment and Default in SME Loans Dataset")
plt.show()
```

6.2.1.13 Appendix 2.1.13: Overall Data Pairing (Whole Dataset, Assumption Set 3).py

```
#Importing libraries
import pandas as pd
import random
from datetime import datetime
import datetime as dt
from dateutil import parser
from scipy.spatial import KDTree
import numpy as np

#Reading in datasets and filtering loan data to only encompass repayers and
defaulters
BusinessAccountData = pd.read_csv("../Menna Dissertation Dataset/MSc
project/open_src_data/synthetic_uk_business_accounts/synthetic_uk_business_curre
nt_accounts.csv")
print(BusinessAccountData)
TransactionData = pd.read_csv("Boosted Transaction Dataset.csv")
LoanData = pd.read_csv("../Menna Dissertation Dataset/MSc
project/open_src_data/sme_loans/sme_loans_data.csv")
print(LoanData)
RepaidLoanData = LoanData[LoanData["status"] == "Paid"]
DefaultedLoanData = LoanData[LoanData["status"] == "Defaulted"]
LoanData = pd.concat([RepaidLoanData, DefaultedLoanData], axis=0,
ignore_index=True)
print(LoanData)
print(DefaultedLoanData["status"])

#Calculating the number of unique business accounts and unique transaction
accounts
UniqueBusinesses =
BusinessAccountData['current_account_number'].drop_duplicates().reset_index(dro
p=True)
UniqueTransactions =
TransactionData['AccountId'].drop_duplicates().reset_index(drop=True)

#Determining the maximum number of business-transaction account pairings that
can be made
MaximumPairs = min(len(UniqueBusinesses), len(UniqueTransactions))

#Converting all Debit transactions into negative values
for i in range(0, len(TransactionData)):
    if TransactionData.iloc[i]["TransactionType"] == "Debit":
        TransactionData.at[i, "Amount"] = -(TransactionData.at[i, "Amount"])

#Converting transaction dates into months
TransactionDates = TransactionData["TransactionDate"]

ParsedTransactionDates = [parser.parse(date) for date in TransactionDates]
```

```
FormattedTransactionMonths = [date.strftime("%m/%Y") for date in
ParsedTransactionDates]

TransactionData["TransactionDate"] = FormattedTransactionMonths

#Grouping transactions by account/month and account and calculating the net
amount for each grouping
GroupedSumOfTransactionsByAccountAndMonth =
TransactionData.groupby(by=["AccountId",
"TransactionDate"]).agg(["sum"])["Amount"]
print(GroupedSumOfTransactionsByAccountAndMonth)

GroupedSumOfTransactionsByAccount =
TransactionData.groupby(by="AccountId").agg(["sum"])["Amount"]
print(GroupedSumOfTransactionsByAccount)

GroupedSumOfTransactionsByAccount =
GroupedSumOfTransactionsByAccount.reset_index(drop=False)
print(GroupedSumOfTransactionsByAccount)

#Calculating each business account's cash flow (revenue-costs)
BusinessAccountData["Cash Flow"] = BusinessAccountData["2019_revenue"] -
BusinessAccountData["costs"]

#Calculating each business account's cash flow percentile and each transaction
amount's transaction amount percentile
BusinessAccountData["Cash Flow Percentile"] = BusinessAccountData["Cash
Flow"].rank(pct=True)
GroupedSumOfTransactionsByAccount["Transaction Amount Percentile"] =
GroupedSumOfTransactionsByAccount["sum"].rank(pct=True)

#Checking for no null percentile values
print(BusinessAccountData['Cash Flow Percentile'].isnull().sum())
print(GroupedSumOfTransactionsByAccount['Transaction Amount
Percentile'].isnull().sum())

BusinessAccountData['Cash Flow'] = BusinessAccountData['Cash Flow'].fillna(0)
BusinessAccountData["Cash Flow Percentile"] = BusinessAccountData["Cash
Flow"].rank(pct=True)

#Matching business accounts to transaction accounts by percentile-based pairing
Tree = KDTree(GroupedSumOfTransactionsByAccount[["Transaction Amount
Percentile"]].values)

Distances, Indices = Tree.query(BusinessAccountData[["Cash Flow
Percentile"]].values, k=1)

used_indices = set()
final_matches = []
```

```
for i, idx in enumerate(Indices.flatten()):
    if idx not in used_indices:
        final_matches.append((i, idx))
        used_indices.add(idx)

print(final_matches)

#Putting the matches into a DataFrame
MatchedDataFrame = pd.DataFrame({
    'Company Registration Number': BusinessAccountData.iloc[[i for i, _ in
final_matches]]['company_reg_number'].values,
    'Current Account Number': BusinessAccountData.iloc[[i for i, _ in
final_matches]]['current_account_number'].values,
    'Cash Flow': BusinessAccountData.iloc[[i for i, _ in final_matches]]['Cash
Flow'].values,
    'Primary Sector': BusinessAccountData.iloc[[i for i, _ in
final_matches]]["primary_sector"].values,
    'Matched AccountID': GroupedSumOfTransactionsByAccount.iloc[[j for _, j in
final_matches]]['AccountId'].values,
    'Transaction Sum': GroupedSumOfTransactionsByAccount.iloc[[j for _, j in
final_matches]]['sum'].values,
})

print(MatchedDataFrame)

#Mapping business accounts to transaction accounts
NumberToIDMappings = {}
for i in range(0, len(MatchedDataFrame)):
    AccountNumber = MatchedDataFrame.iloc[i]["Current Account Number"]
    AccountID = MatchedDataFrame.iloc[i]["Matched AccountID"]
    NumberToIDMappings[AccountNumber] = AccountID

BusinessAccountData["AccountID"] =
BusinessAccountData["current_account_number"].map(NumberToIDMappings)
BusinessAccountData = BusinessAccountData.dropna(subset=["AccountID"])

#Reading in the cumulative and month-by-month forecast variables
CumulativeForecastData = pd.read_csv("Naive Forecasts (Cumulative).csv")
MonthByMonthForecastData = pd.read_csv("SSA Forecasts (Month-by-
Month).csv")

#Ensuring that account IDs can be interpreted
BusinessAccountData["AccountID"] =
BusinessAccountData["AccountID"].astype("str")
BusinessAccountData["AccountID"] =
BusinessAccountData["AccountID"].astype("object")
BusinessAccountData["AccountID"] =
BusinessAccountData["AccountID"].astype(str).str.strip()
CumulativeForecastData["AccountID"] =
CumulativeForecastData["AccountID"].astype(str).str.strip()
```

```
MonthByMonthForecastData["AccountID"] =
MonthByMonthForecastData["AccountID"].astype(str).str.strip()

#Joining businesses with their respective transaction forecasts and creating a merged
DataFrame
MergedBusinessesAndForecasts = pd.merge(BusinessAccountData,
CumulativeForecastData, on="AccountID", how="inner")
MergedBusinessesAndForecasts = pd.merge(MergedBusinessesAndForecasts,
MonthByMonthForecastData, on="AccountID", how="inner")
print(MergedBusinessesAndForecasts)

print(LoanData)

#Setting a random seed for reproducibility
np.random.seed(43)
Today = datetime.today()

#Calculating cash flow percentiles in the merged dataset
MergedBusinessesAndForecasts["Cash Flow Percentile Rank"] =
MergedBusinessesAndForecasts["Cash Flow"].rank(pct=True)

#Calculating loan amount percentiles in the loan dataset
LoanData["Loan Amount Percentile Rank"] =
LoanData["loanAmount"].rank(pct=True)

MergedBusinessesAndForecasts =
MergedBusinessesAndForecasts.sort_values("Cash Flow Percentile
Rank").reset_index(drop=True)
LoanData = LoanData.sort_values("Loan Amount Percentile
Rank").reset_index(drop=True)

#Pairing business accounts to loan accounts on the assumption that businesses with
weaker cash flow request larger loans
UsedBusinesses = set()
BusinessToLoanMatches = []

for IndexLoan, RowLoan in LoanData.iterrows():
    UnusedBusinesses =
MergedBusinessesAndForecasts[~MergedBusinessesAndForecasts["AccountID"].isi
n(UsedBusinesses)]

    Differences = (UnusedBusinesses["Cash Flow Percentile Rank"] - (1 -
RowLoan["Loan Amount Percentile Rank"])).abs()
    MinimumDifference = Differences.min()

    CandidateIndices = Differences[Differences ==
MinimumDifference].index.tolist()
    ChosenIndex = np.random.choice(CandidateIndices)

    BusinessToLoanMatches.append({
```

```
        "Business Account ID" : MergedBusinessesAndForecasts.loc[ChosenIndex,
"AccountID"],
        "Cash Flow" : MergedBusinessesAndForecasts.loc[ChosenIndex, "Cash
Flow"],
        "Cash Flow Percentile Rank" :
MergedBusinessesAndForecasts.loc[ChosenIndex, "Cash Flow Percentile Rank"],
        "Loan Account ID" : RowLoan["accountId"],
        "Loan Amount" : RowLoan["loanAmount"],
        "Loan Amount Percentile Rank" : RowLoan["Loan Amount Percentile Rank"]
    })

    UsedBusinesses.add(MergedBusinessesAndForecasts.loc[ChosenIndex,
"AccountID"])

#Generating pairings of business accounts to loan accounts according to the
percentile-based pairing just done
BusinessAndLoanPairings = pd.DataFrame(BusinessToLoanMatches)

LoanToBusinessMappings = {}
for i in range(0, len(BusinessAndLoanPairings)):
    BusinessAccountID = BusinessAndLoanPairings.iloc[i]["Business Account ID"]
    LoanAccountID = BusinessAndLoanPairings.iloc[i]["Loan Account ID"]
    LoanToBusinessMappings[LoanAccountID] = BusinessAccountID

LoanData["AccountID"] = LoanData["accountId"].map(LoanToBusinessMappings)
LoanData = LoanData.dropna(subset=["AccountID"])

#Merging the business account data with the loan data according to the mappings
MergedDataset = pd.merge(MergedBusinessesAndForecasts, LoanData,
on="AccountID", how="inner")

#Selecting the required columns from the merged dataset and writing it into a CSV
MergedDataset = MergedDataset[["AccountID", "Starting Amount", "Ending
Amount", "Net Change over Forecast Period", "Highest Amount", "Lowest
Amount",
                               "Difference", "primary_sector", "company_type",
"2019_revenue", "costs", "accounts_receivable",
                               "capital_and_reserves", "current_assets", "current_liabilities",
"fixed_assets", "long_term_liabilities",
                               "provisions_for_liabilities", "loanAmount",
"yearsOfCreditHistory", "totalCreditLines", "openCreditLines", "bankruptcies",
                               "delinquencies", "netValue", "status"]]
MergedDataset.to_csv("Merged Dataset (Whole Dataset, Assumption Set 3).csv")
```

6.2.1.14 Appendix 2.1.14: Overall Data Pairing (Whole Dataset, Assumption Set 2).py

```
#Importing libraries
import pandas as pd
import random
from datetime import datetime
import datetime as dt
from dateutil import parser
from scipy.spatial import KDTree
import numpy as np

#Reading in the data and filtering the loan data to only include repayers and
defaulters
BusinessAccountData = pd.read_csv("../Menna Dissertation Dataset/MSc
project/open_src_data/synthetic_uk_business_accounts/synthetic_uk_business_curre
nt_accounts.csv")
print(BusinessAccountData)
TransactionData = pd.read_csv("Boosted Transaction Dataset.csv")
LoanData = pd.read_csv("../Menna Dissertation Dataset/MSc
project/open_src_data/sme_loans/sme_loans_data.csv")
print(LoanData)
RepaidLoanData = LoanData[LoanData["status"] == "Paid"]
DefaultedLoanData = LoanData[LoanData["status"] == "Defaulted"]
LoanData = pd.concat([RepaidLoanData, DefaultedLoanData], axis=0,
ignore_index=True)
print(LoanData)
print(DefaultedLoanData["status"])

#Calculating the number of unique business accounts and unique transaction
accounts and resultantly determining the maximum number of feasible pairs
UniqueBusinesses =
BusinessAccountData['current_account_number'].drop_duplicates().reset_index(dro
p=True)
UniqueTransactions =
TransactionData['AccountId'].drop_duplicates().reset_index(drop=True)

MaximumPairs = min(len(UniqueBusinesses), len(UniqueTransactions))

#Converting Debit transaction amounts into negative values
for i in range(0, len(TransactionData)):
    if TransactionData.iloc[i]["TransactionType"] == "Debit":
        TransactionData.at[i, "Amount"] = -(TransactionData.at[i, "Amount"])

#Converting transaction dates into months
TransactionDates = TransactionData["TransactionDate"]

ParsedTransactionDates = [parser.parse(date) for date in TransactionDates]

FormattedTransactionMonths = [date.strftime("%m/%Y") for date in
ParsedTransactionDates]
```

```
TransactionData["TransactionDate"] = FormattedTransactionMonths

#Grouping transactions by account/month and account and calculating the net
transaction amount per grouping
GroupedSumOfTransactionsByAccountAndMonth =
TransactionData.groupby(by=["AccountId",
"TransactionDate"]).agg(["sum"])["Amount"]
print(GroupedSumOfTransactionsByAccountAndMonth)

GroupedSumOfTransactionsByAccount =
TransactionData.groupby(by="AccountId").agg(["sum"])["Amount"]
print(GroupedSumOfTransactionsByAccount)

GroupedSumOfTransactionsByAccount =
GroupedSumOfTransactionsByAccount.reset_index(drop=False)
print(GroupedSumOfTransactionsByAccount)

#Calculating each business account's cash flow (revenue-costs)
BusinessAccountData["Cash Flow"] = BusinessAccountData["2019_revenue"] -
BusinessAccountData["costs"]

#Working out the cash flow and transaction amount percentiles for the business
accounts and transaction accounts
BusinessAccountData["Cash Flow Percentile"] = BusinessAccountData["Cash
Flow"].rank(pct=True)
GroupedSumOfTransactionsByAccount["Transaction Amount Percentile"] =
GroupedSumOfTransactionsByAccount["sum"].rank(pct=True)

print(BusinessAccountData['Cash Flow Percentile'].isnull().sum())
print(GroupedSumOfTransactionsByAccount['Transaction Amount
Percentile'].isnull().sum())

BusinessAccountData['Cash Flow'] = BusinessAccountData['Cash Flow'].fillna(0)
BusinessAccountData["Cash Flow Percentile"] = BusinessAccountData["Cash
Flow"].rank(pct=True)

#Performing percentile-based mapping of business accounts and transaction accounts
based on transaction amount and cash flow percentiles
Tree = KDTree(GroupedSumOfTransactionsByAccount[["Transaction Amount
Percentile"]].values)

Distances, Indices = Tree.query(BusinessAccountData[["Cash Flow
Percentile"]].values, k=1)

used_indices = set()
final_matches = []

for i, idx in enumerate(Indices.flatten()):
    if idx not in used_indices:
```

```
        final_matches.append((i, idx))
        used_indices.add(idx)

print(final_matches)

#Putting the matches into a DataFrame
MatchedDataFrame = pd.DataFrame({
    'Company Registration Number': BusinessAccountData.iloc[[i for i, _ in
final_matches]]['company_reg_number'].values,
    'Current Account Number': BusinessAccountData.iloc[[i for i, _ in
final_matches]]['current_account_number'].values,
    'Cash Flow': BusinessAccountData.iloc[[i for i, _ in final_matches]]['Cash
Flow'].values,
    'Primary Sector': BusinessAccountData.iloc[[i for i, _ in
final_matches]]["primary_sector"].values,
    'Matched AccountID': GroupedSumOfTransactionsByAccount.iloc[[j for _, j in
final_matches]]['AccountId'].values,
    'Transaction Sum': GroupedSumOfTransactionsByAccount.iloc[[j for _, j in
final_matches]]['sum'].values,
})

print(MatchedDataFrame)

#Pairing the business account and transaction datasets based on the mappings
NumberToIDMappings = {}
for i in range(0, len(MatchedDataFrame)):
    AccountNumber = MatchedDataFrame.iloc[i]["Current Account Number"]
    AccountID = MatchedDataFrame.iloc[i]["Matched AccountID"]
    NumberToIDMappings[AccountNumber] = AccountID

BusinessAccountData["AccountID"] =
BusinessAccountData["current_account_number"].map(NumberToIDMappings)
BusinessAccountData = BusinessAccountData.dropna(subset=["AccountID"])

#Reading in the cumulative and month-by-month forecast datasets
CumulativeForecastData = pd.read_csv("Naive Forecasts (Cumulative).csv")
MonthByMonthForecastData = pd.read_csv("SSA Forecasts (Month-by-
Month).csv")

#Making sure the account IDs are interpretable
BusinessAccountData["AccountID"] =
BusinessAccountData["AccountID"].astype("str")
BusinessAccountData["AccountID"] =
BusinessAccountData["AccountID"].astype("object")
BusinessAccountData["AccountID"] =
BusinessAccountData["AccountID"].astype(str).str.strip()
CumulativeForecastData["AccountID"] =
CumulativeForecastData["AccountID"].astype(str).str.strip()
MonthByMonthForecastData["AccountID"] =
MonthByMonthForecastData["AccountID"].astype(str).str.strip()
```

```
#Merging the forecasts with the business accounts
MergedBusinessesAndForecasts = pd.merge(BusinessAccountData,
CumulativeForecastData, on="AccountID", how="inner")
MergedBusinessesAndForecasts = pd.merge(MergedBusinessesAndForecasts,
MonthByMonthForecastData, on="AccountID", how="inner")
print(MergedBusinessesAndForecasts)

print(LoanData)

#Declaring a random seed for reproducibility and declaring today's date
np.random.seed(43)
Today = datetime.today()

#Working out how many years each business has been in business for
IncorporationDates = MergedBusinessesAndForecasts["incorporation_date"]

ParsedIncorporationDates = [parser.parse(date) for date in IncorporationDates]

MergedBusinessesAndForecasts["incorporation_date"] = ParsedIncorporationDates

MergedBusinessesAndForecasts["Years in Business"] = (Today -
MergedBusinessesAndForecasts["incorporation_date"]).dt.days / 365.25

#Determining time percentile ranks for the business account and loan datasets based
on years in business and years of credit history respectively
MergedBusinessesAndForecasts["Time Percentile Rank"] =
MergedBusinessesAndForecasts["Years in Business"].rank(pct=True)
LoanData["Time Percentile Rank"] =
LoanData["yearsOfCreditHistory"].rank(pct=True)

MergedBusinessesAndForecasts =
MergedBusinessesAndForecasts.sort_values("Time Percentile
Rank").reset_index(drop=True)
LoanData = LoanData.sort_values("Time Percentile Rank").reset_index(drop=True)

#Pairing business accounts with loan accounts under the assumption that older
businesses will have a longer credit history
UsedBusinesses = set()
BusinessToLoanMatches = []

for IndexLoan, RowLoan in LoanData.iterrows():
    UnusedBusinesses =
MergedBusinessesAndForecasts[~MergedBusinessesAndForecasts["AccountID"].isi
n(UsedBusinesses)]

    Differences = (UnusedBusinesses["Time Percentile Rank"] - RowLoan["Time
Percentile Rank"]).abs()
    MinimumDifference = Differences.min()
```

```
    CandidateIndices = Differences[Differences ==
MinimumDifference].index.tolist()
    ChosenIndex = np.random.choice(CandidateIndices)

    BusinessToLoanMatches.append({
        "Business Account ID" : MergedBusinessesAndForecasts.loc[ChosenIndex,
"AccountID"],
        "Incorporation Date" : MergedBusinessesAndForecasts.loc[ChosenIndex,
"incorporation_date"],
        "Years in Business" : MergedBusinessesAndForecasts.loc[ChosenIndex, "Years
in Business"],
        "Business Time Percentile Rank" :
MergedBusinessesAndForecasts.loc[ChosenIndex, "Time Percentile Rank"],
        "Loan Account ID" : RowLoan["accountId"],
        "Years of Credit History" : RowLoan["yearsOfCreditHistory"],
        "Loan Time Percentile Rank" : RowLoan["Time Percentile Rank"]
    })

    UsedBusinesses.add(MergedBusinessesAndForecasts.loc[ChosenIndex,
"AccountID"])

#Pairing business accounts to loan accounts according to the pairings generated
above and merging the datasets
BusinessAndLoanPairings = pd.DataFrame(BusinessToLoanMatches)


LoanToBusinessMappings = {}
for i in range(0, len(BusinessAndLoanPairings)):
    BusinessAccountID = BusinessAndLoanPairings.iloc[i]["Business Account ID"]
    LoanAccountID = BusinessAndLoanPairings.iloc[i]["Loan Account ID"]
    LoanToBusinessMappings[LoanAccountID] = BusinessAccountID

LoanData["AccountID"] = LoanData["accountId"].map(LoanToBusinessMappings)
LoanData = LoanData.dropna(subset=["AccountID"])
MergedDataset = pd.merge(MergedBusinessesAndForecasts, LoanData,
on="AccountID", how="inner")

#Selecting the required columns from the merged dataset and writing it to a CSV
MergedDataset = MergedDataset[["AccountID", "Starting Amount", "Ending
Amount", "Net Change over Forecast Period", "Highest Amount", "Lowest
Amount",
                               "Difference", "primary_sector", "company_type",
"2019_revenue", "costs", "accounts_receivable",
                               "capital_and_reserves", "current_assets", "current_liabilities",
"fixed_assets", "long_term_liabilities",
                               "provisions_for_liabilities", "loanAmount",
"yearsOfCreditHistory", "totalCreditLines", "openCreditLines", "bankruptcies",
                               "delinquencies", "netValue", "status"]]
MergedDataset.to_csv("Merged Dataset (Whole Dataset, Assumption Set 2).csv")
```

6.2.1.15 Appendix 2.1.15: Overall Data Pairing (Whole Dataset, Assumption Set 1).py

```
#Importing libraries
import pandas as pd
import random
from datetime import datetime
import datetime as dt
from dateutil import parser
from scipy.spatial import KDTree
import numpy as np

#Reading in the datasets and filtering the loan data to only include repayers and
defaulters
BusinessAccountData = pd.read_csv("../Menna Dissertation Dataset/MSc
project/open_src_data/synthetic_uk_business_accounts/synthetic_uk_business_curre
nt_accounts.csv")
print(BusinessAccountData)
TransactionData = pd.read_csv("Boosted Transaction Dataset.csv")
LoanData = pd.read_csv("../Menna Dissertation Dataset/MSc
project/open_src_data/sme_loans/sme_loans_data.csv")
print(LoanData)
RepaidLoanData = LoanData[LoanData["status"] == "Paid"]
DefaultedLoanData = LoanData[LoanData["status"] == "Defaulted"]
LoanData = pd.concat([RepaidLoanData, DefaultedLoanData], axis=0,
ignore_index=True)
print(LoanData)
print(DefaultedLoanData["status"])

#Calculating the number of unique business accounts and transaction accounts and
resultantly calculating the maximum number of pairs
UniqueBusinesses =
BusinessAccountData['current_account_number'].drop_duplicates().reset_index(dro
p=True)
UniqueTransactions =
TransactionData['AccountId'].drop_duplicates().reset_index(drop=True)

MaximumPairs = min(len(UniqueBusinesses), len(UniqueTransactions))

#Converting Debit transaction amounts into negative values
for i in range(0, len(TransactionData)):
    if TransactionData.iloc[i]["TransactionType"] == "Debit":
        TransactionData.at[i, "Amount"] = -(TransactionData.at[i, "Amount"])

#Converting transaction dates into months
TransactionDates = TransactionData["TransactionDate"]

ParsedTransactionDates = [parser.parse(date) for date in TransactionDates]

FormattedTransactionMonths = [date.strftime("%m/%Y") for date in
ParsedTransactionDates]
```

```
TransactionData["TransactionDate"] = FormattedTransactionMonths

#Grouping transactions by account/month and account and calculating the net
transaction amounts for each grouping
GroupedSumOfTransactionsByAccountAndMonth =
TransactionData.groupby(by=["AccountId",
"TransactionDate"]).agg(["sum"])["Amount"]
print(GroupedSumOfTransactionsByAccountAndMonth)

GroupedSumOfTransactionsByAccount =
TransactionData.groupby(by="AccountId").agg(["sum"])["Amount"]
print(GroupedSumOfTransactionsByAccount)

GroupedSumOfTransactionsByAccount =
GroupedSumOfTransactionsByAccount.reset_index(drop=False)
print(GroupedSumOfTransactionsByAccount)

#Calculating the cash flow (revenue-costs) for each business account
BusinessAccountData["Cash Flow"] = BusinessAccountData["2019_revenue"] -
BusinessAccountData["costs"]

#Calculating cash flow and transaction amount percentiles for each business account
and transaction account respectively
BusinessAccountData["Cash Flow Percentile"] = BusinessAccountData["Cash
Flow"].rank(pct=True)
GroupedSumOfTransactionsByAccount["Transaction Amount Percentile"] =
GroupedSumOfTransactionsByAccount["sum"].rank(pct=True)

print(BusinessAccountData['Cash Flow Percentile'].isnull().sum())
print(GroupedSumOfTransactionsByAccount['Transaction Amount
Percentile'].isnull().sum())

BusinessAccountData['Cash Flow'] = BusinessAccountData['Cash Flow'].fillna(0)
BusinessAccountData["Cash Flow Percentile"] = BusinessAccountData["Cash
Flow"].rank(pct=True)

#Pairing business accounts with transaction accounts using percentile-based pairing
Tree = KDTree(GroupedSumOfTransactionsByAccount[["Transaction Amount
Percentile"]].values)

Distances, Indices = Tree.query(BusinessAccountData[["Cash Flow
Percentile"]].values, k=1)

used_indices = set()
final_matches = []

for i, idx in enumerate(Indices.flatten()):
    if idx not in used_indices:
        final_matches.append((i, idx))
```

```
        used_indices.add(idx)

print(final_matches)

#Generating mappings of business accounts to transaction accounts based on this
pairing
MatchedDataFrame = pd.DataFrame({
    'Company Registration Number': BusinessAccountData.iloc[[i for i, _ in
final_matches]]['company_reg_number'].values,
    'Current Account Number': BusinessAccountData.iloc[[i for i, _ in
final_matches]]['current_account_number'].values,
    'Cash Flow': BusinessAccountData.iloc[[i for i, _ in final_matches]]['Cash
Flow'].values,
    'Primary Sector': BusinessAccountData.iloc[[i for i, _ in
final_matches]]["primary_sector"].values,
    'Matched AccountID': GroupedSumOfTransactionsByAccount.iloc[[j for _, j in
final_matches]]['AccountId'].values,
    'Transaction Sum': GroupedSumOfTransactionsByAccount.iloc[[j for _, j in
final_matches]]['sum'].values,
})

print(MatchedDataFrame)

NumberToIDMappings = {}
for i in range(0, len(MatchedDataFrame)):
    AccountNumber = MatchedDataFrame.iloc[i]["Current Account Number"]
    AccountID = MatchedDataFrame.iloc[i]["Matched AccountID"]
    NumberToIDMappings[AccountNumber] = AccountID

BusinessAccountData["AccountID"] =
BusinessAccountData["current_account_number"].map(NumberToIDMappings)
BusinessAccountData = BusinessAccountData.dropna(subset=["AccountID"])

#Reading in cumulative and month-by-month forecast data
CumulativeForecastData = pd.read_csv("Naive Forecasts (Cumulative).csv")
MonthByMonthForecastData = pd.read_csv("SSA Forecasts (Month-by-
Month).csv")

#Making account IDs interpretable and merging forecasts with businesses
BusinessAccountData["AccountID"] =
BusinessAccountData["AccountID"].astype("str")
BusinessAccountData["AccountID"] =
BusinessAccountData["AccountID"].astype("object")
BusinessAccountData["AccountID"] =
BusinessAccountData["AccountID"].astype(str).str.strip()
CumulativeForecastData["AccountID"] =
CumulativeForecastData["AccountID"].astype(str).str.strip()
MonthByMonthForecastData["AccountID"] =
MonthByMonthForecastData["AccountID"].astype(str).str.strip()
```

```
MergedBusinessesAndForecasts = pd.merge(BusinessAccountData,
CumulativeForecastData, on="AccountID", how="inner")
MergedBusinessesAndForecasts = pd.merge(MergedBusinessesAndForecasts,
MonthByMonthForecastData, on="AccountID", how="inner")
print(MergedBusinessesAndForecasts)

print(LoanData)

#Setting a random seed for reproducibility
np.random.seed(43)
Today = datetime.today()

#Randomly pairing business accounts to loan accounts
BusinessToLoanMatches = []

MaximumPairs = min(len(MergedBusinessesAndForecasts), len(LoanData))

SampledBusinesses =
MergedBusinessesAndForecasts["AccountID"].sample(n=MaximumPairs,
replace=False).reset_index(drop=True)
SampledLoans = LoanData["accountId"].sample(n=MaximumPairs,
replace=False).reset_index(drop=True)

Matches = list(zip(SampledBusinesses, SampledLoans))
for i in Matches:
    BusinessToLoanMatches.append({
        "Business Account ID" : i[0],
        "Loan Account ID" : i[1]
    })

BusinessAndLoanPairings = pd.DataFrame(BusinessToLoanMatches)

#Merging the business account and loan account datasets based on the generated
random pairings
LoanToBusinessMappings = {}
for i in range(0, len(BusinessAndLoanPairings)):
    BusinessAccountID = BusinessAndLoanPairings.iloc[i]["Business Account ID"]
    LoanAccountID = BusinessAndLoanPairings.iloc[i]["Loan Account ID"]
    LoanToBusinessMappings[LoanAccountID] = BusinessAccountID

LoanData["AccountID"] = LoanData["accountId"].map(LoanToBusinessMappings)
LoanData = LoanData.dropna(subset=["AccountID"])
MergedDataset = pd.merge(MergedBusinessesAndForecasts, LoanData,
on="AccountID", how="inner")

#Selecting only required columns from the merged dataset and writing it into a CSV
MergedDataset = MergedDataset[["AccountID", "Starting Amount", "Ending
Amount", "Net Change over Forecast Period", "Highest Amount", "Lowest
Amount",
```

```
                    "Difference", "primary_sector", "company_type",
"2019_revenue", "costs", "accounts_receivable",
                    "capital_and_reserves", "current_assets", "current_liabilities",
"fixed_assets", "long_term_liabilities",
                    "provisions_for_liabilities", "loanAmount",
"yearsOfCreditHistory", "totalCreditLines", "openCreditLines", "bankruptcies",
                    "delinquencies", "netValue", "status"]]
MergedDataset = MergedDataset.dropna()
MergedDataset.to_csv("Merged Dataset (Whole Dataset, Assumption Set 1).csv")
```

6.2.1.16 Appendix 2.1.16: Month-by-Month Forecast Classifier (Assumption Set 1, Whole Dataset).py

```
#Importing libraries
import xgboost as xgb
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, f1_score
from sklearn.metrics import roc_auc_score
from sklearn.metrics import average_precision_score
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.utils import resample
from imblearn.over_sampling import SMOTE
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import StratifiedKFold

#Setting a random seed for reproducibility
Seed = 43

#Reading in the data
Data = pd.read_csv("Merged Dataset (Whole Dataset, Assumption Set 1).csv")

#Reading in X and Y variables
X = Data[["Highest Amount", "Lowest Amount", "Difference"]]
Y = Data["status"]

#Label encoding Y
Encoder = LabelEncoder()
YEncoded = Encoder.fit_transform(Y)
print(Encoder.classes_)

#Splitting the data into training and testing sets
XTrain, XTest, YTrain, YTest = train_test_split(X, YEncoded, test_size=0.3,
stratify=YEncoded, random_state=Seed)

#Determining the positive class weighting based on the number of negatives over the
number of positives
NumNegative = sum(YTrain == 0)
NumPositive = sum(YTrain == 1)
PositiveWeighting = NumNegative/NumPositive

#Determining the parameter grid for the grid search
Parameters = {"n_estimators" : [25, 50, 100, 200],
              "eta" : [0.01, 0.05, 0.1, 0.2],
              "max_depth" : [1, 2, 4, 8]}

#Performing a grid search to find the optimal XGBoost parameters
print("Train Length After Resampling:",len(YTrain))
print("Test Length:",len(YTest))
```

```
Model = xgb.XGBClassifier(n_estimators=100, objective='binary:logistic',
tree_method='hist', scale_pos_weight=PositiveWeighting, random_state=Seed)
Validator = StratifiedKFold(n_splits=10, shuffle=True, random_state=Seed)
Search = GridSearchCV(Model, Parameters, scoring="roc_auc", n_jobs=1,
cv=Validator)
Results = Search.fit(XTrain, YTrain)

#Taking the best model, making predictions with it and outputting the results
BestModel = Results.best_estimator_

RegularPredictions = BestModel.predict(XTest)
Accuracy = accuracy_score(RegularPredictions, YTest)
Probabilities = BestModel.predict_proba(XTest)[:,1]
AUC = roc_auc_score(YTest, Probabilities)
PR_AUC = average_precision_score(YTest, Probabilities, pos_label=0)
Metrics = classification_report(YTest, RegularPredictions,
target_names=["Defaulted", "Repaid"])
print(Metrics)
ConfusionMatrix = confusion_matrix(YTest, RegularPredictions)
print(ConfusionMatrix)
```

6.2.1.17 Appendix 2.1.17: Cumulative and Month-by-Month Forecast Classifier (Assumption Set 1, Whole Dataset).py

```
#Importing libraries
import xgboost as xgb
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, f1_score
from sklearn.metrics import roc_auc_score
from sklearn.metrics import average_precision_score
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.utils import resample
from imblearn.over_sampling import SMOTE
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import StratifiedKFold

#Setting a random seed for reproducibility
Seed = 43

#Reading in the data
Data = pd.read_csv("Merged Dataset (Whole Dataset, Assumption Set 1).csv")

#Determining X and Y
X = Data[["Starting Amount", "Ending Amount", "Net Change over Forecast
Period", "Highest Amount", "Lowest Amount", "Difference"]]
Y = Data["status"]

#Label encoding Y
Encoder = LabelEncoder()
YEncoded = Encoder.fit_transform(Y)
print(Encoder.classes_)

#Splitting the data into training and testing sets
XTrain, XTest, YTrain, YTest = train_test_split(X, YEncoded, test_size=0.3,
stratify=YEncoded, random_state=Seed)

#Determining positive class weighting
NumNegative = sum(YTrain == 0)
NumPositive = sum(YTrain == 1)
PositiveWeighting = NumNegative/NumPositive

#Declaring parameter grid for the grid search
Parameters = {"n_estimators" : [25, 50, 100, 200],
              "eta" : [0.01, 0.05, 0.1, 0.2],
              "max_depth" : [1, 2, 4, 8]}

#Performing a grid search to find the best XGBoost parameters
print("Train Length After Resampling:",len(YTrain))
print("Test Length:",len(YTest))
```

```
Model = xgb.XGBClassifier(n_estimators=100, objective='binary:logistic',
tree_method='hist', scale_pos_weight=PositiveWeighting, random_state=Seed)
Validator = StratifiedKFold(n_splits=10, shuffle=True, random_state=Seed)
Search = GridSearchCV(Model, Parameters, scoring="roc_auc", n_jobs=1,
cv=Validator)
Results = Search.fit(XTrain, YTrain)

#Selecting the best model, making predictions with it and outputting the results
BestModel = Results.best_estimator_

RegularPredictions = BestModel.predict(XTest)
Accuracy = accuracy_score(RegularPredictions, YTest)
Probabilities = BestModel.predict_proba(XTest)[:,1]
AUC = roc_auc_score(YTest, Probabilities)
PR_AUC = average_precision_score(YTest, Probabilities, pos_label=0)
Metrics = classification_report(YTest, RegularPredictions,
target_names=["Defaulted", "Repaid"])
print(Metrics)
ConfusionMatrix = confusion_matrix(YTest, RegularPredictions)
print(ConfusionMatrix)
```

6.2.1.18 Appendix 2.1.18: Traditional Variables Classifier (Assumption Set 1, Whole Dataset).py

```
#Importing libraries
import xgboost as xgb
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, f1_score
from sklearn.metrics import roc_auc_score
from sklearn.metrics import average_precision_score
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.utils import resample
from imblearn.over_sampling import SMOTE
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import StratifiedKFold

#Setting a random seed for reproducibility
Seed = 43

#Reading in the data
Data = pd.read_csv("Merged Dataset (Whole Dataset, Assumption Set 1).csv")

#Reading in X and Y
X = Data[["primary_sector", "company_type", "2019_revenue", "costs",
"accounts_receivable", "capital_and_reserves", "current_assets",
"current_liabilities",
        "fixed_assets", "long_term_liabilities", "provisions_for_liabilities",
"loanAmount", "yearsOfCreditHistory", "totalCreditLines", "openCreditLines",
        "bankruptcies", "delinquencies", "netValue"]]
Y = Data["status"]

#One-hot encoding categorical X variables
XNumerical = X.drop(columns=["primary_sector", "company_type"])
XCategorical = X[["primary_sector", "company_type"]]

OneHotEncoder = OneHotEncoder(sparse_output=False)
XCategoricalEncoded =
pd.DataFrame(OneHotEncoder.fit_transform(XCategorical),
columns=OneHotEncoder.get_feature_names_out(XCategorical.columns))

XEncoded = pd.concat([XNumerical.reset_index(drop=True),
XCategoricalEncoded.reset_index(drop=True)], axis=1)

#Label encoding Y
LblEncoder = LabelEncoder()
YEncoded = LblEncoder.fit_transform(Y)
print(LblEncoder.classes_)
```

```
#Splitting the data into training and testing sets
XTrain, XTest, YTrain, YTest = train_test_split(XEncoded, YEncoded,
test_size=0.3, stratify=YEncoded, random_state=Seed)

#Determining the positive class weighting
NumNegative = sum(YTrain == 0)
NumPositive = sum(YTrain == 1)
PositiveWeighting = NumNegative/NumPositive

#Declaring the parameter grid for the grid search
Parameters = {"n_estimators" : [25, 50, 100, 200],
              "eta" : [0.01, 0.05, 0.1, 0.2],
              "max_depth" : [1, 2, 4, 8]}

#Performing a grid search to find the ideal XGBoost parameters
print("Train Length After Resampling:",len(YTrain))
print("Test Length:",len(YTest))
Model = xgb.XGBClassifier(n_estimators=100, objective='binary:logistic',
tree_method='hist', scale_pos_weight=PositiveWeighting, random_state=Seed)
Validator = StratifiedKFold(n_splits=10, shuffle=True, random_state=Seed)
Search = GridSearchCV(Model, Parameters, scoring="roc_auc", n_jobs=1,
cv=Validator)
Results = Search.fit(XTrain, YTrain)

#Taking the best model, performing predictions with it and outputting the results
BestModel = Results.best_estimator_

RegularPredictions = BestModel.predict(XTest)
Accuracy = accuracy_score(RegularPredictions, YTest)
Probabilities = BestModel.predict_proba(XTest)[:,1]
AUC = roc_auc_score(YTest, Probabilities)
PR_AUC = average_precision_score(YTest, Probabilities, pos_label=0)
Metrics = classification_report(YTest, RegularPredictions,
target_names=["Defaulted", "Repaid"])
print(Metrics)
ConfusionMatrix = confusion_matrix(YTest, RegularPredictions)
print(ConfusionMatrix)
```

6.2.1.19 Appendix 2.1.19: Traditional Variables and Forecast Classifier (Assumption Set 1, Whole Dataset).py

```
#Importing libraries
import xgboost as xgb
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, f1_score
from sklearn.metrics import roc_auc_score
from sklearn.metrics import average_precision_score
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.utils import resample
from imblearn.over_sampling import SMOTE
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import StratifiedKFold

#Setting a random seed for reproducibility
Seed = 43

#Reading in the data
Data = pd.read_csv("Merged Dataset (Whole Dataset, Assumption Set 1).csv")

#Declaring X and Y
X = Data[["primary_sector", "company_type", "2019_revenue", "costs",
"accounts_receivable", "capital_and_reserves", "current_assets",
"current_liabilities",
        "fixed_assets", "long_term_liabilities", "provisions_for_liabilities",
"loanAmount", "yearsOfCreditHistory", "totalCreditLines", "openCreditLines",
        "bankruptcies", "delinquencies", "netValue", "Highest Amount", "Lowest
Amount", "Difference"]]
Y = Data["status"]

#One-hot encoding categorical X variables
XNumerical = X.drop(columns=["primary_sector", "company_type"])
XCategorical = X[["primary_sector", "company_type"]]

OneHotEncoder = OneHotEncoder(sparse_output=False)
XCategoricalEncoded =
pd.DataFrame(OneHotEncoder.fit_transform(XCategorical),
columns=OneHotEncoder.get_feature_names_out(XCategorical.columns))

XEncoded = pd.concat([XNumerical.reset_index(drop=True),
XCategoricalEncoded.reset_index(drop=True)], axis=1)

#Label encoding Y
LblEncoder = LabelEncoder()
YEncoded = LblEncoder.fit_transform(Y)
print(LblEncoder.classes_)
```

```
#Splitting the data into training and testing sets
XTrain, XTest, YTrain, YTest = train_test_split(XEncoded, YEncoded,
test_size=0.3, stratify=YEncoded, random_state=Seed)

#Determining the positive class weighting
NumNegative = sum(YTrain == 0)
NumPositive = sum(YTrain == 1)
PositiveWeighting = NumNegative/NumPositive

#Declaring the parameter grid for the grid search
Parameters = {"n_estimators" : [25, 50, 100, 200],
              "eta" : [0.01, 0.05, 0.1, 0.2],
              "max_depth" : [1, 2, 4, 8]}

#Performing a grid search to determine the ideal XGBoost parameters
print("Train Length After Resampling:",len(YTrain))
print("Test Length:",len(YTest))
Model = xgb.XGBClassifier(n_estimators=100, objective='binary:logistic',
tree_method='hist', scale_pos_weight=PositiveWeighting, random_state=Seed)
Validator = StratifiedKFold(n_splits=10, shuffle=True, random_state=Seed)
Search = GridSearchCV(Model, Parameters, scoring="roc_auc", n_jobs=1,
cv=Validator)
Results = Search.fit(XTrain, YTrain)

#Taking the best model, performing predictions with it and outputting the results
BestModel = Results.best_estimator_

RegularPredictions = BestModel.predict(XTest)
Accuracy = accuracy_score(RegularPredictions, YTest)
Probabilities = BestModel.predict_proba(XTest)[:,1]
AUC = roc_auc_score(YTest, Probabilities)
PR_AUC = average_precision_score(YTest, Probabilities, pos_label=0)
Metrics = classification_report(YTest, RegularPredictions,
target_names=["Defaulted", "Repaid"])
print(Metrics)
ConfusionMatrix = confusion_matrix(YTest, RegularPredictions)
print(ConfusionMatrix)
```

6.2.1.20 Appendix 2.1.20: Cumulative Forecast Classifier (Assumption Set 1, Whole Dataset).py

```
#Importing libraries
import xgboost as xgb
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, f1_score
from sklearn.metrics import roc_auc_score
from sklearn.metrics import average_precision_score
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.utils import resample
from imblearn.over_sampling import SMOTE
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import StratifiedKFold

#Setting random seed for reproducibility
Seed = 43

#Reading in the data
Data = pd.read_csv("Merged Dataset (Whole Dataset, Assumption Set 1).csv")

#Declaring X and Y
X = Data[["Starting Amount", "Ending Amount", "Net Change over Forecast
Period"]]
Y = Data["status"]

#Label encoding Y
Encoder = LabelEncoder()
YEncoded = Encoder.fit_transform(Y)
print(Encoder.classes_)

#Splitting the data into training and testing sets
XTrain, XTest, YTrain, YTest = train_test_split(X, YEncoded, test_size=0.3,
stratify=YEncoded, random_state=Seed)

#Determining the positive class weighting
NumNegative = sum(YTrain == 0)
NumPositive = sum(YTrain == 1)
PositiveWeighting = NumNegative/NumPositive

#Declaring the parameter grid for the grid search
Parameters = {"n_estimators" : [25, 50, 100, 200],
              "eta" : [0.01, 0.05, 0.1, 0.2],
              "max_depth" : [1, 2, 4, 8]}

#Undertaking a grid search to find the best XGBoost parameters
print("Train Length After Resampling:",len(YTrain))
print("Test Length:",len(YTest))
```

```
Model = xgb.XGBClassifier(n_estimators=100, objective='binary:logistic',
tree_method='hist', scale_pos_weight=PositiveWeighting, random_state=Seed)
Validator = StratifiedKFold(n_splits=10, shuffle=True, random_state=Seed)
Search = GridSearchCV(Model, Parameters, scoring="roc_auc", n_jobs=1,
cv=Validator)
Results = Search.fit(XTrain, YTrain)

#Taking the best model, making predictions with it and outputting the results
BestModel = Results.best_estimator_

RegularPredictions = BestModel.predict(XTest)
Accuracy = accuracy_score(RegularPredictions, YTest)
Probabilities = BestModel.predict_proba(XTest)[:,1]
AUC = roc_auc_score(YTest, Probabilities)
PR_AUC = average_precision_score(YTest, Probabilities, pos_label=0)
Metrics = classification_report(YTest, RegularPredictions,
target_names=["Defaulted", "Repaid"])
print(Metrics)
ConfusionMatrix = confusion_matrix(YTest, RegularPredictions)
print(ConfusionMatrix)
```

6.2.1.21 Appendix 2.1.21: Cumulative Forecast Classifier (Assumption Set 2, Whole Dataset).py

```
#Importing libraries
import xgboost as xgb
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, f1_score
from sklearn.metrics import roc_auc_score
from sklearn.metrics import average_precision_score
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.utils import resample
from imblearn.over_sampling import SMOTE
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import StratifiedKFold

#Declaring a random seed for reproducibility
Seed = 43

#Reading in the data
Data = pd.read_csv("Merged Dataset (Whole Dataset, Assumption Set 2).csv")

#Declaring X and Y
X = Data[["Starting Amount", "Ending Amount", "Net Change over Forecast
Period"]]
Y = Data["status"]

#Label encoding Y
Encoder = LabelEncoder()
YEncoded = Encoder.fit_transform(Y)
print(Encoder.classes_)

#Splitting the data into training and testing sets
XTrain, XTest, YTrain, YTest = train_test_split(X, YEncoded, test_size=0.3,
stratify=YEncoded, random_state=Seed)

#Determining the positive class weighting
NumNegative = sum(YTrain == 0)
NumPositive = sum(YTrain == 1)
PositiveWeighting = NumNegative/NumPositive

#Declaring the parameter grid for the grid search
Parameters = {"n_estimators" : [25, 50, 100, 200],
              "eta" : [0.01, 0.05, 0.1, 0.2],
              "max_depth" : [1, 2, 4, 8]}

#Undertaking a grid search to determine the best XGBoost parameters
print("Train Length After Resampling:",len(YTrain))
print("Test Length:",len(YTest))
```

```
Model = xgb.XGBClassifier(n_estimators=100, objective='binary:logistic',
tree_method='hist', scale_pos_weight=PositiveWeighting, random_state=Seed)
Validator = StratifiedKFold(n_splits=10, shuffle=True, random_state=Seed)
Search = GridSearchCV(Model, Parameters, scoring="roc_auc", n_jobs=1,
cv=Validator)
Results = Search.fit(XTrain, YTrain)

#Taking the best model, performing predictions with it and outputting the results
BestModel = Results.best_estimator_

RegularPredictions = BestModel.predict(XTest)
Accuracy = accuracy_score(RegularPredictions, YTest)
Probabilities = BestModel.predict_proba(XTest)[:,1]
AUC = roc_auc_score(YTest, Probabilities)
PR_AUC = average_precision_score(YTest, Probabilities, pos_label=0)
Metrics = classification_report(YTest, RegularPredictions,
target_names=["Defaulted", "Repaid"])
print(Metrics)
ConfusionMatrix = confusion_matrix(YTest, RegularPredictions)
print(ConfusionMatrix)
```

6.2.1.22 Appendix 2.1.22: Month-by-Month Forecast Classifier (Assumption Set 2, Whole Dataset).py

```
#Importing libraries
import xgboost as xgb
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, f1_score
from sklearn.metrics import roc_auc_score
from sklearn.metrics import average_precision_score
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.utils import resample
from imblearn.over_sampling import SMOTE
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import StratifiedKFold

#Setting a random seed for reproducibility
Seed = 43

#Reading in the data
Data = pd.read_csv("Merged Dataset (Whole Dataset, Assumption Set 2).csv")

#Declaring X and Y
X = Data[["Highest Amount", "Lowest Amount", "Difference"]]
Y = Data["status"]

#Label encoding Y
Encoder = LabelEncoder()
YEncoded = Encoder.fit_transform(Y)
print(Encoder.classes_)

#Splitting the data into training and testing sets
XTrain, XTest, YTrain, YTest = train_test_split(X, YEncoded, test_size=0.3,
stratify=YEncoded, random_state=Seed)

#Determining the positive class weighting
NumNegative = sum(YTrain == 0)
NumPositive = sum(YTrain == 1)
PositiveWeighting = NumNegative/NumPositive

#Declaring the parameter grid for the grid search
Parameters = {"n_estimators" : [25, 50, 100, 200],
              "eta" : [0.01, 0.05, 0.1, 0.2],
              "max_depth" : [1, 2, 4, 8]}

#Performing a grid search to determine the best XGBoost parameters
print("Train Length After Resampling:",len(YTrain))
print("Test Length:",len(YTest))
```

```
Model = xgb.XGBClassifier(n_estimators=100, objective='binary:logistic',
tree_method='hist', scale_pos_weight=PositiveWeighting, random_state=Seed)
Validator = StratifiedKFold(n_splits=10, shuffle=True, random_state=Seed)
Search = GridSearchCV(Model, Parameters, scoring="roc_auc", n_jobs=1,
cv=Validator)
Results = Search.fit(XTrain, YTrain)

#Taking the best model, making predictions with it and outputting the results
BestModel = Results.best_estimator_

RegularPredictions = BestModel.predict(XTest)
Accuracy = accuracy_score(RegularPredictions, YTest)
Probabilities = BestModel.predict_proba(XTest)[:,1]
AUC = roc_auc_score(YTest, Probabilities)
PR_AUC = average_precision_score(YTest, Probabilities, pos_label=0)
Metrics = classification_report(YTest, RegularPredictions,
target_names=["Defaulted", "Repaid"])
print(Metrics)
ConfusionMatrix = confusion_matrix(YTest, RegularPredictions)
print(ConfusionMatrix)
```

6.2.1.23 Appendix 2.1.23: Cumulative and Month-by-Month Forecast Classifier (Assumption Set 2, Whole Dataset).py

```
#Importing libraries
import xgboost as xgb
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, f1_score
from sklearn.metrics import roc_auc_score
from sklearn.metrics import average_precision_score
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.utils import resample
from imblearn.over_sampling import SMOTE
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import StratifiedKFold

#Declaring a random seed for reproducibility
Seed = 43

#Reading in the data
Data = pd.read_csv("Merged Dataset (Whole Dataset, Assumption Set 2).csv")

#Declaring X and Y
X = Data[["Starting Amount", "Ending Amount", "Net Change over Forecast
Period", "Highest Amount", "Lowest Amount", "Difference"]]
Y = Data["status"]

#Label encoding Y
Encoder = LabelEncoder()
YEncoded = Encoder.fit_transform(Y)
print(Encoder.classes_)

#Splitting the data into training and testing sets
XTrain, XTest, YTrain, YTest = train_test_split(X, YEncoded, test_size=0.3,
stratify=YEncoded, random_state=Seed)

#Determining the positive class weighting
NumNegative = sum(YTrain == 0)
NumPositive = sum(YTrain == 1)
PositiveWeighting = NumNegative/NumPositive

#Declaring the parameter grid for the grid search
Parameters = {"n_estimators" : [25, 50, 100, 200],
              "eta" : [0.01, 0.05, 0.1, 0.2],
              "max_depth" : [1, 2, 4, 8]}

#Performing a grid search to find the best XGBoost parameters
print("Train Length After Resampling:",len(YTrain))
print("Test Length:",len(YTest))
```

```
Model = xgb.XGBClassifier(n_estimators=100, objective='binary:logistic',
tree_method='hist', scale_pos_weight=PositiveWeighting, random_state=Seed)
Validator = StratifiedKFold(n_splits=10, shuffle=True, random_state=Seed)
Search = GridSearchCV(Model, Parameters, scoring="roc_auc", n_jobs=1,
cv=Validator)
Results = Search.fit(XTrain, YTrain)

#Picking the best model, performing predictions with it and outputting the results
BestModel = Results.best_estimator_

RegularPredictions = BestModel.predict(XTest)
Accuracy = accuracy_score(RegularPredictions, YTest)
Probabilities = BestModel.predict_proba(XTest)[:,1]
AUC = roc_auc_score(YTest, Probabilities)
PR_AUC = average_precision_score(YTest, Probabilities, pos_label=0)
Metrics = classification_report(YTest, RegularPredictions,
target_names=["Defaulted", "Repaid"])
print(Metrics)
ConfusionMatrix = confusion_matrix(YTest, RegularPredictions)
print(ConfusionMatrix)
```

6.2.1.24 Appendix 2.1.24: Traditional Variables Classifier (Assumption Set 2, Whole Dataset).py

```
#Importing libraries
import xgboost as xgb
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, f1_score
from sklearn.metrics import roc_auc_score
from sklearn.metrics import average_precision_score
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.utils import resample
from imblearn.over_sampling import SMOTE
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import StratifiedKFold

#Declaring random seed for reproducibility
Seed = 43

#Reading in the data
Data = pd.read_csv("Merged Dataset (Whole Dataset, Assumption Set 2).csv")

#Declaring X and Y
X = Data[["primary_sector", "company_type", "2019_revenue", "costs",
"accounts_receivable", "capital_and_reserves", "current_assets",
"current_liabilities",
        "fixed_assets", "long_term_liabilities", "provisions_for_liabilities",
"loanAmount", "yearsOfCreditHistory", "totalCreditLines", "openCreditLines",
        "bankruptcies", "delinquencies", "netValue"]]
Y = Data["status"]

#One-hot encoding categorical X variables
XNumerical = X.drop(columns=["primary_sector", "company_type"])
XCategorical = X[["primary_sector", "company_type"]]

OneHotEncoder = OneHotEncoder(sparse_output=False)
XCategoricalEncoded =
pd.DataFrame(OneHotEncoder.fit_transform(XCategorical),
columns=OneHotEncoder.get_feature_names_out(XCategorical.columns))

XEncoded = pd.concat([XNumerical.reset_index(drop=True),
XCategoricalEncoded.reset_index(drop=True)], axis=1)

#Label encoding Y
LblEncoder = LabelEncoder()
YEncoded = LblEncoder.fit_transform(Y)
print(LblEncoder.classes_)
```

```
#Splitting the data into training and testing sets
XTrain, XTest, YTrain, YTest = train_test_split(XEncoded, YEncoded,
test_size=0.3, stratify=YEncoded, random_state=Seed)

#Determining the positive class weighting
NumNegative = sum(YTrain == 0)
NumPositive = sum(YTrain == 1)
PositiveWeighting = NumNegative/NumPositive

#Declaring the parameter grid for the grid search
Parameters = {"n_estimators" : [25, 50, 100, 200],
              "eta" : [0.01, 0.05, 0.1, 0.2],
              "max_depth" : [1, 2, 4, 8]}

#Performing a grid search to determine the best XGBoost parameters
print("Train Length After Resampling:",len(YTrain))
print("Test Length:",len(YTest))
Model = xgb.XGBClassifier(n_estimators=100, objective='binary:logistic',
tree_method='hist', scale_pos_weight=PositiveWeighting, random_state=Seed)
Validator = StratifiedKFold(n_splits=10, shuffle=True, random_state=Seed)
Search = GridSearchCV(Model, Parameters, scoring="roc_auc", n_jobs=1,
cv=Validator)
Results = Search.fit(XTrain, YTrain)

#Taking the best model, making predictions with it and outputting results
BestModel = Results.best_estimator_

RegularPredictions = BestModel.predict(XTest)
Accuracy = accuracy_score(RegularPredictions, YTest)
Probabilities = BestModel.predict_proba(XTest)[:,1]
AUC = roc_auc_score(YTest, Probabilities)
PR_AUC = average_precision_score(YTest, Probabilities, pos_label=0)
Metrics = classification_report(YTest, RegularPredictions,
target_names=["Defaulted", "Repaid"])
print(Metrics)
ConfusionMatrix = confusion_matrix(YTest, RegularPredictions)
print(ConfusionMatrix)
```

6.2.1.25 Appendix 2.1.25: Traditional Variables and Forecast Classifier (Assumption Set 2, Whole Dataset).py

```
#Importing libraries
import xgboost as xgb
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, f1_score
from sklearn.metrics import roc_auc_score
from sklearn.metrics import average_precision_score
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.utils import resample
from imblearn.over_sampling import SMOTE
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import StratifiedKFold

#Setting random seed for reproducibility
Seed = 43

#Reading in the data
Data = pd.read_csv("Merged Dataset (Whole Dataset, Assumption Set 2).csv")

#Declaring X and Y
X = Data[["primary_sector", "company_type", "2019_revenue", "costs",
"accounts_receivable", "capital_and_reserves", "current_assets",
"current_liabilities",
        "fixed_assets", "long_term_liabilities", "provisions_for_liabilities",
"loanAmount", "yearsOfCreditHistory", "totalCreditLines", "openCreditLines",
        "bankruptcies", "delinquencies", "netValue", "Highest Amount", "Lowest
Amount", "Difference"]]
Y = Data["status"]

#One-hot encoding categorical X variables
XNumerical = X.drop(columns=["primary_sector", "company_type"])
XCategorical = X[["primary_sector", "company_type"]]

OneHotEncoder = OneHotEncoder(sparse_output=False)
XCategoricalEncoded =
pd.DataFrame(OneHotEncoder.fit_transform(XCategorical),
columns=OneHotEncoder.get_feature_names_out(XCategorical.columns))

XEncoded = pd.concat([XNumerical.reset_index(drop=True),
XCategoricalEncoded.reset_index(drop=True)], axis=1)

#Label encoding Y
LblEncoder = LabelEncoder()
YEncoded = LblEncoder.fit_transform(Y)
print(LblEncoder.classes_)
```

```
#Splitting the data into training and testing sets
XTrain, XTest, YTrain, YTest = train_test_split(XEncoded, YEncoded,
test_size=0.3, stratify=YEncoded, random_state=Seed)

#Setting the positive class weighting
NumNegative = sum(YTrain == 0)
NumPositive = sum(YTrain == 1)
PositiveWeighting = NumNegative/NumPositive

#Declaring the parameter grid for the grid search
Parameters = {"n_estimators" : [25, 50, 100, 200],
              "eta" : [0.01, 0.05, 0.1, 0.2],
              "max_depth" : [1, 2, 4, 8]}

#Performing a grid search to find the best XGBoost parameters
print("Train Length After Resampling:",len(YTrain))
print("Test Length:",len(YTest))
Model = xgb.XGBClassifier(n_estimators=100, objective='binary:logistic',
tree_method='hist', scale_pos_weight=PositiveWeighting, random_state=Seed)
Validator = StratifiedKFold(n_splits=10, shuffle=True, random_state=Seed)
Search = GridSearchCV(Model, Parameters, scoring="roc_auc", n_jobs=1,
cv=Validator)
Results = Search.fit(XTrain, YTrain)

#Selecting the best model, performing predictions with it and outputting the results
BestModel = Results.best_estimator_

RegularPredictions = BestModel.predict(XTest)
Accuracy = accuracy_score(RegularPredictions, YTest)
Probabilities = BestModel.predict_proba(XTest)[:,1]
AUC = roc_auc_score(YTest, Probabilities)
PR_AUC = average_precision_score(YTest, Probabilities, pos_label=0)
Metrics = classification_report(YTest, RegularPredictions,
target_names=["Defaulted", "Repaid"])
print(Metrics)
ConfusionMatrix = confusion_matrix(YTest, RegularPredictions)
print(ConfusionMatrix)
```

6.2.1.26 Appendix 2.1.26: Cumulative Forecast Classifier (Assumption Set 3, Whole Dataset).py

```
#Importing libraries
import xgboost as xgb
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, f1_score
from sklearn.metrics import roc_auc_score
from sklearn.metrics import average_precision_score
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.utils import resample
from imblearn.over_sampling import SMOTE
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import StratifiedKFold

#Declaring a random seed for reproducibility
Seed = 43

#Reading in the data
Data = pd.read_csv("Merged Dataset (Whole Dataset, Assumption Set 3).csv")

#Declaring X and Y
X = Data[["Starting Amount", "Ending Amount", "Net Change over Forecast
Period"]]
Y = Data["status"]

#Label encoding Y
Encoder = LabelEncoder()
YEncoded = Encoder.fit_transform(Y)
print(Encoder.classes_)

#Splitting the data into training and testing sets
XTrain, XTest, YTrain, YTest = train_test_split(X, YEncoded, test_size=0.3,
stratify=YEncoded, random_state=Seed)

#Determining the positive class weighting
NumNegative = sum(YTrain == 0)
NumPositive = sum(YTrain == 1)
PositiveWeighting = NumNegative/NumPositive

#Declaring the parameter grid for the grid search
Parameters = {"n_estimators" : [25, 50, 100, 200],
              "eta" : [0.01, 0.05, 0.1, 0.2],
              "max_depth" : [1, 2, 4, 8]}

#Performing a grid search to find the best XGBoost parameters
print("Train Length After Resampling:",len(YTrain))
print("Test Length:",len(YTest))
```

```
Model = xgb.XGBClassifier(n_estimators=100, objective='binary:logistic',
tree_method='hist', scale_pos_weight=PositiveWeighting, random_state=Seed)
Validator = StratifiedKFold(n_splits=10, shuffle=True, random_state=Seed)
Search = GridSearchCV(Model, Parameters, scoring="roc_auc", n_jobs=1,
cv=Validator)
Results = Search.fit(XTrain, YTrain)

#Taking the best model, making predictions with it and outputting the results
BestModel = Results.best_estimator_

RegularPredictions = BestModel.predict(XTest)
Accuracy = accuracy_score(RegularPredictions, YTest)
Probabilities = BestModel.predict_proba(XTest)[:,1]
AUC = roc_auc_score(YTest, Probabilities)
PR_AUC = average_precision_score(YTest, Probabilities, pos_label=0)
Metrics = classification_report(YTest, RegularPredictions,
target_names=["Defaulted", "Repaid"])
print(Metrics)
ConfusionMatrix = confusion_matrix(YTest, RegularPredictions)
print(ConfusionMatrix)
```

6.2.1.27 Appendix 2.1.27: Month-by-Month Forecast Classifier (Assumption Set 3, Whole Dataset).py

```
#Importing libraries
import xgboost as xgb
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, f1_score
from sklearn.metrics import roc_auc_score
from sklearn.metrics import average_precision_score
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.utils import resample
from imblearn.over_sampling import SMOTE
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import StratifiedKFold

#Declaring a random seed for reproducibility
Seed = 43

#Reading in the data
Data = pd.read_csv("Merged Dataset (Whole Dataset, Assumption Set 3).csv")

#Declaring X and Y
X = Data[["Highest Amount", "Lowest Amount", "Difference"]]
Y = Data["status"]

#Label encoding Y
Encoder = LabelEncoder()
YEncoded = Encoder.fit_transform(Y)
print(Encoder.classes_)

#Splitting the data into training and testing sets
XTrain, XTest, YTrain, YTest = train_test_split(X, YEncoded, test_size=0.3,
stratify=YEncoded, random_state=Seed)

#Determining the positive class weighting
NumNegative = sum(YTrain == 0)
NumPositive = sum(YTrain == 1)
PositiveWeighting = NumNegative/NumPositive

#Declaring the parameter grid for the grid search
Parameters = {"n_estimators" : [25, 50, 100, 200],
              "eta" : [0.01, 0.05, 0.1, 0.2],
              "max_depth" : [1, 2, 4, 8]}

#Performing a grid search to determine the best XGBoost parameters
print("Train Length After Resampling:",len(YTrain))
print("Test Length:",len(YTest))
```

```
Model = xgb.XGBClassifier(n_estimators=100, objective='binary:logistic',
tree_method='hist', scale_pos_weight=PositiveWeighting, random_state=Seed)
Validator = StratifiedKFold(n_splits=10, shuffle=True, random_state=Seed)
Search = GridSearchCV(Model, Parameters, scoring="roc_auc", n_jobs=1,
cv=Validator)
Results = Search.fit(XTrain, YTrain)

#Picking the best model, performing predictions with it and outputting the results
BestModel = Results.best_estimator_

RegularPredictions = BestModel.predict(XTest)
Accuracy = accuracy_score(RegularPredictions, YTest)
Probabilities = BestModel.predict_proba(XTest)[:,1]
AUC = roc_auc_score(YTest, Probabilities)
PR_AUC = average_precision_score(YTest, Probabilities, pos_label=0)
Metrics = classification_report(YTest, RegularPredictions,
target_names=["Defaulted", "Repaid"])
print(Metrics)
ConfusionMatrix = confusion_matrix(YTest, RegularPredictions)
print(ConfusionMatrix)
```

6.2.1.28 Appendix 2.1.28: Cumulative and Month-by-Month Forecast Classifier (Assumption Set 3, Whole Dataset).py

```
#Importing libraries
import xgboost as xgb
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, f1_score
from sklearn.metrics import roc_auc_score
from sklearn.metrics import average_precision_score
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.utils import resample
from imblearn.over_sampling import SMOTE
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import StratifiedKFold

#Setting a random seed for reproducibility
Seed = 43

#Reading in the data
Data = pd.read_csv("Merged Dataset (Whole Dataset, Assumption Set 3).csv")

#Declaring X and Y
X = Data[["Starting Amount", "Ending Amount", "Net Change over Forecast
Period", "Highest Amount", "Lowest Amount", "Difference"]]
Y = Data["status"]

#Label encoding Y
Encoder = LabelEncoder()
YEncoded = Encoder.fit_transform(Y)
print(Encoder.classes_)

#Splitting the data into training and testing sets
XTrain, XTest, YTrain, YTest = train_test_split(X, YEncoded, test_size=0.3,
stratify=YEncoded, random_state=Seed)

#Determining positive class weighting
NumNegative = sum(YTrain == 0)
NumPositive = sum(YTrain == 1)
PositiveWeighting = NumNegative/NumPositive

#Declaring parameter grid for the grid search
Parameters = {"n_estimators" : [25, 50, 100, 200],
              "eta" : [0.01, 0.05, 0.1, 0.2],
              "max_depth" : [1, 2, 4, 8]}

#Conducting a grid search to find the best XGBoost parameters
print("Train Length After Resampling:",len(YTrain))
print("Test Length:",len(YTest))
```

```
Model = xgb.XGBClassifier(n_estimators=100, objective='binary:logistic',
tree_method='hist', scale_pos_weight=PositiveWeighting, random_state=Seed)
Validator = StratifiedKFold(n_splits=10, shuffle=True, random_state=Seed)
Search = GridSearchCV(Model, Parameters, scoring="roc_auc", n_jobs=1,
cv=Validator)
Results = Search.fit(XTrain, YTrain)

#Picking the best model, making predictions with it and outputting the results
BestModel = Results.best_estimator_

RegularPredictions = BestModel.predict(XTest)
Accuracy = accuracy_score(RegularPredictions, YTest)
Probabilities = BestModel.predict_proba(XTest)[:,1]
AUC = roc_auc_score(YTest, Probabilities)
PR_AUC = average_precision_score(YTest, Probabilities, pos_label=0)
Metrics = classification_report(YTest, RegularPredictions,
target_names=["Defaulted", "Repaid"])
print(Metrics)
ConfusionMatrix = confusion_matrix(YTest, RegularPredictions)
print(ConfusionMatrix)
```

6.2.1.29 Appendix 2.1.29: Traditional Variables Classifier (Assumption Set 3, Whole Dataset).py

```
#Importing libraries
import xgboost as xgb
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, f1_score
from sklearn.metrics import roc_auc_score
from sklearn.metrics import average_precision_score
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.utils import resample
from imblearn.over_sampling import SMOTE
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import StratifiedKFold

#Declaring a random seed for reproducibility
Seed = 43

#Reading in the data
Data = pd.read_csv("Merged Dataset (Whole Dataset, Assumption Set 3).csv")

#Declaring X and Y
X = Data[["primary_sector", "company_type", "2019_revenue", "costs",
"accounts_receivable", "capital_and_reserves", "current_assets",
"current_liabilities",
        "fixed_assets", "long_term_liabilities", "provisions_for_liabilities",
"loanAmount", "yearsOfCreditHistory", "totalCreditLines", "openCreditLines",
        "bankruptcies", "delinquencies", "netValue"]]
Y = Data["status"]

#One-hot encoding categorical X variables
XNumerical = X.drop(columns=["primary_sector", "company_type"])
XCategorical = X[["primary_sector", "company_type"]]

OneHotEncoder = OneHotEncoder(sparse_output=False)
XCategoricalEncoded =
pd.DataFrame(OneHotEncoder.fit_transform(XCategorical),
columns=OneHotEncoder.get_feature_names_out(XCategorical.columns))

XEncoded = pd.concat([XNumerical.reset_index(drop=True),
XCategoricalEncoded.reset_index(drop=True)], axis=1)

#Label encoding Y
LblEncoder = LabelEncoder()
YEncoded = LblEncoder.fit_transform(Y)
print(LblEncoder.classes_)
```

```
#Splitting the data into training and testing sets
XTrain, XTest, YTrain, YTest = train_test_split(XEncoded, YEncoded,
test_size=0.3, stratify=YEncoded, random_state=Seed)

#Determining the positive class weighting
NumNegative = sum(YTrain == 0)
NumPositive = sum(YTrain == 1)
PositiveWeighting = NumNegative/NumPositive

#Declaring the parameter grid for the grid search
Parameters = {"n_estimators" : [25, 50, 100, 200],
              "eta" : [0.01, 0.05, 0.1, 0.2],
              "max_depth" : [1, 2, 4, 8]}

#Performing a grid search to determine the best XGBoost parameters
print("Train Length After Resampling:",len(YTrain))
print("Test Length:",len(YTest))
Model = xgb.XGBClassifier(n_estimators=100, objective='binary:logistic',
tree_method='hist', scale_pos_weight=PositiveWeighting, random_state=Seed)
Validator = StratifiedKFold(n_splits=10, shuffle=True, random_state=Seed)
Search = GridSearchCV(Model, Parameters, scoring="roc_auc", n_jobs=1,
cv=Validator)
Results = Search.fit(XTrain, YTrain)

#Taking the best model, making predictions with it and outputting the results
BestModel = Results.best_estimator_

RegularPredictions = BestModel.predict(XTest)
Accuracy = accuracy_score(RegularPredictions, YTest)
Probabilities = BestModel.predict_proba(XTest)[:,1]
AUC = roc_auc_score(YTest, Probabilities)
PR_AUC = average_precision_score(YTest, Probabilities, pos_label=0)
Metrics = classification_report(YTest, RegularPredictions,
target_names=["Defaulted", "Repaid"])
print(Metrics)
ConfusionMatrix = confusion_matrix(YTest, RegularPredictions)
print(ConfusionMatrix)
```

6.2.1.30 Appendix 2.1.30: Traditional Variables and Forecast Classifier (Assumption Set 3, Whole Dataset).py

```
#Importing libraries
import xgboost as xgb
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, f1_score
from sklearn.metrics import roc_auc_score
from sklearn.metrics import average_precision_score
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.utils import resample
from imblearn.over_sampling import SMOTE
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import StratifiedKFold

#Declaring a random seed for reproducibility
Seed = 43

#Reading in the data
Data = pd.read_csv("Merged Dataset (Whole Dataset, Assumption Set 3).csv")

#Declaring X and Y
X = Data[["primary_sector", "company_type", "2019_revenue", "costs",
"accounts_receivable", "capital_and_reserves", "current_assets",
"current_liabilities",
        "fixed_assets", "long_term_liabilities", "provisions_for_liabilities",
"loanAmount", "yearsOfCreditHistory", "totalCreditLines", "openCreditLines",
        "bankruptcies", "delinquencies", "netValue", "Highest Amount", "Lowest
Amount", "Difference"]]
Y = Data["status"]

#One-hot encoding categorical X columns
XNumerical = X.drop(columns=["primary_sector", "company_type"])
XCategorical = X[["primary_sector", "company_type"]]

OneHotEncoder = OneHotEncoder(sparse_output=False)
XCategoricalEncoded =
pd.DataFrame(OneHotEncoder.fit_transform(XCategorical),
columns=OneHotEncoder.get_feature_names_out(XCategorical.columns))

XEncoded = pd.concat([XNumerical.reset_index(drop=True),
XCategoricalEncoded.reset_index(drop=True)], axis=1)

#Label encoding Y
LblEncoder = LabelEncoder()
YEncoded = LblEncoder.fit_transform(Y)
print(LblEncoder.classes_)
```

```
#Splitting the data into training and testing sets
XTrain, XTest, YTrain, YTest = train_test_split(XEncoded, YEncoded,
test_size=0.3, stratify=YEncoded, random_state=Seed)

#Determining the positive class weighting
NumNegative = sum(YTrain == 0)
NumPositive = sum(YTrain == 1)
PositiveWeighting = NumNegative/NumPositive

#Declaring the parameter grid for the grid search
Parameters = {"n_estimators" : [25, 50, 100, 200],
              "eta" : [0.01, 0.05, 0.1, 0.2],
              "max_depth" : [1, 2, 4, 8]}

#Performing a grid search to find the best XGBoost parameters
print("Train Length After Resampling:",len(YTrain))
print("Test Length:",len(YTest))
Model = xgb.XGBClassifier(n_estimators=100, objective='binary:logistic',
tree_method='hist', scale_pos_weight=PositiveWeighting, random_state=Seed)
Validator = StratifiedKFold(n_splits=10, shuffle=True, random_state=Seed)
Search = GridSearchCV(Model, Parameters, scoring="roc_auc", n_jobs=1,
cv=Validator)
Results = Search.fit(XTrain, YTrain)

#Taking the best model, making predictions with it and outputting the results
BestModel = Results.best_estimator_

RegularPredictions = BestModel.predict(XTest)
Accuracy = accuracy_score(RegularPredictions, YTest)
Probabilities = BestModel.predict_proba(XTest)[:,1]
AUC = roc_auc_score(YTest, Probabilities)
PR_AUC = average_precision_score(YTest, Probabilities, pos_label=0)
Metrics = classification_report(YTest, RegularPredictions,
target_names=["Defaulted", "Repaid"])
print(Metrics)
ConfusionMatrix = confusion_matrix(YTest, RegularPredictions)
print(ConfusionMatrix)
```

6.2.1.31 Appendix 2.1.31: Random Dummy Classifier (Assumption Set 1, Whole Dataset).py

```
#Importing libraries
import xgboost as xgb
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, f1_score
from sklearn.metrics import roc_auc_score
from sklearn.metrics import average_precision_score
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.utils import resample
from imblearn.over_sampling import SMOTE
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import StratifiedKFold
from sklearn.dummy import DummyClassifier

#Declaring a random seed for reproducibility
Seed = 43

#Reading in the data
Data = pd.read_csv("Merged Dataset (Whole Dataset, Assumption Set 1).csv")

#Declaring X and Y
X = Data[["primary_sector", "company_type", "2019_revenue", "costs",
"accounts_receivable", "capital_and_reserves", "current_assets",
"current_liabilities",
        "fixed_assets", "long_term_liabilities", "provisions_for_liabilities",
"loanAmount", "yearsOfCreditHistory", "totalCreditLines", "openCreditLines",
        "bankruptcies", "delinquencies", "netValue", "Highest Amount", "Lowest
Amount", "Difference", "Starting Amount", "Ending Amount",
        "Net Change over Forecast Period"]]
Y = Data["status"]

#One-hot encoding categorical X columns
XNumerical = X.drop(columns=["primary_sector", "company_type"])
XCategorical = X[["primary_sector", "company_type"]]

OneHotEncoder = OneHotEncoder(sparse_output=False)
XCategoricalEncoded =
pd.DataFrame(OneHotEncoder.fit_transform(XCategorical),
columns=OneHotEncoder.get_feature_names_out(XCategorical.columns))

XEncoded = pd.concat([XNumerical.reset_index(drop=True),
XCategoricalEncoded.reset_index(drop=True)], axis=1)

#Label encoding Y
LblEncoder = LabelEncoder()
```

```
YEncoded = LblEncoder.fit_transform(Y)
print(LblEncoder.classes_)

#Splitting the data into training and testing sets
XTrain, XTest, YTrain, YTest = train_test_split(XEncoded, YEncoded,
test_size=0.3, stratify=YEncoded, random_state=Seed)

#Making predictions with the dummy classifier and outputting the results
Classifier = DummyClassifier(strategy="uniform")
Classifier.fit(XTrain, YTrain)

RegularPredictions = Classifier.predict(XTest)
Metrics = classification_report(YTest, RegularPredictions,
target_names=["Defaulted", "Repaid"])
print(Metrics)
ConfusionMatrix = confusion_matrix(YTest, RegularPredictions)
print(ConfusionMatrix)
```

6.2.1.32 Appendix 2.1.32: Most Frequent Dummy Classifier (Assumption Set 1, Whole Dataset).py

```
#Importing libraries
import xgboost as xgb
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, f1_score
from sklearn.metrics import roc_auc_score
from sklearn.metrics import average_precision_score
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.utils import resample
from imblearn.over_sampling import SMOTE
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import StratifiedKFold
from sklearn.dummy import DummyClassifier

#Declaring a random seed for reproducibility
Seed = 43

#Reading in the data
Data = pd.read_csv("Merged Dataset (Whole Dataset, Assumption Set 1).csv")

#Declaring X and Y
X = Data[["primary_sector", "company_type", "2019_revenue", "costs",
"accounts_receivable", "capital_and_reserves", "current_assets",
"current_liabilities",
        "fixed_assets", "long_term_liabilities", "provisions_for_liabilities",
"loanAmount", "yearsOfCreditHistory", "totalCreditLines", "openCreditLines",
        "bankruptcies", "delinquencies", "netValue", "Highest Amount", "Lowest
Amount", "Difference", "Starting Amount", "Ending Amount",
        "Net Change over Forecast Period"]]
Y = Data["status"]

#One-hot encoding categorical X columns
XNumerical = X.drop(columns=["primary_sector", "company_type"])
XCategorical = X[["primary_sector", "company_type"]]

OneHotEncoder = OneHotEncoder(sparse_output=False)
XCategoricalEncoded =
pd.DataFrame(OneHotEncoder.fit_transform(XCategorical),
columns=OneHotEncoder.get_feature_names_out(XCategorical.columns))

XEncoded = pd.concat([XNumerical.reset_index(drop=True),
XCategoricalEncoded.reset_index(drop=True)], axis=1)

#Label encoding Y
LblEncoder = LabelEncoder()
```

```
YEncoded = LblEncoder.fit_transform(Y)
print(LblEncoder.classes_)

#Splitting the data into training and testing sets
XTrain, XTest, YTrain, YTest = train_test_split(XEncoded, YEncoded,
test_size=0.3, stratify=YEncoded, random_state=Seed)

#Making predictions with the dummy classifier and outputting the results
Classifier = DummyClassifier(strategy="most_frequent")
Classifier.fit(XTrain, YTrain)

RegularPredictions = Classifier.predict(XTest)
Metrics = classification_report(YTest, RegularPredictions,
target_names=["Defaulted", "Repaid"])
print(Metrics)
ConfusionMatrix = confusion_matrix(YTest, RegularPredictions)
print(ConfusionMatrix)
```

6.2.1.33 Appendix 2.1.33: Random Dummy Classifier (Assumption Set 2, Whole Dataset).py

```
#Importing libraries
import xgboost as xgb
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, f1_score
from sklearn.metrics import roc_auc_score
from sklearn.metrics import average_precision_score
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.utils import resample
from imblearn.over_sampling import SMOTE
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import StratifiedKFold
from sklearn.dummy import DummyClassifier

#Setting a random seed for reproducibility
Seed = 43

#Reading in the data
Data = pd.read_csv("Merged Dataset (Whole Dataset, Assumption Set 2).csv")

#Declaring X and Y
X = Data[["primary_sector", "company_type", "2019_revenue", "costs",
"accounts_receivable", "capital_and_reserves", "current_assets",
"current_liabilities",
        "fixed_assets", "long_term_liabilities", "provisions_for_liabilities",
"loanAmount", "yearsOfCreditHistory", "totalCreditLines", "openCreditLines",
        "bankruptcies", "delinquencies", "netValue", "Highest Amount", "Lowest
Amount", "Difference", "Starting Amount", "Ending Amount",
        "Net Change over Forecast Period"]]
Y = Data["status"]

#One-hot encoding categorical X variables
XNumerical = X.drop(columns=["primary_sector", "company_type"])
XCategorical = X[["primary_sector", "company_type"]]

OneHotEncoder = OneHotEncoder(sparse_output=False)
XCategoricalEncoded =
pd.DataFrame(OneHotEncoder.fit_transform(XCategorical),
columns=OneHotEncoder.get_feature_names_out(XCategorical.columns))

XEncoded = pd.concat([XNumerical.reset_index(drop=True),
XCategoricalEncoded.reset_index(drop=True)], axis=1)

#Label encoding Y
LblEncoder = LabelEncoder()
```

```
YEncoded = LblEncoder.fit_transform(Y)
print(LblEncoder.classes_)

#Splitting the dataset into training and testing sets
XTrain, XTest, YTrain, YTest = train_test_split(XEncoded, YEncoded,
test_size=0.3, stratify=YEncoded, random_state=Seed)

#Making predictions with the dummy classifier and outputting the results
Classifier = DummyClassifier(strategy="uniform")
Classifier.fit(XTrain, YTrain)

RegularPredictions = Classifier.predict(XTest)
Metrics = classification_report(YTest, RegularPredictions,
target_names=["Defaulted", "Repaid"])
print(Metrics)
ConfusionMatrix = confusion_matrix(YTest, RegularPredictions)
print(ConfusionMatrix)
```

6.2.1.34 Appendix 2.1.34: Most Frequent Dummy Classifier (Assumption Set 2, Whole Dataset).py

```
#Importing libraries
import xgboost as xgb
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, f1_score
from sklearn.metrics import roc_auc_score
from sklearn.metrics import average_precision_score
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.utils import resample
from imblearn.over_sampling import SMOTE
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import StratifiedKFold
from sklearn.dummy import DummyClassifier

#Declaring a random seed for reproducibility
Seed = 43

#Reading in the data
Data = pd.read_csv("Merged Dataset (Whole Dataset, Assumption Set 2).csv")

#Declaring X and Y
X = Data[["primary_sector", "company_type", "2019_revenue", "costs",
"accounts_receivable", "capital_and_reserves", "current_assets",
"current_liabilities",
        "fixed_assets", "long_term_liabilities", "provisions_for_liabilities",
"loanAmount", "yearsOfCreditHistory", "totalCreditLines", "openCreditLines",
        "bankruptcies", "delinquencies", "netValue", "Highest Amount", "Lowest
Amount", "Difference", "Starting Amount", "Ending Amount",
        "Net Change over Forecast Period"]]
Y = Data["status"]

#One-hot encoding categorical X variables
XNumerical = X.drop(columns=["primary_sector", "company_type"])
XCategorical = X[["primary_sector", "company_type"]]

OneHotEncoder = OneHotEncoder(sparse_output=False)
XCategoricalEncoded =
pd.DataFrame(OneHotEncoder.fit_transform(XCategorical),
columns=OneHotEncoder.get_feature_names_out(XCategorical.columns))

XEncoded = pd.concat([XNumerical.reset_index(drop=True),
XCategoricalEncoded.reset_index(drop=True)], axis=1)

#Label encoding Y
LblEncoder = LabelEncoder()
```

```
YEncoded = LblEncoder.fit_transform(Y)
print(LblEncoder.classes_)

#Splitting the data into training and testing sets
XTrain, XTest, YTrain, YTest = train_test_split(XEncoded, YEncoded,
test_size=0.3, stratify=YEncoded, random_state=Seed)

#Making predictions with the dummy classifier and outputting the results
Classifier = DummyClassifier(strategy="most_frequent")
Classifier.fit(XTrain, YTrain)

RegularPredictions = Classifier.predict(XTest)
Metrics = classification_report(YTest, RegularPredictions,
target_names=["Defaulted", "Repaid"])
print(Metrics)
ConfusionMatrix = confusion_matrix(YTest, RegularPredictions)
print(ConfusionMatrix)
```

6.2.1.35 Appendix 2.1.35: Random Dummy Classifier (Assumption Set 3, Whole Dataset).py

```
#Importing libraries
import xgboost as xgb
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, f1_score
from sklearn.metrics import roc_auc_score
from sklearn.metrics import average_precision_score
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.utils import resample
from imblearn.over_sampling import SMOTE
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import StratifiedKFold
from sklearn.dummy import DummyClassifier

#Setting random seed for reproducibility
Seed = 43

#Reading in the data
Data = pd.read_csv("Merged Dataset (Whole Dataset, Assumption Set 3).csv")

#Declaring X and Y
X = Data[["primary_sector", "company_type", "2019_revenue", "costs",
"accounts_receivable", "capital_and_reserves", "current_assets",
"current_liabilities",
        "fixed_assets", "long_term_liabilities", "provisions_for_liabilities",
"loanAmount", "yearsOfCreditHistory", "totalCreditLines", "openCreditLines",
        "bankruptcies", "delinquencies", "netValue", "Highest Amount", "Lowest
Amount", "Difference", "Starting Amount", "Ending Amount",
        "Net Change over Forecast Period"]]
Y = Data["status"]

#One-hot encoding categorical X variables
XNumerical = X.drop(columns=["primary_sector", "company_type"])
XCategorical = X[["primary_sector", "company_type"]]

OneHotEncoder = OneHotEncoder(sparse_output=False)
XCategoricalEncoded =
pd.DataFrame(OneHotEncoder.fit_transform(XCategorical),
columns=OneHotEncoder.get_feature_names_out(XCategorical.columns))

XEncoded = pd.concat([XNumerical.reset_index(drop=True),
XCategoricalEncoded.reset_index(drop=True)], axis=1)

#Label encoding Y
LblEncoder = LabelEncoder()
```

```
YEncoded = LblEncoder.fit_transform(Y)
print(LblEncoder.classes_)

#Splitting the data into training and testing sets
XTrain, XTest, YTrain, YTest = train_test_split(XEncoded, YEncoded,
test_size=0.3, stratify=YEncoded, random_state=Seed)

#Making predictions with the dummy classifier and outputting the results
Classifier = DummyClassifier(strategy="uniform")
Classifier.fit(XTrain, YTrain)

RegularPredictions = Classifier.predict(XTest)
Metrics = classification_report(YTest, RegularPredictions,
target_names=["Defaulted", "Repaid"])
print(Metrics)
ConfusionMatrix = confusion_matrix(YTest, RegularPredictions)
print(ConfusionMatrix)
```

6.2.1.36 Appendix 2.1.36: Most Frequent Dummy Classifier (Assumption Set 3, Whole Dataset).py

```
#Importing libraries
import xgboost as xgb
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, f1_score
from sklearn.metrics import roc_auc_score
from sklearn.metrics import average_precision_score
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.utils import resample
from imblearn.over_sampling import SMOTE
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import StratifiedKFold
from sklearn.dummy import DummyClassifier

#Setting random seed for reproducibility
Seed = 43

#Reading in the data
Data = pd.read_csv("Merged Dataset (Whole Dataset, Assumption Set 3).csv")

#Declaring X and Y
X = Data[["primary_sector", "company_type", "2019_revenue", "costs",
"accounts_receivable", "capital_and_reserves", "current_assets",
"current_liabilities",
        "fixed_assets", "long_term_liabilities", "provisions_for_liabilities",
"loanAmount", "yearsOfCreditHistory", "totalCreditLines", "openCreditLines",
        "bankruptcies", "delinquencies", "netValue", "Highest Amount", "Lowest
Amount", "Difference", "Starting Amount", "Ending Amount",
        "Net Change over Forecast Period"]]
Y = Data["status"]

#One-hot encoding categorical X variables
XNumerical = X.drop(columns=["primary_sector", "company_type"])
XCategorical = X[["primary_sector", "company_type"]]

OneHotEncoder = OneHotEncoder(sparse_output=False)
XCategoricalEncoded =
pd.DataFrame(OneHotEncoder.fit_transform(XCategorical),
columns=OneHotEncoder.get_feature_names_out(XCategorical.columns))

XEncoded = pd.concat([XNumerical.reset_index(drop=True),
XCategoricalEncoded.reset_index(drop=True)], axis=1)

#Label encoding Y
LblEncoder = LabelEncoder()
```

```
YEncoded = LblEncoder.fit_transform(Y)
print(LblEncoder.classes_)

#Splitting the dataset into training and testing sets
XTrain, XTest, YTrain, YTest = train_test_split(XEncoded, YEncoded,
test_size=0.3, stratify=YEncoded, random_state=Seed)

#Making predictions with the dummy classifier and outputting the results
Classifier = DummyClassifier(strategy="most_frequent")
Classifier.fit(XTrain, YTrain)

RegularPredictions = Classifier.predict(XTest)
Metrics = classification_report(YTest, RegularPredictions,
target_names=["Defaulted", "Repaid"])
print(Metrics)
ConfusionMatrix = confusion_matrix(YTest, RegularPredictions)
print(ConfusionMatrix)
```

6.2.1.37 Appendix 2.1.37: Assumption Set 1 Classification Result Visualisation.py

```
#Importing libraries
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

#Reading in the data
Data = pd.read_csv("Assumption Set 1 Classification Results (Whole Dataset).csv")

#Declaring x and y columns
x = np.arange(len(Data["Variables"]))
y1 = Data["Precision"]
y2 = Data["Recall"]

#Declaring bar width
Width = 0.35

#Visualising results
plt.bar(x - Width/2, y1, Width, label="Average Precision")
plt.bar(x + Width/2, y2, Width, label="Average Recall")
plt.xticks(x, Data["Variables"])
plt.xlabel("Variable Subset")
plt.ylabel("Precision or Recall Score (%)")
plt.title("Results for Assumption Set 1")
plt.legend()
plt.show()
```

6.2.1.38 Appendix 2.1.38: Assumption Set 2 Classification Result Visualisation.py

```
#Importing libraries
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

#Reading in the data
Data = pd.read_csv("Assumption Set 2 Classification Results (Whole Dataset).csv")

#Declaring X and Y columns
x = np.arange(len(Data["Variables"]))
y1 = Data["Precision"]
y2 = Data["Recall"]

#Declaring bar width
Width = 0.35

#Visualising results
plt.bar(x - Width/2, y1, Width, label="Average Precision")
plt.bar(x + Width/2, y2, Width, label="Average Recall")
plt.xticks(x, Data["Variables"])
plt.xlabel("Variable Subset")
plt.ylabel("Precision or Recall Score (%)")
plt.title("Results for Assumption Set 2")
plt.legend()
plt.show()
```

6.2.1.39 Appendix 2.1.39: Assumption Set 3 Classification Result Visualisation.py

```
#Importing libraries
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

#Reading in the data
Data = pd.read_csv("Assumption Set 3 Classification Results (Whole Dataset).csv")

#Declaring X and Y variables
x = np.arange(len(Data["Variables"]))
y1 = Data["Precision"]
y2 = Data["Recall"]

#Declaring bar width
Width = 0.35

#Visualising results
plt.bar(x - Width/2, y1, Width, label="Average Precision")
plt.bar(x + Width/2, y2, Width, label="Average Recall")
plt.xticks(x, Data["Variables"])
plt.xlabel("Variable Subset")
plt.ylabel("Precision or Recall Score (%)")
plt.title("Results for Assumption Set 3")
plt.legend()
plt.show()
```

6.2.2 Appendix 2.2: Python Notebooks (.ipynb files)

These files were originally Python notebooks, for memory reasons. While they are given here as .py files, these should be executed as Python notebooks (.ipynb files) in a notebook environment such as Jupyter or Google Colab. The separation between notebook code blocks is denoted by docstrings, and the URL of the original file on Google Colab is given in the docstring at the top.

6.2.2.1 Appendix 2.2.1: LSTM Forecasts (Cumulative).py (converted from .ipynb)

```
# -*- coding: utf-8 -*-
"""LSTM Forecasts (Cumulative).ipynb

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/1TbVgINQ545K5xJKluIde4-rHWGO99uQc

**Importing Libraries and Data**
"""

#Importing libraries
import pandas as pd
import statsmodels.api as sm
from sklearn.metrics import root_mean_squared_error
from statsmodels.tsa.stattools import acf
import matplotlib.pyplot as plt
import prophet
import numpy as np
import warnings
import gc
import os
import sys
import json
from keras.models import Sequential
from keras.layers import LSTM
from keras.layers import Dense
from keras.layers import Dropout
from keras.callbacks import EarlyStopping
from tensorflow.keras.preprocessing.sequence import TimeseriesGenerator
from sklearn.preprocessing import MinMaxScaler

#Ignoring warnings
warnings.filterwarnings("ignore")

#Reading in transaction dataset
BoostedTransactions = pd.read_csv("Boosted Transaction Dataset.csv")
AccountIDs = BoostedTransactions["AccountId"].values
```

```
#Generating a set of the unique accounts
UniqueAccounts = set(AccountIDs)

#Converting the transaction date values to dates and converting them to months
BoostedTransactions["TransactionDate"] =
pd.to_datetime(BoostedTransactions["TransactionDate"])
BoostedTransactions["TransactionDate"] =
BoostedTransactions["TransactionDate"].dt.to_period("M").dt.to_timestamp()

#Aggregating the net sum of transactions by account and month
GroupedSumOfTransactionsByAccountAndMonth =
BoostedTransactions.groupby(by=["AccountId",
"TransactionDate"]).agg(["sum"])["Amount"]
print(GroupedSumOfTransactionsByAccountAndMonth)

#Creating a list in which to store the complete time series for each account
DataFrames = []

#Filling in gaps in each account's time series to ensure that no months are missing
for i in range(0, len(UniqueAccounts)):
  #Selecting only the data for the selected account ID
  ResetDataFrame =
GroupedSumOfTransactionsByAccountAndMonth.reset_index()
  TimeSeries = ResetDataFrame[ResetDataFrame["AccountId"] ==
list(UniqueAccounts)[i]]
  TimeSeries = TimeSeries.set_index("TransactionDate")

  #Resampling the time series to fill in blank months with 0 values
  TimeSeries = TimeSeries.resample("MS").sum()
  #Making the account ID value for every account equal the one selected
  for j in range(0, len(TimeSeries)):
    if TimeSeries.iloc[j]["AccountId"] == 0:
      TimeSeries.at[TimeSeries.index[j], "AccountId"] = list(UniqueAccounts)[i]

  #Adding the resampled time series to the end of the DataFrames list
  DataFrames.append(TimeSeries)

#Concatenating the time series together to make a new grouped dataset of all
accounts' time series
NewGroupedDataset = pd.concat(DataFrames)
NewGroupedDataset = NewGroupedDataset.reset_index()
NewGroupedDataset = NewGroupedDataset.set_index(["AccountId",
"TransactionDate"])

#Showing the new dataset
print(NewGroupedDataset)

#Grouping the new dataset by account ID and calculating the cumulative sum of
transactions for each account across its time series
```

```
GroupedCumulativeSumOfTransactionsByAccountAndMonth =
NewGroupedDataset.groupby(level=0).cumsum().reset_index()
print(GroupedCumulativeSumOfTransactionsByAccountAndMonth)

"""**Writing JSON file of account IDs (only run before running the first batch)**

Only run this block before running the first batch to ensure that the order of accounts
remains the same and no accounts have duplicate forecasts performed.
"""

#Getting the account ID values
AccountIDs = BoostedTransactions["AccountId"].values
#Creating a list of the unique accounts
UniqueAccounts = list(dict.fromkeys(AccountIDs))

#Dumping the list into a JSON file to maintain order
with open("UniqueAccounts.json", "w") as File:
  json.dump(UniqueAccounts, File)

"""**Reading the JSON file and manipulating the data to ensure that all gaps are
filled**"""

#Reading the accounts from the JSON file
with open("UniqueAccounts.json", "r") as File:
  UniqueAccounts = json.load(File)

#Converting the transaction dates to date values and converting them to months
BoostedTransactions["TransactionDate"] =
pd.to_datetime(BoostedTransactions["TransactionDate"])
BoostedTransactions["TransactionDate"] =
BoostedTransactions["TransactionDate"].dt.to_period("M").dt.to_timestamp()

#Grouping the data by account and month and calculating the net sum of transactions
for each account in each month
GroupedSumOfTransactionsByAccountAndMonth =
BoostedTransactions.groupby(by=["AccountId",
"TransactionDate"]).agg(["sum"])["Amount"]
print(GroupedSumOfTransactionsByAccountAndMonth)

#Creating a list to store the accounts' resampled time series in
DataFrames = []

#Filling in gaps in each account's time series
for i in range(0, len(UniqueAccounts)):
  #Filtering the grouped DataFrame to only show months for the chosen account
  ResetDataFrame =
GroupedSumOfTransactionsByAccountAndMonth.reset_index()
  TimeSeries = ResetDataFrame[ResetDataFrame["AccountId"] ==
list(UniqueAccounts)[i]]
  TimeSeries = TimeSeries.set_index("TransactionDate")
```

```
  #Filling in month gaps in the time series with 0 values to ensure that empty months
are not missed out of the time series
  TimeSeries = TimeSeries.resample("MS").sum()
  #Setting the account ID value throughout the time series to the chosen account ID
value where this is not already the case
  #This is to ensure that months can be correctly selected by account
  for j in range(0, len(TimeSeries)):
    if TimeSeries.iloc[j]["AccountId"] == 0:
      TimeSeries.at[TimeSeries.index[j], "AccountId"] = list(UniqueAccounts)[i]

  #Adding the time series on the end of the DataFrames list
  DataFrames.append(TimeSeries)

#Merging the list of time series into one grouped dataset, containing all
month/account combinations and their net transaction amounts
NewGroupedDataset = pd.concat(DataFrames)
NewGroupedDataset = NewGroupedDataset.reset_index()
NewGroupedDataset = NewGroupedDataset.set_index(["AccountId",
"TransactionDate"])

#Showing the new grouped dataset
print(NewGroupedDataset)

#Grouping the new dataset by account ID and working out the cumulative sum of
transaction amounts over the time series for each account
GroupedCumulativeSumOfTransactionsByAccountAndMonth =
NewGroupedDataset.groupby(level=0).cumsum().reset_index()
print(GroupedCumulativeSumOfTransactionsByAccountAndMonth)

"""**Key LSTM Loops (split up for memory reasons)**

Batch for accounts 0-150
"""

#Creating models and forecasting for the first 150 accounts
for idx, i in enumerate(UniqueAccounts):
    #Ensuring that only the first 150 accounts are used in this batch
    if idx >= 150:
        break
    #Printing the account ID being tested
    print(i)
    try:
        #Fetching the time series for the chosen account
        ForecastingTimeSeries =
GroupedCumulativeSumOfTransactionsByAccountAndMonth[GroupedCumulative
SumOfTransactionsByAccountAndMonth["AccountId"] == i][["TransactionDate",
"sum"]]
```

```
        #Calculating where the cut-off point between training data and testing data will
lie
        TrainEnd = int(0.7 * len(ForecastingTimeSeries))

        #Splitting the data into training and testing sets
        TrainData = ForecastingTimeSeries[:TrainEnd]
        TestData = ForecastingTimeSeries[TrainEnd:]

        #Setting transaction month as the training data index
        TrainData = TrainData.set_index("TransactionDate")

        #Scaling the values in the training dataset using min/max scaling
        Scaler = MinMaxScaler(feature_range=(0,1))
        ScaledTrain = Scaler.fit_transform(TrainData)

        #Generating a batched time series that can be processed by the LSTM
        TimeStep = 1
        TimeSeriesData = TimeseriesGenerator(ScaledTrain, ScaledTrain,
length=TimeStep, batch_size=1)

        #Declaring the various network layers
        Model = Sequential()
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh",
input_shape=(TimeStep, 1)))
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh"))
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh"))
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh"))
        Model.add(LSTM(units=16, activation="tanh"))
        Model.add(Dense(units=1, activation="sigmoid"))
        Model.add(Dense(units=1, activation="sigmoid"))
        Model.add(Dense(1))
        Model.compile(optimizer="adam", loss="mean_squared_error")

        #Fitting the model
        Model.fit(TimeSeriesData, epochs=20, batch_size=1, verbose=0)

        #Generating a list for test dataset predictions to be stored in
        TestPredictions = []

        #Starting to create the first testing batch
        FirstTestingBatch = ScaledTrain[-TimeStep:]
        #print(FirstTestingBatch)
        CurrentBatch = FirstTestingBatch.reshape((1, TimeStep, 1))

        #Making test predictions
        for h in range(len(TestData)):
            CurrentPrediction = Model.predict(CurrentBatch)[0]
            TestPredictions.append(CurrentPrediction)
            CurrentBatch = np.append(CurrentBatch[:, 1:, :], [[CurrentPrediction]],
axis=1)
```

```
        #Inverting the scaling of test predictions
        Forecast = Scaler.inverse_transform(TestPredictions)

        #Calculating RMSE and SI figures and writing the errors for this account to the
errors CSV file (if not already created, it will be created)
        try:
          RMSE = root_mean_squared_error(Forecast, TestData["sum"])
          SI = abs(RMSE/abs(TestData["sum"]).mean())
          AccountError = pd.DataFrame({"AccountID" : [i],
                                       "RMSE" : [RMSE],
                                       "SI" : [SI]})
          AccountError.to_csv("LSTM Errors (Cumulative).csv", mode='a', header=not
os.path.exists("LSTM Errors (Cumulative).csv"), index=False)
        #If a ValueError occurred, the loop is broken
        except ValueError:
          print("ValueError encountered")
          break

        #Creating a list of future predictions
        FuturePredictions = []

        #Creating the first future batch
        FirstFutureBatch = np.array(TestPredictions[-TimeStep:])
        CurrentBatch = FirstFutureBatch.reshape((1, TimeStep, 1))

        #Forecasting the next 12 months
        for j in range(12):
            CurrentPrediction = Model.predict(CurrentBatch)[0]
            FuturePredictions.append(CurrentPrediction)
            CurrentBatch = np.append(CurrentBatch[:, 1:, :], [[CurrentPrediction]],
axis=1)

        #Unscaling the future forecast values
        FutureForecast = Scaler.inverse_transform(FuturePredictions)

        #Calculating starting amount, ending amount and net change over time, adding
them to a DataFrame and writing this to the forecasts CSV
        NetChange = FutureForecast[-1] - FutureForecast[0]
        StartingAmount = FutureForecast[0]
        EndingAmount = FutureForecast[-1]

        ForecastData = pd.DataFrame({"AccountID" : [i],
                                     "Net Change over Forecast Period" : [NetChange],
                                     "Starting Amount" : [StartingAmount],
                                     "Ending Amount" : [EndingAmount]})

        ForecastData.to_csv("LSTM Forecasts (Cumulative).csv", mode='a',
header=not os.path.exists("LSTM Forecasts (Cumulative).csv"), index=False)
```

```
        #Informing the user of successful account processing
        print("Processed account number",i)
    #If an exception has occurred, the user is told about it
    except Exception as e:
        print(f"Error encountered processing account number {i}.")
        ErrorType, ErrorObject, ErrorTraceback = sys.exc_info()

        ErrorFilename = os.path.split(
            ErrorTraceback.tb_frame.f_code.co_filename
        )[1]

        ErrorMessage = str(e)

        ErrorLineNumber = ErrorTraceback.tb_lineno

        print(f'Exception Type: {ErrorType}')

        print(f'Exception Filename: {ErrorFilename}')

        print(f'Exception Line Number: {ErrorLineNumber}')

        print(f'Exception Message: {ErrorMessage}')
        break
    #Deleting all model info from memory to conserve RAM
    finally:
        del Model, Forecast, TrainData, TestData, ForecastingTimeSeries
        gc.collect()

"""Batch for accounts 150-300"""

#Creating models and forecasting for accounts 150-300
for idx, i in enumerate(UniqueAccounts):
    #Ensuring that only accounts 150-300 are processed in this batch
    if idx < 150:
        pass
    else:
      if idx >= 300:
        break
      #Printing the account ID being tested
      print(i)
      try:
          #Selecting only the account/month combination totals for the selected account
          ForecastingTimeSeries =
GroupedCumulativeSumOfTransactionsByAccountAndMonth[GroupedCumulative
SumOfTransactionsByAccountAndMonth["AccountId"] == i][["TransactionDate",
"sum"]]

          #Calculating the cutoff point between training data and testing data
          TrainEnd = int(0.7 * len(ForecastingTimeSeries))
```

```
        #Splitting the data into training and testing sets
        TrainData = ForecastingTimeSeries[:TrainEnd]
        TestData = ForecastingTimeSeries[TrainEnd:]

        #Setting transaction month as an index column
        TrainData = TrainData.set_index("TransactionDate")

        #Scaling the data using min/max scaling
        Scaler = MinMaxScaler(feature_range=(0,1))
        ScaledTrain = Scaler.fit_transform(TrainData)

        #Generating a batched time series that can be processed by the LSTM
        TimeStep = 1
        TimeSeriesData = TimeseriesGenerator(ScaledTrain, ScaledTrain,
length=TimeStep, batch_size=1)

        #Declaring the model layers
        Model = Sequential()
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh",
input_shape=(TimeStep, 1)))
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh"))
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh"))
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh"))
        Model.add(LSTM(units=16, activation="tanh"))
        Model.add(Dense(units=1, activation="sigmoid"))
        Model.add(Dense(units=1, activation="sigmoid"))
        Model.add(Dense(1))
        Model.compile(optimizer="adam", loss="mean_squared_error")

        #Fitting the model on the training data
        Model.fit(TimeSeriesData, epochs=20, batch_size=1, verbose=0)

        #Creating a list to store test dataset predictions within
        TestPredictions = []

        #Creating the first testing batch
        FirstTestingBatch = ScaledTrain[-TimeStep:]
        CurrentBatch = FirstTestingBatch.reshape((1, TimeStep, 1))

        #Making test dataset predictions
        for h in range(len(TestData)):
            CurrentPrediction = Model.predict(CurrentBatch)[0]
            TestPredictions.append(CurrentPrediction)
            CurrentBatch = np.append(CurrentBatch[:, 1:, :], [[CurrentPrediction]],
axis=1)

        #Unscaling test dataset predictions
        Forecast = Scaler.inverse_transform(TestPredictions)

        #Calculating error rates for this model and writing them to the CSV
```

```
        try:
          RMSE = root_mean_squared_error(Forecast, TestData["sum"])
          SI = abs(RMSE/abs(TestData["sum"]).mean())
          AccountError = pd.DataFrame({"AccountID" : [i],
                                       "RMSE" : [RMSE],
                                       "SI" : [SI]})
          AccountError.to_csv("LSTM Errors (Cumulative).csv", mode='a',
header=not os.path.exists("LSTM Errors (Cumulative).csv"), index=False)
        #Where a ValueError occurs, the loop is broken
        except ValueError:
          print("ValueError encountered")
          break

        #Creating a list to store future predictions
        FuturePredictions = []

        #Generating the first future batch
        FirstFutureBatch = np.array(TestPredictions[-TimeStep:])
        CurrentBatch = FirstFutureBatch.reshape((1, TimeStep, 1))

        #Making a forecast of the next 12 months
        for j in range(12):
           CurrentPrediction = Model.predict(CurrentBatch)[0]
           FuturePredictions.append(CurrentPrediction)
           CurrentBatch = np.append(CurrentBatch[:, 1:, :], [[CurrentPrediction]],
axis=1)

        #Unscaling the future forecast values
        FutureForecast = Scaler.inverse_transform(FuturePredictions)

        #Calculating starting amount, ending amount and net change and appending
this to the forecasts CSV
        NetChange = FutureForecast[-1] - FutureForecast[0]
        StartingAmount = FutureForecast[0]
        EndingAmount = FutureForecast[-1]

        ForecastData = pd.DataFrame({"AccountID" : [i],
                             "Net Change over Forecast Period" : [NetChange],
                             "Starting Amount" : [StartingAmount],
                             "Ending Amount" : [EndingAmount]})

        ForecastData.to_csv("LSTM Forecasts (Cumulative).csv", mode='a',
header=not os.path.exists("LSTM Forecasts (Cumulative).csv"), index=False)

        #Informing the user of successful execution
        print("Processed account number",i)
    #Where an error occurs, the loop is broken and the user is informed about it
    except Exception as e:
        print(f"Error encountered processing account number {i}.")
        ErrorType, ErrorObject, ErrorTraceback = sys.exc_info()
```

```
        ErrorFilename = os.path.split(
            ErrorTraceback.tb_frame.f_code.co_filename
        )[1]

        ErrorMessage = str(e)

        ErrorLineNumber = ErrorTraceback.tb_lineno

        print(f'Exception Type: {ErrorType}')

        print(f'Exception Filename: {ErrorFilename}')

        print(f'Exception Line Number: {ErrorLineNumber}')

        print(f'Exception Message: {ErrorMessage}')
        break
    #Deleting all model info from memory to conserve RAM
    finally:
        del Model, Forecast, TrainData, TestData, ForecastingTimeSeries
        gc.collect()

"""Batch for accounts 300-450"""

#Creating models and forecasting for accounts 300-450
for idx, i in enumerate(UniqueAccounts):
    #Ensuring that only accounts 300-450 are processed in this batch
    if idx < 300:
        pass
    else:
      if idx >= 450:
        break
      #Printing the account ID being processed
      print(i)
      try:
        #Selecting only the account/month combination totals for the selected account
        ForecastingTimeSeries =
GroupedCumulativeSumOfTransactionsByAccountAndMonth[GroupedCumulative
SumOfTransactionsByAccountAndMonth["AccountId"] == i][["TransactionDate",
"sum"]]

        #Calculating the cutoff point between training and testing data
        TrainEnd = int(0.7 * len(ForecastingTimeSeries))

        #Splitting the data into training and testing sets
        TrainData = ForecastingTimeSeries[:TrainEnd]
        TestData = ForecastingTimeSeries[TrainEnd:]

        #Setting transaction month as the index
        TrainData = TrainData.set_index("TransactionDate")
```

```
        #Scaling the training data using min/max scaling
        Scaler = MinMaxScaler(feature_range=(0,1))
        ScaledTrain = Scaler.fit_transform(TrainData)

        #Generating a batched time series that the LSTM can process
        TimeStep = 1
        TimeSeriesData = TimeseriesGenerator(ScaledTrain, ScaledTrain,
length=TimeStep, batch_size=1)

        #Declaring the model layers
        Model = Sequential()
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh",
input_shape=(TimeStep, 1)))
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh"))
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh"))
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh"))
        Model.add(LSTM(units=16, activation="tanh"))
        Model.add(Dense(units=1, activation="sigmoid"))
        Model.add(Dense(units=1, activation="sigmoid"))
        Model.add(Dense(1))
        Model.compile(optimizer="adam", loss="mean_squared_error")

        #Fitting the model to the training data
        Model.fit(TimeSeriesData, epochs=20, batch_size=1, verbose=0)

        #Declaring a list to store the test data predictions
        TestPredictions = []

        #Creating the first testing batch
        FirstTestingBatch = ScaledTrain[-TimeStep:]
        CurrentBatch = FirstTestingBatch.reshape((1, TimeStep, 1))

        #Making predictions on the test dataset
        for h in range(len(TestData)):
            CurrentPrediction = Model.predict(CurrentBatch)[0]
            TestPredictions.append(CurrentPrediction)
            CurrentBatch = np.append(CurrentBatch[:, 1:, :], [[CurrentPrediction]],
axis=1)

        #Unscaling the test dataset predictions
        Forecast = Scaler.inverse_transform(TestPredictions)

        #Calculating error rates for the model and appending them to the errors CSV
        try:
          RMSE = root_mean_squared_error(Forecast, TestData["sum"])
          SI = abs(RMSE/abs(TestData["sum"]).mean())
          AccountError = pd.DataFrame({"AccountID" : [i],
                                       "RMSE" : [RMSE],
                                       "SI" : [SI]})
```

```
            AccountError.to_csv("LSTM Errors (Cumulative).csv", mode='a',
header=not os.path.exists("LSTM Errors (Cumulative).csv"), index=False)
        #Where a ValueError occurs, the loop is broken
        except ValueError:
          print("ValueError encountered")
          break

        #Creating a list to store future predictions
        FuturePredictions = []

        #Creating the first future batch
        FirstFutureBatch = np.array(TestPredictions[-TimeStep:])
        CurrentBatch = FirstFutureBatch.reshape((1, TimeStep, 1))

        #Forecasting the next 12 months
        for j in range(12):
            CurrentPrediction = Model.predict(CurrentBatch)[0]
            FuturePredictions.append(CurrentPrediction)
            CurrentBatch = np.append(CurrentBatch[:, 1:, :], [[CurrentPrediction]],
axis=1)

        #Unscaling the forecast values
        FutureForecast = Scaler.inverse_transform(FuturePredictions)

        #Calculating starting amount, ending amount and net change and appending
them into the forecasts CSV
        NetChange = FutureForecast[-1] - FutureForecast[0]
        StartingAmount = FutureForecast[0]
        EndingAmount = FutureForecast[-1]

        ForecastData = pd.DataFrame({"AccountID" : [i],
                             "Net Change over Forecast Period" : [NetChange],
                             "Starting Amount" : [StartingAmount],
                             "Ending Amount" : [EndingAmount]})

        ForecastData.to_csv("LSTM Forecasts (Cumulative).csv", mode='a',
header=not os.path.exists("LSTM Forecasts (Cumulative).csv"), index=False)

        #Informing the user of successful execution
        print("Processed account number",i)
    except Exception as e:
        print(f"Error encountered processing account number {i}.")
        ErrorType, ErrorObject, ErrorTraceback = sys.exc_info()

        ErrorFilename = os.path.split(
            ErrorTraceback.tb_frame.f_code.co_filename
        )[1]

        ErrorMessage = str(e)
```

```
        ErrorLineNumber = ErrorTraceback.tb_lineno

        print(f'Exception Type: {ExceptionType}')

        print(f'Exception Filename: {ExceptionFilename}')

        print(f'Exception Line Number: {ExceptionLineNumber}')

        print(f'Exception Message: {ExceptionMessage}')
        break
    #Deleting model information from memory (to conserve RAM)
    finally:
        del Model, Forecast, TrainData, TestData, ForecastingTimeSeries
        gc.collect()

"""Batch for accounts 450-600"""

#Creating models and forecasting for accounts 450-600
for idx, i in enumerate(UniqueAccounts):
  #Ensuring that only accounts 450-600 are processed in this batch
  if idx < 450:
      pass
  else:
    if idx >= 600:
      break
    #Printing the account ID being processed
    print(i)
    try:
        #Selecting only this account's time series
        ForecastingTimeSeries =
GroupedCumulativeSumOfTransactionsByAccountAndMonth[GroupedCumulative
SumOfTransactionsByAccountAndMonth["AccountId"] == i][["TransactionDate",
"sum"]]

        #Calculating the cutoff point between training and testing sets
        TrainEnd = int(0.7 * len(ForecastingTimeSeries))

        #Splitting the data into training and testing sets
        TrainData = ForecastingTimeSeries[:TrainEnd]
        TestData = ForecastingTimeSeries[TrainEnd:]

        #Setting transaction month as an index
        TrainData = TrainData.set_index("TransactionDate")

        #Scaling the training data using min/max scaling
        Scaler = MinMaxScaler(feature_range=(0,1))
        ScaledTrain = Scaler.fit_transform(TrainData)

        #Creating a batched time series that the LSTM can process
        TimeStep = 1
```

```
        TimeSeriesData = TimeseriesGenerator(ScaledTrain, ScaledTrain,
length=TimeStep, batch_size=1)

        #Declaring the model layers
        Model = Sequential()
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh",
input_shape=(TimeStep, 1)))
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh"))
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh"))
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh"))
        Model.add(LSTM(units=16, activation="tanh"))
        Model.add(Dense(units=1, activation="sigmoid"))
        Model.add(Dense(units=1, activation="sigmoid"))
        Model.add(Dense(1))
        Model.compile(optimizer="adam", loss="mean_squared_error")

        #Fitting the model to the training data
        Model.fit(TimeSeriesData, epochs=20, batch_size=1, verbose=0)

        #Creating a list for test dataset predictions
        TestPredictions = []

        #Creating the first testing batch
        FirstTestingBatch = ScaledTrain[-TimeStep:]
        CurrentBatch = FirstTestingBatch.reshape((1, TimeStep, 1))

        #Making predictions on the test dataset
        for h in range(len(TestData)):
            CurrentPrediction = Model.predict(CurrentBatch)[0]
            TestPredictions.append(CurrentPrediction)
            CurrentBatch = np.append(CurrentBatch[:, 1:, :], [[CurrentPrediction]],
axis=1)

        #Unscaling the prediction values
        Forecast = Scaler.inverse_transform(TestPredictions)

        #Calculating error rates and appending them into the errors CSV
        try:
          #print(IdealNumberOfStreams)
          RMSE = root_mean_squared_error(Forecast, TestData["sum"])
          SI = abs(RMSE/abs(TestData["sum"]).mean())
          AccountError = pd.DataFrame({"AccountID" : [i],
                                       "RMSE" : [RMSE],
                                       "SI" : [SI]})
          AccountError.to_csv("LSTM Errors (Cumulative).csv", mode='a',
header=not os.path.exists("LSTM Errors (Cumulative).csv"), index=False)
        #Where a ValueError occurs, the loop is broken
        except ValueError:
          print("ValueError encountered")
          break
```

```
        #Making a list to store future predictions
        FuturePredictions = []

        #Creating the first future batch
        FirstFutureBatch = np.array(TestPredictions[-TimeStep:])
        CurrentBatch = FirstFutureBatch.reshape((1, TimeStep, 1))

        #Forecasting the next 12 months
        for j in range(12):
            CurrentPrediction = Model.predict(CurrentBatch)[0]
            FuturePredictions.append(CurrentPrediction)
            CurrentBatch = np.append(CurrentBatch[:, 1:, :], [[CurrentPrediction]],
axis=1)

        #Unscaling the forecasted values
        FutureForecast = Scaler.inverse_transform(FuturePredictions)

        #Calculating starting amount, ending amount and net change and appending
them to the forecasts CSV
        NetChange = FutureForecast[-1] - FutureForecast[0]
        StartingAmount = FutureForecast[0]
        EndingAmount = FutureForecast[-1]

        ForecastData = pd.DataFrame({"AccountID" : [i],
                             "Net Change over Forecast Period" : [NetChange],
                             "Starting Amount" : [StartingAmount],
                             "Ending Amount" : [EndingAmount]})

        ForecastData.to_csv("LSTM Forecasts (Cumulative).csv", mode='a',
header=not os.path.exists("LSTM Forecasts (Cumulative).csv"), index=False)

        #Informing the user of successful execution
        print("Processed account number",i)
    #Where an error occurs, the user is informed about it
    except Exception as e:
        print(f"Error encountered processing account number {i}.")
        ErrorType, ErrorObject, ErrorTraceback = sys.exc_info()

        ErrorFilename = os.path.split(
            ErrorTraceback.tb_frame.f_code.co_filename
        )[1]

        ErrorMessage = str(e)

        ErrorLineNumber = ErrorTraceback.tb_lineno

        print(f'Exception Type: {ErrorType}')

        print(f'Exception Filename: {ErrorFilename}')
```

```
        print(f'Exception Line Number: {ErrorLineNumber}')

        print(f'Exception Message: {ErrorMessage}')
        break
    #Deleting model data from memory (to conserve RAM)
    finally:
        del Model, Forecast, TrainData, TestData, ForecastingTimeSeries
        gc.collect()

"""Batch for accounts 600-750"""

#Creating models and forecasting for accounts 600-750
for idx, i in enumerate(UniqueAccounts):
  #Ensuring that only accounts 600-750 are processed in this batch
  if idx < 600:
    pass
  else:
    if idx >= 750:
      break
    #Printing the account ID being processed
    print(i)
    try:
        #Selecting only this account's time series
        ForecastingTimeSeries =
GroupedCumulativeSumOfTransactionsByAccountAndMonth[GroupedCumulative
SumOfTransactionsByAccountAndMonth["AccountId"] == i][["TransactionDate",
"sum"]]

        #Calculating the cutoff point between the training and testing sets
        TrainEnd = int(0.7 * len(ForecastingTimeSeries))

        #Splitting the data into training and testing sets
        TrainData = ForecastingTimeSeries[:TrainEnd]
        TestData = ForecastingTimeSeries[TrainEnd:]

        #Setting the training data index to transaction month
        TrainData = TrainData.set_index("TransactionDate")

        #Scaling the training data using min/max scaling
        Scaler = MinMaxScaler(feature_range=(0,1))
        ScaledTrain = Scaler.fit_transform(TrainData)

        #Generating a batched time series that the LSTM can process
        TimeStep = 1
        TimeSeriesData = TimeseriesGenerator(ScaledTrain, ScaledTrain,
length=TimeStep, batch_size=1)

        #Declaring the model layers
        Model = Sequential()
```

```
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh",
input_shape=(TimeStep, 1)))
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh"))
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh"))
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh"))
        Model.add(LSTM(units=16, activation="tanh"))
        Model.add(Dense(units=1, activation="sigmoid"))
        Model.add(Dense(units=1, activation="sigmoid"))
        Model.add(Dense(1))
        Model.compile(optimizer="adam", loss="mean_squared_error")

        #Fitting the model on the training data
        Model.fit(TimeSeriesData, epochs=20, batch_size=1, verbose=0)

        #Declaring a list to store test dataset predictions
        TestPredictions = []

        #Creating the first testing batch
        FirstTestingBatch = ScaledTrain[-TimeStep:]
        CurrentBatch = FirstTestingBatch.reshape((1, TimeStep, 1))

        #Forecasting the test dataset values
        for h in range(len(TestData)):
            CurrentPrediction = Model.predict(CurrentBatch)[0]
            TestPredictions.append(CurrentPrediction)
            CurrentBatch = np.append(CurrentBatch[:, 1:, :], [[CurrentPrediction]],
axis=1)

        #Unscaling the test predictions
        Forecast = Scaler.inverse_transform(TestPredictions)

        #Calculating error rates for the test forecast and appending them into the errors
CSV
        try:
          RMSE = root_mean_squared_error(Forecast, TestData["sum"])
          SI = abs(RMSE/abs(TestData["sum"]).mean())
          AccountError = pd.DataFrame({"AccountID" : [i],
                                       "RMSE" : [RMSE],
                                       "SI" : [SI]})
          AccountError.to_csv("LSTM Errors (Cumulative).csv", mode='a',
header=not os.path.exists("LSTM Errors (Cumulative).csv"), index=False)
        #Where a ValueError occurs, the loop is broken
        except ValueError:
          print("ValueError encountered")
          break

        #Creating a list to store future forecasts
        FuturePredictions = []

        #Creating the first future batch
```

```
        FirstFutureBatch = np.array(TestPredictions[-TimeStep:])
        CurrentBatch = FirstFutureBatch.reshape((1, TimeStep, 1))

        #Forecasting the next 12 months
        for j in range(12):
            CurrentPrediction = Model.predict(CurrentBatch)[0]
            FuturePredictions.append(CurrentPrediction)
            CurrentBatch = np.append(CurrentBatch[:, 1:, :], [[CurrentPrediction]],
axis=1)

        #Unscaling the future forecast values
        FutureForecast = Scaler.inverse_transform(FuturePredictions)

        #Calculating starting amount, ending amount and net change and appending
them into the forecasts CSV
        NetChange = FutureForecast[-1] - FutureForecast[0]
        StartingAmount = FutureForecast[0]
        EndingAmount = FutureForecast[-1]

        ForecastData = pd.DataFrame({"AccountID" : [i],
                             "Net Change over Forecast Period" : [NetChange],
                             "Starting Amount" : [StartingAmount],
                             "Ending Amount" : [EndingAmount]})

        ForecastData.to_csv("LSTM Forecasts (Cumulative).csv", mode='a',
header=not os.path.exists("LSTM Forecasts (Cumulative).csv"), index=False)

        #Informing the user of successful execution
        print("Processed account number",i)
    #Where an error occurs, the user is informed about it
    except Exception as e:
        print(f"Error encountered processing account number {i}.")
        ErrorType, ErrorObject, ErrorTraceback = sys.exc_info()

        ErrorFilename = os.path.split(
            ErrorTraceback.tb_frame.f_code.co_filename
        )[1]

        ErrorMessage = str(e)

        ErrorLineNumber = ErrorTraceback.tb_lineno

        print(f'Exception Type: {ErrorType}')

        print(f'Exception Filename: {ErrorFilename}')

        print(f'Exception Line Number: {ErrorLineNumber}')

        print(f'Exception Message: {ErrorMessage}')
        break
```

```
    #Wiping the model data from memory (to conserve RAM)
    finally:
      del Model, Forecast, TrainData, TestData, ForecastingTimeSeries
      gc.collect()

"""Batch for accounts 750-872 (final 122)"""

#Creating models and forecasting for the final 122 accounts
for idx, i in enumerate(UniqueAccounts):
  #Ensuring that only the final 122 accounts are processed in this batch
  if idx < 750:
    pass
  else:
    #Printing the account ID being processed
    print(i)
    try:
      #Selecting only the selected account's time series
      ForecastingTimeSeries =
GroupedCumulativeSumOfTransactionsByAccountAndMonth[GroupedCumulative
SumOfTransactionsByAccountAndMonth["AccountId"] == i][["TransactionDate",
"sum"]]

      #Calculating the cutoff point between the training and testing sets
      TrainEnd = int(0.7 * len(ForecastingTimeSeries))

      #Splitting the data into training and testing sets
      TrainData = ForecastingTimeSeries[:TrainEnd]
      TestData = ForecastingTimeSeries[TrainEnd:]

      #Setting the transaction month as the index
      TrainData = TrainData.set_index("TransactionDate")

      #Scaling the training data using min/max scaling
      Scaler = MinMaxScaler(feature_range=(0,1))
      ScaledTrain = Scaler.fit_transform(TrainData)

      #Generating a batched time series that the LSTM can process
      TimeStep = 1
      TimeSeriesData = TimeseriesGenerator(ScaledTrain, ScaledTrain,
length=TimeStep, batch_size=1)

      #Declaring the model layers
      Model = Sequential()
      Model.add(LSTM(units=16, return_sequences=True, activation="tanh",
input_shape=(TimeStep, 1)))
      Model.add(LSTM(units=16, return_sequences=True, activation="tanh"))
      Model.add(LSTM(units=16, return_sequences=True, activation="tanh"))
      Model.add(LSTM(units=16, return_sequences=True, activation="tanh"))
      Model.add(LSTM(units=16, activation="tanh"))
      Model.add(Dense(units=1, activation="sigmoid"))
```

```
        Model.add(Dense(units=1, activation="sigmoid"))
        Model.add(Dense(1))
        Model.compile(optimizer="adam", loss="mean_squared_error")

        #Training the model
        Model.fit(TimeSeriesData, epochs=20, batch_size=1, verbose=0)

        #Creating a list to store test dataset predictions
        TestPredictions = []

        #Creating the first testing batch
        FirstTestingBatch = ScaledTrain[-TimeStep:]
        CurrentBatch = FirstTestingBatch.reshape((1, TimeStep, 1))

        #Making predictions on the test dataset
        for h in range(len(TestData)):
            CurrentPrediction = Model.predict(CurrentBatch)[0]
            TestPredictions.append(CurrentPrediction)
            CurrentBatch = np.append(CurrentBatch[:, 1:, :], [[CurrentPrediction]],
axis=1)

        #Unscaling the test predictions
        Forecast = Scaler.inverse_transform(TestPredictions)

        #Calculating the error rates and appending them into the errors CSV
        try:
          RMSE = root_mean_squared_error(Forecast, TestData["sum"])
          SI = abs(RMSE/abs(TestData["sum"]).mean())
          AccountError = pd.DataFrame({"AccountID" : [i],
                                       "RMSE" : [RMSE],
                                       "SI" : [SI]})
          AccountError.to_csv("LSTM Errors (Cumulative).csv", mode='a',
header=not os.path.exists("LSTM Errors (Cumulative).csv"), index=False)
        #Where a ValueError occurs, the loop is broken
        except ValueError:
          print("ValueError encountered")
          break

        #Creating a list to store future predictions
        FuturePredictions = []

        #Creating the first future batch
        FirstFutureBatch = np.array(TestPredictions[-TimeStep:])
        CurrentBatch = FirstFutureBatch.reshape((1, TimeStep, 1))

        #Forecasting the next 12 months
        for j in range(12):
            CurrentPrediction = Model.predict(CurrentBatch)[0]
            FuturePredictions.append(CurrentPrediction)
```

```
            CurrentBatch = np.append(CurrentBatch[:, 1:, :], [[CurrentPrediction]],
axis=1)

        #Unscaling the future forecast values
        FutureForecast = Scaler.inverse_transform(FuturePredictions)

        #Calculating the starting amount, ending amount and net change and
appending them into the forecasts CSV file
        NetChange = FutureForecast[-1] - FutureForecast[0]
        StartingAmount = FutureForecast[0]
        EndingAmount = FutureForecast[-1]

        ForecastData = pd.DataFrame({"AccountID" : [i],
                             "Net Change over Forecast Period" : [NetChange],
                             "Starting Amount" : [StartingAmount],
                             "Ending Amount" : [EndingAmount]})

        ForecastData.to_csv("LSTM Forecasts (Cumulative).csv", mode='a',
header=not os.path.exists("LSTM Forecasts (Cumulative).csv"), index=False)

        #Informing the user of successful execution
        print("Processed account number",i)
    #Where an error occurs, the user is informed about it
    except Exception as e:
        print(f"Error encountered processing account number {i}.")
        ErrorType, ErrorObject, ErrorTraceback = sys.exc_info()

        ErrorFilename = os.path.split(
            ErrorTraceback.tb_frame.f_code.co_filename
        )[1]

        ErrorMessage = str(e)

        ErrorLineNumber = ErrorTraceback.tb_lineno

        print(f'Exception Type: {ErrorType}')

        print(f'Exception Filename: {ErrorFilename}')

        print(f'Exception Line Number: {ErrorLineNumber}')

        print(f'Exception Message: {ErrorMessage}')
        break
    #Wiping the model data from memory (to conserve RAM)
    finally:
        del Model, Forecast, TrainData, TestData, ForecastingTimeSeries
        gc.collect()

"""**Calculating Error Statistics**"""
```

```
#Calculating average errors from the errors CSV file
Errors = pd.read_csv("LSTM Errors (Cumulative).csv")
MeanRMSE = Errors["RMSE"].mean()
print("Mean RMSE:",MeanRMSE)
MedianRMSE = Errors["RMSE"].median()
print("Median RMSE:",MedianRMSE)
MeanSI = Errors["SI"].mean()
print("Mean SI:",MeanSI)
MedianSI = Errors["SI"].median()
print("Median SI:",MedianSI)
```

6.2.2.2 Appendix 2.2.2: LSTM Forecasts (Month-by-Month).py (converted from .ipynb)

```
# -*- coding: utf-8 -*-
"""LSTM Forecasts (Month-by-Month).ipynb

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/1exDXmGO-Kh0Ru1nlJa3qqhP3RUJWIPdt

**Importing Libraries and Data**
"""

#Importing libraries
import pandas as pd
import statsmodels.api as sm
from sklearn.metrics import root_mean_squared_error
from statsmodels.tsa.stattools import acf
import matplotlib.pyplot as plt
import prophet
import numpy as np
import warnings
import gc
import os
import sys
import json
from keras.models import Sequential
from keras.layers import LSTM
from keras.layers import Dense
from keras.layers import Dropout
from keras.callbacks import EarlyStopping
from tensorflow.keras.preprocessing.sequence import TimeseriesGenerator
from sklearn.preprocessing import MinMaxScaler

#Ignoring warnings
warnings.filterwarnings("ignore")

#Reading in data
BoostedTransactions = pd.read_csv("Boosted Transaction Dataset.csv")
#Extracting unique account ID values
AccountIDs = BoostedTransactions["AccountId"].values
UniqueAccounts = set(AccountIDs)

#Converting the transaction dates into dates and converting them to months
BoostedTransactions["TransactionDate"] =
pd.to_datetime(BoostedTransactions["TransactionDate"])
BoostedTransactions["TransactionDate"] =
BoostedTransactions["TransactionDate"].dt.to_period("M").dt.to_timestamp()
```

```
#Grouping the transactions by account and month and calculating the net sum of
transactions
GroupedSumOfTransactionsByAccountAndMonth =
BoostedTransactions.groupby(by=["AccountId",
"TransactionDate"]).agg(["sum"])["Amount"]
print(GroupedSumOfTransactionsByAccountAndMonth)

#Resetting the dataset index
GroupedSumOfTransactionsByAccountAndMonth =
GroupedSumOfTransactionsByAccountAndMonth.reset_index()

"""**Creating Account IDs set and writing unique accounts list to JSON file (only
execute before running first batch)**

Only execute this code block before running the first batch. This is to maintain the
order of accounts and prevent any duplicate account forecasts.
"""

#Generating account IDs set and writing the unique accounts list to a JSON file
AccountIDs = BoostedTransactions["AccountId"].values
UniqueAccounts = list(dict.fromkeys(AccountIDs))
with open("UniqueAccounts.json", "w") as File:
  json.dump(UniqueAccounts, File)

#Reading in the unique accounts list
with open("UniqueAccounts.json", "r") as File:
  UniqueAccounts = json.load(File)

#Converting the transaction dates to months
BoostedTransactions["TransactionDate"] =
pd.to_datetime(BoostedTransactions["TransactionDate"])
BoostedTransactions["TransactionDate"] =
BoostedTransactions["TransactionDate"].dt.to_period("M").dt.to_timestamp()

#Grouping the transactions by account and month and calculating the net sum
GroupedSumOfTransactionsByAccountAndMonth =
BoostedTransactions.groupby(by=["AccountId",
"TransactionDate"]).agg(["sum"])["Amount"]
print(GroupedSumOfTransactionsByAccountAndMonth)

#Creating a list to store the processed time series in
DataFrames = []

#Imputing 0 values into any missing months in each account's time series and adding
it to the DataFrames list
for i in range(0, len(UniqueAccounts)):
  ResetDataFrame =
GroupedSumOfTransactionsByAccountAndMonth.reset_index()
```

```
  TimeSeries = ResetDataFrame[ResetDataFrame["AccountId"] ==
list(UniqueAccounts)[i]]
  TimeSeries = TimeSeries.set_index("TransactionDate")

  TimeSeries = TimeSeries.resample("MS").sum()
  for j in range(0, len(TimeSeries)):
    if TimeSeries.iloc[j]["AccountId"] == 0:
      TimeSeries.at[TimeSeries.index[j], "AccountId"] = list(UniqueAccounts)[i]

  DataFrames.append(TimeSeries)

#Grouping the time series together into one dataset containing all time series
NewGroupedDataset = pd.concat(DataFrames)
NewGroupedDataset = NewGroupedDataset.reset_index()

print(NewGroupedDataset)

"""**Key LSTM Loops (split up for memory reasons)**

Batch for accounts 0-150
"""

#Creating models and forecasting for the first 150 accounts
for idx, i in enumerate(UniqueAccounts):
    #Ensuring that only the first 150 accounts are processed in this batch
    if idx >= 150:
        break
    print(i)
    try:
        #Selecting only the data for the chosen account and ensuring there are no
missing months
        ForecastingTimeSeries =
NewGroupedDataset[NewGroupedDataset["AccountId"] == i][["TransactionDate",
"sum"]]

        ForecastingTimeSeries = ForecastingTimeSeries.set_index("TransactionDate")

        ForecastingTimeSeries = ForecastingTimeSeries.resample("MS").sum()

        ForecastingTimeSeries = ForecastingTimeSeries.reset_index()

        #Calculating the cutoff point between training and testing sets
        TrainEnd = int(0.7 * len(ForecastingTimeSeries))

        #Splitting the data into training and testing sets
        TrainData = ForecastingTimeSeries[:TrainEnd]
        TestData = ForecastingTimeSeries[TrainEnd:]

        #Setting transaction month as the index
        TrainData = TrainData.set_index("TransactionDate")
```

```
        #Scaling the training data using min/max scaling
        Scaler = MinMaxScaler(feature_range=(0,1))
        ScaledTrain = Scaler.fit_transform(TrainData)

        #Generating a batched time series that the LSTM can process
        TimeStep = 1
        TimeSeriesData = TimeseriesGenerator(ScaledTrain, ScaledTrain,
length=TimeStep, batch_size=1)

        #Declaring the model layers
        Model = Sequential()
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh",
input_shape=(TimeStep, 1)))
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh"))
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh"))
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh"))
        Model.add(LSTM(units=16, activation="tanh"))
        Model.add(Dense(units=1, activation="sigmoid"))
        Model.add(Dense(units=1, activation="sigmoid"))
        Model.add(Dense(1))
        Model.compile(optimizer="adam", loss="mean_squared_error")

        #Training the model
        Model.fit(TimeSeriesData, epochs=20, batch_size=1, verbose=0)

        #Creating a list for the test data predictions
        TestPredictions = []

        #Creating the first testing batch
        FirstTestingBatch = ScaledTrain[-TimeStep:]
        CurrentBatch = FirstTestingBatch.reshape((1, TimeStep, 1))

        #Forecasting the test dataset
        for h in range(len(TestData)):
            CurrentPrediction = Model.predict(CurrentBatch)[0]
            TestPredictions.append(CurrentPrediction)
            CurrentBatch = np.append(CurrentBatch[:, 1:, :], [[CurrentPrediction]],
axis=1)

        #Unscaling the test prediction values
        Forecast = Scaler.inverse_transform(TestPredictions)

        #Calculating the error rates and appending them into the errors CSV
        try:
          RMSE = root_mean_squared_error(Forecast, TestData["sum"])
          SI = abs(RMSE/abs(TestData["sum"]).mean())
          AccountError = pd.DataFrame({"AccountID" : [i],
                                       "RMSE" : [RMSE],
                                       "SI" : [SI]})
```

```
        AccountError.to_csv("LSTM Errors (Month-by-Month).csv", mode='a',
header=not os.path.exists("LSTM Errors (Month-by-Month).csv"), index=False)
      #Where a ValueError occurs, the loop is broken
      except ValueError:
        print("ValueError encountered")
        break

      #Creating a list to store the future predictions in
      FuturePredictions = []

      #Creating the first future batch
      FirstFutureBatch = np.array(TestPredictions[-TimeStep:])
      CurrentBatch = FirstFutureBatch.reshape((1, TimeStep, 1))

      #Forecasting the next 12 months
      for j in range(12):
        CurrentPrediction = Model.predict(CurrentBatch)[0]
        FuturePredictions.append(CurrentPrediction)
        CurrentBatch = np.append(CurrentBatch[:, 1:, :], [[CurrentPrediction]],
axis=1)

      #Unscaling the future forecast values
      FutureForecast = Scaler.inverse_transform(FuturePredictions)

      #Calculating highest amount, lowest amount and difference and appending
them to the forecasts CSV
      HighestAmount = FutureForecast.max()
      LowestAmount = FutureForecast.min()
      Difference = HighestAmount - LowestAmount

      ForecastData = pd.DataFrame({"AccountID" : [i],
                    "Difference over Forecast Period" : [Difference],
                    "Highest Amount" : [HighestAmount],
                    "Lowest Amount" : [LowestAmount]})

      ForecastData.to_csv("LSTM Forecasts (Month-by-Month).csv", mode='a',
header=not os.path.exists("LSTM Forecasts (Month-by-Month).csv"), index=False)

      #Informing the user of successful execution
      print("Processed account number",i)
    #Where an error occurs, the user is informed about it
    except Exception as e:
      print(f"Error encountered processing account number {i}.")
      ErrorType, ErrorObject, ErrorTraceback = sys.exc_info()

      ErrorFilename = os.path.split(
        ErrorTraceback.tb_frame.f_code.co_filename
      )[1]

      ErrorMessage = str(e)
```

```
        ErrorLineNumber = ErrorTraceback.tb_lineno

        print(f'Exception Type: {ErrorType}')

        print(f'Exception Filename: {ErrorFilename}')

        print(f'Exception Line Number: {ErrorLineNumber}')

        print(f'Exception Message: {ErrorMessage}')
        break
    #Wiping the model info from memory (to conserve RAM)
    finally:
        del Model, Forecast, TrainData, TestData, ForecastingTimeSeries
        gc.collect()

"""Batch for accounts 150-300"""

#Creating models and forecasting for accounts 150-300
for idx, i in enumerate(UniqueAccounts):
    #Ensuring that only accounts 150-300 are processed
    if idx < 150:
        pass
    else:
      if idx >= 300:
        break
      print(i)
      try:
          #Selecting only the selected account's time series and ensuring it contains no
missing values
          ForecastingTimeSeries =
NewGroupedDataset[NewGroupedDataset["AccountId"] == i][["TransactionDate",
"sum"]]

          ForecastingTimeSeries =
ForecastingTimeSeries.set_index("TransactionDate")

          ForecastingTimeSeries = ForecastingTimeSeries.resample("MS").sum()

          ForecastingTimeSeries = ForecastingTimeSeries.reset_index()

          #Calculating the cutoff point between training and testing sets
          TrainEnd = int(0.7 * len(ForecastingTimeSeries))

          #Splitting the data into training and testing sets
          TrainData = ForecastingTimeSeries[:TrainEnd]
          TestData = ForecastingTimeSeries[TrainEnd:]

          #Setting transaction month as the index
          TrainData = TrainData.set_index("TransactionDate")
```

```
        #Scaling the training data using min/max scaling
        Scaler = MinMaxScaler(feature_range=(0,1))
        ScaledTrain = Scaler.fit_transform(TrainData)

        #Generating a batched time series that the LSTM can process
        TimeStep = 1
        TimeSeriesData = TimeseriesGenerator(ScaledTrain, ScaledTrain,
length=TimeStep, batch_size=1)

        #Declaring the model layers
        Model = Sequential()
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh",
input_shape=(TimeStep, 1)))
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh"))
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh"))
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh"))
        Model.add(LSTM(units=16, activation="tanh"))
        Model.add(Dense(units=1, activation="sigmoid"))
        Model.add(Dense(units=1, activation="sigmoid"))
        Model.add(Dense(1))
        Model.compile(optimizer="adam", loss="mean_squared_error")

        #Training the model
        Model.fit(TimeSeriesData, epochs=20, batch_size=1, verbose=0)

        #Creating a list to store test predictions
        TestPredictions = []

        #Creating the first test dataset batch
        FirstTestingBatch = ScaledTrain[-TimeStep:]
        CurrentBatch = FirstTestingBatch.reshape((1, TimeStep, 1))

        #Forecasting on the test dataset
        for h in range(len(TestData)):
            CurrentPrediction = Model.predict(CurrentBatch)[0]
            TestPredictions.append(CurrentPrediction)
            CurrentBatch = np.append(CurrentBatch[:, 1:, :], [[CurrentPrediction]],
axis=1)

        #Unscaling the test predictions
        Forecast = Scaler.inverse_transform(TestPredictions)

        #Calculating the error rates and appending them into the errors CSV
        try:
          RMSE = root_mean_squared_error(Forecast, TestData["sum"])
          SI = abs(RMSE/abs(TestData["sum"]).mean())
          AccountError = pd.DataFrame({"AccountID" : [i],
                                       "RMSE" : [RMSE],
                                       "SI" : [SI]})
```

```
          AccountError.to_csv("LSTM Errors (Month-by-Month).csv", mode='a',
header=not os.path.exists("LSTM Errors (Month-by-Month).csv"), index=False)
        #Where a ValueError occurs, the loop is broken
        except ValueError:
          print("ValueError encountered")
          break

        #Creating a list to store future predictions
        FuturePredictions = []

        #Creating the first future batch
        FirstFutureBatch = np.array(TestPredictions[-TimeStep:])
        CurrentBatch = FirstFutureBatch.reshape((1, TimeStep, 1))

        #Forecasting the next 12 months
        for j in range(12):
           CurrentPrediction = Model.predict(CurrentBatch)[0]
           FuturePredictions.append(CurrentPrediction)
           CurrentBatch = np.append(CurrentBatch[:, 1:, :], [[CurrentPrediction]],
axis=1)

        #Unscaling the future forecast values
        FutureForecast = Scaler.inverse_transform(FuturePredictions)

        #Calculating highest amount, lowest amount and difference and appending
them into the forecasts CSV
        HighestAmount = FutureForecast.max()
        LowestAmount = FutureForecast.min()
        Difference = HighestAmount - LowestAmount

        ForecastData = pd.DataFrame({"AccountID" : [i],
                             "Difference over Forecast Period" : [Difference],
                             "Highest Amount" : [HighestAmount],
                             "Lowest Amount" : [LowestAmount]})

        ForecastData.to_csv("LSTM Forecasts (Month-by-Month).csv", mode='a',
header=not os.path.exists("LSTM Forecasts (Month-by-Month).csv"), index=False)

        print("Processed account number",i)
      #Where an error occurs, the user is informed about it
      except Exception as e:
        print(f"Error encountered processing account number {i}.")
        ErrorType, ErrorObject, ErrorTraceback = sys.exc_info()

        ErrorFilename = os.path.split(
           ErrorTraceback.tb_frame.f_code.co_filename
        )[1]

        ErrorMessage = str(e)
```

```
        ErrorLineNumber = ErrorTraceback.tb_lineno

        print(f'Exception Type: {ErrorType}')

        print(f'Exception Filename: {ErrorFilename}')

        print(f'Exception Line Number: {ErrorLineNumber}')

        print(f'Exception Message: {ErrorMessage}')
        break
    #Deleting the model data from memory to conserve RAM
    finally:
        del Model, Forecast, TrainData, TestData, ForecastingTimeSeries
        gc.collect()

"""Batch for accounts 300-450"""

#Creating models and forecasting for accounts 300-450
for idx, i in enumerate(UniqueAccounts):
  #Ensuring that only accounts 300-450 are processed
  if idx < 300:
    pass
  else:
    if idx >= 450:
      break
    print(i)
    try:
      #Selecting only the selected account's time series and ensuring it has no gaps
      ForecastingTimeSeries =
NewGroupedDataset[NewGroupedDataset["AccountId"] == i][["TransactionDate",
"sum"]]

      ForecastingTimeSeries =
ForecastingTimeSeries.set_index("TransactionDate")

      ForecastingTimeSeries = ForecastingTimeSeries.resample("MS").sum()

      ForecastingTimeSeries = ForecastingTimeSeries.reset_index()

      #Calculating the cutoff point between training and testing sets
      TrainEnd = int(0.7 * len(ForecastingTimeSeries))

      #Splitting the dataset into training and testing sets
      TrainData = ForecastingTimeSeries[:TrainEnd]
      TestData = ForecastingTimeSeries[TrainEnd:]

      #Setting transaction month as the index
      TrainData = TrainData.set_index("TransactionDate")

      #Scaling the training data using min/max scaling
```

```
        Scaler = MinMaxScaler(feature_range=(0,1))
        ScaledTrain = Scaler.fit_transform(TrainData)

        #Generating a batched time series that the LSTM can process
        TimeStep = 1
        TimeSeriesData = TimeseriesGenerator(ScaledTrain, ScaledTrain,
length=TimeStep, batch_size=1)

        #Declaring the model layers
        Model = Sequential()
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh",
input_shape=(TimeStep, 1)))
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh"))
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh"))
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh"))
        Model.add(LSTM(units=16, activation="tanh"))
        Model.add(Dense(units=1, activation="sigmoid"))
        Model.add(Dense(units=1, activation="sigmoid"))
        Model.add(Dense(1))
        Model.compile(optimizer="adam", loss="mean_squared_error")

        #Training the model
        Model.fit(TimeSeriesData, epochs=20, batch_size=1, verbose=0)

        #Generating a list to store test predictions
        TestPredictions = []

        #Generating the first testing batch
        FirstTestingBatch = ScaledTrain[-TimeStep:]
        CurrentBatch = FirstTestingBatch.reshape((1, TimeStep, 1))

        #Forecasting on the test dataset
        for h in range(len(TestData)):
            CurrentPrediction = Model.predict(CurrentBatch)[0]
            TestPredictions.append(CurrentPrediction)
            CurrentBatch = np.append(CurrentBatch[:, 1:, :], [[CurrentPrediction]],
axis=1)

        #Unscaling the test prediction values
        Forecast = Scaler.inverse_transform(TestPredictions)

        #Calculating the error rates and appending them into the errors CSV
        try:
          #print(IdealNumberOfStreams)
          RMSE = root_mean_squared_error(Forecast, TestData["sum"])
          SI = abs(RMSE/abs(TestData["sum"]).mean())
          AccountError = pd.DataFrame({"AccountID" : [i],
                                       "RMSE" : [RMSE],
                                       "SI" : [SI]})
```

```
          AccountError.to_csv("LSTM Errors (Month-by-Month).csv", mode='a',
header=not os.path.exists("LSTM Errors (Month-by-Month).csv"), index=False)
        #Where a ValueError occurs, the loop is broken
        except ValueError:
          print("ValueError encountered")
          break

        #Creating a list to store future predictions
        FuturePredictions = []

        #Generating the first future batch
        FirstFutureBatch = np.array(TestPredictions[-TimeStep:])
        CurrentBatch = FirstFutureBatch.reshape((1, TimeStep, 1))

        #Forecasting the next 12 months
        for j in range(12):
           CurrentPrediction = Model.predict(CurrentBatch)[0]
           FuturePredictions.append(CurrentPrediction)
           CurrentBatch = np.append(CurrentBatch[:, 1:, :], [[CurrentPrediction]],
axis=1)

        #Unscaling the future forecast values
        FutureForecast = Scaler.inverse_transform(FuturePredictions)

        #Calculating highest amount, lowest amount and difference and appending
them into the forecasts CSV
        HighestAmount = FutureForecast.max()
        LowestAmount = FutureForecast.min()
        Difference = HighestAmount - LowestAmount

        ForecastData = pd.DataFrame({"AccountID" : [i],
                             "Difference over Forecast Period" : [Difference],
                             "Highest Amount" : [HighestAmount],
                             "Lowest Amount" : [LowestAmount]})

        ForecastData.to_csv("LSTM Forecasts (Month-by-Month).csv", mode='a',
header=not os.path.exists("LSTM Forecasts (Month-by-Month).csv"), index=False)

        print("Processed account number",i)
      #Where an error occurs, the user is informed about it
      except Exception as e:
        print(f"Error encountered processing account number {i}.")
        ErrorType, ErrorObject, ErrorTraceback = sys.exc_info()

        ErrorFilename = os.path.split(
           ErrorTraceback.tb_frame.f_code.co_filename
        )[1]

        ErrorMessage = str(e)
```

```
        ErrorLineNumber = ErrorTraceback.tb_lineno

        print(f'Exception Type: {ErrorType}')

        print(f'Exception Filename: {ErrorFilename}')

        print(f'Exception Line Number: {ErrorLineNumber}')

        print(f'Exception Message: {ErrorMessage}')
        break
    #Wiping the model data from memory to conserve RAM
    finally:
        del Model, Forecast, TrainData, TestData, ForecastingTimeSeries
        gc.collect()

"""Batch for accounts 450-600"""

#Creating models and forecasting for accounts 450-600
for idx, i in enumerate(UniqueAccounts):
  #Ensuring that only accounts 450-600 are processed
  if idx < 450:
    pass
  else:
    if idx >= 600:
      break
    print(i)
    try:
      #Selecting only the current account's time series and ensuring that no gaps
exist
      ForecastingTimeSeries =
NewGroupedDataset[NewGroupedDataset["AccountId"] == i][["TransactionDate",
"sum"]]

      ForecastingTimeSeries =
ForecastingTimeSeries.set_index("TransactionDate")

      ForecastingTimeSeries = ForecastingTimeSeries.resample("MS").sum()

      ForecastingTimeSeries = ForecastingTimeSeries.reset_index()

      #Calculating the cutoff point between training and testing sets
      TrainEnd = int(0.7 * len(ForecastingTimeSeries))

      #Splitting the dataset into training and testing sets
      TrainData = ForecastingTimeSeries[:TrainEnd]
      TestData = ForecastingTimeSeries[TrainEnd:]

      #Setting the transaction month as the index
      TrainData = TrainData.set_index("TransactionDate")
```

```
        #Scaling the training dataset using min/max scaling
        Scaler = MinMaxScaler(feature_range=(0,1))
        ScaledTrain = Scaler.fit_transform(TrainData)

        #Generating a batched time series that the LSTM can process
        TimeStep = 1
        TimeSeriesData = TimeseriesGenerator(ScaledTrain, ScaledTrain,
length=TimeStep, batch_size=1)

        #Declaring model layers
        Model = Sequential()
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh",
input_shape=(TimeStep, 1)))
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh"))
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh"))
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh"))
        Model.add(LSTM(units=16, activation="tanh"))
        Model.add(Dense(units=1, activation="sigmoid"))
        Model.add(Dense(units=1, activation="sigmoid"))
        Model.add(Dense(1))
        Model.compile(optimizer="adam", loss="mean_squared_error")

        #Training the model
        Model.fit(TimeSeriesData, epochs=20, batch_size=1, verbose=0)

        #Creating a list to store the test dataset predictions
        TestPredictions = []

        #Creating the first testing batch
        FirstTestingBatch = ScaledTrain[-TimeStep:]
        CurrentBatch = FirstTestingBatch.reshape((1, TimeStep, 1))

        #Forecasting on the test dataset
        for h in range(len(TestData)):
            CurrentPrediction = Model.predict(CurrentBatch)[0]
            TestPredictions.append(CurrentPrediction)
            CurrentBatch = np.append(CurrentBatch[:, 1:, :], [[CurrentPrediction]],
axis=1)

        #Unscaling the test predictions
        Forecast = Scaler.inverse_transform(TestPredictions)

        #Calculating the error rates and appending them into the errors CSV
        try:
          RMSE = root_mean_squared_error(Forecast, TestData["sum"])
          SI = abs(RMSE/abs(TestData["sum"]).mean())
          AccountError = pd.DataFrame({"AccountID" : [i],
                                       "RMSE" : [RMSE],
                                       "SI" : [SI]})
```

```
          AccountError.to_csv("LSTM Errors (Month-by-Month).csv", mode='a',
header=not os.path.exists("LSTM Errors (Month-by-Month).csv"), index=False)
        #Where a ValueError occurs, the loop is broken
        except ValueError:
          print("ValueError encountered")
          break

        #Creating a list to store future predictions
        FuturePredictions = []

        #Creating the first future batch
        FirstFutureBatch = np.array(TestPredictions[-TimeStep:])
        CurrentBatch = FirstFutureBatch.reshape((1, TimeStep, 1))

        #Forecasting the next 12 months
        for j in range(12):
           CurrentPrediction = Model.predict(CurrentBatch)[0]
           FuturePredictions.append(CurrentPrediction)
           CurrentBatch = np.append(CurrentBatch[:, 1:, :], [[CurrentPrediction]],
axis=1)

        #Unscaling the future forecast
        FutureForecast = Scaler.inverse_transform(FuturePredictions)

        #Calculating highest amount, lowest amount and difference and appending
them into the forecasts CSV
        HighestAmount = FutureForecast.max()
        LowestAmount = FutureForecast.min()
        Difference = HighestAmount - LowestAmount

        ForecastData = pd.DataFrame({"AccountID" : [i],
                             "Difference over Forecast Period" : [Difference],
                             "Highest Amount" : [HighestAmount],
                             "Lowest Amount" : [LowestAmount]})

        ForecastData.to_csv("LSTM Forecasts (Month-by-Month).csv", mode='a',
header=not os.path.exists("LSTM Forecasts (Month-by-Month).csv"), index=False)

        print("Processed account number",i)
      #Where an error occurs, the user is informed about it
      except Exception as e:
        print(f"Error encountered processing account number {i}.")
        ErrorType, ErrorObject, ErrorTraceback = sys.exc_info()

        ErrorFilename = os.path.split(
          ErrorTraceback.tb_frame.f_code.co_filename
        )[1]

        ErrorMessage = str(e)
```

```
        ErrorLineNumber = ErrorTraceback.tb_lineno

        print(f'Exception Type: {ErrorType}')

        print(f'Exception Filename: {ErrorFilename}')

        print(f'Exception Line Number: {ErrorLineNumber}')

        print(f'Exception Message: {ErrorMessage}')
        break
    #Deleting model data from memory to conserve RAM
    finally:
        del Model, Forecast, TrainData, TestData, ForecastingTimeSeries
        gc.collect()

"""Batch for accounts 600-750"""

#Creating models and forecasting for accounts 600-750
for idx, i in enumerate(UniqueAccounts):
  #Ensuring that only accounts 600-750 are processed
  if idx < 600:
    pass
  else:
    if idx >= 750:
      break
    print(i)
    try:
      #Select only the chosen account's time series and ensure it contains no gaps
      ForecastingTimeSeries =
NewGroupedDataset[NewGroupedDataset["AccountId"] == i][["TransactionDate",
"sum"]]

      ForecastingTimeSeries =
ForecastingTimeSeries.set_index("TransactionDate")

      ForecastingTimeSeries = ForecastingTimeSeries.resample("MS").sum()

      ForecastingTimeSeries = ForecastingTimeSeries.reset_index()

      #Calculating the cutoff point between training and testing sets
      TrainEnd = int(0.7 * len(ForecastingTimeSeries))

      #Splitting the dataset into training and testing sets
      TrainData = ForecastingTimeSeries[:TrainEnd]
      TestData = ForecastingTimeSeries[TrainEnd:]

      #Setting the transaction month as the index
      TrainData = TrainData.set_index("TransactionDate")

      #Scaling the data using min/max scaling
```

```
        Scaler = MinMaxScaler(feature_range=(0,1))
        ScaledTrain = Scaler.fit_transform(TrainData)

        #Generating a batched time series that the LSTM can process
        TimeStep = 1
        TimeSeriesData = TimeseriesGenerator(ScaledTrain, ScaledTrain,
length=TimeStep, batch_size=1)

        #Declaring model layers
        Model = Sequential()
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh",
input_shape=(TimeStep, 1)))
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh"))
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh"))
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh"))
        Model.add(LSTM(units=16, activation="tanh"))
        Model.add(Dense(units=1, activation="sigmoid"))
        Model.add(Dense(units=1, activation="sigmoid"))
        Model.add(Dense(1))
        Model.compile(optimizer="adam", loss="mean_squared_error")

        #Training the model
        Model.fit(TimeSeriesData, epochs=20, batch_size=1, verbose=0)

        #Creating a list to store test predictions
        TestPredictions = []

        #Generating the first testing batch
        FirstTestingBatch = ScaledTrain[-TimeStep:]
        CurrentBatch = FirstTestingBatch.reshape((1, TimeStep, 1))

        #Forecasting on the test data
        for h in range(len(TestData)):
            CurrentPrediction = Model.predict(CurrentBatch)[0]
            TestPredictions.append(CurrentPrediction)
            CurrentBatch = np.append(CurrentBatch[:, 1:, :], [[CurrentPrediction]],
axis=1)

        #Unscaling the test forecasts
        Forecast = Scaler.inverse_transform(TestPredictions)

        #Calculating the error rates and appending them into the errors CSV
        try:
          RMSE = root_mean_squared_error(Forecast, TestData["sum"])
          SI = abs(RMSE/abs(TestData["sum"]).mean())
          AccountError = pd.DataFrame({"AccountID" : [i],
                                       "RMSE" : [RMSE],
                                       "SI" : [SI]})
          AccountError.to_csv("LSTM Errors (Month-by-Month).csv", mode='a',
header=not os.path.exists("LSTM Errors (Month-by-Month).csv"), index=False)
```

```
        #Where a ValueError occurs, the loop is broken
        except ValueError:
          print("ValueError encountered")
          break

        #Creating a list to store future predictions
        FuturePredictions = []

        #Creating the first future batch
        FirstFutureBatch = np.array(TestPredictions[-TimeStep:])
        CurrentBatch = FirstFutureBatch.reshape((1, TimeStep, 1))

        #Forecasting the next 12 months
        for j in range(12):
            CurrentPrediction = Model.predict(CurrentBatch)[0]
            FuturePredictions.append(CurrentPrediction)
            CurrentBatch = np.append(CurrentBatch[:, 1:, :], [[CurrentPrediction]],
axis=1)

        #Unscaling the future forecast
        FutureForecast = Scaler.inverse_transform(FuturePredictions)

        #Calculating highest amount, lowest amount and difference and appending
them into the forecasts CSV
        HighestAmount = FutureForecast.max()
        LowestAmount = FutureForecast.min()
        Difference = HighestAmount - LowestAmount

        ForecastData = pd.DataFrame({"AccountID" : [i],
                             "Difference over Forecast Period" : [Difference],
                             "Highest Amount" : [HighestAmount],
                             "Lowest Amount" : [LowestAmount]})

        ForecastData.to_csv("LSTM Forecasts (Month-by-Month).csv", mode='a',
header=not os.path.exists("LSTM Forecasts (Month-by-Month).csv"), index=False)

        print("Processed account number",i)
    #Where an error occurs, the user is informed about it
    except Exception as e:
        print(f"Error encountered processing account number {i}.")
        ErrorType, ErrorObject, ErrorTraceback = sys.exc_info()

        ErrorFilename = os.path.split(
            ErrorTraceback.tb_frame.f_code.co_filename
        )[1]

        ErrorMessage = str(e)

        ErrorLineNumber = ErrorTraceback.tb_lineno
```

```
        print(f'Exception Type: {ErrorType}')

        print(f'Exception Filename: {ErrorFilename}')

        print(f'Exception Line Number: {ErrorLineNumber}')

        print(f'Exception Message: {ErrorMessage}')
        break
    #Deleting model data from memory to conserve RAM
    finally:
        del Model, Forecast, TrainData, TestData, ForecastingTimeSeries
        gc.collect()

"""Batch for last 122 accounts"""

#Creating models and forecasting for the last 122 accounts
for idx, i in enumerate(UniqueAccounts):
   #Ensuring that only the last 122 accounts are processed
   if idx < 750:
      pass
   else:
     print(i)
     try:
        #Selecting only the selected account's time series and ensuring it has no gaps
        ForecastingTimeSeries =
NewGroupedDataset[NewGroupedDataset["AccountId"] == i][["TransactionDate",
"sum"]]

        ForecastingTimeSeries =
ForecastingTimeSeries.set_index("TransactionDate")

        ForecastingTimeSeries = ForecastingTimeSeries.resample("MS").sum()

        ForecastingTimeSeries = ForecastingTimeSeries.reset_index()

        #Calculating the cutoff point between training and testing sets
        TrainEnd = int(0.7 * len(ForecastingTimeSeries))

        #Splitting the data into training and testing sets
        TrainData = ForecastingTimeSeries[:TrainEnd]
        TestData = ForecastingTimeSeries[TrainEnd:]

        #Setting transaction month as the index
        TrainData = TrainData.set_index("TransactionDate")

        #Scaling using min/max scaling
        Scaler = MinMaxScaler(feature_range=(0,1))
        ScaledTrain = Scaler.fit_transform(TrainData)

        #Generating a batched time series that LSTM can process
```

```
        TimeStep = 1
        TimeSeriesData = TimeseriesGenerator(ScaledTrain, ScaledTrain,
length=TimeStep, batch_size=1)

        #Declaring model layers
        Model = Sequential()
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh",
input_shape=(TimeStep, 1)))
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh"))
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh"))
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh"))
        Model.add(LSTM(units=16, activation="tanh"))
        Model.add(Dense(units=1, activation="sigmoid"))
        Model.add(Dense(units=1, activation="sigmoid"))
        Model.add(Dense(1))
        Model.compile(optimizer="adam", loss="mean_squared_error")

        #Training the model
        Model.fit(TimeSeriesData, epochs=20, batch_size=1, verbose=0)

        #Creating a list to store test predictions
        TestPredictions = []

        #Creating the first test batch
        FirstTestingBatch = ScaledTrain[-TimeStep:]
        CurrentBatch = FirstTestingBatch.reshape((1, TimeStep, 1))

        #Forecasting for the test dataset
        for h in range(len(TestData)):
           CurrentPrediction = Model.predict(CurrentBatch)[0]
           TestPredictions.append(CurrentPrediction)
           CurrentBatch = np.append(CurrentBatch[:, 1:, :], [[CurrentPrediction]],
axis=1)

        #Unscaling the test forecast
        Forecast = Scaler.inverse_transform(TestPredictions)

        #Calculating error rates and appending to the errors CSV
        try:
          RMSE = root_mean_squared_error(Forecast, TestData["sum"])
          SI = abs(RMSE/abs(TestData["sum"]).mean())
          AccountError = pd.DataFrame({"AccountID" : [i],
                                       "RMSE" : [RMSE],
                                       "SI" : [SI]})
          AccountError.to_csv("LSTM Errors (Month-by-Month).csv", mode='a',
header=not os.path.exists("LSTM Errors (Month-by-Month).csv"), index=False)
        #Where a ValueError occurs, the loop is broken
        except ValueError:
          print("ValueError encountered")
          break
```

```
        #Creating a list to store future predictions
        FuturePredictions = []

        #Creating first future batch
        FirstFutureBatch = np.array(TestPredictions[-TimeStep:])
        CurrentBatch = FirstFutureBatch.reshape((1, TimeStep, 1))

        #Forecasting the next 12 months
        for j in range(12):
            CurrentPrediction = Model.predict(CurrentBatch)[0]
            FuturePredictions.append(CurrentPrediction)
            CurrentBatch = np.append(CurrentBatch[:, 1:, :], [[CurrentPrediction]],
axis=1)

        #Unscaling the future forecast
        FutureForecast = Scaler.inverse_transform(FuturePredictions)

        #Calculating the highest amount, lowest amount and difference and appending
them into the forecasts CSV
        HighestAmount = FutureForecast.max()
        LowestAmount = FutureForecast.min()
        Difference = HighestAmount - LowestAmount

        ForecastData = pd.DataFrame({"AccountID" : [i],
                                     "Difference over Forecast Period" : [Difference],
                                     "Highest Amount" : [HighestAmount],
                                     "Lowest Amount" : [LowestAmount]})

        ForecastData.to_csv("LSTM Forecasts (Month-by-Month).csv", mode='a',
header=not os.path.exists("LSTM Forecasts (Month-by-Month).csv"), index=False)

        print("Processed account number",i)
    #Where an error occurs, the user is informed about it
    except Exception as e:
        print(f"Error encountered processing account number {i}.")
        ErrorType, ErrorObject, ErrorTraceback = sys.exc_info()

        ErrorFilename = os.path.split(
            ErrorTraceback.tb_frame.f_code.co_filename
        )[1]

        ErrorMessage = str(e)

        ErrorLineNumber = ErrorTraceback.tb_lineno

        print(f'Exception Type: {ErrorType}')

        print(f'Exception Filename: {ErrorFilename}')
```

```
            print(f'Exception Line Number: {ErrorLineNumber}')

            print(f'Exception Message: {ErrorMessage}')
            break
        #Deleting model data from memory to conserve RAM
        finally:
            del Model, Forecast, TrainData, TestData, ForecastingTimeSeries
            gc.collect()

"""**Calculating Error Statistics**"""

#Calculating averaged error statistics
Errors = pd.read_csv("LSTM Errors (Month-by-Month).csv")
MeanRMSE = Errors["RMSE"].mean()
print("Mean RMSE:",MeanRMSE)
MedianRMSE = Errors["RMSE"].median()
print("Median RMSE:",MedianRMSE)
MeanSI = Errors["SI"].mean()
print("Mean SI:",MeanSI)
MedianSI = Errors["SI"].median()
print("Median SI:",MedianSI)
```

6.2.2.3 Appendix 2.2.3: LSTM Forecasts on Longer Subset (Cumulative).py (converted from .ipynb)

```
# -*- coding: utf-8 -*-
"""LSTM Forecasts on Longer Subset (Cumulative).ipynb

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/1cufjDjQkqqYsCn41wds8vI0aP-SLHaWo

**Importing Libraries and Data**
"""

#Importing libraries
import pandas as pd
import statsmodels.api as sm
from sklearn.metrics import root_mean_squared_error
from statsmodels.tsa.stattools import acf
import matplotlib.pyplot as plt
import prophet
import numpy as np
import warnings
import gc
import os
import sys
import json
from keras.models import Sequential
from keras.layers import LSTM
from keras.layers import Dense
from keras.layers import Dropout
from keras.callbacks import EarlyStopping
from tensorflow.keras.preprocessing.sequence import TimeseriesGenerator
from sklearn.preprocessing import MinMaxScaler

#Ignoring warnings
warnings.filterwarnings("ignore")

#Reading in data and extracting account ID values
BoostedTransactions = pd.read_csv("Longer Subset of Transaction Data (14 Months).csv")
AccountIDs = BoostedTransactions["AccountId"].values
UniqueAccounts = set(AccountIDs)

#Converting transaction dates into months
BoostedTransactions["TransactionDate"] = pd.to_datetime(BoostedTransactions["TransactionDate"])
BoostedTransactions["TransactionDate"] = BoostedTransactions["TransactionDate"].dt.to_period("M").dt.to_timestamp()
```

```
#Grouping transactions by account and month and calculating the net sum of
transactions
GroupedSumOfTransactionsByAccountAndMonth =
BoostedTransactions.groupby(by=["AccountId",
"TransactionDate"]).agg(["sum"])["Amount"]
print(GroupedSumOfTransactionsByAccountAndMonth)

#Creating a list to store each account's manipulated time series in
DataFrames = []

#Imputing 0 values into any gaps in each account's time series
for i in range(0, len(UniqueAccounts)):
  ResetDataFrame =
GroupedSumOfTransactionsByAccountAndMonth.reset_index()
  TimeSeries = ResetDataFrame[ResetDataFrame["AccountId"] ==
list(UniqueAccounts)[i]]
  TimeSeries = TimeSeries.set_index("TransactionDate")

  TimeSeries = TimeSeries.resample("MS").sum()
  for j in range(0, len(TimeSeries)):
    if TimeSeries.iloc[j]["AccountId"] == 0:
      TimeSeries.at[TimeSeries.index[j], "AccountId"] = list(UniqueAccounts)[i]

  DataFrames.append(TimeSeries)

#Creating a combined DataFrame containing all accounts' time series
NewGroupedDataset = pd.concat(DataFrames)
NewGroupedDataset = NewGroupedDataset.reset_index()
NewGroupedDataset = NewGroupedDataset.set_index(["AccountId",
"TransactionDate"])
print(NewGroupedDataset)

#Calculating the cumulative transaction sum across the time series of each account
GroupedCumulativeSumOfTransactionsByAccountAndMonth =
NewGroupedDataset.groupby(level=0).cumsum().reset_index()
print(GroupedCumulativeSumOfTransactionsByAccountAndMonth)

"""**Generating unique Account IDs list and writing it into JSON file (only run
before first batch)**

Only run this code block before running the first batch. This is to ensure the order of
accounts is maintained and no accounts are duplicated.
"""

#Extracting unique account IDs and writing them into a JSON file
AccountIDs = BoostedTransactions["AccountId"].values
UniqueAccounts = list(dict.fromkeys(AccountIDs))
with open("UniqueAccounts.json", "w") as File:
  json.dump(UniqueAccounts, File)
```

```
"""**Reading unique account IDs and manipulating time series**"""

#Reading unique accounts JSON file
with open("UniqueAccounts.json", "r") as File:
  UniqueAccounts = json.load(File)

#Converting transaction dates into months
BoostedTransactions["TransactionDate"] =
pd.to_datetime(BoostedTransactions["TransactionDate"])
BoostedTransactions["TransactionDate"] =
BoostedTransactions["TransactionDate"].dt.to_period("M").dt.to_timestamp()

#Grouping the dataset by account and month and calculating the net sum of
transaction amounts per month
GroupedSumOfTransactionsByAccountAndMonth =
BoostedTransactions.groupby(by=["AccountId",
"TransactionDate"]).agg(["sum"])["Amount"]
print(GroupedSumOfTransactionsByAccountAndMonth)

#Creating a list to store each account's time series in
DataFrames = []

#Imputing 0 values into any gaps in each account's time series
for i in range(0, len(UniqueAccounts)):
  ResetDataFrame =
GroupedSumOfTransactionsByAccountAndMonth.reset_index()
  TimeSeries = ResetDataFrame[ResetDataFrame["AccountId"] ==
list(UniqueAccounts)[i]]
  TimeSeries = TimeSeries.set_index("TransactionDate")

  TimeSeries = TimeSeries.resample("MS").sum()
  for j in range(0, len(TimeSeries)):
    if TimeSeries.iloc[j]["AccountId"] == 0:
      TimeSeries.at[TimeSeries.index[j], "AccountId"] = list(UniqueAccounts)[i]

  DataFrames.append(TimeSeries)

#Creating a new DataFrame containing all accounts' time series
NewGroupedDataset = pd.concat(DataFrames)
NewGroupedDataset = NewGroupedDataset.reset_index()
NewGroupedDataset = NewGroupedDataset.set_index(["AccountId",
"TransactionDate"])
print(NewGroupedDataset)

#Calculating the cumulative transaction sum for each account across the time series
GroupedCumulativeSumOfTransactionsByAccountAndMonth =
NewGroupedDataset.groupby(level=0).cumsum().reset_index()
print(GroupedCumulativeSumOfTransactionsByAccountAndMonth)

"""**Key LSTM Loops (split up for memory reasons)**
```

```
Batch for first 150 accounts
"""

#Creating models and forecasting for first 150 accounts
for idx, i in enumerate(UniqueAccounts):
    #Ensuring that only the first 150 accounts are processed
    if idx >= 150:
        break
    print(i)
    try:
        #Selecting only the selected account's time series
        ForecastingTimeSeries =
GroupedCumulativeSumOfTransactionsByAccountAndMonth[GroupedCumulative
SumOfTransactionsByAccountAndMonth["AccountId"] == i][["TransactionDate",
"sum"]]

        #Calculating the cutoff between training and testing sets
        TrainEnd = int(0.7 * len(ForecastingTimeSeries))

        #Splitting the data into training and testing sets
        TrainData = ForecastingTimeSeries[:TrainEnd]
        TestData = ForecastingTimeSeries[TrainEnd:]

        #Setting transaction month as the index
        TrainData = TrainData.set_index("TransactionDate")

        #Scaling the data using min/max scaling
        Scaler = MinMaxScaler(feature_range=(0,1))
        ScaledTrain = Scaler.fit_transform(TrainData)

        #Generating a batched time series for the LSTM to process
        TimeStep = 1
        TimeSeriesData = TimeseriesGenerator(ScaledTrain, ScaledTrain,
length=TimeStep, batch_size=1)

        #Declaring model layers
        Model = Sequential()
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh",
input_shape=(TimeStep, 1)))
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh"))
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh"))
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh"))
        Model.add(LSTM(units=16, activation="tanh"))
        Model.add(Dense(units=1, activation="sigmoid"))
        Model.add(Dense(units=1, activation="sigmoid"))
        Model.add(Dense(1))
        Model.compile(optimizer="adam", loss="mean_squared_error")

        #Training the model
```

```
        Model.fit(TimeSeriesData, epochs=20, batch_size=1, verbose=0)

        #Creating a list to store test predictions in
        TestPredictions = []

        #Creating the first testing batch
        FirstTestingBatch = ScaledTrain[-TimeStep:]
        CurrentBatch = FirstTestingBatch.reshape((1, TimeStep, 1))

        #Making predictions on the test dataset
        for h in range(len(TestData)):
            CurrentPrediction = Model.predict(CurrentBatch)[0]
            TestPredictions.append(CurrentPrediction)
            CurrentBatch = np.append(CurrentBatch[:, 1:, :], [[CurrentPrediction]],
axis=1)

        #Unscaling the test predictions
        Forecast = Scaler.inverse_transform(TestPredictions)

        #Calculating error rates and appending them into the errors CSV
        try:
          RMSE = root_mean_squared_error(Forecast, TestData["sum"])
          SI = abs(RMSE/abs(TestData["sum"]).mean())
          AccountError = pd.DataFrame({"AccountID" : [i],
                                       "RMSE" : [RMSE],
                                       "SI" : [SI]})
          AccountError.to_csv("LSTM Longer Subset Errors (Cumulative).csv",
mode='a', header=not os.path.exists("LSTM Longer Subset Errors
(Cumulative).csv"), index=False)
        #If a ValueError occurs, the loop is broken
        except ValueError:
          print("ValueError encountered")
          break

        #Creating a list to store future predictions
        FuturePredictions = []

        #Creating the first future batch
        FirstFutureBatch = np.array(TestPredictions[-TimeStep:])
        CurrentBatch = FirstFutureBatch.reshape((1, TimeStep, 1))

        #Forecasting the next 12 months
        for j in range(12):
            CurrentPrediction = Model.predict(CurrentBatch)[0]
            FuturePredictions.append(CurrentPrediction)
            CurrentBatch = np.append(CurrentBatch[:, 1:, :], [[CurrentPrediction]],
axis=1)

        #Unscaling the future forecast
        FutureForecast = Scaler.inverse_transform(FuturePredictions)
```

```
        #Calculating the starting amount, ending amount and net change and appending
these into the forecasts CSV
        NetChange = FutureForecast[-1] - FutureForecast[0]
        StartingAmount = FutureForecast[0]
        EndingAmount = FutureForecast[-1]

        ForecastData = pd.DataFrame({"AccountID" : [i],
                                     "Net Change over Forecast Period" : [NetChange],
                                     "Starting Amount" : [StartingAmount],
                                     "Ending Amount" : [EndingAmount]})

        ForecastData.to_csv("LSTM Longer Subset Forecasts (Cumulative).csv",
mode='a', header=not os.path.exists("LSTM Longer Subset Forecasts
(Cumulative).csv"), index=False)

        print("Processed account number",i)
    #Where an error occurs, the user is informed about it
    except Exception as e:
        print(f"Error encountered processing account number {i}.")
        ErrorType, ErrorObject, ErrorTraceback = sys.exc_info()

        ErrorFilename = os.path.split(
            ErrorTraceback.tb_frame.f_code.co_filename
        )[1]

        ErrorMessage = str(e)

        ErrorLineNumber = ErrorTraceback.tb_lineno

        print(f'Exception Type: {ErrorType}')

        print(f'Exception Filename: {ErrorFilename}')

        print(f'Exception Line Number: {ErrorLineNumber}')

        print(f'Exception Message: {ErrorMessage}')
        break
    #Deleting model data from memory to conserve RAM
    finally:
        del Model, Forecast, TrainData, TestData, ForecastingTimeSeries
        gc.collect()

"""Batch for accounts 150-300"""

#Creating models and forecasting for accounts 150-300
for idx, i in enumerate(UniqueAccounts):
    #Ensuring that only accounts 150-300 are processed
    if idx < 150:
        pass
```

```
    else:
      if idx >= 300:
        break
      print(i)
      try:
          #Selecting only the chosen account's time series
          ForecastingTimeSeries =
GroupedCumulativeSumOfTransactionsByAccountAndMonth[GroupedCumulative
SumOfTransactionsByAccountAndMonth["AccountId"] == i][["TransactionDate",
"sum"]]

          #Calculating the cutoff point between training and testing sets
          TrainEnd = int(0.7 * len(ForecastingTimeSeries))

          #Splitting the data into training and testing sets
          TrainData = ForecastingTimeSeries[:TrainEnd]
          TestData = ForecastingTimeSeries[TrainEnd:]

          #Setting transaction month as the index
          TrainData = TrainData.set_index("TransactionDate")

          #Scaling the data using min/max scaling
          Scaler = MinMaxScaler(feature_range=(0,1))
          ScaledTrain = Scaler.fit_transform(TrainData)

          #Generating a batched time series for the LSTM to process
          TimeStep = 1
          TimeSeriesData = TimeseriesGenerator(ScaledTrain, ScaledTrain,
length=TimeStep, batch_size=1)

          #Declaring model layers
          Model = Sequential()
          Model.add(LSTM(units=16, return_sequences=True, activation="tanh",
input_shape=(TimeStep, 1)))
          Model.add(LSTM(units=16, return_sequences=True, activation="tanh"))
          Model.add(LSTM(units=16, return_sequences=True, activation="tanh"))
          Model.add(LSTM(units=16, return_sequences=True, activation="tanh"))
          Model.add(LSTM(units=16, activation="tanh"))
          Model.add(Dense(units=1, activation="sigmoid"))
          Model.add(Dense(units=1, activation="sigmoid"))
          Model.add(Dense(1))
          Model.compile(optimizer="adam", loss="mean_squared_error")

          #Training the model
          Model.fit(TimeSeriesData, epochs=20, batch_size=1, verbose=0)

          #Creating a list to store test predictions
          TestPredictions = []

          #Creating the first testing batch
```

```
        FirstTestingBatch = ScaledTrain[-TimeStep:]
        CurrentBatch = FirstTestingBatch.reshape((1, TimeStep, 1))

        #Forecasting on the test data
        for h in range(len(TestData)):
            CurrentPrediction = Model.predict(CurrentBatch)[0]
            TestPredictions.append(CurrentPrediction)
            CurrentBatch = np.append(CurrentBatch[:, 1:, :], [[CurrentPrediction]],
axis=1)

        #Unscaling the test forecast
        Forecast = Scaler.inverse_transform(TestPredictions)

        #Calculating error rates and appending them into the errors CSV
        try:
          RMSE = root_mean_squared_error(Forecast, TestData["sum"])
          SI = abs(RMSE/abs(TestData["sum"]).mean())
          AccountError = pd.DataFrame({"AccountID" : [i],
                                       "RMSE" : [RMSE],
                                       "SI" : [SI]})
          AccountError.to_csv("LSTM Longer Subset Errors (Cumulative).csv",
mode='a', header=not os.path.exists("LSTM Longer Subset Errors
(Cumulative).csv"), index=False)
        except ValueError:
          print("ValueError encountered")
          break

        #Creating a list to store future predictions
        FuturePredictions = []

        #Creating the first future batch
        FirstFutureBatch = np.array(TestPredictions[-TimeStep:])
        CurrentBatch = FirstFutureBatch.reshape((1, TimeStep, 1))

        #Forecasting the next 12 months
        for j in range(12):
            CurrentPrediction = Model.predict(CurrentBatch)[0]
            FuturePredictions.append(CurrentPrediction)
            CurrentBatch = np.append(CurrentBatch[:, 1:, :], [[CurrentPrediction]],
axis=1)

        #Unscaling the future forecast
        FutureForecast = Scaler.inverse_transform(FuturePredictions)

        #Calculating starting amount, ending amount and net change
        NetChange = FutureForecast[-1] - FutureForecast[0]
        StartingAmount = FutureForecast[0]
        EndingAmount = FutureForecast[-1]

        ForecastData = pd.DataFrame({"AccountID" : [i],
```

```
                        "Net Change over Forecast Period" : [NetChange],
                        "Starting Amount" : [StartingAmount],
                        "Ending Amount" : [EndingAmount]})

        ForecastData.to_csv("LSTM Forecasts (Cumulative).csv", mode='a',
header=not os.path.exists("LSTM Forecasts (Cumulative).csv"), index=False)

        print("Processed account number",i)
    #Where an error occurs, the user is informed about it
    except Exception as e:
        print(f"Error encountered processing account number {i}.")
        ErrorType, ErrorObject, ErrorTraceback = sys.exc_info()

        ErrorFilename = os.path.split(
            ErrorTraceback.tb_frame.f_code.co_filename
        )[1]

        ErrorMessage = str(e)

        ErrorLineNumber = ErrorTraceback.tb_lineno

        print(f'Exception Type: {ErrorType}')

        print(f'Exception Filename: {ErrorFilename}')

        print(f'Exception Line Number: {ErrorLineNumber}')

        print(f'Exception Message: {ErrorMessage}')
        break
    #Deleting model data from memory to conserve RAM
    finally:
        del Model, Forecast, TrainData, TestData, ForecastingTimeSeries
        gc.collect()

"""Batch for accounts 300-450"""

#Creating models and forecasting for accounts 300-450
for idx, i in enumerate(UniqueAccounts):
    #Ensuring that only accounts 300-450 are processed
    if idx < 300:
        pass
    else:
      if idx >= 450:
        break
      print(i)
      try:
        #Selecting only the chosen account's time series
        ForecastingTimeSeries =
GroupedCumulativeSumOfTransactionsByAccountAndMonth[GroupedCumulative
```

```
SumOfTransactionsByAccountAndMonth["AccountId"] == i][["TransactionDate",
"sum"]]

        #Calculating the cutoff point between training and testing sets
        TrainEnd = int(0.7 * len(ForecastingTimeSeries))

        #Splitting the data into training and testing sets
        TrainData = ForecastingTimeSeries[:TrainEnd]
        TestData = ForecastingTimeSeries[TrainEnd:]

        #Setting transaction month as the index
        TrainData = TrainData.set_index("TransactionDate")

        #Scaling the data using min/max scaling
        Scaler = MinMaxScaler(feature_range=(0,1))
        ScaledTrain = Scaler.fit_transform(TrainData)

        #Generating a batched time series that the LSTM can process
        TimeStep = 1
        TimeSeriesData = TimeseriesGenerator(ScaledTrain, ScaledTrain,
length=TimeStep, batch_size=1)

        #Declaring model layers
        Model = Sequential()
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh",
input_shape=(TimeStep, 1)))
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh"))
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh"))
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh"))
        Model.add(LSTM(units=16, activation="tanh"))
        Model.add(Dense(units=1, activation="sigmoid"))
        Model.add(Dense(units=1, activation="sigmoid"))
        Model.add(Dense(1))
        Model.compile(optimizer="adam", loss="mean_squared_error")

        #Training the model
        Model.fit(TimeSeriesData, epochs=20, batch_size=1, verbose=0)

        #Creating a list to store test predictions
        TestPredictions = []

        #Generating the first test batch
        FirstTestingBatch = ScaledTrain[-TimeStep:]
        CurrentBatch = FirstTestingBatch.reshape((1, TimeStep, 1))

        #Forecasting on the test dataset
        for h in range(len(TestData)):
            CurrentPrediction = Model.predict(CurrentBatch)[0]
            TestPredictions.append(CurrentPrediction)
```

```
            CurrentBatch = np.append(CurrentBatch[:, 1:, :], [[CurrentPrediction]],
axis=1)

        #Unscaling the test forecast
        Forecast = Scaler.inverse_transform(TestPredictions)

        #Calculating error rates and appending them into the errors CSV
        try:
          RMSE = root_mean_squared_error(Forecast, TestData["sum"])
          SI = abs(RMSE/abs(TestData["sum"]).mean())
          AccountError = pd.DataFrame({"AccountID" : [i],
                                       "RMSE" : [RMSE],
                                       "SI" : [SI]})
          AccountError.to_csv("LSTM Longer Subset Errors (Cumulative).csv",
mode='a', header=not os.path.exists("LSTM Longer Subset Errors
(Cumulative).csv"), index=False)
        #When a ValueError occurs, the loop is broken
        except ValueError:
          print("ValueError encountered")
          break

        #Creating a list to store future predictions
        FuturePredictions = []

        #Creating the first future batch
        FirstFutureBatch = np.array(TestPredictions[-TimeStep:])
        CurrentBatch = FirstFutureBatch.reshape((1, TimeStep, 1))

        #Forecasting the next 12 months
        for j in range(12):
            CurrentPrediction = Model.predict(CurrentBatch)[0]
            FuturePredictions.append(CurrentPrediction)
            CurrentBatch = np.append(CurrentBatch[:, 1:, :], [[CurrentPrediction]],
axis=1)

        #Unscaling the future forecast
        FutureForecast = Scaler.inverse_transform(FuturePredictions)

        #Calculating starting amount, ending amount and net change and appending
them into the forecasts CSV
        NetChange = FutureForecast[-1] - FutureForecast[0]
        StartingAmount = FutureForecast[0]
        EndingAmount = FutureForecast[-1]

        ForecastData = pd.DataFrame({"AccountID" : [i],
                             "Net Change over Forecast Period" : [NetChange],
                             "Starting Amount" : [StartingAmount],
                             "Ending Amount" : [EndingAmount]})
```

```
        ForecastData.to_csv("LSTM Longer Subset Forecasts (Cumulative).csv",
mode='a', header=not os.path.exists("LSTM Longer Subset Forecasts
(Cumulative).csv"), index=False)

        print("Processed account number",i)
    #When an error occurs, the user is informed about it
    except Exception as e:
        print(f"Error encountered processing account number {i}.")
        ErrorType, ErrorObject, ErrorTraceback = sys.exc_info()

        ErrorFilename = os.path.split(
            ErrorTraceback.tb_frame.f_code.co_filename
        )[1]

        ErrorMessage = str(e)

        ErrorLineNumber = ErrorTraceback.tb_lineno

        print(f'Exception Type: {ErrorType}')

        print(f'Exception Filename: {ErrorFilename}')

        print(f'Exception Line Number: {ErrorLineNumber}')

        print(f'Exception Message: {ErrorMessage}')
        break
    #Deleting model data from memory to conserve RAM
    finally:
        del Model, Forecast, TrainData, TestData, ForecastingTimeSeries
        gc.collect()

"""Batch for final 68 accounts"""

#Creating models and forecasting for the final 68 accounts
for idx, i in enumerate(UniqueAccounts):
    #Ensuring that only the final 68 accounts are processed
    if idx < 450:
        pass
    else:
      if idx >= 600:
        break
      print(i)
      try:
        #Selecting only the chosen account's time series
        ForecastingTimeSeries =
GroupedCumulativeSumOfTransactionsByAccountAndMonth[GroupedCumulative
SumOfTransactionsByAccountAndMonth["AccountId"] == i][["TransactionDate",
"sum"]]

        #Calculating the cutoff point between the training and testing sets
```

```
        TrainEnd = int(0.7 * len(ForecastingTimeSeries))

        #Splitting the data into training and testing sets
        TrainData = ForecastingTimeSeries[:TrainEnd]
        TestData = ForecastingTimeSeries[TrainEnd:]

        #Setting the transaction month as an index
        TrainData = TrainData.set_index("TransactionDate")

        #Scaling the data using min/max scaling
        Scaler = MinMaxScaler(feature_range=(0,1))
        ScaledTrain = Scaler.fit_transform(TrainData)

        #Generating a batched time series that the LSTM can process
        TimeStep = 1
        TimeSeriesData = TimeseriesGenerator(ScaledTrain, ScaledTrain,
length=TimeStep, batch_size=1)

        #Declaring model layers
        Model = Sequential()
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh",
input_shape=(TimeStep, 1)))
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh"))
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh"))
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh"))
        Model.add(LSTM(units=16, activation="tanh"))
        Model.add(Dense(units=1, activation="sigmoid"))
        Model.add(Dense(units=1, activation="sigmoid"))
        Model.add(Dense(1))
        Model.compile(optimizer="adam", loss="mean_squared_error")

        #Training the model
        Model.fit(TimeSeriesData, epochs=20, batch_size=1, verbose=0)

        #Creating a list to store test predictions
        TestPredictions = []

        #Creating the first testing batch
        FirstTestingBatch = ScaledTrain[-TimeStep:]
        CurrentBatch = FirstTestingBatch.reshape((1, TimeStep, 1))

        #Forecasting on the test data
        for h in range(len(TestData)):
            CurrentPrediction = Model.predict(CurrentBatch)[0]
            TestPredictions.append(CurrentPrediction)
            CurrentBatch = np.append(CurrentBatch[:, 1:, :], [[CurrentPrediction]],
axis=1)

        #Unscaling the test forecast
        Forecast = Scaler.inverse_transform(TestPredictions)
```

```
        #Calculating the error rates and appending into the errors CSV file
        try:
          RMSE = root_mean_squared_error(Forecast, TestData["sum"])
          SI = abs(RMSE/abs(TestData["sum"]).mean())
          AccountError = pd.DataFrame({"AccountID" : [i],
                                       "RMSE" : [RMSE],
                                       "SI" : [SI]})
          AccountError.to_csv("LSTM Longer Subset Errors (Cumulative).csv",
mode='a', header=not os.path.exists("LSTM Longer Subset Errors
(Cumulative).csv"), index=False)
        #When a ValueError occurs, the loop is broken
        except ValueError:
          print("ValueError encountered")
          break

        #Creating a list to store future predictions
        FuturePredictions = []

        #Creating the first future batch
        FirstFutureBatch = np.array(TestPredictions[-TimeStep:])
        CurrentBatch = FirstFutureBatch.reshape((1, TimeStep, 1))

        #Forecasting the next 12 months
        for j in range(12):
            CurrentPrediction = Model.predict(CurrentBatch)[0]
            FuturePredictions.append(CurrentPrediction)
            CurrentBatch = np.append(CurrentBatch[:, 1:, :], [[CurrentPrediction]],
axis=1)

        #Unscaling the future forecast
        FutureForecast = Scaler.inverse_transform(FuturePredictions)

        #Calculating the starting amount, ending amount and net change
        NetChange = FutureForecast[-1] - FutureForecast[0]
        StartingAmount = FutureForecast[0]
        EndingAmount = FutureForecast[-1]

        ForecastData = pd.DataFrame({"AccountID" : [i],
                             "Net Change over Forecast Period" : [NetChange],
                             "Starting Amount" : [StartingAmount],
                             "Ending Amount" : [EndingAmount]})

        ForecastData.to_csv("LSTM Longer Subset Forecasts (Cumulative).csv",
mode='a', header=not os.path.exists("LSTM Longer Subset Forecasts
(Cumulative).csv"), index=False)

        print("Processed account number",i)
    #Where an error occurs, the user is informed about it
    except Exception as e:
```

```
        print(f"Error encountered processing account number {i}.")
        ErrorType, ErrorObject, ErrorTraceback = sys.exc_info()

        ErrorFilename = os.path.split(
           ErrorTraceback.tb_frame.f_code.co_filename
        )[1]

        ErrorMessage = str(e)

        ErrorLineNumber = ErrorTraceback.tb_lineno

        print(f'Exception Type: {ErrorType}')

        print(f'Exception Filename: {ErrorFilename}')

        print(f'Exception Line Number: {ErrorLineNumber}')

        print(f'Exception Message: {ErrorMessage}')
        break
    #Deleting the model data from memory to conserve RAM
    finally:
        del Model, Forecast, TrainData, TestData, ForecastingTimeSeries
        gc.collect()

"""**Calculating Error Statistics**"""

#Calculating averaged error statistics
Errors = pd.read_csv("LSTM Longer Subset Errors (Cumulative).csv")
MeanRMSE = Errors["RMSE"].mean()
print("Mean RMSE:",MeanRMSE)
MedianRMSE = Errors["RMSE"].median()
print("Median RMSE:",MedianRMSE)
MeanSI = Errors["SI"].mean()
print("Mean SI:",MeanSI)
MedianSI = Errors["SI"].median()
print("Median SI:",MedianSI)
```

6.2.2.4 Appendix 2.2.4: LSTM Forecasts on Longer Subset (Month-by-Month).py (converted from .ipynb)

```
# -*- coding: utf-8 -*-
"""LSTM Forecasts on Longer Subset (Month-by-Month).ipynb

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/1LAhya1o-qfVy2qS2jNY_NA07Z7WNoikV

**Importing Libraries and Data**
"""

#Importing libraries
import pandas as pd
import statsmodels.api as sm
from sklearn.metrics import root_mean_squared_error
from statsmodels.tsa.stattools import acf
import matplotlib.pyplot as plt
import prophet
import numpy as np
import warnings
import gc
import os
import sys
import json
from keras.models import Sequential
from keras.layers import LSTM
from keras.layers import Dense
from keras.layers import Dropout
from keras.callbacks import EarlyStopping
from tensorflow.keras.preprocessing.sequence import TimeseriesGenerator
from sklearn.preprocessing import MinMaxScaler

#Ignoring warnings
warnings.filterwarnings("ignore")

#Reading in data and extracting unique account ID values
BoostedTransactions = pd.read_csv("Longer Subset of Transaction Data (14 Months).csv")
AccountIDs = BoostedTransactions["AccountId"].values
#print(AccountIDs)
UniqueAccounts = set(AccountIDs)

#Converting transaction dates to months
BoostedTransactions["TransactionDate"] = pd.to_datetime(BoostedTransactions["TransactionDate"])
BoostedTransactions["TransactionDate"] = BoostedTransactions["TransactionDate"].dt.to_period("M").dt.to_timestamp()
```

```
#Grouping transactions by account and month and calculating the net transaction
sum per month for each account
GroupedSumOfTransactionsByAccountAndMonth =
BoostedTransactions.groupby(by=["AccountId",
"TransactionDate"]).agg(["sum"])["Amount"]
print(GroupedSumOfTransactionsByAccountAndMonth)
GroupedSumOfTransactionsByAccountAndMonth =
GroupedSumOfTransactionsByAccountAndMonth.reset_index()

"""**Extracting unique Account IDs and writing them into JSON file (only run
before first batch)**

Only run this block before running the first batch. This is to maintain the order of
accounts and ensure no duplicate forecasts.
"""

#Extracting the unique account IDs and writing them into a JSON file
AccountIDs = BoostedTransactions["AccountId"].values
UniqueAccounts = list(dict.fromkeys(AccountIDs))
with open("UniqueAccounts.json", "w") as File:
  json.dump(UniqueAccounts, File)

"""**Reading JSON file and preparing the data**"""

#Reading unique accounts from JSON file
with open("UniqueAccounts.json", "r") as File:
  UniqueAccounts = json.load(File)

#Converting transaction dates into months
BoostedTransactions["TransactionDate"] =
pd.to_datetime(BoostedTransactions["TransactionDate"])
BoostedTransactions["TransactionDate"] =
BoostedTransactions["TransactionDate"].dt.to_period("M").dt.to_timestamp()

#Grouping the transactions by account and month and summating the net transaction
sum per month for each account
GroupedSumOfTransactionsByAccountAndMonth =
BoostedTransactions.groupby(by=["AccountId",
"TransactionDate"]).agg(["sum"])["Amount"]
print(GroupedSumOfTransactionsByAccountAndMonth)

#Creating a list to store each account's time series
DataFrames = []

#Imputing 0 values into gaps in each account's time series
for i in range(0, len(UniqueAccounts)):
  ResetDataFrame =
GroupedSumOfTransactionsByAccountAndMonth.reset_index()
```

```
  TimeSeries = ResetDataFrame[ResetDataFrame["AccountId"] ==
list(UniqueAccounts)[i]]
  TimeSeries = TimeSeries.set_index("TransactionDate")

  TimeSeries = TimeSeries.resample("MS").sum()
  for j in range(0, len(TimeSeries)):
    if TimeSeries.iloc[j]["AccountId"] == 0:
      TimeSeries.at[TimeSeries.index[j], "AccountId"] = list(UniqueAccounts)[i]

  DataFrames.append(TimeSeries)

#Grouping all time series together into a new DataFrame
NewGroupedDataset = pd.concat(DataFrames)
NewGroupedDataset = NewGroupedDataset.reset_index()
print(NewGroupedDataset)

"""**Key LSTM Loops (split up for memory reasons)**

Batch for first 150 accounts
"""

#Creating models and forecasting for the first 150 accounts
for idx, i in enumerate(UniqueAccounts):
    #Ensuring only the first 150 accounts are processed
    if idx >= 150:
        break
    print(i)
    try:
        #Selecting only the chosen account's time series and ensuring it has no gaps
        ForecastingTimeSeries =
NewGroupedDataset[NewGroupedDataset["AccountId"] == i][["TransactionDate",
"sum"]]

        ForecastingTimeSeries = ForecastingTimeSeries.set_index("TransactionDate")

        ForecastingTimeSeries = ForecastingTimeSeries.resample("MS").sum()

        ForecastingTimeSeries = ForecastingTimeSeries.reset_index()

        #Calculating the cutoff between training and testing sets
        TrainEnd = int(0.7 * len(ForecastingTimeSeries))

        #Splitting the data into training and testing sets
        TrainData = ForecastingTimeSeries[:TrainEnd]
        TestData = ForecastingTimeSeries[TrainEnd:]

        #Setting transaction month as the index
        TrainData = TrainData.set_index("TransactionDate")

        #Scaling the data using min/max scaling
```

```
        Scaler = MinMaxScaler(feature_range=(0,1))
        ScaledTrain = Scaler.fit_transform(TrainData)

        #Generating a batched time series that the LSTM can process
        TimeStep = 1
        TimeSeriesData = TimeseriesGenerator(ScaledTrain, ScaledTrain,
length=TimeStep, batch_size=1)

        #Declaring model layers
        Model = Sequential()
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh",
input_shape=(TimeStep, 1)))
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh"))
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh"))
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh"))
        Model.add(LSTM(units=16, activation="tanh"))
        Model.add(Dense(units=1, activation="sigmoid"))
        Model.add(Dense(units=1, activation="sigmoid"))
        Model.add(Dense(1))
        Model.compile(optimizer="adam", loss="mean_squared_error")

        #Training the model
        Model.fit(TimeSeriesData, epochs=20, batch_size=1, verbose=0)

        #Creating a list to store test predictions
        TestPredictions = []

        #Creating the first testing batch
        FirstTestingBatch = ScaledTrain[-TimeStep:]
        CurrentBatch = FirstTestingBatch.reshape((1, TimeStep, 1))

        #Forecasting for the test dataset
        for h in range(len(TestData)):
            CurrentPrediction = Model.predict(CurrentBatch)[0]
            TestPredictions.append(CurrentPrediction)
            CurrentBatch = np.append(CurrentBatch[:, 1:, :], [[CurrentPrediction]],
axis=1)

        #Unscaling the test forecast
        Forecast = Scaler.inverse_transform(TestPredictions)

        #Calculating error rates and appending them to the errors CSV
        try:
          RMSE = root_mean_squared_error(Forecast, TestData["sum"])
          SI = abs(RMSE/abs(TestData["sum"]).mean())
          AccountError = pd.DataFrame({"AccountID" : [i],
                                       "RMSE" : [RMSE],
                                       "SI" : [SI]})
```

```
        AccountError.to_csv("LSTM Longer Subset Errors (Month-by-Month).csv",
mode='a', header=not os.path.exists("LSTM Longer Subset Errors (Month-by-
Month).csv"), index=False)
      #Where a ValueError occurs, the loop is broken
      except ValueError:
        print("ValueError encountered")
        break

      #Creating a list to store future predictions
      FuturePredictions = []

      #Creating the first future batch
      FirstFutureBatch = np.array(TestPredictions[-TimeStep:])
      CurrentBatch = FirstFutureBatch.reshape((1, TimeStep, 1))

      #Forecasting the next 12 months
      for j in range(12):
         CurrentPrediction = Model.predict(CurrentBatch)[0]
         FuturePredictions.append(CurrentPrediction)
         CurrentBatch = np.append(CurrentBatch[:, 1:, :], [[CurrentPrediction]],
axis=1)

      #Unscaling the future forecast
      FutureForecast = Scaler.inverse_transform(FuturePredictions)

      #Calculating the highest amount, lowest amount and difference and appending
them to the forecasts CSV file
      HighestAmount = FutureForecast.max()
      LowestAmount = FutureForecast.min()
      Difference = HighestAmount - LowestAmount

      ForecastData = pd.DataFrame({"AccountID" : [i],
                           "Difference over Forecast Period" : [Difference],
                           "Highest Amount" : [HighestAmount],
                           "Lowest Amount" : [LowestAmount]})

      ForecastData.to_csv("LSTM Longer Subset Forecasts (Month-by-Month).csv",
mode='a', header=not os.path.exists("LSTM Longer Subset Forecasts (Month-by-
Month).csv"), index=False)

      print("Processed account number",i)
   #Where an error occurs, the user is informed about it
   except Exception as e:
      print(f"Error encountered processing account number {i}.")
      ErrorType, ErrorObject, ErrorTraceback = sys.exc_info()

      ErrorFilename = os.path.split(
         ErrorTraceback.tb_frame.f_code.co_filename
      )[1]
```

```
        ErrorMessage = str(e)

        ErrorLineNumber = ErrorTraceback.tb_lineno

        print(f'Exception Type: {ErrorType}')

        print(f'Exception Filename: {ErrorFilename}')

        print(f'Exception Line Number: {ErrorLineNumber}')

        print(f'Exception Message: {ErrorMessage}')
        break
    #Deleting model data from memory to conserve RAM
    finally:
        del Model, Forecast, TrainData, TestData, ForecastingTimeSeries
        gc.collect()

"""Batch for accounts 150-300"""

#Creating models and forecasting for accounts 150-300
for idx, i in enumerate(UniqueAccounts):
    #Ensuring that only accounts 150-300 are processed
    if idx < 150:
        pass
    else:
      if idx >= 300:
        break
      print(i)
      try:
          #Selecting only the chosen account's time series and ensuring it has no gaps
          ForecastingTimeSeries =
NewGroupedDataset[NewGroupedDataset["AccountId"] == i][["TransactionDate",
"sum"]]

          ForecastingTimeSeries =
ForecastingTimeSeries.set_index("TransactionDate")

          ForecastingTimeSeries = ForecastingTimeSeries.resample("MS").sum()

          ForecastingTimeSeries = ForecastingTimeSeries.reset_index()

          #Calculating the cutoff point between the training and testing sets
          TrainEnd = int(0.7 * len(ForecastingTimeSeries))

          #Splitting the data into training and testing sets
          TrainData = ForecastingTimeSeries[:TrainEnd]
          TestData = ForecastingTimeSeries[TrainEnd:]

          #Setting transaction month as the index
          TrainData = TrainData.set_index("TransactionDate")
```

```
        #Scaling the data
        Scaler = MinMaxScaler(feature_range=(0,1))
        ScaledTrain = Scaler.fit_transform(TrainData)

        #Generating a batched time series that the LSTM can process
        TimeStep = 1
        TimeSeriesData = TimeseriesGenerator(ScaledTrain, ScaledTrain,
length=TimeStep, batch_size=1)

        #Declaring model layers
        Model = Sequential()
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh",
input_shape=(TimeStep, 1)))
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh"))
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh"))
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh"))
        Model.add(LSTM(units=16, activation="tanh"))
        Model.add(Dense(units=1, activation="sigmoid"))
        Model.add(Dense(units=1, activation="sigmoid"))
        Model.add(Dense(1))
        Model.compile(optimizer="adam", loss="mean_squared_error")

        #Training the model
        Model.fit(TimeSeriesData, epochs=20, batch_size=1, verbose=0)

        #Creating a list to store test predictions
        TestPredictions = []

        #Creating the first testing batch
        FirstTestingBatch = ScaledTrain[-TimeStep:]
        CurrentBatch = FirstTestingBatch.reshape((1, TimeStep, 1))

        #Forecasting on the test dataset
        for h in range(len(TestData)):
            CurrentPrediction = Model.predict(CurrentBatch)[0]
            TestPredictions.append(CurrentPrediction)
            CurrentBatch = np.append(CurrentBatch[:, 1:, :], [[CurrentPrediction]],
axis=1)

        #Unscaling the test predictions
        Forecast = Scaler.inverse_transform(TestPredictions)

        #Calculating error rates and appending them into the errors CSV
        try:
          RMSE = root_mean_squared_error(Forecast, TestData["sum"])
          SI = abs(RMSE/abs(TestData["sum"]).mean())
          AccountError = pd.DataFrame({"AccountID" : [i],
                                       "RMSE" : [RMSE],
                                       "SI" : [SI]})
```

```
          AccountError.to_csv("LSTM Longer Subset Errors (Month-by-Month).csv",
mode='a', header=not os.path.exists("LSTM Longer Subset Errors (Month-by-
Month).csv"), index=False)
        #Where a ValueError occurs, the loop is broken
        except ValueError:
          print("ValueError encountered")
          break

        #Creating a list to store future predictions
        FuturePredictions = []

        #Creating the first future batch
        FirstFutureBatch = np.array(TestPredictions[-TimeStep:])
        CurrentBatch = FirstFutureBatch.reshape((1, TimeStep, 1))

        #Forecasting the next 12 months
        for j in range(12):
           CurrentPrediction = Model.predict(CurrentBatch)[0]
           FuturePredictions.append(CurrentPrediction)
           CurrentBatch = np.append(CurrentBatch[:, 1:, :], [[CurrentPrediction]],
axis=1)

        #Unscaling the future forecast
        FutureForecast = Scaler.inverse_transform(FuturePredictions)

        #Calculating highest amount, lowest amount and difference and appending
them to the forecasts CSV
        HighestAmount = FutureForecast.max()
        LowestAmount = FutureForecast.min()
        Difference = HighestAmount - LowestAmount

        ForecastData = pd.DataFrame({"AccountID" : [i],
                             "Difference over Forecast Period" : [Difference],
                             "Highest Amount" : [HighestAmount],
                             "Lowest Amount" : [LowestAmount]})

        ForecastData.to_csv("LSTM Longer Subset Forecasts (Month-by-
Month).csv", mode='a', header=not os.path.exists("LSTM Longer Subset Forecasts
(Month-by-Month).csv"), index=False)

        print("Processed account number",i)
      #When an error occurs, the user is informed about it
      except Exception as e:
        print(f"Error encountered processing account number {i}.")
        ErrorType, ErrorObject, ErrorTraceback = sys.exc_info()

        ErrorFilename = os.path.split(
           ErrorTraceback.tb_frame.f_code.co_filename
        )[1]
```

```
        ErrorMessage = str(e)

        ErrorLineNumber = ErrorTraceback.tb_lineno

        print(f'Exception Type: {ErrorType}')

        print(f'Exception Filename: {ErrorFilename}')

        print(f'Exception Line Number: {ErrorLineNumber}')

        print(f'Exception Message: {ErrorMessage}')
        break
    #Deleting model data from memory to conserve RAM
    finally:
        del Model, Forecast, TrainData, TestData, ForecastingTimeSeries
        gc.collect()

"""Batch for Accounts 300-450"""

#Creating models and forecasting for accounts 300-450
for idx, i in enumerate(UniqueAccounts):
   #Ensuring that only accounts 300-450 are processed
   if idx < 300:
      pass
   else:
     if idx >= 450:
       break
     print(i)
     try:
        #Selecting only the chosen account's time series and ensuring it has no gaps
        ForecastingTimeSeries =
NewGroupedDataset[NewGroupedDataset["AccountId"] == i][["TransactionDate",
"sum"]]

        ForecastingTimeSeries =
ForecastingTimeSeries.set_index("TransactionDate")

        ForecastingTimeSeries = ForecastingTimeSeries.resample("MS").sum()

        ForecastingTimeSeries = ForecastingTimeSeries.reset_index()

        #Calculating the cutoff point between training and testing sets
        TrainEnd = int(0.7 * len(ForecastingTimeSeries))

        #Splitting the data into training and testing sets
        TrainData = ForecastingTimeSeries[:TrainEnd]
        TestData = ForecastingTimeSeries[TrainEnd:]

        #Setting transaction month as the index
        TrainData = TrainData.set_index("TransactionDate")
```

```
        #Scaling the data
        Scaler = MinMaxScaler(feature_range=(0,1))
        ScaledTrain = Scaler.fit_transform(TrainData)

        #Generating a batched time series that the LSTM can process
        TimeStep = 1
        TimeSeriesData = TimeseriesGenerator(ScaledTrain, ScaledTrain,
length=TimeStep, batch_size=1)

        #Declaring model layers
        Model = Sequential()
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh",
input_shape=(TimeStep, 1)))
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh"))
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh"))
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh"))
        Model.add(LSTM(units=16, activation="tanh"))
        Model.add(Dense(units=1, activation="sigmoid"))
        Model.add(Dense(units=1, activation="sigmoid"))
        Model.add(Dense(1))
        Model.compile(optimizer="adam", loss="mean_squared_error")

        #Training the model
        Model.fit(TimeSeriesData, epochs=20, batch_size=1, verbose=0)

        #Creating a list of test predictions
        TestPredictions = []

        #Creating the first testing batch
        FirstTestingBatch = ScaledTrain[-TimeStep:]
        CurrentBatch = FirstTestingBatch.reshape((1, TimeStep, 1))

        #Forecasting the test dataset
        for h in range(len(TestData)):
            CurrentPrediction = Model.predict(CurrentBatch)[0]
            TestPredictions.append(CurrentPrediction)
            CurrentBatch = np.append(CurrentBatch[:, 1:, :], [[CurrentPrediction]],
axis=1)

        #Unscaling the test forecast
        Forecast = Scaler.inverse_transform(TestPredictions)

        #Calculating error rates and appending them into the errors CSV
        try:
          RMSE = root_mean_squared_error(Forecast, TestData["sum"])
          SI = abs(RMSE/abs(TestData["sum"]).mean())
          AccountError = pd.DataFrame({"AccountID" : [i],
                                       "RMSE" : [RMSE],
                                       "SI" : [SI]})
```

```
          AccountError.to_csv("LSTM Longer Subset Errors (Month-by-Month).csv",
mode='a', header=not os.path.exists("LSTM Longer Subset Errors (Month-by-
Month).csv"), index=False)
        #Where a ValueError occurs, the loop is broken
        except ValueError:
          print("ValueError encountered")
          break

        #Creating a list to store future predictions
        FuturePredictions = []

        #Creating the first future batch
        FirstFutureBatch = np.array(TestPredictions[-TimeStep:])
        CurrentBatch = FirstFutureBatch.reshape((1, TimeStep, 1))

        #Forecasting the next 12 months
        for j in range(12):
           CurrentPrediction = Model.predict(CurrentBatch)[0]
           FuturePredictions.append(CurrentPrediction)
           CurrentBatch = np.append(CurrentBatch[:, 1:, :], [[CurrentPrediction]],
axis=1)

        #Unscaling the future forecast
        FutureForecast = Scaler.inverse_transform(FuturePredictions)

        #Calculating the highest amount, lowest amount and difference and appending
them to the forecast CSV files
        HighestAmount = FutureForecast.max()
        LowestAmount = FutureForecast.min()
        Difference = HighestAmount - LowestAmount

        ForecastData = pd.DataFrame({"AccountID" : [i],
                             "Difference over Forecast Period" : [Difference],
                             "Highest Amount" : [HighestAmount],
                             "Lowest Amount" : [LowestAmount]})

        ForecastData.to_csv("LSTM Longer Subset Forecasts (Month-by-
Month).csv", mode='a', header=not os.path.exists("LSTM Longer Subset Forecasts
(Month-by-Month).csv"), index=False)

        print("Processed account number",i)
      #Where an error occurs, the user is informed about it
      except Exception as e:
        print(f"Error encountered processing account number {i}.")
        ErrorType, ErrorObject, ErrorTraceback = sys.exc_info()

        ErrorFilename = os.path.split(
           ErrorTraceback.tb_frame.f_code.co_filename
        )[1]
```

```
        ErrorMessage = str(e)

        ErrorLineNumber = ErrorTraceback.tb_lineno

        print(f'Exception Type: {ErrorType}')

        print(f'Exception Filename: {ErrorFilename}')

        print(f'Exception Line Number: {ErrorLineNumber}')

        print(f'Exception Message: {ErrorMessage}')
        break
    #Deleting model data from memory to conserve RAM
    finally:
        del Model, Forecast, TrainData, TestData, ForecastingTimeSeries
        gc.collect()

"""Batch for final 68 accounts"""

#Creating models and forecasting for the final 68 accounts
for idx, i in enumerate(UniqueAccounts):
   #Ensuring only the final 68 accounts are processed
   if idx < 450:
      pass
   else:
     if idx >= 600:
       break
     print(i)
     try:
        #Selecting only the selected account's time series and ensuring it has no gaps
        ForecastingTimeSeries =
NewGroupedDataset[NewGroupedDataset["AccountId"] == i][["TransactionDate",
"sum"]]

        ForecastingTimeSeries =
ForecastingTimeSeries.set_index("TransactionDate")

        ForecastingTimeSeries = ForecastingTimeSeries.resample("MS").sum()

        ForecastingTimeSeries = ForecastingTimeSeries.reset_index()

        #Calculating the cutoff point between the training and testing sets
        TrainEnd = int(0.7 * len(ForecastingTimeSeries))

        #Splitting the data into training and testing sets
        TrainData = ForecastingTimeSeries[:TrainEnd]
        TestData = ForecastingTimeSeries[TrainEnd:]

        #Setting transaction month as the index
        TrainData = TrainData.set_index("TransactionDate")
```

```
        #Scaling the data
        Scaler = MinMaxScaler(feature_range=(0,1))
        ScaledTrain = Scaler.fit_transform(TrainData)

        #Generating a batched time series that the LSTM can process
        TimeStep = 1
        TimeSeriesData = TimeseriesGenerator(ScaledTrain, ScaledTrain,
length=TimeStep, batch_size=1)

        #Declaring model layers
        Model = Sequential()
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh",
input_shape=(TimeStep, 1)))
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh"))
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh"))
        Model.add(LSTM(units=16, return_sequences=True, activation="tanh"))
        Model.add(LSTM(units=16, activation="tanh"))
        Model.add(Dense(units=1, activation="sigmoid"))
        Model.add(Dense(units=1, activation="sigmoid"))
        Model.add(Dense(1))
        Model.compile(optimizer="adam", loss="mean_squared_error")

        #Training the model
        Model.fit(TimeSeriesData, epochs=20, batch_size=1, verbose=0)

        #Creating a list to store test predictions
        TestPredictions = []

        #Creating the first testing batch
        FirstTestingBatch = ScaledTrain[-TimeStep:]
        CurrentBatch = FirstTestingBatch.reshape((1, TimeStep, 1))

        #Forecasting on the test data
        for h in range(len(TestData)):
            CurrentPrediction = Model.predict(CurrentBatch)[0]
            TestPredictions.append(CurrentPrediction)
            CurrentBatch = np.append(CurrentBatch[:, 1:, :], [[CurrentPrediction]],
axis=1)

        #Unscaling the test forecast
        Forecast = Scaler.inverse_transform(TestPredictions)

        #Calculating error rates and appending them into the errors CSV
        try:
          RMSE = root_mean_squared_error(Forecast, TestData["sum"])
          SI = abs(RMSE/abs(TestData["sum"]).mean())
          AccountError = pd.DataFrame({"AccountID" : [i],
                                       "RMSE" : [RMSE],
                                       "SI" : [SI]})
```

```
          AccountError.to_csv("LSTM Longer Subset Errors (Month-by-Month).csv",
mode='a', header=not os.path.exists("LSTM Longer Subset Errors (Month-by-
Month).csv"), index=False)
        #Where a ValueError occurs, the loop is broken
        except ValueError:
          print("ValueError encountered")
          break

        #Creating a list to store future predictions
        FuturePredictions = []

        #Creating the first future batch
        FirstFutureBatch = np.array(TestPredictions[-TimeStep:])
        CurrentBatch = FirstFutureBatch.reshape((1, TimeStep, 1))

        #Forecasting the next 12 months
        for j in range(12):
           CurrentPrediction = Model.predict(CurrentBatch)[0]
           FuturePredictions.append(CurrentPrediction)
           CurrentBatch = np.append(CurrentBatch[:, 1:, :], [[CurrentPrediction]],
axis=1)

        #Unscaling the future forecast
        FutureForecast = Scaler.inverse_transform(FuturePredictions)

        #Calculating the highest amount, lowest amount and difference and appending
them into the forecasts CSV file
        HighestAmount = FutureForecast.max()
        LowestAmount = FutureForecast.min()
        Difference = HighestAmount - LowestAmount

        ForecastData = pd.DataFrame({"AccountID" : [i],
                             "Difference over Forecast Period" : [Difference],
                             "Highest Amount" : [HighestAmount],
                             "Lowest Amount" : [LowestAmount]})

        ForecastData.to_csv("LSTM Longer Subset Forecasts (Month-by-
Month).csv", mode='a', header=not os.path.exists("LSTM Longer Subset Forecasts
(Month-by-Month).csv"), index=False)

        print("Processed account number",i)
      #Where an error occurs, the user is informed about it
      except Exception as e:
        print(f"Error encountered processing account number {i}.")
        ErrorType, ErrorObject, ErrorTraceback = sys.exc_info()

        ErrorFilename = os.path.split(
           ErrorTraceback.tb_frame.f_code.co_filename
        )[1]
```

```
        ErrorMessage = str(e)

        ErrorLineNumber = ErrorTraceback.tb_lineno

        print(f'Exception Type: {ErrorType}')

        print(f'Exception Filename: {ErrorFilename}')

        print(f'Exception Line Number: {ErrorLineNumber}')

        print(f'Exception Message: {ErrorMessage}')
        break
    #Deleting model data from memory to conserve RAM
    finally:
        del Model, Forecast, TrainData, TestData, ForecastingTimeSeries
        gc.collect()

"""**Calculating Error Statistics**"""

#Calculating averaged error statistics
Errors = pd.read_csv("LSTM Longer Subset Errors (Month-by-Month).csv")
MeanRMSE = Errors["RMSE"].mean()
print("Mean RMSE:",MeanRMSE)
MedianRMSE = Errors["RMSE"].median()
print("Median RMSE:",MedianRMSE)
MeanSI = Errors["SI"].mean()
print("Mean SI:",MeanSI)
MedianSI = Errors["SI"].median()
print("Median SI:",MedianSI)
```

6.2.2.5 Appendix 2.2.5: Naïve Forecasts (Cumulative).py (converted from .ipynb)

```
# -*- coding: utf-8 -*-
"""Naive Forecasts (Cumulative).ipynb

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/19bD82gPhrdI6ZxR36zQehzNzrpeLjk6y

**Importing Libraries and Data**
"""

#Importing libraries
import pandas as pd
import statsmodels.api as sm
from sklearn.metrics import root_mean_squared_error
from statsmodels.tsa.stattools import acf
import matplotlib.pyplot as plt
import prophet
import numpy as np
import warnings
import gc
import os
import sys

#Ignoring warnings
warnings.filterwarnings("ignore")

#Reading in data and extracting unique account ID values
BoostedTransactions = pd.read_csv("Boosted Transaction Dataset.csv")
AccountIDs = BoostedTransactions["AccountId"].values
UniqueAccounts = set(AccountIDs)

#Converting transaction dates to months
BoostedTransactions["TransactionDate"] =
pd.to_datetime(BoostedTransactions["TransactionDate"])
BoostedTransactions["TransactionDate"] =
BoostedTransactions["TransactionDate"].dt.to_period("M").dt.to_timestamp()

#Grouping transactions by account and month and summating the net transaction
amount
GroupedSumOfTransactionsByAccountAndMonth =
BoostedTransactions.groupby(by=["AccountId",
"TransactionDate"]).agg(["sum"])["Amount"]
print(GroupedSumOfTransactionsByAccountAndMonth)

#Creating a list to store each account's time series in
DataFrames = []

#Imputing gaps in each account's time series with 0 values
```

```
for i in range(0, len(UniqueAccounts)):
  ResetDataFrame =
GroupedSumOfTransactionsByAccountAndMonth.reset_index()
  TimeSeries = ResetDataFrame[ResetDataFrame["AccountId"] ==
list(UniqueAccounts)[i]]
  TimeSeries = TimeSeries.set_index("TransactionDate")

  TimeSeries = TimeSeries.resample("MS").sum()
  for j in range(0, len(TimeSeries)):
    if TimeSeries.iloc[j]["AccountId"] == 0:
      TimeSeries.at[TimeSeries.index[j], "AccountId"] = list(UniqueAccounts)[i]

  DataFrames.append(TimeSeries)

#Combining all accounts' time series into one combined dataset
NewGroupedDataset = pd.concat(DataFrames)
NewGroupedDataset = NewGroupedDataset.reset_index()
NewGroupedDataset = NewGroupedDataset.set_index(["AccountId",
"TransactionDate"])
print(NewGroupedDataset)

#Calculating the cumulative sum of transactions across the time period for each
account's time series
GroupedCumulativeSumOfTransactionsByAccountAndMonth =
NewGroupedDataset.groupby(level=0).cumsum().reset_index()
print(GroupedCumulativeSumOfTransactionsByAccountAndMonth)

"""**Key Naive Forecast Loop**"""

#Performing a naive forecast on all accounts
for i in UniqueAccounts:
    print(i)
    try:
        #Selecting only the selected account's time series
        ForecastingTimeSeries =
GroupedCumulativeSumOfTransactionsByAccountAndMonth[GroupedCumulative
SumOfTransactionsByAccountAndMonth["AccountId"] == i][["TransactionDate",
"sum"]]

        #Calculating the cutoff point between training and testing sets
        TrainEnd = int(0.7 * len(ForecastingTimeSeries))

        #Splitting the data into training and testing sets
        TrainData = ForecastingTimeSeries[:TrainEnd]
        TestData = ForecastingTimeSeries[TrainEnd:]

        #Performing a naive forecast (using the last timestamp's record)
        TestData["Naive Forecast"] = TrainData.iloc[-1]["sum"]

        #Calculating error statistics and appending them into the errors CSV
```

```
        try:
          RMSE = root_mean_squared_error(TestData["Naive Forecast"],
TestData["sum"])
          SI = abs(RMSE/abs(TestData["sum"]).mean())
          AccountError = pd.DataFrame({"AccountID" : [i],
                                       "RMSE" : [RMSE],
                                       "SI" : [SI]})
          AccountError.to_csv("Naive Errors (Cumulative).csv", mode='a', header=not
os.path.exists("Naive Errors (Cumulative).csv"), index=False)
        #Where a value error occurs, the loop is broken
        except ValueError:
          print("ValueError encountered")
          break

        #Performing a naive forecast over the next 12 months
        LastDate = TestData["TransactionDate"].max()

        FutureDates = pd.date_range(start=LastDate +
pd.tseries.frequencies.to_offset("MS"),
                                    periods=12,
                                    freq="MS")

        FutureForecast = pd.DataFrame({"TransactionDate": FutureDates})

        FutureForecast["Forecast"] = TestData.iloc[-1]["Naive Forecast"]
        FutureForecastValues = FutureForecast["Forecast"].values

        #Calculating the starting amount, ending amount and net change and appending
them into the forecasts CSV
        NetChange = FutureForecastValues[-1] - FutureForecastValues[0]
        StartingAmount = FutureForecastValues[0]
        EndingAmount = FutureForecastValues[-1]

        ForecastData = pd.DataFrame({"AccountID" : [i],
                                     "Net Change over Forecast Period" : [NetChange],
                                     "Starting Amount" : [StartingAmount],
                                     "Ending Amount" : [EndingAmount]})

        ForecastData.to_csv("Naive Forecasts (Cumulative).csv", mode='a', header=not
os.path.exists("Naive Forecasts (Cumulative).csv"), index=False)

        print("Processed account number",i)
      #Where an error occurs, the user is informed about it
      except Exception as e:
        print(f"Error encountered processing account number {i}.")
        ErrorType, ErrorObject, ErrorTraceback = sys.exc_info()

        ErrorFilename = os.path.split(
            ErrorTraceback.tb_frame.f_code.co_filename
        )[1]
```

```
        ErrorMessage = str(e)

        ErrorLineNumber = ErrorTraceback.tb_lineno

        print(f'Exception Type: {ErrorType}')

        print(f'Exception Filename: {ErrorFilename}')

        print(f'Exception Line Number: {ErrorLineNumber}')

        print(f'Exception Message: {ExceptionMessage}')
        break
    #Deleting model data from memory to conserve RAM
    finally:
        del TrainData, TestData, ForecastingTimeSeries
        gc.collect()

"""**Calculating Error Statistics**"""

#Calculating average error statistics
Errors = pd.read_csv("Naive Errors (Cumulative).csv")
MeanRMSE = Errors["RMSE"].mean()
print("Mean RMSE:",MeanRMSE)
MedianRMSE = Errors["RMSE"].median()
print("Median RMSE:",MedianRMSE)
MeanSI = Errors["SI"].mean()
print("Mean SI:",MeanSI)
MedianSI = Errors["SI"].median()
print("Median SI:",MedianSI)
```

6.2.2.6 Appendix 2.2.6: Naïve Forecasts (Month-by-Month).py (converted from .ipynb)

```
# -*- coding: utf-8 -*-
"""Naive Forecasts (Month-by-Month).ipynb

Automatically generated by Colab.

Original file is located at

https://colab.research.google.com/drive/1HeM_k6ZkL6Jv2zvTMWFLvld00aIWiYpA

**Importing Libraries and Data**
"""

#Importing libraries
import pandas as pd
import statsmodels.api as sm
from sklearn.metrics import root_mean_squared_error
from statsmodels.tsa.stattools import acf
import matplotlib.pyplot as plt
import prophet
import numpy as np
import warnings
import gc
import os
import sys

#Ignoring warnings
warnings.filterwarnings("ignore")

#Reading in data and extracting unique account ID values
BoostedTransactions = pd.read_csv("Boosted Transaction Dataset.csv")
AccountIDs = BoostedTransactions["AccountId"].values
UniqueAccounts = set(AccountIDs)

#Converting transaction dates to months
BoostedTransactions["TransactionDate"] =
pd.to_datetime(BoostedTransactions["TransactionDate"])
BoostedTransactions["TransactionDate"] =
BoostedTransactions["TransactionDate"].dt.to_period("M").dt.to_timestamp()

#Grouping transactions by account and month and summating the net transaction
amount for each grouping
GroupedSumOfTransactionsByAccountAndMonth =
BoostedTransactions.groupby(by=["AccountId",
"TransactionDate"]).agg(["sum"])["Amount"]
print(GroupedSumOfTransactionsByAccountAndMonth)
GroupedSumOfTransactionsByAccountAndMonth =
GroupedSumOfTransactionsByAccountAndMonth.reset_index()
```

```
"""**Key Naive Forecast Loop**"""

#Creating models and forecasting for all accounts
for i in UniqueAccounts:
   print(i)
   try:
      #Selecting only the selected account's time series and ensuring it has no gaps
      ForecastingTimeSeries =
GroupedSumOfTransactionsByAccountAndMonth[GroupedSumOfTransactionsBy
AccountAndMonth["AccountId"] == i][["TransactionDate", "sum"]]

      ForecastingTimeSeries = ForecastingTimeSeries.set_index("TransactionDate")

      ForecastingTimeSeries = ForecastingTimeSeries.resample("MS").sum()

      ForecastingTimeSeries = ForecastingTimeSeries.reset_index()

      #Calculating the cutoff between training and testing sets
      TrainEnd = int(0.7 * len(ForecastingTimeSeries))

      #Splitting the data into training and testing sets
      TrainData = ForecastingTimeSeries[:TrainEnd]
      TestData = ForecastingTimeSeries[TrainEnd:]

      #Performing a naive forecast on the test data
      TestData["Naive Forecast"] = TrainData.iloc[-1]["sum"]

      #Calculating error rates and appending them into the errors CSV
      try:
        RMSE = root_mean_squared_error(TestData["Naive Forecast"],
TestData["sum"])
        SI = abs(RMSE/abs(TestData["sum"]).mean())
        AccountError = pd.DataFrame({"AccountID" : [i],
                                     "RMSE" : [RMSE],
                                     "SI" : [SI]})
        AccountError.to_csv("Naive Errors (Month-by-Month).csv", mode='a',
header=not os.path.exists("Naive Errors (Month-by-Month).csv"), index=False)
      #Where a ValueError occurs, the loop is broken
      except ValueError:
        print("ValueError encountered")
        break

      #Performing a naive forecast over the next 12 months
      LastDate = TestData["TransactionDate"].max()

      FutureDates = pd.date_range(start=LastDate +
pd.tseries.frequencies.to_offset("MS"),
                                  periods=12,
                                  freq="MS")
```

```
        FutureForecast = pd.DataFrame({"TransactionDate": FutureDates})

        FutureForecast["Forecast"] = TestData.iloc[-1]["Naive Forecast"]
        FutureForecastValues = FutureForecast["Forecast"].values

        #Calculating highest amount, lowest amount and difference and appending
them into the forecasts CSV
        Difference = FutureForecastValues.max() - FutureForecastValues.min()
        HighestAmount = FutureForecastValues.max()
        LowestAmount = FutureForecastValues.min()

        ForecastData = pd.DataFrame({"AccountID" : [i],
                                     "Difference" : [Difference],
                                     "Highest Amount" : [HighestAmount],
                                     "Lowest Amount" : [LowestAmount]})

        ForecastData.to_csv("Naive Forecasts (Month-by-Month).csv", mode='a',
header=not os.path.exists("Naive Forecasts (Month-by-Month).csv"), index=False)

        print("Processed account number",i)
    #Where an error occurs, the user is informed about it
    except Exception as e:
        print(f"Error encountered processing account number {i}.")
        ErrorType, ErrorObject, ErrorTraceback = sys.exc_info()

        ErrorFilename = os.path.split(
            ErrorTraceback.tb_frame.f_code.co_filename
        )[1]

        ErrorMessage = str(e)

        ErrorLineNumber = ErrorTraceback.tb_lineno

        print(f'Exception Type: {ErrorType}')

        print(f'Exception Filename: {ErrorFilename}')

        print(f'Exception Line Number: {ErrorLineNumber}')

        print(f'Exception Message: {ErrorMessage}')
        break
    #Deleting model data from memory to conserve RAM
    finally:
        del TrainData, TestData, ForecastingTimeSeries
        gc.collect()

"""**Calculating Error Statistics**"""

#Calculating averaged error statistics
```

```
Errors = pd.read_csv("Naive Errors (Month-by-Month).csv")
MeanRMSE = Errors["RMSE"].mean()
print("Mean RMSE:",MeanRMSE)
MedianRMSE = Errors["RMSE"].median()
print("Median RMSE:",MedianRMSE)
MeanSI = Errors["SI"].mean()
print("Mean SI:",MeanSI)
MedianSI = Errors["SI"].median()
print("Median SI:",MedianSI)
```

6.2.2.7 Appendix 2.2.7: Naïve Forecasts on Longer Subset (Cumulative).py (converted from .ipynb)

```
# -*- coding: utf-8 -*-
"""Naive Forecasts on Longer Subset (Cumulative).ipynb

Automatically generated by Colab.

Original file is located at

https://colab.research.google.com/drive/1ir4M6PdVckDN1PiuGK1BgXoxC9Ywhvca

**Importing Libraries and Data**
"""

#Importing libraries
import pandas as pd
import statsmodels.api as sm
from sklearn.metrics import root_mean_squared_error
from statsmodels.tsa.stattools import acf
import matplotlib.pyplot as plt
import prophet
import numpy as np
import warnings
import gc
import os
import sys

#Ignoring warnings
warnings.filterwarnings("ignore")

#Reading in data and extracting unique account values
BoostedTransactions = pd.read_csv("Longer Subset of Transaction Data (14 Months).csv")
AccountIDs = BoostedTransactions["AccountId"].values
UniqueAccounts = set(AccountIDs)

#Converting transaction dates into months
BoostedTransactions["TransactionDate"] = pd.to_datetime(BoostedTransactions["TransactionDate"])
BoostedTransactions["TransactionDate"] = BoostedTransactions["TransactionDate"].dt.to_period("M").dt.to_timestamp()

#Grouping transactions by account and month and summating the net transaction amount for each grouping
GroupedSumOfTransactionsByAccountAndMonth = BoostedTransactions.groupby(by=["AccountId", "TransactionDate"]).agg(["sum"])["Amount"]
print(GroupedSumOfTransactionsByAccountAndMonth)
```

```
#Creating a list to store each account's time series in
DataFrames = []

#Imputing 0 values into the gaps in each account's time series
for i in range(0, len(UniqueAccounts)):
  ResetDataFrame =
GroupedSumOfTransactionsByAccountAndMonth.reset_index()
  TimeSeries = ResetDataFrame[ResetDataFrame["AccountId"] ==
list(UniqueAccounts)[i]]
  TimeSeries = TimeSeries.set_index("TransactionDate")

  TimeSeries = TimeSeries.resample("MS").sum()
  for j in range(0, len(TimeSeries)):
    if TimeSeries.iloc[j]["AccountId"] == 0:
      TimeSeries.at[TimeSeries.index[j], "AccountId"] = list(UniqueAccounts)[i]

  DataFrames.append(TimeSeries)

#Combining all accounts' time series together into a new DataFrame
NewGroupedDataset = pd.concat(DataFrames)
NewGroupedDataset = NewGroupedDataset.reset_index()
NewGroupedDataset = NewGroupedDataset.set_index(["AccountId",
"TransactionDate"])
print(NewGroupedDataset)

#Calculating the cumulative transaction sum over the time period for each account
GroupedCumulativeSumOfTransactionsByAccountAndMonth =
NewGroupedDataset.groupby(level=0).cumsum().reset_index()
print(GroupedCumulativeSumOfTransactionsByAccountAndMonth)

"""**Key Naive Forecast Loop**"""

#Creating models and forecasting for all accounts
for i in UniqueAccounts:
    print(i)
    try:
        #Selecting only the selected account's time series
        ForecastingTimeSeries =
GroupedCumulativeSumOfTransactionsByAccountAndMonth[GroupedCumulative
SumOfTransactionsByAccountAndMonth["AccountId"] == i][["TransactionDate",
"sum"]]

        #Calculating the cutoff point between training and testing sets
        TrainEnd = int(0.7 * len(ForecastingTimeSeries))

        #Splitting the data into training and testing sets
        TrainData = ForecastingTimeSeries[:TrainEnd]
        TestData = ForecastingTimeSeries[TrainEnd:]

        #Performing a naive forecast on the test data
```

```
        TestData["Naive Forecast"] = TrainData.iloc[-1]["sum"]

        #Calculating error rates and appending them to the errors CSV
        try:
          RMSE = root_mean_squared_error(TestData["Naive Forecast"],
TestData["sum"])
          SI = abs(RMSE/abs(TestData["sum"]).mean())
          AccountError = pd.DataFrame({"AccountID" : [i],
                                       "RMSE" : [RMSE],
                                       "SI" : [SI]})
          AccountError.to_csv("Naive Longer Subset Errors (Cumulative).csv",
mode='a', header=not os.path.exists("Naive Longer Subset Errors
(Cumulative).csv"), index=False)
        #Where a ValueError occurs, the loop is broken
        except ValueError:
          print("ValueError encountered")
          break

        #Performing a naive forecast for the next 12 months
        LastDate = TestData["TransactionDate"].max()

        FutureDates = pd.date_range(start=LastDate +
pd.tseries.frequencies.to_offset("MS"),
                                    periods=12,
                                    freq="MS")

        FutureForecast = pd.DataFrame({"TransactionDate": FutureDates})

        FutureForecast["Forecast"] = TestData.iloc[-1]["Naive Forecast"]
        FutureForecastValues = FutureForecast["Forecast"].values

        #Calculating starting amount, ending amount and net change and appending
them into the forecasts CSV
        NetChange = FutureForecastValues[-1] - FutureForecastValues[0]
        StartingAmount = FutureForecastValues[0]
        EndingAmount = FutureForecastValues[-1]

        ForecastData = pd.DataFrame({"AccountID" : [i],
                                     "Net Change over Forecast Period" : [NetChange],
                                     "Starting Amount" : [StartingAmount],
                                     "Ending Amount" : [EndingAmount]})

        ForecastData.to_csv("Naive Longer Subset Forecasts (Cumulative).csv",
mode='a', header=not os.path.exists("Naive Longer Subset Forecasts
(Cumulative).csv"), index=False)

        print("Processed account number",i)
    #Where an error occurs, the user is informed about it
    except Exception as e:
        print(f"Error encountered processing account number {i}.")
```

```
        ErrorType, ErrorObject, ErrorTraceback = sys.exc_info()

        ErrorFilename = os.path.split(
            ErrorTraceback.tb_frame.f_code.co_filename
        )[1]

        ErrorMessage = str(e)

        ErrorLineNumber = ErrorTraceback.tb_lineno

        print(f'Exception Type: {ErrorType}')

        print(f'Exception Filename: {ErrorFilename}')

        print(f'Exception Line Number: {ErrorLineNumber}')

        print(f'Exception Message: {ErrorMessage}')
        break
    #Deleting model data from memory to conserve RAM
    finally:
        del TrainData, TestData, ForecastingTimeSeries
        gc.collect()

"""**Calculating Error Statistics**"""

#Calculating averaged error statistics
Errors = pd.read_csv("Naive Longer Subset Errors (Cumulative).csv")
MeanRMSE = Errors["RMSE"].mean()
print("Mean RMSE:",MeanRMSE)
MedianRMSE = Errors["RMSE"].median()
print("Median RMSE:",MedianRMSE)
MeanSI = Errors["SI"].mean()
print("Mean SI:",MeanSI)
MedianSI = Errors["SI"].median()
print("Median SI:",MedianSI)
```

6.2.2.8 Appendix 2.2.8: Naïve Forecasts on Longer Subset (Month-by-Month).py (converted from .ipynb)

```
# -*- coding: utf-8 -*-
"""Naive Forecasts on Longer Subset (Month-by-Month).ipynb

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/1ZmV3-
00VAELnPsQEp7k88FZ8MN0aQ5ZN

**Importing Libraries and Data**
"""

#Importing libraries
import pandas as pd
import statsmodels.api as sm
from sklearn.metrics import root_mean_squared_error
from statsmodels.tsa.stattools import acf
import matplotlib.pyplot as plt
import prophet
import numpy as np
import warnings
import gc
import os
import sys

#Ignoring warnings
warnings.filterwarnings("ignore")

#Reading in data and extracting unique account values
BoostedTransactions = pd.read_csv("Longer Subset of Transaction Data (14
Months).csv")
AccountIDs = BoostedTransactions["AccountId"].values
UniqueAccounts = set(AccountIDs)

#Converting transaction dates into months
BoostedTransactions["TransactionDate"] =
pd.to_datetime(BoostedTransactions["TransactionDate"])
BoostedTransactions["TransactionDate"] =
BoostedTransactions["TransactionDate"].dt.to_period("M").dt.to_timestamp()

#Grouping transactions by account and month and summating net transaction
amounts for each grouping
GroupedSumOfTransactionsByAccountAndMonth =
BoostedTransactions.groupby(by=["AccountId",
"TransactionDate"]).agg(["sum"])["Amount"]
print(GroupedSumOfTransactionsByAccountAndMonth)
GroupedSumOfTransactionsByAccountAndMonth =
GroupedSumOfTransactionsByAccountAndMonth.reset_index()
```

```
"""**Key Naive Forecast Loop**"""

#Creating models and forecasting for all accounts
for i in UniqueAccounts:
   print(i)
   try:
      #Selecting only the selected account's time series and ensuring it has no gaps
      ForecastingTimeSeries =
GroupedSumOfTransactionsByAccountAndMonth[GroupedSumOfTransactionsBy
AccountAndMonth["AccountId"] == i][["TransactionDate", "sum"]]

      ForecastingTimeSeries = ForecastingTimeSeries.set_index("TransactionDate")

      ForecastingTimeSeries = ForecastingTimeSeries.resample("MS").sum()

      ForecastingTimeSeries = ForecastingTimeSeries.reset_index()

      #Calculating the cutoff point between the training and testing sets
      TrainEnd = int(0.7 * len(ForecastingTimeSeries))

      #Splitting the data into training and testing sets
      TrainData = ForecastingTimeSeries[:TrainEnd]
      TestData = ForecastingTimeSeries[TrainEnd:]

      #Performing a naive forecast on the test data
      TestData["Naive Forecast"] = TrainData.iloc[-1]["sum"]

      #Calculating error rates and appending them into the errors CSV
      try:
        RMSE = root_mean_squared_error(TestData["Naive Forecast"],
TestData["sum"])
        SI = abs(RMSE/abs(TestData["sum"]).mean())
        AccountError = pd.DataFrame({"AccountID" : [i],
                                     "RMSE" : [RMSE],
                                     "SI" : [SI]})
        AccountError.to_csv("Naive Longer Subset Errors (Month-by-Month).csv",
mode='a', header=not os.path.exists("Naive Longer Subset Errors (Month-by-
Month).csv"), index=False)
      #Where a ValueError occurs, the loop is broken
      except ValueError:
        print("ValueError encountered")
        break

      #Performing a naive forecast for the next 12 months
      LastDate = TestData["TransactionDate"].max()

      FutureDates = pd.date_range(start=LastDate +
pd.tseries.frequencies.to_offset("MS"),
                                  periods=12,
```

```
                        freq="MS")

    FutureForecast = pd.DataFrame({"TransactionDate": FutureDates})

    FutureForecast["Forecast"] = TestData.iloc[-1]["Naive Forecast"]
    FutureForecastValues = FutureForecast["Forecast"].values

    #Calculating highest amount, lowest amount and difference and appending
them into the forecasts CSV
    Difference = FutureForecastValues.max() - FutureForecastValues.min()
    HighestAmount = FutureForecastValues.max()
    LowestAmount = FutureForecastValues.min()

    ForecastData = pd.DataFrame({"AccountID" : [i],
                        "Difference" : [Difference],
                        "Highest Amount" : [HighestAmount],
                        "Lowest Amount" : [LowestAmount]})

    ForecastData.to_csv("Naive Longer Subset Forecasts (Month-by-Month).csv",
mode='a', header=not os.path.exists("Naive Longer Subset Forecasts (Month-by-
Month).csv"), index=False)

    print("Processed account number",i)
  #Where an error occurs, the user is told about it
  except Exception as e:
    print(f"Error encountered processing account number {i}.")
    ErrorType, ErrorObject, ErrorTraceback = sys.exc_info()

    ErrorFilename = os.path.split(
      ErrorTraceback.tb_frame.f_code.co_filename
    )[1]

    ErrorMessage = str(e)

    ErrorLineNumber = ErrorTraceback.tb_lineno

    print(f'Exception Type: {ErrorType}')

    print(f'Exception Filename: {ErrorFilename}')

    print(f'Exception Line Number: {ErrorLineNumber}')

    print(f'Exception Message: {ErrorMessage}')
    break
  #Deleting model data from memory to conserve RAM
  finally:
    del TrainData, TestData, ForecastingTimeSeries
    gc.collect()

"""**Calculating Error Statistics**"""
```

```
#Calculating averaged error statistics
Errors = pd.read_csv("Naive Longer Subset Errors (Month-by-Month).csv")
MeanRMSE = Errors["RMSE"].mean()
print("Mean RMSE:",MeanRMSE)
MedianRMSE = Errors["RMSE"].median()
print("Median RMSE:",MedianRMSE)
MeanSI = Errors["SI"].mean()
print("Mean SI:",MeanSI)
MedianSI = Errors["SI"].median()
print("Median SI:",MedianSI)
```

6.2.2.9 Appendix 2.2.9: Prophet Forecasts (Cumulative).py (converted from .ipynb)

```
# -*- coding: utf-8 -*-
"""Prophet Forecasts (Cumulative).ipynb

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/14FrTV9x2ozlXgxofe24ag84x-Ccm_iAY

**Importing Libraries and Data**
"""

#Importing libraries
import pandas as pd
import statsmodels.api as sm
from sklearn.metrics import root_mean_squared_error
from statsmodels.tsa.stattools import acf
import matplotlib.pyplot as plt
import prophet
import numpy as np
import warnings
import gc
import os
import sys

#Ignoring warnings
warnings.filterwarnings("ignore")

#Reading in data and extracting unique account values
BoostedTransactions = pd.read_csv("Boosted Transaction Dataset.csv")
AccountIDs = BoostedTransactions["AccountId"].values
UniqueAccounts = set(AccountIDs)

#Converting transaction dates into months
BoostedTransactions["TransactionDate"] =
pd.to_datetime(BoostedTransactions["TransactionDate"])
BoostedTransactions["TransactionDate"] =
BoostedTransactions["TransactionDate"].dt.to_period("M").dt.to_timestamp()

#Grouping transactions by account and month and summating the net transaction
amount for each grouping
GroupedSumOfTransactionsByAccountAndMonth =
BoostedTransactions.groupby(by=["AccountId",
"TransactionDate"]).agg(["sum"])["Amount"]
print(GroupedSumOfTransactionsByAccountAndMonth)

#Creating a list to store all account time series in
DataFrames = []

#Imputing 0 values into gaps in each account's time series
```

```
for i in range(0, len(UniqueAccounts)):
  ResetDataFrame =
GroupedSumOfTransactionsByAccountAndMonth.reset_index()
  TimeSeries = ResetDataFrame[ResetDataFrame["AccountId"] ==
list(UniqueAccounts)[i]]
  TimeSeries = TimeSeries.set_index("TransactionDate")

  TimeSeries = TimeSeries.resample("MS").sum()
  for j in range(0, len(TimeSeries)):
    if TimeSeries.iloc[j]["AccountId"] == 0:
      TimeSeries.at[TimeSeries.index[j], "AccountId"] = list(UniqueAccounts)[i]

  DataFrames.append(TimeSeries)

#Combining all accounts' time series into a new grouped dataset
NewGroupedDataset = pd.concat(DataFrames)
NewGroupedDataset = NewGroupedDataset.reset_index()
NewGroupedDataset = NewGroupedDataset.set_index(["AccountId",
"TransactionDate"])
print(NewGroupedDataset)

#Calculating the cumulative transaction sum over the time period for each account's
time series
GroupedCumulativeSumOfTransactionsByAccountAndMonth =
NewGroupedDataset.groupby(level=0).cumsum().reset_index()
print(GroupedCumulativeSumOfTransactionsByAccountAndMonth)

"""**Key Prophet Loop**"""

#Creating models and forecasting for all accounts
for i in UniqueAccounts:
    print(i)
    try:
        #Selecting only the selected account's time series
        ForecastingTimeSeries =
GroupedCumulativeSumOfTransactionsByAccountAndMonth[GroupedCumulative
SumOfTransactionsByAccountAndMonth["AccountId"] == i][["TransactionDate",
"sum"]]

        #Calculating the cutoff between training and testing sets
        TrainEnd = int(0.7 * len(ForecastingTimeSeries))

        #Splitting the data into training and testing sets and resetting column names in
each
        TrainData = ForecastingTimeSeries[:TrainEnd]
        TestData = ForecastingTimeSeries[TrainEnd:]

        TrainData.columns = ["ds", "y"]
        TestData.columns = ["ds", "y"]
```

```
        #Declaring and training the model
        Model = prophet.Prophet()
        Model.fit(TrainData)

        #Forecasting the test data
        Forecast = Model.predict(TestData)

        #Calculating error rates and appending them into the errors CSV
        try:
          #print(IdealNumberOfStreams)
          RMSE = root_mean_squared_error(Forecast["yhat"], TestData["y"])
          SI = abs(RMSE/abs(TestData["y"]).mean())
          AccountError = pd.DataFrame({"AccountID" : [i],
                                       "RMSE" : [RMSE],
                                       "SI" : [SI]})
          AccountError.to_csv("Prophet Errors (Cumulative).csv", mode='a', header=not
os.path.exists("Prophet Errors (Cumulative).csv"), index=False)
        #Where a ValueError occurs, the loop is broken
        except ValueError:
          print("ValueError encountered")
          break

        #Forecasting the next 12 months
        Future = Model.make_future_dataframe(periods=12, freq="MS")
        FutureForecast = Model.predict(Future)

        FutureForecastTail = FutureForecast.tail(12)
        FutureForecastValues = FutureForecastTail["yhat"].values

        #Calculating starting amount, ending amount and net change and appending
them into the forecasts CSV
        NetChange = FutureForecastValues[-1] - FutureForecastValues[0]
        StartingAmount = FutureForecastValues[0]
        EndingAmount = FutureForecastValues[-1]

        ForecastData = pd.DataFrame({"AccountID" : [i],
                                     "Net Change over Forecast Period" : [NetChange],
                                     "Starting Amount" : [StartingAmount],
                                     "Ending Amount" : [EndingAmount]})

        ForecastData.to_csv("Prophet Forecasts (Cumulative).csv", mode='a',
header=not os.path.exists("Prophet Forecasts (Cumulative).csv"), index=False)

        print("Processed account number",i)
    #Where an error occurs, the user is informed about it
    except Exception as e:
        print(f"Error encountered processing account number {i}.")
        ErrorType, ErrorObject, ErrorTraceback = sys.exc_info()

        ErrorFilename = os.path.split(
```

```
            ErrorTraceback.tb_frame.f_code.co_filename
        )[1]

        ErrorMessage = str(e)

        ErrorLineNumber = ErrorTraceback.tb_lineno

        print(f'Exception Type: {ErrorType}')

        print(f'Exception Filename: {ErrorFilename}')

        print(f'Exception Line Number: {ErrorLineNumber}')

        print(f'Exception Message: {ErrorMessage}')
        break
    #Deleting model data to conserve RAM
    finally:
        del Model, Forecast, TrainData, TestData, ForecastingTimeSeries
        gc.collect()

"""**Calculating Error Statistics**"""

#Calculating averaged error statistics
Errors = pd.read_csv("Prophet Errors (Cumulative).csv")
MeanRMSE = Errors["RMSE"].mean()
print("Mean RMSE:",MeanRMSE)
MedianRMSE = Errors["RMSE"].median()
print("Median RMSE:",MedianRMSE)
MeanSI = Errors["SI"].mean()
print("Mean SI:",MeanSI)
MedianSI = Errors["SI"].median()
print("Median SI:",MedianSI)
```

6.2.2.10 Appendix 2.2.10: Prophet Forecasts (Month-by-Month).py (converted from .ipynb)

```
# -*- coding: utf-8 -*-
"""Prophet Forecasts (Month-by-Month).ipynb

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/14ikufrqflxgPtASkRx5fNR1Lw3zkU5ru

**Importing Libraries and Data**
"""

#Importing libraries
import pandas as pd
import statsmodels.api as sm
from sklearn.metrics import root_mean_squared_error
from statsmodels.tsa.stattools import acf
import matplotlib.pyplot as plt
import prophet
import numpy as np
import warnings
import gc
import os
import sys

#Ignoring warnings
warnings.filterwarnings("ignore")

#Reading in data and extracting unique account values
BoostedTransactions = pd.read_csv("Boosted Transaction Dataset.csv")
AccountIDs = BoostedTransactions["AccountId"].values
UniqueAccounts = set(AccountIDs)

#Converting transaction dates into months
BoostedTransactions["TransactionDate"] =
pd.to_datetime(BoostedTransactions["TransactionDate"])
BoostedTransactions["TransactionDate"] =
BoostedTransactions["TransactionDate"].dt.to_period("M").dt.to_timestamp()

#Grouping transactions by account and month and summating each grouping's net
transaction amount
GroupedSumOfTransactionsByAccountAndMonth =
BoostedTransactions.groupby(by=["AccountId",
"TransactionDate"]).agg(["sum"])["Amount"]
print(GroupedSumOfTransactionsByAccountAndMonth)
GroupedSumOfTransactionsByAccountAndMonth =
GroupedSumOfTransactionsByAccountAndMonth.reset_index()

"""**Key Prophet Loop**"""
```

```
#Creating models and forecasting for all accounts
for i in UniqueAccounts:
    print(i)
    try:
        #Selecting only the selected account's time series and ensuring there are no gaps
        ForecastingTimeSeries =
GroupedSumOfTransactionsByAccountAndMonth[GroupedSumOfTransactionsBy
AccountAndMonth["AccountId"] == i][["TransactionDate", "sum"]]

        ForecastingTimeSeries = ForecastingTimeSeries.set_index("TransactionDate")

        ForecastingTimeSeries = ForecastingTimeSeries.resample("MS").sum()

        ForecastingTimeSeries = ForecastingTimeSeries.reset_index()

        #Calculating the cutoff point between training and testing sets
        TrainEnd = int(0.7 * len(ForecastingTimeSeries))

        #Splitting the data into training and testing sets and adjusting column names in
each
        TrainData = ForecastingTimeSeries[:TrainEnd]
        TestData = ForecastingTimeSeries[TrainEnd:]

        TrainData.columns = ["ds", "y"]
        TestData.columns = ["ds", "y"]

        #Declaring and training the model
        Model = prophet.Prophet()
        Model.fit(TrainData)

        #Performing a test forecast
        Forecast = Model.predict(TestData)

        #Calculating error rates and appending them into the errors CSV
        try:
          RMSE = root_mean_squared_error(Forecast["yhat"], TestData["y"])
          SI = abs(RMSE/abs(TestData["y"]).mean())
          AccountError = pd.DataFrame({"AccountID" : [i],
                                       "RMSE" : [RMSE],
                                       "SI" : [SI]})
          AccountError.to_csv("Prophet Errors (Month-by-Month).csv", mode='a',
header=not os.path.exists("Prophet Errors (Month-by-Month).csv"), index=False)
        #Where a ValueError occurs, the loop is broken
        except ValueError:
          print("ValueError encountered")
          break

        #Forecasting the next 12 months
        Future = Model.make_future_dataframe(periods=12, freq="MS")
```

```
        FutureForecast = Model.predict(Future)

        FutureForecastTail = FutureForecast.tail(12)
        FutureForecastValues = FutureForecastTail["yhat"].values

        #Calculating highest amount, lowest amount and difference and appending
them into the forecasts CSV file
        HighestAmount = FutureForecastValues.max()
        LowestAmount = FutureForecastValues.min()

        if np.isnan(HighestAmount) or np.isnan(LowestAmount):
          HighestAmount = 0
          LowestAmount = 0
        else:
          pass

        Difference = HighestAmount - LowestAmount

        ForecastData = pd.DataFrame({"AccountID" : [i],
                                     "Highest Amount" : [HighestAmount],
                                     "Lowest Amount" : [LowestAmount],
                                     "Difference" : [Difference]})

        ForecastData.to_csv("Prophet Forecasts (Month-by-Month).csv", mode='a',
header=not os.path.exists("Prophet Forecasts (Month-by-Month).csv"), index=False)

        print("Processed account number",i)
    #Where an error occurs, the user is informed about it
    except Exception as e:
        print(f"Error encountered processing account number {i}.")
        ErrorType, ErrorObject, ErrorTraceback = sys.exc_info()

        ErrorFilename = os.path.split(
            ErrorTraceback.tb_frame.f_code.co_filename
        )[1]

        ErrorMessage = str(e)

        ErrorLineNumber = ErrorTraceback.tb_lineno

        print(f'Exception Type: {ErrorType}')

        print(f'Exception Filename: {ErrorFilename}')

        print(f'Exception Line Number: {ErrorLineNumber}')

        print(f'Exception Message: {ErrorMessage}')
        break
    #Deleting model data to conserve RAM
    finally:
```

```
    del Model, Forecast, TrainData, TestData, ForecastingTimeSeries
    gc.collect()

"""**Calculating Error Statistics**"""

#Calculating averaged error statistics
Errors = pd.read_csv("Prophet Errors (Month-by-Month).csv")
MeanRMSE = Errors["RMSE"].mean()
print("Mean RMSE:",MeanRMSE)
MedianRMSE = Errors["RMSE"].median()
print("Median RMSE:",MedianRMSE)
MeanSI = Errors["SI"].mean()
print("Mean SI:",MeanSI)
MedianSI = Errors["SI"].median()
print("Median SI:",MedianSI)
```

6.2.2.11 Appendix 2.2.11: Prophet Forecasts on Longer Subset (Cumulative).py (converted from .ipynb)

```
# -*- coding: utf-8 -*-
"""Prophet Forecasts on Longer Subset (Cumulative).ipynb

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/1YSBhYs7CEmmg-BeVwCnf1jeedbFHzmOK

**Importing Libraries and Data**
"""

#Importing libraries
import pandas as pd
import statsmodels.api as sm
from sklearn.metrics import root_mean_squared_error
from statsmodels.tsa.stattools import acf
import matplotlib.pyplot as plt
import prophet
import numpy as np
import warnings
import gc
import os
import sys

#Ignoring warnings
warnings.filterwarnings("ignore")

#Reading in data and extracting unique account values
BoostedTransactions = pd.read_csv("Longer Subset of Transaction Data (14 Months).csv")
AccountIDs = BoostedTransactions["AccountId"].values
UniqueAccounts = set(AccountIDs)

#Converting transaction dates into months
BoostedTransactions["TransactionDate"] = pd.to_datetime(BoostedTransactions["TransactionDate"])
BoostedTransactions["TransactionDate"] = BoostedTransactions["TransactionDate"].dt.to_period("M").dt.to_timestamp()

#Grouping transactions by account and month and summating each grouping's net transaction amount
GroupedSumOfTransactionsByAccountAndMonth = BoostedTransactions.groupby(by=["AccountId", "TransactionDate"]).agg(["sum"])["Amount"]
print(GroupedSumOfTransactionsByAccountAndMonth)

#Creating a list to store all accounts' time series
```

```
DataFrames = []

#Imputing 0 values into the gaps in each account's time series
for i in range(0, len(UniqueAccounts)):
  ResetDataFrame =
GroupedSumOfTransactionsByAccountAndMonth.reset_index()
  TimeSeries = ResetDataFrame[ResetDataFrame["AccountId"] ==
list(UniqueAccounts)[i]]
  TimeSeries = TimeSeries.set_index("TransactionDate")

  TimeSeries = TimeSeries.resample("MS").sum()
  for j in range(0, len(TimeSeries)):
    if TimeSeries.iloc[j]["AccountId"] == 0:
      TimeSeries.at[TimeSeries.index[j], "AccountId"] = list(UniqueAccounts)[i]

  DataFrames.append(TimeSeries)

#Combining all accounts' time series into a new grouped dataset
NewGroupedDataset = pd.concat(DataFrames)
NewGroupedDataset = NewGroupedDataset.reset_index()
NewGroupedDataset = NewGroupedDataset.set_index(["AccountId",
"TransactionDate"])
print(NewGroupedDataset)

#Calculating the cumulative transaction sum over the time period for each account
GroupedCumulativeSumOfTransactionsByAccountAndMonth =
NewGroupedDataset.groupby(level=0).cumsum().reset_index()
print(GroupedCumulativeSumOfTransactionsByAccountAndMonth)

"""**Key Prophet Loop**"""

#Creating models and forecasting for all accounts
for i in UniqueAccounts:
    print(i)
    try:
        #Selecting only the selected account's time series
        ForecastingTimeSeries =
GroupedCumulativeSumOfTransactionsByAccountAndMonth[GroupedCumulative
SumOfTransactionsByAccountAndMonth["AccountId"] == i][["TransactionDate",
"sum"]]

        #Calculating the cutoff between training and testing sets
        TrainEnd = int(0.7 * len(ForecastingTimeSeries))

        #Splitting the data into training and testing sets and adjusting column names in
each
        TrainData = ForecastingTimeSeries[:TrainEnd]
        TestData = ForecastingTimeSeries[TrainEnd:]

        TrainData.columns = ["ds", "y"]
```

```
        TestData.columns = ["ds", "y"]

        #Declaring and training the model
        Model = prophet.Prophet()
        Model.fit(TrainData)

        #Performing a test forecast
        Forecast = Model.predict(TestData)

        #Calculating error rates and appending them into the errors CSV
        try:
          RMSE = root_mean_squared_error(Forecast["yhat"], TestData["y"])
          SI = abs(RMSE/abs(TestData["y"]).mean())
          AccountError = pd.DataFrame({"AccountID" : [i],
                                       "RMSE" : [RMSE],
                                       "SI" : [SI]})
          AccountError.to_csv("Prophet Longer Subset Errors (Cumulative).csv",
mode='a', header=not os.path.exists("Prophet Longer Subset Errors
(Cumulative).csv"), index=False)
        #Where a ValueError occurs, the loop is broken
        except ValueError:
          print("ValueError encountered")
          break

        #Forecasting the next 12 months
        Future = Model.make_future_dataframe(periods=12, freq="MS")
        FutureForecast = Model.predict(Future)

        FutureForecastTail = FutureForecast.tail(12)
        FutureForecastValues = FutureForecastTail["yhat"].values

        #Calculating starting amount, ending amount and net change and appending
them into the forecasts CSV
        NetChange = FutureForecastValues[-1] - FutureForecastValues[0]
        StartingAmount = FutureForecastValues[0]
        EndingAmount = FutureForecastValues[-1]

        ForecastData = pd.DataFrame({"AccountID" : [i],
                                     "Net Change over Forecast Period" : [NetChange],
                                     "Starting Amount" : [StartingAmount],
                                     "Ending Amount" : [EndingAmount]})

        ForecastData.to_csv("Prophet Longer Subset Forecasts (Cumulative).csv",
mode='a', header=not os.path.exists("Prophet Longer Subset Forecasts
(Cumulative).csv"), index=False)

        print("Processed account number",i)
    #Where an error occurs, the user is informed about it
    except Exception as e:
        print(f'Error encountered processing account number {i}.')
```

```
        ErrorType, ErrorObject, ErrorTraceback = sys.exc_info()

        ErrorFilename = os.path.split(
            ErrorTraceback.tb_frame.f_code.co_filename
        )[1]

        ErrorMessage = str(e)

        ErrorLineNumber = ErrorTraceback.tb_lineno

        print(f'Exception Type: {ErrorType}')

        print(f'Exception Filename: {ErrorFilename}')

        print(f'Exception Line Number: {ErrorLineNumber}')

        print(f'Exception Message: {ErrorMessage}')
        break
    #Deleting model data to conserve RAM
    finally:
        del Model, Forecast, TrainData, TestData, ForecastingTimeSeries
        gc.collect()

"""**Calculating Error Statistics**"""

#Calculating averaged error statistics
Errors = pd.read_csv("Prophet Longer Subset Errors (Cumulative).csv")
MeanRMSE = Errors["RMSE"].mean()
print("Mean RMSE:",MeanRMSE)
MedianRMSE = Errors["RMSE"].median()
print("Median RMSE:",MedianRMSE)
MeanSI = Errors["SI"].mean()
print("Mean SI:",MeanSI)
MedianSI = Errors["SI"].median()
print("Median SI:",MedianSI)
```

6.2.2.12 Appendix 2.2.12: Prophet Forecasts on Longer Subset (Month-by-Month).py (converted from .ipynb)

```
# -*- coding: utf-8 -*-
"""Prophet Forecasts on Longer Subset (Month-by-Month).ipynb

Automatically generated by Colab.

Original file is located at

https://colab.research.google.com/drive/1Varld7YOAFpLzxCtvCJxu19qeTMv5SIz

**Importing Libraries and Data**
"""

#Importing libraries
import pandas as pd
import statsmodels.api as sm
from sklearn.metrics import root_mean_squared_error
from statsmodels.tsa.stattools import acf
import matplotlib.pyplot as plt
import prophet
import numpy as np
import warnings
import gc
import os
import sys

#Ignoring warnings
warnings.filterwarnings("ignore")

#Reading in the data and extracting unique account values
BoostedTransactions = pd.read_csv("Longer Subset of Transaction Data (14
Months).csv")
AccountIDs = BoostedTransactions["AccountId"].values
UniqueAccounts = set(AccountIDs)

#Converting transaction dates into months
BoostedTransactions["TransactionDate"] =
pd.to_datetime(BoostedTransactions["TransactionDate"])
BoostedTransactions["TransactionDate"] =
BoostedTransactions["TransactionDate"].dt.to_period("M").dt.to_timestamp()

#Grouping transactions by account and month and summating each grouping's net
transaction amount
GroupedSumOfTransactionsByAccountAndMonth =
BoostedTransactions.groupby(by=["AccountId",
"TransactionDate"]).agg(["sum"])["Amount"]
print(GroupedSumOfTransactionsByAccountAndMonth)
GroupedSumOfTransactionsByAccountAndMonth =
GroupedSumOfTransactionsByAccountAndMonth.reset_index()
```

```
"""**Key Prophet Loop**"""

#Creating models and forecasting for all accounts
for i in UniqueAccounts:
   print(i)
   try:
      #Selecting only the selected amount's time series and ensuring it has no gaps
      ForecastingTimeSeries =
GroupedSumOfTransactionsByAccountAndMonth[GroupedSumOfTransactionsBy
AccountAndMonth["AccountId"] == i][["TransactionDate", "sum"]]

      ForecastingTimeSeries = ForecastingTimeSeries.set_index("TransactionDate")

      ForecastingTimeSeries = ForecastingTimeSeries.resample("MS").sum()

      ForecastingTimeSeries = ForecastingTimeSeries.reset_index()

      #Calculating the cutoff point between training and testing sets
      TrainEnd = int(0.7 * len(ForecastingTimeSeries))

      #Splitting the data into training and testing sets and adjusting column names in
each
      TrainData = ForecastingTimeSeries[:TrainEnd]
      TestData = ForecastingTimeSeries[TrainEnd:]

      TrainData.columns = ["ds", "y"]
      TestData.columns = ["ds", "y"]

      #Declaring and training model
      Model = prophet.Prophet()
      Model.fit(TrainData)

      #Performing a test forecast
      Forecast = Model.predict(TestData)

      #Calculating error rates and appending them to the errors CSV
      try:
        RMSE = root_mean_squared_error(Forecast["yhat"], TestData["y"])
        SI = abs(RMSE/abs(TestData["y"]).mean())
        AccountError = pd.DataFrame({"AccountID" : [i],
                                     "RMSE" : [RMSE],
                                     "SI" : [SI]})
        AccountError.to_csv("Prophet Longer Subset Errors (Month-by-Month).csv",
mode='a', header=not os.path.exists("Prophet Longer Subset Errors (Month-by-
Month).csv"), index=False)
      #Where a ValueError occurs, the loop is broken
      except ValueError:
        print("ValueError encountered")
        break
```

```
        #Forecasting the next 12 months
        Future = Model.make_future_dataframe(periods=12, freq="MS")
        FutureForecast = Model.predict(Future)

        FutureForecastTail = FutureForecast.tail(12)
        FutureForecastValues = FutureForecastTail["yhat"].values

        #Calculating highest amount, lowest amount and difference and appending
them into the forecasts CSV
        HighestAmount = FutureForecastValues.max()
        LowestAmount = FutureForecastValues.min()

        if np.isnan(HighestAmount) or np.isnan(LowestAmount):
          HighestAmount = 0
          LowestAmount = 0
        else:
          pass

        Difference = HighestAmount - LowestAmount

        ForecastData = pd.DataFrame({"AccountID" : [i],
                                     "Highest Amount" : [HighestAmount],
                                     "Lowest Amount" : [LowestAmount],
                                     "Difference" : [Difference]})

        ForecastData.to_csv("Prophet Longer Subset Forecasts (Month-by-
Month).csv", mode='a', header=not os.path.exists("Prophet Longer Subset Forecasts
(Month-by-Month).csv"), index=False)

        print("Processed account number",i)
    #Where an error occurs, the user is informed about it
    except Exception as e:
        print(f'Error encountered processing account number {i}.')
        ErrorType, ErrorObject, ErrorTraceback = sys.exc_info()

        ErrorFilename = os.path.split(
            ErrorTraceback.tb_frame.f_code.co_filename
        )[1]

        ErrorMessage = str(e)

        ErrorLineNumber = ErrorTraceback.tb_lineno

        print(f'Exception Type: {ErrorType}')

        print(f'Exception Filename: {ErrorFilename}')

        print(f'Exception Line Number: {ErrorLineNumber}')
```

```
        print(f'Exception Message: {ErrorMessage}')
        break
    #Deleting model data to conserve RAM
    finally:
        del Model, Forecast, TrainData, TestData, ForecastingTimeSeries
        gc.collect()

"""**Calculating Error Statistics**"""

#Calculating averaged error statistics
Errors = pd.read_csv("Prophet Longer Subset Errors (Month-by-Month).csv")
MeanRMSE = Errors["RMSE"].mean()
print("Mean RMSE:",MeanRMSE)
MedianRMSE = Errors["RMSE"].median()
print("Median RMSE:",MedianRMSE)
MeanSI = Errors["SI"].mean()
print("Mean SI:",MeanSI)
MedianSI = Errors["SI"].median()
print("Median SI:",MedianSI)
```

6.2.2.13 Appendix 2.2.13: SSA Forecasts (Cumulative).py (converted from .ipynb)

```
# -*- coding: utf-8 -*-
"""SSA Forecasts (Cumulative).ipynb

Automatically generated by Colab.

Original file is located at

https://colab.research.google.com/drive/1VdWjpACvWMm4Zu8LS9IsGWu7oCYTylZl

**Importing Libraries and Data**
"""

#Importing libraries
import pandas as pd
import statsmodels.api as sm
from sklearn.metrics import root_mean_squared_error
from statsmodels.tsa.stattools import acf
import matplotlib.pyplot as plt
from mySSA import mySSA
import numpy as np
import warnings
import gc
import os
import sys

#Ignoring warnings
warnings.filterwarnings("ignore")

#Reading in the data and extracting unique account values
BoostedTransactions = pd.read_csv("Boosted Transaction Dataset.csv")
AccountIDs = BoostedTransactions["AccountId"].values
UniqueAccounts = set(AccountIDs)

#Converting transaction dates to months
BoostedTransactions["TransactionDate"] =
pd.to_datetime(BoostedTransactions["TransactionDate"])
BoostedTransactions["TransactionDate"] =
BoostedTransactions["TransactionDate"].dt.to_period("M").dt.to_timestamp()

#Grouping transactions by account and month and summating each grouping's net
transaction amount
GroupedSumOfTransactionsByAccountAndMonth =
BoostedTransactions.groupby(by=["AccountId",
"TransactionDate"]).agg(["sum"])["Amount"]
print(GroupedSumOfTransactionsByAccountAndMonth)

#Creating a list to store all accounts' time series in
```

```
DataFrames = []

#Imputing 0 values into the gaps in each account's time series
for i in range(0, len(UniqueAccounts)):
  ResetDataFrame =
GroupedSumOfTransactionsByAccountAndMonth.reset_index()
  TimeSeries = ResetDataFrame[ResetDataFrame["AccountId"] ==
list(UniqueAccounts)[i]]
  TimeSeries = TimeSeries.set_index("TransactionDate")

  TimeSeries = TimeSeries.resample("MS").sum()
  for j in range(0, len(TimeSeries)):
    if TimeSeries.iloc[j]["AccountId"] == 0:
      TimeSeries.at[TimeSeries.index[j], "AccountId"] = list(UniqueAccounts)[i]

  DataFrames.append(TimeSeries)

#Combining all accounts' time series into a new grouped dataset
NewGroupedDataset = pd.concat(DataFrames)
NewGroupedDataset = NewGroupedDataset.reset_index()
NewGroupedDataset = NewGroupedDataset.set_index(["AccountId",
"TransactionDate"])
print(NewGroupedDataset)

#Calculating the cumulative transaction sum over the time period for each account
GroupedCumulativeSumOfTransactionsByAccountAndMonth =
NewGroupedDataset.groupby(level=0).cumsum().reset_index()
print(GroupedCumulativeSumOfTransactionsByAccountAndMonth)

"""**Key SSA Loop**

This cannot be executed without the mySSA.py Python class in the Colab
environment, which can be found in the code folder.
"""

#Creating models and forecasting for all accounts
for i in UniqueAccounts:
    print(i)
    try:
        #Selecting only the current account's time series
        ForecastingTestTimeSeries =
GroupedCumulativeSumOfTransactionsByAccountAndMonth[GroupedCumulative
SumOfTransactionsByAccountAndMonth["AccountId"] == i][["TransactionDate",
"sum"]]

        ForecastingTestTimeSeries =
ForecastingTestTimeSeries.set_index("TransactionDate")

        #Calculating the cutoff point between training and testing sets
        TrainEnd = int(0.7 * len(ForecastingTestTimeSeries))
```

```
        #Splitting the data into training and testing sets
        TrainData = ForecastingTestTimeSeries[:TrainEnd]
        TestData = ForecastingTestTimeSeries[TrainEnd:]

        #Assigning a window length L based on training data length, with L/2 being
assigned unless L/2 is less than 2
        if len(TrainData) <= 2:
            L = len(TrainData)
        else:
            L = max(2, len(TrainData) // 2)

        #Calculating K and assigning the most significant lag based on significant
autocorrelations, with 0 being assigned if none exist
        K = len(TrainData) - L + 1
        try:
            if K <= 1:
              MostSignificantLag = 0
            else:
              ACFValues = acf(TrainData.iloc[:, 0], nlags=K, fft=False)
              MostSignificantLag = ACFValues[1:].argmax() + 1
              Threshold = 0.2
              if np.isnan(ACFValues).any():
                  MostSignificantLag = 0
              else:
                  MostSignificantLag = ACFValues[1:].argmax() + 1
                  Threshold = 0.2
                  if ACFValues[MostSignificantLag] < Threshold:
                      MostSignificantLag = 0
        except Exception as e:
            MostSignificantLag = 0

        #Creating the SSA model, embedding and decomposing
        if TrainData.empty or TrainData.isnull().all().any():
            print(f"Skipping account {i} due to empty or invalid TrainData")
            #print(TrainData)
            break
        SSA = mySSA(TrainData)

        SSA.embed(embedding_dimension=K,
suspected_frequency=MostSignificantLag)

        SSA.decompose()

        #Getting contributing signals
        Contributions = SSA.view_s_contributions(adjust_scale=True, return_df=True)

        #Creating lists to store stream numbers and their RMSEs and SIs
        StreamNumbers = []
        ErrorsRMSE = []
```

```
    ErrorsSI = []

    #Testing each number of streams from the contributions to see which one
returns the lowest RMSE
    for h in range(1, (len(Contributions)+1)):
        Streams = [j for j in range(h)]
        if hasattr(SSA, 'X_com_hat'):
            del SSA.X_com_hat
        Forecast =
SSA.forecast_recurrent(steps_ahead=(len(ForecastingTestTimeSeries)-TrainEnd),
singular_values=Streams, return_df=True)
        Forecast["Forecast"] = Forecast["Forecast"].clip(lower=-1e6, upper=1e6)
        Forecast["Forecast"] = Forecast["Forecast"].fillna(0)
        try:
            RMSE = root_mean_squared_error(TestData["sum"],
Forecast["Forecast"][TrainEnd:])
            StreamNumbers.append(h)
            ErrorsRMSE.append(RMSE)
            ErrorsSI.append(abs(RMSE/abs(TestData["sum"]).mean()))
        except ValueError as eV:
            print("Bad RMSE value encountered.")
            print(TestData["sum"])
            print(Forecast["Forecast"])
            print(RMSE)
            print(abs(RMSE/abs(TestData["sum"]).mean()))
            ErrorType, ErrorObject, ErrorTraceback = sys.exc_info()

            ErrorFilename = os.path.split(
                ErrorTraceback.tb_frame.f_code.co_filename
            )[1]

            ErrorMessage = str(eV)

            ErrorLineNumber = ErrorTraceback.tb_lineno

            print(f'Exception Type: {ErrorType}')

            print(f'Exception Filename: {ErrorFilename}')

            print(f'Exception Line Number: {ErrorLineNumber}')

            print(f'Exception Message: {ErrorMessage}')
            break

    ErrorData = pd.DataFrame({"Streams" : StreamNumbers,
                "RMSE" : ErrorsRMSE,
                "SI" : ErrorsSI})
    #Taking the number of streams with the lowest RMSE and appending its error
rates into the errors CSV file
    try:
```

```
        IdealNumberOfStreams =
ErrorData.loc[ErrorData["RMSE"].idxmin()]["Streams"]
        LowestRMSE = ErrorData.loc[ErrorData["RMSE"].idxmin()]["RMSE"]
        LowestSI = ErrorData.loc[ErrorData["SI"].idxmin()]["SI"]
        AccountError = pd.DataFrame({"AccountID" : [i],
                                     "RMSE" : [LowestRMSE],
                                     "SI" : [LowestSI]})
        AccountError.to_csv("SSA Errors (Cumulative).csv", mode='a', header=not
os.path.exists("SSA Errors (Cumulative).csv"), index=False)
      except ValueError:
        print("ValueError encountered")
        print(ErrorData)
        break

      #Taking the ideal streams and performing a 12-month forecast based on them
      IdealStreams = [k for k in range(int(IdealNumberOfStreams))]

      if hasattr(SSA, 'X_com_hat'):
         del SSA.X_com_hat

      BestForecast =
SSA.forecast_recurrent(steps_ahead=((len(ForecastingTestTimeSeries)-
TrainEnd)+12), singular_values=IdealStreams, return_df=True)
      #print(BestForecast)

      FutureForecast = BestForecast[-12:]["Forecast"].values
      #print(FutureForecast)

      #Calculating starting amount, ending amount and net change and appending
them into the forecasts CSV file
      NetChange = FutureForecast[-1] - FutureForecast[0]
      StartingAmount = FutureForecast[0]
      EndingAmount = FutureForecast[-1]

      ForecastData = pd.DataFrame({"AccountID" : [i],
                                   "Net Change over Forecast Period" : [NetChange],
                                   "Starting Amount" : [StartingAmount],
                                   "Ending Amount" : [EndingAmount]})

      ForecastData.to_csv("SSA Forecasts (Cumulative).csv", mode='a', header=not
os.path.exists("SSA Forecasts (Cumulative).csv"), index=False)

      print("Processed account number",i)
   #Where an error occurs, the user is informed about it
   except Exception as e:
      print(f"Error encountered processing account number {i}.")
      ErrorType, ErrorObject, ErrorTraceback = sys.exc_info()

      ErrorFilename = os.path.split(
         ErrorTraceback.tb_frame.f_code.co_filename
```

```
        )[1]

        ErrorMessage = str(e)

        ErrorLineNumber = ErrorTraceback.tb_lineno

        print(f'Exception Type: {ErrorType}')

        print(f'Exception Filename: {ErrorFilename}')

        print(f'Exception Line Number: {ErrorLineNumber}')

        print(f'Exception Message: {ErrorMessage}')
        break
    #Deleting model data to conserve RAM
    finally:
        del SSA, Forecast, TrainData, TestData, ForecastingTestTimeSeries
        gc.collect()

print("Code executed successfully.")

#Calculating averaged error statistics
Errors = pd.read_csv("SSA Errors (Cumulative).csv")
MeanRMSE = Errors["RMSE"].mean()
print("Mean RMSE:",MeanRMSE)
MedianRMSE = Errors["RMSE"].median()
print("Median RMSE:",MedianRMSE)
MeanSI = Errors["SI"].mean()
print("Mean SI:",MeanSI)
MedianSI = Errors["SI"].median()
print("Median SI:",MedianSI)
```

6.2.2.14 Appendix 2.2.14: SSA Forecasts (Month-by-Month).py (converted from .ipynb)

```
# -*- coding: utf-8 -*-
"""SSA Forecasts (Month-by-Month).ipynb

Automatically generated by Colab.

Original file is located at

https://colab.research.google.com/drive/1qmj5zGe6LkNrNS4e5WxghjFNOvJTg0N
W

**Importing Libraries and Data**
"""

#Importing libraries
import pandas as pd
import statsmodels.api as sm
from sklearn.metrics import root_mean_squared_error
from statsmodels.tsa.stattools import acf
import matplotlib.pyplot as plt
from mySSA import mySSA
import numpy as np
import warnings
import gc
import os
import sys

#Ignoring warnings
warnings.filterwarnings("ignore")

#Reading in the data and extracting unique account values
BoostedTransactions = pd.read_csv("Boosted Transaction Dataset.csv")
AccountIDs = BoostedTransactions["AccountId"].values
UniqueAccounts = set(AccountIDs)

#Converting transaction dates into months
BoostedTransactions["TransactionDate"] =
pd.to_datetime(BoostedTransactions["TransactionDate"])
BoostedTransactions["TransactionDate"] =
BoostedTransactions["TransactionDate"].dt.to_period("M").dt.to_timestamp()

#Grouping transactions by account and month and summating each grouping's net
transaction amount
GroupedSumOfTransactionsByAccountAndMonth =
BoostedTransactions.groupby(by=["AccountId",
"TransactionDate"]).agg(["sum"])["Amount"]
print(GroupedSumOfTransactionsByAccountAndMonth)
GroupedSumOfTransactionsByAccountAndMonth =
GroupedSumOfTransactionsByAccountAndMonth.reset_index()
```

```
"""**Key SSA Loop**

This will only execute if the mySSA.py class is present within the Colab
environment.
"""

#Creating models and forecasting for all accounts
for i in UniqueAccounts:
  print(i)
  try:
    #Selecting only the selected account's time series and ensuring it has no gaps
    ForecastingTestTimeSeries =
GroupedSumOfTransactionsByAccountAndMonth[GroupedSumOfTransactionsBy
AccountAndMonth["AccountId"] == i][["TransactionDate", "sum"]]

    ForecastingTestTimeSeries =
ForecastingTestTimeSeries.set_index("TransactionDate")

    ForecastingTestTimeSeries =
ForecastingTestTimeSeries.resample("MS").sum()

    #Calculating the cutoff point between training and testing sets
    TrainEnd = int(0.7 * len(ForecastingTestTimeSeries))

    #Splitting the data into training and testing sets
    TrainData = ForecastingTestTimeSeries[:TrainEnd]
    TestData = ForecastingTestTimeSeries[TrainEnd:]

    #Assigning window length L as N/2 unless N/2 is less than 2
    if len(TrainData) <= 2:
      L = len(TrainData)
    else:
      L = max(2, len(TrainData) // 2)

    #Calculating embedding dimension K based on window length L and
determining seasonality based on significant autocorrelations
    #This is set to 0 if none exist
    K = len(TrainData) - L + 1
    try:
      if K <= 1:
        MostSignificantLag = 0
      else:
        ACFValues = acf(TrainData.iloc[:, 0], nlags=K, fft=False)
        MostSignificantLag = ACFValues[1:].argmax() + 1
        Threshold = 0.2
        if np.isnan(ACFValues).any():
          MostSignificantLag = 0
        else:
          MostSignificantLag = ACFValues[1:].argmax() + 1
```

```
            Threshold = 0.2
            if ACFValues[MostSignificantLag] < Threshold:
               MostSignificantLag = 0
      except Exception as e:
         MostSignificantLag = 0

      #Fitting SSA model, embedding and decomposing
      if TrainData.empty or TrainData.isnull().all().any():
         print(f"Skipping account {i} due to empty or invalid TrainData")
         #print(TrainData)
         break
      SSA = mySSA(TrainData)

      SSA.embed(embedding_dimension=K,
suspected_frequency=MostSignificantLag)

      SSA.decompose()

      #Extracting signals and contributions from the SSA
      Contributions = SSA.view_s_contributions(adjust_scale=True, return_df=True)

      #Declaring lists to store stream numbers and error statistics
      StreamNumbers = []
      ErrorsRMSE = []
      ErrorsSI = []

      #Testing stream numbers to find the number with the lowest RMSE
      for h in range(1, (len(Contributions)+1)):
         Streams = [j for j in range(h)]
         if hasattr(SSA, 'X_com_hat'):
            del SSA.X_com_hat
         Forecast =
SSA.forecast_recurrent(steps_ahead=(len(ForecastingTestTimeSeries)-TrainEnd),
singular_values=Streams, return_df=True)
         Forecast["Forecast"] = Forecast["Forecast"].clip(lower=-1e6, upper=1e6)
         Forecast["Forecast"] = Forecast["Forecast"].fillna(0)
         try:
            RMSE = root_mean_squared_error(TestData["sum"],
Forecast["Forecast"][TrainEnd:])
            #print(TestData["sum"])
            StreamNumbers.append(h)
            ErrorsRMSE.append(RMSE)
            ErrorsSI.append(abs(RMSE/abs(TestData["sum"]).mean()))
         except ValueError as eV:
            print("Bad RMSE value encountered.")
            print(TestData["sum"])
            print(Forecast["Forecast"])
            print(RMSE)
            print(abs(RMSE/abs(TestData["sum"]).mean()))
            ErrorType, ErrorObject, ErrorTraceback = sys.exc_info()
```

```
            ErrorFilename = os.path.split(
               ErrorTraceback.tb_frame.f_code.co_filename
            )[1]

            ErrorMessage = str(eV)

            ErrorLineNumber = ErrorTraceback.tb_lineno

            print(f'Exception Type: {ErrorType}')

            print(f'Exception Filename: {ErrorFilename}')

            print(f'Exception Line Number: {ErrorLineNumber}')

            print(f'Exception Message: {ErrorMessage}')
            break

      ErrorData = pd.DataFrame({"Streams" : StreamNumbers,
                  "RMSE" : ErrorsRMSE,
                  "SI" : ErrorsSI})
      #Taking stream number with lowest RMSE and appending its error statistics
into the errors CSV
      try:
        IdealNumberOfStreams =
ErrorData.loc[ErrorData["RMSE"].idxmin()]["Streams"]
        LowestRMSE = ErrorData.loc[ErrorData["RMSE"].idxmin()]["RMSE"]
        LowestSI = ErrorData.loc[ErrorData["SI"].idxmin()]["SI"]
        AccountError = pd.DataFrame({"AccountID" : [i],
                              "RMSE" : [LowestRMSE],
                              "SI" : [LowestSI]})
        AccountError.to_csv("SSA Errors (Month-by-Month).csv", mode='a',
header=not os.path.exists("SSA Errors (Month-by-Month).csv"), index=False)
      except ValueError:
        print("ValueError encountered")
        print(ErrorData)
        break

      #Forecasting the next 12 months with the ideal forecast
      IdealStreams = [k for k in range(int(IdealNumberOfStreams))]

      if hasattr(SSA, 'X_com_hat'):
         del SSA.X_com_hat

      BestForecast =
SSA.forecast_recurrent(steps_ahead=((len(ForecastingTestTimeSeries)-
TrainEnd)+12), singular_values=IdealStreams, return_df=True)
      #print(BestForecast)

      FutureForecast = BestForecast[-12:]["Forecast"].values
```

```
        #Calculating highest amount, lowest amount and difference and appending
them into the forecasts CSV
        HighestAmount = FutureForecast.max()
        LowestAmount = FutureForecast.min()

        if np.isnan(HighestAmount) or np.isnan(LowestAmount):
          HighestAmount = 0
          LowestAmount = 0
        else:
          pass

        Difference = HighestAmount - LowestAmount

        ForecastData = pd.DataFrame({"AccountID" : [i],
                                     "Highest Amount" : [HighestAmount],
                                     "Lowest Amount" : [LowestAmount],
                                     "Difference" : [Difference]})

        ForecastData.to_csv("SSA Forecasts (Month-by-Month).csv", mode='a',
header=not os.path.exists("SSA Forecasts (Month-by-Month).csv"), index=False)

        print("Processed account number",i)
    #Where an error occurs, the user is informed about it
    except Exception as e:
        print(f'Error encountered processing account number {i}.")
        ErrorType, ErrorObject, ErrorTraceback = sys.exc_info()

        ErrorFilename = os.path.split(
            ErrorTraceback.tb_frame.f_code.co_filename
        )[1]

        ErrorMessage = str(e)

        ErrorLineNumber = ErrorTraceback.tb_lineno

        print(f'Exception Type: {ErrorType}')

        print(f'Exception Filename: {ErrorFilename}')

        print(f'Exception Line Number: {ErrorLineNumber}')

        print(f'Exception Message: {ErrorMessage}')
        break
    #Deleting model data to conserve RAM
    finally:
        del SSA, Forecast, TrainData, TestData, ForecastingTestTimeSeries
        gc.collect()

print("Code executed successfully.")
```

```
#Calculating averaged error statistics
Errors = pd.read_csv("SSA Errors (Month-by-Month).csv")
MeanRMSE = Errors["RMSE"].mean()
print("Mean RMSE:",MeanRMSE)
MedianRMSE = Errors["RMSE"].median()
print("Median RMSE:",MedianRMSE)
MeanSI = Errors["SI"].mean()
print("Mean SI:",MeanSI)
MedianSI = Errors["SI"].median()
print("Median SI:",MedianSI)
```

6.2.2.15 Appendix 2.2.15: SSA Forecasts on Longer Subset (Cumulative).py (converted from .ipynb)

```
# -*- coding: utf-8 -*-
"""SSA Forecasts on Longer Subset (Cumulative).ipynb

Automatically generated by Colab.

Original file is located at

https://colab.research.google.com/drive/1BQV_mMQ2EQcsvX6WBUBBM3KuDbv2VPND

**Importing Libraries and Data**
"""

#Importing libraries
import pandas as pd
import statsmodels.api as sm
from sklearn.metrics import root_mean_squared_error
from statsmodels.tsa.stattools import acf
import matplotlib.pyplot as plt
from mySSA import mySSA
import numpy as np
import warnings
import gc
import os
import sys

#Ignoring warnings
warnings.filterwarnings("ignore")

#Reading in the data and extracting unique account values
BoostedTransactions = pd.read_csv("Longer Subset of Transaction Data (14 Months).csv")
AccountIDs = BoostedTransactions["AccountId"].values
UniqueAccounts = set(AccountIDs)

#Converting transaction dates into months
BoostedTransactions["TransactionDate"] = pd.to_datetime(BoostedTransactions["TransactionDate"])
BoostedTransactions["TransactionDate"] = BoostedTransactions["TransactionDate"].dt.to_period("M").dt.to_timestamp()

#Grouping transactions by account and month and summating each grouping's net transaction amount
GroupedSumOfTransactionsByAccountAndMonth = BoostedTransactions.groupby(by=["AccountId", "TransactionDate"]).agg(["sum"])["Amount"]
print(GroupedSumOfTransactionsByAccountAndMonth)
```

```
#Creating a list to store each account's time series in
DataFrames = []

#Imputing 0 values into the gaps in each account's time series
for i in range(0, len(UniqueAccounts)):
  ResetDataFrame =
GroupedSumOfTransactionsByAccountAndMonth.reset_index()
  TimeSeries = ResetDataFrame[ResetDataFrame["AccountId"] ==
list(UniqueAccounts)[i]]
  TimeSeries = TimeSeries.set_index("TransactionDate")

  TimeSeries = TimeSeries.resample("MS").sum()
  for j in range(0, len(TimeSeries)):
    if TimeSeries.iloc[j]["AccountId"] == 0:
      TimeSeries.at[TimeSeries.index[j], "AccountId"] = list(UniqueAccounts)[i]

  DataFrames.append(TimeSeries)

#Combining all accounts' time series into a new grouped dataset
NewGroupedDataset = pd.concat(DataFrames)
NewGroupedDataset = NewGroupedDataset.reset_index()
NewGroupedDataset = NewGroupedDataset.set_index(["AccountId",
"TransactionDate"])
print(NewGroupedDataset)

#Calculating the cumulative transaction sum over the time period for each account
GroupedCumulativeSumOfTransactionsByAccountAndMonth =
NewGroupedDataset.groupby(level=0).cumsum().reset_index()
print(GroupedCumulativeSumOfTransactionsByAccountAndMonth)

"""**Key SSA Loop**

This will only execute if the mySSA.py Python class is imported into the Colab
environment.
"""

#Creating models and forecasting for all accounts
for i in UniqueAccounts:
    print(i)
    try:
        #Selecting only the current account's time series
        ForecastingTestTimeSeries =
GroupedCumulativeSumOfTransactionsByAccountAndMonth[GroupedCumulative
SumOfTransactionsByAccountAndMonth["AccountId"] == i][["TransactionDate",
"sum"]]

        ForecastingTestTimeSeries =
ForecastingTestTimeSeries.set_index("TransactionDate")

        #Calculating the cutoff point between training and testing sets
```

```
        TrainEnd = int(0.7 * len(ForecastingTestTimeSeries))

        #Splitting the dataset into training and testing sets
        TrainData = ForecastingTestTimeSeries[:TrainEnd]
        TestData = ForecastingTestTimeSeries[TrainEnd:]

        #Setting the window length L to N/2 unless N/2 is less than 2
        if len(TrainData) <= 2:
            L = len(TrainData)
        else:
            L = max(2, len(TrainData) // 2)

        #Calculating K based on L and selecting the suspected frequency based on
significant autocorrelations
        #It is set as 0 if none are detected
        K = len(TrainData) - L + 1
        try:
            if K <= 1:
              MostSignificantLag = 0
            else:
              ACFValues = acf(TrainData.iloc[:, 0], nlags=K, fft=False)
              MostSignificantLag = ACFValues[1:].argmax() + 1
              Threshold = 0.2
              if np.isnan(ACFValues).any():
                 MostSignificantLag = 0
              else:
                 MostSignificantLag = ACFValues[1:].argmax() + 1
                 Threshold = 0.2
                 if ACFValues[MostSignificantLag] < Threshold:
                    MostSignificantLag = 0
        except Exception as e:
            MostSignificantLag = 0

        #Training an SSA model, embedding and decomposing
        if TrainData.empty or TrainData.isnull().all().any():
            print(f"Skipping account {i} due to empty or invalid TrainData")
            #print(TrainData)
            break
        SSA = mySSA(TrainData)

        SSA.embed(embedding_dimension=K,
suspected_frequency=MostSignificantLag)

        SSA.decompose()

        #Analysing signal contributions
        Contributions = SSA.view_s_contributions(adjust_scale=True, return_df=True)

        #Creating lists to store stream numbers and error statistics
        StreamNumbers = []
```

```
    ErrorsRMSE = []
    ErrorsSI = []

    #Trying all potential stream numbers to see which one returns the lowest RMSE
    for h in range(1, (len(Contributions)+1)):
        Streams = [j for j in range(h)]
        if hasattr(SSA, 'X_com_hat'):
            del SSA.X_com_hat
        Forecast =
SSA.forecast_recurrent(steps_ahead=(len(ForecastingTestTimeSeries)-TrainEnd),
singular_values=Streams, return_df=True)
        Forecast["Forecast"] = Forecast["Forecast"].clip(lower=-1e6, upper=1e6)
        Forecast["Forecast"] = Forecast["Forecast"].fillna(0)
        try:
            RMSE = root_mean_squared_error(TestData["sum"],
Forecast["Forecast"][TrainEnd:])
            StreamNumbers.append(h)
            ErrorsRMSE.append(RMSE)
            ErrorsSI.append(abs(RMSE/abs(TestData["sum"]).mean()))
        except ValueError as eV:
            print("Bad RMSE value encountered.")
            print(TestData["sum"])
            print(Forecast["Forecast"])
            print(RMSE)
            print(abs(RMSE/abs(TestData["sum"]).mean()))
            ErrorType, ErrorObject, ErrorTraceback = sys.exc_info()

            ErrorFilename = os.path.split(
                ErrorTraceback.tb_frame.f_code.co_filename
            )[1]

            ErrorMessage = str(eV)

            ErrorLineNumber = ErrorTraceback.tb_lineno

            print(f'Exception Type: {ErrorType}')

            print(f'Exception Filename: {ErrorFilename}')

            print(f'Exception Line Number: {ErrorLineNumber}')

            print(f'Exception Message: {ErrorMessage}')
            break

    ErrorData = pd.DataFrame({"Streams" : StreamNumbers,
                "RMSE" : ErrorsRMSE,
                "SI" : ErrorsSI})
    #Taking the stream number with the lowest RMSE and appending its error
statistics into the errors CSV
    try:
```

```
        IdealNumberOfStreams =
ErrorData.loc[ErrorData["RMSE"].idxmin()]["Streams"]
        #print(IdealNumberOfStreams)
        LowestRMSE = ErrorData.loc[ErrorData["RMSE"].idxmin()]["RMSE"]
        LowestSI = ErrorData.loc[ErrorData["SI"].idxmin()]["SI"]
        AccountError = pd.DataFrame({"AccountID" : [i],
                                     "RMSE" : [LowestRMSE],
                                     "SI" : [LowestSI]})
        AccountError.to_csv("SSA Longer Subset Errors (Cumulative).csv", mode='a',
header=not os.path.exists("SSA Longer Subset Errors (Cumulative).csv"),
index=False)
      except ValueError:
        print("ValueError encountered")
        print(ErrorData)
        break

      #Forecasting the next 12 months using the chosen forecast
      IdealStreams = [k for k in range(int(IdealNumberOfStreams))]

      if hasattr(SSA, 'X_com_hat'):
         del SSA.X_com_hat

      BestForecast =
SSA.forecast_recurrent(steps_ahead=((len(ForecastingTestTimeSeries)-
TrainEnd)+12), singular_values=IdealStreams, return_df=True)

      FutureForecast = BestForecast[-12:]["Forecast"].values

      #Calculating starting amount, ending amount and net change and appending
them into the forecasts CSV file
      NetChange = FutureForecast[-1] - FutureForecast[0]
      StartingAmount = FutureForecast[0]
      EndingAmount = FutureForecast[-1]

      ForecastData = pd.DataFrame({"AccountID" : [i],
                                   "Net Change over Forecast Period" : [NetChange],
                                   "Starting Amount" : [StartingAmount],
                                   "Ending Amount" : [EndingAmount]})

      ForecastData.to_csv("SSA Longer Subset Forecasts (Cumulative).csv",
mode='a', header=not os.path.exists("SSA Longer Subset Forecasts
(Cumulative).csv"), index=False)

      print("Processed account number",i)
   #Where an error is returned, the user is informed about it
   except Exception as e:
      print(f"Error encountered processing account number {i}.")
      ErrorType, ErrorObject, ErrorTraceback = sys.exc_info()

      ErrorFilename = os.path.split(
```

```
            ErrorTraceback.tb_frame.f_code.co_filename
        )[1]

        ErrorMessage = str(e)

        ErrorLineNumber = ErrorTraceback.tb_lineno

        print(f'Exception Type: {ErrorType}')

        print(f'Exception Filename: {ErrorFilename}')

        print(f'Exception Line Number: {ErrorLineNumber}')

        print(f'Exception Message: {ErrorMessage}')
        break
    #Deleting model data to conserve RAM
    finally:
        del SSA, Forecast, TrainData, TestData, ForecastingTestTimeSeries
        gc.collect()

print("Code executed successfully.")

#Calculating averaged error statistics
Errors = pd.read_csv("SSA Errors.csv")
MeanRMSE = Errors["RMSE"].mean()
print("Mean RMSE:",MeanRMSE)
MedianRMSE = Errors["RMSE"].median()
print("Median RMSE:",MedianRMSE)
MeanSI = Errors["SI"].mean()
print("Mean SI:",MeanSI)
MedianSI = Errors["SI"].median()
print("Median SI:",MedianSI)
```

6.2.2.16 Appendix 2.2.16: SSA Forecasts on Longer Subset (Month-by-Month).py (converted from .ipynb)

```
# -*- coding: utf-8 -*-
"""SSA Forecasts on Longer Subset (Month-by-Month).ipynb

Automatically generated by Colab.

Original file is located at

https://colab.research.google.com/drive/1guGSA8WktKBy5rwFjR7ekhiwCgrBZHZ
1

**Importing Libraries and Data**
"""

#Importing libraries
import pandas as pd
import statsmodels.api as sm
from sklearn.metrics import root_mean_squared_error
from statsmodels.tsa.stattools import acf
import matplotlib.pyplot as plt
from mySSA import mySSA
import numpy as np
import warnings
import gc
import os
import sys

#Ignoring warnings
warnings.filterwarnings("ignore")

#Reading in the data and extracting unique account values
BoostedTransactions = pd.read_csv("Longer Subset of Transaction Data (14
Months).csv")
AccountIDs = BoostedTransactions["AccountId"].values
UniqueAccounts = set(AccountIDs)

#Converting transaction dates into months
BoostedTransactions["TransactionDate"] =
pd.to_datetime(BoostedTransactions["TransactionDate"])
BoostedTransactions["TransactionDate"] =
BoostedTransactions["TransactionDate"].dt.to_period("M").dt.to_timestamp()

#Grouping transactions by account and month and summating each grouping's net
transaction amount
GroupedSumOfTransactionsByAccountAndMonth =
BoostedTransactions.groupby(by=["AccountId",
"TransactionDate"]).agg(["sum"])["Amount"]
print(GroupedSumOfTransactionsByAccountAndMonth)
```

```
GroupedSumOfTransactionsByAccountAndMonth =
GroupedSumOfTransactionsByAccountAndMonth.reset_index()

"""**Key SSA Loop**

This will only execute if the mySSA.py Python class is imported into the Colab
environment.
"""

#Creating models and forecasting for all accounts
for i in UniqueAccounts:
   print(i)
   try:
      #Selecting only the current account's time series and ensuring there are no gaps
      ForecastingTestTimeSeries =
GroupedSumOfTransactionsByAccountAndMonth[GroupedSumOfTransactionsBy
AccountAndMonth["AccountId"] == i][["TransactionDate", "sum"]]

      ForecastingTestTimeSeries =
ForecastingTestTimeSeries.set_index("TransactionDate")

      ForecastingTestTimeSeries =
ForecastingTestTimeSeries.resample("MS").sum()

      #Calculating the cutoff point between training and testing sets
      TrainEnd = int(0.7 * len(ForecastingTestTimeSeries))

      #Splitting the data into training and testing sets
      TrainData = ForecastingTestTimeSeries[:TrainEnd]
      TestData = ForecastingTestTimeSeries[TrainEnd:]

      #Setting window length L to N/2 unless N/2 is less than 2
      if len(TrainData) <= 2:
         L = len(TrainData)
      else:
         L = max(2, len(TrainData) // 2)

      #Calculating embedding dimension K according to L and setting suspected
frequency based on significant autocorrelations
      #If none are found, it is set to 0
      K = len(TrainData) - L + 1
      try:
         if K <= 1:
           MostSignificantLag = 0
         else:
           ACFValues = acf(TrainData.iloc[:, 0], nlags=K, fft=False)
           MostSignificantLag = ACFValues[1:].argmax() + 1
           Threshold = 0.2
           if np.isnan(ACFValues).any():
              MostSignificantLag = 0
```

```
            else:
                MostSignificantLag = ACFValues[1:].argmax() + 1
                Threshold = 0.2
                if ACFValues[MostSignificantLag] < Threshold:
                    MostSignificantLag = 0
        except Exception as e:
            MostSignificantLag = 0

        #Creating the SSA model, embedding and decomposing
        if TrainData.empty or TrainData.isnull().all().any():
            print(f"Skipping account {i} due to empty or invalid TrainData")
            break
        SSA = mySSA(TrainData)

        SSA.embed(embedding_dimension=K,
suspected_frequency=MostSignificantLag)

        SSA.decompose()

        #Analysing signal contributions
        Contributions = SSA.view_s_contributions(adjust_scale=True, return_df=True)

        #Creating lists to store stream numbers and error statistics
        StreamNumbers = []
        ErrorsRMSE = []
        ErrorsSI = []

        #Testing different stream numbers to see which one is best
        for h in range(1, (len(Contributions)+1)):
            Streams = [j for j in range(h)]
            if hasattr(SSA, 'X_com_hat'):
                del SSA.X_com_hat
            Forecast =
SSA.forecast_recurrent(steps_ahead=(len(ForecastingTestTimeSeries)-TrainEnd),
singular_values=Streams, return_df=True)
            Forecast["Forecast"] = Forecast["Forecast"].clip(lower=-1e6, upper=1e6)
            Forecast["Forecast"] = Forecast["Forecast"].fillna(0)
            try:
                RMSE = root_mean_squared_error(TestData["sum"],
Forecast["Forecast"][TrainEnd:])
                StreamNumbers.append(h)
                ErrorsRMSE.append(RMSE)
                ErrorsSI.append(abs(RMSE/abs(TestData["sum"]).mean()))
            except ValueError as eV:
                print("Bad RMSE value encountered.")
                print(TestData["sum"])
                print(Forecast["Forecast"])
                print(RMSE)
                print(abs(RMSE/abs(TestData["sum"]).mean()))
                ErrorType, ErrorObject, ErrorTraceback = sys.exc_info()
```

```
            ErrorFilename = os.path.split(
               ErrorTraceback.tb_frame.f_code.co_filename
            )[1]

            ErrorMessage = str(eV)

            ErrorLineNumber = ErrorTraceback.tb_lineno

            print(f'Exception Type: {ErrorType}')

            print(f'Exception Filename: {ErrorFilename}')

            print(f'Exception Line Number: {ErrorLineNumber}')

            print(f'Exception Message: {ErrorMessage}')
            break

      ErrorData = pd.DataFrame({"Streams" : StreamNumbers,
                  "RMSE" : ErrorsRMSE,
                  "SI" : ErrorsSI})
      #Taking the stream number with the lowest RMSE and appending its error
statistics into the errors CSV
      try:
        IdealNumberOfStreams =
ErrorData.loc[ErrorData["RMSE"].idxmin()]["Streams"]
        LowestRMSE = ErrorData.loc[ErrorData["RMSE"].idxmin()]["RMSE"]
        LowestSI = ErrorData.loc[ErrorData["SI"].idxmin()]["SI"]
        AccountError = pd.DataFrame({"AccountID" : [i],
                              "RMSE" : [LowestRMSE],
                              "SI" : [LowestSI]})
        AccountError.to_csv("SSA Longer Subset Errors (Month-by-Month).csv",
mode='a', header=not os.path.exists("SSA Longer Subset Errors (Month-by-
Month).csv"), index=False)
      except ValueError:
        print("ValueError encountered")
        print(ErrorData)
        break

      #Forecasting the next 12 months using the ideal forecast
      IdealStreams = [k for k in range(int(IdealNumberOfStreams))]

      if hasattr(SSA, 'X_com_hat'):
         del SSA.X_com_hat

      BestForecast =
SSA.forecast_recurrent(steps_ahead=((len(ForecastingTestTimeSeries)-
TrainEnd)+12), singular_values=IdealStreams, return_df=True)

      FutureForecast = BestForecast[-12:]["Forecast"].values
```

```
        #Calculating highest amount, lowest amount and difference and appending
them into the forecasts CSV file
        HighestAmount = FutureForecast.max()
        LowestAmount = FutureForecast.min()

        if np.isnan(HighestAmount) or np.isnan(LowestAmount):
          HighestAmount = 0
          LowestAmount = 0
        else:
          pass

        Difference = HighestAmount - LowestAmount

        ForecastData = pd.DataFrame({"AccountID" : [i],
                                     "Highest Amount" : [HighestAmount],
                                     "Lowest Amount" : [LowestAmount],
                                     "Difference" : [Difference]})

        ForecastData.to_csv("SSA Longer Subset Forecasts (Month-by-Month).csv",
mode='a', header=not os.path.exists("SSA Longer Subset Forecasts (Month-by-
Month).csv"), index=False)

        print("Processed account number",i)
    #Where an error occurs, the user is informed about it
    except Exception as e:
        print(f"Error encountered processing account number {i}.")
        ErrorType, ErrorObject, ErrorTraceback = sys.exc_info()

        ErrorFilename = os.path.split(
            ErrorTraceback.tb_frame.f_code.co_filename
        )[1]

        ErrorMessage = str(e)

        ErrorLineNumber = ErrorTraceback.tb_lineno

        print(f'Exception Type: {ErrorType}')

        print(f'Exception Filename: {ErrorFilename}')

        print(f'Exception Line Number: {ErrorLineNumber}')

        print(f'Exception Message: {ErrorMessage}')
        break
    #Deleting model data to conserve RAM
    finally:
        del SSA, Forecast, TrainData, TestData, ForecastingTestTimeSeries
        gc.collect()
```

```
print("Code executed successfully.")

#Calculating averaged error statistics
Errors = pd.read_csv("SSA Longer Subset Errors (Month-by-Month).csv")
MeanRMSE = Errors["RMSE"].mean()
print("Mean RMSE:",MeanRMSE)
MedianRMSE = Errors["RMSE"].median()
print("Median RMSE:",MedianRMSE)
MeanSI = Errors["SI"].mean()
print("Mean SI:",MeanSI)
MedianSI = Errors["SI"].median()
print("Median SI:",MedianSI)
```