Masteroppgave

NTNU
Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for datateknologi og informatikk

Alexander Meldal Andersen

Sequence Input Aggregation

Masteroppgave i Informatikk
Veileder: Jon Atle Gulla
Medveileder: Benjamin Kille
Juni 2022

NTNU
Norwegian University of
Science and Technology

Alexander Meldal Andersen

Sequence Input Aggregation

Masteroppgave i Informatikk
Veileder: Jon Atle Gulla
Medveileder: Benjamin Kille
Juni 2022

Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for datateknologi og informatikk

NTNU
Norwegian University of
Science and Technology

Abstract

Sequential input aggregation is the task of condensing a, potentially very large, sequential dataset into simpler features. In the industry, particularly finance and banking, sequential data is common, making this a broadly relevant task. Due to the dynamic nature of transactions, and the way people spend and earn money, an automated way of defining such features could potentially save a lot of manual labor.

We propose two kinds of automatic methods of recognizing complex patterns in sequences of transactions. The first method is fitting autoencoder recurrent neural networks to learn a fixed number of features, and the second is applying Fourier analysis. Both of these can be applied to large amounts of unlabeled, transactional data for effective aggregation of said data.

Using real world data, we evaluate our aggregation methods, by using the aggregates as inputs for tree-based supervised learners. The automatic methods show some promise in recognizing features, with the autoencoders seemingly capturing more than the Fourier analysis, but the automatic methods could not yield any improvement when compared to manual feature engineering.

Sammendrag

Aggregering av sekvensiell input handler om å redusere et, potensielt veldig stort, sekvensielt datasett til enklere variabler. I industrien, og særlig innen bank og finans, er sekvensielle data veldig vanlige, som gjør dette til en høyst relevant oppgave. Grunnet at sekvensene av transaksjoner utvikles dynamisk over tid, i takt med at folk endrer hvordan de tjener og bruker penger, så kan en automatisk måte å definere slike aggregater potensielt spare mye manuelt arbeid.

Vi foreslår to typer automatiske metoder for å gjenkjenne komplekse mønstre i sekvenser av transaksjoner. Den første innebærer å trene et autoenkoder-nevralt nettverk til å komprimere en sekvens til et gitt antall variabler, og den andre er å anvende Fourier-analyse. Begge metodene kan anvendes på store umerkede sekvensielle datamengder, for å effektivt kunne aggregere dem.

Vi bruker data fra den virkelige verden for å evaluere metodene våre, ved å bruke aggregatene vi lager som input-variabler for trebaserte maskinlæringsalgoritmer på merket data. De automatiske metodene viser noe potensiale i å finne kjennetegnende informasjon, og autoenkoderen ser ut til å fange opp mer informasjon enn Fourier-analyse gjør, men de automatiske metodene klarer ikke å forbedre resultatene man får, når de blir sammenlignet med manuelt definerte aggregater.

Preface

This project has been carried out at the Norwegian University of Science and Technology (NTNU) during the spring semester 2022. It is part of the course *IT3920 - Master Thesis for MSIT*, and was conducted in collaboration between NTNU, Norwegian Research Center for AI Innovation (NorwAI), and Sparebank1 SMN.

Special thanks go to the supervisors for the project, Jon Atle Gulla and Benjamin Kille; without them, none of this would have been possible. Thanks also to Lars Ivar Hagfors at Sparebank1 SMN for the assistance he provided in the implementation of the project.

Table of Contents

List of Figures

List of Tables

1 Introduction

This section will briefly present the motivation, project description and research questions upon which this project is based, as well as an outline for the report.

1.1 Motivation

In the world we live in, many phenomena can be described as sequential data — everything from the steps you take while on a walk to the meals you make, and — as will be the focal point in this process — financial transactions.

Transactions are full of data — a single transaction contains information not only about the amount spent, but also how it was spent, where it is being transferred to, and what category, if applicable, the transaction falls under (e.g. entertainment or food). Transactional data is highly granular — typically, it is as finely grained as economic data can be, as it contains every single piece that can be used in more composite kinds of data. They can be considered an atomic unit of bank history, with each transaction being related to a single purchase.

While sequential data like this is full of information, it is often impractical for machine learning models. Many machine learning architectures — including, but not limited to linear regression, k-nearest-neighbor approaches, or random forest approaches — require a constant number of variables in order to predict. Often, this means that sequential data is either overlooked or aggregated manually — a bank might, for example, use sums or averages over the last week, month or year.

However, most manually defined aggregates are imperfect. They do not capture much nuance in the sequences they attempt to summarize. They could, for example, compute how much was used on video games during the last year, month and week — but they would not, for example, capture that most of it is spent during an annual sale. Such patterns can be very important, particularly when it comes to predicting future spending.

Of course, any such example of patterns can be added manually — however, this would result in an unfeasibly large set of features. Additionally, with a constantly changing world, more and more ways of aggregating the transactions would need to be proposed over time, demanding constantly investing time and money into keeping their features and corresponding models up to date. Therefore, an automated way of defining and computing aggregates would be both useful and valuable.

1.2 Project Description

The project task is to research automated ways to create useful aggregations of transaction data for machine learning models, based on real data. The produced aggregates do not need to be easily explainable for humans, but they should be useful as input variables for machine learning models.

We define a transaction t as a feature vector $t \in \mathbb{R}^p$, and a transaction history h of length N as an ordered series of transactions $h = \{t_i | i = 1, 2, ..., N, t_i \in \mathbb{R}^p\}$. Let H be the set of all histories. An aggregation is a function $f : H \longrightarrow \mathbb{R}^k$, where k is a positive integer, i.e. it maps a transaction history of arbitrary length to a fixed number (k) of features.

Given a set of existing features X, we will define our aggregation performance in terms of model improvement. Training one model m_1 using only X, and training another model m_2 on $X \cup f(h)$, as shown in Figure 1. Each of these models are trained to optimize some loss function, e.g. mean squared error, or cross-entropy. We use new data, unknown to both of the models, to compute these loss functions l_1 and l_2, for m_1 and m_2 respectively. We will then define the aggregation performance as

$$\text{score} = l_1 - l_2 \tag{1}$$

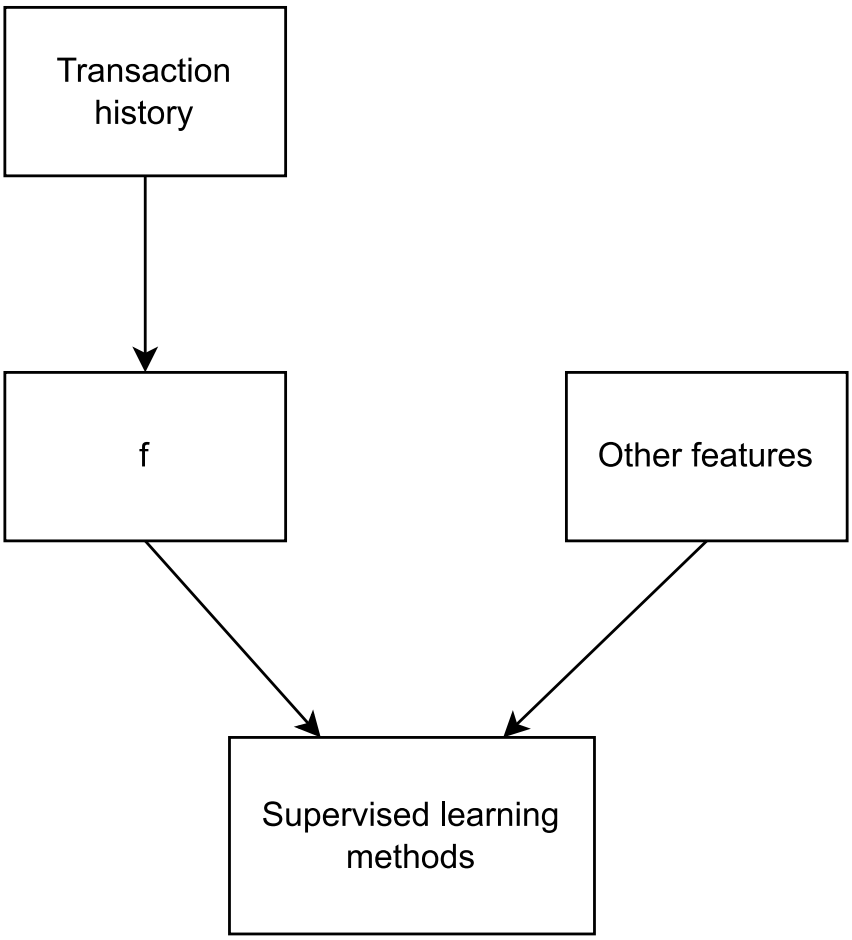


Figure 1: Proposed use of an aggregation function f(h). The aggregated features are used alongside existing features in supervised learning methods.

It is important that the data used for measuring performance is not part of the training set, to avoid overfitting, which would always place m_2 at an advantage, regardless of whether there were any actual patterns recognized in the data.

1.3 Goal and Research Questions

The goal of the project is to produce more informative input variables for Sparebank1 SMN's machine learning models, and to create a framework in which such a set of variables can automatically be redefined and recomputed in the future.

The success of the project will be measured as a supervised learning problem, with labeled data provided by the bank. The exact success metrics will depend on the type of prediction problem, primarily keeping to maximum likelihood estimators - minimizing mean squared error for regression, and cross-entropy for classification.

The project focuses on the following questions:

Research Question 1 *What is the state of the art in aggregation of transactional data?*

Research Question 2 *How well can a sequence autoencoder produce valuable input variables for Sparebank1 SMN's machine learning models?*

Research Question 3 *How can more traditional methods be used to produce inputs, and how do they compare to an autoencoder approach?*

1.4 Outline

This section has presented the project motivation, and the task which the project will attempt to solve. Section 2 provides the theoretical background on which the project is based. In Section 3, several previous works are described, all of which bear key similarities to this project. Then, Section 4 describes the data that is used, and Section 5 covers what methods will be applied to said data, in terms of processing the data and evaluating the proposed methods. Section 6 will describe the results that our experiments yielded. Finally, Section 7 covers some key discussion points before the conclusion in Section 8, in which the key findings of the project are summarized, and possible further works are proposed.

2 Background Theory

This section will lay the foundations on which this thesis is based. This includes feature engineering and selection, representation learning, various supervised learning methods, and Fourier analysis.

2.1 Feature Selection and Engineering

When applying statistical and machine learning methods to a problem, either for prediction or for inference, the model in question will always try to extract knowledge from the data itself. As such, no model will be better than the data on which it is based. It is common to apply statistical methods and domain knowledge, to ensure that the data used is meaningful and adds to whichever solution you are trying to create.

2.1.1 Feature Selection

When working with high-dimensional data, i.e. data with many features, not all features have the same value as predictors in a machine learning context, or any predictive value at all. Using methods for subset selection, features with low or no impact can be discarded, making machine learning upon the data yield better, more stable results.

Selecting the optimal subset of p features is not trivial, and requires to train all 2^p possible subsets. For large values of p, this rapidly becomes computationally infeasible. Furthermore, there is a statistical issue: a very large search space vastly increases the probability of finding a subset that works very well on the training (or validation) set, but performs poorly on new data. For these reasons, it is more common to use stepwise selection methods for subset selection.

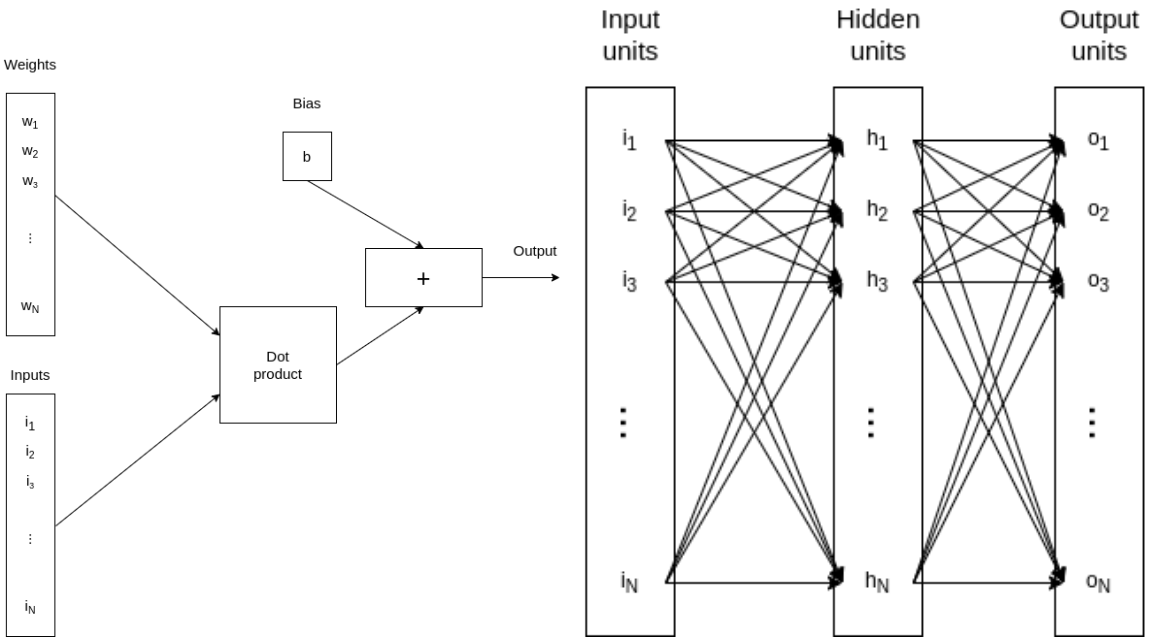
There are three kinds of stepwise selection. Forwards stepwise selection begins with a model with no parameters, and then adds one feature at a time until all predictors are in the model. In each step, it adds the feature that yields the greatest additional improvement when added to the model. Then, select the best of the $p + 1$ models, using cross-validated performance metrics. Backwards stepwise selection is similar, but begins with a model with all predictors, and removes them one at a time instead, removing the predictor that yields the lowest decrease in training performance when removed. The third alternative is using a hybrid solution. Hybrid approaches start similar to forwards stepwise selection, but each time a variable is added, they may also remove any variables that no longer improve the model fit significantly. (James et al., 2014)

2.1.2 Feature Engineering

Feature engineering is, in its essence, the combination of feature selection and domain knowledge. Knowledge about what rules and forces define the domain in which you work, it is likely to include some patterns. If this is the case, knowing about such patterns allows you to construct good features. For example, when modeling house prices, a person with domain knowledge might suggest that distance to the closest city or bus stop are important for most buyers. If a feature is constructed from specific domain knowledge, it should also be considered more important than the statistics alone might indicate - the threshold to include them is lower if you have good reason to believe it is an important feature. (Ramasubramanian and Singh, 2017)

2.1.3 Representation Learning

In some cases, good features can be difficult to define, either because domain is very complex, or because the rules governing the system are not well-known. When features are difficult to define, one possible approach is to apply machine learning methods not only to learn the appropriate function between input features and target features, but also the input features themselves. This



(a) A neural network in its simplest form: a single layer, with no hidden units and one output unit

(b) A neural network with one hidden layer. The outputs from the hidden layer are the inputs for the final (output) layer.

Figure 2: The structure of a neural network. The input units are multiplied by the weights and passed through an activation function to produce output units.

process is known as representation learning. A classic example of representation learning is the autoencoder, described in Section 2.3.

2.2 Neural networks

Neural networks are a type of machine learning models following similar, layered architectures. Each layer has a number of input- and output units, computed by weights. Specifically, the input units are multiplied by a set of weights, added to the bias, then passed through some (differentiable) activation function, i.e.

$$o = \sigma(w \odot i + b),$$

where o are the output values, i the inputs, w the weights, b the biases and σ is the activation function. The structure is shown in Figure 2. Depending on the type of layer, there may be as much as one weight for every pair of input and output nodes - as is the case for fully connected layers. Other architectures have fewer weights, and use the same weights for multiple parts of the input data.

When training a neural network, you need a series of samples, each consisting of input- and output variables, a loss function and the derivative of the loss function. A loss function is a measurement of how poorly a machine learning model is performing — the higher it is, the worse the performance. Strictly speaking, only the derivative of the loss function is necessary to train the network, but knowing the value makes it easier to interpret and compare model performances.

During a single training step in a neural network, three things happen. First, the input variables pass through the network, producing estimates of the output variable(s). Then, the derivative of the loss function with respect to the model weights is computed through backpropagation (repeated use of the chain rule). Finally, all the weights of the network are moved slightly in the opposite

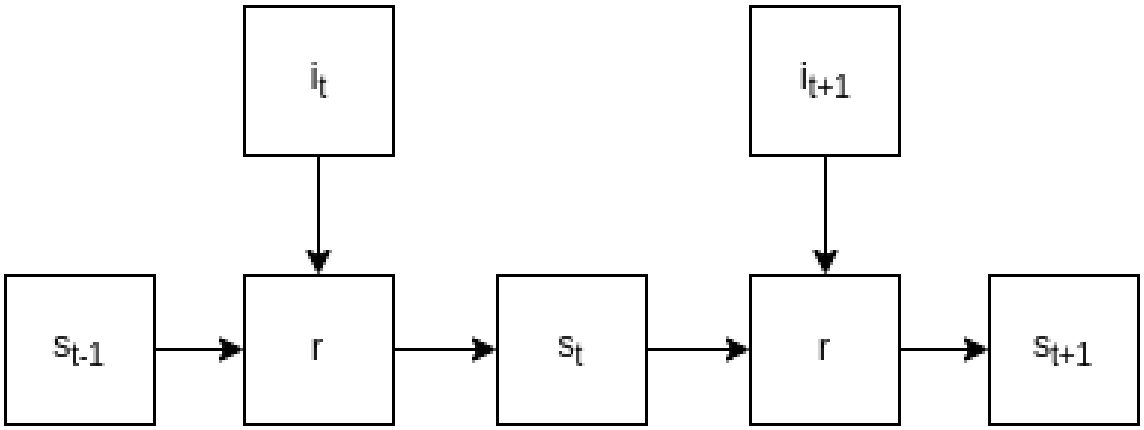


Figure 3: A recurrent neural network r is applied repeatedly to each element i in an input sequence, at each time step updating its inner state s.

direction of the gradient of the loss, in an attempt to make a small reduction in the loss. The training step is repeated several times, to try to achieve as low loss as possible.

2.2.1 Recurrent Neural Networks

Neural networks where layers go directly from input to output, from the previous layer to the next, are known as feed-forward neural networks. When using a neural network to process a sequence, however, it is common to use a recurrent neural network (RNN) instead. RNNs are a type of neural networks specialized to handle sequences of observations. Most of these can handle sequences with differing length.

Unlike feed-forward neural networks, RNNs do not only feed their outputs forward, but also into themselves, illustrated in Figure 3. For each element i_t in the input sequence, a recurrent layer updates its own hidden state s_t, which is computed as a function of i_t and s_{t-1}. It also computes an output o_t as a function of s_{t-1} and i_t. Depending on the further use in the next layer, the output can either be produced as a sequence, or only the final output vector is used.

A key idea in RNNs is parameter sharing. Because the same layer is applied on each time step, there is only one set of trainable parameters in an RNN, rather than one set of weights for each element. This property is especially important when it comes to capturing information that may occur in different places. For example, if processing someone's transactions, consider a person going to the supermarket, and stopping by a gas station on the way home to fill up their car. This is only a minuscule difference from doing them in the opposite order, yet in traditional, fully-connected networks, recognizing both orders of the occurrences would require several more trainable parameters. This would, in turn, require even more data to fit properly, and generalize more poorly. RNNs, on the other hand, use shared parameters for each time step, allowing for better generalization. They can also handle sequences of arbitrary length, which is very useful, as the number of transactions per person in a bank is not fixed.

Training an RNN requires a process known as back-propagation through time (BPTT). BPTT is similar to running normal backpropagation through the same layer repeatedly, once for each element in the input sequence. Recurrent Neural Networks often run in to the problem known as the Vanishing Gradient Problem, where the gradients diminish into almost nothing for the earlier parts of the sequence (Goodfellow et al., 2016). This problem exists for feed-forward neural networks as well, but due to the nature of BPTT, it happens a lot more often, because sequences can easily get longer than the depth of most feed-forward neural nets.

Because the vanishing gradient problem has been somethings RNNs struggle with for a long time, several attempts have been made to solve it. Among these attempts are a class of RNNs called gated RNNs. The idea is to create gates through time, to allow some information from the past to

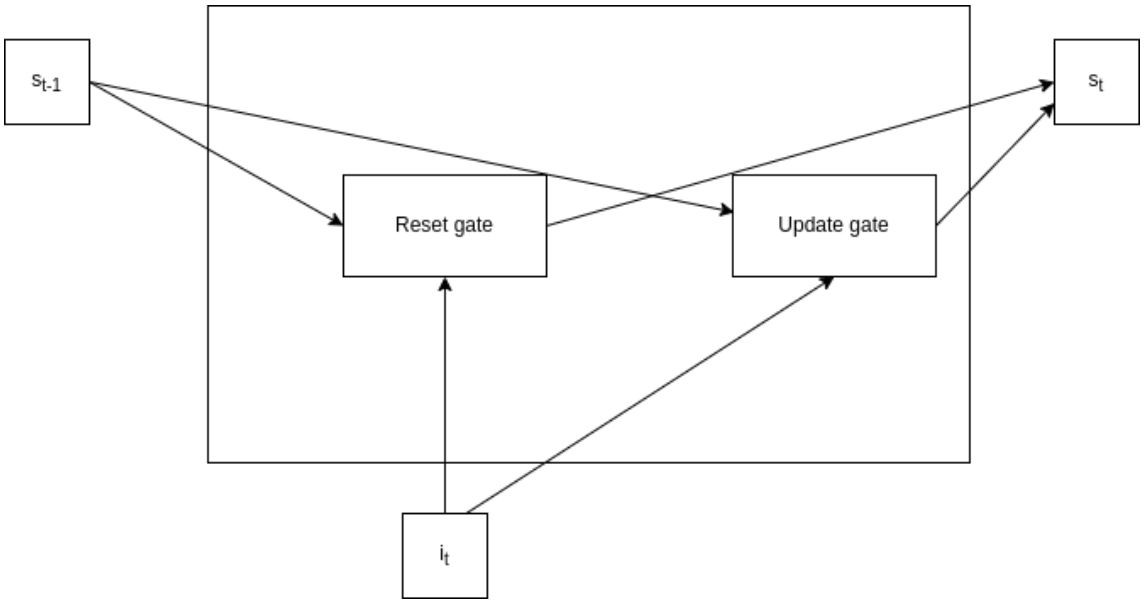


Figure 4: A GRU cell has two gates: a forget gate, responsible for deleting old information, and an update gate, responsible for adding new information to the hidden state

be used again at a later time step. There are many different kinds of gated RNNs, but the most commonly used ones are Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU). The structural difference between a regular and gated RNN can be seen Figure 3 and Figure 4. They follow similar principles — the key difference being that an LSTM cell has three gates, but GRU only has two. Both LSTM and GRU architectures have shown themselves to be effective on different problems, but neither has been proven strictly superior to the other. (Goodfellow et al., 2016)

2.2.2 Transformers

Traditional sequence-to-sequence modeling based is on recurrent or convolutional neural networks as decoders and encoders. A new architecture for sequence-to-sequence learning, called the transformer, was suggested in Vaswani et al. (2017), using only an attention mechanism. This led to a significant improvement in both training time and model performance. In later days, the transformer architecture has shown great promise in natural language processing, most notably in GPT-3(Brown et al., 2020).

The introduction of transformers increased training speed and performance in many problems, but the attention mechanisms involved requires quadratic space in relation to the input sequence length. Typical solutions for this have involved limiting the input to fixed size neighborhoods, where elements outside of each others' neighborhoods are assumed to have no interaction, also limiting the expressive power of the models. Choromanski et al. (2020) propose an accurate approximation of a full-rank transformer, but requiring only linear space complexity.

2.3 Semi-supervised Learning with Autoencoders

Traditionally, machine learning is in one of three categories: supervised learning, in which all the data has labels which are learned, unsupervised learning, in which there are no labels, and reinforcement learning, in which the agent tries to learn optimal actions for long-term reward in an environment. However, neither of these methods properly learn from data which is partially labeled. When dealing with very large data sets that require manual labeling, it is beneficial to be able to use a large unlabeled dataset in addition to a smaller labeled one. This problem is known as semi-supervised learning, and can be considered a combination of supervised and unsupervised

learning.

Generally, an autoencoder is a form of representation learning, which consists of an encoder function and a decoder function. In this report, the term will be used about autoencoder neural networks, i.e. an autoencoder where both the encoder and decoder functions are neural networks. The encoder applies itself, encoding the data, and the decoder tries to reconstruct the original data from the encoded data. Finally, a third neural network, the classifier head, is applied to the output from the decoder network to create a classifier network.

Typically, the encoder network will attempt to reduce the dimensionality of the data. An autoencoder with such an encoder is known as an undercomplete autoencoder. By making the model attempt to reproduce the data from a reduced number of variables, the encoder needs to find the most salient features in the data. An alternative variant of autoencoders, known as regularized autoencoders, can have larger encodings - in some cases even increased from the initial data dimensionality, but impose other restrictions or targets when optimizing, such as stability - where you could penalize the model for having a high derivative.

Often, autoencoders are shallow, with a single-layered encoder and decoders. However, there are many benefits to using deeper autoencoders. It greatly broadens the extent to which the autoencoder can generalize. It can also reduce both computational cost and training data required for some functions. They have also been shown to yield more effective compression than their shallow counterparts.

Training a semi-supervised classifier happens in two stages. First, the autoencoder is trained, using the unlabeled data. After the encoder (as part of the autoencoder) is trained on unlabeled data, the classifier network is trained on the smaller, labeled dataset. The weights of the encoder network may be frozen during the second stage. Using the first stage to learn what patterns exist in the data, and the second stage to learn to label these patterns correctly, this efficiently uses both unlabeled and labeled data, and can perform well even with a very small labeled portion. (Goodfellow et al., 2016)

2.4 Principal Component Analysis

In the field of statistics, Principal Component Analysis is a well known method of dimensionality reduction. It utilizes the correlation between a large set of features to summarize them into a smaller number of representative variables. Each principal component represents a direction (i.e. a unit norm vector) in the N-dimensional space spanned by the features in the data.

The principal components are selected in order to be highly variable - the first principal component signifies the direction in which the data has the highest variance. The later principal components are selected with the same criteria, with the added constraint of being orthogonal to each of the earlier principal components. While a dataset with p features can have up to p distinct principal components, most of the variance in the data can usually be explained by a small number of them, unless the data is uncorrelated.

Principal component analysis is highly scale-dependent, so generally speaking it should be applied to normalized data - otherwise, the principal components will be determined more by the scale of the features than their correlation. (James et al., 2014)

2.5 Fourier Analysis

Fourier analysis concerns finding periodic patterns by decomposing a function into a (potentially infinite) sum of trigonometric functions (Sundararajan, 2001). Because periodic patterns are likely to exist within our problem space, applying discrete Fourier transforms to each individuals' histories might grant some valuable information about patterns in their spending, assuming some level of periodicity — such as a yearly visit to the dentist, or a weekly shopping trip.

Because it finds periodic patterns, we will attempt to use the discrete Fourier transform of indi-

vidual customers, possibly combined with principal component analysis. Ideally, if all shopping habits were perfectly periodic, this would create an efficient encoding of each person's shopping habits, which should be valuable when it comes to understanding their needs.

Of course, such idealized scenarios are rarely ever the case, but applying machine learning to variables in a different domain can always be worth considering, and using predictors in the frequency domain (i.e. Fourier transformed features) rather than time or space has been used successfully before.

It is worth noting that, when applying a discrete Fourier transform (DFT), the highs and lows of the transformed data often differs by several degrees of magnitude. It might, therefore, be advisable to apply either a logarithmic transform after the DFT, or use methods that are insensitive to scale. An ideal candidate in such a scenario is using tree-based methods, as these are independent not only with respect to scale, but to any strictly monotonous transformations. Such methods include Random forests and Gradient Boosting.

2.6 Supervised Learning Methods

When applying semi-supervised learning to a problem, some sort of supervised learning method should be applied at the end. These are methods where each training sample has a set of input variables, as well as one or more target variables with corresponding values. There are several different methods for this, all of which have distinct strengths and weaknesses.

2.6.1 Neural Networks

Neural networks, as described in Section 2.2, are a highly versatile method of supervised learning. They provide great flexibility, but require vast amounts of data, and always run the risk of overfitting very easily.

2.6.2 Tree-based Methods

Decision trees and regression trees are simple methods of classification and regression, based on a recursive binary splitting of a dataset. Evaluating a decision or regression tree is similar to a series of yes or no questions, each concerning the value of one of the features being greater than a given threshold value or not. For each question, or "split", the model passes the sample to the appropriate sub-tree. This is repeated until there are no more splits in the subtree - at which point a terminal node, or leaf, has been reached, and the tree provides an estimate of the variable which you are trying to predict. An example of a fitted decision tree is shown in Figure 5.

Training a decision tree, you begin with the full dataset at the root of the tree. For each feature, you compute an optimal split on the dataset, so that the loss function — typically MSE for regression and gini index for classification — is as low as possible for the new groups. The example in Figure 5 has divided the dataset until each leaf node is unanimously the same type. This can often be the case for small, or linearly separable, datasets. For more noisy data, it can often be beneficial to stop at a certain depth, and instead yield probabilities for each of the classes. Usually, this is done by first growing out a very large tree, and later pruning away the splits that do more harm than good, e.g. by comparing on a validation set.

In most cases, the predictive power of a single decision tree is very limited, and they are prone to overfitting. They do, however, have some very useful properties, that are also inherited by other tree-based methods. Each split in a decision tree is binary, only dependent on which samples are higher or lower than a given value. This means that it is unchanged, and invariant with respect to any strictly monotonous transformations of the features. It is not invariant of any transformation of the labels — this is, however, only a possible problem for regression trees, as categorical variables have no meaningful transformations.

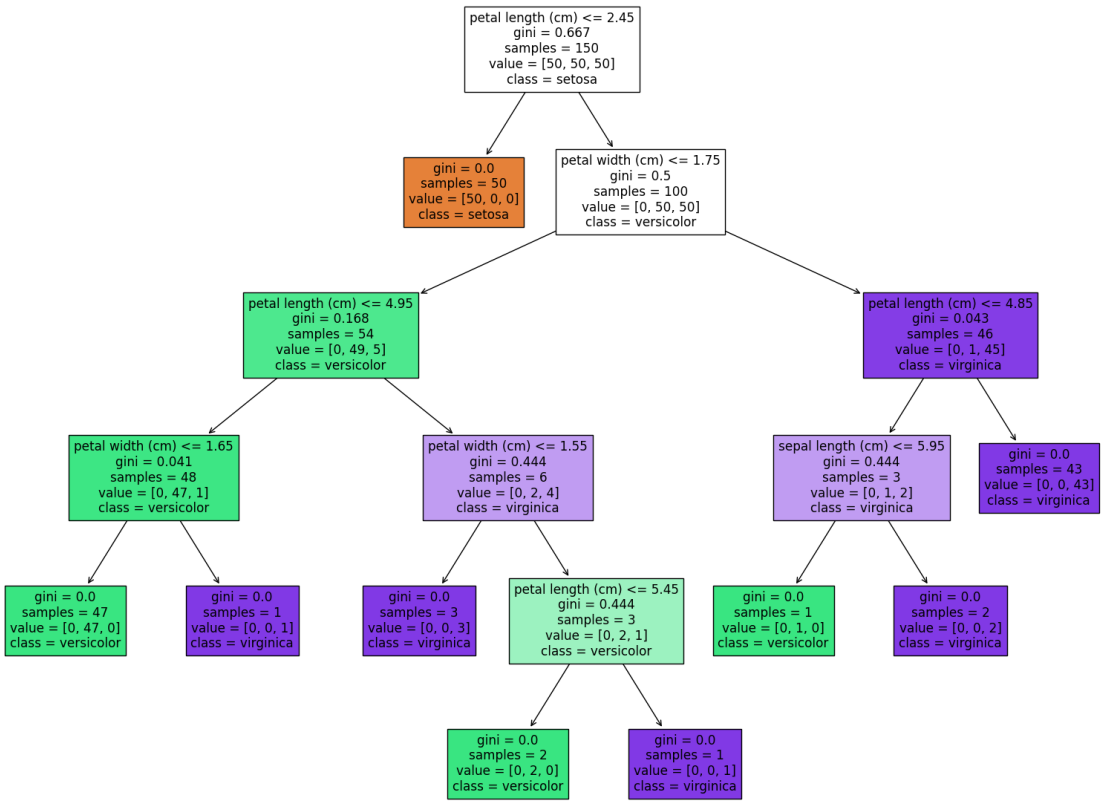


Figure 5: A single decision tree, fitted to the iris flower dataset (Fisher, 1936). All the setosa samples were captured by the first split, while it took a few more questions to separate versicolor from virginica samples.

An extension of decision trees, based on bootstrapping, is bagging. Instead of training and pruning a single tree based on the entire dataset, we can construct and fit B separate trees from B corresponding bootstrapped training sets. Each tree trains individually and independently, and they process all inputs individually and independently. When all trees have computed a prediction, the bagging model takes the mean of the individual tree predictions as its own estimate, as shown in Figure 6.

Random forests are based on bagging, but also attempt to decorrelate the different bagged trees. It is common that some variables are selected more often than others, which is often beneficial for the individual trees. However, when training several trees to act together, they all benefit from being different — to reduce their correlation, and in turn decrease the total variance. To accomplish this, random forests impose another restriction on the trees. Whenever a node in the tree proposes a divide of the data, it only gets m randomly selected features to choose from. It is common to use $m = \sqrt{p}$, where p is the total number of features. The m eligible features are selected at random for each node, so each tree can use most of the features during the training period, but none of the features are considered at every split. This makes each tree in the random forest a lot more different from each other, because they are trained with independent restrictions. Because they are now largely decorrelated, variance decreases substantially the more trees are added to the forest. One interesting, and highly beneficial, property for random forests is that they don't risk training for too long — adding more trees to the forest will never lead to an overfit, because every tree is uncorrelated.

A final tree-based method is boosting. Similarly to bagging, this method also uses a large number of decision trees. However, instead of training large, independent trees in parallel, boosting grows several trees sequentially, and in a highly correlated manner. The basic idea is slow learning, by fitting several small trees. However, unlike bagging and random forest, only the first tree is fit on the original data, and the subsequent trees are fit to the residuals after the previous tree — that is, how much the previous prediction missed by. For each new tree, new residuals are computed based on the sum of all the previous trees.

To ensure that learning happens slowly, all tree predictions are shrunk by a shrinkage parameter of λ, which is a positive, typically very low value, like 0.01 or 0.001. Assume for the sake of an example that two trees are fit to two samples, with ground truth values $y = (-3, 3)$, and shrinkage parameter $\lambda = 0.4$. Assuming a perfect fit — which is usually unlikely with real data, but trivial in a simple example like this — the first tree provides estimates $\hat{y} = (-3, 3)$, which is then shrunk to $\lambda\hat{y} = (-1.2, 1.2)$. The second tree is then fit to the residuals, i.e. the amount by which the prediction missed, $y' = (-1.8, 1.8)$. Because boosting trains slowly, it also overfits slowly, but it can train for too long and yield an overfit to the training data. (James et al., 2014)

2.7 Performance Metrics

When defining metrics in a supervised scenario, it is common to define it in terms of the predicted labels up against the correct ones, e.g. accuracy or R-squared. However, the target of this project is not to find an optimal model for transaction classification, but rather to compute useful input variables for other machine learning models to take advantage of.

2.7.1 Feature Importance

In bagging and in random forests, on every split, one out of a given number of features is chosen. Because the target when generating a split is to minimize either Gini index or MSE, the relevant metric is computed before and after each split of the dataset. Based on this metric, feature importance is computed in each individual tree, defined as the total amount by which the metric is improved over all of the splits where the given feature is chosen. The overall importance in the forest is typically expressed as the mean of importances in each tree. (James et al., 2014)

Because computing feature importance requires the target variables, it is imperfect in terms of judging the general quality of a feature. Nonetheless, noting the (mean) variable importance for

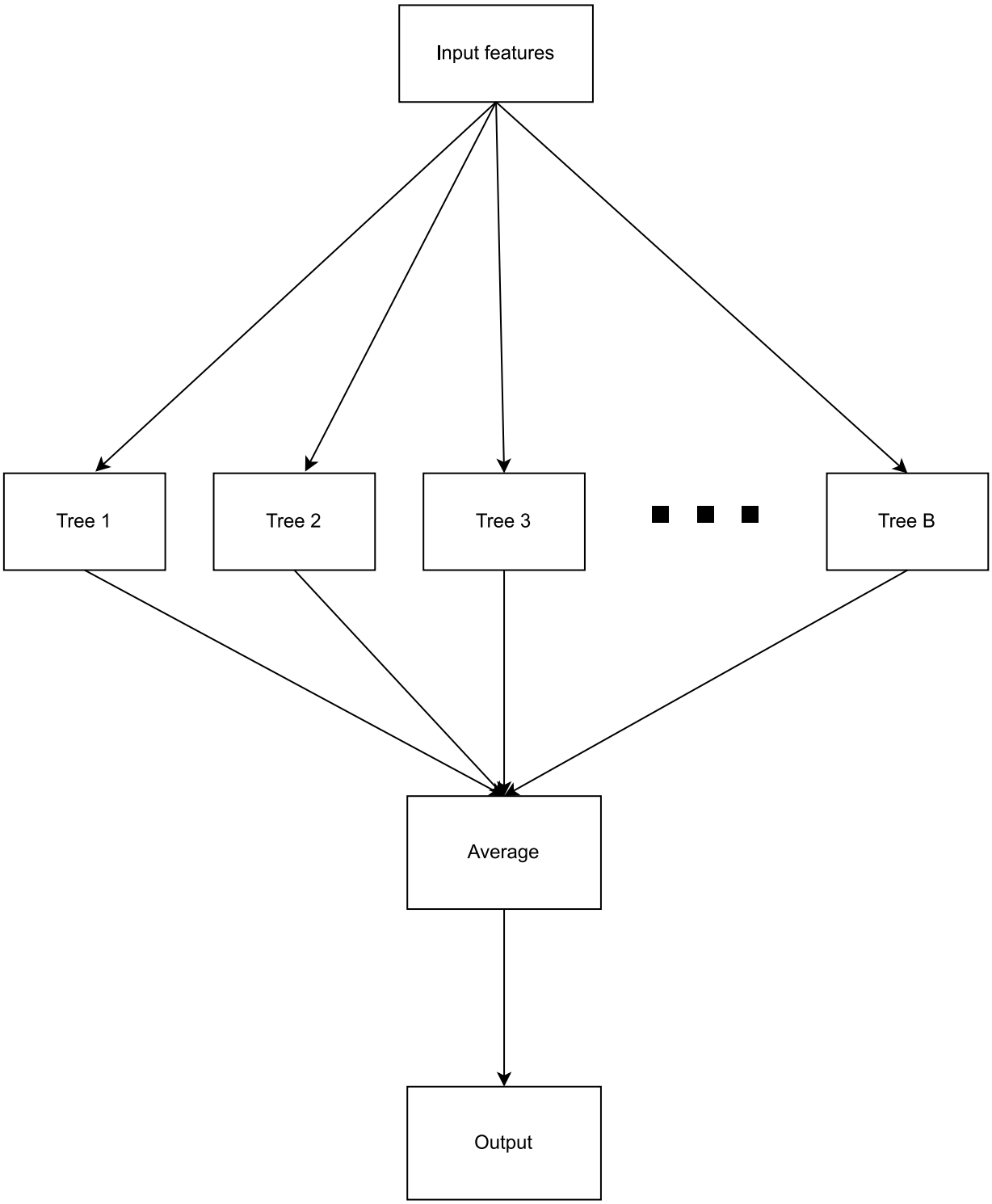


Figure 6: Bagging uses several different trees, and uses the average of their predictions to make a single prediction

the aggregated features, particularly compared to manually-defined aggregates, is a good metric to know that the model utilizes the new features well.

2.7.2 Model Improvement

Another metric to take into account is how the additional features impact the general supervised models. By evaluating the accuracy of a model before and after adding the new features, we get what is arguably the best metric for feature quality.

Much like feature importance, this is specific to the labels in whichever dataset is chosen. A difference between model improvement and feature importance is that feature importance is easily computable in training, but also slightly more naive. For example, if you compute feature importance of two highly correlated variables in a larger dataset, both will have similar values for feature importance, but the inclusion of one reduces the feature importance of the other, and the inclusion of both variables could yield no actual improvement compared to just one.

Hence, measuring the actual improvement of the model might be a better metric, as it also takes into account the novelty of the new features. A reasonable feature importance score can be achieved by repeatedly adding the same feature, assuming it is a feature that has a considerable amount of predictive power. This strategy is, however, not likely to mark a notable improvement in the actual predictions, but only dilute the importance of each occurrence of the identical features.

Typical metrics depend on the type of problem. In classification problems, typical metrics are cross-entropy and prediction accuracy. Cross-entropy is a better metric when attempting to predict probabilities accurately, but accuracy is easier for humans to interpret, and can arguably be a better performance metric in cases where the goal is only predicting, with no consideration for uncertainty. In regression problems, typical metrics are mean squared error and R squared (portion of variance explained). These two metrics are closely related, and quite similar. The primary difference is that R squared is easier to interpret across models — as R squared is always a value between 0 and 1, whereas the scale of the mean squared error depends entirely on the data in question.

Most of the problems relevant to SMN are classification problems, and more specifically, the target feature we will be working with is whether a customer will change their address during the next three months. As such, the features we will be observing will be accuracy and cross-entropy, with a priority on cross-entropy — because it is important to the bank that they can find accurate probabilities, not just binary estimates.

3 Related Works

This section looks at earlier works that have attempted similar things to our project, including the key similarities and differences between the fields.

In order to preserve the privacy of their customers, banks rarely share their data. Therefore, there are not many closely related works available to the public. However, since the methods proposed are not necessarily exclusive to the finance industry, we can look to related works applying more or less similar methods on different kinds of data.

3.1 Natural Language Processing

Natural languages are a common example of sequential data, and is also a field where there is an abundance of, mostly unlabeled, data available. Furthermore, words are commonly encoded as vectors in these works, which makes the shape of the data very similar to that of bank transfers — a variable length sequence of vectors. As such, this is one of the most directly applicable fields to ours.

Dai and Le (2015) describes successful use of Semi-Supervised Sequence learning in text classification. Their findings showed that Semi-Supervised LSTM networks had more stable performance, and generalized better than their counterparts without pretraining. Using this method, they beat the previous best reported results for four different datasets. This applied to multiple types of task, spanning from sentiment analysis to more general text classification, and even showing some promise for image object classification.

One key difference between bank transactions and natural language is the question of noise. A sentence or article written in a natural language is designed to bear some meaning, because the intention of the sentence itself is to express a message. If you add a new word into a sentence, or change one of the existing ones, it might change the meaning. Anyone writing a sentence will typically keep this in mind, and construct it in a way in which their message clearly gets through. When comparing words and bank transactions, the bank customer rarely intends to express anything with their monthly spending, and as such it must be expected to be much more noisy.

3.2 Genetics

Another field in which sequences play a central role is genetics. Semi-supervised learning has seen some different use cases in genetics, and has shown itself to be a versatile method when working with sequential data. Like in languages, we know that sequences play some role in genetic expression, but unlike in languages, some noise will naturally occur due to random mutations. The dimensionality of each element in a genome is also quite small, being limited to four bases (A, T, C and G), which is a lot smaller than the vocabulary size required in language models. Transactional data will likely be somewhere inbetween the two in terms of dimensionality - almost certainly including more than four simple features per transaction, but very unlikely to go in to the thousands.

3.2.1 Imputation of Single-cell Gene Expression

RNA-sequencing is an emerging technology, but has limitations, including a high rate of dropout. Badsha et al. (2020) describes using an autoencoder to impute such values, showing that deep autoencoders have ability to beat existing methods' MSE, picking up on nonlinear relations between genes, and additionally being highly effective in terms of computational efficiency and scalability.

3.2.2 Genetic Prediction

Semi-supervised learning has also been used for prediction on gene sequences. Xie et al. (2017) used a deep autoencoder model to predict gene expression, which is determined by many factors, all the way down to DNA-level. Peng et al. (2019) uses similar methods to reduce the dimensionality of genetic data, in order predict the occurrence of Parkinson's disease.

3.3 Power System Dynamics

Being able to predict the changes in a power grid efficiently, and in real time, is very useful and, in the long term, necessary, because renewable energy sources like solar and wind power are unstable, and harder to plan for in advance. Currently, doing this requires numerically solving a set of ordinary differential equations (ODEs), an operation which is too computationally intensive for real time use. Therefore, it is a common practice to limit the amount of e.g. wind power generated into the grid, to keep everything as stable as possible. In the longer term, using effective and accurate predictors could become a necessity when moving away from non-renewable energy sources.

Cui et al. (2021) proposes a framework for prediction of power system dynamics in the frequency domain (i.e. using a Fourier transformation). Applying machine learning methods in the frequency domain, they demonstrate high accuracy in prediction and fault detection, while reducing computation time by orders of magnitude compared to traditional methods.

They compare metrics of their methods to those of the state of the art, which runs prediction in the time domain. Their proposed methods had less than half of the previous mean squared error, when compared to the state of the art machine learning methods, including Physics-Informed neural networks.

3.4 Learning in the Frequency Domain

As a Fourier transform can be applied in several dimensions, it has also been used for higher-dimensional data, such as image processing. Xu et al. (2020) describe some benefits of the fourier transform. Importantly, image downsampling (in the spatial domain) makes no distinction between important and redundant data when removing information. Because many machine learning models require a fixed input size - as is the case for this project as well — this can be problematic.

For this purpose, methods for data preprocessing and feature pruning were also proposed, including a learning based method for frequency selection. The method involves a gate module, which scales importance from 0 (completely redundant) to 1 (highly salient).

Applying their models to a static selection of frequencies (i.e. a fixed set of features in the frequency domain) instead of fixed size images, they improve upon the top accuracy provided by some of the best performing models on the ImageNet dataset, including ResNet-50 and MobileNetV2. They also observed an 0.8% accuracy improvement on image segmentation on the COCO dataset.

3.5 Use of the Fourier Transform for Feature Extraction

Heidari et al. (2021) describes the challenge of learning from data with nonlinear redundancies, and the use of the fourier transform to effectively extract information in such scenarios. Their methods apply to the supervised problem, using the relations between features and labels, i.e. feature extraction for a single supervised problem.

Other alternatives considered therein had drawbacks when compared to the discrete fourier transform based methods. The most common methods for dimensionality reduction only capture linear relationships, which is not sufficient to cover most real-life patterns. Other more flexible methods of feature extraction include kernel-based methods, which can capture nonlinear dependencies —

but the $\omega(n)$ time complexity of their computation process makes them scale poorly, which is not ideal when the sample size grows very large.

Column	Description
Customer ID	a unique identifier, necessary to make connections between customers and connect them to transactions
Month	The month for which this row was recorded
Gender	Gender of the customer
Age	Age of customer, measured in whole years
Customer Start Date	The time of the beginning of the customer's relationship with the bank
Post Code	An anonymized version of the customer's postal code
Municipality	An anonymized code for the municipality in which the customer lives
Country	An anonymized code for the customer's country of residency
Sum of loans	The total amount the customer owes in loans
Sum of deposits	The total amount deposited into the customer's accounts
Customer segment	Which customer segment the customer is in. Indicates how many of the bank's products the customer uses
Moved	Whether the customer changed their address during the month in question

Table 1: Data format in the customer table

4 Data

This section describes the data which is used throughout the practical part of the project. The data in focus for this project is customer data, provided by Sparebank 1 SMN.

4.1 Format

The dataset consists of four tables: one with background information on each customer, one for account transactions, and two for debit and credit cards, respectively.

Several columns here are anonymized. The anonymization process is simply using a simple index representing the value, instead of the actual value. For example, the postal code 7030 might be assigned the number 2, and the assigned number will be consistent for all postal codes throughout the dataset. This sacrifices information regarding the proximity of postal codes, such as 7031 being closer to 7030 than 7100, but it was necessary in order to keep the data sufficiently anonymous. There are potential ways to work around this, e.g. making sure the indices are assigned in ascending order of postal codes, but it will always end up being a matter of balancing salient data with the concern for customer privacy, in which case the bank will rather err on the side of caution.

4.1.1 Customer

The customer information table contains basic, but anonymized, information about each individual customer. The table contains monthly updates about the customer, i.e. there is one row per customer per month. This background data is shown in Table 1.

4.1.2 Account Transactions

Account transactions are any transactions that transfer money into or out of a customer's bank account. The data from them are shown in Table 2.

Column	Description
Customer ID	the unique identifier of the customer
Date	The date on which the transaction happened
Account number	An anonymized version of the account number
Other Account	the account number of the counterpart of the transaction, following the same anonymization as account number
Classification code	An internal classification of transactions, specifying if they are related to e.g. a minibank, an electronic invoice or salary payments
Currency	The currency in which the transaction is made
Amount	The amount being transferred into an account, in original currency. Negative numbers represent outgoing transfers.
Amount in NOK	The amount of money being transferred into the account, converted to NOK
SMN Transaction	Flag representing whether the transaction is between two SMN accounts.
Customer Internal Transaction	Flag representing whether the transaction is between accounts belonging to the same customer
Debet flag	Flag representing whether money is outgoing
System Generated Transaction	Flag representing whether the transaction was generated by the system, e.g. interest and fees.

Table 2: Data format in the account transaction table

4.1.3 Debit and Credit Card Transactions

The final two tables follow a very similar format, as they both relate to card payments. The format of the data stored in these tables is shown in Table 3.

4.2 Scope

The data contains data pertaining to bank transactions for Sparebank 1 SMN's private customers over 2 years, with certain exceptions, which are described in Section 4.3.

4.3 Limitations

Due to the General Data Protection Regulations (GDPR), there are strict limits on what can and cannot be done with this data - as a person's transaction history includes a lot of personal data. This means, among other limitations, that no part of the data may leave Sparebank 1's premises, and all analysis must run on their systems, on their own hardware. It also means we are not allowed to inspect the data - only apply the code and test its performance. In addition to these usage guidelines, the data itself has been filtered, removing certain transactions that are considered too sensitive to use in machine learning models, such as visits to the doctor.

Furthermore, GDPR imposes strict rules for anything that can be considered identifiable, including combinations of several features. As such, any combination of personalia that is too rare is also removed before the data is made available to our machine learning models. The factors considered to be identifiable here are age, gender, and postal code. While the postal codes themselves are anonymized (by mapping to an index), they could be possible to identify or narrow down from e.g. the number of users in each postal code. Therefore, any combination of these factors with less than 5 unique customers are filtered out. Finally, all customers below the age of 18 are removed

Column	Description
Customer ID	The unique identifier of the customer
Date	The date on which the transaction happened
Account number	An anonymized version of the account number, corresponding to those in the Account transactions table
Currency	The currency in which the transaction is made
Amount	The amount being transferred, in original currency.
Amount in NOK	The amount of money being transferred, converted to NOK
Transaction time	Time of day on which the transaction was made
Transaction counterpart	An anonymized code representing the place in which the transaction was made
MCC	The Merchant Category Code of the counterpart, as defined in the ISO 18245 standard (International Organization for Standardization, 2003)
Group	Categorization of merchants, more coarsely grained than standard MCC
Group description	Description of said group
Main group	A more coarse grouping
Main group description	Description of the main group

Table 3: Data format in the credit- and debit card transaction tables

from the data used in this project. In total, this leaves approximately half of the bank's private customers remaining in our data set.

Anyone under the age of 18, and anyone who belongs to a too narrow demographic, are excluded in the project. As a consequence, the data, and by extension the machine learning models fitted to the data, will likely be skewed towards adults in densely populated areas.

5 Methods

The methods section covers all the key methods that will be applied in Section 6, as well as an outline for how the experiments will be conducted.

The practical methods applied to the data are written in Python 3.7. This includes preprocessing, learning and evaluation.

5.1 Tools and Libraries

The python libraries used for the project are:

- numpy, version 1.19.2 [1]
- tensorflow, version 2.3.0 [2]
- scipy, version 1.6.2 [3]
- scikit-learn, version 0.24.2 [4]
- pandas, version 1.3.4 [5]

5.2 Preprocessing

5.2.1 Data Scale

Different columns in the data can often take on values that differ by several orders of magnitude. This can bring to light several issues.

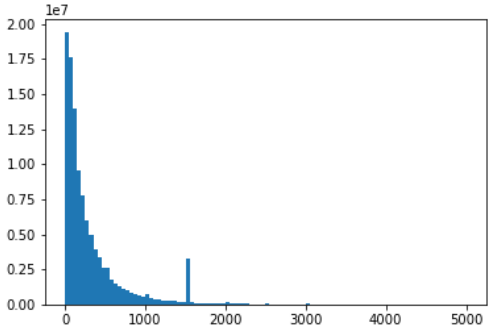
For example, if a bank transaction can be anything from NOK 20 to buy a soda in a store to NOK 5000000 to buy a house, the scale at which an autoencoder model might miss is extremely high, and can very easily become problematic in terms of convergence. Typically, neural networks train more effectively when the data is normalized to zero mean and unit norm (LeCun et al., 1998), which is not the case for our data set. Histograms for the transaction sizes, sorted by transaction types (as described in Section 4), are displayed in Figure 7.

If we simply normalize the data by dividing by a fixed number (e.g. standard deviation), only the very large transactions retain any influence at all. We propose two alternatives to avoid this problem.

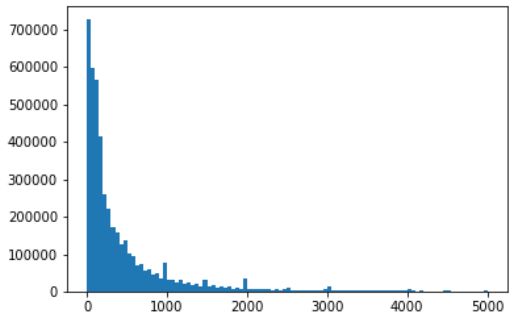
The first alternative, applying a logarithmic transform, is a simple and easily applicable mathematical function, which narrows the dynamic range of the dataset down. Another benefit is that it makes differences more relative to the size of the transaction. Before a log transform, estimating 4200 when the actual value is 4000 is considered equally bad as estimating 240 when the actual value is 40, as both miss by 200. After applying a log transform, it becomes more scale-sensitive, and predicting 4200 instead of 4000 is the equivalent of predicting 42 instead of 40. It does not guarantee any distribution - but does bring all the data into a much narrower frame, in which it might be plausible to simply divide by the standard deviation.

The results of applying a logarithmic transform to amounts in question resulted in distributions shown in Figure 8. The histograms all appear to be close to a gaussian distribution, and normal Q-Q plots also seem to indicate this being the case.

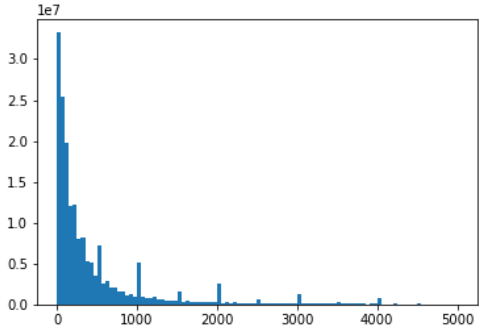
[1] https://numpy.org/doc/1.19/
[2] https://www.tensorflow.org/versions/r2.3/api_docs/python/tf
[3] https://docs.scipy.org/doc/scipy-1.6.2/reference/
[4] https://scikit-learn.org/0.24/
[5] https://pandas.pydata.org/pandas-docs/version/1.3/index.html



(a) Histogram for debit card transactions



(b) Histogram for credit card transactions



(c) Histogram for account transactions

Figure 7: Histograms showing the distributions for the size of debit card, credit card, and account transactions, respectively. Apart from taking the absolute value, to get the size without a positive or negative sign, no transformations have been applied to either of the distributions. The histograms were cut off at 5000 in order to display them properly, as larger and larger transactions get rarer and rarer.

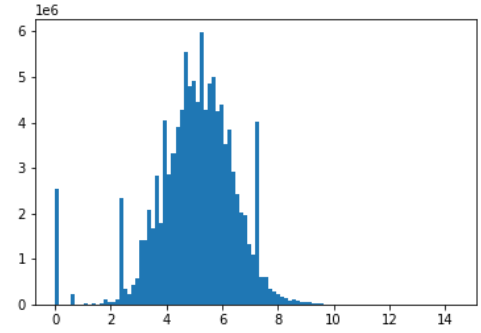
The other alternative is using quantiles. Having a significant number of transactions available, it is simple to compute quantiles, i.e. estimate the distribution of the transaction sizes. These quantiles can, in turn, be used to translate transaction sizes to the portion of the dataset that it is greater than, which will yield a uniformly distributed feature. This can either be used as is, or it can be used to translate to the corresponding quantile in e.g. a gaussian distribution.

5.2.2 Joining the Tables Together

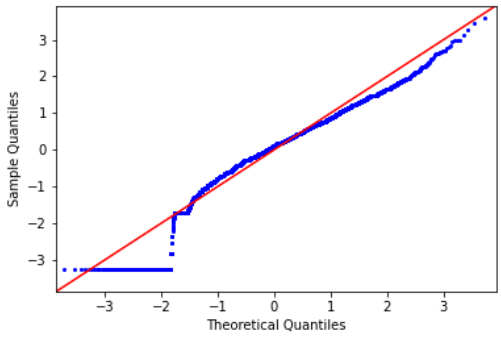
The first piece of the preprocessing pipeline will be joining the data from several tables together, and piecing them all in to customer objects. Each customer should have a three transaction histories - one for account transactions, one for debit card and one for credit cards.

The procedure for joining the tables is relatively simple: For each unique Customer ID, initialize background information with the latest available data from the Customer table. Then, for each of the three other tables, select the transactions with matching customer ID, and insert them into three arrays, each corresponding to one of the three tables. After processing all N customers, this should result in N sets of background information, and $3 \cdot N$ arrays of transactions.

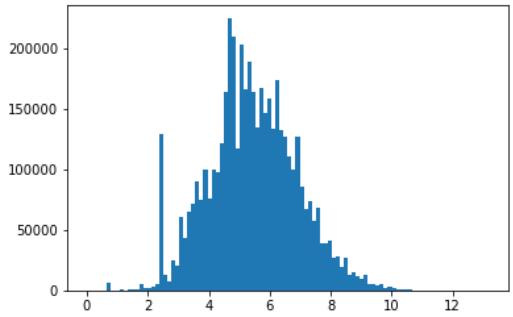
Naively, this algorithm runs in quadratic time. This can be reduced to $O(n \log n)$ by first sorting the data with respect to customer IDs and dates, and then iterating sequentially through the tables.



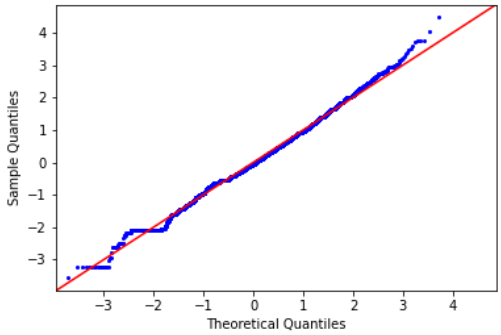
(a) Histogram for log transformed debit card transactions



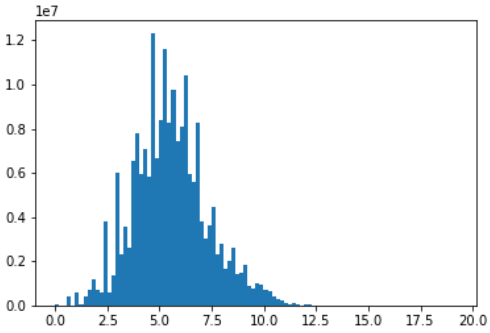
(b) Q-Q plot for debit card transactions



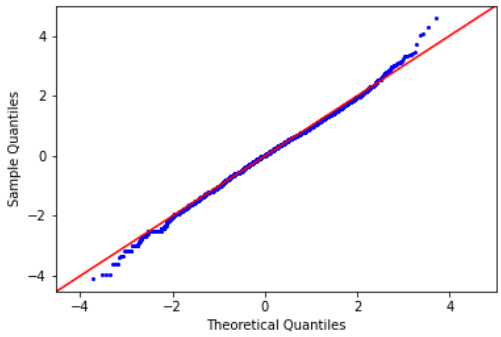
(c) Histogram for log transformed credit card transactions



(d) Q-Q plot of debit card transactions



(e) Histogram for log transformed account transactions



(f) Q-Q plot of account transactions

Figure 8: Histograms and Q-Q plots showing the distributions for the size of debit card, credit card, and account transactions, after a log transform has been applied. Q-Q plots seem to indicate that they are close to gaussian, although the lower quantiles of the debit and credit card distributions seem to have some consistent mismatch, so it's likely not perfectly gaussian.

5.2.3 Data Units

For both of the proposed methods, there is a question of how to shape the data before applying the learning process. On the one hand, the discrete fourier transform has well-known properties when applied to time series, i.e. when each step in the series is represents a given unit of time - where we'd know that periodic patterns, such as a weekly shopping trip, will be recognized - which is not necessarily true if patterns are recognized by the number of transactions that are made, rather than actual time. Furthermore, when attempting to extract features to summarize a

person's shopping patterns, it is likely to be more reasonable to train an RNN to recognize what happens in given time steps, rather than e.g. punishing a model for not knowing the order in which two store visits take place. Both of these can be reasons to use period-based data, e.g. summaries per day or per week, instead of pure transaction-level data.

On the other hand, translating from transactional data to per-day or per-week summaries, inherently requires a way to aggregate the data in question. As such, it misses out on the possibility that there might be something important in the transaction-level data. Furthermore, it is possible that some periodic patterns could apply when measuring time by the transactions themselves - if a person has very clear-cut shopping habits, making the same purchases every week, but the day and time of the trip to the store varies from week to week, a discrete fourier transformation of the raw transactions would capture the periodic tendencies better.

Because both of the alternative time units - both transactions and real-world time units, have apparent benefits and drawbacks, both should be attempted and their results compared to one another.

Another choice that must be made is how to measure each transaction - such as account balance after transaction, or simply using the transaction size and direction. In this case, we will use the value of the transaction itself. This will, in itself, be easier, because it's already stored in the data that is provided. Furthermore, because certain transactions are filtered for privacy reasons, the balance we can measure with the available data will be heavily time-dependent, and by the end, the balances we can compute with our data will likely be off by quite a lot.

5.3 Autoencoder

After applying either preprocessing pipeline, a deep autoencoder model will be fitted to the data. It is possible that some of the background data, such as age and gender, are already good predictors, and that some detectable patterns in transactional data might be attributable to such factors.

To prevent our model from learning features that are already well-known and defined, we will attempt to decorrelate the encoder from these features. This will be done by allowing the decoder to use these variables directly, as external regressors, as shown in Figure 9. In an idealized scenario, this would mean that the decoder can use the background data, that is now supplied directly, to predict patterns related to those, and the encoder will be encouraged to find other, uncorrelated features in the data.

5.4 Fourier Approach

After all preprocessing is complete, a discrete fourier transform will be applied to the transaction histories to each of the customers in question. First, a large, fixed number of the frequency domain values will be used directly in machine learning models. Then, feature selection or another form of dimensionality reduction (such as principal component analysis) will be applied, and compared to the naive approach of using all values in the set.

Strictly speaking, the size of a discrete fourier transform depends on the length of the sequence in question, and as such, it does not directly solve the problems that motivated the problem. This is, however, a minimal problem; it is quite simple to fix the size of the fourier transform to a given range, by substituting the (theoretically consistent) value of 0 to anything outside the selected range, and removing frequencies that span over longer time spans than the same range.

5.5 Evaluation

Encoder performance will be quantified in a supervised manner. Applying various supervised learning methods to each of the different data encodings, we get an indication of how well the different encoders, and their hyperparameters, work.

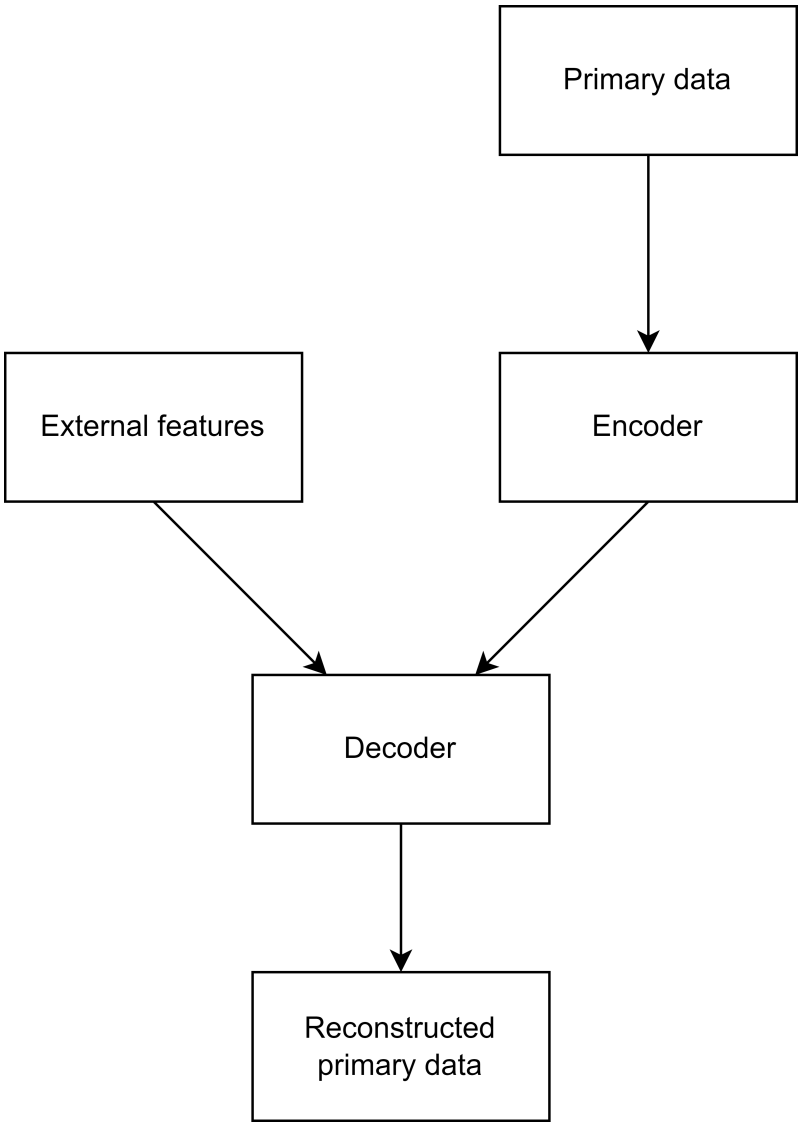


Figure 9: An autoencoder corrected for external regressors

The best models of each type (deep autoencoder and fourier encoder) will be selected based on their validation set performance, and finally their test set performance will be reported.

5.6 Experiment Flow

The experiment will be conducted in three overall phases: unsupervised training, supervised training, and test set evaluation.

5.6.1 Unsupervised Training

During the first phase, we will apply unsupervised experiments, primarily with sequence autoencoders, and experimenting with different kinds of preprocessing. This will give us indications of what kinds of training are plausible, and result in having a few different, saved models for feature extraction.

5.6.2 Supervised Training

During the supervised training phase, we will use our automatic feature extraction methods to produce features. We will try to use these features as input features to gradient boosting- and random forest classifier models. We will also measure and report the validation cross-entropy and accuracy for each feature extractor and supervised learning method.

In addition to automatic feature extractors, we will define some features manually, which we will also use to train supervised models, similarly to what we did with the automatically generated features. Metrics will also be reported for these.

Furthermore, we will try to combine some manually and automatically generated features, and train corresponding models based on the combined feature set, to see whether they work better in combination than either feature set does separately. We will also report the validation metrics for these models.

5.6.3 Test Set Evaluation

After training all of these models and observing their validation metrics, the best manual model will be selected based on validation set metrics. The best model using either automatically created or combined features will also be selected, as the target is to know whether automatic methods can provide useful features.

Finally, the best feature set will be determined based on their performance when applied to the test set.

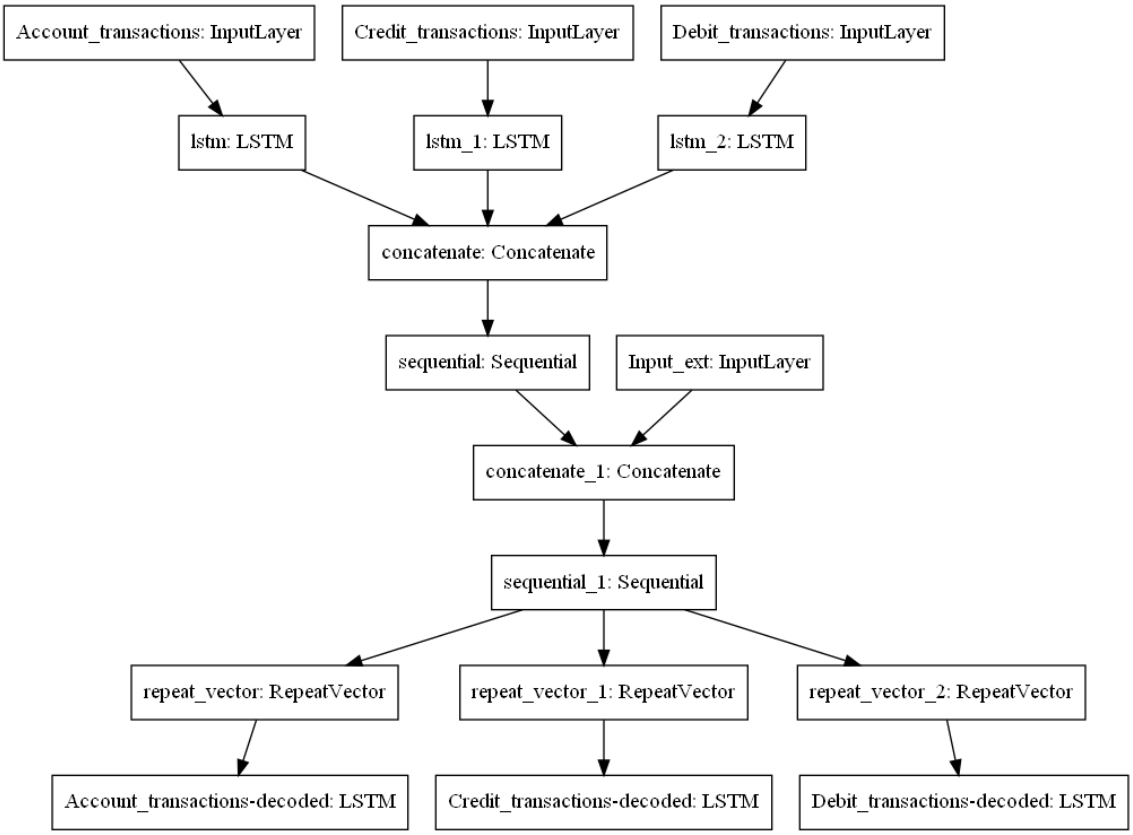


Figure 10: The data flow of the autoencoder begins with three sequences, each being processed, combined, and then being processed back into three separate sequences afterwards. The Sequential boxes consist of one or more fully-connected layers, allowing for deeper patterns in encoding and decoding.

6 Results

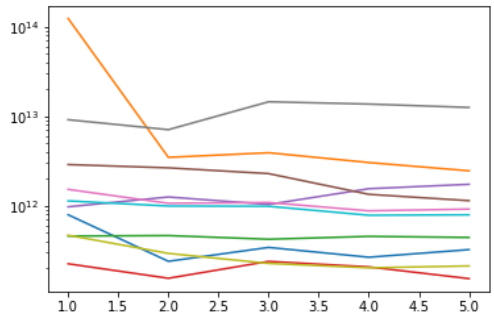
6.1 Aggregation

Preliminary results were produced with a subset of the data containing 5% of the total customer data, to experiment with different configurations and preprocessing pipelines with a lower time investment.

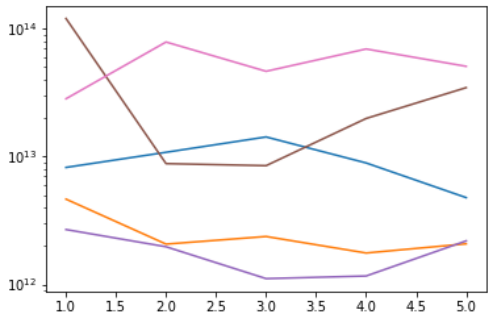
The autoencoder network architecture is displayed in Figure 10. There are three input- and output-sequences - the fields differ slightly between the three tables (credit card, debit card and account transactions), and as such each table is treated as a separate sequence of independent variables and dimensionality. Before any target variables were considered, unsupervised training history was a useful indicator to see how well the model was adapting itself to the data. If the autoencoder's own loss does not decrease, that is a clear indicator that it could not find any patterns.

After some trial and error, it became apparent that processing the raw transactions was infeasible: a single epoch on the preliminary 5% of the data would take 8 hours of training time, and the full data would require 20 times that, almost a full week per epoch. Furthermore, training often failed very early in the training process, with an abundance of NaN values in its training history, despite the data being scaled properly. This problem appared very regularly for raw transactions and 7-day aggregated data, less commonly on data encoded to 14-day periods, and was never encountered on data encoded for 31-day periods.

The computation time was also significantly reduced by longer periods, with an epoch on 14-day encoded data only requiring 30 minutes (when applied to the preliminary 5%). This improvement in

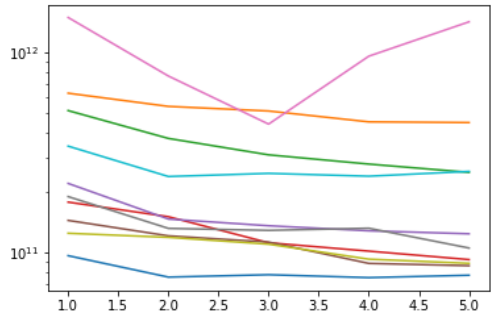


(a) Training history for 10 trials with equal configurations, using the more complex model on data summarized for 14 days. The x axis represents epoch, the y axis mean squared error, and differently colored graphs represent training histories for different trials.

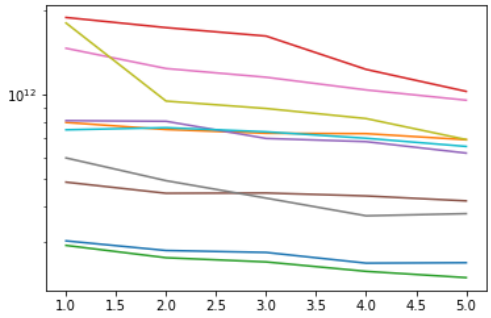


(b) Training history for 10 trials with equal configurations, using the simplified model on data summarized for 14 days. The x axis represents epoch, the y axis mean squared error, and differently colored graphs represent training histories for different trials. Out of the 10 trials that were initiated, only 5 completed without failure, with the remaining 5 rapidly resulting in many NaN values.

Figure 11: Unsupervised training results on 14-day summarized data, before and after simplifying the autoencoder model. The simplified model appears less stable, resulting in frequent NaN values occuring. The y-axes are log scaled, as different initial states impact the training results by orders of magnitude.



(a) Training history for 10 trials with equal configurations, using the more complex model. The x axis represents epoch, the y axis mean squared error, and differently colored graphs represent training histories for different trials.



(b) Training history for 10 trials with equal configurations, using the simplified model. The x axis represents epoch, the y axis mean squared error, and differently colored graphs represent training histories for different trials.

Figure 12: Unsupervised training results on 31-day summarized data, before and after simplifying the autoencoder model. It appears that the more complex model performed better on average in unsupervised training, but somewhat less stable.

computation time is inversely proportional to the period lengths, as longer periods effectively reduce the sequence lengths, and recurrent computations take the majority of the time in computation.

However, even though shorter sequences lead to more stable results, with training loss following a downwards trend, the values depend entirely on the initial weights. Running training several times with equal hyperparameters, we can observe loss values varying by several orders of magnitude. Training histories from 10 separate were observed for 14- and 31-day periodic encodings, respectively. It is also worth noting that the mean loss over each epoch is often heavily influenced by a single data point, in some cases representing as much as 92% of the total loss - but the position of the highest loss changes between different runs of the same code, so there is no reason to believe

the problem is a few outliers in the data itself.

After noting that training is highly unstable, and likely approaching several different local minima, attempts were made to decrease the complexity of the network, by reducing the number of fully connected layer in the middle of the autoencoder, as well as reducing the size of the remaining layers. This did not notably change the results, but had a marginal improvement in computation time. The training histories before and after simplifying the model are displayed in Figure 11 for 14-day encoded data, and Figure 12 for 31-day encoded data. It appears to have had little notable impact on the 14-day data, but on the shorter sequences of 31-day summarized data, the more complex model appears to consistently outperform the simpler one. It is difficult to conclude exactly why this disparity happens, but noting that the learning curve on the longer 14-day encoded sequences is rather flat in both cases, it is certainly a possibility that neither model manages to learn the longer sequences, and the more complex one more effectively learns to encode the shorter sequences.

6.1.1 Complete Dataset

Once we felt confident that our methods were working as intended, we moved from using the preliminary 5% of the dataset to the full dataset. We set aside 20% of the customers as the validation set, and 10% for the test set, leaving 70% as the training set.

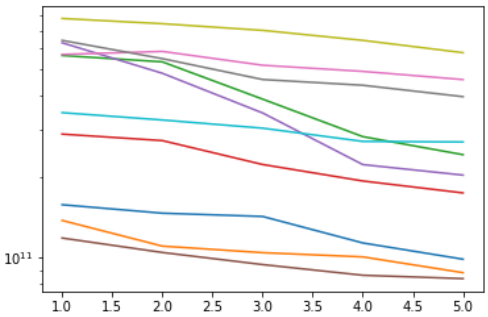
The increase in training data significantly increased time required to run our methods, but also made the unsupervised training appear more reliable, although there are still points on which it varies. The increase in training time does, however, make it harder to run several trials with the same configuration, and creating 10 models to create a single of the plots in Figure 11 would take more than a week if re-applied to the complete dataset, totalling almost a month to create plots corresponding to all of Figure 11 and Figure 12 for the full dataset.

6.1.2 Alternative Preprocessing

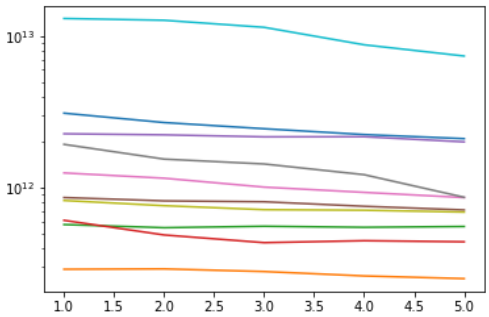
One key problem with the method we are using, summarizing transactions over fixed-length periods, is how we choose to represent these sums. Summarizing the data directly is a simple approach, but simple is not always best. We therefore propose an alternative way to summarize the transactions, in which the sequences contain data about how much money is spent in each category. While the transactions contain essentially the same information as before, the sums now have a much simpler real-world interpretation; the rows for each category all represent how much money is spent in that category, rather than the number of purchases made in the category. Like earlier, a logarithmic transform is applied to these sums, in order to keep the scale from blowing out of proportion.

Trying this new preprocessing, we trained new models. Before we get to their predictive performance in Section 6.2, we will look at how well they capture the patterns in the modified data. Initial trials indicate that a model trained with the alternative preprocessing captures much more information, reducing training loss by much more than the previous models could, indicative that this encoding of the data is more suitable for an autoencoder to read than the initial version - with 2 different configurations of varying complexity, whose training histories are displayed in Figure 13, seemingly outperforming what we have seen earlier in terms of learning progress. This is once again a reminder that the format we feed into the model can be as important as the architecture of the model itself.

The new preprocessing also changes the output of a Fourier feature extractor, as the fourier transform is applied to a differently processed data point. Running principal component analysis on the data in the frequency domain, the variance captured in the first 16 principal components is displayed in Figure 14 for the original preprocessing, and Figure 15 for alternative.



(a) Training history from 10 trials with equal configurations, using the more complex model. The x axis represents epoch, the y axis mean squared error, and differently colored graphs represent training histories for different trials.



(b) Training history from 10 trials with equal configurations, using the simplified model. The x axis represents epoch, the y axis mean squared error, and differently colored graphs represent training histories for different trials.

Figure 13: Unsupervised training history after applying the alternative preprocessing to the data. Due to the unstable results we got from 14-day encoded data earlier, as well as the time required, only 31-day periods were used in this case.

6.2 Supervised Training

After achieving somewhat stable unsupervised training results, one autoencoder with fitting hyperparameters was used for the prediction of certain target variables, unknown to the autoencoder model.

For the purpose of this thesis, the target variable used was whether the customer changed their address within the next 3 months. This was selected because it is one of the more common events that the bank monitors, and although it is still rather uncommon (with about 4% positives in the dataset), it is much more common than any other target variable of interest, and as such provides for less imbalance in the data.

6.2.0 Hyperparameter Tuning

Whilst random forest classification is a method which requires little hyperparameter tuning, gradient boosting classifiers can easily run the risk of over- and underfitting, so tuning hyperparameters is much more important. We use the validation data for model selection, so to avoid validation set overfitting, we use 5-fold cross-validation to find the optimal selection of hyperparameters.

Because each gradient boosting classifier has different variables to work with, they also need to be tuned according to the variables they have available, so the cross-validation is run on each gradient boosting model individually. This is a relatively heavy process, with a round of cross-validation taking up to a few hours — the exact time required being heavily dependent on how many variables the classifier has available. However, in the grand scheme of things, the extra hours required for cross-validation are not that much, and they are necessary in order to provide accurate estimates of how well the algorithm utilizes the different variables generated by the methods we proposed.

6.2.1 Aggregate-free Supervised Learning

As a benchmark for how much information the background variables by themselves contain, one model of each type was trained with only the background variables, utilizing none of the transactional data in the dataset. Their corresponding performances are listed as the baseline in Table 4.

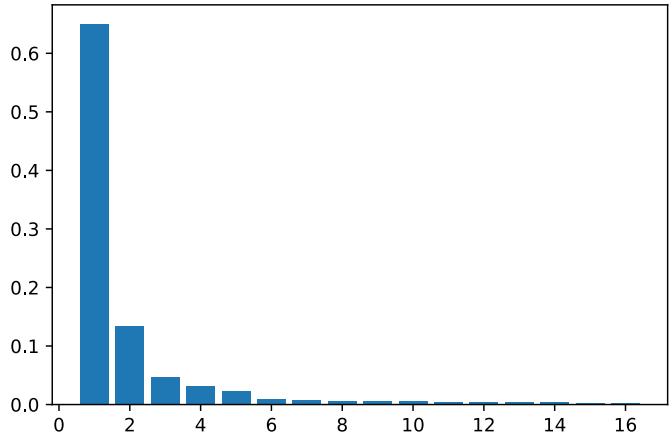


Figure 14: Proportion of variance captured by the first principal components, using PCA and a discrete Fourier transform on top of the original preprocessing. The first principal component captured 65% of the variance in the data, and the first 16 capture a total of 94%.

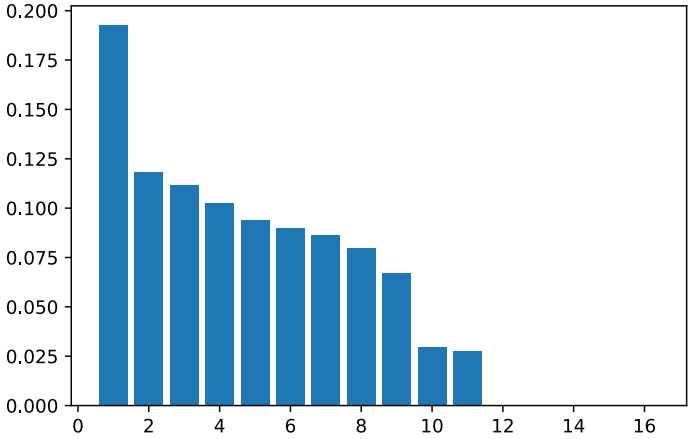


Figure 15: Proportion of variance captured by the first principal components, using PCA and a discrete Fourier transform on top of the alternative preprocessing, the first principal component captured 19% of the variance in the data, and the first 16 capture over 99%.

6.2.2 Autoencoder Aggregates

Using the methods described in Section 5, a fitted autoencoder creates automatic aggregates for each customer, which can be used in a supervised manner. Their training process is similar to that of the previous subsections, but using said automatic encodings. Once training is complete, running the autoencoder is a reasonably efficient process, taking only 30 milliseconds per customer using 31-day encoded data, totalling a transformation time of 3 minutes for the preliminary 5% of the data (and 1 hour for the full dataset).

These aggregates were then combined with background knowledge, and used to train various supervised models. Autoencoders were fit at different scales, creating between 16 and 128 variables, and different supervised learning methods were used to evaluate them.

6.2.3 Fourier Aggregates

Using a discrete Fourier transform, a given set of frequencies must be chosen for the resulting aggregates to be of equal dimension. For the sake of simplicity, our early attempts will be limiting themselves to using the lower ends of the spectrum, focusing primarily on longer term temporal patterns. Attempts are also made to apply principal component analysis to the Fourier transformed data, so that more general frequencies may be used, rather than only long-term patterns. The exact numbers of frequencies to choose are easy to tweak, and similarly to the latent dimension of the autoencoder, these processes will also be applied using several different numbers of aggregates.

Whilst the discrete Fourier transformation does not possess the expressive power of a neural network, it proved much easier to tweak, as there is no costly training period. There are also much fewer hyperparameters and architecture choices involved in producing this encoding of the data.

Applying the Fourier encoding to the customer base is also vastly faster, processing over 2000 customers per second on 14-day encoded data, making the computational cost nearly nothing, in addition to not requiring any training. Principal component analysis is also highly efficient, and takes a relatively small amount of time to apply on top of the earlier methods. As such, it could be worth using these methods even if the impact is very small.

The validation set results of using the autoencoder, Fourier, and Fourier-PCA aggregates are displayed in Table 4. The top performer, both for random forest and gradient boosting, was the autoencoder for 64 variables, with alternative preprocessing. More generally, all methods perform similarly when creating few variables, but when scaling up, the autoencoders using the alternative preprocessing outperform the other methods when scaling up, whereas the autoencoder on the original preprocessing performs poorly, particularly when scaling up.

The different Fourier encoders appear to have a small, positive impact on validation performance, but there is little change to the supervised performance when scaling them up to produce a higher number of variables.

6.2.4 Using Manual Aggregates

There are functionally infinite ways to manually define aggregates for sequences of arbitrary length. Being humans, the aggregates we use will typically be things that we would consider salient information, and that can be plausibly expected to have some predictive power. We will use varying numbers of variables, to see how well our models work compared to a human defining either a few or very many variables.

Our suggested manual aggregates primarily consist of spending in given categories over given time periods, e.g. a variable for spending on groceries in the last month, and one for the last year. These have the benefit of being easier to interpret and compute, but their more rigid structure could also result in less salient variables.

While these are manually defined, we will not be spending a lot of time defining each variable; for

		Gradient Boosting		Random Forest	
Variable count	Method	Cross-entropy	Accuracy	Cross-entropy	Accuracy
0	Baseline	1.3369	0.96129	1.4199	0.95888
16	Autoencoder orig.	1.3389	0.96124	1.3331	0.96140
	Autoencoder alt.	1.3350	0.96135	1.3389	0.96124
	Naive Fourier orig.	1.3369	0.96129	1.3466	0.96101
	Naive Fourier alt.	1.3369	0.96129	1.3369	0.96129
	Fourier PCA orig.	1.3350	0.96135	1.3389	0.96124
	Fourier PCA alt.	1.3369	0.96129	1.3427	0.96112
32	Autoencoder orig.	1.3427	0.96112	1.3350	0.96135
	Autoencoder alt.	1.3330	0.96140	1.3350	0.96135
	Naive Fourier orig.	1.3369	0.96129	1.3485	0.96096
	Naive Fourier alt.	1.3369	0.96129	1.3389	0.96124
	Fourier PCA orig.	1.3331	0.96140	1.3389	0.96124
	Fourier PCA alt.	1.3369	0.96129	1.3389	0.96124
64	Autoencoder orig.	1.3466	0.96101	1.3350	0.96124
	Autoencoder alt.	1.3273	0.96157	1.3254	0.96163
	Naive Fourier orig.	1.3369	0.96129	1.3369	0.96129
	Naive Fourier alt.	1.3369	0.96129	1.3369	0.96129
	Fourier PCA orig.	1.3447	0.96107	1.3389	0.96124
	Fourier PCA alt.	1.3408	0.96118	1.3389	0.96124
128	Autoencoder orig.	1.3466	0.96101	1.3312	0.96146
	Autoencoder alt.	1.3369	0.96129	1.3292	0.96151
	Naive Fourier orig.	1.3369	0.96129	1.3350	0.96135
	Naive Fourier alt.	1.3350	0.96135	1.3408	0.96118
	Fourier PCA orig.	1.3389	0.96124	1.3350	0.96135
	Fourier PCA alt.	1.3369	0.96129	1.3389	0.96124

Table 4: Supervised results from automatically generated variables, used in Gradient Boosting and Random Forest classification. Variable count is excluding the preexisting variables, i.e. the number of new variables added to the model. orig. and alt. signifies feature extraction based on the original and the alternative preprocessing, respectively.

example, when defining a variable for spending on groceries in the last year, it is a trivial task to define similar metrics from the other transaction categories (as described in Section 4).

The results of all trials using manually defined aggregates are described in Table 5.

6.2.5 Combining Methods

Until this point we have focused on trying to find a single way to extract useful features, but there is no guarantee that a single method alone is optimal. Therefore, we also attempted some mixed learners, using variables from multiple different methods - for example, running supervised learning on 64 autoencoder variables and 64 manually defined variables, or 64 autoencoder variables, 32 manually defined variables, and 32 fourier variables.

We also attempted PCA on the manually defined aggregates, as the variables we defined are likely highly correlated (e.g. someone who has a high salary is likely to have higher expenses as well). The variance contained in the first 16 principal components is shown in Figure 16.

This is a task that can be almost infinitely iterated upon - in addition to the near infinite ways to summarize a year, we now also have an abundance of different combinations that can be tried. The point of this project is not, however, to infinitely iterate using slightly different configurations every time, and for the sake of brevity, we tried to limit ourselves to using just a few mixed aggregates. From Table 4 we can see that the autoencoders perform best when producing 64 variables, so we will focus on autoencoder estimates from the 64-variable encoder. In Table 5 it appears that

		Gradient Boosting		Random Forest	
Variable count	Description	Cross-entropy	Accuracy	Cross-entropy	Accuracy
18	Monthly debit card summary	1.3369	0.96129	1.3852	0.95989
	Annual debit card summary	1.3369	0.96129	1.3369	0.96090
36	Monthly debit- and credit card summary	1.3369	0.96129	1.3871	0.95984
	Annual debit- and credit card summary	1.3369	0.96129	1.3369	0.96090
44	Monthly account summary	1.3427	0.96112	1.3254	0.96162
	Annual account summary	1.3177	0.96185	1.3196	0.96179
62	Monthly account and debit card summary	1.3369	0.96129	1.3775	0.96012
	Annual account and debit card summary	1.3273	0.96157	1.3254	0.96163
80	Monthly account, debit- and credit card summary	1.3350	0.96134	1.3775	0.96012
	Annual account, debit- and credit card summary	1.3273	0.96157	1.3196	0.96179
124	Annual and monthly account debit card summary	1.2964	0.96246	1.3022	0.96230

Table 5: Supervised results from using manually defined variables, used in Gradient Boosting and Random Forest classification. Variable count is excluding the preexisting variables, i.e. only counting the number of new variables added to the model. Due to the nature of the variables available, there were no natural ways to define e.g. exactly 64 variables to compare with the automatic aggregators. Although the debit and credit card tables are structurally similar, debit data has been preferred when selecting variables, as a large portion of the customer base do not have credit cards.

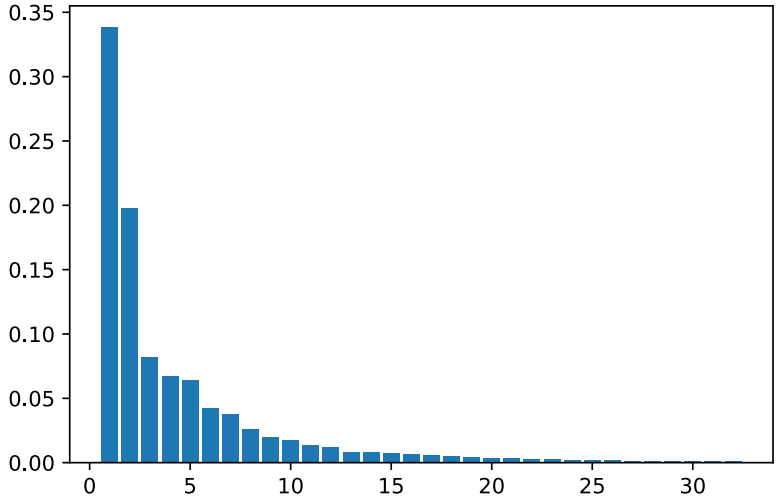


Figure 16: The proportion of variance contained in the first 32 principal components, when applying PCA to a set of manually defined aggregates. The first 16 contain 95% of the variance in the dataset.

the most salient variable combinations are a summary of annual account transactions, and the combination of annual and monthly account and debit card transactions. Finally, we will be using the first 16 principal components, which were computed on the basis of several manually defined aggregates, including account, debit and credit card spending in different categories in the last month, 3 months, and year. We found that these 16 principal components captured 95% of the variance in the data. The results of this final round of supervised evaluation is summarized in Table 6.

6.3 Test Results

Having trained several models and evaluated the different aggregates produced on the validation data, we compare two model performances on the test dataset. Specifically, we will be using the most promising models using only manually defined variables, as well as the most promising models including automatic aggregates, which are the ones including 124 manually defined variables and 64 automatic. These will both include the same 124 manually defined variables, as these were involved in the strongest performers in both Table 5 and Table 6, and the test results stand as a test of whether the autoencoder provides salient values to the supervised learning models.

As shown in Table 7, the additional variables did not impact the performance of the gradient boosting model at all, but it worsened the test performance of the random forest models.

		Gradient Boosting		Random Forest	
Variable count	Description	Cross-entropy	Accuracy	Cross-entropy	Accuracy
80	PCA of manual aggregates (16) and autoencoder (64)	1.3254	0.96163	1.3273	0.96157
108	Annual account summary (44) and autoencoder (64)	1.3138	0.96196	1.3196	0.96179
188	Monthly and annual account and debit summary (124) and autoencoder (64)	1.2945	0.96252	1.3099	0.96207

Table 6: Supervised results from combining manually defined variables with automatically generated ones, used in Gradient Boosting and Random Forest classification. Variable count is excluding the preexisting variables, i.e. only counting the number of new variables added to the model.

	Gradient Boosting		Random Forest	
Description	Cross-entropy	Accuracy	Cross-entropy	Accuracy
Manual aggregates	1.3078	0.96214	1.3309	0.96147
Mixed manual and automatic aggregates	1.3078	0.96214	1.3387	0.96124

Table 7: Test results comparing the best models using manually defined variables, to the best models using a mix of manually and automatically defined variables. The automatically defined variables do not appear to provide sufficiently salient information.

7 Discussion

The discussion section aims to discuss the choices that were made during the project, in terms of methods and the completion of the experiment, as well as discussing the results and issues we encountered.

7.1 Limitations

While we have seen some limited success applying our methods to real-world data, it should be noted that the proposed method of feature extraction is very much a "black box" approach — it uses inputs to produce outputs, but we have no reasonable way to interpret its reasoning.

For certain tasks, a level of explainability is required. For example, when evaluating credit scores, Norwegian banks are required to provide an explanation for their conclusion — and with humans in the other end, an explanation like "the third feature extracted from your transaction history through a series of matrix multiplications and activation functions has a value between 1.4 and 3.8" would not be satisfactory when explaining why you cannot get a credit card.

Because many of the workings of a bank are required to have some level of transparency — which a black box most certainly does not — the results achieved with these methods will be applicable only to the parts that do not have such restrictions, such as advertising.

7.2 Computation Cost

Due to the poor computational performance of recurrent neural networks, and the difficulties involved in computing them in parallell — particularly when sequences differ in length — it is likely not feasible to train and evaluate a sequential autoencoder very often.

A lot of the methods described in this paper involve heavy computations, and even the best performances we have seen have only provided marginally salient features. While the results indicate that the aggregated features contain some value, it is a different question entirely how much they are worth. From a practical standpoint, they must represent not only the slightest of improvements, but the value they provide must outweigh the cost of computing them. This aspect would speak more in favor of the fourier transform than the sequential autoencoder — as the computation of the fourier transform has proven to be 1000 times faster, even when disregarding the time it takes to train the autoencoder.

7.3 Periodic Summaries

In order to reduce the training and computation time of the autoencoders, we have summarized transactions over time periods of differing lengths, typically from 7 to 31 days. Before this simplification, training the autoencoder would take a full week to run a single epoch, and upwards of a month to complete a training of 5 epochs.

As valuable as the decrease in computation time is, it brings up an important new problem; by summarizing weeks and months before the autoencoder sees the data, we are training the machine learning model on a human-defined summary of the given time period — effectively re-introducing the problem we are trying to solve, albeit on a much smaller scale.

As shown in Section 6.2, using different ways to summarize the data can vastly impact the quality of the variables produced. Even when the algorithms used to extract features remained entirely unchanged, the difference between the original and the alternate preprocessing pipelines were massive, with the original preprocessing often resulting in variables that hurt the performance of the models.

Ideally, we would have a method that could extract features from pure transactional data within

a reasonable time frame, but with the methods described herein, that proved to be infeasible. If new recurrent architectures are developed, or hardware more optimized for them emerges, this could be looked in to, but until that happens, the sheer computation time involved — even before considering hyperparameter optimization — will remain a major obstacle.

7.4 Training, Validation and Test Sets

In traditional time series analysis, it is common to use different time periods as training, validation and test sets — typically with the training set chronologically first, followed by the validation set, with the test set at the end. This practice is typically adopted to assert whether historic data can be used, to some extent, to predict future events — and as such, ensuring that the findings are long-term patterns, rather than merely picking up on what is trending at the time. A key idea is that even though the model may recognize a short-term trend, it is much less useful if that information is not applicable to any other time period.

When training and evaluating our methods, the training, validation and test sets were selected as subsets of the customers, rather than periods of time. This was done deliberately. Among other reasons, it ensures that an encoding applies on a customer-level, and that the autoencoder has not seen any of the validation set — or test set — before running on them. Unlike ARIMA and ARCH-models, which are common in time series analysis, recurrent neural networks work on long term dependencies, and not just the latest few data points. This means that, although we are not certain of how large the impact would have been, there could be real overlap in the training, validation, and even test data — which is not ideal.

Furthermore, because the majority of the data is in the training set, it would result in the evaluation being performed on data that is fundamentally different — either slightly longer sequences, if including both training and validation data in validation, or much shorter, if only including the validation segment. In any case, the autoencoder model would be evaluated on data that was systematically different from what it was trained on. For use in a business, this would ideally not be an obstacle, as repeated training can be costly. As such, our results here should not be considered to be necessarily maximizing the business value of the model, but rather an evaluation of the autoencoder's ability to learn to aggregate features from the data in question.

We have no research regarding how much the autoencoder aggregates change over time, nor do we have the time available to train enough models to monitor such a statistic. Knowing might change the way one goes about training and fine-tuning a model — but although it is an interesting question, is not one that this project was ever meant to answer, and answering it is a different task altogether.

7.5 Model Architecture

As described earlier, we based our autoencoder on the architecture shown in Figure 10, using LSTM layers as encoders and decoders, as well as some fully connected layers between them to add flexibility. There are several other architectures possible, many of which we discarded due to time constraints, as well as running on limited hardware. Thus, there is likely room for improvement by changing the model architecture. This includes some simple changes, such as hyperparameter tuning and regularization, but also more structural differences, like trying different recurrent network architectures, such as the GRU, and even having deeper recurrent steps, using multiple recurrent networks following one another.

Once again, similar to what we discussed in Section 7.3, we run in to a fundamental problem of the task: whenever we define a model to extract salient features from the data, it will always, to some extent, be colored by the humans who construct the network architecture. Although we can add more flexibility, and try to make the models find patterns on their own, humans are always defining the architecture of the model, and therein we also constrain the patterns it might find[6]. A

[6]This is not to say a human must understand whatever pattern the machine learning model finds - merely that

different architecture may yield better results, or perhaps a different approach entirely is necessary in order to actually learn features from such transactional data. No matter what architecture one applies to the problem, there is no way to get away from the fact that a human will be involved in describing it. Fundamentally, we must accept this, before looking for ways to improve the feature extraction portion.

Finally, our model definition is rather naive. It makes no attempt to separate different variables, even though some variables are fundamentally different. One distinction that one can make, could be separating different pieces of transaction data into different types, to add more information to the model; if applied correctly, this would allow us, the humans, to describe more of the logical necessities as a priori information. For example, we could work with the following four variable groups, with different activation functions in the decoder (which is trying to reconstruct the data from the reduced dimensions): boolean (using a sigmoid activation), categorical (using softmax in the decoder), nonnegative (ReLU), and numerical (linear) variables. Furthermore, the loss function could also be different for separate variables, to optimize correctly for different kinds of data.

Adding more a priori knowledge to the model architecture would be a time-consuming endeavour, but the benefit could be substantial, as the model wouldn't need to spend time finding well-known logical patterns. If done correctly, it won't increase the model bias at all, while it could substantially reduce the variance, thus reducing the overall error.

7.6 Manual Aggregates in Training

When validating our automatic aggregates, they performed better as part of a larger ensemble, and the best performances on our validation set were achieved by mixing automatic aggregates with manual aggregates. This is still true for the test set, although the manual aggregates did yield the exact same metrics for their corresponding model, and the declining test performance of the random forest model might indicate that the automatic aggregates are not useful enough.

As part of our model architecture, we included some known variables, as an attempt to make the model learn variables unknown to us. However, because it was designed by itself, rather than as part of a larger ensemble, we did not take into account that other aggregates might be used alongside it. The value of the autoencoder, when used in an ensemble, could possibly increase by including the variables produced by the rest of the ensemble (the manual aggregates) in its training process, as part of the external features (as shown in Figure 9).

7.7 Evaluating an Unsupervised Solution

Although the feature extraction methods we have employed have been unsupervised, all the metrics we use to evaluate their importance have been measured in a supervised context. Whether the metric is cross-entropy or accuracy, there are two key issues to keep in mind.

First, different supervised learning methods function differently. While their goal is usually the same, their way to reach that goal varies, and they have different strengths and weaknesses. When we selected random forests and gradient boosting to evaluate our model, it was because they are robust methods, they can handle and — importantly — they are invariant to any strictly monotonous transform, which allows us to focus less on the scale on which our autoencoded data is presented to the algorithms. There is, however, no guarantee that tree-based methods are the ones that will get the most out of these variables, and it is quite likely that different machine learning methods would change the evaluation performance of the new variables.

Second, different target variables will yield different results. In our final experiments, we focused on the target variable of whether the person would change their address within the next 3 months, primarily because it is a less imbalanced target variable than the rest — as most of the events monitored by the bank are pretty uncommon. Ideally, we would like to evaluate the features based

humans define the equation that the algorithm tries to solve

on their pure quality, but the metrics we use are unequivocally connected to the target variables chosen.

Because of the structural imperfections in our metrics, the exact quality might need to be taken with a grain of salt. The impact of these imperfections could be reduced somewhat by running tests using more different supervised methods and more target variables, but they cannot be completely removed no matter how many trials are run, unless every feature extraction method is applied to every single supervised problem the bank has — which is clearly not feasible. Due to time and hardware constraints, and because most of the available target features were extremely imbalanced, we limited our experiments to including only one target variable and two supervised learning methods.

7.8 Unsupervised Loss as a Feature

An interesting feature of an autoencoder neural network is that they can, inherently, know how well they are doing on any given data point. Because the full network tries to match the input to the output, the network does not require a separate target variable when computing its loss, as the target variables are the same as the input. This also means that it can compute its own loss, which is an indication of how well it captures the essence of any one data point.

Due to time constraints, we do not know whether the loss value would be a salient feature. It is not the typical way to use an autoencoder, but it is possible that it could give other learners useful information. For example, a high loss value from the autoencoder might indicate that it is an unusual customer, or that the autoencoder simply does a poor job of summarizing their transactions.

There are, however, some significant drawbacks to proposing this. One is the computation time, as this would require using the entire autoencoder, as opposed to only the encoder half, effectively doubling the time required to encode anything. Another, perhaps even more severe, is the difference between seen and unseen data. Training data would, by the nature of the learning process, have lower loss values than validation and test, as it is already known to the model. As such, we could risk increasing the bias significantly by using this feature in training.

Until someone tries, we cannot say certainly how useful this feature would be, or whether the added information it provided would outweigh the possible increase in bias that might occur. Because the time we have is limited, there was not enough time to test this, but in the future, it could be interesting to look into.

8 Conclusion and Further Works

The conclusion section aims to summarize our findings, answer the research questions posed in Section 1, and discuss what future developments might improve further upon this project.

Bank transaction data is high-dimensional, highly variant and takes many different forms. Using autoencoder neural networks, and a discrete fourier transform, we have explored ways to utilize the abundance of transactional data that exists within the world of banking. We got some limited results. We found that our autoencoder could find features that were useful in the prediction task it was tested on, but no improvements were made when compared to hand-picking features.

8.1 Answering the Research Questions

Research Question 1 *What is the state of the art in aggregation of transactional data?*

For related issues, particularly natural language processing, the state of the art is using recurrent neural network architectures, often with unsupervised pretraining. In terms of banking, the state of the art is kept closer to the chest of the individual banks, but it is no secret that manually defined variables play a large part, because these are legally required for a lot of the areas in which they incorporate machine learning.

Research Question 2 *How well can a sequence autoencoder produce valuable input variables for Sparebank1 SMN's machine learning models?*

An issue we failed to predict was the shape of the transactional data. Even though the dataset contained several gigabytes of transactional data, which could be the equivalent of several hundred million sentences, these were distributed across just 100000 customers, each of which had long sequences of data attached to them — much longer than any meaningful sentence. That is to say, even though the dataset was large, it was more complex and had fewer samples than one might expect, given its size.

In our experiments, we had some moderate success, showing that a sequence autoencoder can provide variables that yield a positive impact on the validation performance. This did, however, prove to be very prone to the data fed into it, and applying different layers of preprocessing can drastically change the value of the variables that are used.

Research Question 3 *How can more traditional methods be used to produce inputs, and how do they compare to an autoencoder approach?*

More traditional methods are primarily feature engineering, manually defining what features we are interested in, e.g. spending on groceries in the last month. By itself, manually defined variables appeared to produce significantly better variables than the autoencoder, at least for the problem that we used to evaluate our features. When combining engineered features with features produced by an autoencoder, we saw a slight improvement in validation set performance (compared to only feature engineering). When they were finally tried on the test set, however, they yielded equal performances on their gradient boosting models, and the manually defined variables still had the strongest performance when applying random forest classification. As such, it appears that manual feature engineering is still more useful than automatic aggregates.

8.2 Further Works

Concluding this project, we have not solved everything related to the problem at hand, and so we would like to mention some of the ways forward, to continue the research in the field.

8.2.1 Computational Performance

There is more work to be done in improving the computational performance of the process. Some of this can likely be helped in part through parallellism, which could potentially allow significantly reducing the time required to use the autoencoder, both in training and execution. Other alternatives that can increase the speed can include other neural network architectures, as well as more specialized hardware for the exact task.

8.2.2 Using More of the Data

When defining our model, we selected what data to try to learn from and what to discard in preprocessing very early. While there is no guarantee that it will be important, it would be naive to assume that none of it bears any information of interest — it is simply more difficult information to extract. For example, transactions from one bank account to another contain a text field, which was discarded for the purpose of this task. It will, without a doubt, prove rather difficult to accurately reconstruct this data in the decoder, but using information from the text in each transaction could be something to look into, delving into the domain of natural language processing.

Furthermore, there was a lot of relational data that was also discarded, particularly account numbers. This goes primarily for accounts on the other side of the transaction — e.g. the person to whom a renter pays their rent. Because this is such an individual data point, we could not include it directly, but there may be ways to incorporate it partially — e.g. a flag for whether the customer has transferred money to the same account earlier, or a field describing how long it was since the last time a transaction was made between these accounts, or any other way to describe a preexisting relation between two accounts. It could also be worth including what kind of accounts are involved in a transactions, to the degree that a bank possesses this information — for example, a person spending money from their housing savings account (BSU) is likely in a different situation than one spending it from their current account, but with the data we had available, there is no way to tell the difference.

Perhaps more mundane, the currencies were not fully explored for their informative value. They were included indirectly, by including all transaction amounts as both local currency and NOK, but not directly. The sample size would likely be too small for most currencies, which is a large part of the reason this field was excluded, but it is possible that it could provide some information — for example, that some currencies are more indicative that a customer might be a frequent traveler.

8.2.3 Explainability

There are several challenges and drawbacks to the proposed methods, and explainability of any features produced, even by the fourier methods, are severely lacking. Ideally, a transactional aggregator such as the one proposed here should also extract features that are explainable even to humans, which could vastly improve the value of the aggregated features, because it would be legally applicable to many more use cases.

The target of this task was never explainability, and the methods used reflected it. The lack of explainability is not always a problem, but a set of features that can be explained can be used in many more cases, which would make them considerably more valuable from a business standpoint, as opposed to anything coming out of a black box.

The field of explainability in artificial intelligence (XAI) can likely bring a lot of value to society, and explainability in representation learning will likely yield some valuable insight in the future. Knowing more about what features a machine learning models can find can also help working against bias in AI, either through removing biased features directly, or ideally, ensuring that a model never produces them in the first place. Furthermore, from a societal standpoint, understanding the patterns that a machine sees can allow governments to take action more effectively, by finding

more clear ways to solve various problems.

Making an automatic, explainable feature-extractor will require a lot more progress in both the field of XAI and feature extraction, but if they do get made, they can provide businesses and governments with a lot of value.

Bibliography

Badsha, Md Bahadur et al. (2020). 'Imputation of single-cell gene expression with an autoencoder neural network'. In: *Quantitative Biology* 8.1, pp. 78–94.

Brown, Tom B. et al. (2020). 'Language Models are Few-Shot Learners'. In: *CoRR* abs/2005.14165. arXiv: 2005.14165. URL: https://arxiv.org/abs/2005.14165.

Choromanski, Krzysztof et al. (2020). 'Rethinking Attention with Performers'. In: *CoRR* abs/2009.14794. arXiv: 2009.14794. URL: https://arxiv.org/abs/2009.14794.

Cui, Wenqi, Weiwei Yang and Baosen Zhang (2021). *Predicting Power System Dynamics and Transients: A Frequency Domain Approach*. arXiv: 2111.01103 [eess.SY].

Dai, Andrew M. and Quoc V. Le (2015). *Semi-supervised Sequence Learning*. arXiv: 1511.01432 [cs.LG].

Fisher, Ronald A. (1936). 'The use of multiple measurements in taxonomic problems'. In: *Annals Eugenics* 7, pp. 179–188.

Goodfellow, Ian, Yoshua Bengio and Aaron Courville (2016). *Deep Learning*. http://www.deeplearningbook.org. MIT Press.

Heidari, Mohsen et al. (18–24 Jul 2021). 'Finding Relevant Information via a Discrete Fourier Expansion'. In: *Proceedings of the 38th International Conference on Machine Learning*. Ed. by Marina Meila and Tong Zhang. Vol. 139. Proceedings of Machine Learning Research. PMLR, pp. 4181–4191. URL: https://proceedings.mlr.press/v139/heidari21a.html.

International Organization for Standardization (Apr. 2003). *Retail financial services — Merchant category codes*. Standard. Geneva, CH: International Organization for Standardization.

James, G. et al. (2014). *An Introduction to Statistical Learning: with Applications in R*. Springer Texts in Statistics. Springer New York. ISBN: 9781461471370. URL: https://books.google.no/books?id=at1bmAEACAAJ.

LeCun, Yann et al. (1998). 'Efficient BackProp'. In: *Neural Networks: Tricks of the Trade*. Lecture Notes in Computer Science. Springer Berlin / Heidelberg. Chap. 2, p. 546. ISBN: 978-3-540-65311-0. DOI: 10.1007/3-540-49430-8_2. URL: https://dx.doi.org/10.1007/3-540-49430-8_2.

Peng, Jiajie, Jiaojiao Guan and Xuequn Shang (2019). 'Predicting Parkinson's Disease Genes Based on Node2vec and Autoencoder'. In: *Frontiers in Genetics* 10, p. 226. ISSN: 1664-8021. DOI: 10.3389/fgene.2019.00226. URL: https://www.frontiersin.org/article/10.3389/fgene.2019.00226.

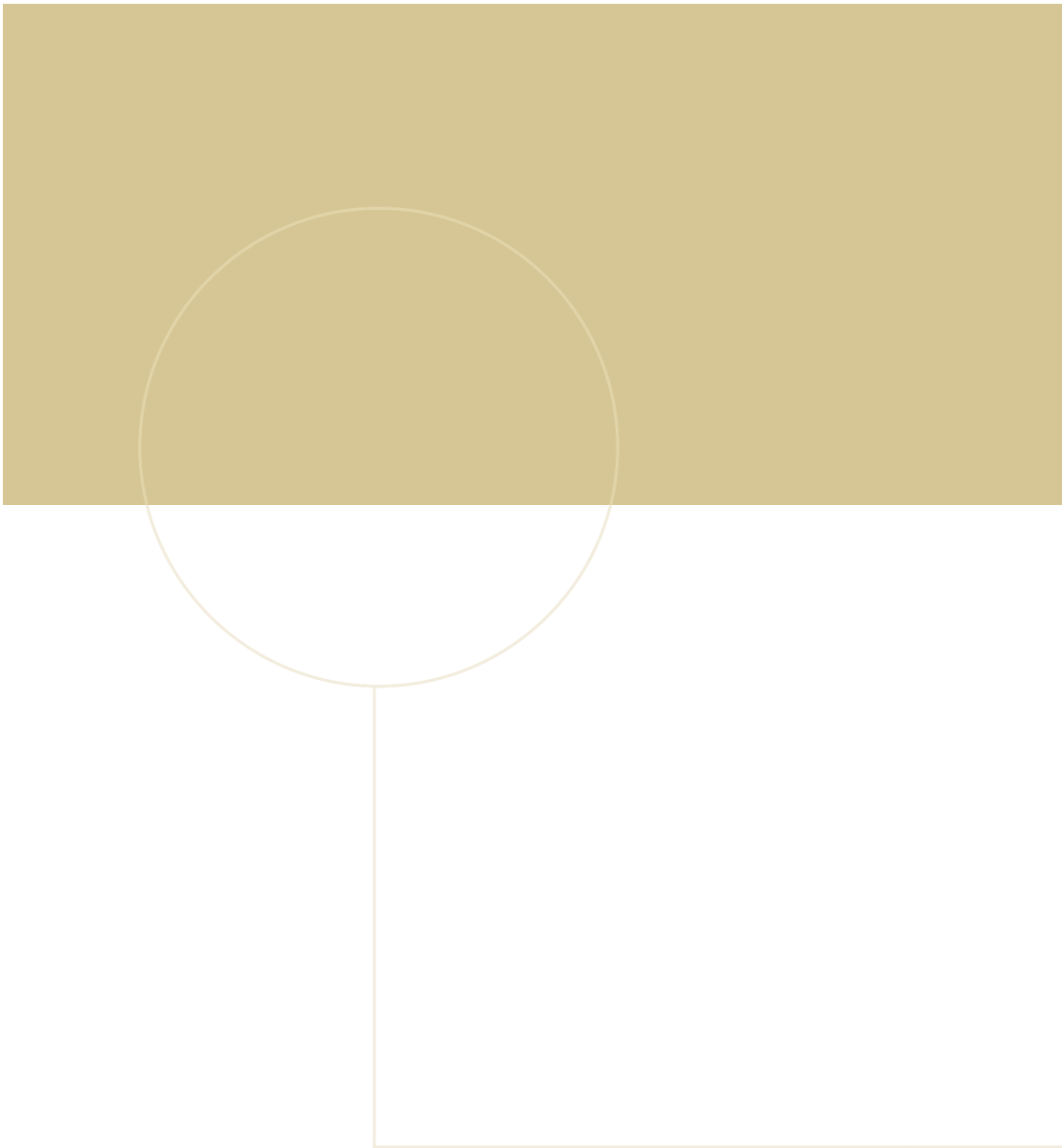
Ramasubramanian, Karthik and Abhishek Singh (2017). 'Feature Engineering'. In: *Machine Learning Using R*. Berkeley, CA: Apress, pp. 181–217. ISBN: 978-1-4842-2334-5. DOI: 10.1007/978-1-4842-2334-5_5. URL: https://doi.org/10.1007/978-1-4842-2334-5_5.

Sundararajan, D. (2001). *The Discrete Fourier Transform: Theory, Algorithms and Applications*. World Scientific. ISBN: 9789812810298. URL: https://books.google.no/books?id=54kTgFg5IVgC.

Vaswani, Ashish et al. (2017). *Attention Is All You Need*. arXiv: 1706.03762 [cs.CL].

Xie, Rui et al. (2017). 'A deep auto-encoder model for gene expression prediction'. In: *BMC genomics* 18.9, pp. 39–49.

Xu, Kai et al. (2020). *Learning in the Frequency Domain*. arXiv: 2002.12416 [cs.CV].



NTNU

Norwegian University of
Science and Technology