Using deep learning to estimate the probability of a credit-card applicant defaulting

Internship report

Eklavya Sharma

June 2018

Abstract

Credit card companies use estimates of risk to profitably provide credit. This internship project aims to explore the viability of deep neural networks for estimating the probability of a credit-card applicant defaulting. The data has both static and time- varying components, so MLPs, CNNs and hybrid architectures were explored. Furthermore, the data is semi-structured, so a custom architecture had to be used.

Contents

Chapter 1

Introduction

Credit card companies do not give credit cards to all applicants. This is because giving a credit card to a customer introduces risk for them. If the customer uses the credit card for purchases and doesn't pay her credit card bills, the company will suffer a loss.

Therefore, credit cards are given to only to low-risk applicants, that is, those applicants who have a low probability of defaulting on those cards. Credit card companies obtain data about an applicant from various sources. Sources include the application form, credit bureau, etc. They then use that data to estimate the probability of the applicant defaulting. This estimation is generally done using machine learning models.

Recent advances in Deep Learning, a branch of Machine Learning, have led to major accomplishments in various data-science tasks. This project aims to explore the viability of deep neural networks for estimating the probability of a credit-card applicant defaulting.

The credit card applicant data has static as well as time-varying components to it. This project explores Deep Neural network architectures like Multi-Layer Perceptrons (MLPs) and 1-dimensional Convolutional Neural Networks (CNNs). CNNs are known to perform well on time-series classification tasks. The data is also semi-structured, which means that common neural network architectures cannot be applied as-is. A way of dealing with semi-structured data was also developed as part of this project.

The main reason for using deep learning over other machine learning models is that deep learning internally uses automatic feature creation. The current production model, which we aim to beat, is an ensemble of decision trees. Due to the semi-structured nature of the data, raw data cannot be fed to the decision trees. Therefore, manual feature engineering is used to bring the data into a structured form. This has two disadvantages. First, manually creating features limits the number of features that can be created and checked for relevance. The space of created features is also limited to features which are interpretable by humans. Secondly, extensive feature creation requires a team of skilled domain experts, which is expensive to hire and maintain.

Chapter 2

Methodology

This internship project had several phases:

- Getting familiar with machine learning and learning about neural networks, especially Convolutional Neural Networks (CNNs).
- Understanding the meaning of all attributes of data.
- Preprocessing data.
- Training and evaluating neural network models on the data.

This document assumes that the reader knows the basics of machine learning and knows about neural networks, including CNNs.

2.1 Training

We were told to read about machine learning and neural networks. We revised Multi-layer perceptrons and then studied Convolutional Neural Networks (CNNs) and a bit about Long Short-Term Memory (LSTM) Neural Networks. Stanford University has made its course notes on 'Convolution Neural Networks for Visual Recognition' [6] available online. Christopher Olah has an informative article on RNNs and LSTMs [4].

Along with that, we were also given sessions about our company. Some of them were general, like what our company does, what values we are supposed to follow, what is expected from employees, etc. Some sessions were specific, like sessions on card economics, risk (both card member risk and merchant risk), customer life-cycle, security policies, etc.

After that our project was finalized and we were given data. The data had to first be cleaned and brought into an appropriate form for processing by neural networks. Based on the kind of data we had, it was decided that I would use Apache Spark DataFrames for data preprocessing.

2.2 Technologies used

2.2.1 Apache Spark

Apache Spark [7] is an open-source cluster-computing framework. PySpark is a python API for using Apache Spark.

Apache Spark is based on Resilient Distributed Dataset (RDD), a read-only multiset of data items distributed over a cluster of machines, that is maintained in a fault-tolerant way.

RDDs can be created by reading data from a data source or by creating them from other RDDs. RDDs creation is lazy, so when an operation is called on an RDD, the operation specification is stored but not actually applied on the RDD. This not only makes it possible to instantiate RDDs in a streaming fashion, but also recover from data loss.

Spark SQL [2] is a component build on top of Apache Spark. Spark SQL exposes DataFrames, which are built on top of RDDs. A DataFrame is like an RDD, but uses an optimization engine (which is similar to SQL query execution engines in database systems) to optimize operations on DataFrames before running them.

2.2.2 JupyterLab

JupyterLab (formerly known as IPython) [5] is a web-based integrated development environment. 'Jupyter Notebooks' is one of the features of JupyterLab. A Jupyter notebook is a document which contains both code and rich-text elements. Jupyter notebooks allow one to run code and see intermediate outputs. They are very good for data exploration.

The instance of JupyterLab which I used supported Python 3 as the kernel. It also has a lot of data analysis packages including PySpark, Pandas, NumPy and Matplotlib.

2.2.3 TensorFlow

TensorFlow [1] is a python and C++ library for neural networks and computational graphs. In this project, TensorFlow was used as a backend by Keras, so it was only used to create and run computational graphs.

2.2.4 Keras

Keras [3] is a neural networks library which uses a computational graph library as a backend.

2.3 Data description

Each credit-card applicant is identified using an identifier called PCN. Credit bureau (a data source) provides data on each financial instrument (also called 'trade') that an applicant is associated with. Examples of trades include credit cards, loans and leases. This data is called trade-level data. Since the number of trades can be different for different applicants, data is semi-structured.

Apart from data on trades of an applicant, there is some data available about the applicant itself, which includes data from the application form. This data is called applicant-level data.

The data is of past customers, and we know whether those customers defaulted on their credit cards. Therefore, data labels are available for each applicant, which can be used to train machine learning models.

Types of attributes in trade-level data:

- Date attributes
- Nominal attributes
- Real-valued attributes
- A time-series attribute

Variables have varying fill rates. Some of them have a fill rate of 100%.

Data is large enough to not fit in the main memory of most personal computers.

The data was partitioned into 3 parts - training, validation and testing. Training partition is used for training models. Validation partition is used for hyper-parameter tuning and continuous monitoring of performance. Testing partition is used for finally evaluating a model.

2.4 Preprocessing

Preprocessing data is an important step to ensure that data is clean and in the correct form which learning algorithms expect them to be in.

Some preprocessing steps, namely one-hot-encoding and time-series preprocessing, drastically increase the size of data. The data would then take around 100 times more space. Current storage size and access speeds would preclude such preprocessing. Therefore, these steps are done online just before feeding them to neural networks.

Here is a list of data cleaning and preparation steps taken during the project:

2.4.1 Remove corrupt rows

Some rows failed basic consistency checks. Such rows were removed.

Some variables had a fill rate of more than 99.99%. It was found out that some rows had null values for some important variables. Therefore, we deemed such rows as corrupt and removed them.

2.4.2 Use consistent representation of null

Null values were represented using various strings like the empty string, `'null'`, `'missing'`, `'UNK'`, `'unknown'`, etc. I replaced all such instances by a consistent representation of null.

2.4.3 Reduce the number of nominal values

There are 2 ways in which this is done:

- Collect all values which have a very low occurrence and set all of them to a single value (like `'other'`).
- Group related classes together into larger classes. This is very time-consuming and requires understanding the data very well. I did this for only one nominal attribute.

This is needed because categorical values have to be one-hot encoded before feeding to a neural network. If the number of classes in each categorical variable is large, the dimensionality will become very high. A very large dimensionality introduces problems in effectively and efficiently training machine learning models. This is called 'curse of dimensionality'. For neural networks, a high dimensionality means increased number of parameters, which makes training more difficult and time-consuming.

2.4.4 Normalize numeric variables

If neural network weights are initialized randomly, very large values and very small values can cause the exploding gradients problem and the vanishing gradients problem respectively. Therefore, all numeric variables were normalized to be roughly between -1 and 1.

Firstly, a distribution of values was plotted for each variable (cumulative quantile on x-axis and value on y-axis). It was observed that all variables were highly skewed. Each variable was replaced by its logarithm. This made their distribution more uniform.

After that, the variables were linearly transformed to fit the -1 to 1 range. For most variables, this was done using the Z-transform. The Z-transform includes subtracting the mean and then dividing by the standard deviation. For variables which had many outliers, an linear mapping was generated by hand by observing the plot of the distribution of the variable.

2.4.5 Create boolean attributes for numeric variable nullness

A boolean variable was created for some numeric variables indicating whether that variable was null or not.

2.4.6 Fill missing values in numeric variables

All missing values in numeric variables were filled with 0. This is roughly equivalent to replacing null values by the mean of the variable, since we have scaled all numeric variables to roughly the range -1 to 1. Exactly replacing by mean is not as good because mean is affected by outliers.

2.4.7 Normalize dates

Dates were converted to 'number of months since January 2000'.

2.4.8 Explode time-series (online)

The time-series variable is represented as a string. This form is not suitable for feeding a neural network. The strings were padded and converted to series of boolean attributes.

2.4.9 One-hot encode categorical attributes (online)

For categorical variable x and each value v which that categorical variable can take, create a boolean column x_v which is set to value 1 if x equals v and 0 otherwise.

2.5 Neural Network input format

There are 2 components of the data — static and temporal. Temporal data consists of data represented as a time-series. All time series are of the same length n_{months}. The rest of the data, that is, non-temporal data, is called static data.

Data is fed to neural networks (both during training and during prediction) in batches. Each batch consists of n_{pcns} applicants. Each applicant consists of n_{trades} trades. (During preprocessing, applicants were grouped on the basis of number of trades such that each batch consists of applicants with the same number of trades). Different batches can have different values of n_{pcns} and n_{trades}.

Static data is represented as a 3-dimensional array of shape $(n_{\text{pcns}}, n_{\text{trades}}, n_{\text{svars}})$. Temporal data is represented as a 4-dimensional array of shape $(n_{\text{pcns}}, n_{\text{trades}}, n_{\text{months}}, n_{\text{tvars}})$. n_{svars}, n_{tvars} and n_{months} don't vary across different batches.

Although data passed to a neural network is in batches, the architecture of the neural network is oblivious to the batch size. Therefore, for simplicity of notation, from now on we will omit mentioning the first dimension of the data, which is the batch size. So, static part will be represented as a 2-dimensional array of shape $(n_{\text{trades}}, n_{\text{svars}})$ and the temporal part will be represented as a 3-dimensional array of shape $(n_{\text{trades}}, n_{\text{months}}, n_{\text{tvars}})$.

2.6 Neural Network Architectures

Activation function in all neurons of all neural networks is ReLU, except the activation function in the last neuron. The last neuron outputs a scalar and uses sigmoid as the activation function.

Multiple neural network architectures were considered. Since shortcomings of one architecture motivated the use of another architecture, I'll describe all architectures which were considered, even those which weren't instantiated and trained.

2.6.1 Accepting static and temporal data together

Suppose we have 2 inputs: static input of shape (α) and temporal input of shape (t, β). Suppose we want to build a neural network which works on these inputs and produces a scalar output.

If we only consider the static input, we can use an MLP to get a scalar output. If we only consider the temporal input, we can use a 1-dimensional CNN to get a scalar output.

More work needs to be done if we want the neural network to accept both static and temporal data. One option is to duplicate the static part along an additional axis to get data of shape (t, α). This way there will be no variation of the data along the first axis. This part can then be concatenated with the temporal part along the last axis. The whole thing can then be passed in a CNN. This option is not good because it will blow up the size of scalar data by a factor of t, which can be prohibitively large.

Another option is to pass static data through an MLP to get output of shape (γ), pass temporal data through a CNN to get output of shape (δ), concatenate them to get output of shape $(\gamma + \delta)$ and then pass that through an MLP to get a scalar output. This option has a limitation that the CNN can't take the static part into consideration. But we hope that the CNN will learn features which will be informative enough after they are combined with the first MLP's output.

In the next subsections, we discuss strategies of dealing with semi-structuredness of data, that is, how to deal with an additional dimension of size n_{trades}.

2.6.2 Strategy 0

Fix the number of trades to a constant n_{trades}. If an applicant has less trades than this, add more trades with default values. If an applicant has more trades than this, ignore the extra trades. Probably ignoring the least recent trades would be best.

Now reshape (or rearrange) the data. The static part should be 1-dimensional of shape $(n_{\text{trades}} n_{\text{svars}})$ and the temporal part should be 2-dimensional with shape $(n_{\text{months}}, n_{\text{trades}} n_{\text{tvars}})$. We can now construct a neural network which accepts these parts using options discussed in the previous subsection (subsection 2.6.1).

The current production model aggregates data along the trade and time dimension, so it is losing information before it is passed to the learning algorithm (ensemble of decision trees). However, this neural network strategy doesn't suffer from that since it considers data in the rawest form possible.

However, this strategy isn't able to take variable trades into account.

There is some loss of information for applicants who have more than n_{trades} trades. Also, since most people have very few trades, a significant amount of bloat is generated.

Another problem with this approach is redundancy. There could be traits of a trade which indicate defaulting. However, the neural network will have different parameters corresponding to the same variable across different trades. Not only does this unnecessarily increase the number of parameters, it also makes learning difficult since a good value will have to be learned for all these parameters separately and the occurrence of the trait will be spread across different trade indexes. The order of trades will also begin to matter and the first few trades will be weighted more strongly than the last few trades.

Because of the shortcomings of this strategy, we didn't even try it.

2.6.3 Strategy 1

This strategy tries to extract features from each trade, aggregate those features across trades and then learn from those aggregated features.

The neural network architecture is organized sequentially in 3 slices:

- Slice A: Apply a neural network to each trade. The neural network's input shape is $[(n_{\text{svars}}), (n_{\text{months}}, n_{\text{tvars}})]$. The neural network uses approaches from subsection 2.6.1 to handle a mix of static and temporal inputs. The weights of the neural network will be shared across trades. In Keras, this can be done using the `TimeDistributed` wrapper. The output of this slice has shape $(n_{\text{trades}}, \alpha)$.

- Slice B: This slice aggregates across trades. A simple aggregation is an associative function like sum, max, min, etc. Applying a simple aggregation on the input of shape $(n_{\text{trades}}, \alpha)$ gives output of shape (α). A complex aggregation is a concatenation of multiple simple aggregation operation. For example, a complex aggregation which concatenates the results of sum and max produces an output of shape (2α).

- Slice C: This slice is just an MLP with a scalar output.

This approaches fixes all the shortcomings of Strategy 0. However, this approach doesn't capture temporal interactions between trades.

2.6.4 Strategy 2

This strategy also looks at pairs (and optionally triplets and quadruplets and so on) of trades in addition to just looking at individual trades. This has an advantage that temporal interactions between trades can be captured.

However, for an applicant with m trades, the number of trade pairs is $m(m-1)/2$. For our data, the average number of trade pairs is much more than the average number of trades. Therefore, training a neural network with trade pairs will take a lot more time. This is why this approach hasn't been tried yet.

2.7 Results

There are 2 kinds of cards — lending and charge. Since these kinds of cards are very different from each other, models are trained separately for each of them.

We use a metric called gini to evaluate model performance. Gini can be defined as 2 times ROCAUC minus 1. The performance of our neural network models is presented via a metric called 'relative gini'. The relative gini of a model on a dataset is defined as the gini of the model on that dataset divided by the gini of the production model on the testing dataset.

We show the results of 2 of our neural network models trained for 10 epochs.

Table 2.1: Relative gini of Strategy 1 model with both inputs

	Lending	Charge
Training	97.90 %	107.57 %
Validation	96.68 %	103.86 %
Testing	99.47 %	99.62 %

Table 2.2: Relative gini of Strategy 1 model with static input only

	Lending	Charge
Training	97.51 %	106.37 %
Validation	96.45 %	103.42 %
Testing	99.43 %	99.56 %

It can be seen that our models perform almost as good as the current production models. Since there isn't much difference between training, validation and testing scores, the model is not overfitting. It is also evident that including temporal data hardly contributes anything to model performance.

Chapter 3

Conclusion

A neural network was trained to estimate the probability that a credit-card applicant would default. Data had to be preprocessed before it could be fed into the neural network. Preprocessing involved several steps because the data was composed of different types of attributes. The neural network has a novel architecture because the data is semi-structured. Neural networks were found to be roughly as good as current production models.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark SQL: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1383–1394, New York, NY, USA, 2015. ACM.

[3] François Chollet et al. Keras. `https://keras.io`, 2015.

[4] Christopher Olah. Understanding lstm networks. `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`, 2015.

[5] Fernando Pérez and Brian E. Granger. IPython: a system for interactive scientific computing. *Computing in Science and Engineering*, 9(3):21–29, May 2007.

[6] Stanford University. CS231n: Convolutional neural networks for visual recognition. `http://cs231n.github.io/`, Spring 2017.

[7] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram

Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache spark: A unified engine for big data processing. *Commun. ACM*, 59(11):56–65, October 2016.