Erasmus University Rotterdam

Erasmus School of Economics

Bachelor Thesis [Business Analytics & Quantitative Marketing]

Default prediction with machine learning on peer-to-peer lending data

Author

Johnny Z.K. Cao (448 645)

Supervisor

Utku Karaca

Second assessor

Oktay Karabag

July 2, 2021

Abstract

With the rise of social lending platforms and their distinct role within the financial market, the demand for predictive machine learning models is high. The ability to distinguish loan borrowers based on their high chance to default is the most essential aspect of these classification models. This paper evaluates 5 models: k-nearest neighbours (K-NN), logistic regression (LR), support vector machines (SVM), random forest (RF) and feedforward neural network (FFNN) on their predictive power using a imbalanced data set from Lending Club. Furthermore, an approach to implementing neural networks on peer-to-peer lending is proposed and feature importance is investigated. The evaluation metrics show that RF outperforms other models but is unable to correctly predict the majority of bad borrowers. A FFNN with 3 hidden layers significantly improves in this aspect but suffers a great reduction in overall accuracy.

Contents

1 Introduction

Through social lending, sometimes referred to as peer-to-peer (P2P) lending, users can get access to funding without the support of traditional financial institutions. Although this alternative to official monetary services offers distinct features that are more appealing to its users (e.g. more potential for mutual benefit through competitive interest rates) the primary objective of the lender and borrower remains the same. In general the objective of both parties are as follows: the borrower seeks a loan with the lowest interest rate and the lender offers his or her financial backing with an interest rate that would, among other things, cover the risk of defaulting. Although defaulting on a loan is a scenario all parties would like to avoid, it is nonetheless a unavoidable misfortune that happens on these social lending platforms. As such, the ability to distinguish users that have a significant higher probability to default is invaluable for both the social lending platform as for investors trying to minimize the risk of their investments.

Predicting defaults or generally speaking risk assessment has been an field of interest for financial institutions merely because of the fundamental nature of investing, riskier investments and loans reward higher returns. One of the pioneering work done in this field is the modern portfolio theory developed by Markowitz, 1952, in his paper *"Portfolio selection"*. In terms of research on risk assessment for social lending platforms there have been several papers that focused on analyzing defaults on P2P loans, specifically on outperforming standard credit scores such as FICO scores or the inherent borrower grade given by the lending platform. As each social lending platform implements their own distinct model to predict the default rate and with it the corresponding interest rate, models that better predict these rates with more consistency have always been in high demand. Carmichael, 2014, and Malekipirbazari and Aksakalli, 2015, both attempt to provide more accurate default predictions by applying a discrete-time hazard model and a random forest classification model (Ho, 1995) respectively with positive results that outperform the grade given by the social lending platform, Lending Club (LC).

In this research five types of machine learning classifiers will be evaluated on their predictive power regarding borrower default rates. K-nearest neighbor is the first and simplest classifier. Logistic regression is a widely popular classifier used for credit scoring classification. Support vector machines has been applied on credit score data sets with positive results (Han et al., 2013). Random forests classification is an ensemble machine learning classifier that improves upon popular classifiers by implementing stochastic discrimination (Kleinberg, 1990). Lastly, a feedforward neural network will be implemented to predict borrower default rates. The focus of this research is to present and evaluate multiple models in order to improve classification within the context

of social lending platforms and to provide a approach for implementing deep learning on social lending data. Evaluation will primarily be focused on the accuracy of the classifiers, their ability to correctly predict bad borrowers and individual feature importance.

2 Relevant literature

Related studies have been primarily implemented on traditional consumer loans granted by official financial institutions as social lending platforms have been a recent development. One of the earlier work done in this field has been Berger and Gleisner, 2009, which aimed at investigating the role of P2P lending platforms within the financial market. The findings have indicated that these social lending platforms have improved borrower's credit scores by removing traditional financial intermediaries and thus reducing the information asymmetry between private borrowers and lenders. A study that heavily influenced this research has been Malekipirbazari and Aksakalli, 2015, who implemented 4 similar machine learning classifiers to attempt to outperform LC grades and traditional FICO credit scores regarding predicting bad borrower reputation. Their results indicate that random forests, with the help of non-standard financial features, improve the reliability of predictions compared to other machine learning classifiers.

In recent years neural networks have been implemented in many fields for classifications tasks due to its flexibility. As stated before many of these studies have been done on traditional consumer loans, yet the results have been positive. Zurada and Zurada, 2011 have shown that artificial neural networks in combination with traditional credit scoring models (ensemble models) implemented on data from credit unions can, in practice, yield better results regarding identifying bad loans. These positive results carry over when applied to P2P loans, works like Byanjankar et al., 2015 have shown that even a simple 1-layer neural network can capture the particular features of the social lending platform *Bondora* effectively and provide a reliable classification of borrowers into default and non-default groups.

3 Lending Club & Data

Before delving deeper into the data set the workings of LC have to be briefly mentioned to get a better understanding of the features of the data set. Users on this platform can create a loan listing if they meet the selection criteria of LC. The interest rate attached to each loan is solely determined by LC, based on its own credit grade. Once the loan is listed on the LC platform

potential lenders can decide to partially or fully fund the loan. All lenders have access to the borrower's information (personal information, financial information and the reason for the loan) and assume the risk of defaults. A loan can only be granted if the entire amount can be covered by (multiple) lenders before the expiration date. The borrower has either a 36-month or 60-month time window to pay back the loan with interest. Finally, any time the lenders receive a payment from the borrower, LC collects a service fee from the lenders.

The data set used in this study to predict the borrower loan status is obtained from the LC website Lendingclub.com. It contains roughly 350.000 loan observations for the period between January 2012 and September 2014. Out of the hundred of thousands of observations about 68.000 are loans that were successfully paid back or defaulted. Earlier research done with this identical data set includes *"Risk assessment in social lending via random forests"* by Malekipirbazari and Aksakalli, 2015. The 15 features used for training the models set can be categorized into 2 groups: numerical and nominal features. To reduce the effect of exponentially distributed features a logarithmic transformation has been applied. Additionally all 12 numerical features are normalized to have a unit variance and a zero mean. For the nominal features to be implemented in the models dummy variables have been made for each category. A total of 30 features will be used to train all the models. The dependent variable *loan_status_bin* is a binary variable with value 1, indicating a good borrower loan status and 0 for a bad borrower loan status. The data set contains 14012 (20.5%) loans with a default status indicating a moderately imbalanced data set. A full list containing description for each feature can be found in the appendix A: Feature list.

4 Methodology

As this data set is imbalanced the possibility that our trained classifiers classify every new data point as the majority class is high. To compensate for this problem a weighted cost matrix will be implemented when training these models. A 5-to-1 cost ratio will be implemented in the training phase. Thus misclassifications of bad borrowers will be 5 times more costly than misclassifications of good borrowers.

4.1 K-nearest neighbours

One of the simplest classifiers is k-nearest neighbors (K-NN) developed by Fix and Hodges, 1951. This non-parametric classification method is a supervised machine learning algorithm that categorizes new data points by its k nearest neighbours. The simplicity of this classifier comes from its

straightforward training phase as it only needs to store the feature vector and its corresponding class label from the training set, $\mathbf{x}^{tr} \in \mathbf{X}_{tr}$ and $\mathbf{y}^{tr} \in \mathbf{Y}_{tr}$ respectively. Afterwards, a new data point from the testing set $\mathbf{x}^* \notin \mathbf{X}_{tr}$, is classified by the majority vote of its closest data points. In this research it holds $k = 1$ thus the classification is only dependent on its closest neighbour. The distance of $(\mathbf{x}^*, \mathbf{x}^{tr})$ is obtained by calculating the total euclidean distance in p dimensions, where m is the number of features. The distance function is denoted by the following equation.

$$dist(\mathbf{x}^*, \mathbf{x}^{tr}) = \sum_{j=1}^{m} \sqrt{(x_j^* - x_j^{tr})^2} \tag{1}$$

The simplicity of this classifier however comes with several drawbacks. As the classification makes use of euclidean distances in higher dimensions the curse of dimensionality can affect the prediction power of the classifier. To go over the curse of dimensionality briefly, as the amount of dimensions increases, the volume of empty space increases exponentially making the available data sparse. In this situation the nearest neighbor and the farthest neighbour are not significantly different (Beyer et al., 1997). Furthermore, this classifier is heavily dependent on the distribution of the dependent variable of the data. Training on a data set where there are few data points of one type of class tends to result in poor predictions when testing new data points of the similar class. All things considered this classifier may still provide good predictive power and can be considered as a good baseline when evaluating more advanced classifiers.

4.2 Logistic regression

When dealing with binary dependent variables traditional linear classifiers that implement linear regressions are not applicable. This is where logistic regression (LR) can be implemented to guarantee an outcome between 0 and 1, which can be interpreted as a likelihood prediction. Assuming the dependent variable y follows a Bernoulli distribution ($y \sim Bin(p)$), the odds are defined as $P(y = 1) = p$ divided by $P(y = 0) = 1 - p$. Taking the logistic function of this provides an outcome between 0 and 1:

$$\begin{aligned} \mathbf{P}(\mathbf{y}) &= log(\frac{p}{1-p}) = \beta' \mathbf{x} \\ \mathbf{P}(\mathbf{y}) &= \frac{e^{\beta' \mathbf{x}}}{1 + e^{\beta' \mathbf{x}}} \end{aligned} \tag{2}$$

Where β represents a vector of regression coefficients. Note that LR still assumes a linear relation-

ship between the features as shown by $\beta'\mathbf{x} = \beta_1 x_1 + \beta_2 x_2 + ... + \beta_m x_m$. Training the LR model is done using maximum likelihood estimation to obtain the optimal coefficients, $\hat{\beta}$. The function which will be maximized is the following log likelihood function:

$$N^{-1} logL(\beta|Y;X) = N^{-1} \sum_{i=1}^{N} logP(y_i|x_i;\beta) \tag{3}$$

Classifying new data points is furthermore done by having a cut-off point of 0.5. Hence predictions where it holds $\hat{y_i} \geq 0.5$ will be classified as good borrowers ($\hat{y_i} = 1$) and as bad borrower ($\hat{y_i} = 0$) otherwise when calculating the evaluation metrics.

4.3 Support vector machines

Support vector machines (SVM) tries to find a optimal hyperplane in m dimensions such that the data points of different classes are well separated, the margin width between the hyperplane and the closest data points (support vectors) are maximized. Classifying data points is done according to the following equation:

$$y = \begin{cases} 0, & w'x + b < 0 \\ 1, & w'x + b \geq 0 \end{cases} \tag{4}$$

Training a SVM for a binary dependent variable can be formulated as a constrained optimization problem which looks as follows:

$$\begin{aligned} \min_{w,b} \quad & \frac{1}{2}\|w\|^2 \\ \text{s.t.} \quad & y_i(w'x + b) \geq 1 \qquad i = 1, .., N \end{aligned} \tag{5}$$

By using Lagrange optimization and incorporating the constraints, multiplied by a Lagrange multiplier α we get the following Lagrangian function:

$$\mathcal{L}(w, b, \alpha) = \frac{1}{2} w'w - \sum_{i=1}^{N} \alpha_i (y_i(w'x + b) - 1) \tag{6}$$

solving the original optimization problem corresponds to solving its dual problem:

$$\begin{aligned} \max_{\alpha} \quad & \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{k=1}^{N} \alpha_i \alpha_k y_i y_k (x_i' x_k) \\ \text{s.t.} \quad & \alpha' y = 0 \\ & \alpha_i \geq 0 \quad i = 1, .., N \end{aligned} \tag{7}$$

Where it holds $w = \sum_{i=1}^{\mathbf{N}} \alpha_i y_i x_i$ and $b = \frac{1}{N} \sum_{i=1}^{N} y_i - w'x$ as a result of the stationary constraints. As this optimal hyperplane can still be extremely difficult to find in practice due to the fact that the data may not be linearly separable in the original feature space a soft margin will be implemented which incorporates potential errors in its objective function. Furthermore a kernel function (Aizerman et al., 1964) can be added to the optimization problem to transform the feature space from a lower dimension space to a higher dimension space to fit a hyperplane with a maximum margin that will better separate data points based on their class types. The final constrained optimization problem will look as follows:

$$\begin{aligned} \max_{\alpha} \quad & \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{k=1}^{N} \alpha_i \alpha_k y_i y_k K(x_i' x_k) \\ \text{s.t.} \quad & \alpha' y = 0 \\ & 0 \leq \alpha_i \leq C \quad i = 1, .., N \end{aligned} \tag{8}$$

In this research a value of 1 for the regularization parameter C and a quadratic polynomial kernel function $K(x_i' x_k) = (1 + x_i' x_k)^2$ will be chosen for the SVM classifier.

4.4 Random forest

A decision tree tries to predict the class of a new data point by going through nodes, which represent binary assessments of a single feature, and will ultimately be classified by the last node of the tree, the leaf node. Like K-NN this supervised machine learning method is widely popular thanks to its simplicity, both in terms of implementation and interpretation of the method. However, the predictive power of these decision trees for a new group of data, especially when it doesn't resemble the training set, can be highly inaccurate. Further shortcomings are the likelihood of overfitting and non-robustness when working with a highly varied data set.

Unlike decision trees, random forests (RF) is a ensemble machine learning method first proposed by Ho, 1995. RF makes use of a technique called bagging which entails that it will bootstrap the data set, generate multiple decision trees and then combine all the results to improve

the accuracy of the classifier. As RF still utilizes decision trees some inherent issues still have to be addressed when building these multiple decision trees. Namely, which target feature should be assessed in each node split and how many layers each individual tree should have until the leaf node. RF further adds problems regarding forest size and random feature selection. As the calibration of these parameters will be crucial for a robust classifier, building a accurate RF will consist of the following steps.

Step 1: **Bagging** (Breiman, 1996). Given $b = 1, ..., B$, where B represents the number of decisions trees for the RF, choose at random N samples out of the training set X_{tr} and Y_{tr} which will make up X_b and Y_b. A starting value of $B = 10$ will be chosen. Note that duplicate samples within X_b and Y_b are allowed.

Step 2: **Training phase**. Train distinct decision trees $T_1, ..., T_B$ on the corresponding data sets $X_1, ..., X_B$ and $Y_1, ..., Y_B$. As RF makes use of feature bagging to decorrelate individual trees not all variables will be considered when choosing the optimal target variable for a node. Consider for a single tree at node E, v randomly chosen variables from the feature vector of size p. The suggested value for v is shown to be $log_2(m)$ (Breiman, 2001), thus $v = 5$. The node split is then based on the Gini impurity of these v variables. The Gini impurity calculates the probability of incorrectly classifying a randomly chosen data point at node E (y_E) after splitting with one of the v features. Thus the Gini impurity when splitting with $x_j \in X_b$ at node E with child nodes E_1 and E_2 is calculated as follows:

$$G_{E_1}(x_j) = 1 - \mathbf{P}(y_{E_1} = good\ borrower)^2 - \mathbf{P}(y_{E_1} = bad\ borrower)^2 \tag{9}$$

$$G_{E_2}(x_j) = 1 - \mathbf{P}(y_{E_2} = good\ borrower)^2 - \mathbf{P}(y_{E_2} = bad\ borrower)^2 \tag{10}$$

$$G_E(x_j) = \frac{N_{E_1}}{N_E} G_{E_1}(x_j) + \frac{N_{E_2}}{N_E} G_{E_2}(x_j) \tag{11}$$

Where N_{E_1}, N_{E_2} and N_E represent the amount of data points at their respective node. Finally the feature that provides the smallest Gini impurity value is chosen:

$$x^* = argmin_x G_E(x) = argmin_x \frac{N_{E_1}}{N_E} G_{E_1}(x) + \frac{N_{E_2}}{N_E} G_{E_2}(x) \tag{12}$$

The optimal value for the depth (d) will provide the highest accuracy given a starting value $B = 10$. Note that node splitting and searching for the optimal depth will be done simultaneously multiple times, this can be quite time computationally expensive.

Step 3: **Fine-tuning parameters**. Although the forest size $B = 10$ is chosen at the start of the algorithm, another value of B could lead to a higher accuracy. Evaluation metrics like the root mean squared error (RMSE) could provide further insight when fine tuning the forest

size. Given a new found value for B, step 1-3 are repeated until no significant increase in accuracy and RMSE can be found.

Step 4: **Testing phase**. Gather the evaluation metrics of the RF by predicting the borrower loan reputation of the testing set X_{test} where classification is done by taking the average predicted value of all decision trees:

$$\hat{y} = \begin{cases} 1, & \frac{1}{B}\sum_{i=1}^{B} T_i(x) \geq 0.5 \\ 0, & otherwise \end{cases} \tag{13}$$

Implementation of RF will be done using *sklearn.ensemble.RandomForestClassifier* where additional parameters such as *min_samples_split* will be estimated using a cross-validated gridsearch.

4.5 Feedforward neural network

Artificial neural networks are models that attempt to emulate the biological process of the human brain. Similar to how a brain consists of neurons that communicate with each other, an artificial neural network consists of nodes that are interconnected with each other and feed each other with either input or output information. These nodes are generally grouped within 3 layers: input layer, hidden layer and output layer.

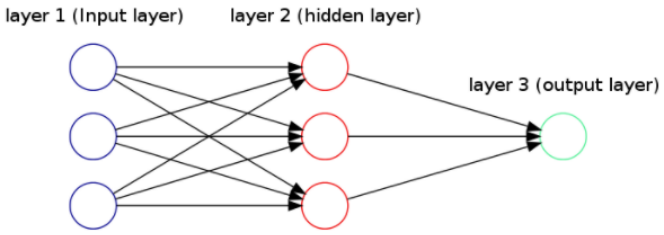

Figure 1: Feedforward neural network with 1 hidden layer

In a feedforward neural network (FFNN) all the output of the nodes are directed in one direction, towards the output node which ultimately predicts the class. As information goes from the input nodes to the subsequent nodes in the hidden layer, weights are attributed to the output and are summed. This weighted sum is often referred to as the activation, often a additional bias value is added to this sum to improve the overall network. In this research a sigmoid activation function

will be implemented which is expressed as follows:

$$a_A = \sigma_A(\mathbf{z}) = \frac{1}{1 + e^{-\mathbf{z}}} \tag{14}$$

Where a_A represents the output value of node A and $\mathbf{z}$ the linear combination of the input features and their corresponding weights ($\mathbf{z} = w_{A,1}x_1 + w_{A,2}x_2 + ... + w_{A,q}x_q$) for Node A. This process of combining the output of previous nodes with weights and receiving an activation value between 0 and 1 continues until the output layer. The output node finally determines the class for which it has the highest probability.

Training a neural network will ultimately lead to optimal values for the weights and biases. The training process primarily consists of minimizing the loss function of the neural network. Each data point from the training set passed through the network will result in some prediction error as the output will be an real value ranging from 0 to 1 while the true class will be an binary value. In this research a binary cross-entropy loss function will be implemented which according to *Deep Learning (p.226)* by Goodfellow et al., 2016 is shown to improve the performance of binary classification models. The cross-entropy loss function also referred to as the logarithmic loss function looks as follows:

$$L = -y_i log(\hat{y}_i) - (1 - y_i) log(1 - \hat{y}_i) \tag{15}$$

Where L represents the loss, log the natural logarithm, $\hat{y}$ the predicted value and y the true value. After the loss function is calculated a technique called backpropagation (Rumelhart et al., 1986) is implemented. Backpropagation computes the gradient of the loss function with respect to the weights and tries to find values for which the loss function is minimum. It does this by implementing the chain rule backwards, starting from the output layer. Training a FFNN with a backpropagation algorithm (Rojas, 1996) works as follows:

Consider the following notations:

- node r corresponds to the output node which ultimately makes the classification.
- $a_j^{[k]}$ represents the output value of node j in layer k. Note that the following holds $a^{[0]} = \mathbf{x}$ and $a_r = \hat{y}$.
- Furthermore note that $a_j^{[k]} = \sigma_j^{[k]}(\mathbf{z}) = \frac{1}{1+e^{-\mathbf{z}}}$, $\mathbf{z} = \sum_{i=1}^{q} w_i^{[k-1]} a_i^{[k-1]}$ where $a^{[k-1]}$ and $w^{[k-1]}$ represent vector of inputs and weights from the previous layer coming into node j.

Step 1: **Forward phase**. Starting off with random values for weights (w) and biases (b). After passing a batch size of $N^* = 32$ through the network, the total loss will be calculated:

$$L = -\sum_{i=1}^{N^*} y_i log(\hat{y}_i) - (1 - y_i) log(1 - \hat{y}_i) \tag{16}$$

Step 2: **Backward phase**. Minimize the loss function with regards to the neural network's weights.

$$\min_w L(w) = \min_w -\sum_{i=1}^{N^*} y_i log(\hat{y}_i) - (1 - y_i) log(1 - \hat{y}_i) \tag{17}$$

The following holds for the first derivatives with respect to the weights $w^{[k]}$ in the first layer (k) connected to the output node:

$$\begin{aligned}
\frac{\partial L}{\partial w^{[k]}} &= \frac{\partial L}{\partial \sigma^{[k]}} \frac{\partial \sigma^{[k]}}{\partial z^{[k]}} \frac{\partial z^{[k]}}{\partial w^{[k-1]}} \\
\frac{\partial z^{[k]}}{\partial w^{[k-1]}} &= a^{[k-1]} \\
\frac{\partial \sigma^{[k]}}{\partial z^{[k]}} &= \sigma^{[k]}(\mathbf{z})(1 - \sigma^{[k]}(\mathbf{z})) \\
\frac{\partial L}{\partial \sigma^{[k]}} &= \frac{\sigma^{[k]}(\mathbf{z}) - y}{\sigma^{[k]}(\mathbf{z})(1 - \sigma^{[k]}(\mathbf{z}))}
\end{aligned} \tag{18}$$

Thus the derivative with respect to these weight looks as follows:

$$\frac{\partial L}{\partial w^{[k]}} = (\sigma^{[k]}(\mathbf{z}) - y) a^{[k-1]} \tag{19}$$

The first derivative of the loss function with respect to the weights corresponding to node j in the hidden layers ($w_j^{[l]}$) are:

$$\frac{\partial L}{\partial w_j^{[l]}} = \frac{\partial L}{\partial \sigma_j^{[l]}} \frac{\partial \sigma_j^{[l]}}{\partial z_j^{[l]}} \frac{\partial z_j^{[l]}}{\partial \sigma^{[l-1]}} \frac{\partial \sigma^{[l-1]}}{\partial z^{[l-1]}} \tag{20}$$

It holds for the derivative of the loss function with respect to the vector of weights of node i before layer k:

$$\frac{\partial L}{\partial w_i^{[k-1]}} = \frac{\partial L}{\partial \sigma_i^{[k]}} \frac{\partial \sigma_i^{[k]}}{\partial z_i^{[k]}} \frac{\partial z_i^{[k]}}{\partial \sigma^{[k-1]}} \frac{\partial \sigma^{[k-1]}}{\partial z^{[l-1]}} = \frac{\partial L}{\partial w^{[k]}} \frac{\partial \sigma^{[k-1]}}{\partial z^{[k-1]}} = ((\sigma^{[k]}(\mathbf{z}) - y) a^{[k-1]}) a^{[k-2]} \tag{21}$$

Note how the loss function derivative makes use of the calculation made a layer before them, hence the name backpropagation.

Step 3: **Update the weights**. The weights are updated using stochastic gradient descent with the following function:

$$w_j^* = w_j - \alpha \frac{\partial L}{\partial w_j} \tag{22}$$

Where α represents the learning rate. Step 1-3 will be repeated with all batches of the training set where the weights will be constantly adjusted to minimize the loss function.

After having found the optimal weights and biases through backpropagation the final evaluation of the FFNN will be similar to RF where all necessary evaluation metrics of the classifier are gathered from the testing set. Implementation of the FFNN will be done through *Keras* where the number of hidden layers, the amount of nodes within each layer and the learning rate will be estimated through *Keras Tuner* API. The *Keras Tuner* API provides a reliable approach to deciding the optimal node structure given a additional objective alongside loss minimization. Given a training- and validation split of the data set of 80%/20% the optimal node structure will be chosen according to the best objective score. In this research this objective score will be the AUC score of the validation set. Finally the amount of training cycles (epochs), which could further alleviate the problem of overfitting, will be selected by investigating the metrics of the training- and validation set at various epoch values.

4.6 Methods of evaluation

A 5-fold cross-validation (CV) will be implemented to get the evaluation of the following metrics. As shown by Huang et al., 2007, this machine learning approach brings a good trade-off between underfitting and overfiting the model. Thus the data set will be sliced into 5 parts. 4 parts will be used for training the model and estimating the optimal parameters and the remaining part will be used to obtain the evaluation metrics. This will be repeated for each individual part. The final evaluation scores will be the average of all five testing parts.

4.6.1 Confusion matrix

The confusion matrix is a popular table layout used to visualize the performance of a supervised classifier on the testing set.

y \ $\hat{y}$	1	0
1	True positive (TP)	False negative (FN)
0	False positive (FP)	True negative (TN)

The table above shows such instance of a confusion matrix with its corresponding terminology in each cell. After the testing phase of each classifier the following metrics will be gathered:

$$\begin{aligned} Accuracy &= \frac{TP + TN}{TP + FP + FN + TN} \\ True\ positve\ rate\ (TPR) &= \frac{TP}{TP + FN} \\ False\ Positive\ rate\ (FPR) &= \frac{FP}{FP + TN} \end{aligned} \tag{23}$$

The interpretation of the *accuracy* metric is self-explanatory. TPR measures the proportion of good borrowers identified correctly whereas FPR measures the proportion of bad borrowers incorrectly. In this study a high TPR and a low FPR would indicate a great classifier.

4.6.2 Area Under the Curve

The curve this metric is referring to is the receiver operating characteristic (ROC) curve. This curve plots the true positive rate (TPR) against the false positive rate (FPR) at various classification threshold values other than 0.5. The AUC metric measures the area under this curve.

The ROC curve could be interpreted as the classifier's trade-off between TPR and FPR at various classification thresholds. Likewise we can interpret AUC as follows: AUC measures the classifier's ability to more likely classify a random individual as a good borrower than a bad one. Ultimately, a high AUC score is desirable. Note that the interpretation of this metric should not be associated with accuracy and both metrics should be taken into account when evaluating a binary classifier, especially when dealing with imbalanced data sets.

4.6.3 Root Mean Square Error

As each classifier makes misclassifications the difference between the predicted value and the real value can be measured with the RMSE metric:

$$RMSE(y, \hat{y}) = \frac{1}{N} \sum_{i=1}^{N} \sqrt{(y_y - \hat{y}_i)^2} \tag{24}$$

The interpretation of this metric is how well the model is able to fit the data points of the test set. A RMSE value of 0 would thus mean a perfect fit.

4.7 Measuring feature importance

To measure the importance of a individual feature according to a model I implement the permutation feature importance technique. This model inspection method tries to measure the decrease in the classifier's accuracy when the values within the feature column are randomly shuffled. This shuffling of values will ensure that the original relationship with the dependent variable is broken. Ultimately a greater reduction in accuracy will represent a relative larger feature importance.

5 Results

In this section I will show the results for the parameter optimization process and how the optimal parameters are chosen for RF and FFNN. Afterwards I will go over the evaluation metrics for all five classifiers and assess their performance on the LC data set. Lastly I will analyze the feature importance according to each classifier.

5.1 Parameter selection

5.1.1 Random forest parameters

This section explains how the hyperparameters of RF have been chosen. As stated before in section 4.4, starting with a forest size of $B = 10$ multiple RF have been built with various tree depths with the first training- and testing set. As shown in figure 2 with the $forest\ size = 10$ line, the performance of RF seems to flatten at a tree depth of 22.

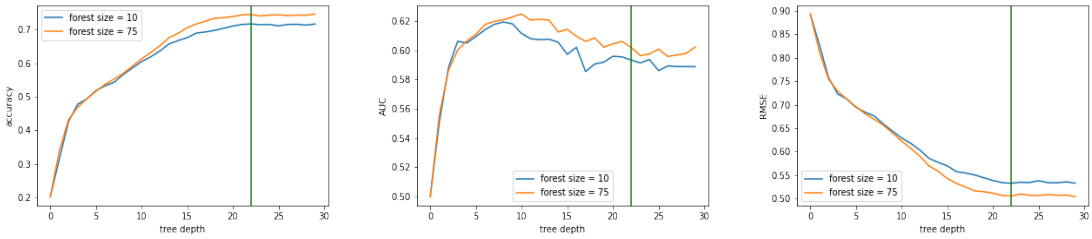

Figure 2: RF performance metrics with respect to varying forest sizes

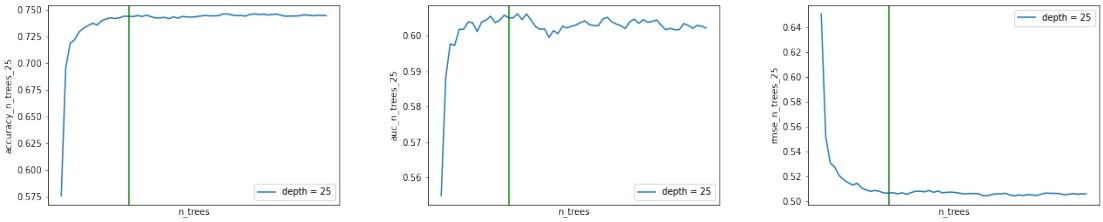

Figure 3: RF performance metrics with respect to varying tree depths

Next we replicate the same process but now varying the forest size from 1 to 300 in increments of 5 and having a fixed tree depth of 22. In a similar style, RF performance seems to flatten at a forest size of 75. Further investigations to other values of the tree depth with a fixed forest size of 75 lead to no significant improve of the RF classifier as shown in figure 2. Ultimately I settled on a tree depth of 22 and a forest size of 75.

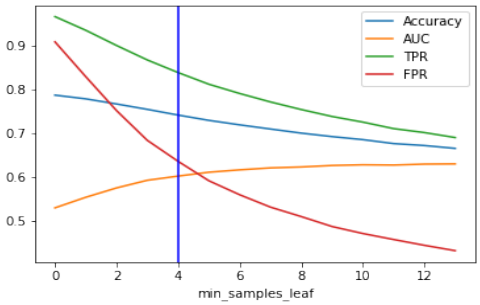

Figure 4: Min. samples leaf

Next is the *min_samples_leaf* parameter which represents the minimum number of data points required to be at the leaf node. As decision trees have a tendency to overfit the data this parameter serves to alleviate this problem. As you would expects, by increasing this value the model loses its predictive power regarding accuracy but in return the FPR decreases implying the model is able to better predict bad borrowers. Ultimately a value of 4 is chosen for this parameter.

5.1.2 Feedforward neural network parameters

As the problem of default prediction combined with a imbalanced data can easily lead to overfitting, parameters corresponding to a shallow neural network (2 to 4 hidden layers) have been chosen for the *Keras Tuner*. The code and a list of the best 10 scoring models can be found Appendix: Python code and Appendix: Keras Tuner results. Interestingly enough, the majority of the top 10 scoring models indicate that a learning rate of 0.01 provides the most stable training process. The model that presented the highest AUC score for the validation set seems the be a node structure with 3 hidden layers with 21 nodes in the first layer, 13 in the second layer and 17 in the final hidden layer.

Next we investigate the training- and validation metrics at various epochs for the FFNN classifier with aforementioned 3 hidden layer structure.

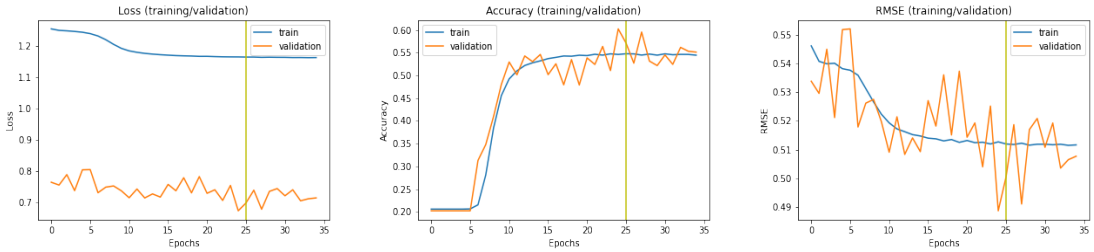

Figure 5: FFNN performance metrics with respect to varying epochs

As shown by the figure 5, increasing the amount of epochs, thereby allowing the model to cycle through the data set multiple times, improves the FFNN training scores but eventually flattens at around a epoch value of 25. Concerning the validation scores, FFNN shows quite peculiar behaviour especially with regards to the loss metric. These fluctuations could possible be explained by having a misspeficied FFNN. However, when dismissing the variations of the results for the validation set there are no strong signs of overfitting when comparing it to the training set scores. Evaluation of FFNN will be done with a value of 25 for the epoch parameter.

5.2 Classifiers evaluation

In this section I will evaluate the five classifiers on the LC data based on their average evaluation metrics .

	K-NN	LR	SVM	RF	FFNN
Accuracy	70.19%	53.18%	65.52%	75.44%	56.76%
AUC	0.533	0.619	0.512	0.592	0.623
RMSE	0.546	0.684	0.587	0.496	0.657
TPR	0.819	0.471	0.755	0.867	0.529
FPR	0.753	0.233	0.732	0.683	0.284

Table 1: Evaluation metrics

As seen in table 1, RF seems to outperform all models in all metrics except FPR and AUC. Only LR and FFNN are able to predict bad borrowers more consistently but sacrifice a significant portion of accuracy to do so as shown by having more than 18% less accuracy and the highest RMSE scores. Both K-NN and SVM seem unable to distinguish bad borrowers from good borrowers as these 2 models have the lowest AUC score and the highest FPR. The relative high accuracy of K-NN can be contributed to the imbalanced data set. The 3 hidden layer structure of FFNN and LR seem to be quite comparable as both models are able to correctly predict more than 70% of the bad borrowers. It can however be argued that FFNN is an improvement over LR as both the accuracy and RMSE scores are superior. It can thus be concluded that the bagging technique of RF is quite effective on the LC data set and out of the 5 classifiers seems to be the overall best classifier. But in situations where avoiding bad borrowers is relatively more important than finding good borrowers, FFNN is preferred.

5.3 Feature importance

In this section I will evaluate the feature importance according to all models. Table 2 displays the accuracy differences of several features when implementing the permutation importance method.

	KNN	LR	SVM	RF	FFNN
inc/payment ratio	-0.20%	0.30%	-0.20%	-0.30%	-0.35%
debt/income ratio	0.10%	1.00%	0.90%	-0.30%	0.57%
annual income	0.20%	0.30%	-0.70%	-1.60%	0.18%
revol. utilization rate	-0.20%	-0.10%	-0.20%	-0.80%	-0.12%
total accounts	0.00%	0.90%	0.00%	-0.60%	0.56%
loan amount	0.00%	0.10%	-0.60%	-1.20%	0.24%
open accounts	-0.40%	0.00%	0.60%	-0.20%	0.45%
purpose: credit card	0.10%	0.40%	0.20%	0.30%	0.42%
60-month term	-0.30%	0.90%	0.00%	-0.50%	0.51%
inquiries	-0.30%	0.30%	0.50%	0.40%	0.88%

Table 2: Feature importance (loss in accuracy)

The reductions in accuracy in table 2 are shown to be quite small indicating that when predicting the borrower's loan status, permutating a single feature isn't enough to significantly reduce the accuracy of the model. Additionally, there are signs that by permutating certain features the models gain accuracy, as indicated by the negative percentages. The randomness of the shuffling could be a cause of this unexpected increase. The increase in accuracy is especially evident with RF. One explanation explanation could be that RF determines its feature importance differently, namely on how nodes splits are being made: these node splits are based on the decrease in Gini impurity. Thus a more reliable way to measure the feature importance according to RF is by looking at the impurity-based feature importance.

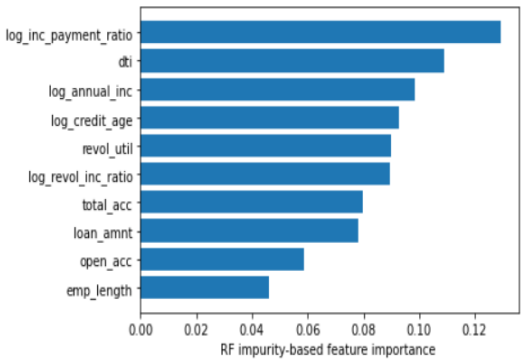

Figure 6: Feature importance according to RF (impurity-based)

All things considered the following conclusions can be drawn regarding feature importance. *Debt/income ratio* seems to be a consistently important feature across all models and should thus be considered essential when predicting the borrower's loan status. Other features

that are also frequently important are *total accounts* and *purpose: credit card*. Lastly, LR and FFNN display matching decreases and increases, with the exception of *inc/payment ratio*, when permutating features. A possible explanation could be that the weights associated with each feature of both models are combined into a sigmoid function thereby making their feature importance comparable.

6 Conclusion

In this paper I evaluated five types of machine learning classifiers on their predictive power regarding borrower loan status and investigated the feature importance according to each classifier. The five machine learning classifiers (K-NN, LR, SVM, RF and FFNN) can be used to further distinguish good and bad borrowers on social lending platforms and help private investors avoid bad loan borrowers that have a significant higher chance to default. To alleviate the problem of a moderately imbalanced data set of LC a weighted cost matrix has been implemented when training each model. Hyperparameter tuning for the FFNN is done with the *Keras Tuner* API with the objective to find a neural network structure which gives the best AUC score for the validation set. A 3 layer structure of $21 - 13 - 17$ nodes is found to be the optimal node structure for this data set. Evaluation metrics have been gathered with the help of a 5-fold cross-validation to reduce the possibility of over- and underfitting. The evaluation scores show that RF outperforms all models but is unable to correctly predict the majority of bad borrowers. In contrast, LR and FFNN are able to predict bad borrowers significantly better but trade in a large amount of accuracy to do so. Both K-NN and SVM seem unable to distinguish good and bad borrowers consistently and rely on the imbalanced data set to achieve their relative high accuracy. The most important feature according to all five models is *debt/income ratio* as shown by consistent reduction in accuracy when implementing the permutation feature importance technique.

7 Discussion

When dealing with machine learning models, fine-tuning the (hyper)parameters combined with the computationally expensive training processes are always some of the biggest limitations. The five models in this paper could all benefit from more parameter settings, more iterations and longer run times when implementing grid searches and the *Keras Tuner* API but doing so would increase the run time exponentially.

For future research, I suggest to implement different variants of the RF and FFNN models. The *extremely randomized trees* is a variant of RF which computes the node splits at random. Theoretically this would reduce the variance of the model. Possible extensions for the FFNN are countless as this machine learning model is highly flexible and thus remarkably customizable. One of the most straightforward extension is the addition of dropout layers. These layers have the characteristic of deactivating random nodes during the training process. The effect of this dropout layer is comparable to an ensembling technique as the dropout of nodes results in distinct neural networks during the training phase.

References

Aizerman, M. A., Braverman, E. A., & Rozonoer, L. (1964). Theoretical foundations of the potential function method in pattern recognition learning. *Automation and Remote Control,* (25), 821–837.

Berger, S. C., & Gleisner, F. (2009). Emergence of financial intermediaries in electronic markets: The case of online p2p lending [urn:nbn:de:0009-20-19400]. *BuR - Business Research*, *2*(1), 39–65. https://doi.org/10.1007/BF03343528

Beyer, K., Goldstein, J., Ramakrishnan, R., & Shaft, U. (1997). When is "nearest neighbor" meaningful? *ICDT 1999. LNCS*, *1540*.

Breiman, L. (1996). Bagging predictors. *Mach. Learn.*, *24*(2), 123–140. https://doi.org/10.1023/A:1018054314350

Breiman, L. (2001). Machine learning, volume 45, number 1 - springerlink. *Machine Learning*, *45*, 5–32. https://doi.org/10.1023/A:1010933404324

Byanjankar, A., Heikkilä, M., & Mezei, J. (2015). Predicting credit risk in peer-to-peer lending: A neural network approach. *2015 IEEE Symposium Series on Computational Intelligence*, 719–725. https://doi.org/10.1109/SSCI.2015.109

Carmichael, D. (2014). Modeling default for peer-to-peer loans. https://papers.ssrn.com/sol3/papers.cfm?abstract_id=2529240

Fix, E., & Hodges, J. L. (1951). Discriminatory analysis: Nonparametric discrimination: Consistency properties. *Randolph Field, Tex. : USAF School of Aviation Medicine*. https://doi.org/10.1037/e471672008-001

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning* [http://www.deeplearningbook.org]. MIT Press.

Han, L., Han, L., & Zhao, H. (2013). Orthogonal support vector machine for credit scoring. *Engineering Applications of Artificial Intelligence*, *26*(2), 848–862. https://doi.org/https://doi.org/10.1016/j.engappai.2012.10.005

Ho, T. K. (1995). Random decision forests. *Proceedings of 3rd international conference on document analysis and recognition*, *1*, 278–282.

Huang, C.-L., Chen, M.-C., & Wang, C.-J. (2007). Credit scoring with a data mining approach based on support vector machines. *Expert Systems with Applications*, *33*(4), 847–856. https://doi.org/https://doi.org/10.1016/j.eswa.2006.07.007

Kleinberg, E. M. (1990). Stochastic discrimination. *Annals of Mathematics and Artificial Intelligence*, *1*(1-4), 207–239. https://doi.org/10.1007/bf01531079

Malekipirbazari, M., & Aksakalli, V. (2015). Risk assessment in social lending via random forests. *Expert Systems with Applications*. https://doi.org/10.1016/j.eswa.2015.02.001

Markowitz, H. (1952). Portfolio selection. *The Journal of Finance*, *7*(1), 77–91. http://www.jstor.org/stable/2975974

Rojas, R. (1996). The backpropagation algorithm. *Neural networks: A systematic introduction* (pp. 149–182). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-61068-4_7

Rumelhart, D., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, *323*, 533–536.

Zurada, J., & Zurada, M. (2011). How secure are good loans: Validating loan-granting decisions and predicting default rates on consumer loans. *Review of Business Information Systems (RBIS)*, *6*, 65. https://doi.org/10.19030/rbis.v6i3.4563

A Feature list

The table below contains the abbreviation and description of all variables remaining after initial data processing.

Variable	Description
loan_status_bin	The target variable where 1 indicates that the borrower has fully paid off the loan and 0 in the case of a default
loan_amnt	The loan amount expressed in dollars
log_annual_inc	The log transformed annual income of the borrower
delinq_2yrs	The number of delinquencies in the last 2 years, values greater than 2 are set to 2
dti	Monthly debt to income ratio
log_credit_age	The log transformed value of the amount of months since the opening of the first credit line
emp_length	Length of employment in years, values greater than 10 are set to 10
inq_last_6mths	The amount of credit inquiries in the last 6 months
open_acc	The amount of open credit lines
total_acc	The amounf of total credit lines
log_inc_payment_ratio	Monthly loan payments to income ratio
log_revol_inc_ratio	The log transformed value of revolving credit (credit that do not have a fixed number of paymens) relative to monthly income
revol_util	the amount of credit relative to revolving credit
d_term_60	dummy variable indicating a 60 monthly payments on the loan
d_term_36	dummy variable indicating a 36 monthly payments on the loan
d_home_ownership_mortgage	dummy variable indicating home ownership in the form of a mortgage
d_home_ownership_own	dummy variable indicating a home ownership status
d_home_ownership_rent	dummy variable indicating home ownership in the form of rent
d_purpose_car	dummy variable indicating the categorical purpose of the loan request
d_purpose_credit_card	dummy variable indicating the categorical purpose of the loan request
d_purpose_debt_consolidation	dummy variable indicating the categorical purpose of the loan request
d_purpose_home_improvement	dummy variable indicating the categorical purpose of the loan request
d_purpose_house	dummy variable indicating the categorical purpose of the loan request
d_purpose_major_purchase	dummy variable indicating the categorical purpose of the loan request
d_purpose_medical	dummy variable indicating the categorical purpose of the loan request

d_purpose_moving	dummy variable indicating the categorical purpose of the loan request
d_purpose_other	dummy variable indicating the categorical purpose of the loan request
d_purpose_renewable_energy	dummy variable indicating the categorical purpose of the loan request
d_purpose_small_businss	dummy variable indicating the categorical purpose of the loan request
d_purpose_vacation	dummy variable indicating the categorical purpose of the loan request
d_purpose_wedding	dummy variable indicating the categorical purpose of the loan request

B Keras Tuner results

10 best trials Objective(name = 'val_auc', direction = 'max')	
Trial summary Hyperparameters: num_layers: 3 units_0: 21 units_1: 13 learning_rate: 0.01 units_2: 17 Score: 0.6635230481624603	Trial summary Hyperparameters: num_layers: 3 units_0: 24 units_1: 16 learning_rate: 0.01 units_2: 23 Score: 0.6634736061096191
Trial summary Hyperparameters: num_layers: 3 units_0: 29 units_1: 30 learning_rate: 0.01 units_2: 18 Score: 0.662992388010025	Trial summary Hyperparameters: num_layers: 3 units_0: 3 units_1: 29 learning_rate: 0.01 units_2: 27 Score: 0.6423858106136322
Trial summary Hyperparameters: num_layers: 3 units_0: 5	Trial summary Hyperparameters: num_layers: 3 units_0: 18

units_1: 2 learning_rate: 0.01 units_2: 8 Score: 0.6222719550132751	units_1: 4 learning_rate: 0.01 units_2: 2 Score: 0.6148495078086853
Trial summary Hyperparameters: num_layers: 3 units_0: 24 units_1: 24 learning_rate: 0.001 units_2: 11 Score: 0.5735157430171967	Trial summary Hyperparameters: num_layers: 4 units_0: 16 units_1: 18 learning_rate: 0.01 units_2: 15 units_3: 2 Score: 0.5546828210353851
Trial summary Hyperparameters: num_layers: 2 units_0: 29 units_1: 27 learning_rate: 0.0001 Score: 0.5309717655181885	Trial summary Hyperparameters: num_layers: 2 units_0: 28 units_1: 18 learning_rate: 0.0001 Score: 0.5175150632858276

C Python code

```
import pandas as pd
import numpy as np
import scipy
import matplotlib.pyplot as plt
import urllib
import sklearn
import math


from sklearn import metrics
```

```
from sklearn.metrics import confusion_matrix
from sklearn.metrics import roc_auc_score
from sklearn.metrics import mean_squared_error
from sklearn.inspection import permutation_importance
from sklearn.model_selection import KFold

import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Activation, Dense
from tensorflow.keras.metrics import categorical_crossentropy
from tensorflow.keras.callbacks import EarlyStopping
from keras_tuner import Objective, HyperParameters
from keras_tuner.tuners import RandomSearch
from keras.models import Sequential
from keras.layers import Dense

from sklearn import neighbors
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier

import os
import tempfile
import seaborn as sns
import random as rn

from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
import math

tf.random.set_seed(123)
os.environ['PYTHONHASHSEED'] = str(123)
```

```
np.random.seed(123)
rn.seed(123)

df = pd.read_csv('data.csv')

def perf_measure(y_actual, y_pred):
    TP = 0
    FP = 0
    TN = 0
    FN = 0

    for i in range(len(y_pred)):
        if y_actual[i]==y_pred[i]==1:
           TP += 1
        if y_pred[i]==1 and y_actual[i]!=y_pred[i]:
           FP += 1
        if y_actual[i]==y_pred[i]==0:
           TN += 1
        if y_pred[i]==0 and y_actual[i]!=y_pred[i]:
           FN += 1

    return(TP, FP, TN, FN)

kfold = KFold(n_splits=5, random_state=1, shuffle=True)

X_train_folds = []
X_test_folds = []
y_train_folds = []
y_test_folds = []

for train_index, test_index in kfold.split(X, y):
    X_train_folds.append(X.iloc[train_index])
    X_test_folds.append(X.iloc[test_index])
    y_train_folds.append(y.iloc[train_index])
    y_test_folds.append(y.iloc[test_index])
```

```
## get the evaluation metrics from the K-NN classifier
accuracy_avg = 0
rmse_avg = 0
auc_avg = 0
tpr = 0
fpr = 0
tp_avg = 0
fp_avg = 0
tn_avg = 0
fn_avg = 0

KNN = neighbors.KNeighborsClassifier(n_neighbors = 1, metric = 'euclidean', n_jobs = -1)
for i in range(0,5):
    KNN.fit(X_train_folds[i], y_train_folds[i])
    y_knn_predict = KNN.predict(X_test_folds[i])

    score = KNN.score(X_test_folds[i], y_test_folds[i])
    conf_matrix = confusion_matrix(y_test_folds[i], y_knn_predict)
    auc_score = roc_auc_score(y_test_folds[i], y_knn_predict)
    rmse = math.sqrt(mean_squared_error(y_test_folds[i], y_knn_predict))

    accuracy_avg = accuracy_avg + score
    rmse_avg = rmse_avg + rmse
    auc_avg = auc_avg + auc_score

    tp, fp, tn, fn = perf_measure(np.array(y_test_folds[i]), y_knn_predict)

    tpr = tpr + (tp/(tp+fn))
    fpr = fpr + (fp/(fp+tn))

    tp_avg = tp_avg + tp
    fp_avg = fp_avg + fp
    tn_avg = tn_avg + tn
```

```
    fn_avg = fn_avg + fn

print("accuracy_avg:", accuracy_avg/5)
print("auc_avg:", auc_avg/5)
print("rmse_avg", rmse_avg/5)
print("TPR:", tpr/5)
print("fpr:", fpr/5)
print('TP:', tp_avg/5)
print("FP:", fp_avg/5)
print("TN:", tn_avg/5)
print("FN:", fn_avg/5)


## get the evaluation metrics from the LR classifier
LR = LogisticRegression(penalty = 'none', fit_intercept = True, class_weight= {0:5, 1:1},
                        solver = 'newton-cg', random_state= 123, n_jobs = -1)

accuracy_avg = 0
rmse_avg = 0
auc_avg = 0
tpr = 0
fpr = 0
tp_avg = 0
fp_avg = 0
tn_avg = 0
fn_avg = 0

for i in range(0,5):
    LR.fit(X_train_folds[i], y_train_folds[i])
    y_lr_predict = LR.predict(X_test_folds[i])

    score = LR.score(X_test_folds[i], y_test_folds[i])
    conf_matrix = confusion_matrix(y_test_folds[i], y_lr_predict)
    auc_score = roc_auc_score(y_test_folds[i], y_lr_predict)
    rmse = math.sqrt(mean_squared_error(y_test_folds[i], y_lr_predict))
```

```
    accuracy_avg = accuracy_avg + score
    rmse_avg = rmse_avg + rmse
    auc_avg = auc_avg + auc_score

    tp, fp, tn, fn = perf_measure(np.array(y_test_folds[i]), y_lr_predict)

    tpr = tpr + (tp/(tp+fn))
    fpr = fpr + (fp/(fp+tn))

    tp_avg = tp_avg + tp
    fp_avg = fp_avg + fp
    tn_avg = tn_avg + tn
    fn_avg = fn_avg + fn

print("accuracy_avg:", accuracy_avg/5)
print("auc_avg:", auc_avg/5)
print("rmse_avg", rmse_avg/5)
print("TPR:", tpr/5)
print("fpr:", fpr/5)
print('TP:', tp_avg/5)
print("FP:", fp_avg/5)
print("TN:", tn_avg/5)
print("FN:", fn_avg/5)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20, random_state=123)



## measure feature importance according to K-NN
KNN.fit(X_train, y_train)
r_KNN = permutation_importance(KNN, X_test, y_test, scoring = 'accuracy',
                               n_repeats = 5, random_state =123)
r_KNN_sorted = []

for j in r_KNN.importances_mean.argsort()[::-1]:
```

```
    r_KNN_sorted.append(f"{X_test.columns[j]:<8}: "
          f"{r_KNN.importances_mean[j]:.3f}"
          f" +/- {r_KNN.importances_std[j]:.3f}")

r_KNN_sorted = pd.DataFrame(r_KNN_sorted)


## measure feature importance according to LR
LR.fit(X_train, y_train)
r_LR = permutation_importance(LR, X_test, y_test, scoring = 'accuracy',
                              n_repeats = 5, random_state =123, n_jobs = -1)
r_sorted_LR = []

for j in r_LR.importances_mean.argsort()[::-1]:
    r_sorted_LR.append(f"{X_test.columns[j]:<8}: "
          f"{r_LR.importances_mean[j]:.3f}"
          f" +/- {r_LR.importances_std[j]:.3f}")

r_sorted_LR = pd.DataFrame(r_sorted_LR)


## get the evaluation metrics from the SVM classifier
svm = SVC(C= 1, kernel= 'poly', degree = 2, gamma = 1, coef0 = 1, cache_size = 2500,
          class_weight= {0:5, 1:1}, max_iter = 10000)

accuracy_avg = 0
rmse_avg = 0
auc_avg = 0
tpr = 0
fpr = 0
tp_avg = 0
fp_avg = 0
tn_avg = 0
fn_avg = 0
```

```
for i in range(0,5):
    svm.fit(X_train_folds[i], y_train_folds[i])
    y_svm_predict = svm.predict(X_test_folds[i])

    score = svm.score(X_test_folds[i], y_test_folds[i])
    conf_matrix = confusion_matrix(y_test_folds[i], y_svm_predict)
    auc_score = roc_auc_score(y_test_folds[i], y_svm_predict)
    rmse = math.sqrt(mean_squared_error(y_test_folds[i], y_svm_predict))

    accuracy_avg = accuracy_avg + score
    rmse_avg = rmse_avg + rmse
    auc_avg = auc_avg + auc_score

    tp, fp, tn, fn = perf_measure(np.array(y_test_folds[i]), y_svm_predict)

    tpr = tpr + (tp/(tp+fn))
    fpr = fpr + (fp/(fp+tn))

    tp_avg = tp_avg + tp
    fp_avg = fp_avg + fp
    tn_avg = tn_avg + tn
    fn_avg = fn_avg + fn

print("accuracy_avg:", accuracy_avg/5)
print("auc_avg:", auc_avg/5)
print("rmse_avg", rmse_avg/5)
print("TPR:", tpr/5)
print("fpr:", fpr/5)
print('TP:', tp_avg/5)
print("FP:", fp_avg/5)
print("TN:", tn_avg/5)
print("FN:", fn_avg/5)


## measure feature importance according to SVM
```

```
svm.fit(X_train, y_train)

r_svm = permutation_importance(svm, X_test, y_test, scoring = 'accuracy',
                               n_repeats = 5, random_state =123, n_jobs = -1)
r_svm_sorted = []

for j in r_svm.importances_mean.argsort()[::-1]:
    r_svm_sorted.append(f"{X_test.columns[j]:<8}: "
          f"{r_svm.importances_mean[j]:.3f}"
          f" +/- {r_svm.importances_std[j]:.3f}")

r_svm_sorted = pd.DataFrame(r_svm_sorted)



## get the evaluation metrics from the RF classifier
depth =  22
n_trees = 75
rf = RandomForestClassifier(n_estimators = n_trees, criterion = 'gini', max_depth = depth, min_s
                            max_features = 5, bootstrap = True, n_jobs = -1, random_state = 123,
                            class_weight = {0:5, 1:1})

accuracy_avg = 0
rmse_avg = 0
auc_avg = 0
tpr = 0
fpr = 0
tp_avg = 0
fp_avg = 0
tn_avg = 0
fn_avg = 0

for i in range(0,5):
    rf.fit(X_train_folds[i], y_train_folds[i])
    y_rf_predict = rf.predict(X_test_folds[i])
```

```
    score = rf.score(X_test_folds[i], y_test_folds[i])
    conf_matrix = confusion_matrix(y_test_folds[i], y_rf_predict)
    auc_score = roc_auc_score(y_test_folds[i], y_rf_predict)
    rmse = math.sqrt(mean_squared_error(y_test_folds[i], y_rf_predict))

    accuracy_avg = accuracy_avg + score
    rmse_avg = rmse_avg + rmse
    auc_avg = auc_avg + auc_score

    tp, fp, tn, fn = perf_measure(np.array(y_test_folds[i]), y_rf_predict)

    tpr = tpr + (tp/(tp+fn))
    fpr = fpr + (fp/(fp+tn))

    tp_avg = tp_avg + tp
    fp_avg = fp_avg + fp
    tn_avg = tn_avg + tn
    fn_avg = fn_avg + fn

print("accuracy_avg:", accuracy_avg/5)
print("auc_avg:", auc_avg/5)
print("rmse_avg", rmse_avg/5)
print("TPR:", tpr/5)
print("fpr:", fpr/5)
print('TP:', tp_avg/5)
print("FP:", fp_avg/5)
print("TN:", tn_avg/5)
print("FN:", fn_avg/5)


## measure feature importance according to RF
sorted_importance = rf.feature_importances_.argsort()
sorted_importance = sorted_importance[-10:]
plt.barh(X.columns[sorted_importance], rf.feature_importances_[sorted_importance])
plt.xlabel('RF impurity-based feature importance ')
```

```
## Implementation of Keras Tuner
def model_builder(hyper_param: HyperParameters):
    #create model and input layer
    model = keras.Sequential()
    model.add(keras.layers.InputLayer(input_shape=x_train.shape[1:]))

    #create hidden layers according to 'num_layers' and 'units_' within a for-loop
    for i in range(hyper_param.Int('num_layers', min_value = 2, max_value = 4, step = 1)):
        model.add(Dense(units = hyper_param.Int('units_' + str(i),
                                                min_value = 2,
                                                max_value = 30,
                                                step = 1),
                                        activation = 'sigmoid'))

    #create output layer
    model.add(Dense(1, activation = 'sigmoid', name = 'output_node'))

    hyper_param_lr = hyper_param.Choice('learning_rate', values = [1e-2, 1e-3, 1e-4])
    model.compile(optimizer = keras.optimizers.SGD(learning_rate = hyper_param_lr),
                  loss = keras.losses.BinaryCrossentropy(),
                  metrics = [tf.keras.metrics.BinaryCrossentropy('cross_entropy'),
                             tf.keras.metrics.AUC(name = 'auc')]
                 )

    return model

tuner = RandomSearch( model_builder,
                      objective= Objective('val_auc', direction = 'max'),
                      max_trials = 15,
                      executions_per_trial = 2,
                      seed = 123,
                      overwrite = True,
                      project_name= 'tuner')
```

```
tuner.search(x_train, y_train, epochs = 10,
             validation_data = (x_test, y_test),
             class_weight = {0: 5, 1: 1})


## Building the FFNN
nn = keras.Sequential()
l1_nodes = 21
l2_nodes = 13
l3_nodes = 17
lrate = 0.01

#input layer
nn.add(keras.layers.InputLayer(input_shape=x_train.shape[1:]))

#hidden layers
nn.add(Dense(l1_nodes, activation = 'sigmoid', name = 'layer1'))
nn.add(Dense(l2_nodes, activation = 'sigmoid', name = 'layer2'))
nn.add(Dense(l3_nodes, activation = 'sigmoid', name = 'layer3'))

#output layer
nn.add(Dense(1, activation = 'sigmoid', name = 'output_node'))

METRICS = [ keras.metrics.BinaryAccuracy(name='accuracy'),
            keras.metrics.AUC(name='auc'),
            keras.metrics.RootMeanSquaredError(name='rmse')
]

nn.compile(optimizer = keras.optimizers.SGD(learning_rate = lrate),
           loss = keras.losses.BinaryCrossentropy(),
           metrics = METRICS)

nn.save_weights('start.h5', overwrite = True)
```

```
## estimate optimal epoch amount
nn.load_weights('start.h5')

monitor = EarlyStopping(monitor = 'val_rmse', patience = 10, verbose = 1,
                        mode = 'min', restore_best_weights = True)

history = nn.fit(x_train, y_train, epochs = 250, batch_size = 32, verbose = 2,
                 callbacks = [monitor], validation_data = (x_test, y_test),
                 class_weight = {0: 5, 1: 1}, initial_epoch = 0)

## gather evaluation metrics for the FFNN classifier
accuracy_avg = 0
rmse_avg = 0
auc_avg = 0
tpr = 0
fpr = 0
tp_avg = 0
fp_avg = 0
tn_avg = 0
fn_avg = 0

optimal_epochs = 25

for i in range(0,5):
    nn.load_weights('start.h5')

    nn.fit(
        X_train_folds[i], y_train_folds[i], epochs = optimal_epochs,
        batch_size = 32, verbose = 2, initial_epoch = 0,
        class_weight = {0: 5, 1: 1}
    )

    y_predict_prob = nn.predict(X_test_folds[i], verbose = 1)
    y_predict = y_predict_prob >= 0.5
```

```
    score = accuracy_score(y_test_folds[i], y_predict)
    conf_matrix = confusion_matrix(y_test_folds[i], y_predict)
    auc_score = roc_auc_score(y_test_folds[i], y_predict)
    rmse = math.sqrt(mean_squared_error(y_test_folds[i], y_predict))

    accuracy_avg = accuracy_avg + score
    rmse_avg = rmse_avg + rmse
    auc_avg = auc_avg + auc_score

    tp, fp, tn, fn = perf_measure(np.array(y_test_folds[i]), y_predict)

    tpr = tpr + (tp/(tp+fn))
    fpr = fpr + (fp/(fp+tn))

    tp_avg = tp_avg + tp
    fp_avg = fp_avg + fp
    tn_avg = tn_avg + tn
    fn_avg = fn_avg + fn

print("accuracy_avg:", accuracy_avg/5)
print("auc_avg:", auc_avg/5)
print("rmse_avg", rmse_avg/5)
print("TPR:", tpr/5)
print("fpr:", fpr/5)
print('TP:', tp_avg/5)
print("FP:", fp_avg/5)
print("TN:", tn_avg/5)
print("FN:", fn_avg/5)

## measure feature importance according to FFNN
optimal_epochs = 25

nn.load_weights('start.h5')
nn.fit(
```

```
    X_train, y_train, epochs = optimal_epochs,
    batch_size = 32, verbose = 2, initial_epoch = 0,
    class_weight = {0: 5, 1: 1}
)

y_predict_prob = nn.predict(X_test, verbose = 1)
y_predict = y_predict_prob >= 0.5

score = accuracy_score(y_test, y_predict)
print(score)

feature_importance = []
for col in X_test.columns:
    #print("feature permutated: ", col)
    X_test_perm = X_test.copy()
    X_test_perm[col] = np.random.permutation(X_test_perm[col])

    y_predict_prob = nn.predict(X_test_perm, verbose = 0)
    y_predict_perm = y_predict_prob >= 0.5

    score_perm = accuracy_score(y_test, y_predict_perm)
    diff = score - score_perm
    feature_importance.append([col, diff])

feature_importance = pd.DataFrame(feature_importance)
```