



ML Project Report(Lend or Lose Dataset)

Team Name: Blitz Coders

Members:

Aayush Bhargav (IMT2022089)

Praveen Peter Jay (IMT2022064)

Shannon Muthanna I B (IMT2022552)

GitHub Link

<https://github.com/Aayush-Bhargav/ML-Codes>

Overview

The dataset which was provided to us was the **lend and lose dataset**. This dataset has a total of eighteen columns. The output column is named 'Default', which denotes whether the borrower is a defaulter. This project aims to predict if a borrower is a defaulter or not. The detailed description of the dataset is given in the following link


<https://www.kaggle.com/competitions/lend-or-lose/data>.

In the course of this project, we have tried out different models and tried to determine the best model based on the accuracy score. The dataset was thoroughly analyzed using EDA techniques. The training data was split into train and test sets, and all models were applied along with cross-validation to measure their accuracy. After training the model, the test data was applied to it, and the predicted values were written to a CSV file for submission.

EDA (Exploratory Data Analysis)

Different sets of data analysis techniques were applied to the dataset to understand the data. Firstly, the correlation of each of these input columns with the output columns was found to see how much each feature contributes to the output. The output of the correlation inferred that all of the features are important in the dataset. The percentage of defaulters and non-defaulters was found to get an overview of the distribution of the defaulters in the dataset. The dataset was dominated by the amount of non-defaulters with a percentage of **88.37** of the entire dataset. (Imbalanced dataset; one class dominates another).

The numerical columns of the dataset were analyzed by plotting their distribution and boxplot. These columns had **no outliers** and most of them had their distribution similar to that of an uniform distribution. The dataset was also **free of null values**. The categorical columns were analyzed using the countplot and we have observed that the data is almost equally distributed among all categories. But when you saw the distribution of defaulters among various categories, you could find a sense of ordering (Eg: People with "High School" level education caused more defaults than those with "Bachelor's" than those with "Master's" than those with "PhD".) This ordering was present in every single categorical



column and thus ordinal encoding made sense here. These categorical columns were encoded to numbers using the 'map' function (Equivalent to ordinal encoding where you assign a number with a category (Eg: 1 to High School, 2 to Bachelor's, 3 to Master's, and 4 to PhD) . The same process was followed for the assignment of numerical values to categories for all the categorical columns with the category having the highest defaulters being assigned value = 1 and the category with the least defaulters assigned the value equal to the number of categories.

The column 'Loan ID' was removed from the training data since it does not contribute much in predicting the labels. Since the data has sixteen input columns we thought of applying PCA to reduce the dimensionality of the data. All sixteen principle components were obtained from the data and their corresponding eigenvalues were printed. There **was no steep decrease in the eigenvalues** which proves our point that all the features are important for the prediction. PCA is only useful when the data is inherently in a lower dimension. The data is in a lower dimension if the eigenvalues decrease drastically after a certain point.

Models

The dataset was split into train and test sets and passed through various models to validate its performance. Following is the description of various models along with their performance on our dataset. In most of the models, RandomizedGridSearchCV was used to get the best parameters since GridSearchCV took a long time. Also, we have not used Logistic Regression in this part of the project since we will be doing it in the next part.

XGBoost

XGBoost builds an ensemble of decision trees in sequence, where each tree aims to correct errors from the previous ones, optimizing both accuracy and generalization through advanced regularization and parallelized computation.

This model gave us the best accuracy of **88.829** percent. The reason why XGBoost is so effective is because it is an ensemble technique which combines numerous weak classifiers into one strong classifier. XGBoost lets us select many hyperparameters which helps in reducing overfitting.

Unlike traditional gradient boosting, XGBoost uses both first and second derivatives (Hessian) in its loss function optimization. This allows it to adjust each step's weight updates more precisely, improving convergence speed and accuracy.

XGBoost does not depend on scaling, our model gave the same accuracy for both scaled and unscaled data.

Random Forest:

This is a Bagging-based method where multiple models are trained simultaneously and the final decision is taken based on the majority vote of all these models. This model also uses random subsets of the data for each of its sub-models. It also uses random feature splitting at each node which makes it a model that reduces overfitting. This model gave us a slightly lower accuracy of **88.674** percent. The reason for slightly lower accuracy than XGBoost is because Random Forest uses a Bagging-based ensemble method. In Bagging each model doesn't have any relation with the other models. Thus every model is independent of the other and doesn't know where they are going wrong.

AdaBoost

This is another Boosting-based model that assigns higher weights to misclassified samples and lower weights to correctly classified ones in each iteration. The final decision is taken as a weighted combination of the outputs of each model. These weights are obtained from the accuracy of each weak classifier. The accuracy of this model for our data was **88.676** percent. This model was slower compared to the other models since it iteratively updates weights for each sub-model. Thus for a large dataset, this is a very slow model. The reasons for its slightly lower accuracy compared to XGBoost are due to the way it updates weights and lack of explicit regularization. Adaboost updates weights based on previous models' predictions whereas XGBoost optimizes based on gradient optimization. XGBoost has in-built L1 and L2 regularization whereas Adaboost lacks it.

Gradient Boosting

Gradient Boosting is a machine learning technique that uses boosting-based ensemble methods to predict the labels. It focuses on minimizing the residuals generated by the previous iteration in each successive iteration. This model gave us an accuracy of **88.748** percent. XGBoost Uses a second-order Taylor approximation (both gradient and Hessian) for optimizing the loss function, providing more accurate updates and faster convergence. Gradient Boosting Uses only the gradient (first derivative) to update weights, which can be slower to converge and less precise in certain cases.

Decision Tree

A machine learning technique that builds a tree-like structure to predict the labels. The tree is constructed with the help of Gini index splitting. The accuracy of the best decision tree is **88.447** percent. This model's lower performance is due to overfitting since it uses the entire data and all features to build the tree at each node. Also, we rely on the prediction of a single model.

KNN (K-Nearest Neighbours)

One more classification technique that predicts the label based on its nearest neighbors. Here 'K' is a hyperparameter that denotes the number of nearest neighbors. This is a distance-based algorithm thus it requires us to standardize the data for better performance. To determine the best 'K' value we iterated through different K's and plotted their cross-val score. The K value with the highest cross-val score was chosen. This model gave us an accuracy of **88.447** percent. The reason for this low accuracy is due to the fact that the data is very imbalanced with only 12 percent of the data being defaulters. We visualized our data with the help of PCA and found out that the labels of defaulters are very far apart. Thus KNN predicted non-defaulters for all the test samples which gave us a lower accuracy.

Naive Bayes

This model is a probabilistic classifier that calculates the class conditional probabilities based on the training data. A new data point is classified into one of the classes by choosing the one with the highest probability among all the class conditional probabilities of this point. In our project, we have used the Gaussian Naive Bayes. This model gave us the lowest accuracy of **88.4** percent. This is because Naive Bayes assume all input columns are independent of each other i.e. there is no correlation between different inputs. This is not true for our data since inputs are correlated with each other. For example, Education and Income have a high correlation between them. On scaled data, this model performed slightly better giving an accuracy of **88.5**%.

Top accuracies with their parameters

The first set of parameters gave us an accuracy of 88.789% on submission. For the second and third attempts, the parameters were tweaked, i.e., the number of estimators was increased gradually. These two parameters gave us an accuracy of 88.807% and 88.809% respectively. For the final attempt, we narrowed down the parameters based on the inferences from the last three attempts to get the best accuracy (88.829%). We also experimented by changing the random states which lead to distinct accuracies.

XGBoost Best Parameter finding evolution

Model parameters that gave us **88.789** % accuracy.

```
] : # Let us try to improve model accuracy by hyperparameter tuning using gridSearchCV

]: # Define parameter grid for GridSearchCV
param_grid = {
    'n_estimators': [80,90,100,110,120], # default is 100, so let us search for 80,90,100,110,120
    'learning_rate': [0.1,0.2, 0.25,0.3,0.35,0.4], # default is 0.3, so let us search for 0.1,0.2,0.25,0.3,0.35,0.4
    'max_depth': [6,7,8,9], # default is 6, so let us search for 6,7,8,9
    'subsample': [0.7,0.8,0.9,1.0], # default is 1, so let us search for 0.7,0.8,0.9,1
    'colsample_bytree': [0.8,0.9,1.0], # default is 1, so let us search for 0.8,0.9,1
}

# Initialize the XGBoost classifier
xgb_clf = XGBClassifier(eval_metric='logloss', random_state=42)

# Set up GridSearchCV
xgb_grid = GridSearchCV(
    estimator=xgb_clf,
    param_grid=param_grid,
    cv=5, # 5-fold cross-validation
    n_jobs=-1, # Use all available cores
    verbose=1, # Print progress
)
```

```
print(xgb_random_search.best_estimator_)
```

```
XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=0.8, device=None, early_stopping_rounds=None,
              enable_categorical=False, eval_metric='logloss',
              feature_types=None, gamma=None, grow_policy=None,
              importance_type=None, interaction_constraints=None,
              learning_rate=0.1, max_bin=None, max_cat_threshold=None,
              max_cat_to_onehot=None, max_delta_step=None, max_depth=6,
              max_leaves=None, min_child_weight=None, missing=nan,
              monotone_constraints=None, multi_strategy=None, n_estimators=120,
              n_jobs=None, num_parallel_tree=None, random_state=15, ...)
```

Model parameters that gave us **88.807%** accuracy.

```
# Define parameter grid for GridSearchCV
param_grid = {
    'n_estimators': [120,140,160,180], # previous best is 120, let us search for 120,140,160,180
    'learning_rate': [0.1,0.2,0.3], # previous best is 0.1, so let us search for 0.1,0.2,0.3
    'max_depth': [3,4,5,6,7], # previous best is 6, so let us search for 3,4,5,6,7
    'subsample': [0.7,0.8,0.9,1.0], # default is 1, so let us search for 0.7,0.8,0.9,1
    'colsample_bytree': [0.8,0.9,1.0], # default is 1, so let us search for 0.8,0.9,1
}
|
# Initialize the XGBoost classifier
xgb_clf = XGBClassifier(eval_metric='logloss', random_state=7)

# Set up GridSearchCV
xgb_grid = GridSearchCV(
    estimator=xgb_clf,
    param_grid=param_grid,
    cv=5, # 5-fold cross-validation
    n_jobs=-1, # Use all available cores
    verbose=1, # Print progress
)
```

```
print(xgb_random_search.best_estimator_)
```

```
XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=0.8, device=None, early_stopping_rounds=None,
              enable_categorical=False, eval_metric='logloss',
              feature_types=None, gamma=None, grow_policy=None,
              importance_type=None, interaction_constraints=None,
              learning_rate=0.2, max_bin=None, max_cat_threshold=None,
              max_cat_to_onehot=None, max_delta_step=None, max_depth=3,
              max_leaves=None, min_child_weight=None, missing=nan,
              monotone_constraints=None, multi_strategy=None, n_estimators=140,
              n_jobs=None, num_parallel_tree=None, random_state=7, ...)
```


Model parameters that gave us **88.809%** accuracy

```
# Define parameter grid for GridSearchCV
param_grid = {
    'n_estimators': [175,200,225,250], # prev best is 140 and increasing the estimators is showing increase in accuracy.
    'learning_rate': [0.1,0.2,0.3], # default is 0.3, so let us search for 0.1,0.2,0.3
    'max_depth': [3,4,5,6,7], # prev best is 3, so let us search for 3,4,5,6,7
    'subsample': [0.7,0.8,0.9,1.0], # default is 1, so let us search for 0.7,0.8,0.9,1
    'colsample_bytree': [0.8,0.9,1.0], # default is 1, so let us search for 0.8,0.9,1
}

# Initialize the XGBoost classifier
xgb_clf = XGBClassifier(eval_metric='logloss', random_state=17)

# Set up GridSearchCV
xgb_grid = GridSearchCV(
    estimator=xgb_clf,
    param_grid=param_grid,
    cv=5, # 5-fold cross-validation
    n_jobs=-1, # Use all available cores
    verbose=1, # Print progress
)
```

```
print(xgb_random_search.best_estimator_)

XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=1.0, device=None, early_stopping_rounds=None,
              enable_categorical=False, eval_metric='logloss',
              feature_types=None, gamma=None, grow_policy=None,
              importance_type=None, interaction_constraints=None,
              learning_rate=0.1, max_bin=None, max_cat_threshold=None,
              max_cat_to_onehot=None, max_delta_step=None, max_depth=3,
              max_leaves=None, min_child_weight=None, missing=nan,
              monotone_constraints=None, multi_strategy=None, n_estimators=200,
              n_jobs=None, num_parallel_tree=None, random_state=17, ...)
```

Model parameters that gave us **88.829%** accuracy (the best accuracy)


```
# Define parameter grid for GridSearchCV
param_grid = {
    'n_estimators': [180,190,200,210], # let us search for 180,190,200,210
    'learning_rate': [0.1,0.2,0.3], # default is 0.3, so let us search for 0.1,0.2,0.3
    'max_depth': [3,4,5,6,7], # default is 6, so let us search for 3,4,5,6,7
    'subsample': [0.7,0.8,0.9,1.0], # default is 1, so let us search for 0.7,0.8,0.9,1
    'colsample_bytree': [0.8,0.9,1.0], # default is 1, so let us search for 0.8,0.9,1
}

# Initialize the XGBoost classifier
xgb_clf = XGBClassifier(eval_metric='logloss', random_state=9)

# Set up GridSearchCV
xgb_grid = GridSearchCV(
    estimator=xgb_clf,
    param_grid=param_grid,
    cv=5, # 5-fold cross-validation
    n_jobs=-1, # Use all available cores
    verbose=1, # Print progress
)
```

```
print(xgb_random_search.best_estimator_)
```

```
XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=0.8, device=None, early_stopping_rounds=None,
              enable_categorical=False, eval_metric='logloss',
              feature_types=None, gamma=None, grow_policy=None,
              importance_type=None, interaction_constraints=None,
              learning_rate=0.1, max_bin=None, max_cat_threshold=None,
              max_cat_to_onehot=None, max_delta_step=None, max_depth=5,
              max_leaves=None, min_child_weight=None, missing=nan,
              monotone_constraints=None, multi_strategy=None, n_estimators=200,
              n_jobs=None, num_parallel_tree=None, random_state=9, ...)
```

Assignment Part 2 Models

Logistic Regression


Logistic Regression is a statistical method used for binary classification problems, where the outcome is categorical (e.g., 0 or 1, true or false). It models the probability of an event occurring by applying the logistic function to a linear combination of the input features. Unlike linear regression, which predicts continuous values, logistic regression outputs a probability score between 0 and 1, which is then thresholded to classify data points.

Key components of logistic regression include:

- **Sigmoid Function:** Transforms the linear output into a probability.
- **Cost Function:** Binary Cross Entropy Loss function optimizes the model.

We implemented Logistic Regression in 2 ways. We modeled Logistic Regression initially as a single-layer, single-neuron neural network. It had 16 input features and we ran it for 1000 epochs. On the test data, it gave an accuracy of **88.43%**. On submitting on Kaggle for a late submission, it gave **88.586%** accuracy.

In the second method, we used Scikit-Learn's logistic regression method and it gave the exact same predictions as our neural network implementation.

 predictions_lr.csv Complete (after deadline) · Aayush Bhargav · 1d ago	0.88586	0.88586	<input type="checkbox"/>
--	----------------	----------------	--------------------------

Support Vector Machines

Support Vector Machines (SVM) are supervised learning models used for both classification and regression tasks. SVMs work by finding the hyperplane that best separates data points into classes while maximizing the margin between the classes. They are effective for high-dimensional spaces and work well with different kernel functions to handle non-linear relationships in the data.

For this project, we used **cuML**, a GPU-accelerated machine learning library, due to the large size of our dataset. The traditional SVM implementation from sci-kit-learn was computationally expensive and could not efficiently handle the dataset size on a CPU. By leveraging **cuML**, which is optimized for NVIDIA GPUs, we significantly reduced training time. The dataset was scaled and converted into GPU-compatible data structures using `cudf.DataFrame` and `cudf.Series`.

We trained two models with different kernels to capture complex data patterns:

- **Radial Basis Function (RBF) Kernel:** Handles non-linear relationships by mapping data into a higher-dimensional space.
- **Polynomial Kernel:** Suitable for capturing interactions of features up to a specific degree, useful for moderately complex patterns.

The GPU acceleration provided by cuML enabled faster computation, making SVM a viable option for large-scale data analysis. This approach highlights the importance of leveraging modern hardware for handling large datasets in machine learning workflows.

With the Radial Basis Function (RBF) Kernel, the model gave a modest accuracy of **88.453 %** on submission in Kaggle.

	predictions_svm(1).csv Complete (after deadline) · Aayush Bhargav · 13s ago	0.88453	0.88453	<input type="checkbox"/>
--	---	----------------	----------------	--------------------------

With the polynomial kernel of degree 3, we got an even lesser accuracy of **88.447 %**.

	predictions_svm2.csv Complete (after deadline) · Aayush Bhargav · 14s ago	0.88447	0.88447	<input type="checkbox"/>
--	---	----------------	----------------	--------------------------

The SVM was predicting almost everyone as non-defaulters.

Neural Networks

Neural Networks are a class of machine learning models inspired by the structure and function of the human brain. They consist of layers of interconnected nodes (neurons) that process and learn from data. Neural networks are highly versatile and are used for tasks such as image classification, natural language processing, and time-series prediction.

Each neuron applies a transformation to the input using a weighted sum, adds a bias, and passes the result through an activation function to introduce non-linearity. The network learns by adjusting weights through backpropagation, using optimization techniques like gradient descent to minimize the loss function.

Implementation Details:

To compare performance and flexibility, we implemented two neural network models using different deep learning frameworks:

1. PyTorch:

- PyTorch offers dynamic computational graphs, making it highly flexible for experimentation.
- We designed and trained a neural network using PyTorch's **torch.nn** module for defining layers and **torch.optim** for optimization.
- We tried 3 models and in each of the models, for each of the hidden layers, we used the ReLU (Rectified Linear Unit) Activation Function and for the output layer, we used the Sigmoid Activation Function and we used **Adam** Optimizer to update weights and biases.

- In the first model, we had 3 layers - 2 hidden layers and 1 output layer. We followed a funnel structure with 128 layers in the first layer, 64 in the second one and 1 in the output layer. We ran the neural network for 5000 epochs and on the test data, it gave an accuracy of **88.11 %**.
- In the second model, we had 4 layers - 3 hidden layers and 1 output layer. We followed a funnel structure again with the layers having 128 nodes, 64 nodes, 32 nodes and then 1 node in the output layer. In this model, we used the **Dropout** technique. Dropout is a regularization technique used in neural networks to prevent overfitting. Overfitting occurs when the model learns to memorize the training data instead of generalizing patterns that work well on unseen data. Dropout randomly sets a fraction of neurons in a layer (e.g., 50% when `nn.Dropout(0.5)` is used) to zero. This ensures that the network does not rely too heavily on any single neuron or feature, promoting redundancy and collaboration among neurons. This model gave a better accuracy of **88.46%** on the test data. And on kaggle submission, we got an accuracy of **88.576%**.

	predictions(3).csv Complete (after deadline) · Aayush Bhargav · 1d ago	0.88576	0.88576	<input type="checkbox"/>
--	--	---------	---------	--------------------------

- In the third model, the structure was very similar to the second model except that we used Batch Normalization. **Batch Normalization (BatchNorm)** is a technique used in neural networks to improve training stability and speed by normalizing the inputs to each layer. It reduces internal covariate shift, which occurs when the distribution of layer inputs changes during training, leading to instability and slower convergence. This model gave a higher accuracy of **88.54 %** on the test data set and on submitting on Kaggle, it gave an even higher accuracy of **88.746 %**.



	predictions1(3).csv Complete (after deadline) · Aayush Bhargav · 1d ago	0.88746	0.88746	<input type="checkbox"/>
--	---	---------	---------	--------------------------

2. TensorFlow:

- TensorFlow is a widely-used framework known for its performance and scalability.
- We implemented another neural network using TensorFlow's `tf.keras` API, which provides a user-friendly interface for building and training models.
- Two models were built using this library. One with 3 layers and the other with four layers. The first model had 2 hidden layers and one output layer while

the second model had 3 hidden layers with one output layer. The hidden layers used the **ReLU** activation function and the output layer used the **Sigmoid** activation function. Dropout of various fractions was used to prevent overfitting. Batch Normalization was used to standardize the output of the layers. **Adam** Optimizer was used to update weights and biases. The loss function used was the binary-cross entropy loss function. The final weights and biases were stored in a .keras file. Early stopping was used to stop the training when the validation loss didn't change for 25 epochs.

- Model 1 has hidden layers with 64 and 32 neurons while Model 2 has hidden layers with 128, 64, and 32 neurons. The accuracies of both the models are as follows: Model 1 had the best accuracy of **88.784** percent while Model 2 had an accuracy of **88.709** percent. These accuracies suggest that Model 2 is overfitting.

	Sub_model1(6).csv Complete (after deadline) · Shannon Muthanna I B · 19h ago	0.88784	0.88784
	Sub_model2(7).csv Complete (after deadline) · Shannon Muthanna I B · 19h ago	0.88709	0.88709

By using two frameworks, we were able to explore their unique features. PyTorch's debugging-friendly dynamic computation graph and TensorFlow's robust production-ready environment provided complementary insights into model development and training.

Table Comparing Various Models

Index	Model Name	Best Parameters	Accuracy
1.	XGBoost	n_estimators=200, learning_rate=0.1, max_depth=5, colsample_bytree=0.8	88.829%
2.	Neural Network (TensorFlow)	2 hidden layers with 64 and 32 neurons respectively.	88.784%
3.	Gradient Boosting	n_estimators=120	88.748%
4.	Neural Network (PyTorch)	3 hidden layers with 128, 64 and 32 neurons respectively.	88.746%

5.	AdaBoost	max_depth=1, learning_rate=0.6, n_estimators=100	88.676%
6.	Random Forest	max_depth=10, max_features=16, min_samples_leaf=4, min_samples_split=5, n_estimators=120	88.674%
7.	Logistic Regression	Default Parameters	88.586%
8.	Naive Bayes	Gaussian Naive Bayes with Default Parameters	88.5%
10.	Support Vector Machines (SVM)	kernel='rbf', C=1.4, gamma='scale'	88.453%
9.	Decision Trees	criterion='entropy', max_depth=4, min_samples_leaf=2, min_samples_split=5	88.447%
11.	K Nearest Neighbours	n_neighbours=6	88.447%

Leaderboard Position

We finished at position **1** on the leaderboard.

Contributions

Aayush Bhargav

EDA/Pre-processing, XGBoost, Naive Bayes, Neural Network Using PyTorch, Logistic Regression.

Praveen Peter Jay

Gradient Boosting, Decision Trees, Support Vector Machines.

Shannon Muthanna I B

Random Forest, AdaBoost, KNN, Neural Network Using TensorFlow.