Loan Default Prediction

Youssef Ashraf Kandil
Computer Engineering
The American University in Cairo
youssefkandil@aucegypt.edu

Abdullah Mohamed Kassem
Computer Engineering
The American University in Cairo
Abdullahkassem@aucegypt.edu

Introduction:

During previous project phases, we always encountered a problem in the model metrics when trying to use the sklearn library models and that problem was due to the class imbalance in our dataset. After trying many different techniques we settled that we need to modify the loss function in our implementation of the model. In phase 4 we had the idea to change the learning rate based on the label, in order to increase the change in weights and learning that occurs from points with the label 1 (our minority class). We attempted to do so in this phase and also we implemented the Utility application that was proposed in the previous phase as well.

Model Implementation:

We used the code of lab five for Neural Networks as a foundation for our implementation. We first added a tanh activation function as it produced the best results from the cross validation we did in phase 4. However, it did not work as expected and we could not handle it so we used the sigmoid activation function.

Tanh:

```
def _tanh(self,z):
 return (np.exp(z) - np.exp(-z))/
              (np.exp(z) + np.exp(-z))
```

Sigmoid:

```
def _sigmoid(self,z):
   return (1/(1+np.exp(-1*z)))
```

The Original lab code used “Batch” gradient descent to solve the loss function, thus the idea we proposed wouldn’t work, because it depends on changing the learning rate for each sample point.

Our solution then was to implement a stochastic gradient descent model, where the weights get updated by iterating over each sample instead of each epoch. We managed to implement it, however since we have 120k+ points to train, it would have taken an unreasonable amount of time to finish, so we had to find another solution.

Our next solution was instead of multiplying the learning rate by a factor to bias it, to multiply the delta changes calculated after each backward propagation by that factor. So the loss function would be:

Fro output layer:

$$\delta = \begin{cases} \sigma(1-\sigma)(y-\sigma) & y=0 \\ 4\sigma(1-\sigma)(y-\sigma) & y=1 \end{cases}$$

For hidden layers:

$$\delta = \begin{cases} \sigma\,(1-\sigma)\sum w\delta & y=0 \\ 4\sigma(1-\sigma)\sum w\delta & y=1 \end{cases}$$

To do so, we created a matrix called factor that contains 1 if label is 0 and 4 if label was 1.

$$factor = \begin{cases} 1 & y=0 \\ 4 & y=1 \end{cases}$$

```
def _diff_MSE(self,y,yhat, factor):
    d=(yhat-y)
    return  pd.DataFrame(d.values*factor.values, columns=d.columns, index=d.index)
```

We made the ratio 4:1 because the ratio between both our classes is 4:1. Then we multiplied the factor matrix to the delta matrix produced at each epoch.

Though our ideas should work in principle we could not produce reliable results. We kept trying to change the parameters and tweak the model as much as we could but it still did produce the intended results. We also tried Mini-batch gradient descent, which combines both gradient descent and stochastic gradient descent. Weights are updated every small batch, that way we can produce relevantly reliable results in a reasonable time, yet it showed no progress in the right direction. Thus we stuck with the following hyperparameters as they showed the best results.

	precision	recall	f1-score	support
0	0.83	0.28	0.41	22319
1	0.27	0.83	0.41	7415
accuracy			0.41	29734
macro avg	0.55	0.55	0.41	29734
weighted avg	0.69	0.41	0.41	29734

Hyperparameters:

Layers:

```
model_classifier.add_layer (25,8,activation = "sigmoid"

model_classifier.add_layer (8,8, activation = "sigmoid"

model_classifier.add_layer (8,8, activation = "sigmoid"

model_classifier.add_layer (8,1, activation = "sigmoid"
```

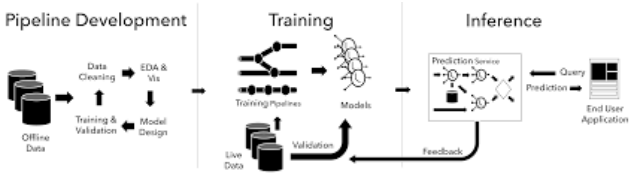
Activation function: sigmoid
Class bias ratio: 1:4

Final optimization:
Based on the semantic importance of some of the features we have in the dataset, we decided to give them higher weights to give them higher priority in the learning process. This is done by multiplying the normalized data after cleaning by a factor of 10. These features are:

- Income
- # Terms
- Loan amount
- Credit score

Utility Application:

After creating, training and testing our model (the training phase) we need to move to the next step which is to provide a way for the user to use our algorithm to predict whether a loan default will happen or not (the inference phase).



Why Create a Web app?

Our intended users are bank employees or people who are interested in taking up a loan, it is not wise to assume that our intended demographic will have the required programming knowledge to be able to run the model on jupyter notebooks for example. For this reason we need to create either an API or a complete web application. Because we need a user-friendly interface so our end users can benefit from our model we chose not to implement an API like for example FastAPI and we instead will create a erb application

Architecture

For the utility application we followed the architecture design we proposed in the previous

phase. We implemented the MVC architecture (Model View Controller), however we decided not to use Django in the implementation, instead we used streamlit. Streamlit is a python based app framework that has been designed for Machine Learning and Data Science purposes and has been growing in popularity in recent years.

Choosing a suitable framework

First thing we had to do was to choose a framework to use for our web application. After some searching we found that Django, Flask were 2 of the most popular python web frameworks. However, although not as popular we choose streamlit. There are a lot of reasons for why we preferred Streamlit over the other two options.

- Both Django and Flask would require front-end development, to produce a user-friendly GUI, while Streamlit does not.
- According to Streamlit it was made for machine learning and Data science teams.
- Streamlit provides the option to deploy the application on “Streamlit cloud” easily and for free.
- Only python knowledge is needed
- Streamlit is a full dashboarding solution while Flask and Django is not.
- Streamlit is running on Tornado and Flask web frameworks.

References:

- Bhattacharyya, S. (2021, February 22). *A loss function suitable for class imbalanced data: "Focal loss"*. Medium. Retrieved April 8, 2022, from https://towardsdatascience.com/a-loss-function-suitable-for-class-imbalanced-data-focal-loss-af1702d75d75
- JohnJohn 5311 silver badge44 bronze badges, oW_♦oW_ 5, Dhruv MahajanDhruv Mahajan 34811 silver badge1111 bronze badges, & PaulVDPaulVD 3111 bronze badge. (1965, April 1). *Is Gini coefficient a good metric for measuring predictive model performance on highly imbalanced data*. Data Science Stack Exchange. Retrieved April 8, 2022, from https://datascience.stackexchange.com/questions/19755/is-gini-coefficient-a-good-metric-for-measuring-predictive-model-performance-on
- Brownlee, J. (2020, August 20). *Cost-sensitive decision trees for imbalanced classification*. Machine Learning Mastery. Retrieved April 8,2022, from https://machinelearningmastery.com/cost-sensitive-decision-trees-for-imbalanced-classification/
- https://streamlit.io/
- https://blog.streamlit.io/how-to-master-streamlit-for-data-science/
- https://www.datarevenue.com/en-blog/data-dashboarding-streamlit-vs-dash-vs-shiny-vs-voila
- https://blog.streamlit.io/how-to-master-streamlit-for-data-science/
- kdnuggets.com/2021/04/deploy-machine-learning-models-to-web.html