Predicting Student Dropouts in Higher Education

STA 141C Final Project

Celine Nygaard Weiseth, Timothy Shen, Melanie Buck, Leah Khan, Aurian Saidi

Abstract

Abstract Student dropouts in higher education continue to increase year by year. Currently, higher education institutions lack the necessary comprehensive support system required for a typical diverse academic landscape. Fortunately, educational institutions collect vast amounts of data on their student's attributes giving way for potential analysis. Thus we aim to provide a proof-of-concept classification model for predicting student dropouts in higher education. Past research on analyzing education datasets largely centers on three classification models: logistic regression, support vector machines, and random foresting. Therefore we attempt to investigate these three models in their viability in predicting student dropouts. Logistic regression showed high predictive accuracy in identifying student dropouts across all different iterative methods. However, Gradient Descent signified a more efficient computational speed, especially under significantly larger datasets. Furthermore, Support Vector Machines (SVM) under a linear kernel showed similar high predictive accuracy, but are comparatively more computationally inefficient. Random Foresting showed relatively similar accuracy; however like SVM, it was computationally inferior to that of Logistic Regression. Finally, we propose further research in subsampling techniques to remedy the skewed nature of university datasets.

Keywords: *Classification Models; Logistic Regression; Newton's Method; Gradient Descent; Support Vector Machines, Random Foresting*

1. Introduction

Almost one in three students drop out of college before completing their degree [4]. There are vast amounts and combinations of factors that can contribute to students dropping out, for example, financial concerns, mental health, and external impacts like political instability [4]. It is imperative for higher education institutions to understand and establish methods to address the diverse learning styles and academic performances of students. Predicting and anticipating potential difficulties that students may face is a critical aspect of building effective support and guidance strategies for those at risk of academic failure or dropout.

Fortunately, many college institutions annually collect large amounts of data on student demographics, socio-economic factors, and academic performance. This presents an opportunity to leverage classification approaches on these datasets to predict student performance. This could provide a way for colleges to identify and support struggling students to enhance academic performance.

Past research has successfully implemented classification models analyzing large datasets from universities[1][3]. Under this literature, logistic regression, support vector machines, and random foresting emerge as the three most popular classification models. Therefore, this research paper aims to analyze the applicability of these three models in predicting student dropouts in higher education. Our research will center around the dataset from the Polytechnic Institute of Portalegre, Portugal, and then simulated datasets for larger universities. For each classification model, we hope to answer its viability under two contexts: predictive accuracy and computational speed.

1.1. Dataset

The dataset of interest comes from the UCI Machine Learning Repository which contains background information on students at the Polytechnic Institute of Portalegre, Portugal from 2008-2019[6]. The dataset includes 4,424 students across 37 categorical and quantitative variables. Variables include demographic status (age, nationality, etc.), socioeconomic status (parent's occupation, debt, grants, etc.), and academic information (grade upon admission, enrolled courses, high school courses, etc.). It also includes a target variable that defines a student as a "dropout", "enrolled", or "graduate" at the end of the normal duration of their degree program. Those who are "enrolled" are identified as students who are still in the process of completing their degree beyond the normal duration.

Table 1. Dataset from Polytechnic Institute of Portalegre, Portugal

Attributes	Description	Data Type
Marital Status	1-single 2-married 3-widower etc.	Discrete
Application Mode	1-1st Phase 2-Special 1st Phase 3- International etc.	Discrete
Application Order	1-First Choice ... 9-Last Choice	Discrete
Course	1-Non-STEM 2-STEM	Discrete
Daytime/Evening Attendance	1-daytime 2-evening	Discrete
Previous Qualification	1-Secondary Education 2-Bachelor's etc.	Discrete
Previous Qualification (Grade)	Grade of Previous Qualification (0-200)	Continuous
Nationality	1-Portuguese 2-German, etc.	Discrete
Mother's Qualification	1-Secondary Education 2-Bachelor's	Discrete
Father's Qualification	1-Secondary Education 2-Bachelor's	Discrete
Mother's Occupation	0-Student 1-Blue Collar etc.	Discrete
Father's Occupation	0-Student 1-Blue Collar etc.	Discrete
Admission Grade	Grade upon Admission (0-200)	Continuous
Displaced	1-yes 0-no	Discrete
Educational Special Needs	1-yes 0-no	Discrete
Debtor	1-yes 0-no	Discrete
Tuition Fees Up-to-Date	1-yes 0-no	Discrete
Gender	1-yes 0-no	Discrete
Scholarship Holder	1-yes 0-no	Discrete
Age at Enrollment	Age of Student Upon Enrollment	Discrete
International	1-yes 0-no	Discrete
Curricular Units 1st sem (credited)	Credited Units	Discrete
Curricular Units 1st sem (enrolled)	Enrolled Units	Discrete
Curricular Units 1st sem (evaluations)	Graded Units	Discrete
Curricular Units 1st sem (approved)	Approved Units	Discrete
Curricular Units 1st sem (grade)	Average Grade (0-20)	Discrete
Curricular Units 2nd sem (credited)	Credited Units	Discrete
Curricular Units 2nd sem (enrolled)	Enrolled Units	Discrete
Curricular Units 2nd sem (evaluations)	Graded Units	Discrete
Curricular Units 2nd sem (approved)	Approved Units	Discrete
Curricular Units 2nd sem (grade)	Average Grade (0-20)	Discrete
Unemployment Rate	Percentage (%)	Continuous
Inflation Rate	Percentage (%)	Continuous
GDP		Continuous
Target	1-Dropout 2-Enrolled 3-Graduated	Discrete

1.2. Exploratory Data Analysis

During the pre-processing, we removed any outliers in the dataset. Then conducted a factor analysis of mixed data (FAMD) to further investigate the dataset.

1.2.1. Factor Analysis

Since the data contains 36 features, it should ideally be reduced in dimension to simplify any models and reduce the risk of overfitting. Due to the mixed nature of our data – it contains both continuous and categorical variables – PCA is not a suitable dimension reduction technique. Instead, an alternative method was used, factor analysis of mixed data (FAMD).

The table below, which contains the first five components, displays the results of the FAMD. The components it constructed explain woefully little variance and are useless. No benefit is gained in terms of either data visualization or feature selection. Therefore, the components aren't used in any part of this study.

Table 2. FAMD Results

Component	Eigenvalue	% Variance	% Variance (Cumulative)
1	37.401	1.82	1.82
2	28.514	1.39	3.21
3	27.191	1.33	4.54
4	23.206	1.13	5.67
5	20.944	1.02	6.69

2. Proposed Methods

For the sake of classification, we define our target variable, y_i, as the following (Table 3):

Table 3. Target Variable

Target Variable	Student Status
0 - Failure	dropout
1 - Success	enrolled, graduate

We included *enrolled* as part of success for two reasons. Future real-life models from universities will also contain observations from students who will still be enrolled after four years. Removing them during training would eliminate an entire class of data that the model is bound to encounter after deployment. Furthermore, we are most concerned with empowering universities to prevent students from dropping out, an outcome that wastes university resources and damages the institution's reputation, whereas a student who remains enrolled after four years presents no such problems. On the contrary, universities may even favor such students as they generate more revenue. Therefore, for the sake of performing binary classification, we group *enrolled* together with *graduate* as a single class.

2.1. Logistic Regression

Suppose you have a sample, $x_1, .., x_n$ where $X \in \mathbf{R}^n$. Then we define the linear model: $Y = \beta^T X$ where β are coefficients. It follows that

$$h_\beta(x_i) = \frac{1}{1 + e^{-\beta^T X}} \tag{1}$$

transforms the linear combination, Y, to the range $[0,1]$. Then the probability mass function (pmf) of Y is defined as[1]

$$P(y|x_i,\beta) = h(x_i)^y(1-h(x))^{1-y}. \tag{2}$$

Therefore, through logistic regression we aim to find β such that

$$\hat{\beta} = \arg\max_{\beta} f(\beta|Y) = \arg\max_{\beta} \log f(\beta|y) \tag{3}$$

where $f(\beta|Y)$ is the likelihood function of $P(Y|\beta)$

2.1.1. Gradient Descent

Gradient Descent (GD) is a first-order optimization iterative algorithm. Here we apply Gradient Descent to find the optimal β by minimizing the negative of the log-likelihood function of $f(\beta|Y)$, denote $l(\beta)$:[1]

$$\hat{\beta} = \arg\min_{\beta} -l(\beta). \tag{4}$$

The algorithm works by first starting at an initial point, β_0. Then at each iteration:

1. Determines the direction through the gradient of $l(\beta)$:[2]

$$\nabla(l(\beta)) = X^T(y-h(x_i)) \tag{5}$$

2. Then the gradient is scaled by a set step-size parameter, ν
3. Then is subtracted from the initial point for the new β.

The iteration repeats until a convergence threshold is met. Numerically, at each iteration it updates $\hat{\beta}$ the parameter:

$$\hat{\beta}^{(t+1)} = \hat{\beta}^{(t)} - \nu \cdot \nabla l(\beta^{(t)}) \tag{6}$$

2.1.2. Logistic Regression with Newton's Method

Newton's Method (NM) is a second-order optimization iterative algorithm. We use Newton's Method to find the optimal β through maximizing the log-likelihood function, $l(\beta)$:

$$\hat{\beta} = \arg\max_{\beta} l(\beta) \tag{7}$$

The algorithm is similar to GD, in which it starts at an initial point, β_0. Then for each iteration:

1. Like GD, finds the gradient of $l(Y|\beta)$ (Eq. 5)
2. Then computes the inverse Hessian matrix, second-order of $l(\beta)$:[3]

$$H^{-1} = -X^T DX \tag{8}$$

where D is the diagonal of $h(X)(1-h(X))$

3. Updates the initial point for a new β

$$\hat{\beta}^{(t+1)} = \hat{\beta}^{(t)} - H^{-1} \cdot \nabla l(\beta^{(t)}) \tag{9}$$

Computing the Inverse of the Hessian Matrix

Computationally, it is incredibly expensive to compute the inverse of a matrix. Therefore, we employ matrix decomposition in three ways to solve the H^{-1} in our Newton's Method Algorithm:

[1]Derivation of the pmf and log-likelihood are found in *Appendix B: Mathematical Proofs* 8.1
[2]Derivation of the Gradient is found in *Appendix B: Mathematical Proofs* 8.1
[3]Derivation of the Hessian Inverse is found in *Appendix B: Mathematical Proofs* 8.3

GE/LU Decomposition: $A = LU$. We decompose the Hessian Matrix, H (barring it must be positive semi-definite) to obtain a lower triangular matrix L and upper triangular matrix U. Then compute the inverse by solving $LUx = I$ through backward and forward substitution (I is the identity matrix). The result will be H^{-1}, since $HH^{-1} = I$.
Cholesky Decomposition: $A = RR'$. If the H is positive semi-definite, we can decompose H to an upper triangular matrix and its corresponding inverse. (Note: Cholesky Decomposition also works for $A = LL'$ where L is a lower triangular matrix) Then we solve the inverse $H^{-1} = R^{-1}R'^{-1}$.
QR Decomposition: $A = QR$. Through QR Decomposition, we decompose H to an orthogonal matrix Q and upper triangular matrix R. Then, H^{-1} is found by the following linear equation $Rx = Q^T I$ where x is the Hessian Inverse Matrix.

2.2. Support Vector Machine

The support vector classifier classifies an observation based on which side of a hyperplane it lies on. The classifier finds a hyperplane among the feature space such that most of the training data lies on the correct sides according to their class labels. It is found by optimizing: [7]

maximize M subject to

$$\sum_{j=1}^{p} \beta_j^2 = 1, \tag{10}$$

$$y_i(\beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + ... + \beta_p x_{ip}) \geq M(1 - \epsilon_i), \tag{11}$$

$$\epsilon_i \geq 0, \sum_{i=1}^{n} \epsilon_i \geq C, \tag{12}$$

where C is a non-negative tuning parameter and M is the width of the margin.

A support vector machine incorporates a kernel K which measures the distance between observations.

In the case of a linear kernel,

$$K(x_i, x_{i'}) = \sum_{j=1}^{p} x_{ij} x_{i'j} \tag{13}$$

whereas in a polynomial kernel of degree d,

$$K(x_i, x_{i'}) = (1 + \sum_{j=1}^{p} x_{ij} x_{i'j})^d \tag{14}$$

2.3. Random Foresting

Random forest is a classification model that combines the predictions of multiple decision trees to define a final outcome based on majority voting. A decision tree is defined as at a given node, w_i, it makes a binary decision from a set importance. Then goes to the decided node and makes another binary decision until it reaches a prediction. This is where random foresting comes in:[2]

1. Takes a random sample n with replacement
2. Constructs a decision tree from the n samples by:
   - 2a. Choosing m features randomly
   - 2b. Selects the best features among the m to split

2c. Split the node, w_i into two daughter nodes.

3. Then outputs the ensemble of trees, T_i
4. Then makes a prediction by majority vote.

3. Real Data Analysis

We test the predictive power of these methods on the Polytechnic Institute of Portalegre, Portugal (IP Portalegre) Dataset in predicting student dropouts.

3.1. Logistic Regression

For our data set, we applied Logistic Regression with our defined target variable (Table 1) as the, y. Then the remaining attributes of a student in IP Portalegre are the predictor variables, X. Before we begin fitting a model, we test whether the assumptions of a logistic regression hold true for a data set. We first check the presence of multicollinearity through a correlation matrix (Appendix C: 8) and find that variable type 'Curricular Units' are correlated (ie. grade, approved, enrolled, without evaluated, evaluated units). To remedy this, for each semester we kept 'Curricular Units' that were 'Evaluated' and its 'Grade' which eliminates multicollinearity. Then through our log-odds linear plots and QQ plot (Appendix C: 9), we verify the assumption of linearity between our continuous variables and logit response variables and independence of error. Thus now with our reduced dataset, all assumptions of logistic regression hold true.

3.1.1. Gradient Descent

The gradient descent (GD) for logistic regression is one of the mostly common methods in machine learning to classify which category an observation belongs to. In this context, our features are the variables within the different categories such as success and failure. GD minimizes the loss function by updating the weights. Due to the fact that Logistic Regression does not have a closed-form solution and its $l(\beta)$ is convex, we can use iterative optimization algorithms like Gradient Descent (GD) to approximate a solution. Our GD implementation will iteratively run until the previous and current β vectors reach a threshold of $\epsilon = 10^{-3}$. We implement our GD algorithm to classify our IP Portalegre Dataset. The dataset was split into 80% training and 20% test data randomly. To investigate the optimal stepsize parameter, α, we tested the algorithm under five different settings: $\alpha = [0.01, 0.25, 0.5, 0.75, 1]$. Running 100 times at each iteration splitting the dataset randomly, all five different parameters were computed.

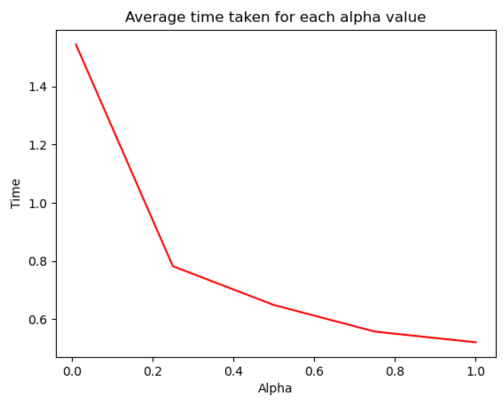


Figure 1. Average Computational Speed over different α

As α increases, it was found that the computational time decreases rapidly (Figure 1). In fact, when $\alpha = 1$, the Gradient Descent converged in 0.521 secs. It is also important to note the average accuracy is inaccurate at $\alpha = 0.1$, whereas the remaining αs showed a relatively high accuracy of 86%. (Figure 7)

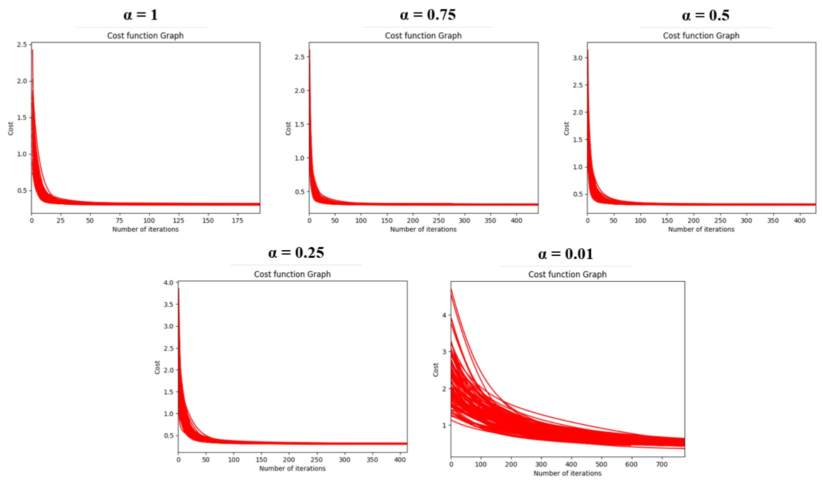


Figure 2. Cost Function, $l(\beta)$ at each iteration for every α

3.1.2. Newton's Method

We implement Newton's Method (NM) for Logistic Regression to classify our IP Portalegre Dataset to investigate the computational speed and predictive accuracy. Our NM Algorithm, similarily to GD, will iteratively run until our set convergence threshold of $\epsilon = 10^{-3}$. We employed our NM Algorithm in three ways. In each way, we use a different method when computing the Hessian Matrix: Cholesky Decomposition, QR Decomposition, and LU Decomposition. The dataset was split into 80% training and 20% test data randomly. Running 100 times with each iteration splitting the dataset randomly and all three methods being computed. In terms of computational speed, all three decomposition methods showed little difference with all of them running an average of 0.48 seconds. (Table 4)

Table 4. Average Computational Speeds on IP Portalegre Dataset

Decomposition Method	Speed
LU	0.489 secs
Cholesky	0.485 secs
QR	0.483 secs

In terms of predictive accuracy, NM Logistic Regression was on average accurate 87.5% of the time against the test dataset with a negligible variance around 10^{-5}.

3.1.3. Comparing Iterative Methods

Under the IP Portalegre Dataset with only 4,424 students, Logistic Regression with Newton's Method consistently had a higher computational speed than through Gradient Descent. This result would differ significantly when ran under larger datasets (See Section 4). We hypothesize that the relatively higher computational speed for Newton's Method in this instance is due to the relatively small size of the dataset. This is because, under smaller datasets, Newton's Method tends to converge much more rapidly than that of Gradient Descent.[5] The method also takes information directly from the likelihood function, the Hessian matrix, whereas Gradient Descent iterates linearly by a set stepsize. Thus under small datasets, like IP Portalegre's, where computing the inverse of the Hessian matrix is relatively computationally inexpensive, Newton's Method would be the optimal Logistic Regression classification method.

Both methods showed relatively similar with high predictive accuracy in identifying dropouts. However, they differ in their type of misclassifications. A confusion matrix (Figure 7) is produced

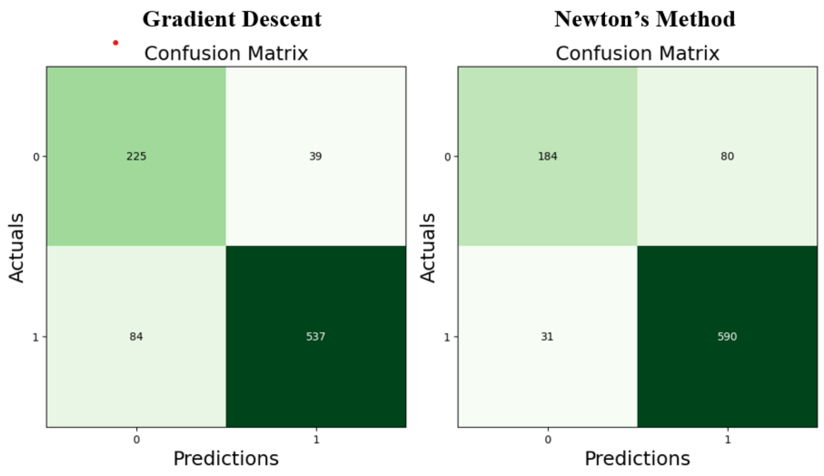


Figure 3. Confusion Matrix on 885 test data between the two methods

to further evaluate the predictive performance between the two Logistic Regression methods. For NM, false-positives (predicting a dropout as a success) were far more common than false-negatives (predicting a success as a dropout). Whereas for GD, false-negatives were much more frequent than false-positives. Therefore, although Newton's Method is relatively optimal for this dataset, in terms of computational speed, its false predictions suggest a heavier bias toward labeling a potential dropout as a success than vice versa. A far worse outcome than the latter.

3.2. Support Vector Machines

With the data containing 36 features and no useful components from FAMD, we weren't able to predetermine which kernel type is likely to suit the data. Instead, after scaling the continuous features, support vector machines (SVMs) with several kernel types were fitted using default hyperparameters, and their accuracies were compared. The table below (Table 5) displays the kernel types used and their corresponding prediction accuracy scores. We proceeded with a linear kernel, as its accuracy was clearly superior. It's also worth noting that since the data is imbalanced (non-dropouts exceed dropouts), the accuracies of the non-linear kernels may well be attributed to chance, explaining their similar scores. The regularization parameter, denoted as C, determines

Table 5. Initial SVM Fit Results

Kernel	% Accuracy
Linear	83
Quadratic	68
Cubic	68
Radial	68
Sigmoid	66

the model's tolerance for misclassifications while determining the hyperplane that separates the classes. To tune C, we used 5-fold cross-validation with F1 scores as the scoring method, as the F1 score accounts for the imbalanced class frequencies by measuring both precision and recall. The results of the cross-validation are displayed in the graph below (Figure 4). Note the very small range of values taken by the y-axis. There is essentially no difference in scores for different values of C. For the sake of simplicity, we proceed with the default value of C = 1.

Next, we perform recursive feature selection to reduce the number of variables in the model. The algorithm selected 18 out of the original 36 features to keep, and those are listed below. They are the features we use in the reduced model (Table 6).

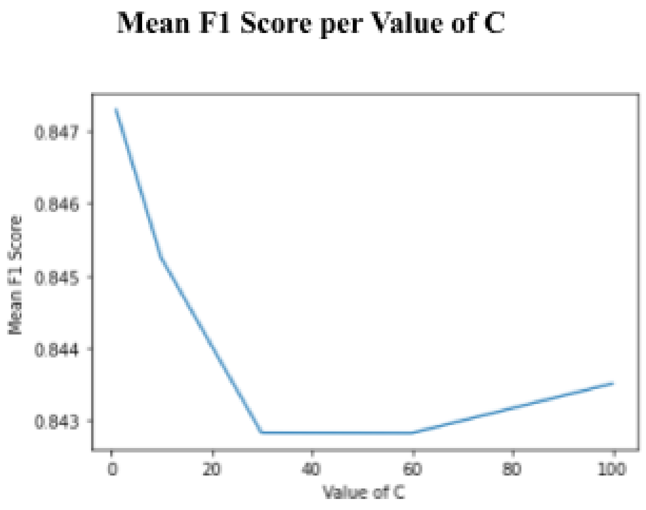


Figure 4. Mean F1 Score per Value of C

Table 6. Features For Reduced Model

Previous Qualification (Grade)	Admission Grade
Debtor	Tuition Fees Up to Date
Gender	Scholarship Holder
Age at Enrollment	International
Curricular Units 1st sem (enrolled)	Curricular Units 1st sem (evaluations)
Curricular Units 1st sem (grade)	Curricular Units 1st sem (without evaluations)
Curricular Units 2nd sem (enrolled)	Curricular Units 2nd sem (evaluations)
Curricular Units 2nd sem (grade)	Unemployment Rate

Upon performing 5-fold cross-validation on the reduced model, the mean F1 score is 0.8267, and the variance is near 0. This is roughly equivalent to the full model but slightly lower. As a final comparison between the full and reduced models, we plotted the confidence scores of each model across 5 resamples of the data, where the data were randomly split into new training and testing sets for each resample. The plots differentiate between the confidence scores for the training and testing sets in each resample. The corresponding prediction accuracy scores are included in each plot's title. For both the full (Figure 5) and reduced models (Figure 6), only the plots for the first two resamples are included for the sake of brevity, as the plots are exceptionally similar across resamples for their respective models. The confidence score for a given data point is defined as its distance from the separating hyperplane, with a higher distance indicating a greater confidence in the predicted class. Ideally, the data should be at least 1 unit away from the hyperplane, indicated by the dashed lines. Data classified as 'Dropout' are colored blue, and data classified as 'Not Dropout' are colored orange.

Figure 5. Full Model

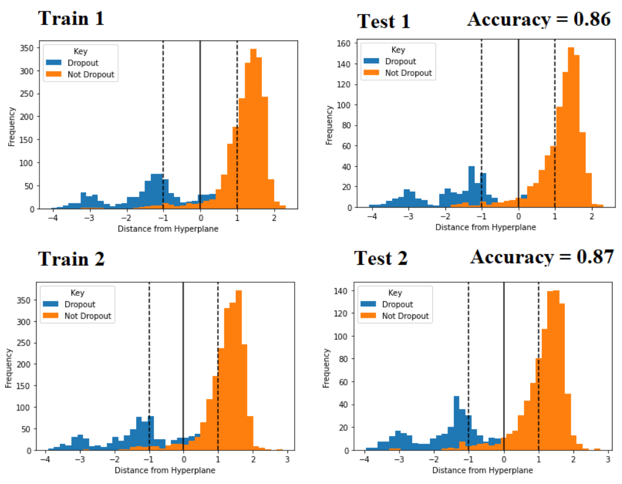
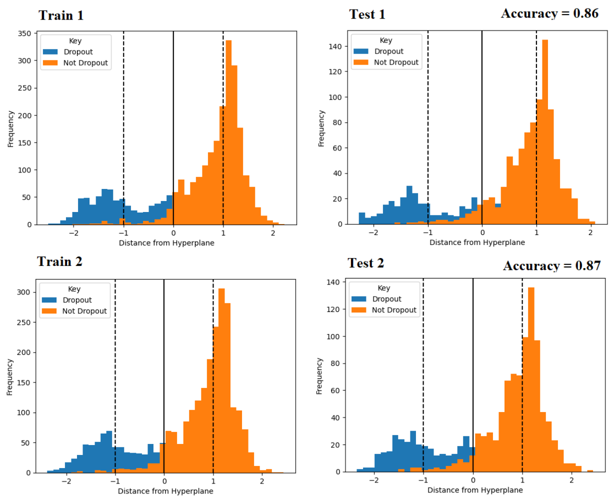


Figure 6. Reduced Model



3.3. Random Foresting

We implemented random foresting to identify student dropouts as our final classification model. Hyperparameter tuning was an important part of maximizing the performance of our random forest classifier. We focused on the max depth of a tree, maximum features, minimum samples, and total trees as our hyperparameters to tune. In order to prevent overfitting, a 10-fold cross validation was implemented to fine tune the parameter. Without this, the model may perform very well on a training set, but very poorly in a real world scenario. This increased the accuracy rate from 84% to 86%, a slight increase. Another method we utilized was grid search cross validation which we imported from the Sklearn library. This method tries out various combinations of hyperparameter values and finds the optimal combination. This again resulted in a similar accuracy of 86%.

However in terms of misclassification, the classifier had significantly more false-positives than false-negatives. Thus a noticeable bias of misclassifying graduates as dropouts compared to the other way around, as seen by the 99 counts of false positives compared to the 42 counts of false negatives. This bias can pose issues as the goal of the project is to identify students at risk of dropping out so they can receive the help they need. This causes at risk students to be identified at a disproportionately lower rate. Finally, in terms of computational speed, the classifier took around 8 minutes with all the hyperparameter tuning procedures. However the initial computational speed

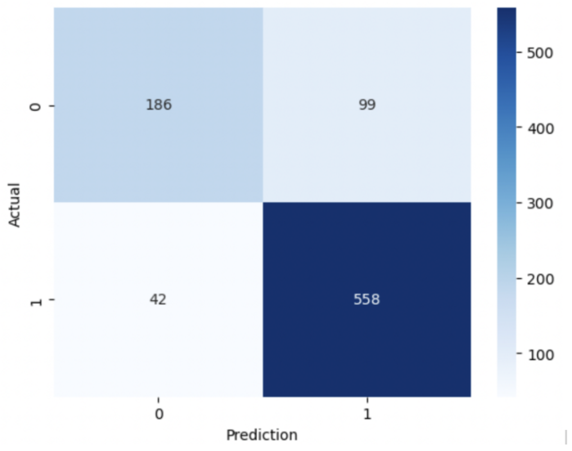


Figure 7. Confusion Matrix on the Test Data

took 3.5 seconds - still significantly slower than all the previous classification models.

4. Simulated Data Analysis

To understand the reproducibility of the model for larger universities. We evaluate the expectation of our algorithm on two simulated datasets. Our real dataset, (IP Portalegre), contained around 4,200 instances. Thus we examine our algorithms under simulated datasets of 10,000 instances and another of 50,000 instances. We ran each classifier 100 times for each simulated dataset, then collected their average predictive accuracy, the corresponding variance, and the computational time. Note that we did not run Random Foresting for our simulation study since the computational time for the real dataset is already significantly slow.

Table 7. Average Accuracy of Each Algorithm over 100 Iterations

Classification Algorithm	Accuracy (%) True Data (4,424)	Accuracy (%) (10,000)	Accuracy (%) (50,000)
LR with Gradient Descent, $\alpha = 1$	84.571	91.524	88.625
LR with Gradient Descent, $\alpha = 0.75$	84.589	91.523	88.632
LR with Newton's Method (LU)	87.5	93.123	91.240
LR with Newton's Method (QR)	87.5	93.123	91.240
LR with Newton's Method (Cholesky)	87.5	93.123	91.240
Support Vector Machine	0.826	0.920	0.907
Random Foresting			

In terms of accuracy (Table 7) both logistic regression iterative methods appear to reach their peak under the 10,000 dataset and then reduce in accuracy under the 50,000 dataset. However, both appear to have higher accuracy compared to the much smaller IP Portalegre dataset. Overall it seems the expectations of high accuracy across all methods appear to remain true at larger sample-size datasets.

It is quite clear under a simulation study on larger datasets (Table 8), Logistic Regression with Gradient Descent seems to be the most optimal. It is important to note, in terms of just Newton's Method for Logistic Regression, Cholesky decomposition appear to run faster than the other methods. In fact, LU decomposition significantly lags behind as the dataset size increases, Furthermore, it appears both NM Logistic Regression and Support Vector Machine become exponentially worse as the dataset increases. Whereas Gradient Descent seems to run efficiently at 50,000 instances.

Table 8. Average Computational Speed of Each Algorithm over 100 Iterations

Classification Algorithm	Speed (secs) True Data (4,424)	Speed (secs) (10,000)	Speed (secs) (50,000)
LR with Gradient Descent, $\alpha = 1$	0.521	0.388	0.773
LR with Gradient Descent, $\alpha = 0.75$	0.558	0.456	0.950
LR with Newton's Method (LU)	0.489	3.723	5.39
LR with Newton's Method (QR)	0.485	2.928	4.56
LR with Newton's Method (Cholesky)	0.483	2.851	4.85
Support Vector Machine	2.309	5.577	35.139

5. Conclusion

In this report, we present our findings on the ability of two classification models, Logistic Regression and Support Vector Machines, to predict student success in higher education in two contexts: predictive power and computational efficiency. Our aim was to determine the optimal classification model for higher education institutions to use in identifying potential student dropouts.

Our results showed that all classification methods had high accuracy in identifying student dropouts, with Logistic Regression consistently outperforming the other models in terms of predictive power. Therefore, for the highest predictive power, Logistic Regression appears to be the best classification model under this context.

Moreover, Logistic Regression also displayed significantly more efficient running times than any other models, especially with Gradient Descent. The Gradient Descent algorithm showed the most stability under larger datasets, while other classification algorithms performed exponentially worse as the sample size increased. In contrast, Gradient Descent only grew linearly. Thus, we conclude that Logistic Regression with Gradient Descent is the most viable model for higher education institutions to classify dropouts in terms of both predictive power and efficiency.

However, it should be noted the nature of the real dataset (IP Portalegre) presents a true problem for future similar university datasets: there will also be far more graduated and enrolled students compared to dropouts. The shortcomings were made apparent under the discussion of SVM (3.2). Thus more research is necessary for subsampling techniques to remedy the skewed nature of these datasets.

6. References

Bibliography

[1] Cédric Beaulac and Jeffrey S. Rosenthal. “Predicting university students’ academic success and major using random forests”. In: *Research in Higher Education* 60.7 (2019), pp. 1048–1064. DOI: 10.1007/s11162-019-09546-y.

[2] Alan F. Blackwell. *Objective Functions, Deep Learning and Random Forests*. May 2017. URL: https://www.cl.cam.ac.uk/~afb21/publications/Blackwell-ObjectiveFunctions.pdf.

[3] Anne-Sophie Hoffait and Michaël Schyns. “Early detection of university students with potential difficulties”. In: *Decision Support Systems* 101 (May 2017), pp. 1–11. DOI: 10.1016/j.dss.2017.05.003.

[4] Victoria Masterson. *More students are dropping out of college in the US – here’s why*. Sept. 2022. URL: https://www.weforum.org/agenda/2022/09/college-student-dropouts-2022/.

[5] Schmidt. *CPSC 540 - Machine Learning*. 2017. URL: https://www.cs.ubc.ca/~schmidtm/Courses/540-W17/L3.pdf.

[6] Machado Valentim and Baptista. *Predicting Students' Dropout and Academic Sucess*. 2021. URL: https://archive-beta.ics.uci.edu/dataset/697/predict+students+dropout+and+academic+success.

[7] Daniella Witten et al. *An Introduction to Statistical Learning with Applications in R Second Edition*. Springer, 2021.

7. Appendix A: Code

7.1. Import Libraries and Dataset

```
## Import Libraries
import pandas as pd
import numpy as np
import scipy.linalg as linalg
import matplotlib.pyplot as plt
import time
from prince import FAMD
from sklearn import svm
from sklearn.model_selection import train_test_split
from sklearn.model_selection import cross_val_score
from sklearn.feature_selection import RFE
from sklearn.datasets import make_classification
from sklearn import metrics
from mlxtend.plotting import plot_confusion_matrix

##Import Dataset
dropout = pd.read_csv("../Datasets/dropoutlr.csv", sep=",")
dropout = dropout.iloc[: , 1:]
X = dropout.drop(['Target'],axis=1)
y = dropout[["Target"]]
```

7.2. Factor Analysis with Mixed Data (FAMD)

```
'''
* Algorithm: Factor Analysis with Mixed Data
* Package: famd from prince
'''

#encode Target as 0=dropout, 1=graduate, 2=enrolled
df['Target'].replace(['Dropout', 'Graduate', 'Enrolled'], ['0', '1', '2'], inplace=True)

#convert nominal variables to strings
#otherwise, famd will treat them as quantitative and throw error "use PCA instead"
for col in ['Marital status',
            'Application mode',
            'Application order',
            'Course',
            'Daytime/evening attendance',
            'Previous qualification',
            'Nationality',
            "Mother's qualification",
            "Father's qualification",
            "Mother's occupation",
            "Father's occupation",
            'Displaced',
            'Educational special needs',
            'Debtor',
            'Tuition fees up to date',
            'Gender',
            'Scholarship holder',
            'International']:
    df[col] = df[col].apply(str)

#check data types
{df.columns[i]:x for i,x in enumerate(df.dtypes)}

#FAMD
famd = FAMD(n_components = 10, n_iter = 3, random_state = 101)
famd.fit(df.drop('Target', axis=1)) #leave out response variable 'Target'
famd.eigenvalues_summary
```

7.3. Logistic Regression Model Assumptions

```
'''
*   Algorithm: Logistic Regression Model Assumptions
'''

# import libraries
import numpy as np
import pandas as pd
import seaborn as sns

## Check for Multicollinearity: Correlation Matrix
corr_matrix = X.corr()
sns.heatmap(corr_matrix, cmap='coolwarm')

# Drop Correlated Variables
dropout = dropout.drop(['Curricular units 1st sem (enrolled)',
                "Curricular units 2nd sem (enrolled)",
                "Curricular units 1st sem (without evaluations)",
                "Curricular units 2nd sem (without evaluations)",
                "Curricular units 2nd sem (evaluations)",
                "Curricular units 1st sem (evaluations)"],axis=1)

## Check for Linearity
# continuous variables
cont = ["Previous qualification (grade)",
            "Admission grade",
            "Curricular units 1st sem (grade)",
            "Curricular units 2nd sem (grade)",
            "Unemployment rate","Inflation rate",
            "GDP"]
# Plot for Each Continuous Variable
for i in range(7):
    plt.figure()
    gre= sns.regplot(x= cont[i], y= "Target",
            data= dropout, logistic= True).set_title(
                "Log Odds Linear Plot")
```

7.4. Logistic Regression with Newton's Method

```
'''
*   Algorithm: Logistic Regression with Newton's Method
*   Code Partially Adapted From: Jia Rui Ong (2017)
'''
## Citation of Adapted Code Below ##
'''
* Title: Logistic Regression with Newton's Method
* Author: Jia Rui Ong
* Date: 2017
* Code Version: 1.0
* Availability: https://github.com/jrios6/Math-of-Intelligence/
'''
## This is our second method for running Logistic Regression.
## We first optimized by running a second-order optimization iterative
## technique of Newton's Method
## At each iteration it finds the Inverse Hessian Matrix.
## Thus will converge significantly faster when it finds f'=0
## To further optimize, we employed matrix decomposition in computing the Hessian Inverse
## We added LU/QR/Cholesky Decomposition method into our algorithm to do so
## Then tested to see witch one is optimal

# import libraries
import numpy as np
import scipy.linalg as linalg
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn import metrics
from mlxtend.plotting import plot_confusion_matrix

## Newton's Method Algorithm

# logistic function
def logfun(x):
    p = 1/(1+np.exp(-x)) # probability
    return p

# Updating beta Vector function
def newton_method_lrstep(beta0,y,X, method):
    betaX = np.dot(X,beta0) # beta*X
    yx = logfun(betaX) # y_hat
    y_hat = np.array(yx,ndmin=2) # converts to y_hat array
    gradient = np.dot(X.T, (y-y_hat)) # gradient

    ny_hat = 1-y_hat # 1-y_hat
    d = np.dot(y_hat,ny_hat.T) # finds D
    diag = np.diag(d) # computes diag of D
    hessian = X.to_numpy().T.dot(d).dot(X.to_numpy()) # hessian matrix
    n = hessian.shape[1] # ncol of hessian matrix
    # LU Decomposition to find Inverse
    if method == "LU":
        hessian_inv = np.linalg.inv(hessian)
    # QR Decomposition to find Inverse
    elif method == "QR":
        Q,R = np.linalg.qr(hessian)
        Rinv = linalg.solve_triangular(R,np.identity(n))
        hessian_inv = np.dot(Rinv,Q.T)
    # Cholesky Decompsition to find Inverse
    else:
        # uses LAPACK routines
        hessian_inv = torch.cholesky_inverse(hessian)

    gd = np.dot(hessian_inv,gradient) # finds the step direction

    beta = beta0 + gd # updates coefficients

    return beta # new vector

# Checks Convergence
def tolerance_check(beta0, beta, eps):
    diff = np.abs(beta0-beta) # norm
    if np.any(diff>eps): # if norm crosses threshold
        return False
    else:
        return True

# Logistic Regression via Newton's Method
def newton_method_logreg(beta0, y, X,eps,method):
    iterations = 0 # initial iterations
    converge = False # sets converge to false
    while not converge: # while converge is false
        beta = newton_method_lrstep(beta0,y,X,method) # finds new beta
        converge = tolerance_check(beta0,beta,eps) # checks convergence
        beta0 = beta # updates beta
        iterations +=1 # updates iterations
        print ("Iteration: {}".format(iterations))
    return beta

# Predictive Accuracy Function for Logistic Regression
def predict_lr(X,y,beta):
    xbeta = np.dot(X,beta) # x*beta
    predict = np.array(logfun(xbeta)) # predicted y
    threshold = 0.5*np.ones((predict.shape[0],1)) # sets 0.5 threshold
    pred_class = np.greater(predict,threshold) # converts y to 0,1
    accuracy = np.count_nonzero(np.equal(pred_class, y))/pred_class.shape[0] # checks accuracy
    return accuracy

## Testing Under True Dataset
accuracy = []
timeLU = []
timeQR = []
timeChol = []

# 100 Iterations with a Random Split of Training/Test Dataset
for i in range(100):
    # Splits data randomly
    X_train,x_test,y_train, y_test = train_test_split(X,y ,random_state = 10+i,
            test_size=0.20, shuffle=True)
    n = X.shape[1] # column of X
    beta0 = np.zeros((n,1)) # initial vector
    eps = 10**(-3) # set convergence threshold
    start_timeLU = time.time() # start time
    # Newton's Method via LU Decomposition
    temp = newton_method_logreg(beta0,y_train,X_train,eps,method="LU")
    end_timeLU = time.time() # end time
    timediffLU = end_timeLU-start_timeLU # time difference
    timeLU.append(timediffLU)

    start_timeQR = time.time() # start time
    # Newton's Method via QR Decomposition
    temp = newton_method_logreg(beta0,y_train,X_train,eps,method="QR")
    end_timeQR = time.time() # end time
    timediffQR = end_timeQR-start_timeQR # time difference
    timeQR.append(timediffQR)

    start_timeChol = time.time() # start time
    # Newton's Method via Cholesky Decomposition
    temp = newton_method_logreg(beta0,y_train,X_train,eps,method="Chol")
    end_timeChol = time.time() # end time
    timediffChol = end_timeChol-start_timeChol # time difference
    timeChol.append(timediffChol)

    acc = predict_lr(x_test,y_test,temp)
    accuracy.append(acc)

## Plotting Confusion Matrix
conf_matrix = metrics.confusion_matrix(y_test, predict) # confusion matrix
# Plotting Confusion Matrix
fig, ax = plot_confusion_matrix(conf_mat=conf_matrix, figsize=(6, 6), cmap=plt.cm.Greens)
plt.xlabel('Predictions', fontsize=18)
plt.ylabel('Actuals', fontsize=18)
plt.title('Confusion Matrix', fontsize=18)
plt.show()

## Testing Under Simulated Data

# Generated Dataset, 10000 instances
X,y = make_classification(n_samples=10000, n_features=36, n_informative=36,
                    n_redundant=0, n_repeated=0, n_classes=2, n_clusters_per_class=2,
                          class_sep=1.5)
X = pd.DataFrame(X)
y = pd.DataFrame(y)

# Generated Dataset, 50000 instances
X,y = make_classification(n_samples=50000, n_features=36, n_informative=36,
                    n_redundant=0, n_repeated=0, n_classes=2, n_clusters_per_class=2,
                          class_sep=1.5)
X = pd.DataFrame(X)
y = pd.DataFrame(y)

# note: same code as true dataset to iterate, loop ran for each generated dataset above
for i in range(100):
    X_train,x_test,y_train, y_test = train_test_split(X,y ,random_state = 10+i,
                test_size=0.20, shuffle=True)
    n = X.shape[1]
    beta0 = np.zeros((n,1))
    eps = 10**(-3)
    start_timeLU = time.time()
    temp = newton_method_logreg(beta0,y_train,X_train,eps,method="LU")
    end_timeLU = time.time()
    timediffLU = end_timeLU-start_timeLU
    timeLU.append(timediffLU)

    start_timeQR = time.time()
    temp = newton_method_logreg(beta0,y_train,X_train,eps,method="QR")
    end_timeQR = time.time()
    timediffQR = end_timeQR-start_timeQR
    timeQR.append(timediffQR)

    start_timeChol = time.time()
    temp = newton_method_logreg(beta0,y_train,X_train,eps,method="Chol")
    end_timeChol = time.time()
    timediffChol = end_timeChol-start_timeChol
    timeChol.append(timediffChol)

    acc = predict_lr(x_test,y_test,temp)
    accuracy.append(acc)

np.mean(accuracy)
np.var(accuracy)
np.mean(timediffLU)
np.var(timediffLU)
np.mean(timediffQR)
np.var(timediffQR)
np.mean(timediffChol)
np.mean(timediffChol)
```

7.5. Logistic Regression with Gradient Descent

```
'''
*   Algorithm: Logistic Regression with Gradient Descent
*   Code Partially Adapted From: Phong Hoangg
'''
## Citation of Adapted Code Below ##
'''
* Title: Logistic Regression with Gradient Descent
* Author: Phong Hoangg
* Date: 2017
* Code Version: 1.0
* Availability: shorturl.at/euwKQ
'''
## First method for Logistic Regression.
## Maximizes loss function
## Tested to see which learning rate is optimal in terms of accuray and time

# libraries
import time
import numpy as np
from math import e
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split

def generateXvector(X):
    """ Taking the original independent variables matrix and add a row of 1
    which corresponds to x_0
        Parameters:
          X:  independent variables matrix
        Return value: the matrix that contains all the values in the dataset,
        not include the outcomes values
    """
    vectorX = np.c_[np.ones((len(X), 1)), X]
    return vectorX

def theta_init(X):
    """Generate an initial value of vector  from the original independent variables matrix
         Parameters:
          X:  independent variables matrix
        Return value: a vector of theta filled with initial guess
    """
    theta = np.random.randn(len(X[0])+1, 1)
    return theta

def sigmoid_function(X):
    """ Calculate the sigmoid value of the inputs
         Parameters:
          X:  values
        Return value: the sigmoid value
    """
    return 1/(1+e**(-X))

def tolerance_check(beta0, beta, eps):
    """ Check tolerance againts difference
         Parameters:
          beta0: initial value
          beta: gradient
          eps: tolerance limit for cheking difference between beta0  and beta
        Return value: True if the differnece is bigger than tolerance
    """
    diff = np.abs(beta0-beta) # absolute value
    if np.any(diff>eps):
        return False
    else:
        return True

def Logistics_Regression(X, y, learningrate, eps):
    """ Find the Logistics regression model for the data set
         Parameters:
          X: independent variables matrix
          y: dependent variables matrix
          learningrate: learningrate of Gradient Descent
          eps: tolerance limit for cheking difference
        Return value: the final theta vector and the plot of cost function
    """
    y_new = np.reshape(y, (len(y), 1))
    cost_lst = []
    vectorX = generateXvector(X)
    theta = theta_init(X)
    m =  X.shape[0]
    iterations = 0
    #for i in range(iterations):
    converge = False
    while not converge:
        gradients = 2/m * vectorX.T.dot(sigmoid_function(vectorX.dot(theta)) - y_new)
        theta1 = theta - learningrate * gradients
        y_pred = sigmoid_function(vectorX.dot(theta))
        cost_value = - np.sum(y_new*np.log(y_pred) + ((1-y_new)*np.log(1-y_pred)))/(len(y_pred))
        converge = tolerance_check(theta,theta1,eps) # checks convergence
        theta = theta1
        iterations += 1
    #Calculate the loss for each training instance
        cost_lst.append(cost_value)

    return theta

#%% with cost function plot
def Logistics_Regression_plot_cost(X, y, learningrate, eps):
    """ Find the Logistics regression model for the data set
         Parameters:
          X: independent variables matrix
          y: dependent variables matrix
          learningrate: learningrate of Gradient Descent
          eps: tolerance limit for cheking difference
        Return value: the final theta vector and the plot of cost function
    """
    y_new = np.reshape(y, (len(y), 1))
    cost_lst = []
    vectorX = generateXvector(X)
    theta = theta_init(X)
    m =  X.shape[0]
    iterations = 0
    converge = False
    while not converge:
        gradients = 2/m * vectorX.T.dot(sigmoid_function(vectorX.dot(theta)) - y_new)
        theta1 = theta - learningrate * gradients
        y_pred = sigmoid_function(vectorX.dot(theta))
        cost_value = - np.sum(y_new*np.log(y_pred) + ((1-y_new)*np.log(1-y_pred)))/(len(y_pred))
        converge = tolerance_check(theta,theta1,eps) # checks convergence
        theta = theta1
        iterations += 1
    #Calculate the loss for each training instance
        cost_lst.append(cost_value)

    plt.figure()
    plt.plot(np.arange(1,iterations),cost_lst[1:], color = 'red')
    plt.title('Cost Function')
    plt.xlabel('Iterations')
    plt.ylabel('Cost')
    plt.xlim(0, iterations)
    return theta


#%% initial test
```

```
dropout = pd.read_csv('data.csv', header=0, sep=';')
dropout['Target'].replace(['Dropout', 'Graduate',"Enrolled"],[0, 1,1], inplace=True)

X = dropout.drop(['Target'],axis=1)
y = dropout[["Target"]]

X_train,X_test,y_train, y_test = train_test_split(X,y ,random_state=100, test_size=0.20, shuffle=T

sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)

eps = 10**(-3)
beta_init = Logistics_Regression(X_train,y_train, 1, eps)

#%% Testing - from online resource

def predict_lr(X,y,beta):
    xbeta = np.dot(X,beta)
    predict = np.array(sigmoid_function(xbeta))
    threshold = 0.5*np.ones((predict.shape[0],1))
    pred_class = np.greater(predict,threshold)
    accuracy = np.count_nonzero(np.equal(pred_class, y))/pred_class.shape[0]
    return accuracy, pred_class

acc, predict = predict_lr(X_test,y_test,beta_init[1:])

#%%
# from sklearn import metrics
# from mlxtend.plotting import plot_confusion_matrix
from sklearn.metrics import ConfusionMatrixDisplay
from sklearn.metrics import confusion_matrix

# conf_matrix = metrics.confusion_matrix(y_test, predict)
# fig, ax = confusion_matrix(conf_mat=conf_matrix, figsize=(6, 6), cmap=plt.cm.Greens)
# plt.xlabel('Predictions', fontsize=18)
# plt.ylabel('Actuals', fontsize=18)
# plt.title('Confusion Matrix', fontsize=18)
# plt.show()

# conf matrix TEST
cm = confusion_matrix(y_test, predict)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=['Dropout', 'Graduate'])
disp.plot(cmap='Blues')
plt.title('Confusion Matrix - Test')
plt.show()

# conf matrix TRAIN - maybe unecerasy
# cm = confusion_matrix(y_train, predict)
# disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=['Dropout', 'Graduate'])
# disp.plot(cmap='Blues')
# plt.title('Train')
# plt.show()
```

```
#%% Learning rate (alpha) exploration

# initialize lists to store values
accuracy1 = []
accuracy75 = []
accuracy25 = []
accuracy5 = []
accuracy01 = []

time1 = []
time75 = []
time25 = []
time5 = []
time01 = []

#%% alpha = 1
for i in range(100):
    X_train,X_test,y_train, y_test = train_test_split(X,y ,random_state = 10+i,
        test_size=0.20, shuffle=True)
    sc = StandardScaler()
    X_train = sc.fit_transform(X_train)
    X_test = sc.transform(X_test)
    eps = 10**(-3)

    start_time1 = time.time()
    beta = Logistics_Regression(X_train,y_train, 1, eps)
    end_time1 = time.time()
    timediff1 = end_time1 - start_time1
    time1.append(timediff1)
    temp = beta[1:]
    acc = predict_lr(X_test,y_test,temp)
    accuracy1.append(acc[0])


#%% alpha = 0.75
for i in range(100):
    X_train,X_test,y_train, y_test = train_test_split(X,y ,random_state = 10+i,
        test_size=0.20, shuffle=True)
    sc = StandardScaler()
    X_train = sc.fit_transform(X_train)
    X_test = sc.transform(X_test)
    eps = 10**(-3)

    start_time75 = time.time()
    beta = Logistics_Regression(X_train,y_train, 0.75, eps)
    end_time75 = time.time()
    timediff75 = end_time75 - start_time75
    time75.append(timediff75)
    temp = beta[1:]
    acc = predict_lr(X_test,y_test,temp)
    accuracy75.append(acc[0])


#%% alpha = 0.5
for i in range(100):
    X_train,X_test,y_train, y_test = train_test_split(X,y ,random_state = 10+i,
        test_size=0.20, shuffle=True)
    sc = StandardScaler()
    X_train = sc.fit_transform(X_train)
    X_test = sc.transform(X_test)
    eps = 10**(-3)

    start_time5 = time.time()
    beta = Logistics_Regression(X_train,y_train, 0.5, eps)
    end_time5 = time.time()
    timediff5 = end_time5-start_time5
    time5.append(timediff5)
    temp = beta[1:]
    acc = predict_lr(X_test,y_test,temp)
    accuracy5.append(acc[0])


#%% alpha = 0.25
for i in range(100):
    X_train,X_test,y_train, y_test = train_test_split(X,y ,random_state = 10+i,
        test_size=0.20, shuffle=True)
    sc = StandardScaler()
    X_train = sc.fit_transform(X_train)
    X_test = sc.transform(X_test)
    eps = 10**(-3)

    start_time25 = time.time()
    beta = Logistics_Regression(X_train,y_train, 0.25, eps)
    end_time25 = time.time()
    timediff25 = end_time25-start_time25
    time25.append(timediff25)
    temp = beta[1:]
    acc = predict_lr(X_test,y_test,temp)
    accuracy25.append(acc[0])


#%% alpha = 0.01
for i in range(100):
    X_train,X_test,y_train, y_test = train_test_split(X,y ,random_state = 10+i,
        test_size=0.20, shuffle=True)
    sc = StandardScaler()
    X_train = sc.fit_transform(X_train)
    X_test = sc.transform(X_test)
    eps = 10**(-3)

    start_time01 = time.time()
    beta = Logistics_Regression(X_train,y_train, 0.01, eps)
    end_time01 = time.time()
    timediff01 = end_time01-start_time01
    time01.append(timediff01)
    temp = beta[1:]
    acc = predict_lr(X_test,y_test,temp)
    accuracy01.append(acc[0])


#%% plotting the accuracy of the different learning rates
x_axis = np.arange(1,101)
x_axis = x_axis.tolist()

# alpha = 1
plt.figure()
plt.plot(x_axis, accuracy1, color = 'red')
plt.title('Accuracy alpha = 1')
plt.xlabel('Number of iterations')
plt.ylabel('Accuracy')

# alpha = 0.75
plt.figure()
plt.plot(x_axis, accuracy75, color = 'red')
plt.title('Accuracy alpha = 0.75')
plt.xlabel('Number of iterations')
plt.ylabel('Accuracy')

# alpha = 0.5
plt.figure()
plt.plot(x_axis, accuracy5, color = 'red')
plt.title('Accuracy alpha = 0.5')
plt.xlabel('Number of iterations')
plt.ylabel('Accuracy')

# alpha = 0.25
plt.figure()
plt.plot(x_axis, accuracy25, color = 'red')
plt.title('Accuracy alpha = 0.25')
plt.xlabel('Number of iterations')
plt.ylabel('Accuracy')

# alpha = 0.01
plt.figure()
plt.plot(x_axis, accuracy01, color = 'red')
plt.title('Accuracy alpha = 0.1')
plt.xlabel('Number of iterations')
plt.ylabel('Accuracy')

#%%
# x axis values (for plotting)
x_axis_alpha = [0.01, 0.25, 0.5, 0.75, 1]

# convert to np array in order to find mean
acc1_np = np.array(accuracy1)
acc75_np = np.array(accuracy75)
acc5_np = np.array(accuracy5)
acc25_np = np.array(accuracy25)
acc01_np = np.array(accuracy01)

acc_list = [acc01_np, acc25_np, acc5_np, acc75_np, acc1_np]
# boxplot
plt.figure()
plt.boxplot(acc_list)
plt.title('Accuracy boxplot for different learning rates')
plt.show()

# list of average accurasies for the five different alphas
```

```
avg_acc_list = [acc01_np.mean(), acc25_np.mean(), acc5_np.mean(),
    acc75_np.mean(), acc1_np.mean()]

# plot ACCURACY
plt.figure()
plt.plot(x_axis_alpha, avg_acc_list, color = 'red')
plt.title('Average accuracy for each alpha value')
plt.xlabel('Alpha')
plt.ylabel('Accuracy')


# convert to np array in order to find mean
time1_np = np.array(time1)
time75_np = np.array(time75)
time5_np = np.array(time5)
time25_np = np.array(time25)
time01_np = np.array(time01)

# list of average times for the five different alphas
avg_time_list = [time01_np.mean(), time25_np.mean(), time5_np.mean(),
    time75_np.mean(),time1_np.mean()]

# plot TIME
plt.figure()
plt.plot(x_axis_alpha, avg_time_list, color = 'red')
plt.title('Average time taken for each alpha value')
plt.xlabel('Alpha')
plt.ylabel('Time')


# plot COST
gd_01 = Logistics_Regression_plot_cost(X_train,y_train, 0.01, eps)
gd_25 = Logistics_Regression_plot_cost(X_train,y_train, 0.25, eps)
gd_5 = Logistics_Regression_plot_cost(X_train,y_train, 0.5, eps)
gd_75 = Logistics_Regression_plot_cost(X_train,y_train, 0.75, eps)
gd_1 = Logistics_Regression_plot_cost(X_train,y_train, 1, eps)

## Test on Simulated Data

# 10,000 Instances
X,y = make_classification(n_samples=10000, n_features=36, n_informative=36,
                    n_redundant=0, n_repeated=0, n_classes=2, n_clusters_per_class=2,
                          class_sep=1.5)

# 50,000 Instances
X,y = make_classification(n_samples=50000, n_features=36, n_informative=36,
                    n_redundant=0, n_repeated=0, n_classes=2, n_clusters_per_class=2,
                          class_sep=1.5)

# Initializing
X = pd.DataFrame(X)
y = pd.DataFrame(y)
eps = 10**(-3)
accuracy11 = []
accuracy175 = []
time11 = []
```

```
time175 = []

# 100 Iterations
for i in range(100):
    X_train,x_test,y_train, y_test = train_test_split(X,y ,random_state = 10+i,
        test_size=0.20, shuffle=True)
    sc = StandardScaler()
    X_train = sc.fit_transform(X_train)
    X_test = sc.transform(x_test)
    eps = 10**(-3)

    # Gradient Descent alpha = 1
    start_time11 = time.time()
    beta = Logistics_Regression(X_train,y_train, 1, eps)
    end_time11 = time.time()
    timediff11 = end_time11-start_time11
    time11.append(timediff11)
    temp = beta[1:]
    acc = predict_lr(x_test,y_test,temp)
    accuracy11.append(acc)

    # Gradient Descent alpha = 0.75
    start_time175 = time.time()
    beta = Logistics_Regression(X_train,y_train, 0.75, eps)
    end_time175 = time.time()
    timediff175 = end_time175-start_time175
    time175.append(timediff175)
    temp = beta[1:]
    acc = predict_lr(x_test,y_test,temp)
    accuracy175.append(acc)
```

7.6. Support Vector Machine

```
#min max scale numeric variables
def scale(column):
    return (df[column] - df[column].min()) / (df[column].max() - df[column].min())

for col in ['Previous qualification (grade)',
                'Admission grade',
                'Age at enrollment',
                'Curricular units 1st sem (credited)',
                'Curricular units 1st sem (enrolled)',
                'Curricular units 1st sem (evaluations)',
                'Curricular units 1st sem (approved)',
                'Curricular units 1st sem (grade)',
                'Curricular units 1st sem (without evaluations)',
                'Curricular units 2nd sem (credited)',
                'Curricular units 2nd sem (enrolled)',
                'Curricular units 2nd sem (evaluations)',
                'Curricular units 2nd sem (approved)',
                'Curricular units 2nd sem (grade)',
                'Curricular units 2nd sem (without evaluations)',
                'Unemployment rate',
                'Inflation rate',
                'GDP'
            ]:
```

```
    df[col] = scale(col)

#split into train and test sets
trainX, testX, trainY, testY = train_test_split(df.drop('Target', axis=1),
                                                 df['Target'], test_size=0.3, random_state=0)

#try various kernels ----------------------------------------

#linear
svmLinear = svm.SVC(kernel='linear')
fitLinear = svmLinear.fit(trainX, trainY)
testLinear = svmLinear.score(testX, testY)

#poly, degree = 3
svmSquare = svm.SVC(kernel='poly', degree=2)
fitSquare = svmSquare.fit(trainX, trainY)
testSquare = svmSquare.score(testX, testY)

#poly, degree = 3
svmCube = svm.SVC(kernel='poly', degree=3)
fitCube = svmCube.fit(trainX, trainY)
testCube = svmCube.score(testX, testY)

#radial
svmRadial = svm.SVC(kernel='rbf')
fitRadial = svmRadial.fit(trainX, trainY)
testRadial = svmRadial.score(testX, testY)

#sigmoid
svmSigmoid = svm.SVC(kernel='sigmoid')
fitSigmoid = svmSigmoid.fit(trainX, trainY)
testSigmoid = svmSigmoid.score(testX, testY)

print(testLinear)
print(testSquare)
print(testCube)
print(testRadial)
print(testSigmoid)

#tune C parameter -------------------------------------
mean_score = []
x = df.drop('Target', axis=1)
y = df['Target']

for C in [1, 10, 30, 60, 100]:
    model = svm.SVC(kernel='linear', C=C)

    scores = cross_val_score(model, x, y, scoring='f1_macro')

    mean_score.append(np.mean(scores))

plt.plot([1,10,30,60,100], mean_score)
plt.title('Cross Validation Score per Value of C')
plt.xlabel('Value of C')
plt.ylabel('Mean F1 Score')
plt.show()
```

```
#recursive feature selection -----------------------------
#train test split
trainX, testX, trainY, testY = train_test_split(df.drop('Target', axis=1),
                                                 df['Target'], test_size=0.3, random_state=0)

#perform rfe
model = svm.SVC(kernel='linear')
rfe = RFE(estimator = model)
fit = rfe.fit(trainX, trainY)
feature = pd.Series(data = fit.ranking_, index = trainX.columns)
sig_features = feature[feature==1].index

print(sig_features)

#cross validation on reduced model --------------------------
selected_df = df[['Previous qualification (grade)',
                'Admission grade',
                'Debtor',
                'Tuition fees up to date',
                'Gender',
                'Scholarship holder',
                'Age at enrollment',
                'International',
                'Curricular units 1st sem (enrolled)',
                'Curricular units 1st sem (evaluations)',
                'Curricular units 1st sem (approved)',
                'Curricular units 1st sem (grade)',
                'Curricular units 1st sem (without evaluations)',
                'Curricular units 2nd sem (enrolled)',
                'Curricular units 2nd sem (evaluations)',
                'Curricular units 2nd sem (approved)',
                'Curricular units 2nd sem (grade)',
                'Unemployment rate',
                'Target']]

selectedX = selected_df.drop('Target', axis=1)
y = selected_df['Target']

scores = cross_val_score(model, selectedX, y, scoring='f1_macro')

print(np.mean(scores))
print(np.var(scores))

#use basic resampling to evaluate the confidence scores --------------------------------
#train model again with reshuffled data, 5 times

for _ in range(5):
    trainX, testX, trainY, testY = train_test_split(selected_df.drop('Target', axis=1),
                                                     selected_df['Target'], test_size=0.3)

    model = svm.SVC(kernel='linear')
    model.fit(trainX, trainY)
    accuracy = model.score(testX, testY)

    #plot confidence scores of training data
    trainConfidence = pd.DataFrame(data={'Key': trainY.replace([0,1],['Dropout', 'Not Dropout']),
                                    'Dist': model.decision_function(trainX)})
    trainConfidence.pivot(columns='Key', values='Dist').plot.hist(bins=40)
    plt.axvline(x=-1, color='k', linestyle='--')
    plt.axvline(x=1, color='k', linestyle='--')
    plt.axvline(x=0, color='k')
    plt.xlabel('Distance from Hyperplane')
    plt.title('Training Data Confidence Scores')
    plt.show()

    #plot confidence scores of test data
    testConfidence = pd.DataFrame(data={'Key': testY.replace([0,1],['Dropout', 'Not Dropout']),
                                    'Dist': model.decision_function(testX)})
    testConfidence.pivot(columns='Key', values='Dist').plot.hist(bins=40)
    plt.axvline(x=-1, color='k', linestyle='--')
    plt.axvline(x=1, color='k', linestyle='--')
    plt.axvline(x=0, color='k')
    plt.xlabel('Distance from Hyperplane')
    plt.title('Test Data Confidence Scores')
    plt.show()

    confusion = confusion_matrix(testY, model.predict(testX))
    print(confusion)

    print(accuracy)

#simulations -------------------------------------------------------
#test on real data
scores = []
times = []
for _ in range(100):
    trainX, testX, trainY, testY = train_test_split(df.drop('Target', axis=1),
                                                df['Target'], test_size=0.3, shuffle=True)

    time1 = time.time()

    model = svm.SVC(kernel='linear')
    model.fit(trainX, trainY)
    accuracy = model.score(testX, testY)

    time2 = time.time() - time1

    scores.append(accuracy)
    times.append(time2)

print(np.mean(scores))
print(np.mean(times))

#generate dataset, n = 10,000
x,y = make_classification(n_samples=10000, n_features=36, n_informative=36,
    n_redundant=0, n_repeated=0,
    n_classes=2, n_clusters_per_class=2,class_sep=1.5)


x_10 = pd.DataFrame(x)
y_10 = pd.DataFrame(y)
```

```
#generate dataset, n = 50,000
x,y = make_classification(n_samples=50000, n_features=36, n_informative=36,
    n_redundant=0, n_repeated=0,
    n_classes=2, n_clusters_per_class=2,class_sep=1.5)

x_50 = pd.DataFrame(x)
y_50 = pd.DataFrame(y)

#time n = 10,000

time_10 = []
accuracy_10 = []

for _ in range(100):

    time1 = time.time()

    trainX, testX, trainY, testY = train_test_split(x_10, y_10, shuffle=True)


    model = svm.SVC(kernel='linear')
    model.fit(trainX, trainY)
    accuracy = model.score(testX, testY)

    time2 = time.time() - time1

    time_10.append(time2)
    accuracy_10.append(np.mean(accuracy))

print(np.mean(time_10), np.var(time_10), np.mean(accuracy_10), np.var(accuracy_10))

#time n = 50,000

time_50 = []
accuracy_50 = []

for _ in range(100):

    time1 = time.time()

    trainX, testX, trainY, testY = train_test_split(x_50, y_50, shuffle=True)


    model = svm.SVC(kernel='linear')
    model.fit(trainX, trainY)
    accuracy = model.score(testX, testY)

    time2 = time.time() - time1

    time_50.append(time2)
    accuracy_50.append(np.mean(accuracy))

print(np.mean(time_50), np.var(time_50), np.mean(accuracy_50), np.var(accuracy_50))
```

7.7. Random Foresting

```
'''
* Algorithm: Random Foresting
* Package: sklearn
'''

# Create a pipeline with feature scaling and random forest classifier
pipe = make_pipeline(StandardScaler(), RandomForestClassifier(max_depth=10, random_state=0))

# Fit the pipeline to the training data
pipe.fit(X_train, Y_train)

# Make predictions on the test data
Y_pred = pipe.predict(X_test)

# Evaluate the accuracy of the model without cross-validation
print("Without CV: ", accuracy_score(Y_test, Y_pred))

# Evaluate the accuracy of the model with cross-validation
scores = cross_val_score(pipe, X_train, Y_train, cv=10)
print("With CV: ", scores.mean())

print("Precision Score: ", precision_score(Y_test, Y_pred,average='micro'))
print("Recall Score: ", recall_score(Y_test, Y_pred,average='micro'))
print("F1 Score: ", f1_score(Y_test, Y_pred,average='micro'))

param_grid = {
    'bootstrap': [False,True],
    'max_depth': [5, 10, 15, 20],
    'max_features': [4, 5, 6, None],
    'min_samples_split': [2, 10, 12],
    'n_estimators': [100, 200, 300]
}

rfc = RandomForestClassifier()

clf = GridSearchCV(estimator=rfc, param_grid=param_grid, cv=10, n_jobs=-1, verbose=1)

clf.fit(X_train, Y_train)
best_rfc = clf.best_estimator_

y_pred = best_rfc.predict(X_test)
accuracy = accuracy_score(Y_test, Y_pred)
print("Accuracy: ", accuracy)
print("Best Hyperparameters: ", clf.best_params_)

# Evaluate cross-validation score for the best model
cv_score = cross_val_score(best_rfc, X_train, Y_train, cv=10)
print("Cross-validation score: ", cv_score.mean())
print(classification_report(Y_test,y_pred))
print(confusion_matrix(Y_test,y_pred))

# Confusion Matrix
cm = confusion_matrix(Y_test, y_pred)
ax = plt.subplot()
sns.heatmap(cm, annot=True, ax=ax, fmt='d', cmap='Blues')
```

```
ax.set_xlabel('Predictions')
ax.set_ylabel('Actuals')
ax.xaxis.set_ticklabels(['0', '1'])
ax.yaxis.set_ticklabels(['0', '1'])
plt.show()
```

8. Appendix B: Mathematical Proofs

8.1. Logistic Regression: Probability Mass Function

Suppose the same conditions as section 2.1. Then the linear combination of Y can be subject to the range [0,1] through the sigmoid function $h_\beta(x_i)$ (Eq. ??). Then it follows that the probability of y is defined as:

$$P(y|\beta) = \begin{cases} h_\beta(x_i) & y = 1 \\ (1 - h_\beta(x_i) & y = 0 \end{cases} \tag{15}$$

Then the probability mass function of $P(Y|x, \beta)$ follows a binomial distribution:

$$P(y|x, \beta) = h_\beta(x)^y (1 - h_\beta(x))^{1-y} \tag{16}$$

It follows that the log-likelihood function becomes:

$$l(\beta) = \Pi_{i=1}^n \log P(y_i|x_i, \beta) = \sum_{i=1}^n y_i \log(h_\beta(x_i)) + (1 - y_i) \log(1 - h_\beta(x_i)) \tag{17}$$

Notation: n is the sample size, β is the parameter of interest, y is the response variable, x are the predictor variables.

8.2. Logistic Regression: Gradient

Continuing from the section above 8.1. We find the derivative of the $l(\beta)$:

$$l'(\beta) = \sum_{i=1}^n [\frac{y_i h'_\beta(x_i)}{h_\beta(x_i)} - \frac{(1 - y_i)(h'_\beta(x_i))}{1 - h_\beta(x_i)}] \tag{18}$$

$$= \sum_{i=1}^n [\frac{y_i(x_i h_\beta(x_i)(1 - h_\beta(x_i)}{h_\beta(x_i)} - (\frac{(1 - y_i)((x_i h_\beta(x_i)(1 - h_\beta(x_i))}{h_\beta(x_i)}] \tag{19}$$

$$= \sum_{i=1}^n (y_i x_i(1 - h_\beta(x_i)) - (1 - y_i)(x_i h_\beta(x_i)) = \sum_{i=1}^n (y_i x_i - x_i h_\beta(x_i)) \tag{20}$$

$$= \sum_{i=1}^n x_i(y_i - h_\beta(x_i)) \tag{21}$$

Then in matrix form it becomes

$$\nabla l(\beta) = X^T(Y - h_\beta(X)) \tag{22}$$

8.3. Logistic Regression: Inverse Hessian

Continuing from section 8.2. We find the second derivative of $l(\beta)$:

$$l^2(\beta) = \frac{d}{d\beta} \sum_{i=1}^n x_i(y_i - h_\beta(x_i)) \tag{23}$$

$$= -\sum_{i=1}^{n} x_i(h'_\beta(x_i)) = -\sum_{i=1}^{n} x_i x_i (h_\beta(x_i))(1 - h_\beta(x_i)) \tag{24}$$

$$= -\sum_{i=1}^{n} x_i(h'_\beta(x_i)) = -\sum_{i=1}^{n} x_i x_i (h_\beta(x_i))(1 - h_\beta(x_i)) \tag{25}$$

In matrix form the scalar term $h_\beta(x_i)(1 - h_\beta(x_i))$ becomes the diagonal matrix W such that $W_{ii} = (h_\beta(x_i))(1 - h_\beta(x_i))$. So then the matrix form:

$$H = \nabla^2 l(\beta) = X^T W X \tag{26}$$

where W is the diagonal of $h_\beta(x_i)(1 - h_\beta(x_i))$.

9. Appendix C: Miscellaneous Tables

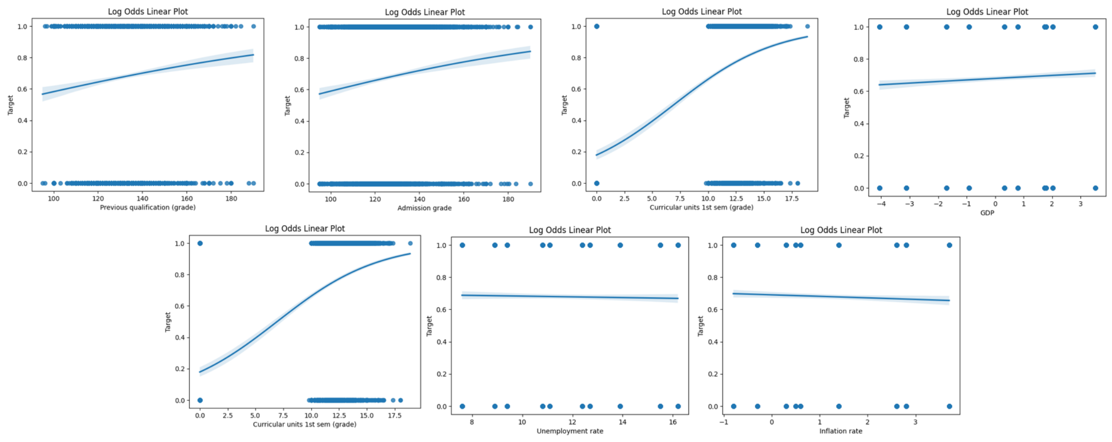


Figure 8. Log Linear Plots for Logistic Regression Assumptions

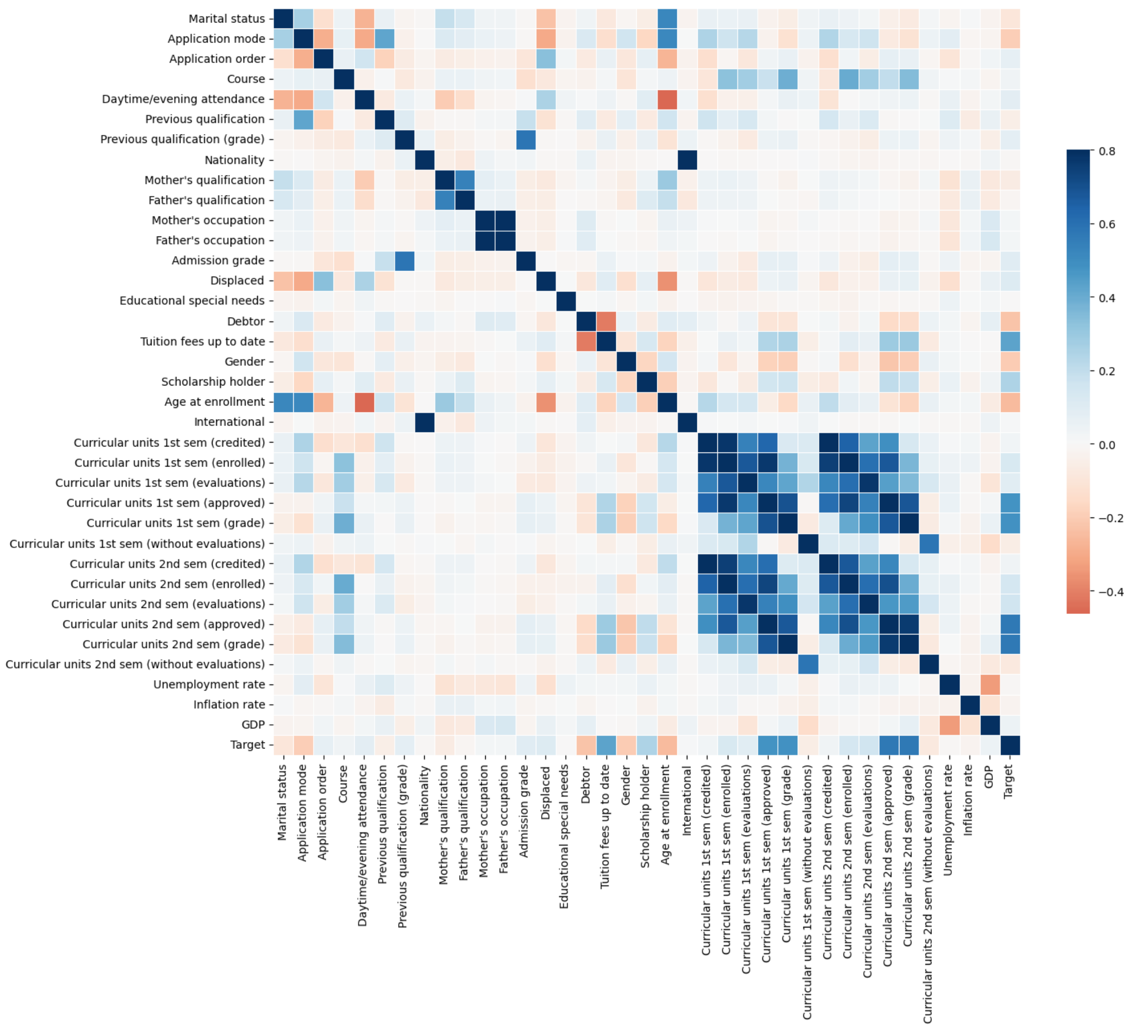


Figure 9. Correlation Matrix for Logistic Regression Assumptions