Feature Selection Using Metaheuristic Algorithms

Jack Bedinger
George Mason University
Fairfax, USA
jbedinge@gmu.edu

***Abstract*—The aim of this project is to use metaheuristic algorithms, specifically hill climbing and simulated annealing, to identify unimportant or detrimental features that can be removed, with the goal of improving the efficiency and performance of classification models. The algorithm was tested using five different PySpark classification models one at a time: LogisticRegression, DecisionTreeclassifier, RandomForestClassifier, GBTClassifier, and LinearSVC. All models saw improved F1 scores in almost all cases. Additionally, lists of the features most frequently dropped by each algorithm were obtained. Removing these without making any other adjustments resulted in around the same quality of predictions for these models.**

I. Introduction

One of the most significant obstacles facing any application of data mining is high dimensionality. Of course, machine learning models will generally be able to make better and more accurate predictions when more data are present to be trained on, but there is a significant trade-off in the form of high computational complexity. Trying to train a model and make predictions based on hundreds or thousands of variables can require an extremely large amount of computing power and time, making it infeasible for practical use when there are millions or billions of records in the dataset. Many of these variables can have mostly null or undefined values, or may actually be completely irrelevant for the predictions being made. To combat this, feature selection is often used to remove these unnecessary variables, making building and testing models faster and more efficient.

Metaheuristic algorithms are, as defined in a survey of the topic by Bianchi et al, "general algorithmic frameworks, often nature-inspired, designed to solve complex optimization problems" [1]. These find a variety of practical uses, including in machine learning, where they are frequently used for tuning model parameters. However, this experiment proposes a different use for these, instead utilizing them to determine which features in a high-dimensional dataset should be removed.

The dataset used to test this method of feature selection was provided by the Home Credit Group, as part of a competition held on Kaggle in 2018 [2]. Home Credit Group are a financial institution that specializes in lending to customers with little to no prior credit history. These customers often struggle to get loans or are taken advantage of by untrustworthy lenders. In order to determine which of these people will most likely be able to repay these loans, they use a wide variety of other information on each client, including where they live, where they work, and information about their telephone, car, family members, and more. The dataset used for this competition is 2.68 gigabytes in size and contains records for more than 350,000 clients spread across eight different CSV files, as illustrated in Figure 1. "application_train.csv" and "application_test.csv" contain information collected about clients who are applying for a loan as well as the loan itself. The training set contains a "TARGET" column, indicating whether this applicant had difficulties repaying this loan, while this is absent from the testing set. Clients may have one or more previous loans with Home Credit or a different financial institution, with information on these being found in the "previous_application.csv" and "bureau.csv" files, respectively. Additional information regarding these previous loans can in turn be found in "bureau_balance.csv" for loans from other institutions, and in "POS_CASH_balance.csv", "installments_payments.csv", and "credit_card_balance.csv" for Home Credit loans. In total, there are 221 features across all of these tables, necessitating an effective method to identify any which may be unimportant to determining an applicant's ability to repay their loan.

II. Approach

To fulfill the requirements of this assignment, programs were written in Python utilizing PySpark, the Python interface for Apache Spark [3]. Additionally, the Pandas [4] and Matplotlib [5] packages were used for creating the bar graphs shown in Figures 2 and 3.

The first Python program, "feature_engineering.py", was mainly used for initial importing of the data and feature engineering. First, each of the CSV files, with the exception of application_test.csv (which contains no TARGET column making judging performance impossible) and HomeCredit_columns_description.csv (which contains descriptions on the information each column represents) were imported as a Spark DataFrame. To prepare the data, any feature in any of the DataFrames which contained more than one percent null values was dropped. Then, any column with categorical string values was transformed to an integer column using StringIndexer. With this, new features could be engineered.

The original application file contained twenty different columns, each containing binary values indicating whether a client had provided a specific numbered form. The values for each of these were summed into a single feature, "FLAG_DOCUMENTS".

For the tables containing information on additional loans with Home Credit and/or other lenders, the corresponding DataFrames were filtered to only consider the most recent of

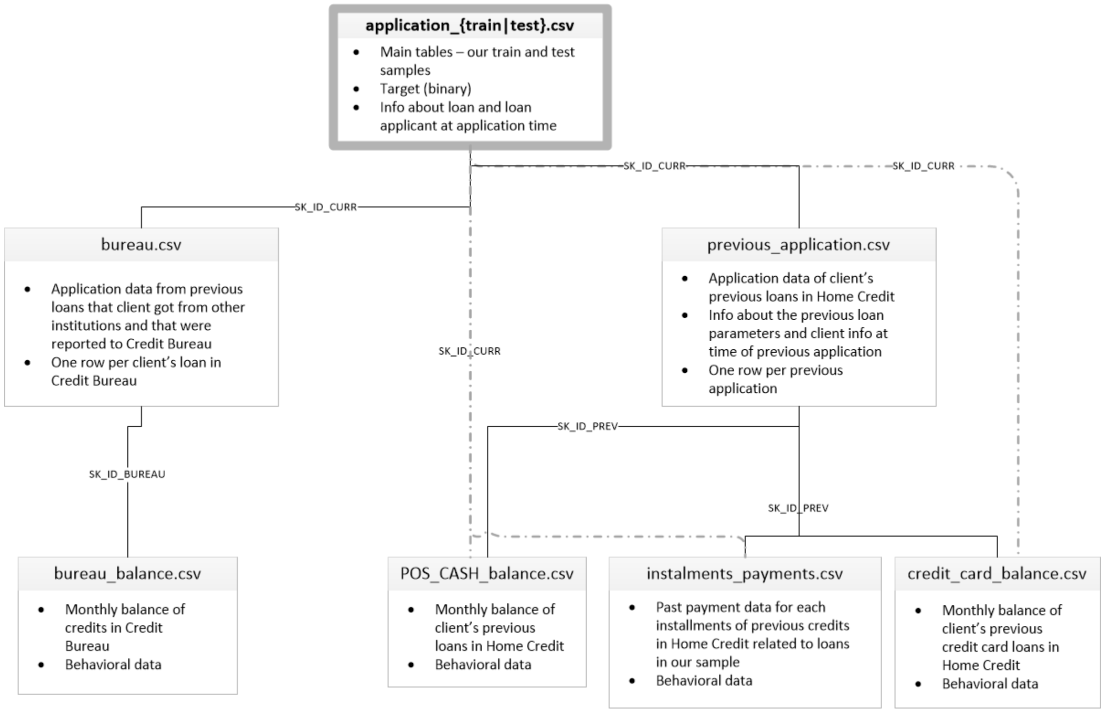
application_{train|test}.csv
Main tables – our train and test samples
Target (binary)
Info about loan and loan applicant at application time
SK_ID_CURR
SK_ID_CURR
SK_ID_CURR
SK_ID_CURR
bureau.csv
Application data from previous loans that client got from other institutions and that were reported to Credit Bureau
One row per client's loan in Credit Bureau
previous_application.csv
Application data of client's previous loans in Home Credit
Info about the previous loan parameters and client info at time of previous application
One row per previous application
SK_ID_BUREAU
SK_ID_PREV
SK_ID_PREV
bureau_balance.csv
Monthly balance of credits in Credit Bureau
Behavioral data
POS_CASH_balance.csv
Monthly balance of client's previous loans in Home Credit
Behavioral data
instalments_payments.csv
Past payment data for each installments of previous credits in Home Credit related to loans in our sample
Behavioral data
credit_card_balance.csv
Monthly balance of client's previous credit card loans in Home Credit
Behavioral data

Fig. 1. Visualization of the files included in the Home Credit dataset.

these previous loans, and the most recent relevant balances for each of these.

For the installments_payments table, a slightly different approach was taken. For each payment on a Home Credit loan made, a payment was marked as missed if it was paid after the loan was due ("DAYS_ENTRY_PAYMENT" > "DAYS_INSTALLMENT") or if the amount paid was less than the amount due ("AMT_INSTALLMENT" > "AMT_PAYMENT"). Then, the DataFrame was grouped by the previous client ID corresponding to the loan and the number of missed payments was obtained by summing all payments marked as missed.

After these new features were created, all of the DataFrames were joined together based on the applicant's current and previous IDs, as well as the associated IDs given to any outside financial bureaus. To handle any missing values for records that did not have associated previous loans from Home Credit and/or a different lender, integer columns were filled with the mode value for that column, and floating point column nulls were replaced with the mean.

The second program, "sampling.py", was used to perform either oversampling or undersampling on the newly merged DataFrame. This was done to counter the imbalance present in the dataset, as there are only around 25,000 records of applicants who had difficulties repaying loans out of the roughly 300,000 in the training set. Undersampling was performed using PySpark's sampleBy function to remove records with TARGET value 0 (indicating they had no payment issues) until the number of these roughly matched the number of records with TARGET value 1. For oversampling the minority class, sampleBy could not be used, so the DataFrame was split between the majority and minority classes. The minority DataFrame had its members duplicated by creating an array of numbers from zero to the rounded ratio of false to true TARGET values, which was then exploded. This temporary column was removed, and the oversampled minority DataFrame was unioned back with the majority DataFrame.

The third program, "feature_selection.py", implemented metaheuristic algorithms and tested them with different classification models provided by PySpark. The two specific metaheuristic algorithms implemented for this project are hill climbing and simulated annealing.

Hill climbing is a simple greedy search algorithm that keeps iterating until it stops improving. To better explain how it works, a brief walk-through of the algorithm follows:

1) Generate an initial solution x.
2) Repeat the following until there are no neighbors with a lower cost:
   a) Generate a set of "neighbor" solutions based on x.
   b) If there is a neighbor x_n in the set of neighbors with a cost lower than that of x, set x equal to x_n.

Simulated annealing is a bit more complex, mimicking the process by which metals are heated and then slowly cooled to reduce impurities. A more detailed explanation follows:

1) Generate an initial solution x.
2) Set the lowest cost c to the cost of x.
3) While the "temperature" T is greater than or equal to one and freezing factor f is greater than zero:
   a) Generate a set of "neighbor" solutions based on x.

b) Get the neighbor x_n from the set of neighbors with the lowest cost c_n. If one or more neighbors are tied for the lowest cost, randomly pick from one of these.
c) If c_n is lower than c, set x equal to x_n.
d) Otherwise, generate a random floating point number from $[0, 1)$. If this is less than $\exp(\frac{c-c_n}{T})$, set x equal to x_n.
e) If x was updated, reset f to its initial value. Otherwise, decrease it by one.
f) Multiply T by a "cooling factor" α between $[0, 1)$ (In this case, $\alpha = 0.95$).

Unlike hill climbing, simulated annealing is designed to prevent itself from getting stuck on a local minimum when there may be a better global minimum. It does this by giving itself a chance to move to a worse solution if a better solution is not found. This chance is higher when the algorithm first begins, then decreases as it goes on to prevent itself from moving away from the global minimum if it has found it.

For the purposes of this project, each "solution" is a dictionary consisting of all columns in the sampled DataFrame (excluding "SK_ID_CURR" and "TARGET"), as well as a one or zero to indicate whether this column should be included or dropped. "Neighbors" are generated by randomly selecting five of these columns and flipping its value from one to zero or vice versa. This way, a column could be added back after it has been dropped if including it improves the model's performance. Also, it should be noted that rather than searching for the lowest cost, these specific implementations for both hill climbing and simulated annealing search for the solution whose predictions yield the highest F1 score.

Testing was done by running the algorithms using five different PySpark classification models one at a time: LogisticRegression, DecisionTreeClassifier, RandomForestClassifier, GBTClassifier (which uses gradient-boosted trees), and LinearSVC (which uses Linear SVM). All of these models had default parameters and were not altered. This process was performed five times for each algorithm, with both the undersampled and oversampled dataset, for a total of fifty runs of each algorithm. For each combination of algorithm and dataset, the average of the five F1 scores obtained from the predictions yielded by the best solution was taken, as shown in Tables II and III. Additionally, the columns discarded by the best solution for each run of the algorithm were recorded, and are shown in Figures 2 and 3.

III. Challenges

As with all the assignments performed for this class, there were notable challenges faced when developing this project. One of these had to do with joining all of the DataFrames together. As stated previously, only the most recent previous loan from Home Credit and/or a third-party lender was considered for each applicant. However, even after filtering these DataFrames accordingly, joining them with the main application DataFrame resulted in numerous duplicate records. For the DataFrames obtained from bureau.csv and previous_application.csv, this filtering was performed by selecting only rows that had the maximum value for "DAYS_CREDIT" and "DAYS_DECISION", respectively. Both of these indicate how long ago these loans were taken or applied for relative to when the applicant filed for the loan in application_train.csv. This did not work properly at first because some clients had multiple loans with this maximum value. To combat this, the bureau DataFrame was additionally filtered by how recently the last information on the previous loan was obtained ("DAYS_CREDIT_UPDATE"), and any remaining duplicates for both had their float values averaged and their integer values replaced with the mode.

Originally, in addition to random oversampling and undersampling of the data, SMOTE (Synthetic Minority Oversampling TEchnique) was meant to be performed and tested. Rather than simply duplicating entries of the minority class, SMOTE creates new synthetic data by finding nearest neighbors for members of the minority class, randomly selecting a neighbor, and computing weighted averages and inserting these into the dataset. However, PySpark does not offer any native way to perform SMOTE, so this would have to be done manually. Like with performing oversampling, the dataset was split between entries with a TARGET value of 1 and 0. For the former, its entries were converted to vectors using VectorAssembler, and the distance between each pair of vectors was estimated using BucketedRandomProjectionLSH. Of the five closest neighbors, one was randomly chosen for each record and averaged to create a new synthetic record. This was fairly easy to accomplish, but another problem was soon encountered. In order to test the performance of each of the classification models, the implementations of hill climbing and simulated annealing also used VectorAssembler to convert records into vector form based on a list of columns (not including those to be dropped), and then passed these vectors to each model. Because there is no easy way to remove entries from vectors in PySpark, these synthetic vectors would need to be converted back into DataFrame columns. This would be done by using the function vector_to_array() to convert the entries into arrays, and then the columns would be reassembled from these arrays and given their original names. Unfortunately, vector_to_array() does not support Python 3.6.8, the version that is used by GMU's Perseus cluster. Ultimately, the decision was made to abandon SMOTE for this project.

Another much more minor issue came up when implementing the simulated annealing algorithm. As mentioned earlier, if no better solution is found, one with a higher cost can be selected if $\exp(\frac{c-c_n}{T}) >$ some random number in $[0, 1)$, where c is the cost of the current best solution, c_n is the cost of the neighbor, and T is the temperature. However, using F1 score to determine the quality of each solution caused the algorithm to always choose a worse neighbor. This is because any difference between F1 scores would be very small, so $\exp(\frac{c-c_n}{T})$ would always be incredibly close to one. To fix this, the difference was multiplied by an additional large constant (set to 10,000).

IV. Results

TABLE I
Performance Without Metaheuristics

Classifier	Avg. F1 Score w/o sampling	Avg. F1 Score w/ undersampling	Avg. F1 Score w/ oversampling
Logistic Regression	0.008	0.656	0.647
Decision Tree	0	0.620	0.622
Random Forest	0	0.649	0.627
Gradient-Boosted Trees	0.0004	0.657	0.658
Linear SVM	0	0.651	0.642

To show the improvements in quality caused by using hill climbing and simulated annealing for feature selection, Table I gives baseline averages for both the undersampled and oversampled data without either algorithm applied. Also included are average scores from training the models without sampling the data. Clearly, this sampling is vital in building an effective classifier when working with unbalanced data such as this. Because of the small number of positive TARGET values, these models would otherwise struggle with identifying applicants who had trouble paying their loans.

TABLE II
Performance With Hill Climbing

Classifier	Avg. F1 Score w/ undersampling	Avg. F1 Score w/ oversampling
Logistic Regression	0.658	0.647
Decision Tree	0.624	0.629
Random Forest	0.653	0.632
Gradient-Boosted Trees	0.663	0.660
Linear SVM	0.652	0.641

TABLE III
Performance With Simulated Annealing

Classifier	Avg. F1 Score w/ undersampling	Avg. F1 Score w/ oversampling
Logistic Regression	0.662	0.648
Decision Tree	0.644	0.641
Random Forest	0.655	0.634
Gradient-Boosted Trees	0.664	0.660
Linear SVM	0.654	0.641

Tables II and III show the quality of predictions made by each of the classification models after feature selection was performed using hill climbing and simulated annealing, respectively. With the exception of the Linear SVM classifier, all models tested had their F1 scores slightly improve as a result.

As stated previously, Figures 2 and 3 showcase which features in the combined DataFrame were dropped most frequently in the best solutions given by each algorithm. Because of how each algorithm operated, simulated annealing was much more likely to drop features than hill climbing. In fact, several runs of the latter ended before a better solution was found, leading to no features being dropped at all. Descriptions provided by Home Credit for each of these most frequently dropped features follow:

Dropped by hill climbing:

- NAME_PRODUCT_TYPE_INDEXED: “Was the previous application x-sell o walk-in”
- FLAG_CONT_MOBILE: “Was mobile phone reachable (1=YES, 0=NO)”
- NAME_CONTRACT_STATUS_PREV_APP_INDEXED: “Contract status (approved, cancelled, ...) of previous application”
- NAME_PORTFOLIO_INDEXED: “Was the previous application for CASH, POS, CAR, . . . ”
- OBS_30_CNT_SOCIAL_CIRCLE: “How many observation of client’s social surroundings with observable 30 DPD (days past due) default”
- NAME_CONTRACT_STATUS_POS_CASH_INDEXED: “Contract status during the month”
- FLAG_DOCUMENTS: See Approach section
- AMT_PAYMENT_TOTAL_CURRENT: “How much did the client pay during the month in total on the previous credit”
- HOUR_APPR_PROCESS_START: “Approximately at what hour did the client apply for the loan”

Dropped by simulated annealing:

- AMT_APPLICATION: “For how much credit did client ask on the previous application”
- FLAG_MOBIL: “Did client provide mobile phone (1=YES, 0=NO)”
- SELLERPLACE_AREA: “Selling area of seller place of the previous application”
- AMT_RECIEVABLE_PRINCIPAL: “Amount receivable for principal on the previous credit”
- NAME_CONTRACT_STATUS_CREDIT_CARD_INDEXED: “Contract status (active signed,...) on the previous credit”
- AMT_PAYMENT_TOTAL_CURRENT: See above
- FLAG_LAST_APPL_PER_CONTRACT_INDEXED: “Flag if it was last application for the previous contract. Sometimes by mistake of client or our clerk there could be more applications for one single contract”

Note that some of these column names have been altered in order to mark them as having been converted from a categorical string column to an integer column using StringIndexer (these have “_INDEXED” appended to them), or to signify which DataFrame they were originally part of before being joined together (appended with “_POS_CASH”, “_CREDIT_CARD”, or “_PREV_APP”).

To confirm the effectiveness of this method of feature selection, baseline averages were once again taken without use of metaheuristics, but this time with the most frequently dropped features listed above removed. The results can be seen in Table IV. Interestingly, when compared with the initial F1 scores seen in Table I, some combinations of sampling method and model performed better, while others did slightly worse.

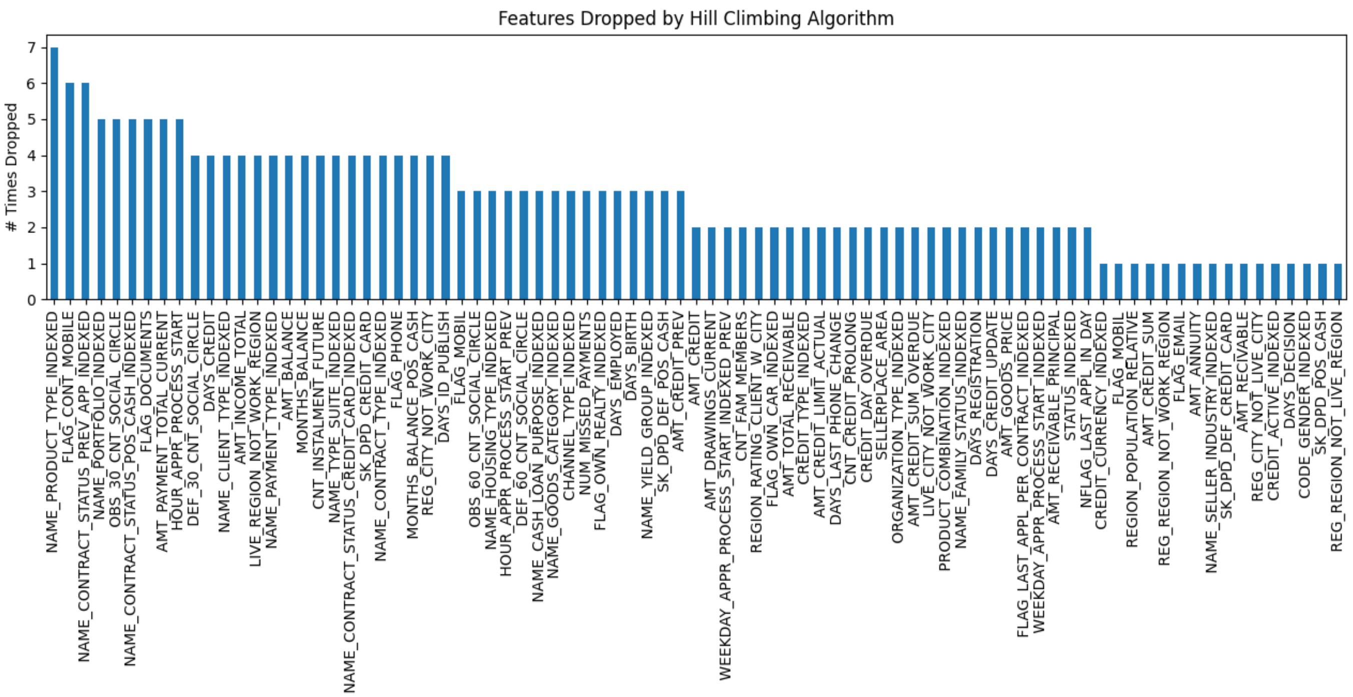

Fig. 2. Bar graph displaying the number of times each feature in the DataFrame was dropped by the Hill Climbing Algorithm.

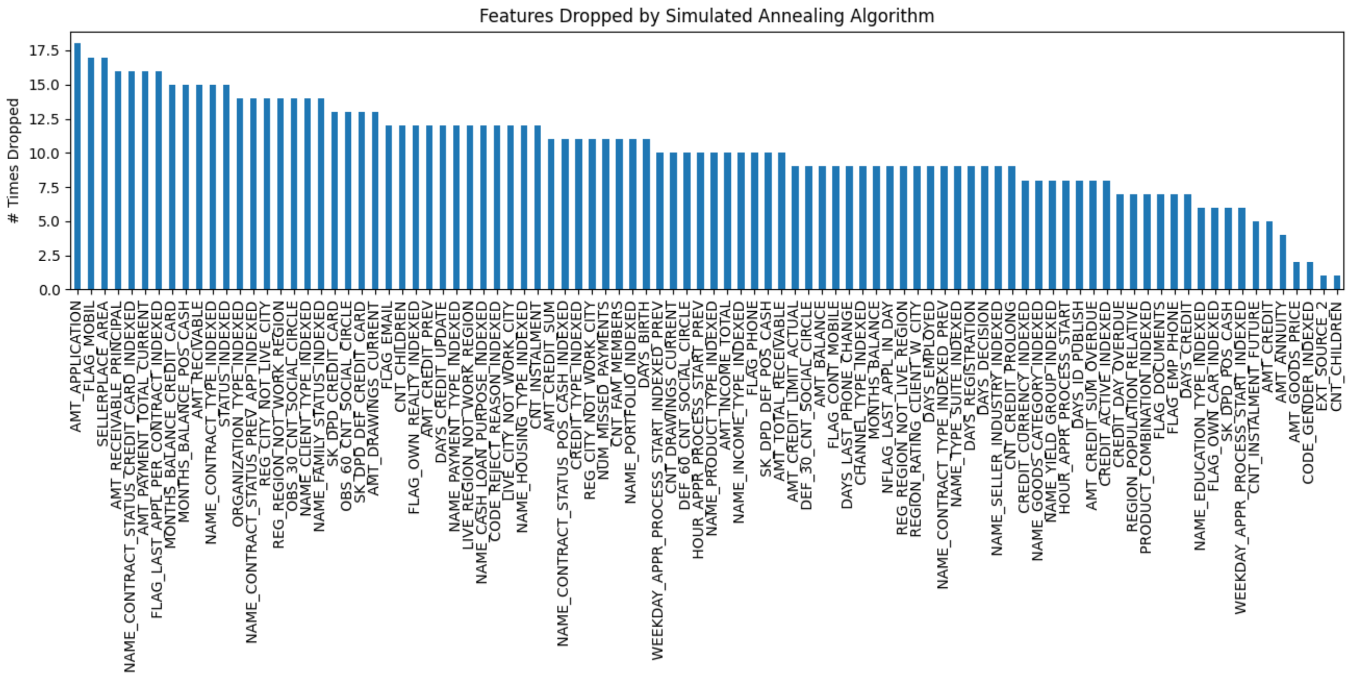

Fig. 3. Bar graph displaying the number of times each feature in the DataFrame was dropped by the Simulated Annealing Algorithm.

TABLE IV
PERFORMANCE WITH MOST FREQUENTLY DROPPED FEATURES OMITTED

Classifier	**Avg. F1 Score w/ undersampling**	**Avg. F1 Score w/ oversampling**
Logistic Regression	0.656	0.648
Decision Tree	0.623	0.621
Random Forest	0.647	0.621
Gradient-Boosted Trees	0.658	0.659
Linear SVM	0.649	0.641

V. CONCLUSIONS

To conclude, I feel that experiment was a success. Although results did not always improve after dropping the columns suggested by the algorithms, any decrease in F1 score was very minor. It is clear that more testing of this method needs to be done, perhaps with different datasets, other methods of sampling (such as SMOTE), and different classification models such as multi-layer perceptrons or a hybrid model similar to what was done for Assignment 4. If more repetitions of these algorithms were performed for each classifier, it is likely that we would be able to obtain a more definitive list of irrelevant features.

If I were to perform this experiment again, I would alter the metaheuristic algorithms such that they would also change the parameters of each model. I was initially considering having the implementation for each alternate between generating neighbors by dropping columns and changing hyperparameters each iteration, but I ran out of time and was unfortunately not able to do this. Personally, I feel that combining this method of feature selection with intelligent hyperparameter tuning through metaheuristics could be a powerful tool for creating accurate and precise classification models.

REFERENCES

[1] L. Bianchi, M. Dorigo, L. M. Gambardella, and W. J. Gutjahr, "A survey on metaheuristics for stochastic combinatorial optimization," *Natural Computing*, vol. 8, no. 2, pp. 239–287, Jun 2009. [Online]. Available: https://doi.org/10.1007/s11047-008-9098-4

[2] Home Credit Group, "Home credit default risk," https://www.kaggle.com/competitions/home-credit-default-risk, accessed: 2024-12-10.

[3] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin *et al.*, "Apache spark: a unified engine for big data processing," *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, 2016.

[4] W. McKinney *et al.*, "Data structures for statistical computing in python," in *Proceedings of the 9th Python in Science Conference*, vol. 445. Austin, TX, 2010, pp. 51–56.

[5] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in science & engineering*, vol. 9, no. 3, pp. 90–95, 2007.