MGMT 590

TEAM 4

Performing analysis on Car Purchase dataset

Sachin Arakeri | Anirudh Buchalli | Wenying Huang | Tirthankar Mukhopadhyay | Xema Vinod Pathak | Juily Vasandani

10-12-2019

Introduction

Applying the various algorithms and methods discussed in class to a sales dataset obtained from Kaggle, our team chose to focus our project mainly on building a predictive model for our target variable – from the initial exploratory visualization and analysis to hyperparameter tuning to find the best model – using an object-oriented programming approach. The objective of the project would be to keep all the logic written inside the classes we've created in order to ensure ease of replication for further improvements.

The model fits into a classification problem because the target variable determines whether a customer purchases a given commodity. This type of problem is incredibly important because determining the probabilities of a customer purchasing a given commodity is an everyday problem for any business. Given this information, businesses can adjust their sales strategies in order to take advantage of those groups that are more likely to purchase their product.

Across the scope of our project, we have written functions to perform every aspect of the predictive model building process, using the features of object-oriented programming to improve execution times as compared to using more traditional techniques, such as in-built functions within packages like pandas. Keeping all our functions within the class, we implemented various visualization packages and techniques for exploratory data analysis, implemented Monte Carlo simulations to calculate probabilities of purchasing the commodity per customer segment. compared time complexity between sorting techniques, and again to brute-force and randomized-search algorithms to find the ideal combination of parameters for hyperparameter tuning.

The dataset has approximately 23,500+ observations and 15 attributes giving whether the customer has purchased a car or not. As the dataset on Kaggle is only a year old, there are not many other related projects and those that do exist focus primarily on building various predictive models – ranging from an XGBoost model to a Random Forest model and even a logistic regression model, with their accuracy ranging from approximately 55% to 70%.

Our initial exploratory visualization allowed us to glean a few insights about the various categories of customers that are more likely to purchase a car. The Monte Carlo simulation generated the average probability of a customer from a given customer segment purchasing a car over 1,000 simulations. Time complexity comparisons have highlighted the fact that although the brute force algorithm will produce the best combination of parameters, the randomized search algorithm we built produced results nearly as good as the brute force algorithm, while taking substantially less time to obtain the solution. In addition, our project has shown that out of nearly 10 models, XGBoost proved to be our best.

Computational Setup

Code structure and computational steps

Create a 'Database' class to define our analytical functions and a 'Customer' class to get the attributes for each customer

Define the only function for the 'Customer' class:

a. genProb() - for each instance of a customer, generate a probability of a customer buying a car based on the categorical 'Rank' column in the dataset) - for each instance of a customer, generate a probability of a customer buying a car based on the categorical 'Rank' column in the dataset

Define the functions underneath the 'Database' class:

a. createDatabase() - this function looks at the file where the data comes from and creates + stores all the information in objects from the 'Customer' class
b. importFile() - this function that imports the file into the user's environment

c. total() - this function appends all instances of a given attribute into a list corresponding to that attribute
d. summary() - this function that gives statistical summaries for each attribute in the database
e. sorting() - this function compares the execution time between the quicksort and bubble sort algorithms for the dataset (*bubble sort & quicksort*)
    i. We test the scalability of these functions by dividing the dataset into 4 incremental buckets
    ii. By measuring the execution time and comparing them to each other, we can determine the most efficient function if our data doubles or triples
    iii. Plot a graph looking at the relationship between the length of the dataset and the time it takes for each algorithm to perform the sorting operation
f. to_frame() - this function converts the given dataset to a dataframe in order to allow us to perform the functions required for further analysis
g. plot_graphs() - this function performs exploratory data analysis to illustrate the characteristics of the dataset, allowing for further understanding (*matplotlib, seaborn, bokeh & HoloViews*)*HoloViews*)
    i. Identify attributes to be used for EDA, order the categorical variables, and select plots to visualize:
        - Interactive box and whisker plot to analyze the relationships between marital status, child, and house value
        - Bar graph to explore relationship between target variable and several categorical variables including: education level, customer's age interval, customer segment group and marital + child status
        - Use seaborn to map each variable while passing the target 'flags' variable into the hue parameter
    ii. Save each plot separately and export them to pdf
    iii. For location-based data visualization, build an interactive map of regions:
        - Categorize the states into a list separated into the dataset's given regions: Northeast, South, West, Midwest, and Rest
        - Group the dataset by region and compute median house price, percentage of people who purchased a car, percentage of house owners, and percentage of people with higher education (has either a 'Bach' or 'Grad') and store values in a separate dataframe
        - Import US longitude and latitude dataset from bokeh and allot categories to each state
        - Merge the two dataframes to build an interactive US map color-coded by region and with a hover tool that summarizes the details of each region in the dataset dataframes to build an interactive US map color-coded by region and with a hover tool that summarizes the details of each region in the dataset
h. Simulate() - this function was designed to run a simulation *n* number of times to determine the probability of a car purchase for each customer segment (*Monte Carlo simulations*)
    i. Initialize an empty dictionary to store the probability values of buying a car per customer segment – added at the end of each simulation
    ii. To compute the probability, randomly generate a probability between 0 and 1 for each segment and compare it with the probability defined in the customer class (assigned by the Rank column in the dataset)
    iii. If the randomly generated probability is less than that defined for that customer, then we increment the counter for car purchase in that segment
    iv. At the end of each iteration, compute the final probabilities by dividing the number of purchases per segment by the total number of people in it
    v. Return a dictionary with customer segment as the key and the final probability (averaged over all iterations) as the value
i. model_comparison() - build predictive models to investigate customer purchase decisions and compare them in order to find the best model (*XGBoost*)
    i. Import data as a dataframe through the pandas package

ii. Keep column names and syntax consistent for modeling formula by performing data cleaning to remove unnecessary symbols from column names, leaving only characters, numbers, and underscores
iii. Since some predictors are numeric and some are categorical, convert the latter into dummy variables to allow for modeling – done through the in-built get_dummies() function in pandas before merging the datasets together() function in pandas before merging the datasets together
iv. Utilizing different methodologies, and through various packages (sklearn, and XGBoost), to develop a diverse range of models for accuracy comparison
v. Split the raw data to train and test datasets with a 70/30 ratio, before training the models with the training set and fitting the models with the testing set
vi. Select best predictive model using AUC as our accuracy metric

j. greedy_vs_bruteforce() - this function was designed to use two different algorithms to test different combinations of the learning rate and number of estimator parameters in order to find the combination that would give us the highest prediction error (*brute force algorithm & greedy algorithm*)

Brute Force:

i. Write two for loops using np.linspace to generate all test values for learning rate and number of iterations (significant parameters)
ii. Create a model fit to each combination of parameters and predict values for the test dataset using AUC as our metric
iii. For each combination, store the metric in a list and keep track of the max AUC achieved and corresponding parameter values
iv. Plot a graph of the number of estimators vs AUC achieved for each learning rate value

Randomized Search:

i. Randomly select values for learning rate and number of estimators using np.random.choice from an existing list of values to form combination of parameters
ii. For each of these randomly generated combinations, create a model fit and predict corresponding values for the target variable
iii. Store max AUC for each combination into a list

Plot a graph of time taken for each algorithm vs AUC achieved to compare how fast each algorithm takes to reach a good AUC score

Computational Challenges:

Certain operations are more effectively performed if the data structure is a DataFrame. However, loading our data into the class structure made it difficult to implement these functions. In order to keep all our code within the object-oriented programming approach, we built a function to convert our imported data into a DataFrame to perform these operations. When running and compiling our code, it was evident that the slowest parts was the bubble sort function, and the process related to XGBoost hyperparameter tuning. In its current functionality, increasing the size of the data would impact the execution time of the bubble sort function. However, the hyperparameter tuning function is more significantly dependent on the number of parameters that need to be tuned, relative to the size of the dataset.

Results

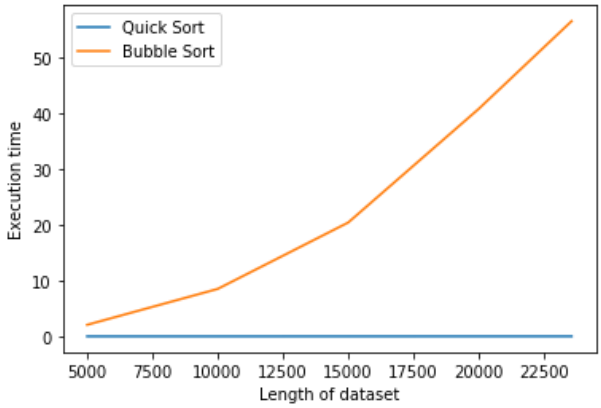
Exploratory Data Analysis

Performing EDA on the data, we found that:

- Married people had more expensive houses on an average.
- College educated customers are more likely to buy the car.
- People between the ages of 45 and 65 are more likely to buy the car.
- Customers falling in segments ‘B’ and ‘C’ are more likely to buy the car.

- Married people with a child have a significantly higher probability of buying the target product have a significantly higher probability of buying the target product

Comparison of Quick Sort and Bubble Sort



It is intuitive that Quick Sort performs better in case the data is scaled up.

Model Simulation

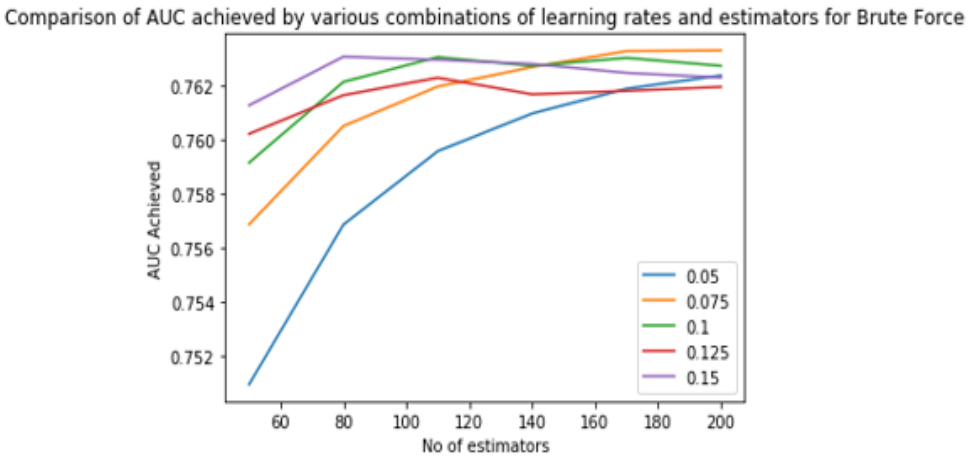
Using Monte Carlo simulation approach, we ran 1000 simulations to obtain the probability of car purchase for each customer segment.

{'A': 0.635493781094528, 'B': 0.8055607531055886, 'C': 0.7825024752475248, 'D': 0.7013902809415334, 'E': 0.6233454157782516, 'F': 0.536360895386021, 'G': 0.35255479744136453, 'H': 0.5845900783289817, 'I': 0.5689294926913161, 'J': 0.5575054121565349}

Predictive Model Comparison Summary

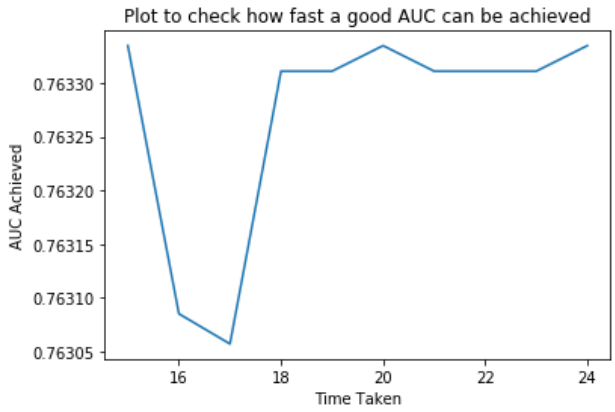
Model	*XGB Classifier*	*Gradient Boosting Classifier*	*Ada Boost Classifier*	*Linear Discriminant Analysis*	*Quadratic Discriminant Analysis*	*Random Forest Classifier*	*Gaussian NB*	*Decision Tree Classifier*	*K Neighbors Classifier*
Accuracy	69.85%	69.60%	68.85%	68.82%	66.21%	65.66%	60.94%	59.97%	58.66%
Log Loss	0.584128	0.584685	0.6882838	0.58987145	2.91396093	1.1716548	0.6994255	13.820008	4.4787639

Results of Brute Force algorithm for hyperparameter tuning:



- AUC achieved: 0.7633346543333083
- Number of Estimators: 200
- Learning Rate: 0.075
- Time taken to compute by Brute Search: 56.39084720611572 seconds.

Results of Randomized algorithm for hyperparameter tuning:



Conclusion

Computational Issues:

If the dataset were scaled up, there would be several issues that would impact the execution time of our function. These issues could be mitigated by the further implementation of parallel processing within the project. This would offset the increase in execution time and allow us to test more combination of parameters for model tuning and run our Monte Carlo simulation for a larger number of iterations.

Difficulty with Slicing and Dicing of Data using Primitive Data Structures:

Data manipulation or cleaning a dataset that's been loaded into primitive data structures like lists, dictionaries, etc., proved to be difficult as it was very hard to visualize the dataset in its entirety when it was stored as bits and pieces across multiple objects. The Pandas package, although computationally expensive when working with large datasets, is ideal for the slicing and dicing of data in our case, as we only had to work with 23,000 datapoints.