Abstract:

Regression and classification are both categorized under supervised learning of machine learning problems. Regression is used for problems with a continuous value for their label, and classification is used when there are discrete values as labels. Python, which is the best programming language when it comes to machine learning problems, has several packages and built-in tools for working with these algorithms. In this project, we write the code for a famous regression problem, called linear regression, and implement it on a dataset. We then use Python's existing tools for classification and implement those on another dataset.

Introduction:

This project consists of two parts. The first part is about implementing linear regression from scratch in Python and then using it to train a linear regression model on both a univariate and a multivariate dataset. In the univariate case, the scatter plot of the dataset, the actual line fitted on it, the surface plot of cost function, and the contour plot of cost function are also plotted. Both cases are then being evaluated on a test set.

The second part is about classification. Using Python libraries, we fit six different classification algorithms on a dataset to predict whether a customer will download an app that is being promoted in an advertisement or not. Then we test the models on test data and calculate the accuracy score and plot ROC and decision boundary for it.

Methods:

Part 1

Supervised learning - Regression

As said before, in this part, we are going to implement linear regression. In "LinearRegression" class, everything starts with "fit" method. In this method, the best value for theta, which is the matrix for weights and is the main thing being learned, is being filled by calling "gradientDescent" method. In "gradientDescent" method, the values of cost and theta are printed in each epoch. After that, I computed the derivative of the cost function and then updated the value of theta for that epoch based on the formulas shown below.

$$\frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{m}(x_j^{(i)})^T(\theta_j^T x_j^{(i)} - y^{(i)})$$

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

Gradient descent formulas

In the above equations, "J(θ)" is the cost function and "j" is the index for coefficients (based on how many features we have) and "m" is the total number of data points. Note that when multiplying matrices, I have used matrix multiplication (denoted with "@" or "np.matmul()"), and not the standard algebraic multiplication. "gradientDescent" method looks for the optimum value for theta, which minimizes the cost function and converges more and more as the epochs go on.

The cost function is computed in "computeCost" method. It is being calculated with a modification in the regular mean squared error function shown below. In this way, we eliminate the sigma and power of 2 by using matrices and vectorization.

$$J(\theta) = \frac{1}{2m}(\theta^T x^{(i)} - y^{(i)})^T(\theta^T x^{(i)} - y^{(i)})$$

Linear regression cost function formula

Lastly, I wrote the "predict" method that given matrix X, multiplies it by theta (which is the matrix for weights and now has its optimum values and is an attribute of the class's object) and then returns it.

$$h_\theta(x^{(i)}) = \theta^T x^{(i)} = \begin{bmatrix} \theta_0 & \theta_1 & \dots & \theta_j \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \dots \\ x_j \end{bmatrix}$$

Hypothesis function for linear regression

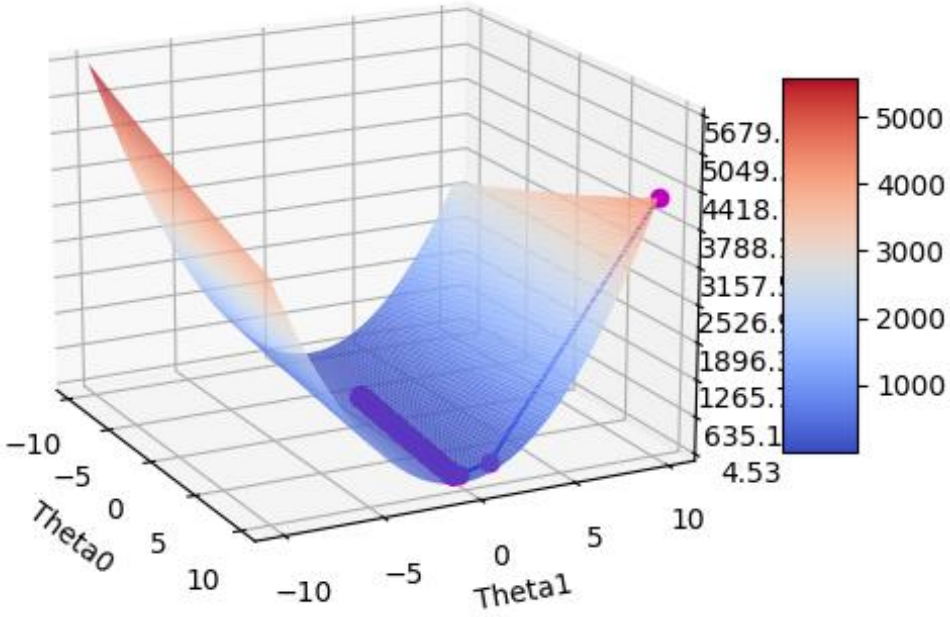
In the "test_linreg_univariate.py" file, "LinearRegression" class is being tested for a univariate problem, which is a dataset having only one feature. I fit the model on training data, and here are theta and loss value after 1500 epochs:

```
Iteration:  1500  Cost:  [[4.54458059]]  Theta:  [[-3.03401714]
 [ 1.10646013]]
```

Final values of cost function and theta for univariate linear regression

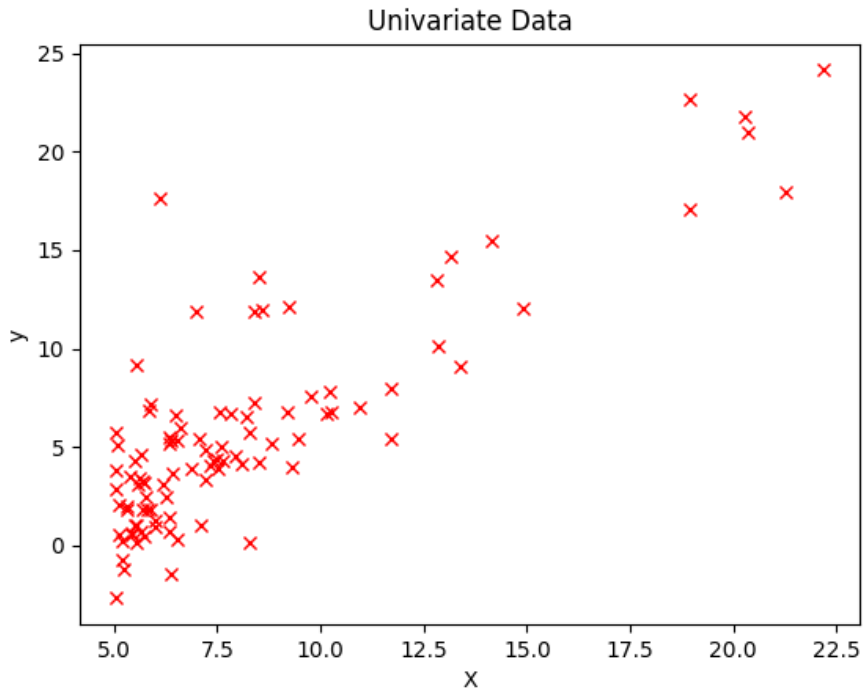
Cost value started somewhere around 4256.5 and converged at about 4.54 .

The figure below is plotted by "visualizeObjective" function and shows the algorithm trying to find the minimum cost value using gradient descent.

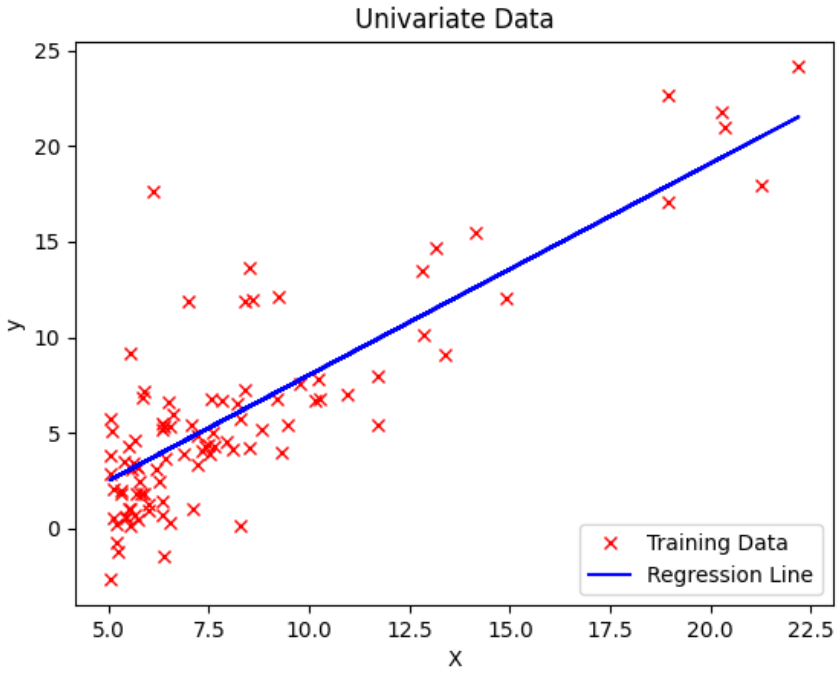


Cost function plot and gradient descent on it

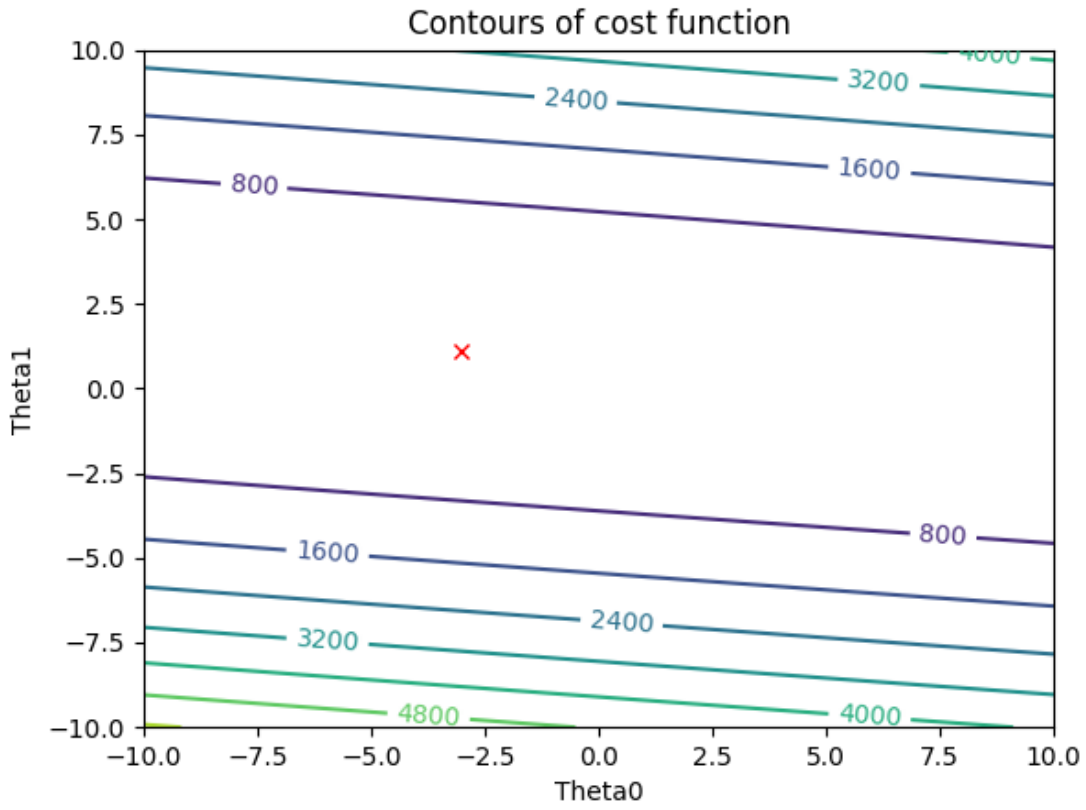
Here are the scatter plot of univariate data and the final line fitted on the training data plotted by "plotRegLine1D" function:



Distribution of univariate data



Plot of univariate linear regression



Contour plot of cost function of univariate data

Finally, I tested the model on unseen data "holdout.npz". After loading, reshaping, and appending an ones matrix to it, I passed it to "predict" method and predicted its labels. Then, I calculated the root mean squared error (RMSE) between true labels and predicted ones:

```
RMSE on test data:  310276.24126972223
```

RMSE on test data in univariate linear regression

Similarly, "test_linreg_multivariate.py" is testing "LinearRegression" class on a multivariate linear regression problem; a problem with multiple features. I fit the model on the training data set and here are theta and loss value after 2000 epochs:

```
Iteration:  2000  Cost:  [[2.04328008e+09]]  Theta:  [[340412.65893361]
 [109439.18788947]
 [ -6569.74627399]]
```

Final values of cost function and theta for multivariate linear regression

Then, I tested the model on "holdout.npz" data and calculated the RMSE for it:

```
RMSE on test data:  206944166.2930521
```

RMSE on test data in multivariate linear regression

Note: "holdout.npz" included a matrix of (20, 3) size. I assumed the last column as the label; I used the first two columns as features when testing multivariate linear regression, and I arbitrarily chose the first column as the feature when testing univariate linear regression.

Part 2

Supervised learning - Classification

Data acquisition and preprocessing:

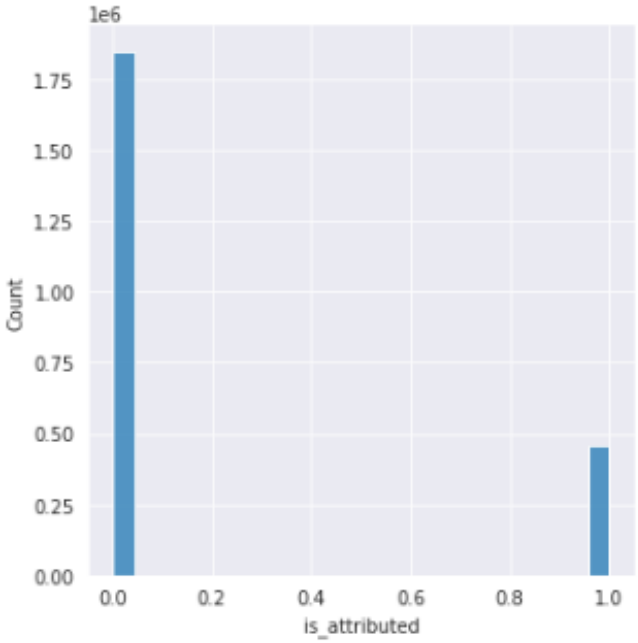
I wrote the code for this part in Google colab notebook, which is a convenient environment for data science projects. To begin with, I downloaded the lighter version of the ad-tracking fraud dataset[1]. The folder consists of a csv file, which is the main dataset, and some parquet files, which are some extra features made by the owner. However, I couldn't find a meaningful relationship between those features and the main dataset, so I didn't use the parquet files and only moved on with the csv file.

After reading the dataset with Pandas, I realized that "attributed_time" column doesn't provide any further information about predicting the target column, so I omitted it. Besides, I extracted the year, month, day, and hour from "click_time" column into separate new columns and deleted the column itself. Then, because year and month consisted of only one value, I dropped them too.

The dataset was really huge (about two million rows) which needed a high demand on resources, so I took a random sample of 8000 rows to work on.

By plotting distribution of the dataset, it was clear that the dataset is unbalanced, so I used SMOTE package of Python to undersample the majority class (class #0) and oversample the minority class (class #1).

[1] https://www.kaggle.com/matleonard/feature-engineering-data



Distribution of target column

I also used StandardScaler to bring all feature values under the same scale to avoid classifier bias. Then, I splitted the dataset into train, validation, and test set. I used validation set for finding best parameter values with GridSearchCV, train set to train the actual models, and test set to evaluate them.

Working with the models:

I used these six classification algorithms throughout this project:

1 - logistic regression

2 - support vector machine

3 - k nearest neighbors

4 - decision tree

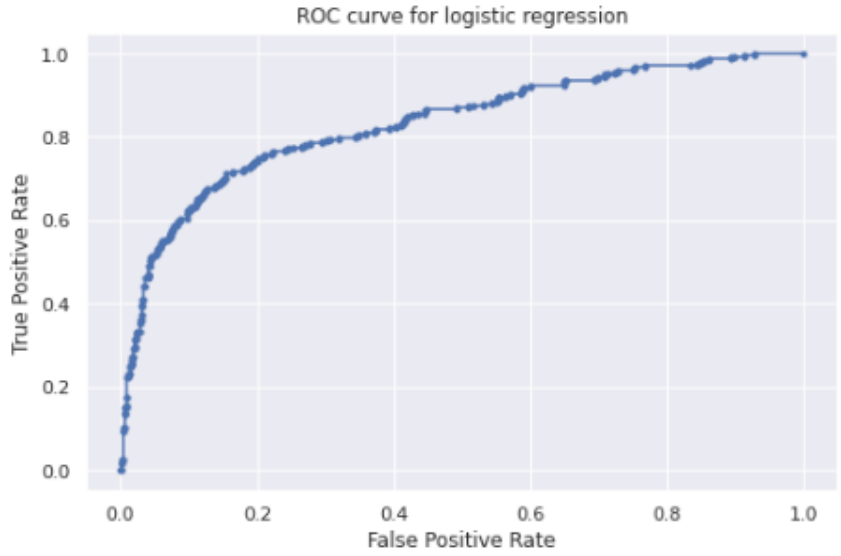
5 - random forest

6 - naïve bayes

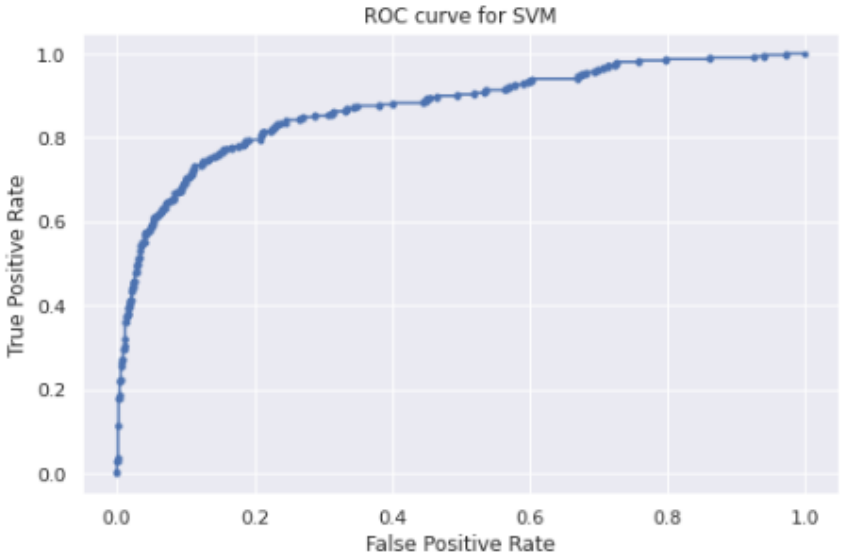
For all the above models, except for naïve bayes, I used cross validation to find the optimum values to some arbitrary parameters of each classifier. It is obvious that it's impossible to experiment with every value on every parameter of the model because that would be a heavy process.

Afterward, I trained each model with the parameters found in the previous step, on the train set. Then, I tested the model for both train and test sets and got the accuracy

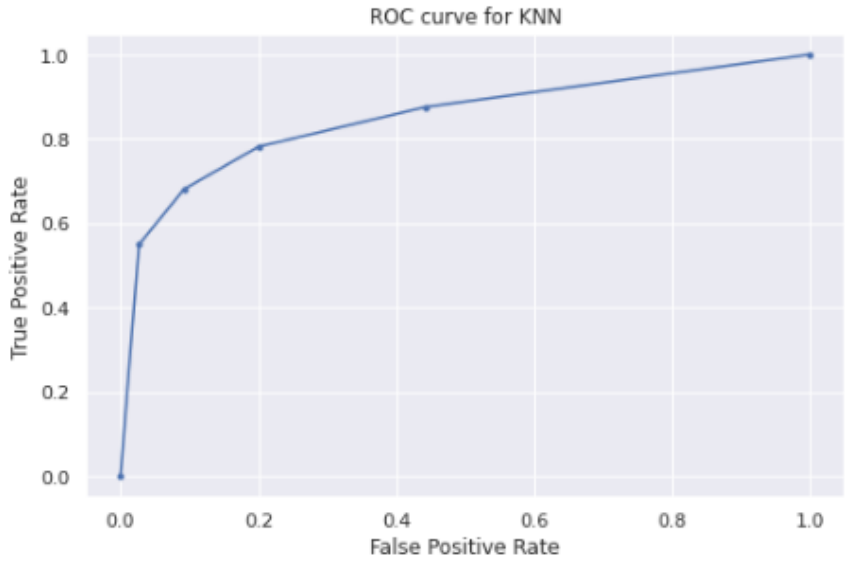
scores on both. Moreover, I plotted ROC and computed AUC score (area under ROC curve) for each model's prediction on test set.



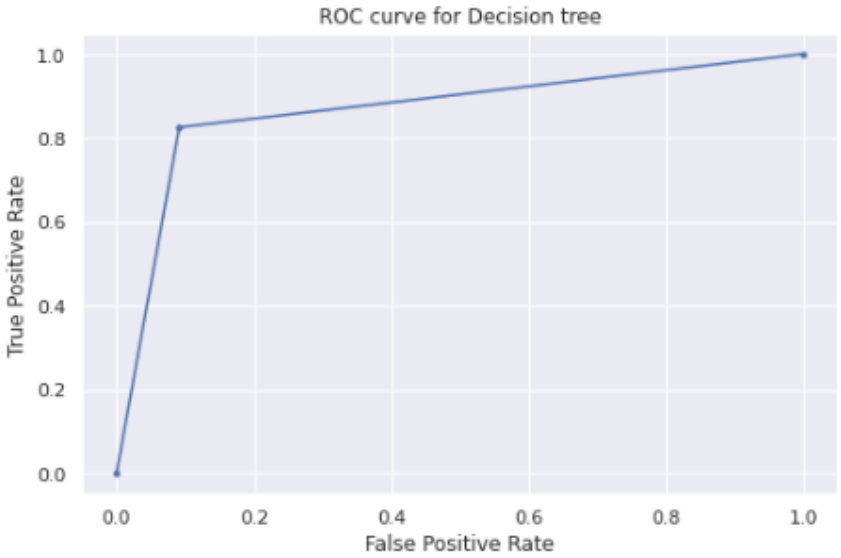
ROC for logistic regression



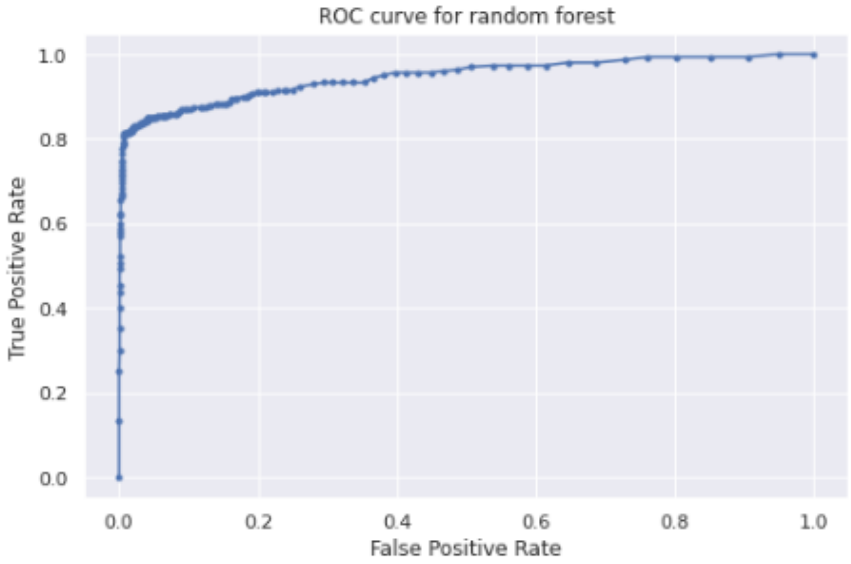
ROC for SVM



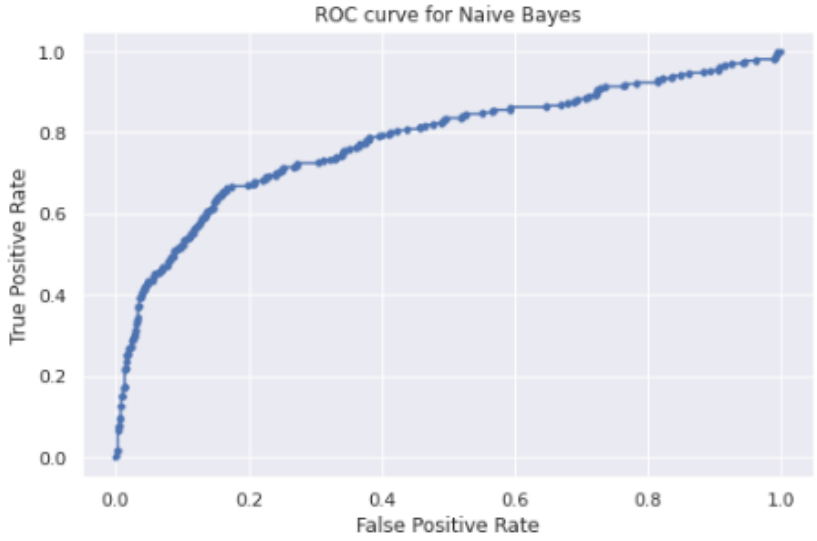
ROC for k nearest neighbors



ROC for decision tree

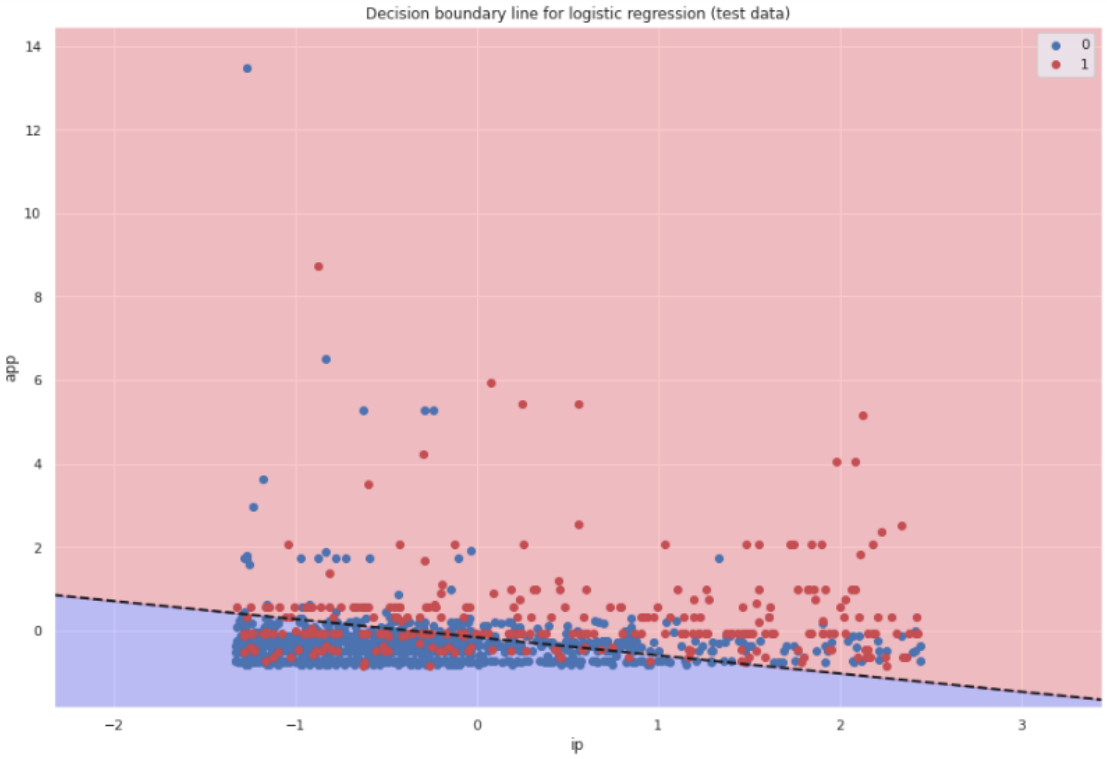


ROC for random forest

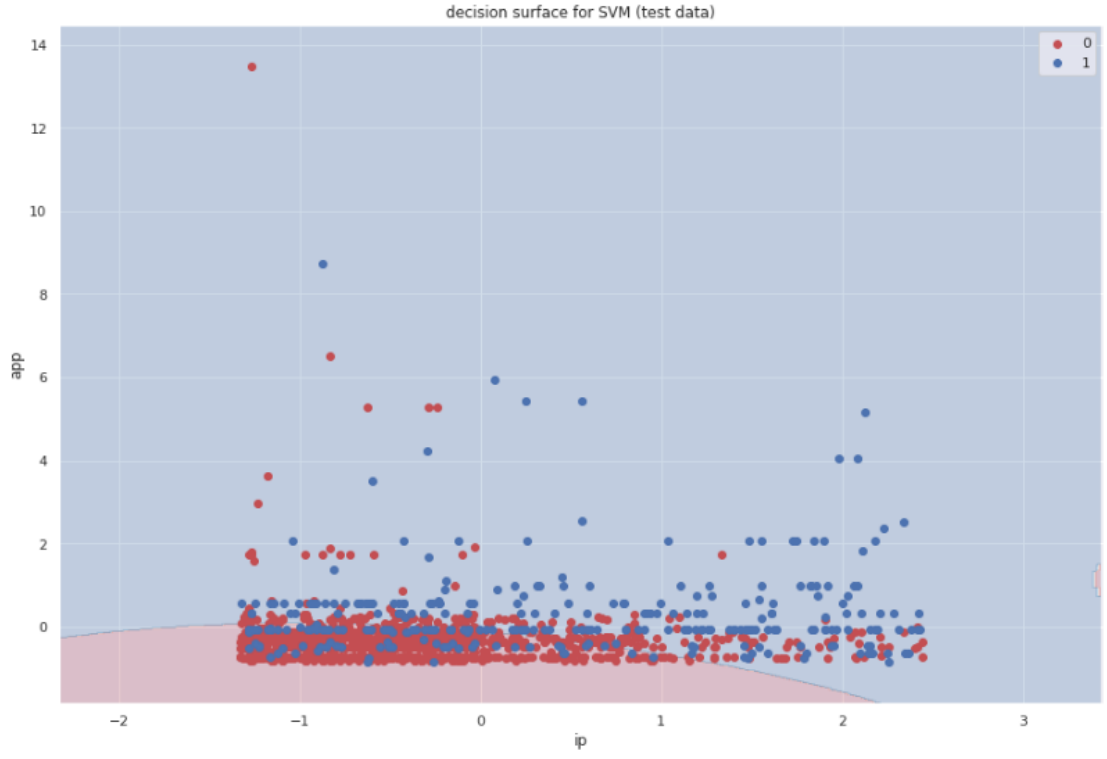


ROC for naïve bayes

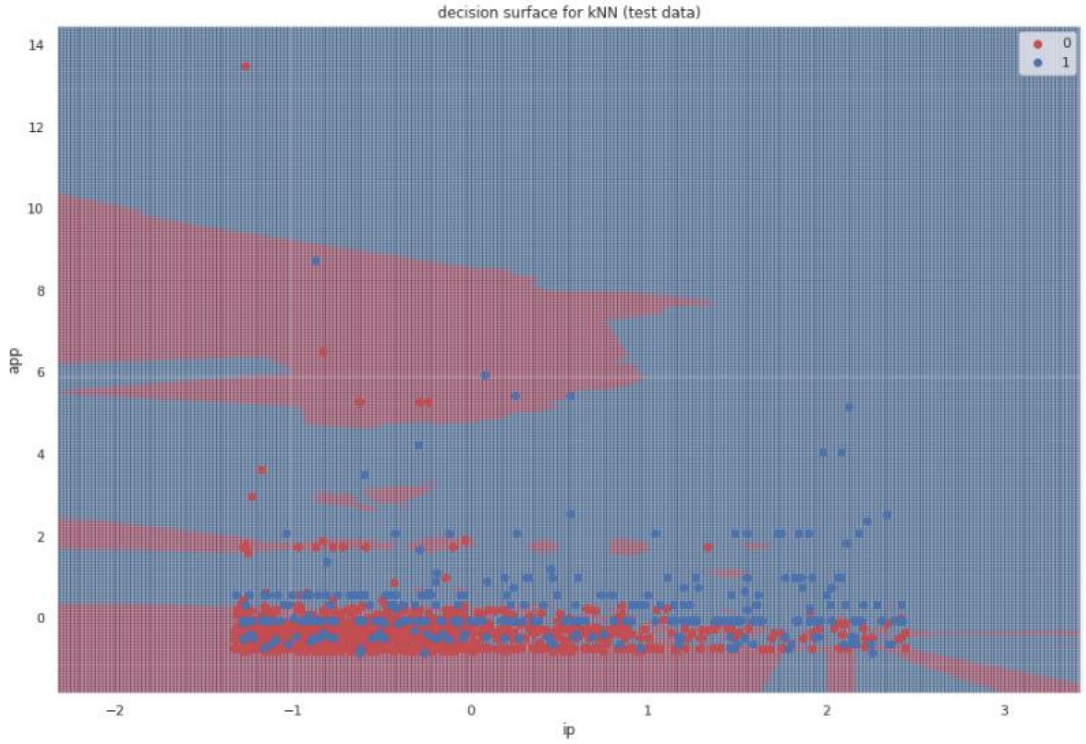
SelectKBest method of scikit learn module helped determine which two features have the most impact on the target column; "ip" and "app". So, I also trained each model on these two features in order to get a 2D visualization of the classifier's decision boundary, which separates the two classes for this problem.



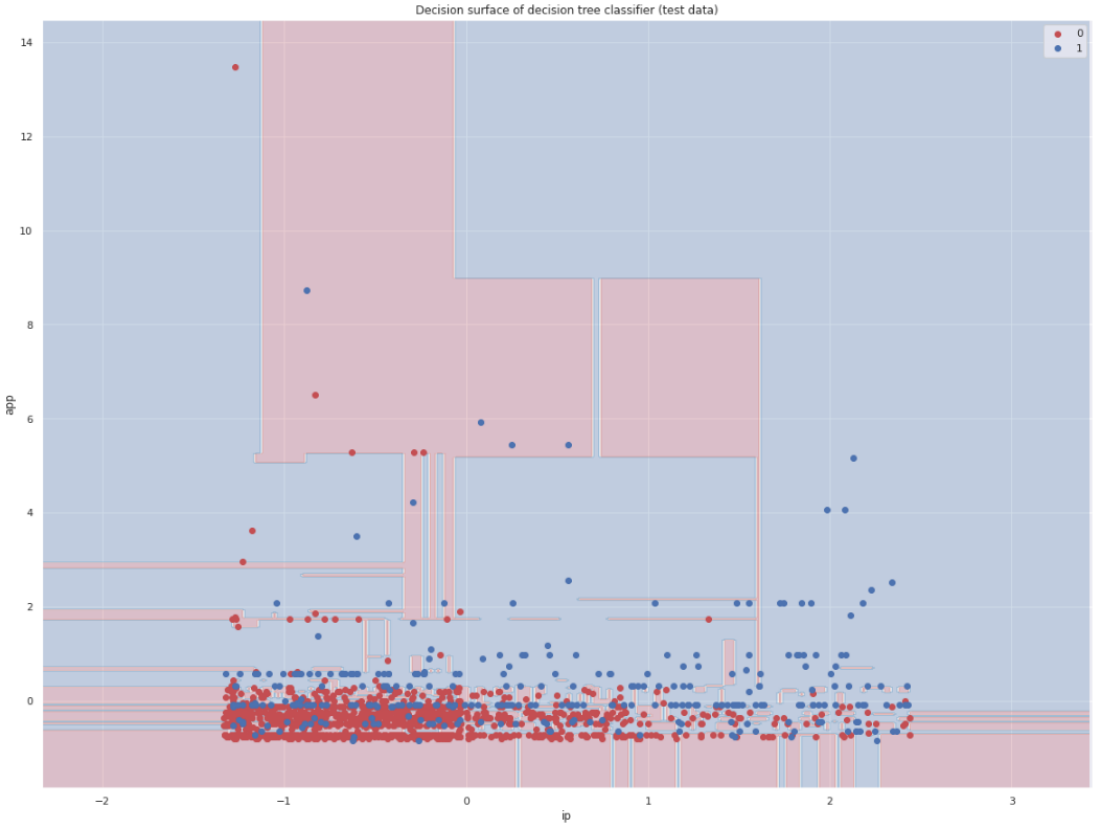
Decision boundary of logistic regression



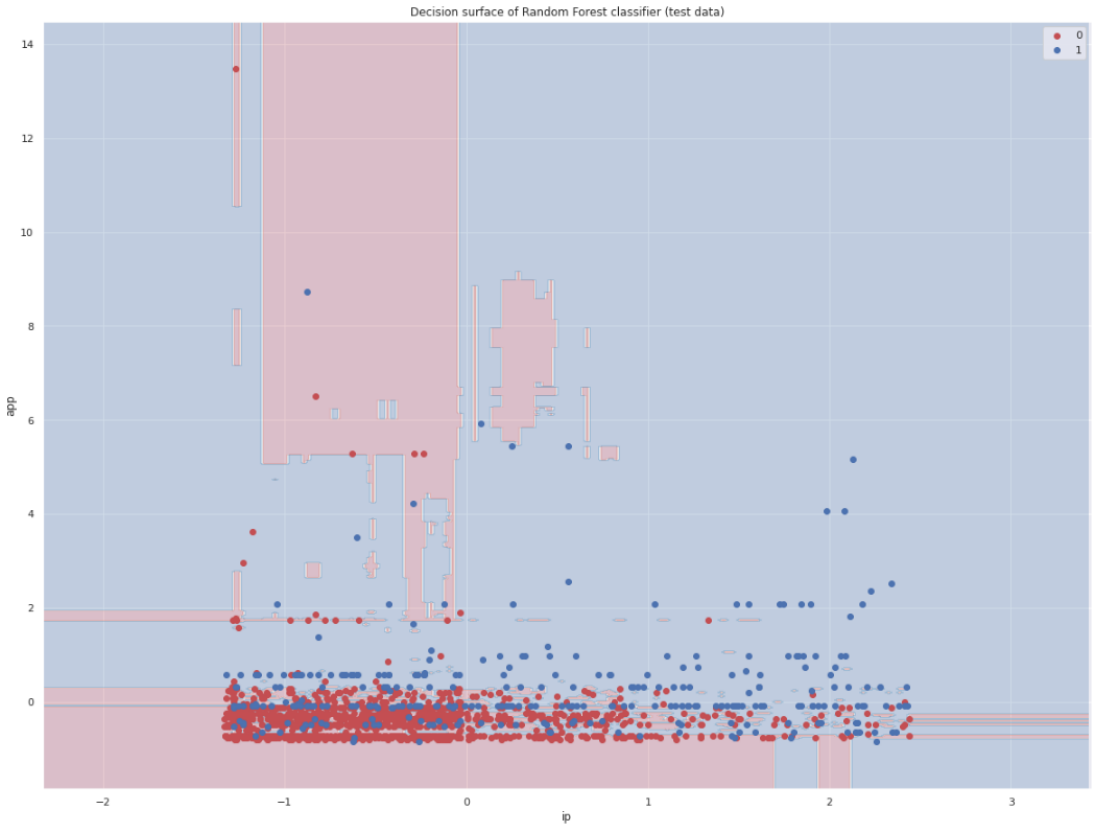
Decision boundary of SVM



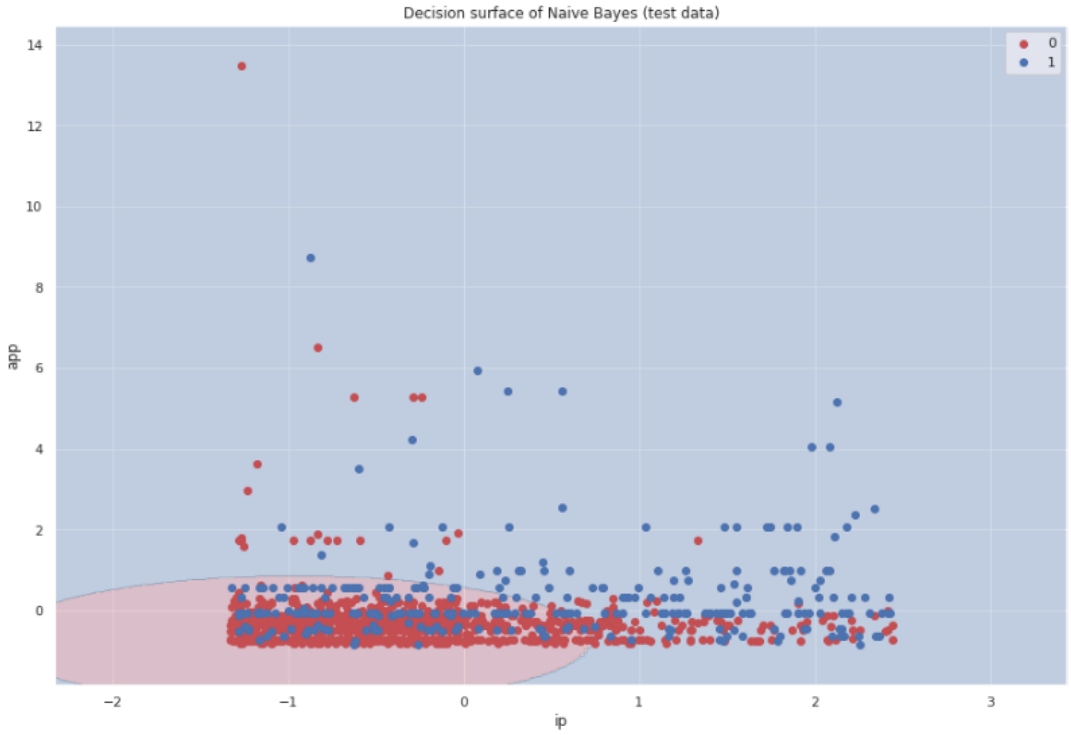
Decision boundary of kNN



Decision boundary of decision tree



Decision boundary of random forest



Decision boundary of naïve bayes

Conclusion:

Here are my findings out of this project:

1 - Out of all of the six models that I used for classification, only logistic regression showed a line as its decision boundary, which is why it's called a linear model. Decision boundaries for decision tree and random forest (which is an ensemble method based on decision trees) were crossing lines with sharp edges, like some overlapping rectangles. While for other methods, the decision boundary was smoother and looked like a curve.

2 - These are the accuracy scores that I got on train and test sets for each model:

	Logistic regression	SVM	kNN	Decision tree	Random forest	Naïve bayes
Train accuracy	0.8	0.85	0.91	1	1	0.75
Test accuracy	0.8	0.85	0.87	0.89	0.95	0.69

There might be overfitting in decision tree model, because it has a really high accuracy on train set (100%) and a pretty low one on test set (89%).

Random forest did the best job and had the highest accuracy among all, which can be because it's the only ensemble model here. Naïve bayes did a poor job and had the lowest accuracy among all, which can be because of its hypothesis of the dataset; this model assumes that all features are independent pairwise and that all features have the same impact on the target column, which is not true most of the times.

3 - Accuracy by itself is not adequate for evaluating a model. So, let's take a look at AUC scores:

	Logistic regression	SVM	kNN	Decision tree	Random forest	Naïve bayes
AUC	0.832	0.871	0.85	0.867	0.947	0.78

Once again, random forest got the best score, and naïve bayes got the worst.