I. Introduction

During this semester, I was introduced to using various machine learning techniques in both classification and regression problems. In this study, I applied this knowledge on two datasets. I applied classification techniques on the dataset regarding credit card transactions, training a model to identify fraudulent transactions with acceptable accuracy. In this experiment I used classification techniques including logistic regression, decision trees, and neural networks. I applied regression techniques on the dataset regarding energy efficiency in the heating and cooling of buildings. In this experiment I used regression techniques including linear regression, polynomial regression, and regularized regression.

I. Credit Card Fraud Dataset

A. *Data and Preprocessing*

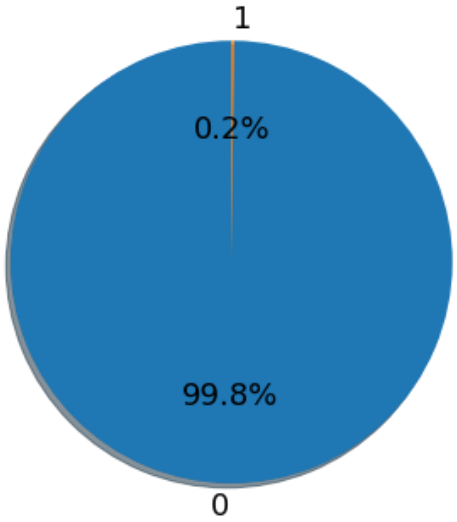
This dataset was a binary classification problem. I needed to train an algorithm to be able to classify a transaction as either non-fraudulent, or fraudulent. I set my X variable to everything in the dataset, except for the classification. I set my Y variable to the classification. A label of 0 classified the transaction as non-fraudulent, while a label of 1 classified the transaction as fraudulent. The classification was based on the behavior of the transaction including the timing and size/value. The immediate issue with this dataset was the imbalance. Non-fraudulent transactions were much more common than fraudulent transactions as expected. This is less than ideal when training a model, because the real-world accuracy of the model would be skewed because it is trained to detect non-fraudulent transactions much more than fraudulent ones. In preprocessing, I used train_test_split to separate my data in training and testing sets. In my testing set I identified the imbalance as shown here:

```
[0 1] [85297   146]
```

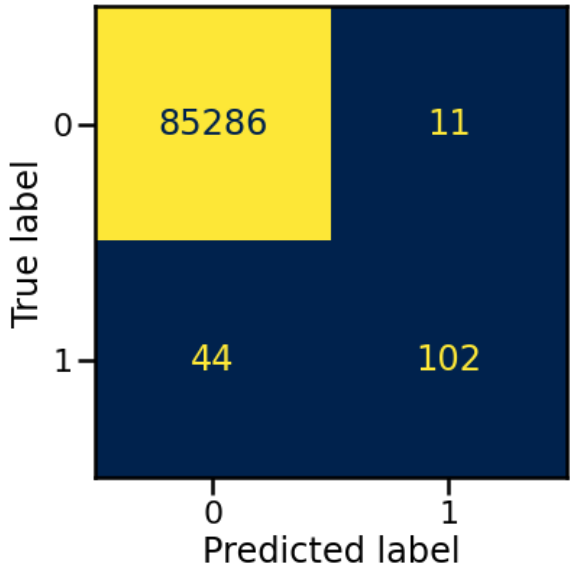
I then used StandardScaler to scale my X_train and X_test datasets. Doing this removes the mean and scales each feature/variable to unit variance.

B. *Logistic Regression Model Creation*

Once my training and testing sets were created and I scaled my X sets, I began to use Logistic Regression. Logistic Regression measures the relationship between the dependent and independent variable and uses a logistic function to estimate the probability that a certain piece of data belongs in a certain classification. I created a pie chart showing the distribution of the data to visualize the imbalance. I then created a Logistic Regression model using the imbalance data and displayed the confusion matrix as well as accuracy, precision, recall, and the classifier score.



The data is so imbalanced that 99.8% of the transactions were labeled as non-fraudulent and only 0.2% were labeled as fraudulent. This lets me know that the data I created by the Logistic Regression model will most likely cause accuracy issues. To confirm this I will show the confusion matrix below.



85286 True Negatives and 102 True Positives existed. This means that with minimal training the model successfully predicted 85286 non-fraudulent transactions and 102 fraudulent transactions. 11 False Positives and 44 False Negatives existed. This means that the model only predicted 11 non-fraudulent transactions incorrectly, but it missed 44 fraudulent transactions. This can be explained better with the performance metrics shown below:

```
Accuracy:  0.9993562960102056
Precision:  0.9026548672566371
Recall:  0.6986301369863014
Classifier Score:
0.9993562960102056
```

hAccuracy and Precision are high, but Recall suffers. In this dataset, Recall is the most important factor. Precision and Recall are defined by:

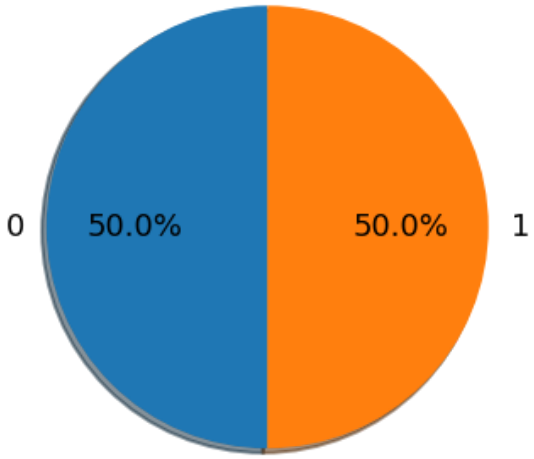
$$\text{Precision} = \frac{True\ Positive}{True\ Positive + False\ Positive}$$

$$\text{Recall} = \frac{True\ Positive}{True\ Positive + False\ Negative}$$

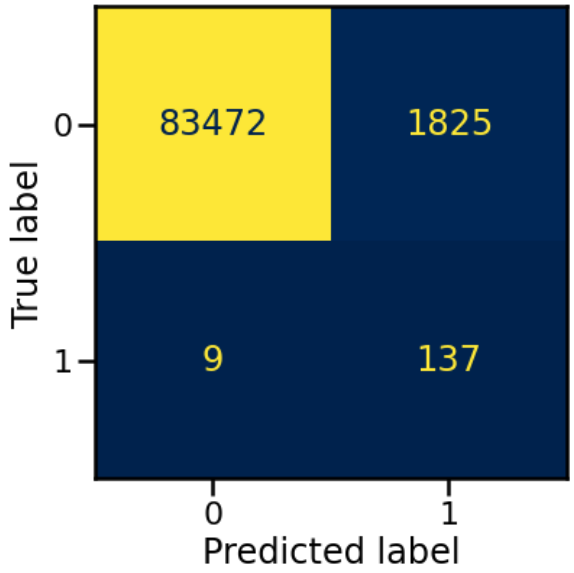
We want to focus on False Negatives, because the whole point of the model is to make sure it identifies as many fraudulent transactions as possible, so we want to maximize recall at the cost of precision. The accuracy is deceptively high. Even though we have 99.9% accuracy, the model missed over 30% of fraudulent transactions. At this point, the model is unusable in the real world.

C. Rebalancing the Data for Logistic Regression

It is clear that the data needs to be rebalanced to improve recall. I tried to rebalance the data using many methods. I used a class_weight="balanced", SMOTE resampling, RandomUnderSampling, and BaggingClassifiers, but all of these methods underperformed RandomOverSampling. I used RandomOverSampling to split the data more evenly, retrained a new LogisticRegression model, and created a pie chart, confusion matrix, and performance metrics of the new model as shown below:



Now, the data is completely evenly split between fraudulent and non-fraudulent transactions. This will allow the new model to be trained much more evenly. Let's look at the new confusion matrix below:



True Negatives decreased and False Positives increased. This isn't "good", but in a real world scenario credit card companies will report False Positives quite frequently because their models maximize recall, or, in other words, their models allow more false positives if it allows less fraudulent transactions to go unnoticed (False Negatives). As you can see, False Negatives decreased from 44 → 9 and True Positives increased from 102 → 137. This is very good. 35 less fraudulent transactions went unnoticed. Let's look at the performance metrics to see how this model performs:

```
Accuracy:  0.9785353978675843
Precision:  0.06982670744138635
Recall:  0.9383561643835616
Classifier Score:
0.9785353978675843
```

While accuracy, precision, and classifier score took a hit, recall improved from 69% → 94%. The ability of the model to detect fraudulent transactions increased by 25%! Lastly, I checked the classification_report to better understand the performance metrics specific to each classification:

	precision	recall	f1-score	support
0	1.00	0.98	0.99	85297
1	0.07	0.94	0.13	146
accuracy			0.98	85443
macro avg	0.53	0.96	0.56	85443
weighted avg	1.00	0.98	0.99	85443

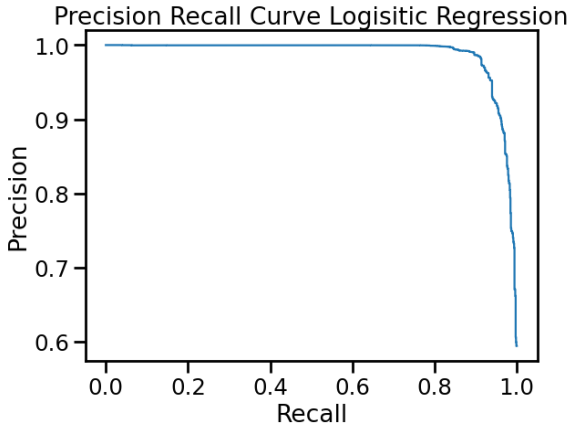
Recall was quite high for both classifications and while precision was very low for non-fraudulent cases, I was comfortable with this trade-off.

D. Training and Validation of the Logistic Regression Model

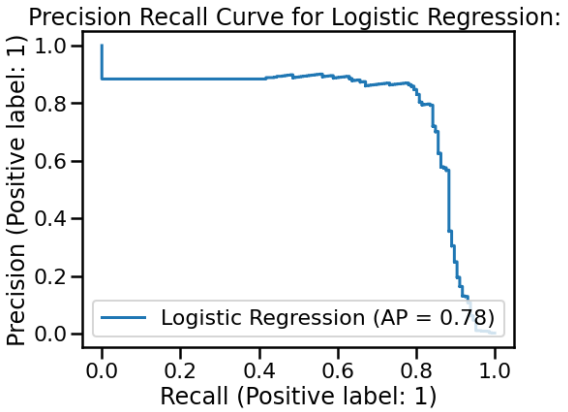
After identifying the better performing dataset balanced with RandomOverSampling I was confident enough to move to train my model with this data. I used cross_val_predict to make predictions using the rebalanced data and used cross_val_score to assess its performance shown here:

```
[0.94501767 0.94372131 0.94603476]
```

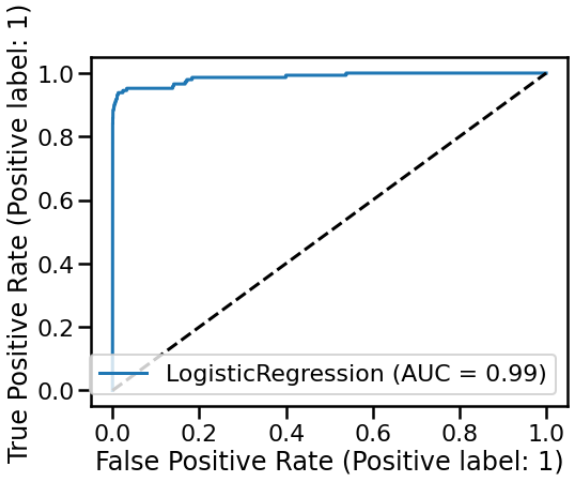
The performance was strong and while using the unbalanced dataset would have given me a higher score here, using the balanced dataset still returns acceptable performance while being more accurate in identifying fraudulent transactions. I then moved on to visualizing the relationship between precision and recall. I created a graph that plotted Precision against Recall showing that as Recall increased, Precision suffers:



I then plotted the actual Precision Recall curve using plot_precision_recall_curve:



Analyzing this graph, it looks like the level of recall that would still allow precision to be at an acceptable rate falls somewhere around 85-90%. Because my recall is around 94%, precision has already steeply declined leaving me with a Precision of about 7%. This is not ideal because of the high rate of false positives, but the model is extremely adept at identifying fraudulent transactions! Lastly, I checked the ROC curve and AUC score as shown below:




```
0.9584801678454381
```

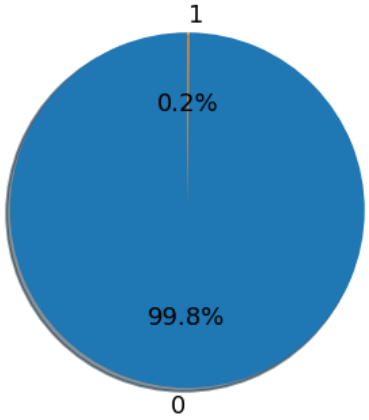
Because my ROC curve falls so highly in the top left corner, it shows strong performance despite the low precision. Because low precision was a purposeful trade off, I feel confident that this model would perform well in the real world.

E. Decision Trees Model Creation

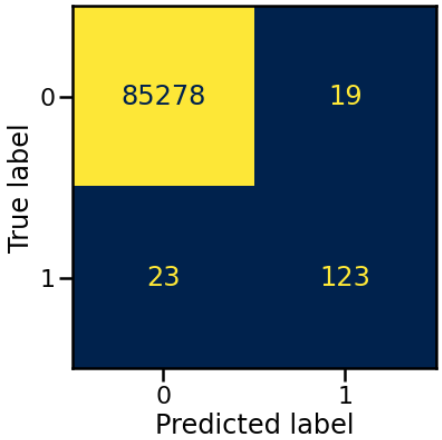
I follow the same process with Decision Trees. I will explain in a slightly expedited process how I was able to train the model with different methods.

F. Rebalancing the Data

Once again I create a pie chart and confusion matrix with the raw data before rebalancing. I did this because it is not guaranteed that Decision Trees will work as well with RandomOverSampling sas Logistic Regression did.

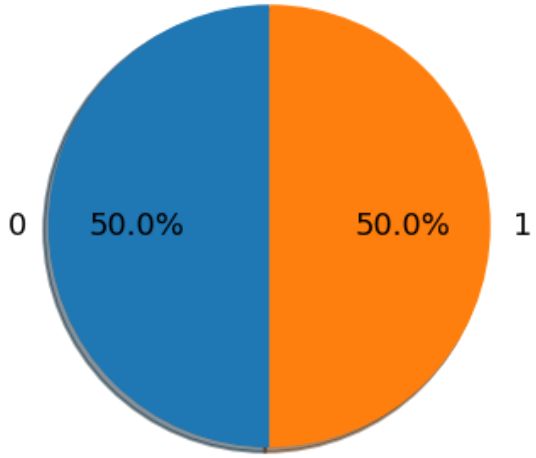


This chart shows that the data used to generate the confusion matrix and metrics below was in fact the unbalanced data.

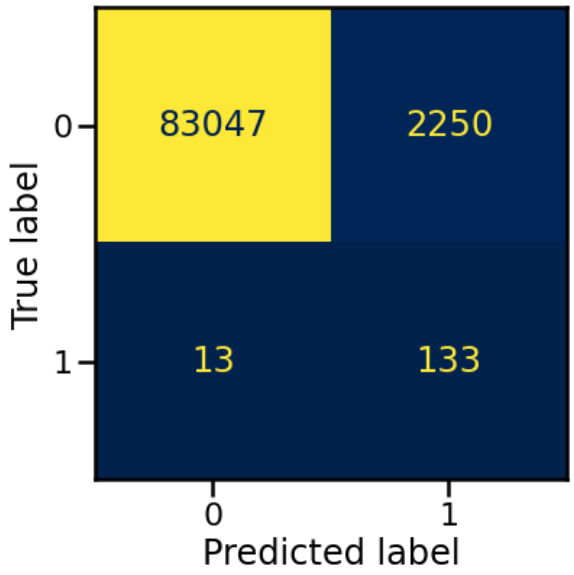



```
Accuracy:  0.9995084442259752
Precision:  0.8661971830985915
Recall:  0.8424657534246576
Classifier Score:
0.9995084442259752
```

As shown above, Recall was actually quite high for the DecisionTree model without data rebalancing. Precision was also quite high as well. This is good because the performance of the model is more balanced than with Logistic Regression and rebalancing. Despite this, I still feel as if the False Negatives are too high, I tried to train another DecisionTree model using the RanomOverSampler rebalanced data shown below:



The data has been balanced before the confusion matrix was generated.




```
Accuracy:  0.9735145067471882
Precision:  0.05581200167855644
Recall:  0.910958904109589
Classifier Score:
0.9735145067471882
```

Recall increased, but didn't quite get to the level that it did with LogisticRegression. The number of false positives were also much higher than with Logistic Regression. Despite this, I moved on to training with this rebalanced data because the 84% recall of the unbalanced dataset was not as high as I would have liked.

G. *Rebalancing the Data for Decision Trees*

To make sure that I wasn't mistaken, I trained the model with both the balanced and unbalanced data and printed performance metrics as shown below:

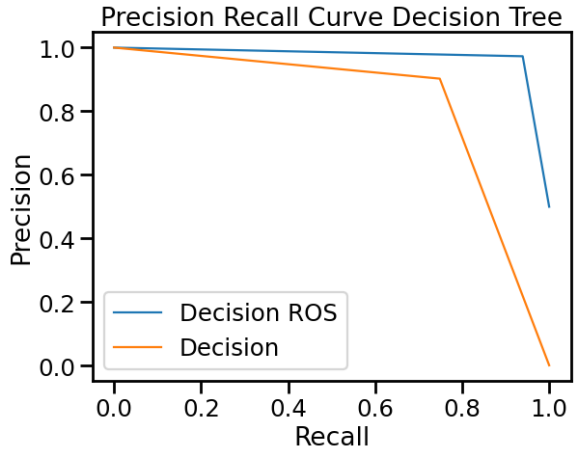
```
With ROS:
Accuracy:  0.9583605503019827
Precision:  0.9738411352822622
Recall:  0.9420253444412063
Classifier Score:
0.9735145067471882
Without ROS:
Accuracy:  0.9993730061595875
Precision:  0.8850174216027874
Recall:  0.7341040462427746
Classifier Score:
0.9995084442259752
```

nIt seems that I made the correct decision training on the balanced dataset. The precision and recall for the model trained on the unbalanced dataset were lower than those of the model trained on the balanced dataset. Because of this, I felt much more comfortable continuing with the model trained on the balanced data despite the confusion matrices showing some ambiguity at first glance.

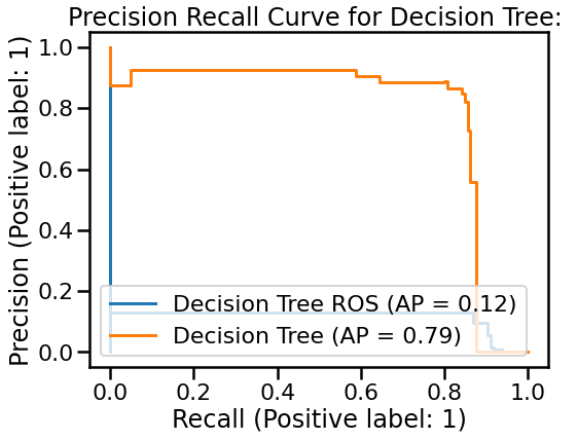
H. *Training and Validation of the Decision Tree Model*

I used cross_val_predict once again to assist in getting values for precision and recall that I may plot on a graph. I compared the

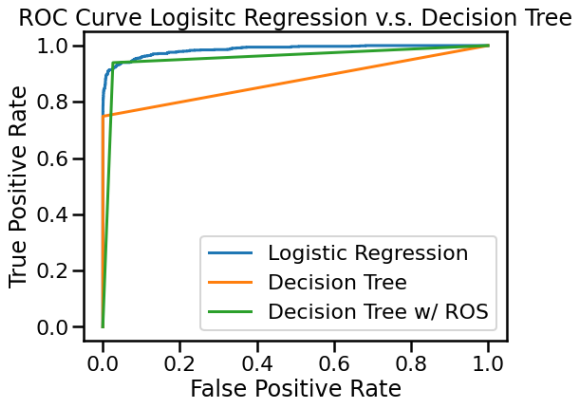
precision and recall curve for my Decision Tree trained with and without balanced data on the same graph below:



As shown, the model trained on balanced data performed much better, once again proving that the balancing of the data before training was a very positive step in the process. I wanted to emphasize the effect of rebalancing the data since it was integral to the experiment. I then used plot_precision_recall to show once again the differences between the two models:



This result was interesting to me. The model that used unbalanced data had an expected precision recall curve, but the model using the balanced data had precision fall to a very low level immediately, falling even more once the recall passed 80%. This was odd to me, but since I was maximizing recall in this model, I did not think that it made the model unusable. It was interesting to see that the decision tree model performed very well without balanced data if the users of the model were willing to have slightly more false negatives for the trade off that the number of false positives would fall drastically. Because of this, I see the two models as each being valid, but in different scenarios. I also plotted multiple ROC curves to better visualize this. Below is the ROC curve of the Decision Tree using the unbalanced dataset and using the balanced dataset compared with the ROC curve of the logistic regression model as well:



This shows that the logistic regression model performs the best, but the decision tree using the balanced dataset was close by.

I. Neural Network Model Creation

For the Neural Network I created two separate models and fit them to the training sets. The first model shown below used unbalanced data:

```
loss: 0.0058 - accuracy: 0.9992

loss: 0.0034 - accuracy: 0.9994

loss: 0.0031 - accuracy: 0.9994

loss: 0.0030 - accuracy: 0.9994

loss: 0.0026 - accuracy: 0.9994
```

The second model shown on the below used balanced data and performed marginally better:

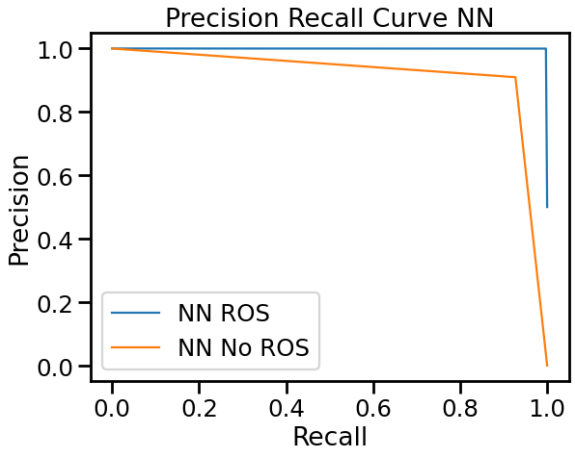
```
loss: 0.0058 - accuracy: 0.9992

loss: 0.0034 - accuracy: 0.9994

loss: 0.0031 - accuracy: 0.9994

loss: 0.0030 - accuracy: 0.9994

loss: 0.0026 - accuracy: 0.9994
```

J. *Training and Validation of the Neural Network Model*

I then predicted the classification of the test set using both models and the results of both the confusion matrices and performance metrics are shown below:

```
CNF w/o ROS:
[[85254    24]
 [   43   122]]
Accuracy:  0.9992158515033414
Precision:  0.7393939393939394
Recall:  0.8356164383561644
CNF w/ ROS:
[[ 5206     4]
 [80091   142]]
Accuracy:  0.06259143522582307
Precision:  0.0017698453254895118
Recall:  0.9726027397260274
```

Without rebalancing data the model had strong accuracy and recall, with decent precision. The balanced model struggled in all aspects, yet had very high recall. Looking at the confusion matrix for the rebalanced data, something looks very odd. It seems as if the values are in the wrong spots. I decided to continue on using both models. I plotted the precision and recall graph for the two version of the model as shown below:



It seems that the model with data rebalancing actually worked very well.

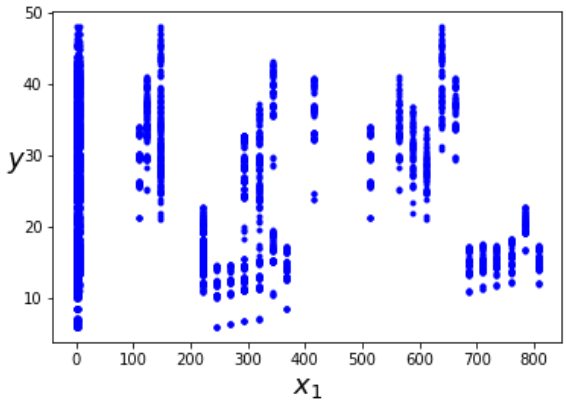
K. *Comparison of Models*

I believe that the Logistic Regression model using rebalanced data from the RandomOverSampler was the best model for detecting credit card fraud. This is because of the recall score in relation to the number of false positives. The Logistic Regression model; had the lowest number of false negatives and the lowest number of false positives out of the models with acceptable recall.

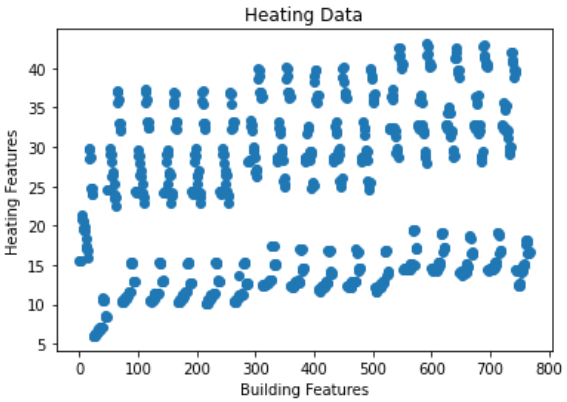
II. Energy Efficiency Dataset

A. *Data and Preprocessing*

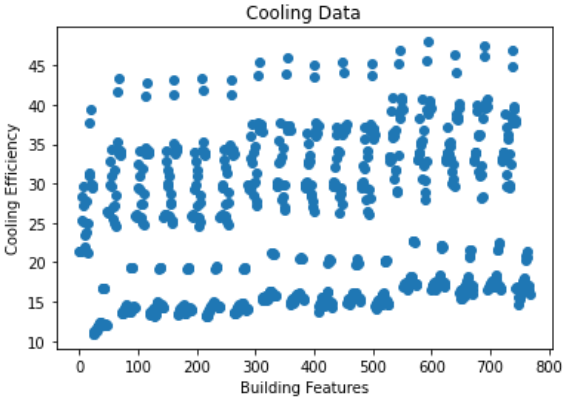
In this dataset I was given the features of a building and the energy efficiency of cooling and heating said building. For preprocessing I checked null values and removed them. I then renamed the data columns to something more readable. I used data.describe() to see the dataset and created my X and y values. I set my y variable to include both the heating and cooling efficiency. I also created a y variable for just the cooling and heating data to help with a scatter plot for just those isolated values. Below is the plot of X vs y. This graph shows the distribution of X values (building features) across the possible y values (energy efficiency):



Because the y value had two columns in it, I wanted to visualize the differences between those columns. Below is the relationship between the building features and heating:



Below is the relationship between the building features and cooling:



I executed a train_test_split on X and y (both columns) for my data analysis.

B. Linear Regression Training and Validation

Linear Regression models try to find a linear relationship between the X and y variables. I scaled the X_train and X_test data and created a linear regression model and a variable for expected and predicted values. After creating the model I fit X_train and y_train to the model and calculated the score. I printed this value to assess the performance of the model:

```
R2: 0.9026379224548287
```

This score was before I predicted any y values with the model, so we will use this as more of a baseline. I then predicted y_pred with the linear regression model and calculated the R2 score shown below:

```
r2_score 0.8912863062954878
```

Once I found the R2 score, I validated the data with cross_val_predict and got the score with cross_val_score as shown below:

```
[0.90124043 0.90430618 0.89212117]
```

Lastly, I found the MSE shown below:

```
10.327209706449818
```

C. Polynomial Regression Training and Validation

Next, I tried to use polynomial regression since a polynomial equation may be better at predicting values in this scenario. I created a polynomial regression model and fit it on X_test and X_train to create new training and testing vars. I found y_pred using predict, but found that the R2 value was negative when poly = 8, and simply got worse as the dimensions increased/decreased. This showed that this dataset did not work well with more dimensions:

```
r2_score -12.03610071473415
```

Then I found the MSE which was equally as poor:

```
7.229164146938164e+20
```

I continued on and used cross_val_predict and cross_val_score and it returned these values:

```
[0.90124043 0.90430618 0.89212117]
```

Why did this model perform so poorly? I referred back to the split of data points and realized that they most certainly did not follow a polynomial line, so this makes sense.

D. Neural Network Training and Validation

Because the data points were not perfectly linear or even close to following a polynomial line, I decided to try a form of regression that was not reliant on one of these conditions to be met. I created a NN with one hidden layer. I set the input layer shape to 200 and the output shape to 2 because the y value had 2 columns. I ran through 300 epochs of a batch size of 10. I calculated the R2 score below:

```
r2_score 0.8901006652306644
```

This was much better than polynomial regression and slightly worse than linear regression.

E. Comparison

The linear regression model seemed to perform the best while trying to find a correlation between building features and energy efficiency. My NN performed well on this dataset, but not as well as linear regression, this may be because of the shape of my input layers, number/size of epochs, or any other adjustable value that would affect my predictions. Polynomial regression performed terribly. This may have been user error, but I believe that it was because of the shape of the data lacking any curve at all. Personally, if I was to try another method, I would use decision trees. The way that the data appears in the plot suggests that it may be divided in a semi-categorical way, but with numbers/values rather than named categories. I do not have experience with using decision trees like this, so I used the suggested methods, but it would be interesting to see how it would perform.

III. Future Works

If I was able to spend more time on this study, I would certainly re approach the regression of energy efficiency and attempt to use tree based algorithms. I feel confident that my classification analysis was strong, but I found room for improvement balancing my recall score before precision fell sharply. I very much enjoyed exploring the classification dataset and the methods for rebalancing.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
from imblearn.over_sampling import RandomOverSampler
from imblearn.over_sampling import SMOTE
from imblearn.under_sampling import RandomUnderSampler
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import plot_confusion_matrix
from sklearn.ensemble import BaggingClassifier
import seaborn as sns
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score
from sklearn.metrics import f1_score
from sklearn.model_selection import cross_val_predict
from sklearn.model_selection import cross_val_score
from sklearn.metrics import precision_recall_curve
from sklearn.metrics import plot_precision_recall_curve
from sklearn.metrics import roc_curve
from sklearn.metrics import roc_auc_score
from sklearn.tree import DecisionTreeClassifier
import tensorflow as tf
from sklearn.compose import ColumnTransformer
from tensorflow import keras
from keras.models import Sequential
from keras.layers import Dense
from sklearn.preprocessing import LabelEncoder, OneHotEncoder
from keras.wrappers.scikit_learn import KerasClassifier
from sklearn.metrics import classification_report
from matplotlib.pyplot import figure
from sklearn import metrics
```

```
# import the dataset
from google.colab import drive
drive.mount('/content/drive')
dataset = pd.read_csv("/content/drive/My Drive/School/Machine Learning/Final/creditcard.cs
print(dataset)
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mour
            Time         V1         V2  ...       V28  Amount  Class
0            0.0  -1.359807  -0.072781  ... -0.021053  149.62      0
1            0.0   1.191857   0.266151  ...  0.014724    2.69      0
2            1.0  -1.358354  -1.340163  ... -0.059752  378.66      0
3            1.0  -0.966272  -0.185226  ...  0.061458  123.50      0
4            2.0  -1.158233   0.877737  ...  0.215153   69.99      0
...          ...        ...        ...  ...       ...     ...    ...
284802  172786.0 -11.881118  10.071785  ...  0.823731    0.77      0
284803  172787.0  -0.732789  -0.055080  ... -0.053527   24.79      0
```

```
284804  172788.0   1.919565  -0.301254  ... -0.026561   67.88      0
284805  172788.0  -0.240440   0.530483  ...  0.104533   10.00      0
284806  172792.0  -0.533413  -0.189733  ...  0.013649  217.00      0

[284807 rows x 31 columns]
```

```
# install imbalanced-learn
!pip install imbalanced-learn
```

```
Requirement already satisfied: imbalanced-learn in /usr/local/lib/python3.7/dist-package
Requirement already satisfied: scikit-learn>=0.24 in /usr/local/lib/python3.7/dist-packa
Requirement already satisfied: scipy>=0.19.1 in /usr/local/lib/python3.7/dist-packages (
Requirement already satisfied: joblib>=0.11 in /usr/local/lib/python3.7/dist-packages (f
Requirement already satisfied: numpy>=1.13.3 in /usr/local/lib/python3.7/dist-packages (
Requirement already satisfied: threadpoolctl>=2.0.0 in /usr/local/lib/python3.7/dist-pac
```

```
# create X and y
X = dataset.drop(columns = ['Class'])
y = dataset['Class']
```

```
# train test split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3, random_state =
print(X_train.shape)
print(X_test.shape)
```

```
(199364, 30)
(85443, 30)
```

```
# show unique elements
# this shows the class imbalance
unique_elements, counts_elements = np.unique(y_test, return_counts=True)
print(unique_elements, counts_elements)
```

```
[0 1] [85297   146]
```

```
# scale the data
scaler = StandardScaler()

X_train = scaler.fit_transform(X_train)
X_test = scaler.fit_transform(X_test)
```

For classification I will use 3 of the following algorithms:

> logistic regression, support vector machines, decision trees, random forest, and neural network models

For classification I will use the following metrics:

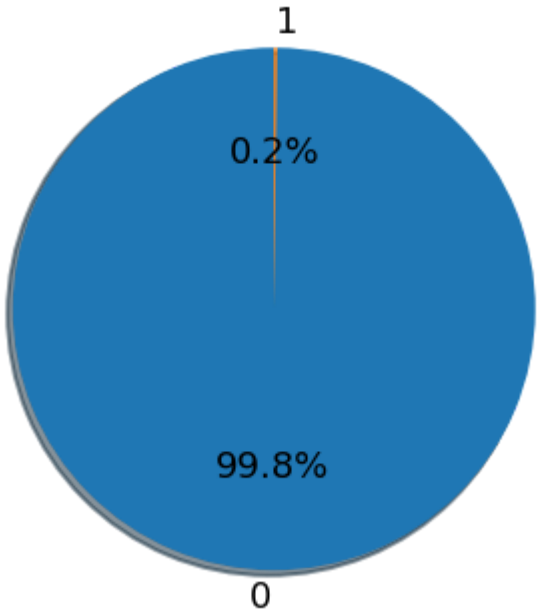
> precision, recall, and ROC curves

First I will use Logisitc Regression:

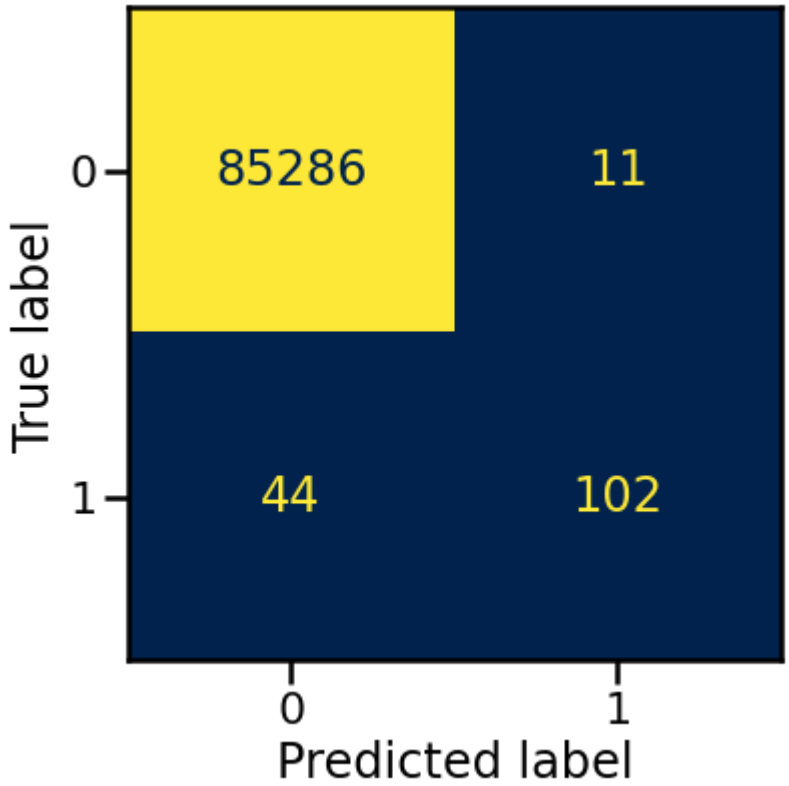
I am going to attempt to rebalance the data and visualize the effect it has on important metrics like precision, recall, etc before I choose the best method of rebalancing.

> It is important that I show the rebalance, but this will be the same for all of my rebalancing algorithms. I will also show the confusion matrix and some important metrics after creating a new logistic regression algorithm so I may compare their performance.

```
#visualize the imbalance of the data in the base set
#the data is incredibly imbalanced towards non-fraudulent transactions
plt.rcParams["figure.figsize"] = (6,6)
unique_elements, counts_elements = np.unique(y, return_counts=True)
fig1, ax1 = plt.subplots()

ax1.pie(counts_elements, labels=unique_elements, autopct='%1.1f%%',
        shadow=True, startangle=90, textprops={'fontsize': 18})

plt.show()

# default logistic regression
log_reg = LogisticRegression(max_iter=1000).fit(X_train, y_train)
y_pred = log_reg.predict(X_test)

#display the confusion matrix
sns.set_context("poster")
disp = plot_confusion_matrix(log_reg, X_test, y_test,
                     cmap = 'cividis', colorbar=False)

print("Accuracy: ", accuracy_score(y_test, y_pred))
print("Precision: ", precision_score(y_test, y_pred))
print("Recall: ", recall_score(y_test, y_pred))
print("Classifier Score: ", log_reg.score(X_test, y_test))
```




```
/usr/local/lib/python3.7/dist-packages/sklearn/utils/deprecation.py:87: FutureWarn
  warnings.warn(msg, category=FutureWarning)
Accuracy:  0.9993562960102056
Precision:  0.9026548672566371
Recall:  0.6986301369863014
Classifier Score:  0.9993562960102056
```



> In the above model, accuracy and precision are strong, but recall is low. Recall is the most important metric here, because it shows times when the algorithm missed a

> fraudulent transaction. Precision simply shows how rare false positives are, but in real life examples, false positives are quite often. This is because they are negligible compared to a missed fraudulent transaction. We will optimize recall because of this.

> Here I changed the class_weight to "balanced". This makes the precision much worse (more false positives), but actual fraudulent transactions are caught at a much higher rate. This is very important, so the algorithm has already improved.

```
# log_reg_b = LogisticRegression(max_iter = 1000, class_weight="balanced").fit(X_train, y_
# y_pred_b = log_reg_b.predict(X_test)

# sns.set_context("poster")

# disp = plot_confusion_matrix(log_reg_b, X_test, y_test,
#                      cmap = 'cividis', colorbar=False)

# print("Accuracy: ", accuracy_score(y_test, y_pred_b))
# print("Precision: ", precision_score(y_test, y_pred_b))
# print("Recall: ", recall_score(y_test, y_pred_b))
# print("Classifier Score: ", log_reg.score(X_test, y_test))
```

> Now I'll try to actually balance the data. I'll start with a random over sampler. This will over sample the minority class. Since we know that using class_weight="balanced" provides better results I'll use that while creating my new algorithm.

```
ros = RandomOverSampler(random_state=0)
X_train_ros, y_train_ros = ros.fit_resample(X_train, y_train)

plt.rcParams["figure.figsize"] = (6,6)
unique_elements, counts_elements = np.unique(y_train_ros, return_counts=True)
fig1, ax1 = plt.subplots()

ax1.pie(counts_elements, labels=unique_elements, autopct='%1.1f%%',
        shadow=True, startangle=90, textprops={'fontsize': 18})

plt.show()

log_reg_ros = LogisticRegression(max_iter = 1000, class_weight="balanced").fit(X_train_ros
y_pred_ros = log_reg_ros.predict(X_test)

sns.set_context("poster")

disp = plot_confusion_matrix(log_reg_ros, X_test, y_test,
                     cmap = 'cividis', colorbar=False)
```

```
20
21 print("Accuracy: ", accuracy_score(y_test, y_pred_ros))
22 print("Precision: ", precision_score(y_test, y_pred_ros))
23 print("Recall: ", recall_score(y_test, y_pred_ros))
24 print("Classifier Score: ", log_reg_ros.score(X_test, y_test))
```

> Though our metric scores have overall lowered, our algorithm is actually performing better than the other iterations. We have successfully lowered our false positives by a small amount.

> Now I'll try another method. I'll use smote to resample the dataset.

```
# smote = SMOTE()
# X_train_smote, y_train_smote = smote.fit_resample(X_train, y_train)

# plt.rcParams["figure.figsize"] = (6,6)
# unique_elements, counts_elements = np.unique(y_train_smote, return_counts=True)
# fig1, ax1 = plt.subplots()

# ax1.pie(counts_elements, labels=unique_elements, autopct='%1.1f%%',
#         shadow=True, startangle=90, textprops={'fontsize': 18})

# plt.show()

# log_reg_smote = LogisticRegression(max_iter = 1000).fit(X_train_smote, y_train_smote)
# y_pred_smote = log_reg_smote.predict(X_test)

# sns.set_context("poster")

# disp = plot_confusion_matrix(log_reg_smote, X_test, y_test,
#                      cmap = 'cividis', colorbar=False)

# print("Accuracy: ", accuracy_score(y_test, y_pred_smote))
# print("Precision: ", precision_score(y_test, y_pred_smote))
# print("Recall: ", recall_score(y_test, y_pred_smote))
# print("Classifier Score: ", log_reg_smote.score(X_test, y_test))
```

> Using smote, I was able to lower the false negatives by 1, but it raised the false positives quite a bit. Because of this, I will prefer the rebalancing using random over sampling, but let's keep trying new methods.

> I'm going to try using the random under sampler now. This will under sample the majority dataset.

```
# rus = RandomUnderSampler(random_state=0)
# X_train_rus, y_train_rus = rus.fit_resample(X_train, y_train)
```

```
 3
 4 # plt.rcParams["figure.figsize"] = (6,6)
 5 # unique_elements, counts_elements = np.unique(y_train_rus, return_counts=True)
 6 # fig1, ax1 = plt.subplots()
 7
 8 # ax1.pie(counts_elements, labels=unique_elements, autopct='%1.1f%%',
 9 #         shadow=True, startangle=90, textprops={'fontsize': 18})
10
11 # plt.show()
12
13 # log_reg_rus = LogisticRegression(max_iter = 1000).fit(X_train_rus, y_train_rus)
14 # y_pred_rus = log_reg_rus.predict(X_test)
15
16 # sns.set_context("poster")
17
18 # disp = plot_confusion_matrix(log_reg_rus, X_test, y_test,
19 #                      cmap = 'cividis', colorbar=False)
20
21 # print("Accuracy: ", accuracy_score(y_test, y_pred_rus))
22 # print("Precision: ", precision_score(y_test, y_pred_rus))
23 # print("Recall: ", recall_score(y_test, y_pred_rus))
24 # print("Classifier Score: ", log_reg_rus.score(X_test, y_test))
```

> It seems like the random under sampler provides the worst rebalaning of all algorithms so far (aside from the first). From the random over sampler, it simply increased the number of false positives. We won't be using this algorithm.

> Now I'll try using a bagging classifier with my best performing algorithm. That would be the random over sampler.

```
 1 # here you can see that im using bagging classifier, but im setting the base estimator to
 2 # also make sure to set the class weight to "balanced" again here, since we proved that it
 3 # log_reg_bc = BaggingClassifier(base_estimator=LogisticRegression(max_iter=1000, class_we
 4 #                                 n_estimators=10,
 5 #                                 random_state=0).fit(X_train_ros, y_train_ros)
 6 # y_pred_bc = log_reg_bc.predict(X_test)
 7
 8 # sns.set_context("poster")
 9
10 # disp = plot_confusion_matrix(log_reg_bc, X_test, y_test,
11 #                      cmap = 'cividis', colorbar=False)
12
13 # print("Accuracy: ", accuracy_score(y_test, y_pred_bc))
14 # print("Precision: ", precision_score(y_test, y_pred_bc))
15 # print("Recall: ", recall_score(y_test, y_pred_bc))
16 # print("Classifier Score: ", log_reg_bc.score(X_test, y_test))
```

> It seems that the bagging classifier provides decent resutls, but the false positive rate is still higher than simple random over sampling. Because of this, our best resampled data will be X_train_ros and y_train_ros and our best Logistic Regression algorithm will be log_reg_ros.

> I'm going to set the values from the random over sampler to be more simple, since we are going to use these for the rest of the project.

```
# simplifying vars
#X_train = X_train_ros
#y_train = y_train_ros
#log_reg = log_reg_ros
#y_pred = y_pred_ros

# creating classification report
log_reg_report = classification_report(y_test, y_pred_ros)
print(log_reg_report)
```

```
                precision    recall  f1-score   support

             0       1.00      0.98      0.99     85297
             1       0.07      0.94      0.13       146

      accuracy                           0.98     85443
     macro avg       0.53      0.96      0.56     85443
  weighted avg       1.00      0.98      0.99     85443
```

> As you can see in the classification report, our recall is very high for both the majority and minority class!

> Now I'll use cross val predict to train the model on this data

```
# note that y_scores could be higher if I useed the unbalanced dataset, but it would perfo
y_train_pred_log_reg = cross_val_predict(log_reg_ros, X_train_ros, y_train_ros, cv = 3)
y_scores_log_reg = cross_val_score(log_reg_ros, X_train_ros, y_train_ros, cv = 3)
print(y_scores_log_reg)
```

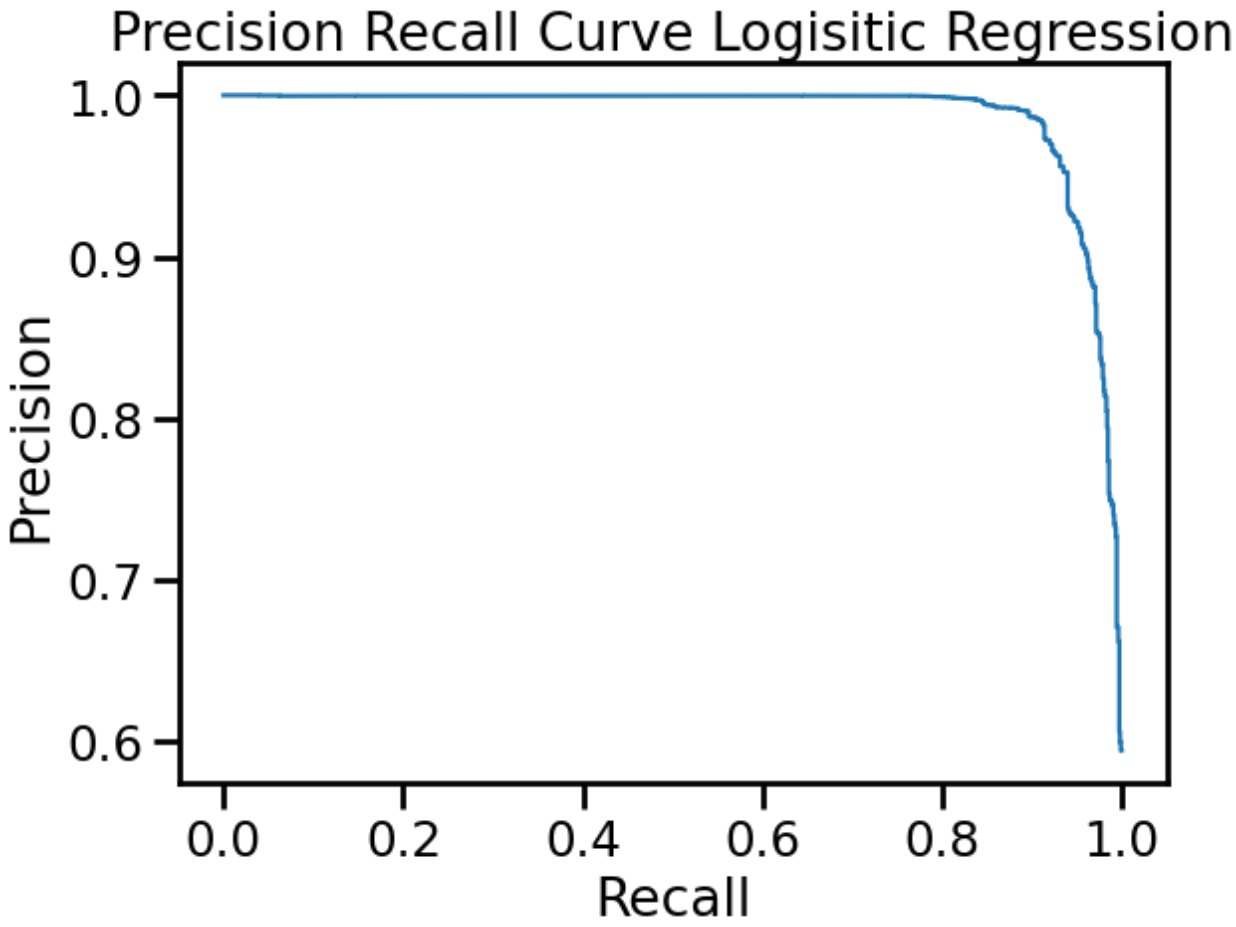
```
[0.94501767 0.94372131 0.94603476]
```

```
# i'll have to train y_scores with cross val predict so i can plot it on a precision and r
# im using decision_function since the classification is binary
y_scores_log_reg = cross_val_predict(log_reg_ros, X_train_ros, y_train_ros, cv = 3, method
```

```
precisions_log_reg, recalls_log_reg, thresholds_log_reg = precision_recall_curve(y_train_r
```

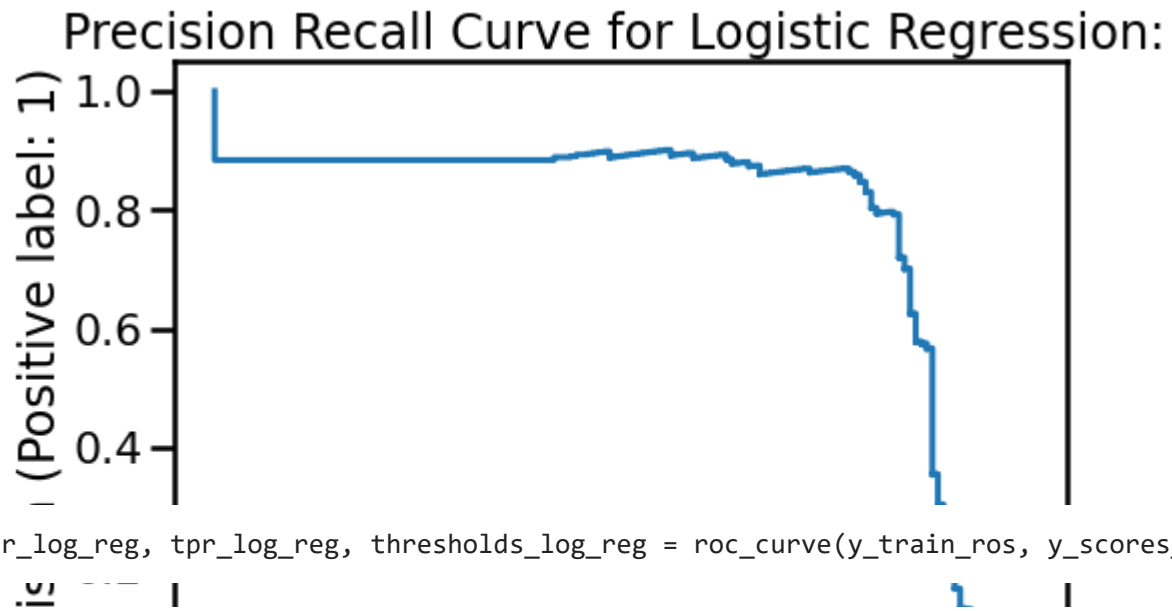
```
figure(figsize=(8, 6), dpi=80)
plt.plot(recalls_log_reg, precisions_log_reg, linewidth=2, label='Logistic')
plt.xlabel('Recall')
plt.title('Precision Recall Curve Logisitic Regression')
plt.ylabel('Precision')
```

```
Text(0, 0.5, 'Precision')
```




```
fig, ax = plt.subplots(figsize=(8, 6))
disp = plot_precision_recall_curve(log_reg_ros, X_test, y_test, name = "Logistic Regressio
disp.ax_.set_title('Precision Recall Curve for Logistic Regression: ')
```

```
/usr/local/lib/python3.7/dist-packages/sklearn/utils/deprecation.py:87: FutureWarn
  warnings.warn(msg, category=FutureWarning)
Text(0.5, 1.0, 'Precision Recall Curve for Logistic Regression: ')
```




```
1 fpr_log_reg, tpr_log_reg, thresholds_log_reg = roc_curve(y_train_ros, y_scores_log_reg)
```

```
1 fig, ax = plt.subplots(figsize=(8, 6))
2 metrics.plot_roc_curve(log_reg_ros, X_test, y_test, ax=ax)
3 plt.plot([0,1],[0,1], 'k--')
```

```
/usr/local/lib/python3.7/dist-packages/sklearn/utils/deprecation.py:87: FutureWarn
```

```
# check the auc score
auc_score = roc_auc_score(y_test, y_pred_ros)
print(auc_score)
```

```
0.9584801678454381
```

> Everything looks good! I rebalanced tha data multiple times, found the best fit, then trained the data. Then I showed the metrics I should pay attention to when assessing the performance of my model. Finally, the auc score shows that the model performs well!

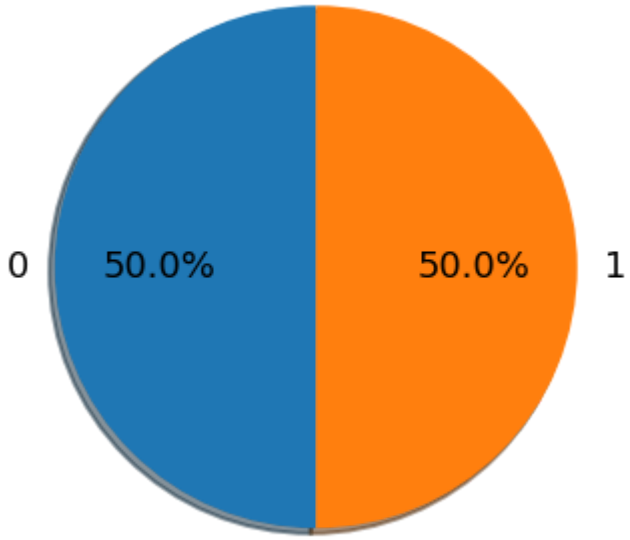
Let's try to use decision trees now:

> I'll be using the same rebalanced X_train and y_train, but I'll need to create a new classification algorithm and find/assess a new y_pred.

```
#X_train, y_train = smote.fit_resample(X_train, y_train)
#class_weight="balanced"

#d_tree = BaggingClassifier(base_estimator=DecisionTreeClassifier(max_depth=5, random_stat
#                           n_estimators=10,
#                           random_state=0).fit(X_train, y_train)

d_tree = DecisionTreeClassifier(max_depth = 5, random_state = 42)
d_tree.fit(X_train, y_train)
y_pred_dt = d_tree.predict(X_test)

plt.rcParams["figure.figsize"] = (6,6)
unique_elements, counts_elements = np.unique(y_train, return_counts=True)
fig1, ax1 = plt.subplots()

ax1.pie(counts_elements, labels=unique_elements, autopct='%1.1f%%',
        shadow=True, startangle=90, textprops={'fontsize': 18})

plt.show()

#d_tree = DecisionTreeClassifier(max_depth=5, random_state=42, class_weight="balanced").fi
#y_pred_dt = d_tree.predict(X_test)

sns.set_context("poster")

disp = plot_confusion_matrix(d_tree, X_test, y_test,
                     cmap = 'cividis', colorbar=False)
```

```
28
29 print("Accuracy: ", accuracy_score(y_test, y_pred_dt))
30 print("Precision: ", precision_score(y_test, y_pred_dt))
31 print("Recall: ", recall_score(y_test, y_pred_dt))
32 print("Classifier Score: ", d_tree.score(X_test, y_test))
```

```
#X_train, y_train = smote.fit_resample(X_train, y_train)
#class_weight="balanced"

#d_tree = BaggingClassifier(base_estimator=DecisionTreeClassifier(max_depth=5, random_stat
#                           n_estimators=10,
#                           random_state=0).fit(X_train, y_train)

d_tree_ros = DecisionTreeClassifier(max_depth = 5, random_state = 42)
d_tree_ros.fit(X_train_ros, y_train_ros)
y_pred_dt_ros = d_tree_ros.predict(X_test)

plt.rcParams["figure.figsize"] = (6,6)
unique_elements, counts_elements = np.unique(y_train_ros, return_counts=True)
fig1, ax1 = plt.subplots()

ax1.pie(counts_elements, labels=unique_elements, autopct='%1.1f%%',
        shadow=True, startangle=90, textprops={'fontsize': 18})

plt.show()

#d_tree = DecisionTreeClassifier(max_depth=5, random_state=42, class_weight="balanced").fi
#y_pred_dt = d_tree.predict(X_test)

sns.set_context("poster")

disp = plot_confusion_matrix(d_tree_ros, X_test, y_test,
                     cmap = 'cividis', colorbar=False)

print("Accuracy: ", accuracy_score(y_test, y_pred_dt_ros))
print("Precision: ", precision_score(y_test, y_pred_dt_ros))
print("Recall: ", recall_score(y_test, y_pred_dt_ros))
print("Classifier Score: ", d_tree_ros.score(X_test, y_test))
```




```
/usr/local/lib/python3.7/dist-packages/sklearn/utils/deprecation.py:87: FutureWarn
  warnings.warn(msg, category=FutureWarning)
Accuracy:  0.9735145067471882
Precision:  0.05581200167855644
Recall:  0.910958904109589
Classifier Score:  0.9735145067471882
```

> Using the default dataset without rebalancing works acceptably here. I tried to rebalance and it wasn't worth the tradeoff.

> I will now train the data using the d_tree classifier

```
1 # note that i am blowing away the old vals for these vars to keep naming simple
2 y_train_pred_dt = cross_val_predict(d_tree, X_train, y_train, cv = 3)
3
4 y_scores_dt = cross_val_score(d_tree, X_train, y_train, cv = 3)
```

```
1 print("Accuracy: ", accuracy_score(y_train, y_train_pred_dt))
2 print("Precision: ", precision_score(y_train, y_train_pred_dt))
3 print("Recall: ", recall_score(y_train, y_train_pred_dt))
4 print("Classifier Score: ", d_tree.score(X_test, y_test))
```

```
Accuracy:  0.9993730061595875
Precision:  0.8850174216027874
Recall:  0.7341040462427746
Classifier Score:  0.9995084442259752
```

> Recall actually is pretty low. Let's train it with ROS.

```
# note that i am blowing away the old vals for these vars to keep naming simple
y_train_pred_dt_ros = cross_val_predict(d_tree_ros, X_train_ros, y_train_ros, cv = 3)

y_scores_dt_ros = cross_val_score(d_tree_ros, X_train_ros, y_train_ros, cv = 3)

y_train_pred_dt = cross_val_predict(d_tree, X_train, y_train, cv = 3)

y_scores_dt = cross_val_score(d_tree, X_train, y_train, cv = 3)
```

```
print("With ROS: ")
print("Accuracy: ", accuracy_score(y_train_ros, y_train_pred_dt_ros))
print("Precision: ", precision_score(y_train_ros, y_train_pred_dt_ros))
print("Recall: ", recall_score(y_train_ros, y_train_pred_dt_ros))
print("Classifier Score: ", d_tree_ros.score(X_test, y_test))

print("Without ROS: ")
print("Accuracy: ", accuracy_score(y_train, y_train_pred_dt))
print("Precision: ", precision_score(y_train, y_train_pred_dt))
print("Recall: ", recall_score(y_train, y_train_pred_dt))
print("Classifier Score: ", d_tree.score(X_test, y_test))
```

```
With ROS:
Accuracy:  0.9583605503019827
Precision:  0.9738411352822622
Recall:  0.9420253444412063
Classifier Score:  0.9735145067471882
Without ROS:
Accuracy:  0.9993730061595875
Precision:  0.8850174216027874
Recall:  0.7341040462427746
Classifier Score:  0.9995084442259752
```

> Okay, looks like the results are way better using ROS, so I'll stick with that.

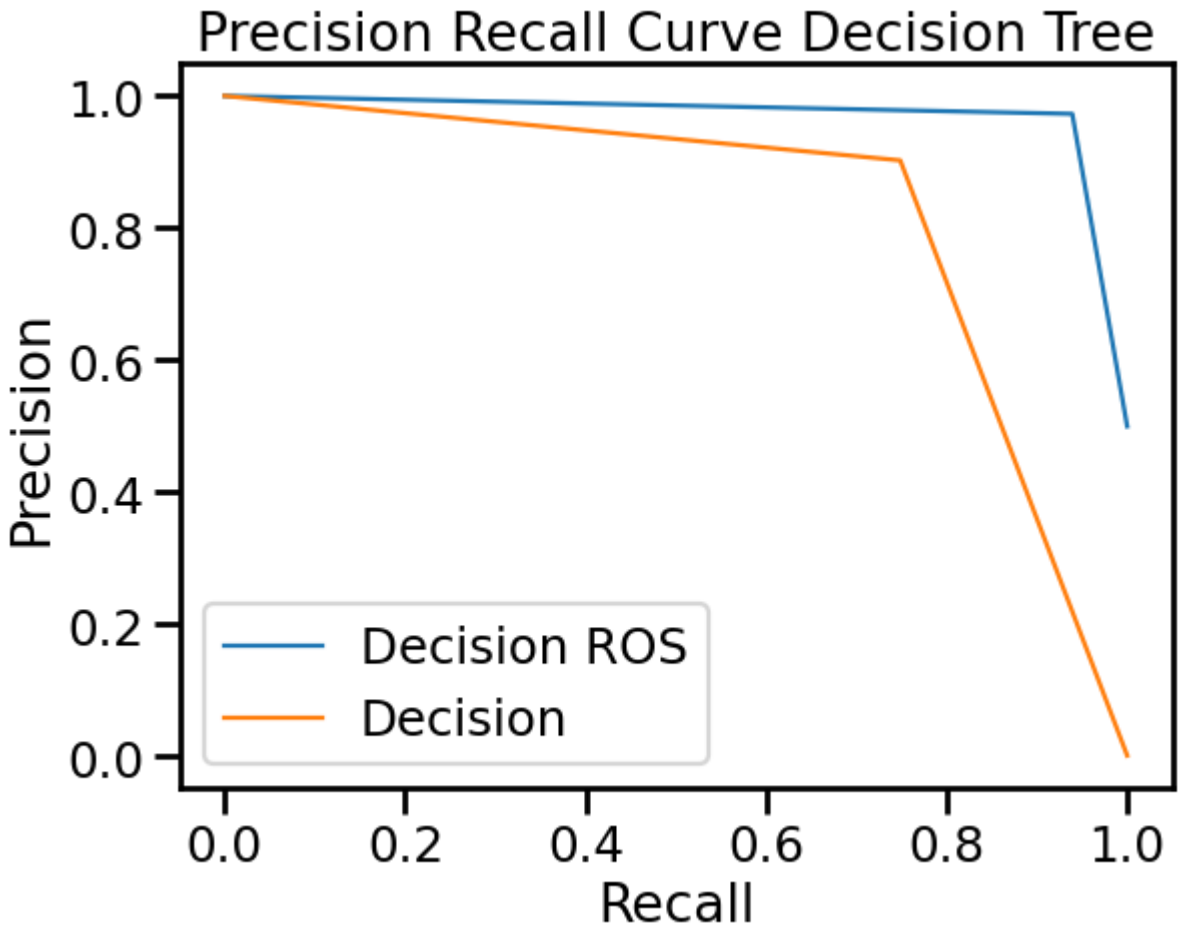
> Decision Trees seem to be preforming worse than logistic regression. I am not able to get the false negatives to lower under 11 without making false positives skyrocket, and that's using a custom weight.

```
# training again to makes scores usable in graphing against y_train_pred
y_scores_dt_ros = cross_val_predict(d_tree_ros, X_train_ros, y_train_pred_dt_ros, cv = 3)
y_scores_dt = cross_val_predict(d_tree, X_train, y_train_pred_dt, cv = 3)
```

```
precisions_dt_ros, recalls_dt_ros, thresholds_dt_ros = precision_recall_curve(y_train_ros,
precisions_dt, recalls_dt, thresholds_dt = precision_recall_curve(y_train, y_scores_dt)
```

```
figure(figsize=(8, 6), dpi=80)
plt.plot(recalls_dt_ros, precisions_dt_ros, linewidth=2, label='Decision ROS')
plt.plot(recalls_dt, precisions_dt, linewidth=2, label='Decision')
plt.xlabel('Recall')
plt.title('Precision Recall Curve Decision Tree')
plt.ylabel('Precision')
plt.legend()
```

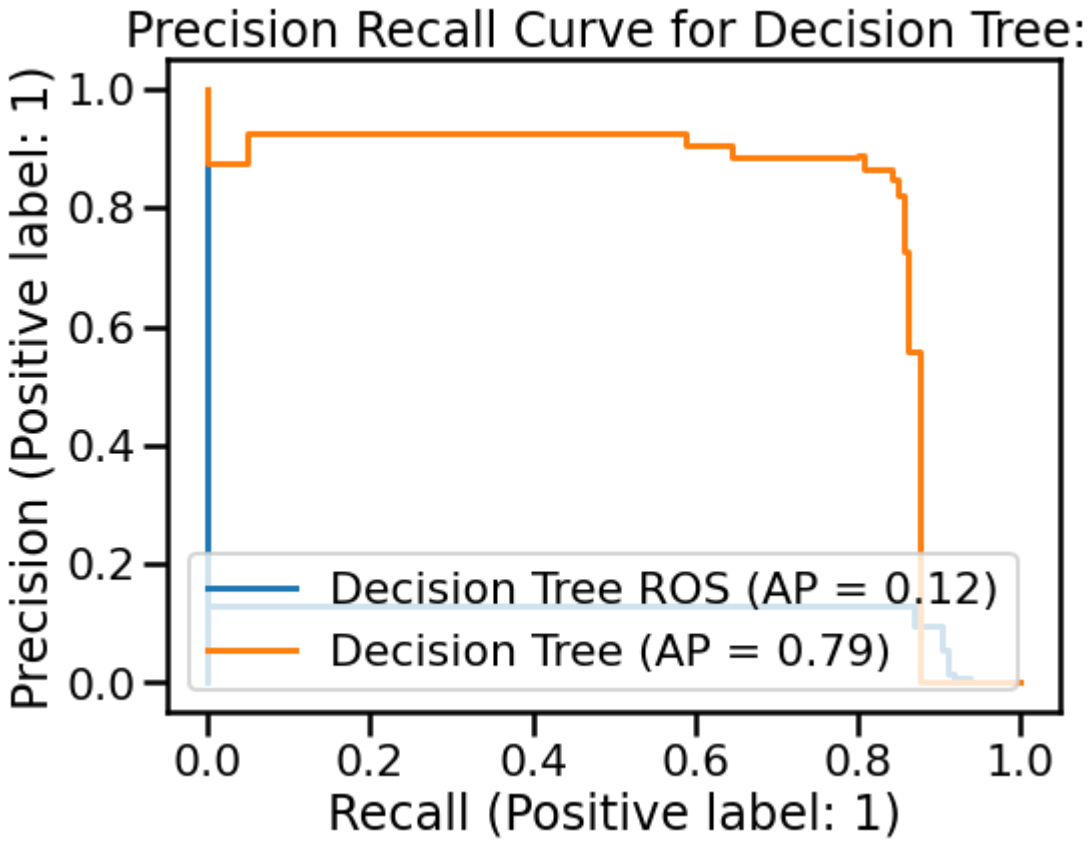
```
<matplotlib.legend.Legend at 0x7f74d617c5d0>
```



> Precision is so low with ROS that the PRC looks like this. Lets look at the PRC with the original data to compare.

```
fig, ax = plt.subplots(figsize=(8, 6))
disp = plot_precision_recall_curve(d_tree_ros, X_test, y_test, name = "Decision Tree ROS",
disp = plot_precision_recall_curve(d_tree, X_test, y_test, name = "Decision Tree", ax=ax)
disp.ax_.set_title('Precision Recall Curve for Decision Tree: ')
```

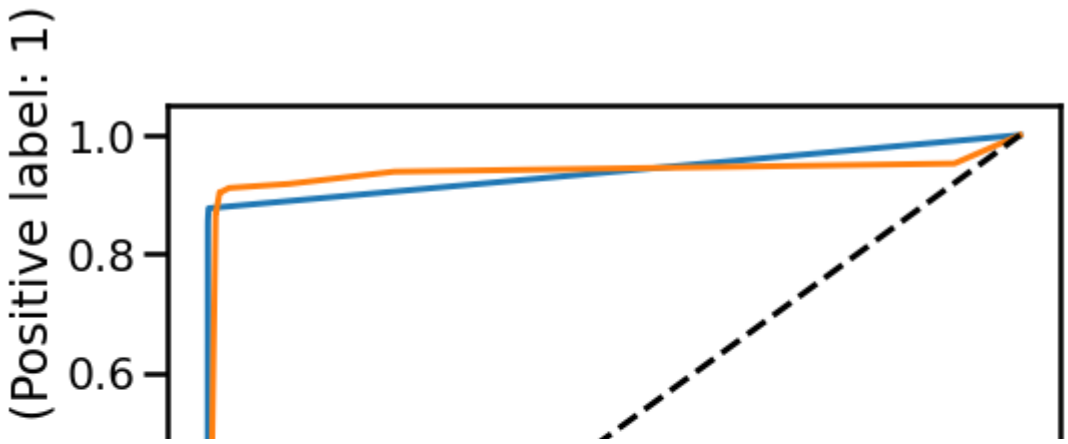
```
/usr/local/lib/python3.7/dist-packages/sklearn/utils/deprecation.py:87: FutureWarn
  warnings.warn(msg, category=FutureWarning)
/usr/local/lib/python3.7/dist-packages/sklearn/utils/deprecation.py:87: FutureWarn
  warnings.warn(msg, category=FutureWarning)
Text(0.5, 1.0, 'Precision Recall Curve for Decision Tree: ')
```




```
1 fpr_d_tree_ros, tpr_d_tree_ros, thresholds_d_tree_ros = roc_curve(y_train_ros, y_scores_dt
2 fpr_d_tree, tpr_d_tree, thresholds_d_tree = roc_curve(y_train, y_scores_dt)
```

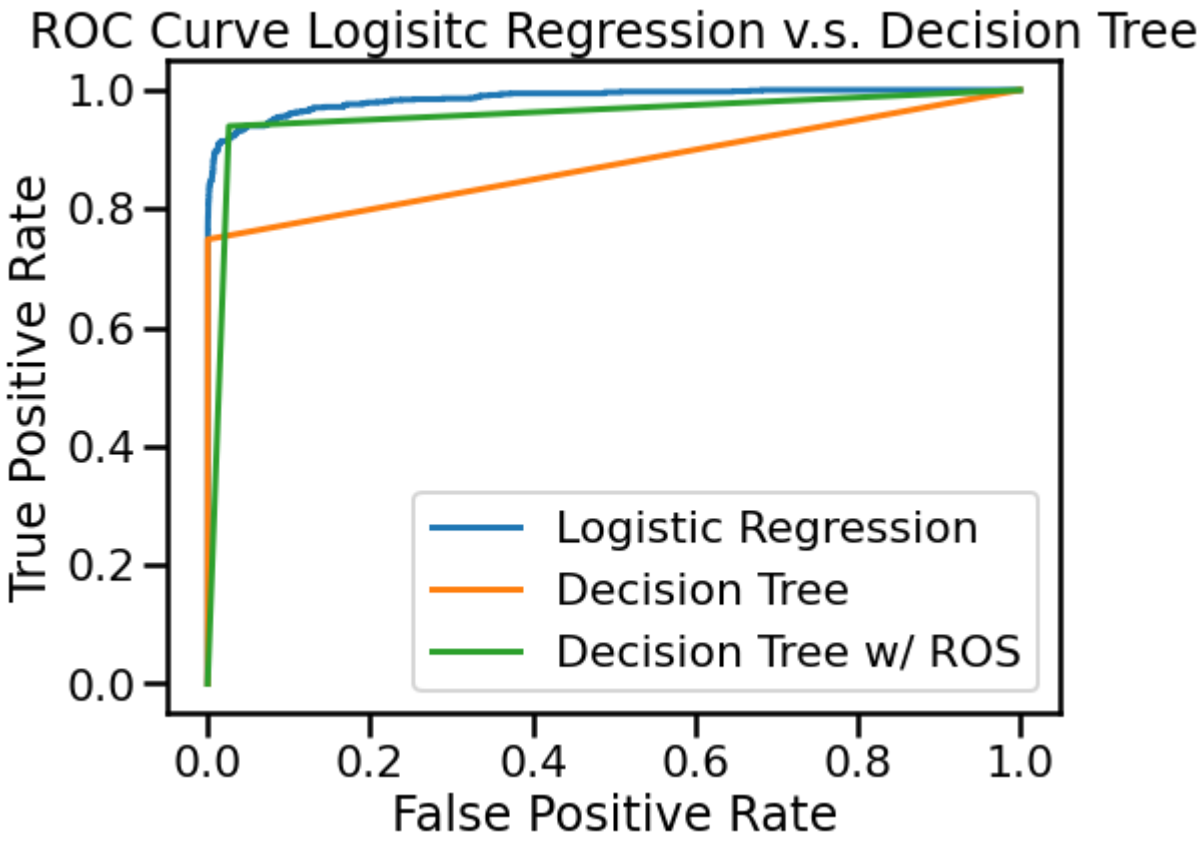
```
1 fig, ax = plt.subplots(figsize=(8, 6))
2 metrics.plot_roc_curve(d_tree, X_test, y_test, ax=ax, label="No Balanacing")
3 metrics.plot_roc_curve(d_tree_ros, X_test, y_test, ax=ax, label="ROS")
4 plt.plot([0,1],[0,1], 'k--')
```

```
/usr/local/lib/python3.7/dist-packages/sklearn/utils/deprecation.py:87: FutureWarn
  warnings.warn(msg, category=FutureWarning)
/usr/local/lib/python3.7/dist-packages/sklearn/utils/deprecation.py:87: FutureWarn
  warnings.warn(msg, category=FutureWarning)
[<matplotlib.lines.Line2D at 0x7f74d6488710>]
```




```
1 fig, ax = plt.subplots(figsize=(8, 6))
2 plt.plot(fpr_log_reg, tpr_log_reg, label="Logistic Regression")
3 plt.plot(fpr_d_tree, tpr_d_tree, label="Decision Tree")
4 plt.plot(fpr_d_tree_ros, tpr_d_tree_ros, label="Decision Tree w/ ROS")
5 plt.xlabel("False Positive Rate")
6 plt.ylabel("True Positive Rate")
7 plt.title("ROC Curve Logisitc Regression v.s. Decision Tree")
8
9 plt.legend(loc=0)
```

```
<matplotlib.legend.Legend at 0x7f74d64f74d0>
```



> As I said earlier, the algorithm for logistic regression behaved much nicer with balancing algorithms. The best I could do with decisions trees performs much worse compared to logistic regression as shown by the graph above.

Now, Let's try a Neural Network

```
def create_model():
    model = keras.models.Sequential()
    model.add(keras.layers.Dense(100, activation = 'relu'))
    model.add(keras.layers.Dense(100, activation = 'relu'))
    model.add(keras.layers.Dense(2, activation = 'sigmoid'))
    model.compile(loss = "sparse_categorical_crossentropy", optimizer = 'adam', metrics =
    return model
```

```
NN_model = create_model()
model = KerasClassifier(build_fn = create_model, epochs = 5, batch_size = 50, verbose = 0)
history = NN_model.fit(X_train, y_train, epochs = 5)
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:2: DeprecationWarning: Kera

Epoch 1/5
6231/6231 [==============================] - 15s 2ms/step - loss: 0.0058 - accuracy: 0.9
Epoch 2/5
6231/6231 [==============================] - 14s 2ms/step - loss: 0.0034 - accuracy: 0.9
Epoch 3/5
6231/6231 [==============================] - 14s 2ms/step - loss: 0.0031 - accuracy: 0.9
Epoch 4/5
6231/6231 [==============================] - 15s 2ms/step - loss: 0.0030 - accuracy: 0.9
Epoch 5/5
6231/6231 [==============================] - 14s 2ms/step - loss: 0.0026 - accuracy: 0.9
```

```
NN_model_ros = create_model()
model_ros = KerasClassifier(build_fn = create_model, epochs = 5, batch_size = 50, verbose
history_ros = NN_model.fit(X_train_ros, y_train_ros, epochs = 5)
```

```
Epoch 1/5
   20/12439 [..............................] - ETA: 33s - loss: 0.1671 - accuracy: 0.950

12439/12439 [==============================] - 28s 2ms/step - loss: 0.0066 - accuracy: 0
Epoch 2/5
12439/12439 [==============================] - 28s 2ms/step - loss: 0.0031 - accuracy: 0
Epoch 3/5
12439/12439 [==============================] - 28s 2ms/step - loss: 0.0023 - accuracy: 0
Epoch 4/5
12439/12439 [==============================] - 29s 2ms/step - loss: 0.0024 - accuracy: 0
```

```
Epoch 5/5
12439/12439 [==============================] - 28s 2ms/step - loss: 0.0018 - accuracy: 0
```

> Looks like once again the ROS set ended up with better results in both loss and accuracy!

```
predict_y = NN_model.predict(X_test)
classes_y = np.argmax(predict_y,axis=1)

predict_y_ros = NN_model_ros.predict(X_test)
classes_y_ros = np.argmax(predict_y_ros,axis=1)
```

```
from sklearn.metrics import confusion_matrix
NN_cnf = confusion_matrix(classes_y, y_test)
NN_cnf_ros = confusion_matrix(classes_y_ros, y_test)

print("CNF w/o ROS: ")
print(NN_cnf)
print("Accuracy: ", accuracy_score(y_test, classes_y))
print("Precision: ", precision_score(y_test, classes_y))
print("Recall: ", recall_score(y_test, classes_y))

print("CNF w/ ROS: ")
print(NN_cnf_ros)
print("Accuracy: ", accuracy_score(y_test, classes_y_ros))
print("Precision: ", precision_score(y_test, classes_y_ros))
print("Recall: ", recall_score(y_test, classes_y_ros))
```

```
CNF w/o ROS:
[[85254    24]
 [   43   122]]
Accuracy:  0.9992158515033414
Precision:  0.7393939393939394
Recall:  0.8356164383561644
CNF w/ ROS:
[[ 5206     4]
 [80091   142]]
Accuracy:  0.06259143522582307
Precision:  0.0017698453254895118
Recall:  0.9726027397260274
```

```
NN_report = classification_report(y_test, classes_y)
NN_report_ros = classification_report(y_test, classes_y_ros)
print(NN_report)
print(NN_report_ros)
```

```
              precision    recall  f1-score   support
```

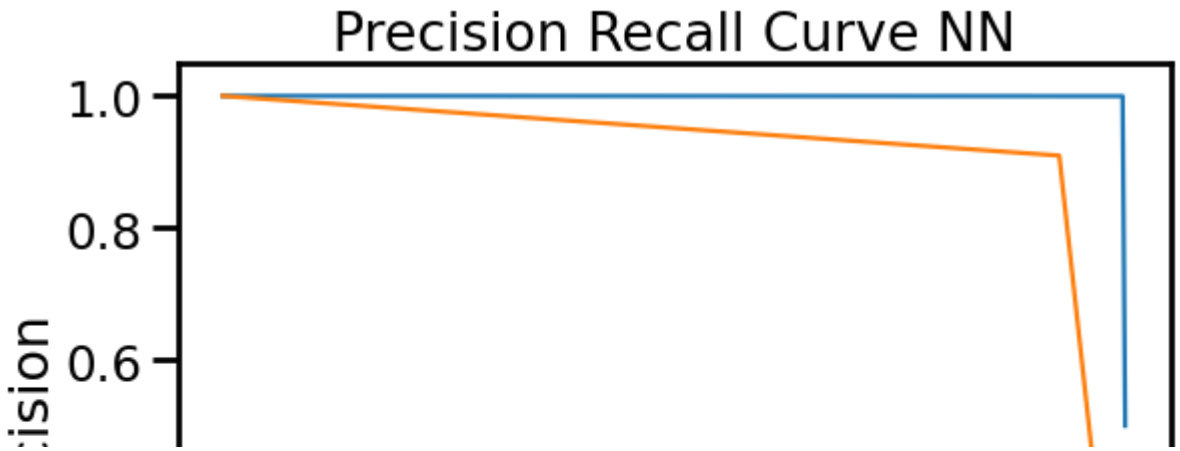
```
                0       1.00      1.00      1.00     85297
                1       0.74      0.84      0.78       146

         accuracy                           1.00     85443
        macro avg       0.87      0.92      0.89     85443
     weighted avg       1.00      1.00      1.00     85443

                   precision    recall  f1-score   support

                0       1.00      0.06      0.12     85297
                1       0.00      0.97      0.00       146

         accuracy                           0.06     85443
        macro avg       0.50      0.52      0.06     85443
     weighted avg       1.00      0.06      0.11     85443
```

```
y_scores_NN = cross_val_predict(model, X_train, y_train, cv = 3)
y_train_pred_NN = cross_val_predict(model, X_train, y_train, cv = 3)

precisions_NN, recalls_NN, thresholds_NN = precision_recall_curve(y_train_pred_NN, y_score

```

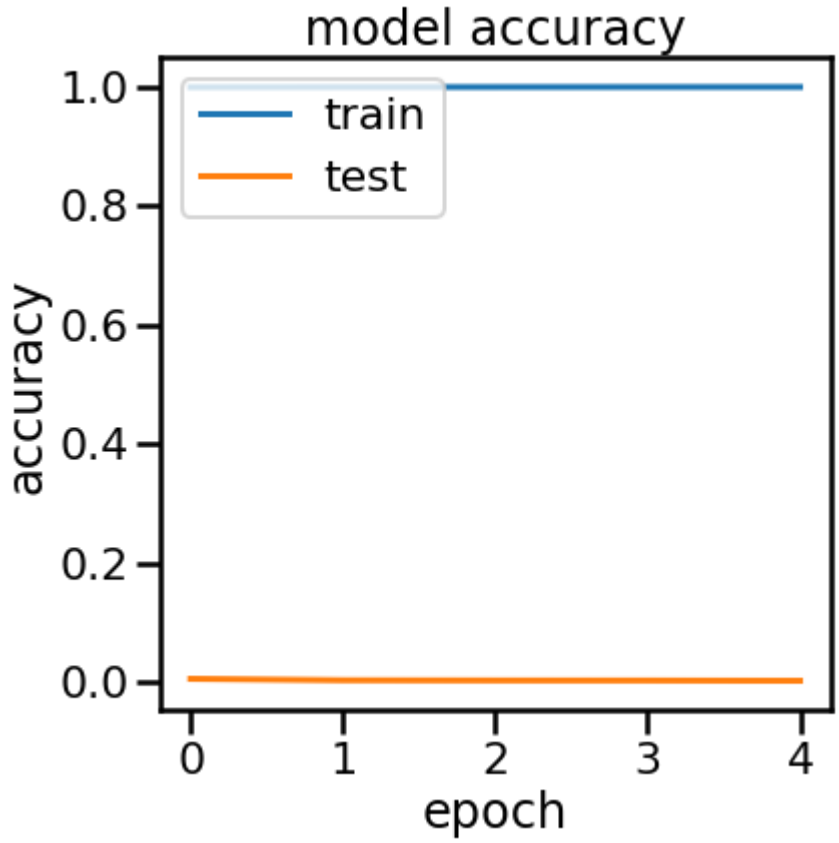
```
y_scores_NN_ros = cross_val_predict(model, X_train_ros, y_train_ros, cv = 3)
y_train_pred_NN_ros = cross_val_predict(model, X_train_ros, y_train_ros, cv = 3)

precisions_NN_ros, recalls_NN_ros, thresholds_NN_ros = precision_recall_curve(y_train_pred

```

```
figure(figsize=(8, 6), dpi=80)
plt.plot(recalls_NN_ros, precisions_NN_ros, linewidth=2, label='NN ROS')
plt.plot(recalls_NN, precisions_NN, linewidth=2, label='NN No ROS')
plt.xlabel('Recall')
plt.title('Precision Recall Curve NN')
plt.ylabel('Precision')
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7f74d1f66e90>
```




```
plt.plot(history.history['accuracy'])
plt.plot(history.history['loss'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```



```
%matplotlib inline

import os
import requests
import pandas as pd
import matplotlib.pyplot as plt
import sklearn

from sklearn.metrics import r2_score
from sklearn.metrics import mean_squared_error as mse
```

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
from imblearn.over_sampling import RandomOverSampler
from imblearn.over_sampling import SMOTE
from imblearn.under_sampling import RandomUnderSampler
from sklearn.linear_model import LinearRegression
from sklearn.metrics import plot_confusion_matrix
from sklearn.ensemble import BaggingClassifier
import seaborn as sns
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score
from sklearn.metrics import f1_score
from sklearn.model_selection import cross_val_predict
from sklearn.model_selection import cross_val_score
from sklearn.metrics import precision_recall_curve
from sklearn.metrics import plot_precision_recall_curve
from sklearn.metrics import roc_curve
from sklearn.metrics import roc_auc_score
from sklearn.tree import DecisionTreeClassifier
import tensorflow as tf
from sklearn.compose import ColumnTransformer
from tensorflow import keras
from keras.models import Sequential
from keras.layers import Dense
from sklearn.preprocessing import LabelEncoder, OneHotEncoder
from keras.wrappers.scikit_learn import KerasClassifier
from sklearn.metrics import classification_report
from matplotlib.pyplot import figure
from sklearn import metrics
```

> For example, for solving a regression problem, you may use linear regression, polynomial regression, regularized regression (either linear or polynomial)

```
# import the dataset
from google.colab import drive
drive.mount('/content/drive')
data = pd.read_excel("/content/drive/My Drive/School/Machine Learning/Final/ENB2012_data.x
print(data)
```

```
Mounted at /content/drive
       X1     X2     X3      X4   X5  X6   X7  X8     Y1     Y2
0    0.98  514.5  294.0  110.25  7.0   2  0.0   0  15.55  21.33
1    0.98  514.5  294.0  110.25  7.0   3  0.0   0  15.55  21.33
2    0.98  514.5  294.0  110.25  7.0   4  0.0   0  15.55  21.33
3    0.98  514.5  294.0  110.25  7.0   5  0.0   0  15.55  21.33
4    0.90  563.5  318.5  122.50  7.0   2  0.0   0  20.84  28.28
..    ...    ...    ...     ...  ...  ..  ...  ..    ...    ...
763  0.64  784.0  343.0  220.50  3.5   5  0.4   5  17.88  21.40
764  0.62  808.5  367.5  220.50  3.5   2  0.4   5  16.54  16.88
765  0.62  808.5  367.5  220.50  3.5   3  0.4   5  16.44  17.11
766  0.62  808.5  367.5  220.50  3.5   4  0.4   5  16.48  16.61
767  0.62  808.5  367.5  220.50  3.5   5  0.4   5  16.64  16.03

[768 rows x 10 columns]
```

```
data.isnull().sum()
```

```
X1    0
X2    0
X3    0
X4    0
X5    0
X6    0
X7    0
X8    0
Y1    0
Y2    0
dtype: int64
```

```
data.dropna()
```

	X1	X2	X3	X4	X5	X6	X7	X8	Y1	Y2
0	0.98	514.5	294.0	110.25	7.0	2	0.0	0	15.55	21.33
1	0.98	514.5	294.0	110.25	7.0	3	0.0	0	15.55	21.33
2	0.98	514.5	294.0	110.25	7.0	4	0.0	0	15.55	21.33
3	0.98	514.5	294.0	110.25	7.0	5	0.0	0	15.55	21.33
4	0.90	563.5	318.5	122.50	7.0	2	0.0	0	20.84	28.28

```
1 data.columns = ['compactness','surface_area','wall_area','roof_area','height',\
2                 'orientation','glazing_area','distribution','heating_load','cooling_load
```

763	0.64	784.0	343.0	220.50	3.5	5	0.4	5	17.88	21.40

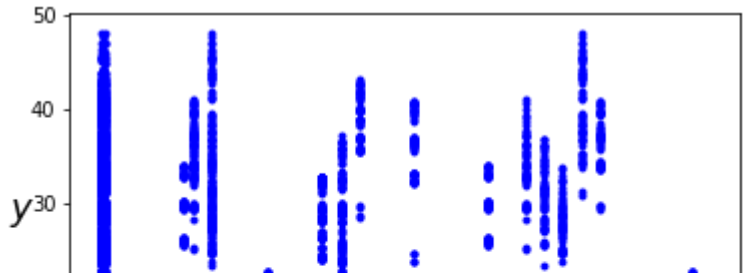
```
1 data.describe()
```

	compactness	surface_area	wall_area	roof_area	height	orientation	g
count	768.000000	768.000000	768.000000	768.000000	768.00000	768.000000	
mean	0.764167	671.708333	318.500000	176.604167	5.25000	3.500000	
std	0.105777	88.086116	43.626481	45.165950	1.75114	1.118763	
min	0.620000	514.500000	245.000000	110.250000	3.50000	2.000000	
25%	0.682500	606.375000	294.000000	140.875000	3.50000	2.750000	
50%	0.750000	673.750000	318.500000	183.750000	5.25000	3.500000	
75%	0.830000	741.125000	343.000000	220.500000	7.00000	4.250000	
max	0.980000	808.500000	416.500000	220.500000	7.00000	5.000000	

```
1 X = data.drop(['heating_load','cooling_load'], axis=1)
2
3 y = data[['heating_load','cooling_load']]
4 y_heat = data[['heating_load']]
5 y_cool = data[['cooling_load']]
```

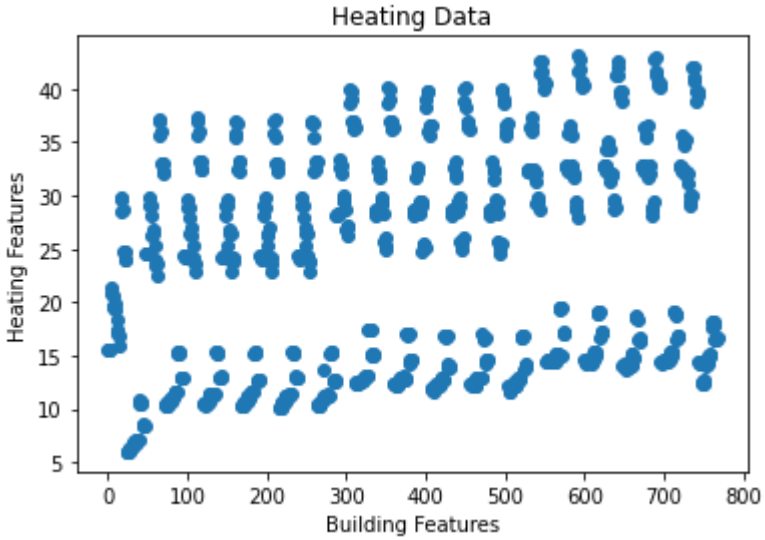
```
1 plt.plot(X, y, "b.")
2 plt.xlabel("$x_1$", fontsize=18)
3 plt.ylabel("$y$", rotation=0, fontsize=18)
4 plt.show()
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:1: MatplotlibDeprecat
  """Entry point for launching an IPython kernel.
```



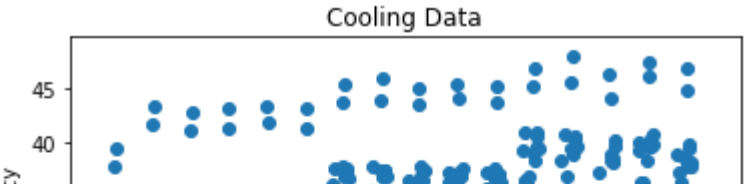

```
1 import matplotlib.pyplot as plt
2 X1 = np.arange(0,len(X),1)
3 plt.scatter(X1, y_heat)
4 plt.xlabel("Building Features")
5 plt.ylabel("Heating Features")
6 plt.title("Heating Data")
```

```
Text(0.5, 1.0, 'Heating Data')
```




```
1 import matplotlib.pyplot as plt
2 X1 = np.arange(0,len(X),1)
3 plt.scatter(X1, y_cool)
4 plt.xlabel("Building Features")
5 plt.ylabel("Cooling Efficiency")
6 plt.title("Cooling Data")
```

```
Text(0.5, 1.0, 'Cooling Data')
```




```
1 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3, random_state =
2 print(X_train.shape)
3 print(X_test.shape)
4 print(y_train.shape)
5 print(y_test.shape)
```

```
(537, 8)
(231, 8)
(537, 2)
(231, 2)
```

```
1 scaler = StandardScaler()
2 X_train = scaler.fit_transform(X_train)
3 X_test = scaler.fit_transform(X_test)
```

```
1 model = LinearRegression()
2 model.fit(X_train, y_train)
3
4 expected = y_test
5 predicted = model.predict(X_test)
```

```
1 from sklearn.linear_model import LinearRegression
2 lin_reg = LinearRegression()
3 lin_reg.fit(X_train, y_train)
4 R2 = lin_reg.score(X_train, y_train)
5 print('R2:', R2)
6 print('intercept:', lin_reg.intercept_)
7 print('slope:', lin_reg.coef_)
```

```
R2: 0.9026379224548287
intercept: [22.05050279 24.28750466]
slope: [[-6.68692391 -3.70147238  0.76114173 -3.99242943  7.21681807 -0.02606662
   2.6700316   0.35604219]
 [-7.59913319 -4.16524045  0.06702459 -4.10865079  7.21120279  0.12145137
   1.93173983  0.07779349]]
```

```
1 y_pred = lin_reg.predict(X_test)
2 print(y_pred)
```

```
[35.32208765 36.23546719]
[13.33304605 16.15314628]
[17.20430841 18.54281722]
[ 8.25986377 12.68471528]
```

```
[39.60615865 39.41017518]
[27.97705297 29.38150793]
[30.46421575 31.82003158]
[33.31520648 33.87548931]
[10.44625997 13.82346147]
[ 7.69925992 11.72449054]
[26.47305883 29.51545307]
[33.54323972 33.92531346]
[31.80972898 33.92462356]
[32.60837499 33.83193027]
[28.72108232 30.89249705]
[28.49304908 30.84267289]
[12.02298746 14.93946963]
[27.32616352 30.34208002]
[17.18157664 18.64873065]
[28.67561878 31.10432391]
[29.55208279 31.62073495]
[10.8568629  14.13493664]
[29.82557958 31.45873225]
[32.58564321 33.9378437 ]
[31.60442751 33.76888597]
[18.28310719 19.04534408]
[10.84095902 14.17651586]
[14.14801622 16.3316219 ]
[11.50232697 14.43190176]
[27.73676646 30.65355519]
[32.83220632 34.73239865]
[33.76707105 34.82578185]
[ 9.13336108 13.03150455]
[ 9.26632526 12.73769975]
[32.80947455 34.83831208]

[ 7.9727567  11.56248784]
[31.03334516 31.59270325]
[12.39855604 14.67277761]
[35.16224973 35.86790274]
[10.40079642 14.03528833]
[14.06260931 16.0907919 ]
[33.2882728  34.83204696]
[33.31100457 34.72613354]
[ 6.06249772 10.84302062]
[15.31414078 17.13615489]
[32.81367645 33.98766785]
[14.03987754 16.19670533]
[29.59754634 31.40890809]
[11.1144558  14.01451316]
[27.99978474 29.2755945 ]
[13.12774458 15.9974087 ]
[11.31615597 14.89591059]
[34.88875295 36.02990544]
[24.23022923 27.88558527]
[16.51338081 18.45767896]
[36.11653175 36.99341812]
[ 7.03789197 11.46910464]
[31.8779243  33.60688327]
```

```
1 from sklearn.metrics import r2_score
```

```
lin_reg_y_pred = lin_reg.predict(X_test)
print('r2_score', r2_score(y_test, lin_reg_y_pred))
```

```
r2_score 0.8912863062954878
```

```
from sklearn.model_selection import cross_val_predict
from sklearn.model_selection import cross_val_score
y_train_pred = cross_val_predict(lin_reg, X_train, y_train, cv = 3)
y_scores = cross_val_score(lin_reg, X_train, y_train, cv = 3)
print(y_scores)
```

```
[0.90124043 0.90430618 0.89212117]
```

```
from sklearn.metrics import mean_squared_error
mean_squared_error(y_test, y_pred)
```

```
10.327209706449818
```

Polynomial Regression

```
from sklearn.preprocessing import PolynomialFeatures
poly = PolynomialFeatures(degree=8)

X_poly_train = poly.fit_transform(X_train)
X_poly_test = poly.fit_transform(X_test)

poly_reg = LinearRegression()
poly_reg.fit(X_poly_train, y_train)
```

```
LinearRegression()
```

```
poly_reg_y_pred = poly_reg.predict(X_poly_test)
print('r2_score', r2_score(y_test, poly_reg_y_pred))
```

```
r2_score -12.03610071473415
```

```
mean_squared_error(y_test, poly_reg_y_pred)
```

```
1207.6128808026892
```

```
from sklearn.model_selection import cross_val_predict
from sklearn.model_selection import cross_val_score
y_train_pred3 = cross_val_predict(poly_reg, X_train, y_train, cv = 3)
y_scores = cross_val_score(poly_reg, X_train, y_train, cv = 3)
print(y_scores)
```

```
[0.90124043 0.90430618 0.89212117]
```

NN

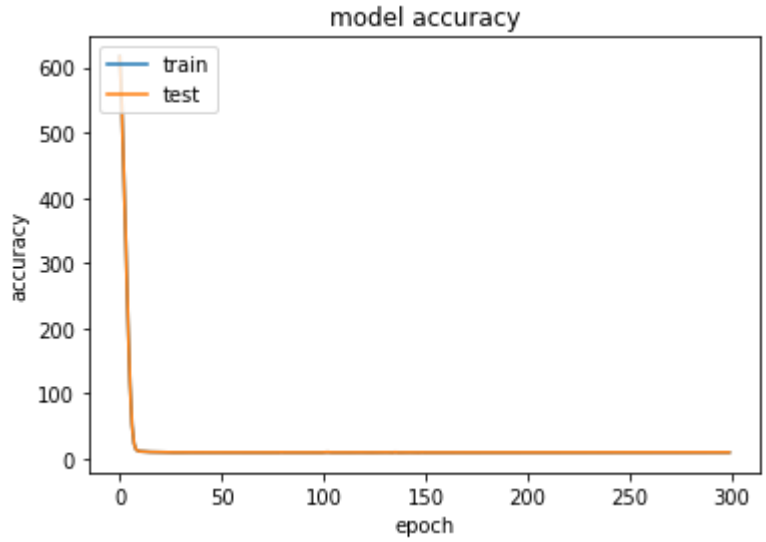
```
from keras.layers import Dense, Activation, Dropout
from keras.models import Sequential
model = Sequential()
model.add(Dense(200, input_dim=8))
model.add(Dense(2, activation = 'linear'))
model.compile(loss = "mse", optimizer = 'adam', metrics = ['mse'])
```

```
history = model.fit(X_train, y_train, validation_split = 0.2, epochs = 300, batch_size = 1
```

```
Epoch 1/300
43/43 [==============================] - 1s 5ms/step - loss: 616.4240 - mse: 616.4240
Epoch 2/300
43/43 [==============================] - 0s 2ms/step - loss: 540.9752 - mse: 540.9752
Epoch 3/300
43/43 [==============================] - 0s 2ms/step - loss: 438.5612 - mse: 438.5612
Epoch 4/300
43/43 [==============================] - 0s 2ms/step - loss: 329.4164 - mse: 329.4164
Epoch 5/300
43/43 [==============================] - 0s 2ms/step - loss: 215.1784 - mse: 215.1784
Epoch 6/300
43/43 [==============================] - 0s 2ms/step - loss: 116.7740 - mse: 116.7740
Epoch 7/300
43/43 [==============================] - 0s 2ms/step - loss: 52.0054 - mse: 52.0054 -
Epoch 8/300
43/43 [==============================] - 0s 2ms/step - loss: 23.0885 - mse: 23.0885 -
Epoch 9/300
43/43 [==============================] - 0s 2ms/step - loss: 13.7938 - mse: 13.7938 -
Epoch 10/300
43/43 [==============================] - 0s 2ms/step - loss: 11.7998 - mse: 11.7998 -
Epoch 11/300
43/43 [==============================] - 0s 2ms/step - loss: 11.1780 - mse: 11.1780 -
Epoch 12/300
43/43 [==============================] - 0s 2ms/step - loss: 10.9360 - mse: 10.9360 -
Epoch 13/300
43/43 [==============================] - 0s 2ms/step - loss: 10.7590 - mse: 10.7590 -
Epoch 14/300
43/43 [==============================] - 0s 2ms/step - loss: 10.5853 - mse: 10.5853 -
Epoch 15/300
43/43 [==============================] - 0s 2ms/step - loss: 10.2442 - mse: 10.2442 -
Epoch 16/300
43/43 [==============================] - 0s 2ms/step - loss: 10.1617 - mse: 10.1617 -
Epoch 17/300
43/43 [==============================] - 0s 2ms/step - loss: 10.0864 - mse: 10.0864 -
Epoch 18/300
43/43 [==============================] - 0s 2ms/step - loss: 9.9074 - mse: 9.9074 - v
Epoch 19/300
43/43 [==============================] - 0s 3ms/step - loss: 9.7867 - mse: 9.7867 - v
Epoch 20/300
```

```
43/43 [==============================] - 0s 3ms/step - loss: 9.8023 - mse: 9.8023 - v
Epoch 21/300
43/43 [==============================] - 0s 2ms/step - loss: 9.6899 - mse: 9.6899 - v
Epoch 22/300
43/43 [==============================] - 0s 2ms/step - loss: 9.7063 - mse: 9.7063 - v
Epoch 23/300
43/43 [==============================] - 0s 2ms/step - loss: 9.6098 - mse: 9.6098 - v
Epoch 24/300
43/43 [==============================] - 0s 2ms/step - loss: 9.5988 - mse: 9.5988 - v
Epoch 25/300
43/43 [==============================] - 0s 2ms/step - loss: 9.5932 - mse: 9.5932 - v
Epoch 26/300
43/43 [==============================] - 0s 2ms/step - loss: 9.6751 - mse: 9.6751 - v
Epoch 27/300
43/43 [==============================] - 0s 2ms/step - loss: 9.4795 - mse: 9.4795 - v
Epoch 28/300
43/43 [==============================] - 0s 2ms/step - loss: 9.4316 - mse: 9.4316 - v
Epoch 29/300
```

```
1 plt.plot(history.history['mse'])
2 plt.plot(history.history['loss'])
3 plt.title('model accuracy')
4 plt.ylabel('accuracy')
5 plt.xlabel('epoch')
6 plt.legend(['train', 'test'], loc='upper left')
7 plt.show()
```




```
1 model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 200)               1800

 dense_1 (Dense)             (None, 2)                 402
```

```
=================================================================
Total params: 2,202
Trainable params: 2,202
Non-trainable params: 0
_________________________________________________________________
```

```
1 model_y_pred = model.predict(X_test)
```

```
1 model_y_pred = model.predict(X_test)
2 print('r2_score', r2_score(y_test, model_y_pred))
```

```
r2_score 0.8901006652306644
```