SFML Project

This is the project of **Statistical foundation of machine learning** course at **Vrije Universiteit Brussels**.</br> Team members:</br>

1. **Name:** Ardavan Khalij </br> **Student number:** 0585706
2. **Name:** Mudabbir Faheem Hamza </br> **Student number:** 0574459

1 Algorithms that we are using

1.1 Decision tree

Decision trees contain decision nodes and leaf nodes. The decision nodes contain conditions to split the data. So in each decision node, we make a condition that classifies some of the data. For example, if x=<-2, then the class is equal to 1, so we save this condition based on the data we have and use this condition later for the classification of the other data. So in each step and condition, the pure results will become a leaf node, and the result that is not pure will be another decision node. So, in the end, we have several conditions that define each class, so for testing, the algorithm only checks the conditions and then classifies the data in the class that matches the data. </br> Of course, we should mention that for more complex data structures, it is possible that the algorithm does not find a condition that makes pure data, so it uses the majority class for that area. </br> However, this is not all of it. There can be infinite conditions for our tree, so how the decision tree will find these conditions? This is the central part of the algorithm. The answer to this question lies in the information theory. So the model chooses the conditions that gain maximum data.</br> So how the tree calculates the gain amount of data? It uses entropy to find it based on this formula:

$$Entropy = \sum p(i) \times log(\frac{1}{p(i)})$$

That p(i) is the probability of i th class. So we calculate the information gained with this formula:

$$IG = Entropy(Parent) - \sum (w_i) \times Entropy(Child_i)$$

So the algorithm checks all possible ways and chooses the best one.

1.2 Gaussian naive bayes

1.2.1 Bayes Theorem

It is basically combining two equations. We know that

$$P(A|B) = \frac{P(A \cap B)}{P(B)}$$

and

$$P(B|A) = \frac{P(B \cap A)}{P(A)}$$

and because

$$P(A \cap B) = P(B \cap A)$$

So based on these, we have the Bayes formula

$$P(A|B) = \frac{P(B|A) \times P(A)}{P(B)}$$

1.2.2 Gaussian Naive Bayes

It is used for numerical or continuous features. The distribution of continuous values is assumed to be Gaussian. Moreover, therefore the likelihood probabilities are computed based on Gaussian distribution.</br> So for classification, we should use

$$P(Class|Data) = \frac{P(Data|Class) \times P(Class)}{P(Data)}$$

So based on this formula and the data we already have, it calculates the possibility of data being in each class and chooses the bigger probability. For example, if we want to do this classification on a dataset. First of all, we calculate the probability of being in each class, and then for the Gaussian part, we need to calculate μ and σ for each factor and each class of the data, and then, based on these results, new data can be classified.

1.3 Linear regression

As you know, a linear equation is an equation with this form:

$$y = \beta_0 + \beta_1 x$$

The aim here is to fit a line on our data. This algorithm checks the ε or errors of data based on the line and tries to make these errors minimum, and finally, it chooses the line based on the errors. So our linear model will look like this:

$$y = \beta_0 + \beta_1 x + \varepsilon$$

1.4 Polynomial regression

As you know a polynomial equation is a equation with this form:

$$y = \beta_0 + \beta_1 x + \beta_2 x^2 + \beta_3 x^3 + \ldots + \beta_n x^n$$

The other name for polynomial regression is polynomial linear regression, so the same approach. This algorithm checks the ε or errors of data based on the line and tries to make these errors minimum, and finally, it chooses the curve based on the errors. So our polynomial model will look like this:

$$y = \beta_0 + \beta_1 x + \beta_2 x^2 + \beta_3 x^3 + \ldots + \beta_n x^n + \varepsilon$$

2 Coding, experiments, and scientific questions

2.1 Install libraries

First, we need to install the libraries for our project. You can use this part if you do not have one of these libraries on your computer.

In [356…

```
# pip install numpy
# pip install matplotlib
# pip install pandas
# pip install sklearn
```

2.2 Import Libraries

After we install the libraries, we should import all of the libraries.

In [357…

```
import numpy as np
import matplotlib.pylab as plt
import pandas as pd
from sklearn.naive_bayes import GaussianNB
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix
from sklearn.metrics import precision_score, recall_score, f1_score, accu
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import mean_squared_error
from sklearn.metrics import r2_score
```

2.3 Producing the synthetic data

For this part, we used the pandas library. We Choose the line y=x; if y›x, then y=1, else y=0. So we made this dataset for some experiments on the classification.

2.3.1 Number of Records

We have 5000 records in our synthetic dataset.

In [358…

```
N = 5000
```

2.3.2 Generate data

We start to generate data. Our data is 2D, and numbers are integers between 0 to 250.

In [359…

```
X = np.random.randint(0, 250, (N, 2))
Y1 = np.ones(N)
```

2.3.3 Label the data

We label the data for our experiments. We Choose the line y=x, if y>x, then y=1, else y=0.

In [360…

```
Y1[X[:, 0] <= X[:, 1]] = 0
Y2 = Y1
```

2.3.4 Flip the labels make the data non-separable

For now, we only flip 1% of the data

In [361…

```
for i in range(0, len(Y1)):
    r = np.random.uniform(0, 1)
    if r <= 0.01:
        if Y1[i] == 0:
            Y1[i] = 1
        else:
            Y1[i] = 0
```

2.3.5 Name of the columns

We are setting names for the columns.

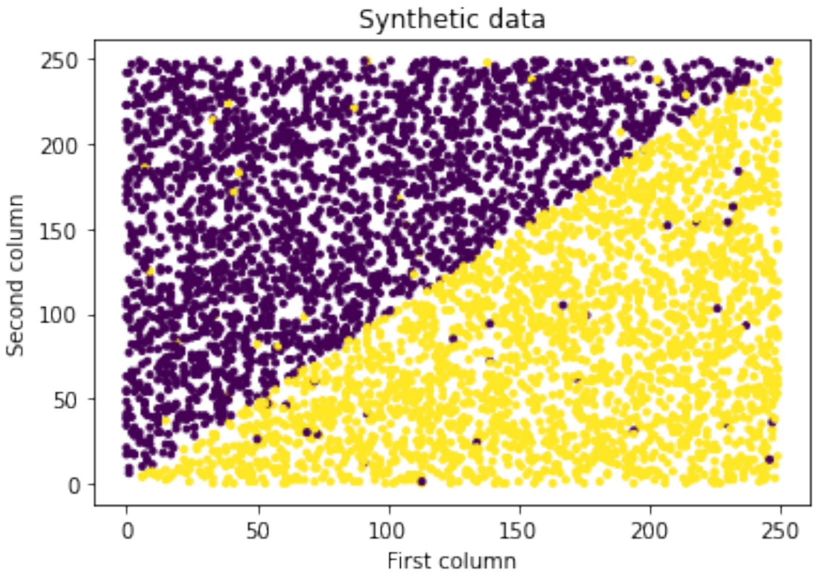
In [362…

```
Synthetic_df_X = pd.DataFrame(X, columns=['First column', 'Second column'
Synthetic_df_Y1 = pd.DataFrame(Y1, columns=['Result'])
```

You can also see the data in a scatter plot.

In [363…

```
fig = plt.figure()
ax = plt.axes()
First_element = Synthetic_df_X['First column']
Second_element = Synthetic_df_X['Second column']
Result = Synthetic_df_Y1['Result']
ax.scatter(First_element, Second_element, c=Result, marker=".")
plt.title("Synthetic data")
plt.xlabel('First column')
plt.ylabel('Second column')
plt.show()
```



We can start our scientific questions now that we have made our synthetic dataset.

3 Scientific question 1: What is the impact of mislabeled training examples on the performance?

So first, we try the classification algorithms that we mentioned before, and we do two experiments and compare both classification algorithms with 1% fliped label, and 10% fliped label data. Train_test_split 80% data for training and keep 20% testing

In [364…
```
X_train1, X_test1, Y_train1, Y_test1 = train_test_split(Synthetic_df_X, S
```

3.1 Experiment 1: impact of mislabeled training examples on the performance of decision tree classification

First we try it with 1% fliped data.

In [365…
```
dt = DecisionTreeClassifier(criterion='entropy', random_state=0)
dt.fit(X_train1, Y_train1)
y_prediction_Decision_Tree_1_percent = dt.predict(X_test1)
```

And then we have the results. The first one is Accuracy:

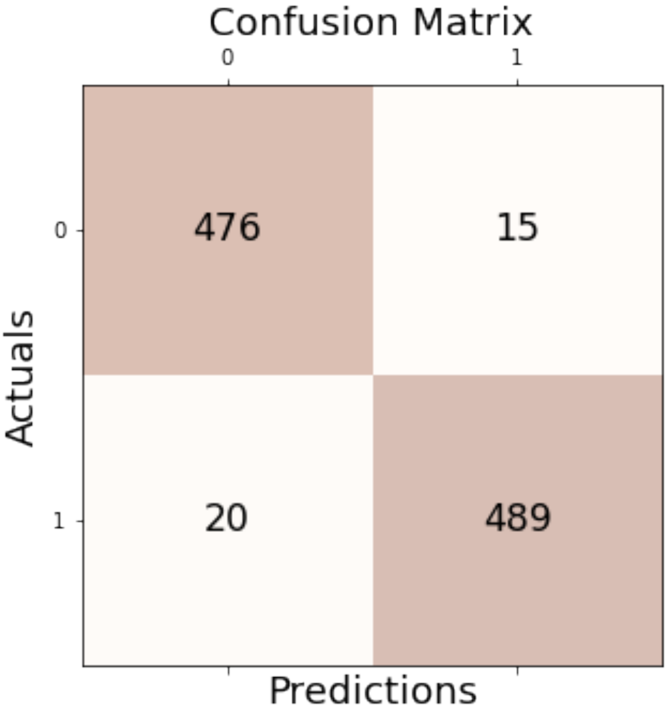
In [366…
```
print("Accuracy: ", str(100*accuracy_score(Y_test1, y_prediction_Decision
```

```
Accuracy:  96.5%
```

Confusion matrices can give us more information about how well our model does for each outcome, but because our data is balanced, this factor is not that important.

In [367…

```
conf_matrix = confusion_matrix(y_true=Y_test1, y_pred=y_prediction_Decisi
fig, ax = plt.subplots(figsize=(5, 5))
ax.matshow(conf_matrix, cmap=plt.cm.Oranges, alpha=0.3)
for i in range(conf_matrix.shape[0]):
    for j in range(conf_matrix.shape[1]):
        ax.text(x=j, y=i,s=conf_matrix[i, j], va='center', ha='center', s

plt.xlabel('Predictions', fontsize=18)
plt.ylabel('Actuals', fontsize=18)
plt.title('Confusion Matrix', fontsize=18)
plt.show()
```



So based on this result, the algorithms' performance is acceptable.

The model accuracy score is the percentage of correctly predicted labels that are right. The positive predictive value is another name for precision. To balance false positives and negatives, precision is utilized in combination with recall. The class distribution has an impact on precision. Precision will be worse if there are more samples in the minority class. Precision may be thought of as a metric for how accurate or good something is. We will use a model with high precision if we wish to reduce false negatives. </br> In contrast, we would pick a model with a high recall if we wanted to reduce false positives. Precision is more useful when predicting the positive class. False positives have a higher price tag than false negatives.

$$Precision = \frac{TP}{FP + TP}$$

In [368…

```
print('Precision: %.3f' % precision_score(Y_test1, y_prediction_Decision_
```

```
Precision: 0.970
```

$$Recall = \frac{TP}{FN + TP}$$

In [369…
```
print('Recall: %.3f' % recall_score(Y_test1, y_prediction_Decision_Tree_1
```

```
Recall: 0.961
```

The model F1 score represents the model score as a function of accuracy and recall. F-score is a machine learning model performance statistic that weighs Precision and Recalls equally when evaluating the accuracy, making it a viable alternative to Accuracy metrics (it does not require us to know the entire number of observations). It is frequently utilized as a single value that conveys high-level information regarding the output quality of the model. This is a valuable model metric in situations where one tries to optimize accuracy or recall score, and the model performance suffers.

$$F1 - Score = \frac{2 \times Precision \times Recall}{Precision + Recall}$$

In [370…
```
print('F1 Score: %.3f' % f1_score(Y_test1, y_prediction_Decision_Tree_1_p
```

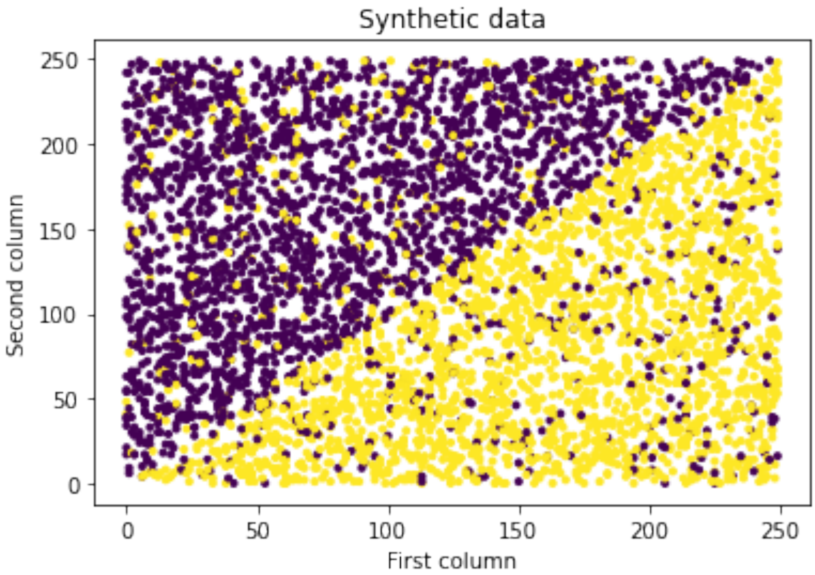
```
F1 Score: 0.965
```

Now we are testing with 10% fliped data.

In [371…
```
for i in range(0, len(Y2)):
    r = np.random.uniform(0, 1)
    if r <= 0.1:
        if Y2[i] == 0:
            Y2[i] = 1
        else:
            Y2[i] = 0
```

In [372…
```
Synthetic_df_X = pd.DataFrame(X, columns=['First column', 'Second column'
Synthetic_df_Y2 = pd.DataFrame(Y2, columns=['Result'])
```

In [373…
```
fig = plt.figure()
ax = plt.axes()
First_element = Synthetic_df_X['First column']
Second_element = Synthetic_df_X['Second column']
Result = Synthetic_df_Y2['Result']
ax.scatter(First_element, Second_element, c=Result, marker=".")
plt.title("Synthetic data")
plt.xlabel('First column')
plt.ylabel('Second column')
plt.show()
```



In [374…

```
X_train2, X_test2, Y_train2, Y_test2 = train_test_split(Synthetic_df_X, S
```

In [375…

```
dt = DecisionTreeClassifier(criterion='entropy', random_state=0)
dt.fit(X_train2, Y_train2)
y_prediction_Decision_Tree_10_percent = dt.predict(X_test2)
```

In [376…

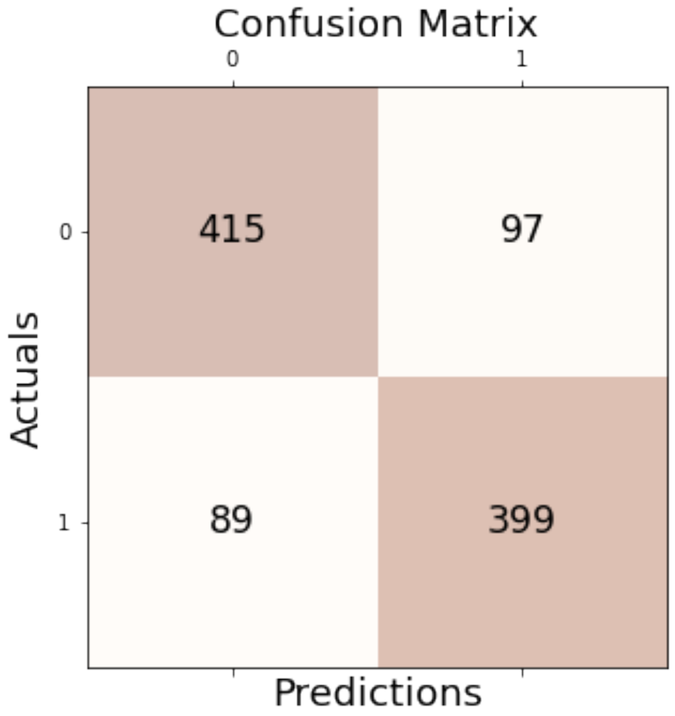
```
print("Accuracy: ", str(100*accuracy_score(Y_test2, y_prediction_Decision
```

```
Accuracy:  81.39999999999999%
```

So the accuracy is reduced by about 20%, and it is clear that we have more data that does not follow the general pattern.

In [377…

```
conf_matrix = confusion_matrix(y_true=Y_test2, y_pred=y_prediction_Decisi
fig, ax = plt.subplots(figsize=(5, 5))
ax.matshow(conf_matrix, cmap=plt.cm.Oranges, alpha=0.3)
for i in range(conf_matrix.shape[0]):
    for j in range(conf_matrix.shape[1]):
        ax.text(x=j, y=i,s=conf_matrix[i, j], va='center', ha='center', s

plt.xlabel('Predictions', fontsize=18)
plt.ylabel('Actuals', fontsize=18)
plt.title('Confusion Matrix', fontsize=18)
plt.show()
```



However, we can see that the correct predictions in 0 and 1 are still balanced, and this is because the data is balanced.

In [378…

```
print('Precision: %.3f' % precision_score(Y_test2, y_prediction_Decision_
print('Recall: %.3f' % recall_score(Y_test2, y_prediction_Decision_Tree_1
print('F1 Score: %.3f' % f1_score(Y_test2, y_prediction_Decision_Tree_10_
```

```
Precision: 0.804
Recall: 0.818
F1 Score: 0.811
```

Again we can see a reduction of about 20% in all of the results, all because of the balance of the data.

3.2 Experiment 2: impact of mislabeled training examples on the performance of Gaussian Naive Bayes classification

So we repeat the exact experiment on Gaussian Naive Bayes as well.

In [379…

```
nb = GaussianNB()
nb.fit(X_train1, Y_train1)
y_prediction_Naive_Bayes = nb.predict(X_test1)
```

```
/usr/local/lib/python3.9/site-packages/sklearn/utils/validation.py:1111:
DataConversionWarning: A column-vector y was passed when a 1d array was e
xpected. Please change the shape of y to (n_samples, ), for example using
ravel().
  y = column_or_1d(y, warn=True)
```

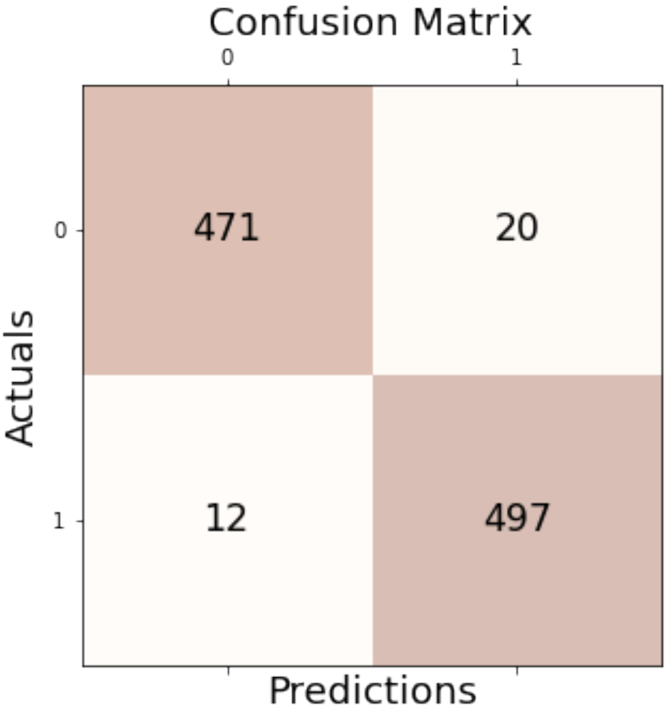
In [380…

```
print("Accuracy: ", str(100*accuracy_score(Y_test1, y_prediction_Naive_Ba
```

```
Accuracy:  96.8%
```

In [381…

```
conf_matrix = confusion_matrix(y_true=Y_test1, y_pred=y_prediction_Naive_
fig, ax = plt.subplots(figsize=(5, 5))
ax.matshow(conf_matrix, cmap=plt.cm.Oranges, alpha=0.3)
for i in range(conf_matrix.shape[0]):
    for j in range(conf_matrix.shape[1]):
        ax.text(x=j, y=i,s=conf_matrix[i, j], va='center', ha='center', s

plt.xlabel('Predictions', fontsize=18)
plt.ylabel('Actuals', fontsize=18)
plt.title('Confusion Matrix', fontsize=18)
plt.show()
```



In [382…

```
print('Precision: %.3f' % precision_score(Y_test1, y_prediction_Naive_Bay
print('Recall: %.3f' % recall_score(Y_test1, y_prediction_Naive_Bayes))
print('F1 Score: %.3f' % f1_score(Y_test1, y_prediction_Naive_Bayes))
```

```
Precision: 0.961
Recall: 0.976
F1 Score: 0.969
```

So we have the results for the 1 percent fliped dat in Gaussian Naive Bayes

In [383…

```
nb = GaussianNB()
nb.fit(X_train1, Y_train1)
y_prediction_Naive_Bayes_2 = nb.predict(X_test2)
```

```
/usr/local/lib/python3.9/site-packages/sklearn/utils/validation.py:1111:
DataConversionWarning: A column-vector y was passed when a 1d array was e
xpected. Please change the shape of y to (n_samples, ), for example using
ravel().
  y = column_or_1d(y, warn=True)
```

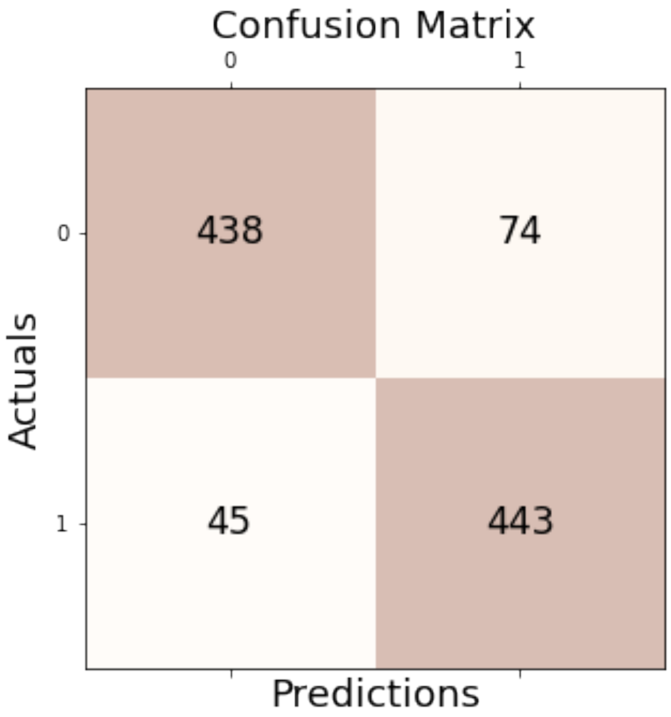
In [384…

```
print("Accuracy: ", str(100*accuracy_score(Y_test2, y_prediction_Naive_Ba
```

```
Accuracy:  88.1%
```

In [385…

```
conf_matrix = confusion_matrix(y_true=Y_test2, y_pred=y_prediction_Naive_
fig, ax = plt.subplots(figsize=(5, 5))
ax.matshow(conf_matrix, cmap=plt.cm.Oranges, alpha=0.3)
for i in range(conf_matrix.shape[0]):
    for j in range(conf_matrix.shape[1]):
        ax.text(x=j, y=i,s=conf_matrix[i, j], va='center', ha='center', s

plt.xlabel('Predictions', fontsize=18)
plt.ylabel('Actuals', fontsize=18)
plt.title('Confusion Matrix', fontsize=18)
plt.show()
```



In [386…

```
print('Precision: %.3f' % precision_score(Y_test2, y_prediction_Naive_Bay
print('Recall: %.3f' % recall_score(Y_test2, y_prediction_Naive_Bayes_2))
print('F1 Score: %.3f' % f1_score(Y_test2, y_prediction_Naive_Bayes_2))
```

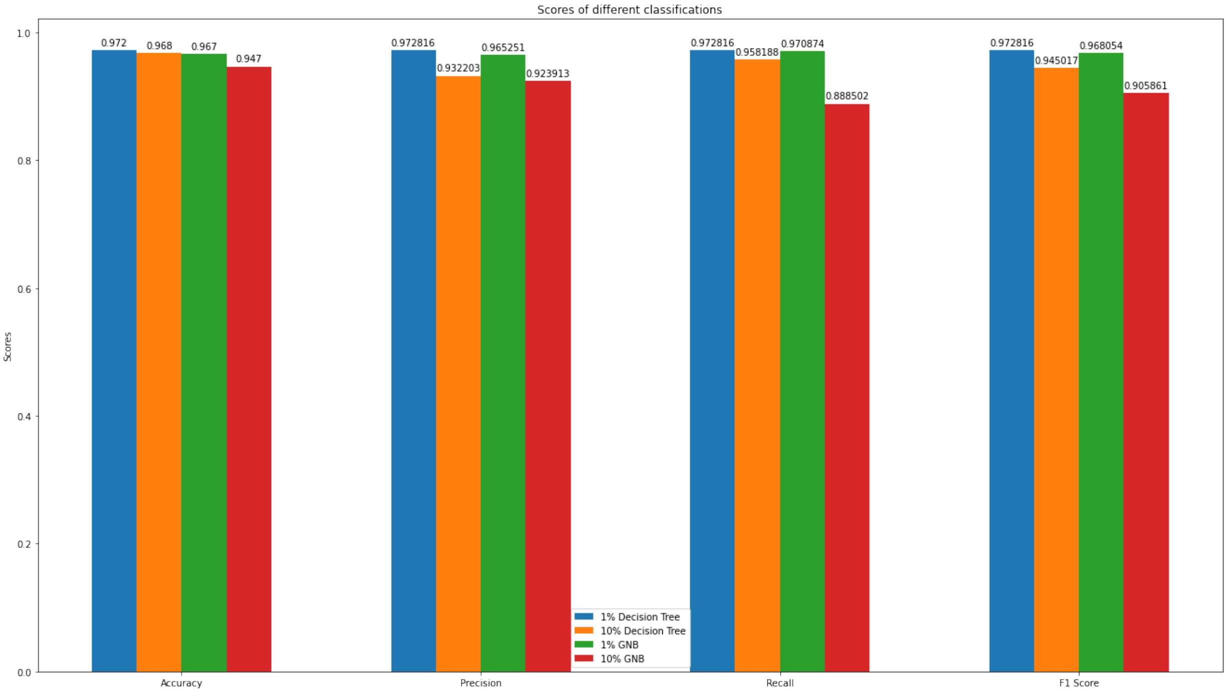
```
Precision: 0.857
Recall: 0.908
F1 Score: 0.882
```

3.3 Conclusion

So as you can see, the impact of flipped data is more on the decision tree. Moreover, That makes sense because, as we mentioned in the first of the report, the decision tree algorithm works based on the conditions, so this kind of fliped data can significantly impact it. However, the gaussian naive Bayes classification only checks the probability. It does not depend on some conditions and areas in the graph, so it costs less accuracy and other factors. So overall, we can conclude that it is better to use gaussian naive Bayes for data with more fliped records than a decision tree, but if the data is not flipped, there is no significant difference. </br> You can also see the graph of the final results in a bar chart.

In [387…

```
labels = ['Accuracy', 'Precision', 'Recall', 'F1 Score']

OnePercentDecisionTree = [
    accuracy_score(Y_test1, y_prediction_Decision_Tree_1_percent),
    precision_score(Y_test1, y_prediction_Decision_Tree_1_percent),
    recall_score(Y_test1, y_prediction_Decision_Tree_1_percent),
    f1_score(Y_test1, y_prediction_Decision_Tree_1_percent)
]
TenPercentDecisionTree = [
    accuracy_score(Y_test2, y_prediction_Decision_Tree_10_percent),
    precision_score(Y_test2, y_prediction_Decision_Tree_10_percent),
    recall_score(Y_test2, y_prediction_Decision_Tree_10_percent),
    f1_score(Y_test2, y_prediction_Decision_Tree_10_percent)
]
OnePercentGNB = [
    accuracy_score(Y_test1, y_prediction_Naive_Bayes),
    precision_score(Y_test1, y_prediction_Naive_Bayes),
    recall_score(Y_test1, y_prediction_Naive_Bayes),
    f1_score(Y_test1, y_prediction_Naive_Bayes)
]
TenPercentGNB = [
    accuracy_score(Y_test2, y_prediction_Naive_Bayes_2),
    precision_score(Y_test2, y_prediction_Naive_Bayes_2),
    recall_score(Y_test2, y_prediction_Naive_Bayes_2),
    f1_score(Y_test2, y_prediction_Naive_Bayes_2)
]

x = np.arange(len(labels))  # the label locations
width = 0.15  # the width of the bars

fig, ax = plt.subplots()
rects1 = ax.bar(x - 1.5*width, BalancedDecisionTree, width, label='1% Dec
rects2 = ax.bar(x - 0.5*width, UnbalancedDecisionTree, width, label='10%
rects3 = ax.bar(x + 0.5*width, BalancedGNB, width, label='1% GNB')
rects4 = ax.bar(x + 1.5*width, UnbalancedGNB, width, label='10% GNB')

fig.set_size_inches(18.5, 10.5)

# Add some text for labels, title and custom x-axis tick labels, etc.
ax.set_ylabel('Scores')
ax.set_title('Scores of different classifications')
ax.set_xticks(x, labels)
ax.legend()

ax.bar_label(rects1, padding=3)
ax.bar_label(rects2, padding=3)
ax.bar_label(rects3, padding=3)
ax.bar_label(rects4, padding=3)

fig.tight_layout()

plt.show()
```



4 Scientific question 2: What is the impact of class unbalances on the performance of classification algorithms?

In this part, we also have two other experiments to check the effect of unbalance data on the prediction of these two classification algorithms.

4.1 Producing the synthetic data

For this part, we used the pandas library. We Choose the line 2y=x; if 2y>x, then y=1, else y=0. So we made this dataset for some experiments on the classification.

4.1.1 Number of Records

We have 5000 records in our synthetic dataset.

In [388...

```
N = 5000
```

4.1.2 Generate data

We start to generate data. Our data is 2D, and numbers are integers between 0 to 250.

In [389...

```
X = np.random.randint(0, 250, (N, 2))
Y = np.ones(N)
Y2 = Y
```

4.1.3 Label the data

We label the data for our experiments. We Choose the line 2y=x, if 2y›x, then y=1, else y=0.

In [390…

```
Y[X[:, 0] <= 2 * X[:, 1]] = 0
```

4.1.4 Flip the labels make the data non-separable

For now, we only flip 1% of the data

In [391…

```
for i in range(0, len(Y)):
    r = np.random.uniform(0, 1)
    if r <= 0.01:
        if Y[i] == 0:
            Y[i] = 1
        else:
            Y[i] = 0
```

4.1.5 Name of the columns

We are setting names for the columns.

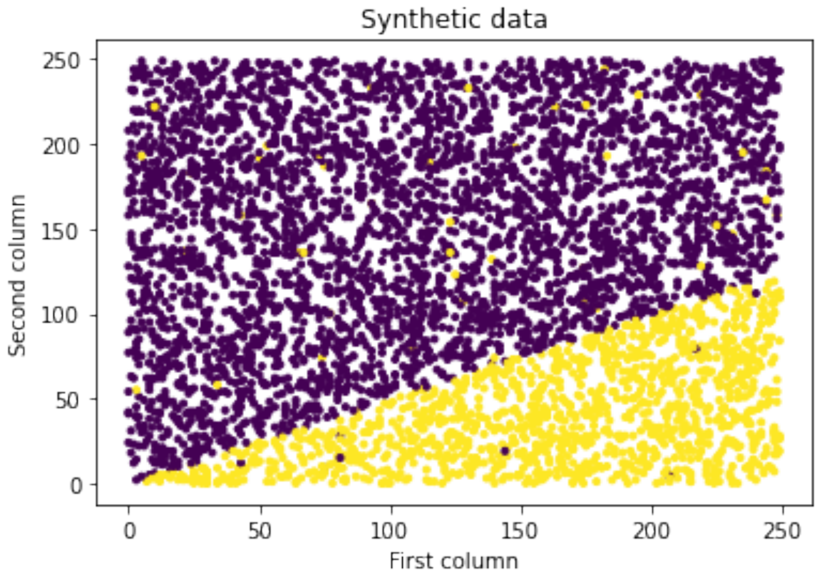
In [392…

```
Synthetic_df_X = pd.DataFrame(X, columns=['First column', 'Second column'
Synthetic_df_Y1 = pd.DataFrame(Y, columns=['Result'])
```

You can also see the data in a scatter plot.

In [393…

```
fig = plt.figure()
ax = plt.axes()
First_element = Synthetic_df_X['First column']
Second_element = Synthetic_df_X['Second column']
Result = Synthetic_df_Y1['Result']
ax.scatter(First_element, Second_element, c=Result, marker=".")
plt.title("Synthetic data")
plt.xlabel('First column')
plt.ylabel('Second column')
plt.show()
```



train_test_split 80% data for training and keep 20% testing

In [394...

```
X_train, X_test, Y_train, Y_test = train_test_split(Synthetic_df_X, Synth
```

So we have unbalance in our data.

4.2 Experiment 3: impact of unbalanced training examples on the performance of decision tree classification

So we start with the unbalanced data. Based on what we explained, 33.3% of data was labeled as 0 and 66.6% labeled as 1. So now we try a decision tree classifier for it.

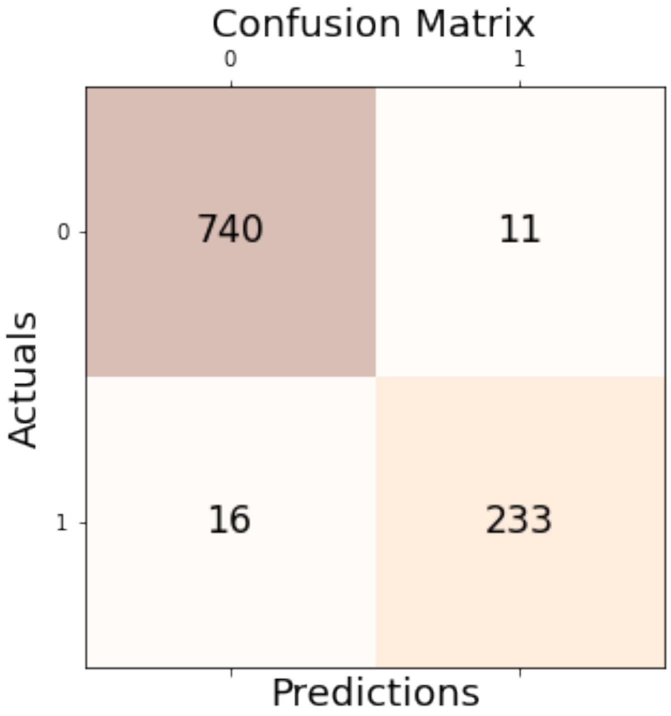
In [395...

```
dt = DecisionTreeClassifier(criterion='entropy', random_state=0)
dt.fit(X_train, Y_train)
y_prediction_Decision_Tree = dt.predict(X_test)
```

And we have the results here:

In [396...

```
conf_matrix = confusion_matrix(y_true=Y_test, y_pred=y_prediction_Decisio
fig, ax = plt.subplots(figsize=(5, 5))
ax.matshow(conf_matrix, cmap=plt.cm.Oranges, alpha=0.3)
for i in range(conf_matrix.shape[0]):
    for j in range(conf_matrix.shape[1]):
        ax.text(x=j, y=i,s=conf_matrix[i, j], va='center', ha='center', s

plt.xlabel('Predictions', fontsize=18)
plt.ylabel('Actuals', fontsize=18)
plt.title('Confusion Matrix', fontsize=18)
plt.show()
print("Accuracy: ", str(100*accuracy_score(Y_test, y_prediction_Decision_
print('Precision: %.3f' % precision_score(Y_test, y_prediction_Decision_T
print('Recall: %.3f' % recall_score(Y_test, y_prediction_Decision_Tree))
print('F1 Score: %.3f' % f1_score(Y_test, y_prediction_Decision_Tree))
```

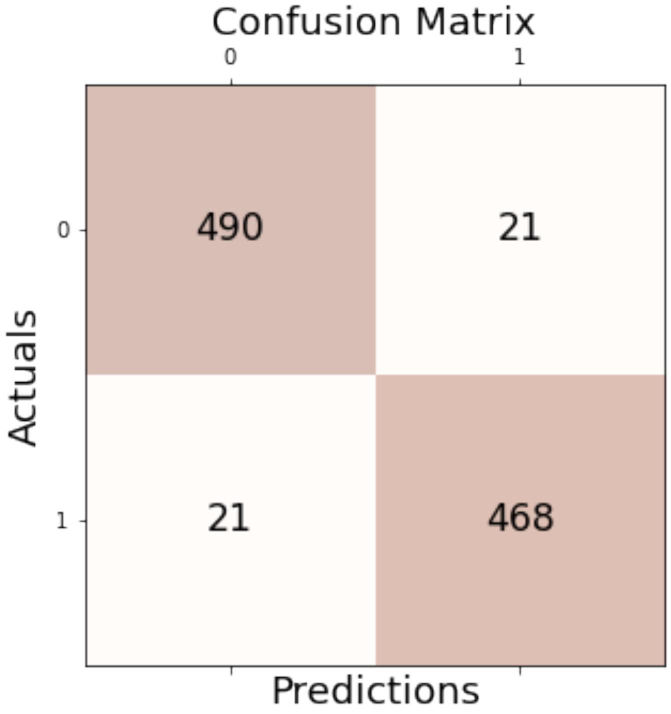


```
Accuracy:  97.3%
Precision: 0.955
Recall: 0.936
F1 Score: 0.945
```

Now we do this experiment again on a balanced data:

In [397…

```
# Label the data
Y2 = np.ones(N)
Y2[X[:, 0] <= X[:, 1]] = 0

# Flip the labels make the data non-separable
for i in range(0, len(Y2)):
    r = np.random.uniform(0, 1)
    if r <= 0.01:
        if Y2[i] == 0:
            Y2[i] = 1
        else:
            Y2[i] = 0

# Name of the columns
Synthetic_df_X = pd.DataFrame(X, columns=['First column', 'Second column'
Synthetic_df_Y2 = pd.DataFrame(Y2, columns=['Result'])

# train_test_split 80% data for training and keep 20% testing
X_train22, X_test22, Y_train22, Y_test22 = train_test_split(Synthetic_df_

# Decision Tree classification
dt = DecisionTreeClassifier(criterion='entropy', random_state=0)
dt.fit(X_train22, Y_train22)
y_prediction_Decision_Tree2 = dt.predict(X_test22)

# Result
conf_matrix = confusion_matrix(y_true=Y_test22, y_pred=y_prediction_Decis
fig, ax = plt.subplots(figsize=(5, 5))
ax.matshow(conf_matrix, cmap=plt.cm.Oranges, alpha=0.3)
for i in range(conf_matrix.shape[0]):
    for j in range(conf_matrix.shape[1]):
        ax.text(x=j, y=i,s=conf_matrix[i, j], va='center', ha='center', s
plt.xlabel('Predictions', fontsize=18)
plt.ylabel('Actuals', fontsize=18)
plt.title('Confusion Matrix', fontsize=18)
plt.show()

print("Accuracy: ", str(100*accuracy_score(Y_test22, y_prediction_Decisio
print('Precision: %.3f' % precision_score(Y_test22, y_prediction_Decision
print('Recall: %.3f' % recall_score(Y_test22, y_prediction_Decision_Tree2
print('F1 Score: %.3f' % f1_score(Y_test22, y_prediction_Decision_Tree2))
```



```
Accuracy:  95.8%
Precision: 0.957
Recall: 0.957
F1 Score: 0.957
```

So as you can see, the overall accuracy has not got a significant change, and they are almost the same. However, Precision, Recall, and F1 Score can show us that the accuracy of predictions for the 0 class (class with less data) changed significantly. If we compare the confusion matrix of these two states, it is visible that the number of wrong predictions in class 0 is equal to class 1 in the unbalanced data. However, the number of class 0 data is half the number of class 1 data and based on these factors, we can say that unbalanced data helps the accuracy of prediction for the class that has the majority and reduce the accuracy of prediction for the class with fewer data.

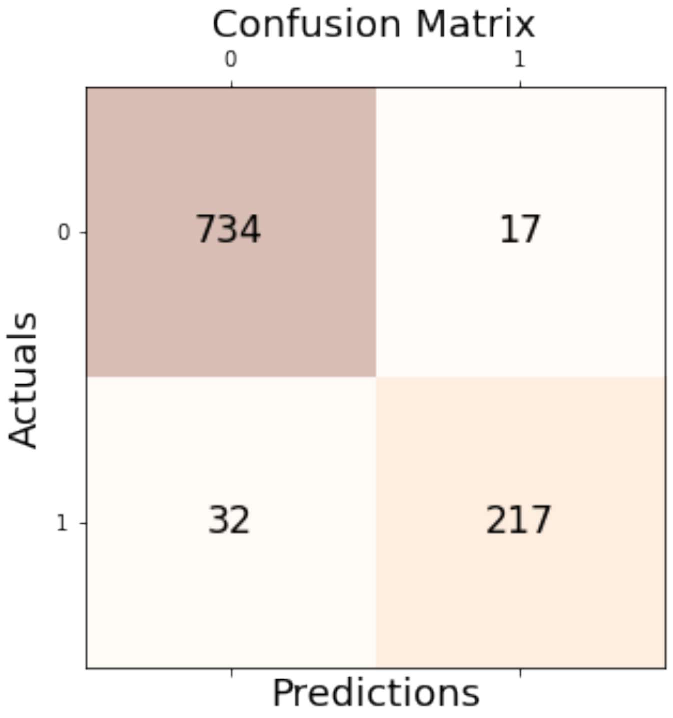
4.3 Experiment 4: impact of unbalanced training examples on the performance of Gaussian Naive Bayes classification

So we start with the unbalanced data. Based on what we explained, 33.3% of data was labeled as 0 and 66.6% labeled as 1. So now we try a decision tree classifier for it.

In [398…

```
# Naive Bayes classification
nb = GaussianNB()
nb.fit(X_train, Y_train)
y_prediction_Naive_Bayes11 = nb.predict(X_test)

# Result
conf_matrix = confusion_matrix(y_true=Y_test, y_pred=y_prediction_Naive_B
fig, ax = plt.subplots(figsize=(5, 5))
ax.matshow(conf_matrix, cmap=plt.cm.Oranges, alpha=0.3)
for i in range(conf_matrix.shape[0]):
    for j in range(conf_matrix.shape[1]):
        ax.text(x=j, y=i,s=conf_matrix[i, j], va='center', ha='center', s
plt.xlabel('Predictions', fontsize=18)
plt.ylabel('Actuals', fontsize=18)
plt.title('Confusion Matrix', fontsize=18)
plt.show()

print("Accuracy: ", str(100*accuracy_score(Y_test, y_prediction_Naive_Bay
print('Precision: %.3f' % precision_score(Y_test, y_prediction_Naive_Baye
print('Recall: %.3f' % recall_score(Y_test, y_prediction_Naive_Bayes11))
print('F1 Score: %.3f' % f1_score(Y_test, y_prediction_Naive_Bayes11))
```

```
/usr/local/lib/python3.9/site-packages/sklearn/utils/validation.py:1111:
DataConversionWarning: A column-vector y was passed when a 1d array was e
xpected. Please change the shape of y to (n_samples, ), for example using
ravel().
  y = column_or_1d(y, warn=True)
```



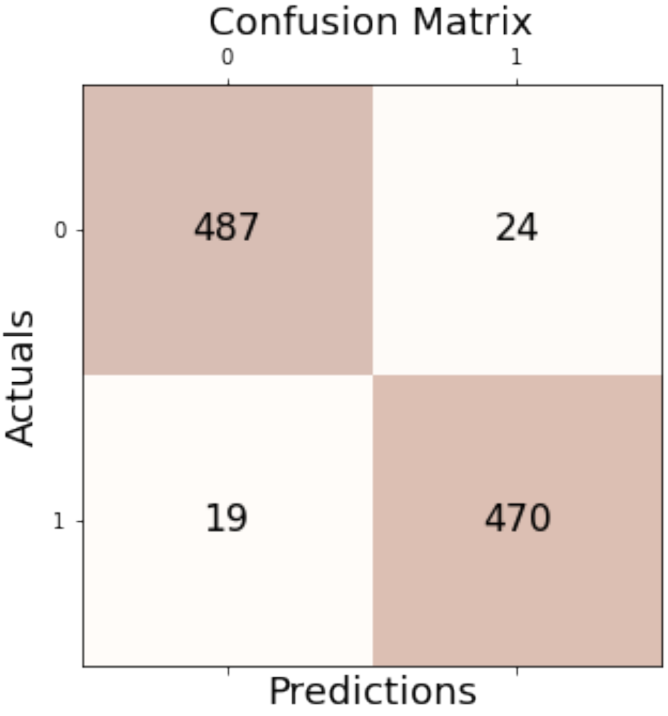
```
Accuracy:  95.1%
Precision: 0.927
Recall: 0.871
F1 Score: 0.899
```

Now we do this experiment again on a balanced data:

In [399…

```
# Naive Bayes classification
nb = GaussianNB()
nb.fit(X_train2, Y_train2)
y_prediction_Naive_Bayes22 = nb.predict(X_test22)

# Result
conf_matrix = confusion_matrix(y_true=Y_test22, y_pred=y_prediction_Naive
fig, ax = plt.subplots(figsize=(5, 5))
ax.matshow(conf_matrix, cmap=plt.cm.Oranges, alpha=0.3)
for i in range(conf_matrix.shape[0]):
    for j in range(conf_matrix.shape[1]):
        ax.text(x=j, y=i,s=conf_matrix[i, j], va='center', ha='center', s
plt.xlabel('Predictions', fontsize=18)
plt.ylabel('Actuals', fontsize=18)
plt.title('Confusion Matrix', fontsize=18)
plt.show()

print("Accuracy: ", str(100*accuracy_score(Y_test22, y_prediction_Naive_B
print('Precision: %.3f' % precision_score(Y_test22, y_prediction_Naive_Ba
print('Recall: %.3f' % recall_score(Y_test22, y_prediction_Naive_Bayes22)
print('F1 Score: %.3f' % f1_score(Y_test22, y_prediction_Naive_Bayes22))
```

```
/usr/local/lib/python3.9/site-packages/sklearn/utils/validation.py:1111:
DataConversionWarning: A column-vector y was passed when a 1d array was e
xpected. Please change the shape of y to (n_samples, ), for example using
ravel().
  y = column_or_1d(y, warn=True)
```



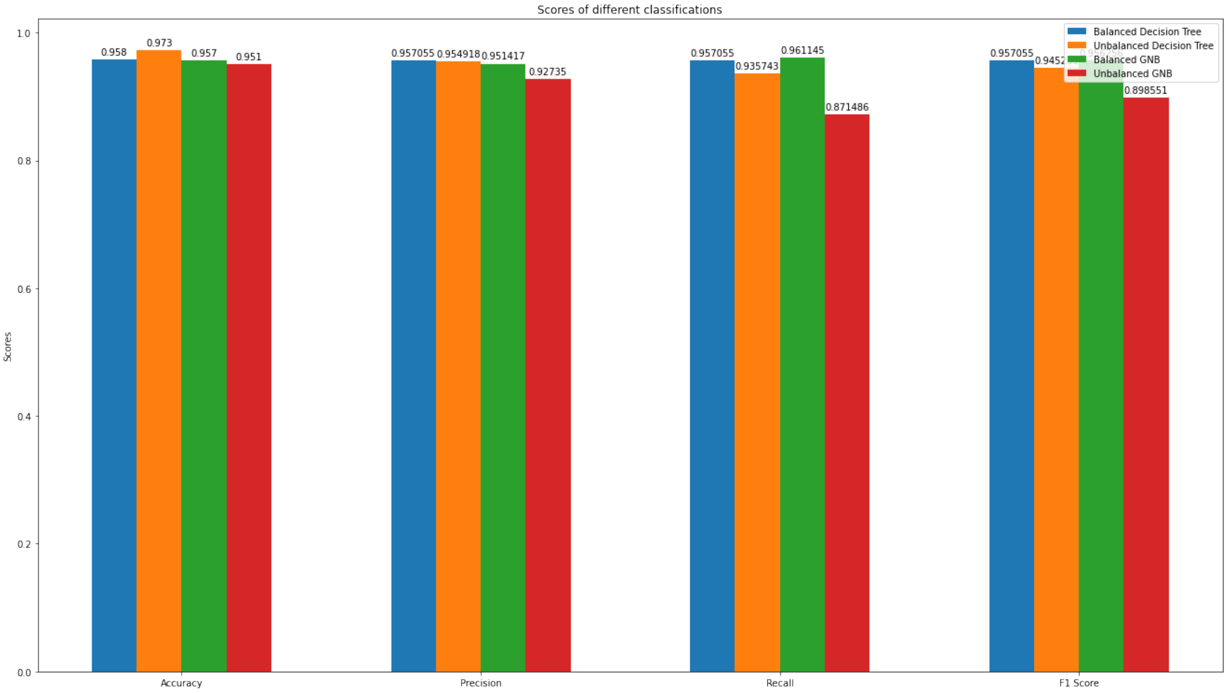
```
Accuracy:  95.7%
Precision: 0.951
Recall: 0.961
F1 Score: 0.956
```

4.4 Conclusion

So when we do this experiment, we can see that unbalanced data has more effect on Gaussian Naive Bayes than the decision tree classifier. Moreover, this makes sense because Gaussian Naive Bayes depends more on the number of data. In decision tree classification, the number of data matters for finding the conditions, but there is no need for the number of data when the conditions are found.</br> You can also see the graph of the final results in a bar chart.

In [400…

```
labels = ['Accuracy', 'Precision', 'Recall', 'F1 Score']

BalancedDecisionTree = [
    accuracy_score(Y_test22, y_prediction_Decision_Tree2),
    precision_score(Y_test22, y_prediction_Decision_Tree2),
    recall_score(Y_test22, y_prediction_Decision_Tree2),
    f1_score(Y_test22, y_prediction_Decision_Tree2)
]
UnbalancedDecisionTree = [
    accuracy_score(Y_test, y_prediction_Decision_Tree),
    precision_score(Y_test, y_prediction_Decision_Tree),
    recall_score(Y_test, y_prediction_Decision_Tree),
    f1_score(Y_test, y_prediction_Decision_Tree)
]
BalancedGNB = [
    accuracy_score(Y_test22, y_prediction_Naive_Bayes22),
    precision_score(Y_test22, y_prediction_Naive_Bayes22),
    recall_score(Y_test22, y_prediction_Naive_Bayes22),
    f1_score(Y_test22, y_prediction_Naive_Bayes22)
]
UnbalancedGNB = [
    accuracy_score(Y_test, y_prediction_Naive_Bayes11),
    precision_score(Y_test, y_prediction_Naive_Bayes11),
    recall_score(Y_test, y_prediction_Naive_Bayes11),
    f1_score(Y_test, y_prediction_Naive_Bayes11)
]

x = np.arange(len(labels))  # the label locations
width = 0.15  # the width of the bars

fig, ax = plt.subplots()
rects1 = ax.bar(x - 1.5*width, BalancedDecisionTree, width, label='Balanc
rects2 = ax.bar(x - 0.5*width, UnbalancedDecisionTree, width, label='Unba
rects3 = ax.bar(x + 0.5*width, BalancedGNB, width, label='Balanced GNB')
rects4 = ax.bar(x + 1.5*width, UnbalancedGNB, width, label='Unbalanced GN

fig.set_size_inches(18.5, 10.5)

# Add some text for labels, title and custom x-axis tick labels, etc.
ax.set_ylabel('Scores')
ax.set_title('Scores of different classifications')
ax.set_xticks(x, labels)
ax.legend()

ax.bar_label(rects1, padding=3)
ax.bar_label(rects2, padding=3)
ax.bar_label(rects3, padding=3)
ax.bar_label(rects4, padding=3)

fig.tight_layout()

plt.show()
```



5 Scientific question 3: Which regression can be a better regression for the real dataset?

Before starting this question, we will introduce the dataset we used. This dataset is from kaggle.com, and it is data on TV, Influencer, Radio, and Social Media advertisement budgets to predict Sales. For this experiment, we only use the Radio budget, and the aim is to find out the relation between the radio advertisement budget and sales. We measure the error of our two regression algorithms and decide which one is the better algorithm.</br> At first, we show the data of radio advertisements based on the sales, but we should add the data before that.

In [411...

```
Data = pd.read_csv('https://drive.google.com/uc?export=download&id=1ZRNoD
```

Delete the first row because they are name.

In [412...

```
Data = Data.iloc[1:, :]
```

Rename the columns

In [413...

```
column_names = {0: 'TV', 1: 'Radio', 2: 'Social_Media', 3: 'Influencer',
Data.rename(columns=column_names, inplace=True)
```

Drop nan rows in Radio column

In [414...

```
Data.dropna(subset=["Radio"], inplace=True)
```

Drop nan rows in Sales column

In [415...

```
Data.dropna(subset=["Sales"], inplace=True)
```

Separate X and Y

In [416…

```
predictors = ['Radio']
outcome = ['Sales']
X = Data[predictors]
Y = Data[outcome]
```

train_test_split 80% data for training and keep 20% testing

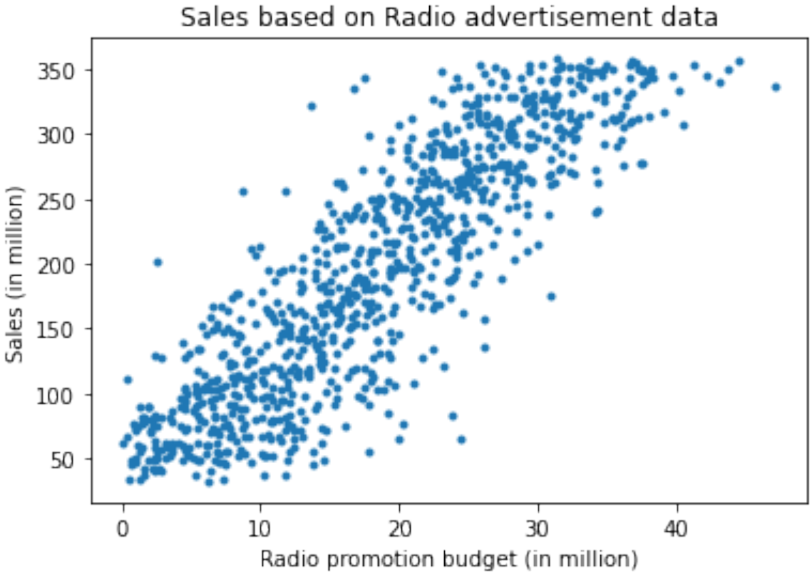
In [417…

```
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2,
```

Figure of Radio advertisement cost and the sales

In [419…

```
a1 = X_test['Radio'].tolist()
a2 = Y_test['Sales'].tolist()
Radio = [float(x) for x in a1]
Sales = [float(x) for x in a2]
plt.plot(Radio, Sales, '.')
plt.title("Sales based on Radio advertisement data")
plt.xlabel('Radio promotion budget (in million)')
plt.ylabel('Sales (in million)')
plt.show()
```



Linear regression

In [420…

```
linReg = LinearRegression()
model_lin = linReg.fit(X_train, Y_train)
y_prediction_lin = linReg.predict(X_test)
```

Now we can evaluate the linear regression. We check the mean absolute error, mean squared error, and coefficient of determination (R2 score).

5.1 The mean absolute error

The mean absolute error (MAE) is one of the most frequent metrics for calculating the model's prediction error. A single row of data has a prediction error of:

$$PredictionError = ActualValue - Predictedalue$$

For each row of data, we must compute prediction errors, obtain their absolute value, and then find the mean of all absolute prediction errors. </br> The formula for MAE is as follows:

$$MAE = \frac{1}{N}\sum|y_i - \hat{y_i}|$$

Because MAE employs the absolute value of the residuals, it cannot tell if the model is doing well or poorly. Because we are adding separate residuals, each adds linearly to the overall error. As a result, a low MAE indicates that the model is good at forecasting. On the other hand, a big MAE indicates that your model may have difficulty generalizing. Our approach produces flawless predictions with an MAE of 0, although this is unlikely to happen in real-world settings.</br> Because it does not reflect huge residuals, MAE is the best statistic for distinguishing between various models.

5.2 The mean squared error

The mean squared difference between the target and forecasted values is used to calculate the mean squared error (MSE). More significant mistakes have proportionately larger squared contributions to the mean error. Therefore this value is extensively employed for many regression situations. </br> MSE is calculated using the following formula:

$$MSE = \frac{1}{N}\sum(y_i - \hat{y_i})^2$$

Because in MAE, residuals contribute linearly to the overall error, but in MSE, the error rises quadratically with each residual, MSE will almost always be larger than MAE. Because MSE heavily penalizes the heavy outliers, it is used to determine how the model fits the data.

5.3 The coefficient of determination (R2 score)

The R2 value indicates how well the regression predictions match the actual data points. </br> The following formula is used to compute the value of R2:

$$R^2 = 1 - \frac{\sum(y_i - \hat{y_i})^2}{\sum(y_i - \bar{y_i})^2}$$

where i is the projected value of yi and is the mean of the observed data, computed as

$$\bar{y} = \sum y_i$$

R2 has a range of values ranging from 0 to 1. The regression predictions exactly fit the data if the value is 1.

In [421...

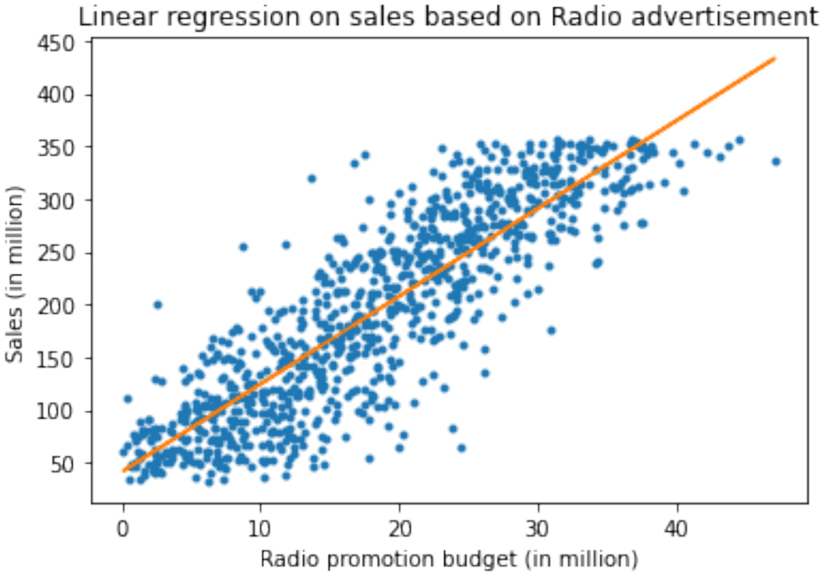
```
mae = mean_absolute_error(Y_test, y_prediction_lin)
mse = mean_squared_error(Y_test, y_prediction_lin)
r2 = r2_score(Y_test, y_prediction_lin)

print('MAE is {}'.format(mae))
print('MSE is {}'.format(mse))
print('R2 score is {}'.format(r2))
```

```
MAE is 35.9082260163343
MSE is 2111.4669534348354
R2 score is 0.7592140564975793
```

Figure Radio advertisement cost and sales for linear regression.

In [422...

```
a1 = X_test['Radio'].tolist()
a2 = Y_test['Sales'].tolist()
Radio = [float(x) for x in a1]
Sales = [float(x) for x in a2]
m, b = np.polyfit(Radio, Sales, 1)
plt.plot(Radio, Sales, '.')
Radio_Y = [x * m + b for x in Radio]
plt.plot(Radio, Radio_Y)
plt.title("Linear regression on sales based on Radio advertisement")
plt.xlabel('Radio promotion budget (in million)')
plt.ylabel('Sales (in million)')
plt.show()
```



Moreover, now we repeat it for polynomial linear regression.

In [423...

```
lin_reg = LinearRegression()
lin_reg.fit(X_train, Y_train)
poly_reg = PolynomialFeatures(degree=5)
X_poly = poly_reg.fit_transform(X_train)
poly_reg.fit(X_poly, Y_train)
lin_reg2 = LinearRegression()
lin_reg2.fit(X_poly, Y_train)
y_prediction_poly = lin_reg2.predict(poly_reg.fit_transform(X_test))
```

And you can see the evaluation of it here

In [424…

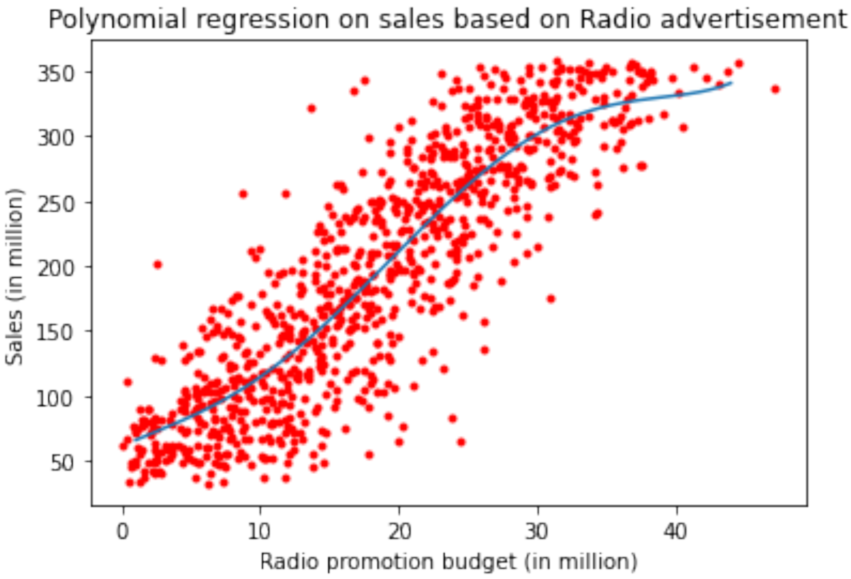
```
mae = mean_absolute_error(Y_test, y_prediction_poly)
mse = mean_squared_error(Y_test, y_prediction_poly)
r2 = r2_score(Y_test, y_prediction_poly)

print('MAE is {}'.format(mae))
print('MSE is {}'.format(mse))
print('R2 score is {}'.format(r2))
```

```
MAE is 34.32115945730862
MSE is 1985.2979983086886
R2 score is 0.7736020206811265
```

Figure Radio advertisement cost and sales for polynomial linear regression.

In [425…

```
a1 = X_test['Radio'].tolist()
a2 = Y_test['Sales'].tolist()
Radio = [float(x) for x in a1]
Sales = [float(x) for x in a2]
myModel = np.poly1d(np.polyfit(Radio, Sales, 5))
myLine = np.linspace(1, 44, 360)
plt.plot(Radio, Sales, '.', color="red")
plt.plot(myLine, myModel(myLine))
plt.title("Polynomial regression on sales based on Radio advertisement")
plt.xlabel('Radio promotion budget (in million)')
plt.ylabel('Sales (in million)')
plt.show()
```

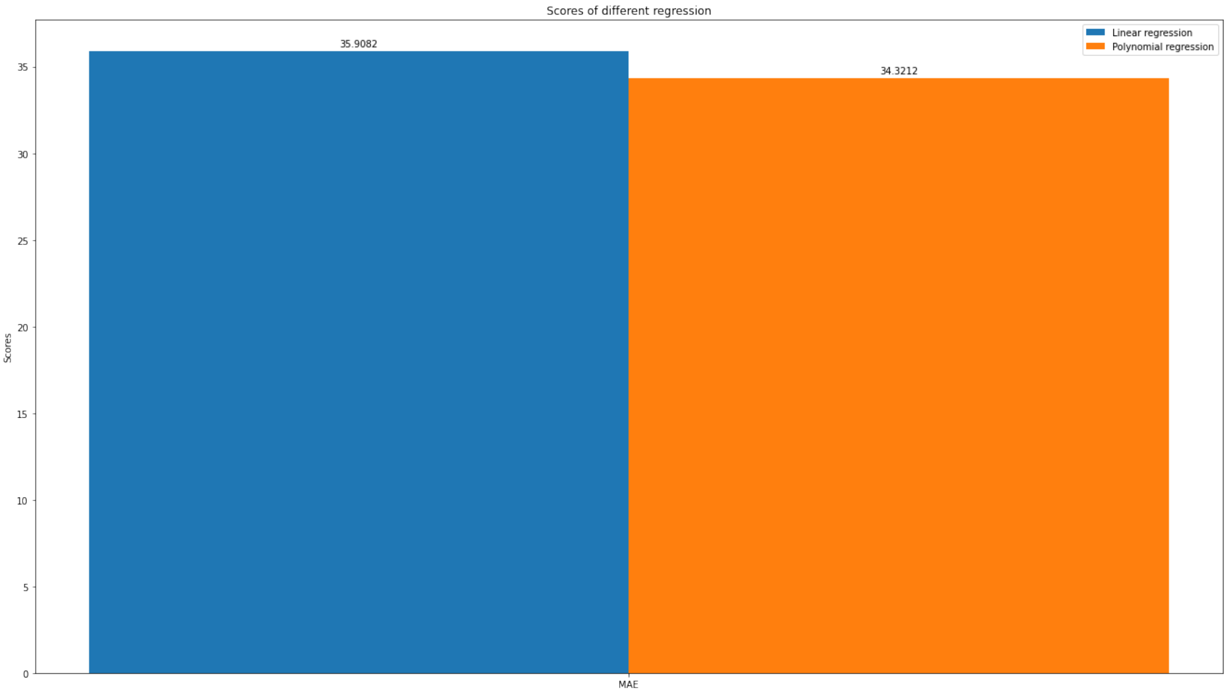


5.4 Conclusion

And then we can compare these results:

In [426…

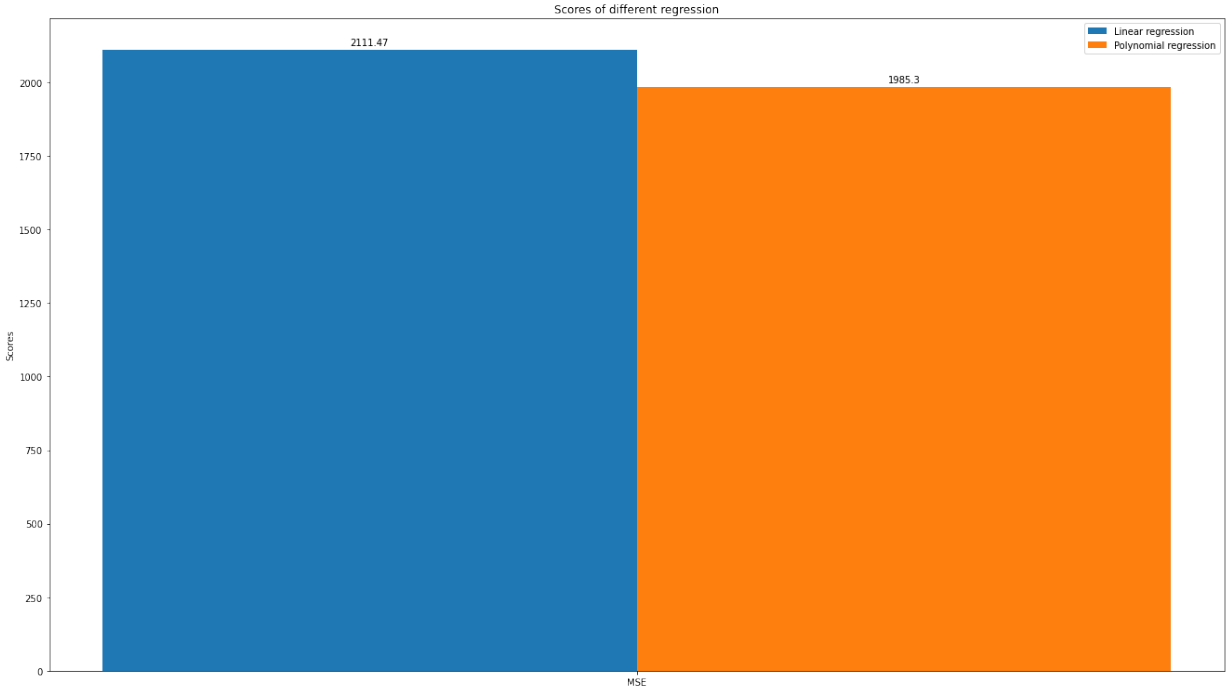
```
labels = ['MAE']

linear = [
    mean_absolute_error(Y_test, y_prediction_lin),
]
polynomial = [
    mean_absolute_error(Y_test, y_prediction_poly),
]

x = np.arange(len(labels))  # the label locations
width = 0.15  # the width of the bars

fig, ax = plt.subplots()
rects1 = ax.bar(x - width/2, linear, width, label='Linear regression')
rects2 = ax.bar(x + width/2, polynomial, width, label='Polynomial regress

fig.set_size_inches(18.5, 10.5)

# Add some text for labels, title and custom x-axis tick labels, etc.
ax.set_ylabel('Scores')
ax.set_title('Scores of different regression')
ax.set_xticks(x, labels)
ax.legend()

ax.bar_label(rects1, padding=3)
ax.bar_label(rects2, padding=3)

fig.tight_layout()

plt.show()
```



So here we have the results. As you can see, MAE in polynomial regression is 34.32115945730862. It is 35.9082260163343 in the linear regression, so we can see that the results for polynomial regression are slightly better, and the reason that there is not much difference is because of the form of the data we have. Nevertheless, it is interesting that the improvement is visible even in such a linear dataset.

In [427…

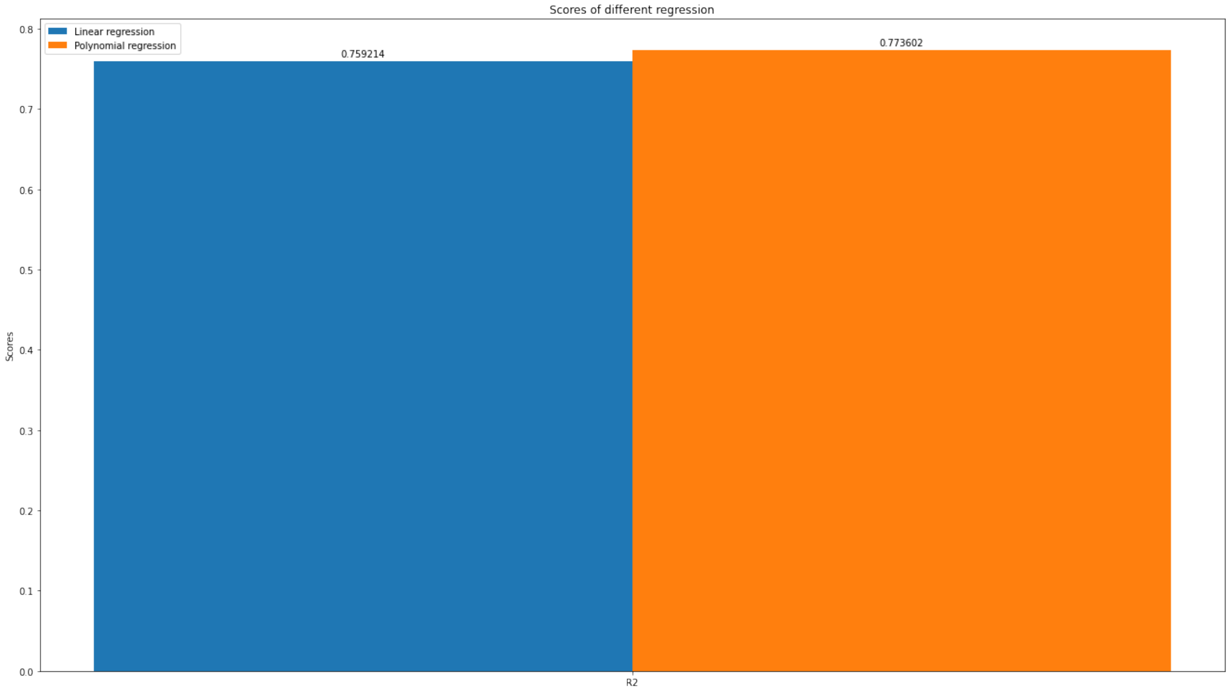
```
labels = ['MSE']

linear = [
    mean_squared_error(Y_test, y_prediction_lin),
]
polynomial = [
    mean_squared_error(Y_test, y_prediction_poly),
]

x = np.arange(len(labels))  # the label locations
width = 0.15  # the width of the bars

fig, ax = plt.subplots()
rects1 = ax.bar(x - width/2, linear, width, label='Linear regression')
rects2 = ax.bar(x + width/2, polynomial, width, label='Polynomial regress

fig.set_size_inches(18.5, 10.5)

# Add some text for labels, title and custom x-axis tick labels, etc.
ax.set_ylabel('Scores')
ax.set_title('Scores of different regression')
ax.set_xticks(x, labels)
ax.legend()

ax.bar_label(rects1, padding=3)
ax.bar_label(rects2, padding=3)

fig.tight_layout()

plt.show()
```



So here we have the results. As you can see, MSE in polynomial regression is 1985.2979983086886. It is 2111.4669534348354 in the linear regression, so we can see that the results for polynomial regression are better, and the reason that there is not much difference is because of the form of the data we have. However, it is interesting that the improvement is visible even in such a linear dataset.

In [428…

```
labels = ['R2']

linear = [
    r2_score(Y_test, y_prediction_lin),
]
polynomial = [
    r2_score(Y_test, y_prediction_poly),
]

x = np.arange(len(labels))  # the label locations
width = 0.15  # the width of the bars

fig, ax = plt.subplots()
rects1 = ax.bar(x - width/2, linear, width, label='Linear regression')
rects2 = ax.bar(x + width/2, polynomial, width, label='Polynomial regress

fig.set_size_inches(18.5, 10.5)

# Add some text for labels, title and custom x-axis tick labels, etc.
ax.set_ylabel('Scores')
ax.set_title('Scores of different regression')
ax.set_xticks(x, labels)
ax.legend()

ax.bar_label(rects1, padding=3)
ax.bar_label(rects2, padding=3)

fig.tight_layout()

plt.show()
```



So here we have the results. As you can see, R2 in polynomial regression is 0.7736020206811265, and it is 0.7592140564975793 in the linear regression, so we can see that the results for polynomial regression are better and it is a better fit. The reason that there is not much difference is because of the form of the data we have. However, it is interesting that the improvement is visible even in such a linear dataset. </br> So overall, the polynomial regression is better than the linear regression.

$$The End$$

```
In [ ]:
```