In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

In [2]:
```
df = pd.read_csv('../COMM054_6442385_CW/cancerdata.csv')
```

Train / Test Splits

In [3]:
```
X = df.drop('Class',axis=1)
y = df['Class']
```

In [4]:
```
#Importing train test split used for splitting data
from sklearn.model_selection import train_test_split
```

In [5]:
```
#Tuple Unpacking /split train/test size to 60/40 percent  of dataset with a random state
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4, random_state=15
```

Feature Scaling

In [6]:
```
#Import StandardScaler used for feature scaling
from sklearn.preprocessing import StandardScaler
```

In [7]:
```
#Initialize Scaler
scaler = StandardScaler()
```

In [8]:
```
# fit transform data on train set and transform only on test set to avoid data leakage
scaled_X_train = scaler.fit_transform(X_train)
scaled_X_test = scaler.transform(X_test)
```

Logistic Regression Overview

Logistic Regression is a supervised machine-learning algorithm mainly used for solving classification problems. In our case, we will be using binary logistic regression as the target label has only two outputs i.e class 0(benign) or class 1(malignant). The logistic regression line is fit to data using an s-shaped curve with probability values from 0 to 1. The higher the probability values, the higher the chance of belonging to a class. Based on the probability threshold set the data points are then classified accordingly. Building on linear regression, logistic regression uses the logistic function shown in figure (1) to convert all values to be between 0 and 1.

Figure 1

Using the logistic function we can then transform a linear regression to a logistic regression by plugging in the well-known linear regression formula i.e the sum of Beta coefficients multiplied by their respective x features, into the x value of the logistic function we can then generate the formula for logistic regression which can then be plotted out to create the s-shaped curve seen in figure(1)

Figure 2

From some basic math applied to the formula in figure(2), we see the logistic regression formula can be expressed in terms of the probability odds formula (P/1-P). To get rid of the exponential value (e) value we multiply through by the natural log(ln). This then transforms the original logistic regression formula to the log odds of the logistic regression and scales it from positive infinity to negative infinity (fig 3). Converting the probability to log odds probability is useful as it helps improve the interpretability of the coefficients

Figure 3

As the values are scaled from positive to negative infinity it would be impossible to use least squares to find the best-fitting line as the residual values will also be equal to positive or negative infinity. Hence to find the best fitting line for logistic regression we use maximum likelihood. Maximum Likelihood Estimation is a method used to estimate the parameters of a probability distribution given some data. Firstly we assume a best-fitting line for the log odds graph (fig 3) and project the data observations onto it. This line is then transformed back to the probability line (fig 1) using the formula e^log(odds)/ 1+ e^log(odds) and then used to calculate the likelihood of that squiggle i.e by adding the log-likelihoods of each point on the probability line. Using gradient descent this process is then repeated until we find the line with the minimum negative likelihood value cost function. After the line of best fit is found, a decision boundary is set for classifying new data points with values above the boundary belonging to one class and values below the boundary belonging to another class

Logistic Regression Model Overview

The following steps will be taken for logistic regression model implementation:

1. Initialize the ML model to be used. For this task, the built-in LogisticRegression model from the linear_model package will be used
2. Decide if feature scaling is necessary. As we will be adding a regularization penalty to the logistic regression model it will need to be scaled in order for the correct penalizations to be applied.
3. Decide if hyperparameter tuning will be necessary before fitting the model. As Logistic Regression is a model prone to overfitting we will consider regularization techniques such as **L1 regularization(Lasso Regression)** and **L2 Regularization (Ridge Regression)** to apply a penalty to our model. A grid search to determine the optimal penalty to be applied to our model will be used as the dataset is small enough hence a lower computation expense. This is also a mitigation approach as the dataset has an extremely sensitive target label i.e incorrectly classifying a person's cancer as benign when it was malignant could prove catastrophic
4. Fit the hyper tuned log model to the training set and make predictions using the test set

Logistic Regression Model Building

In [9]:
```
#Logistic Regression Model Import
from sklearn.linear_model import LogisticRegression
```

In [10]:
```
#Define Base Logistic Model
#Set solver to LibLinear as this works better for small datasets
#Set maximum Iterations to a reasonable high value so gradient descent has enough iterat
log_model= LogisticRegression(solver = 'liblinear' , max_iter= 5000)
```

In [11]:
```
#Import Grid SearchCV which will be to be used for Cross Validation / Hyper parameter Tu
from sklearn.model_selection import GridSearchCV
```

In [12]:
```
penalty= ['l1','l2']## Define Regularization Penalty to be used L1(Lasso Regression) or
C= np.logspace(0,10,20) # Regularization Strength Parameter (C) with smaller values mean
param_grid={'penalty':penalty, 'C':C }
```

In [13]:
```
#create instance of grid search to be used
grid_model = GridSearchCV(log_model,param_grid=param_grid)
```

In [14]:
```
#fit training data to grid model
grid_model.fit(scaled_X_train,y_train)
```

Out[14]:
```
GridSearchCV(estimator=LogisticRegression(max_iter=5000, solver='liblinear'),
             param_grid={'C': array([1.00000000e+00, 3.35981829e+00, 1.12883789e+01, 3.7
9269019e+01,
       1.27427499e+02, 4.28133240e+02, 1.43844989e+03, 4.83293024e+03,
       1.62377674e+04, 5.45559478e+04, 1.83298071e+05, 6.15848211e+05,
       2.06913808e+06, 6.95192796e+06, 2.33572147e+07, 7.84759970e+07,
       2.63665090e+08, 8.85866790e+08, 2.97635144e+09, 1.00000000e+10]),
                         'penalty': ['l1', 'l2']})
```

In [15]:
```
#check best performign parameters which will be used for the log model
grid_model.best_params_
```

Out[15]:
```
{'C': 1.0, 'penalty': 'l1'}
```

In [16]:
```
#fit log model using best paameters from grid search
log_model = LogisticRegression(C = 1.0,
                               penalty = 'l1',
                               solver = 'liblinear')
log_model.fit(scaled_X_train,y_train)
```

Out[16]:
```
LogisticRegression(penalty='l1', solver='liblinear')
```

Naive Bayes Overview

Naive Bayes is a supervised machine learning model used to solve classification problems through the use of bayes theorem. Bayes theorem describes the probability of an event A given prior knowledge of another event B. This proves useful as it makes it possible to predict the probability of a target label given some set of features as seen in figure(4). Unlike bayes theorem, naive bayes assumes independence amongst features to help computation when calculating joint distributions. However, assuming independence between features is impractical in a real-world setting hence the name ‘naive’ bayes.

Figure 4

Guassian Naive Bayes

Gaussian Naive Bayes is an extension of the Naive Bayes classifier. It assumes continuous features follow a normal distribution. With this assumption, the probability density function of the normal distribution (figure 5) can be used to determine the probability of likelihoods.

Figure 5

By calculating the mean and variance of the distribution, predictions can be made by plugging said parameters into the gaussian probability density function(PDF) with the new variable inputs for the variable. The Gaussian PDF will then provide estimates of the probability of new input values for the target classes i.e class 0 and class 1.

Naive Bayes Model Overview

The following steps will be taken for naive bayes model implementation:

1. Decide on what type of naive bayes is suitable for the dataset problem definition. There are different implementations of naive bayes used to solve different problems such as Gaussian, Bernoulli and Multinomial as the dataset is made up of continuous data an assumption is made that this follows a normal distribution hence **gaussian naive bayes** will be used to train the data
2. Decide if the data needs feature scaling. Although naive bayes is a probability-based algorithm which means it doesn't necessarily need feature scaling, as we are dealing with gaussian naive bayes I will scale the data so it has a mean of 0 and a standard deviation of 1 on the normal distribution curve
3. Decide if hyperparameter tuning will be necessary before fitting the model. As naive Bayes has very few hyperparameters, hyperparameter tuning won't be necessary to improve model performance so will not be used
4. Fit the model to the training set and make predictions from the test set

Naive Bayes Model Building

In [17]:
```
#Import Gaussian NaiveBayes as our feature set is continuous data
from sklearn.naive_bayes import GaussianNB
```

In [18]:
```
naive_model = GaussianNB()
```

In [19]:
```
naive_model.fit(scaled_X_train,y_train)
```

Out[19]:
```
GaussianNB()
```

Model Evaluation Overview

To check model performance for both models the following evaluation metrics will be used:

1. Confusion Matrix - This is a table used to visualize the performance of the model's classification on unseen data. It comprises 4 values (True Positives, True Negatives, False Positives, and False Negatives)

- True Positives -This metric tells how many correct predictions the model made for classifying malignant cancer cases
- True Negatives -This metric tells how many correct predictions the model made for classifying benign cancer cases
- False Negatives-This metric tells how many cancer cases the model predicted as benign but was malignant
- False Positives-This metric tells how many cancer cases the model predicted as malignant but was benign

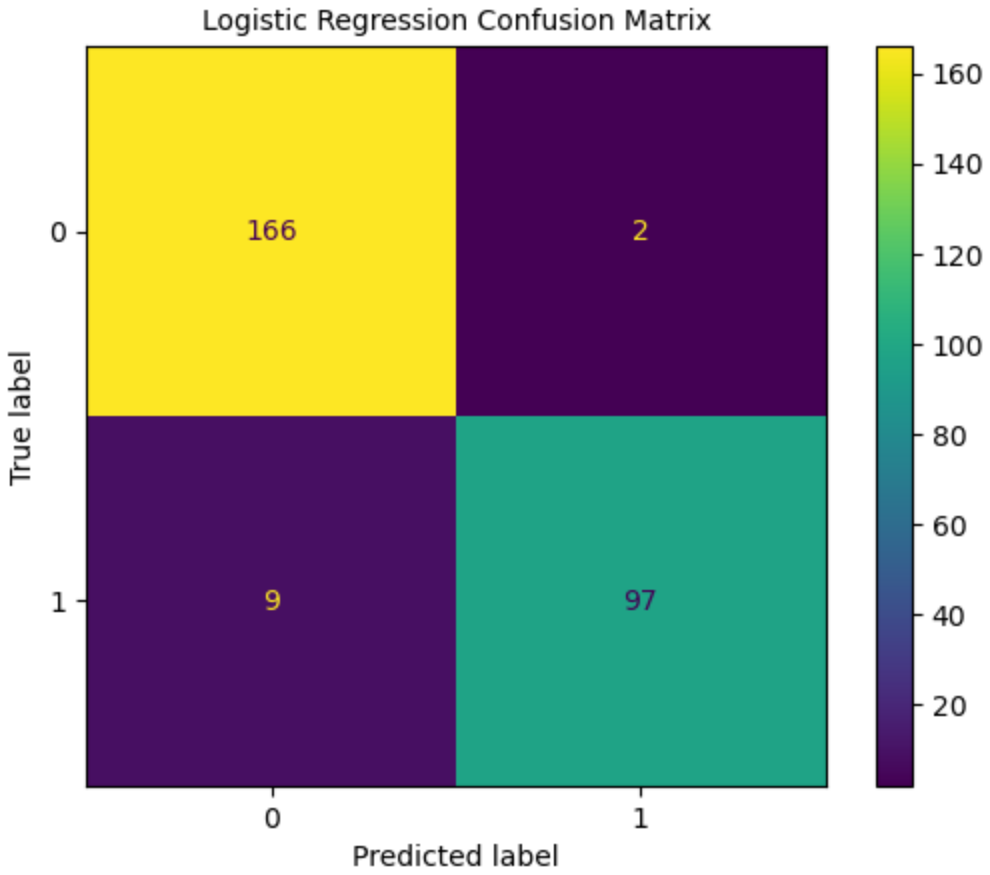
1. Accuracy - This metric shows overall how well the model is performing at classifying both true positives and True Negatives. The formula to calculate this is gotten from the confusion matrix(True Positives + True Negatives / True Positives +True Negatives +False Positives + False Negatives). A high accuracy score shows the model performed well overall when making classifications
2. Precision- This metric tells the proportion of positive predictions that were actually positive i.e for cases of cancer that were predicted as malignant, what proportion of cases did our model correctly predict. The formula to calculate this can be derived from the confusion matrix(True Positives/ True Positives + False Positives)
3. Recall - This metric tells the model's ability to find all the relevant cases in the data set i.e for all the cases of cancer that were malignant positive, what percentage was the model able to correctly predict. The formula to calculate this can be derived from the confusion matrix(True Positives/ True Positives + False Negatives). This metric is particularly important to calculate as a low recall value would mean a lot of patients' cancer was incorrectly classified as benign even tho it was malignant which is a very big health issue if not addressed **hence recall will be the most important metric considered for final model selection**
4. F1-Score - This metric combines precision and recall by taking the harmonic mean between them hence a high f1 score would mean good performance for both precision and recall. We will be considering the f1 score due to the slight imbalance of the target label
5. ROC/AUC Curve - The ROC(Reciever Operating Characteristic) curve visualizes the true positive rate/sensitivity against the false positive rate/specificity. It's a visual way of representing the predictive power of a model i.e classifiers with curves closer to the top left corner are indicative of better model performance. The AUC is the area Under the Roc Curve. It tells how well the model performed at distinguishing between classes. A high AUC value means the model performed well at distinguishing the type of cancer a patient had

Model Evaluation

In [20]:
```
from sklearn.metrics import (classification_report,ConfusionMatrixDisplay ,confusion_mat
                             precision_recall_curve ,precision_score,recall_score , accu
from sklearn.metrics import SCORERS
from sklearn import metrics
```

Logistic Regression Model Evaluation

In [21]:
```
log_pred=log_model.predict(scaled_X_test)
log_cm = confusion_matrix(y_test, log_pred, labels=log_model.classes_)
disp = ConfusionMatrixDisplay(confusion_matrix=log_cm,display_labels=log_model.classes_)
disp.plot()
plt.title('Logistic Regression Confusion Matrix' , size = 10)
plt.show()
```



From the confusion matrix, we see the logistic regression model performed well, however, the high amount of false negatives is a concern as this would mean 9 patients had their cancer wrongly classified as benign when it was malignant

In [22]:
```
print(classification_report(y_test,log_pred))
```

```
              precision    recall  f1-score   support

           0       0.95      0.99      0.97       168
           1       0.98      0.92      0.95       106

    accuracy                           0.96       274
   macro avg       0.96      0.95      0.96       274
weighted avg       0.96      0.96      0.96       274
```

The classification report shows high performance for the logistic regression model across all metrics for both the benign classifications(0) and the malignant classifications(1)

Beta Coefficient Plot For Logistic Regression

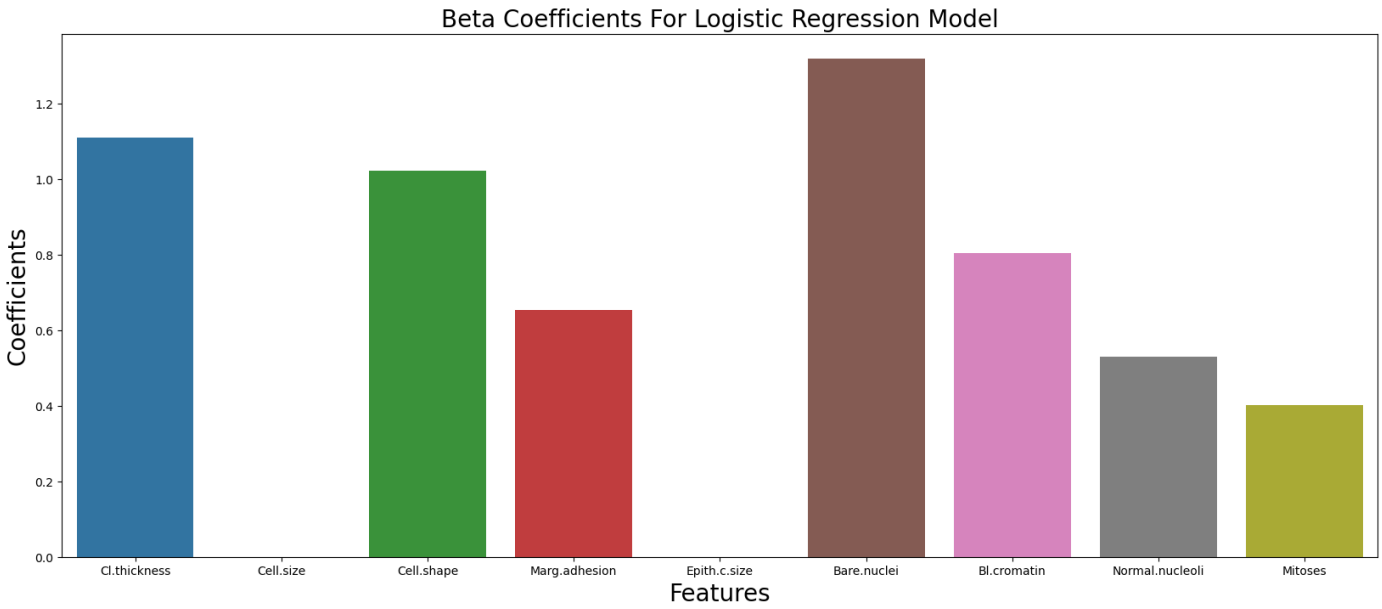
In [23]:
```
#set of dataframe of logistic regression model coefficients and reshape
coefDf = pd.DataFrame( data = log_model.coef_.reshape(9,1))
#add featurecolumns to dataframe
coefDf['Features'] = X_train.columns
coefDf.columns = ['Coefficient', 'Features']
coefDf
```

Out[23]:

	Coefficient	Features
0	1.108903	Cl.thickness
1	0.000000	Cell.size

2	1.021952	Cell.shape
3	0.652393	Marg.adhesion
4	0.000000	Epith.c.size
5	1.317985	Bare.nuclei
6	0.804331	Bl.cromatin
7	0.530234	Normal.nucleoli
8	0.401141	Mitoses

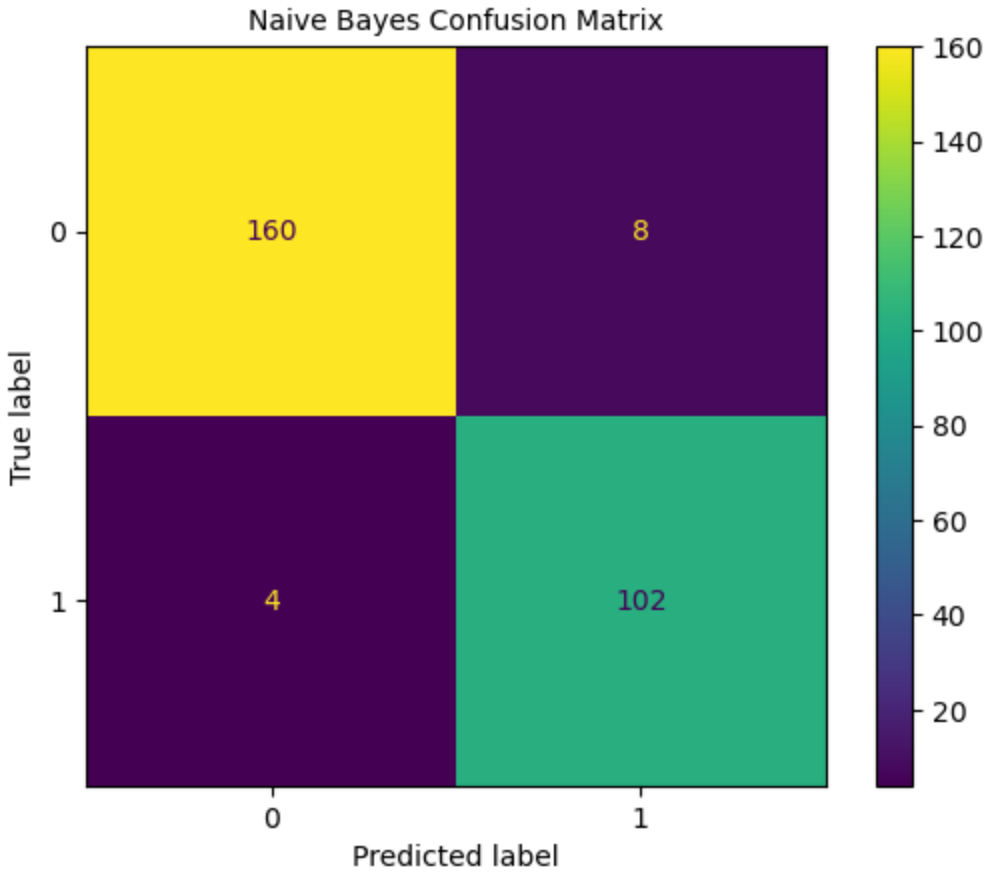
In [24]:
```
#bar plot of log model coefficients for respective features
plt.figure(figsize=(20,8),dpi=100)
sns.barplot(x = 'Features',
            y = 'Coefficient',
            data = coefDf)

plt.xlabel('Features' , size=20)
plt.ylabel('Coefficients' , size=20)
plt.title('Beta Coefficients For Logistic Regression Model' , size =20)
plt.show()
```



We can see from the beta coefficients Cl thickness and Bare.nuclei were the most important features used by the logistic regression model for making cancer classifications cases(benign or malignant)

Naive Bayes Model Evaluation

In [25]:
```
naive_pred = naive_model.predict(scaled_X_test)
naive_cm = confusion_matrix(y_test, naive_pred, labels=naive_model.classes_)
disp = ConfusionMatrixDisplay(confusion_matrix=naive_cm,display_labels=naive_model.class
disp.plot()
plt.title('Naive Bayes Confusion Matrix' , size = 10)
plt.show()
```



From the confusion matrix, we see the naive bayes model was able to minimize false negatives better with less than double the number of false negatives of the logistic regression model

In [26]:
```
print(classification_report(y_test,naive_pred))
```

```
              precision    recall  f1-score   support

           0       0.98      0.95      0.96       168
           1       0.93      0.96      0.94       106

    accuracy                           0.96       274
   macro avg       0.95      0.96      0.95       274
weighted avg       0.96      0.96      0.96       274
```

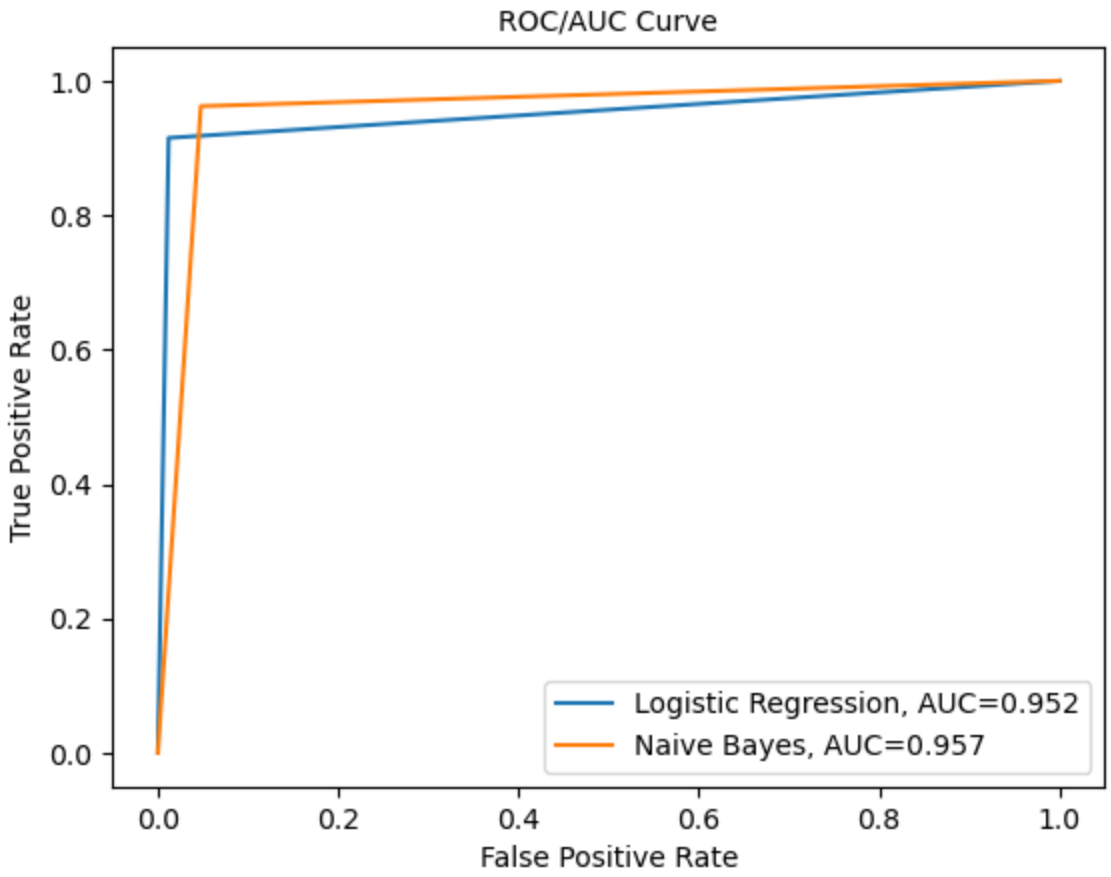
The classification report shows high performance for the naive bayes model across all metrics for both the benign classifications(0) and the malignant classifications(1)

Combined ROC/AUC Curve

In [27]:
```
#define false postive and true positive rate for logistic regression
fpr, tpr, _ = metrics.roc_curve(y_test, log_pred)

auc = round(metrics.roc_auc_score(y_test, log_pred), 3)
plt.plot(fpr,tpr,label="Logistic Regression, AUC="+str(auc))

#define false postive and true positive rate for naive bayes
fpr, tpr, _ = metrics.roc_curve(y_test, naive_pred)
auc = round(metrics.roc_auc_score(y_test, naive_pred), 3)
plt.plot(fpr,tpr,label="Naive Bayes, AUC="+str(auc))
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC/AUC Curve' , size =10)
```

```
#add legend
plt.legend()
```

Out[27]:
```
<matplotlib.legend.Legend at 0x7f98b15d0250>
```

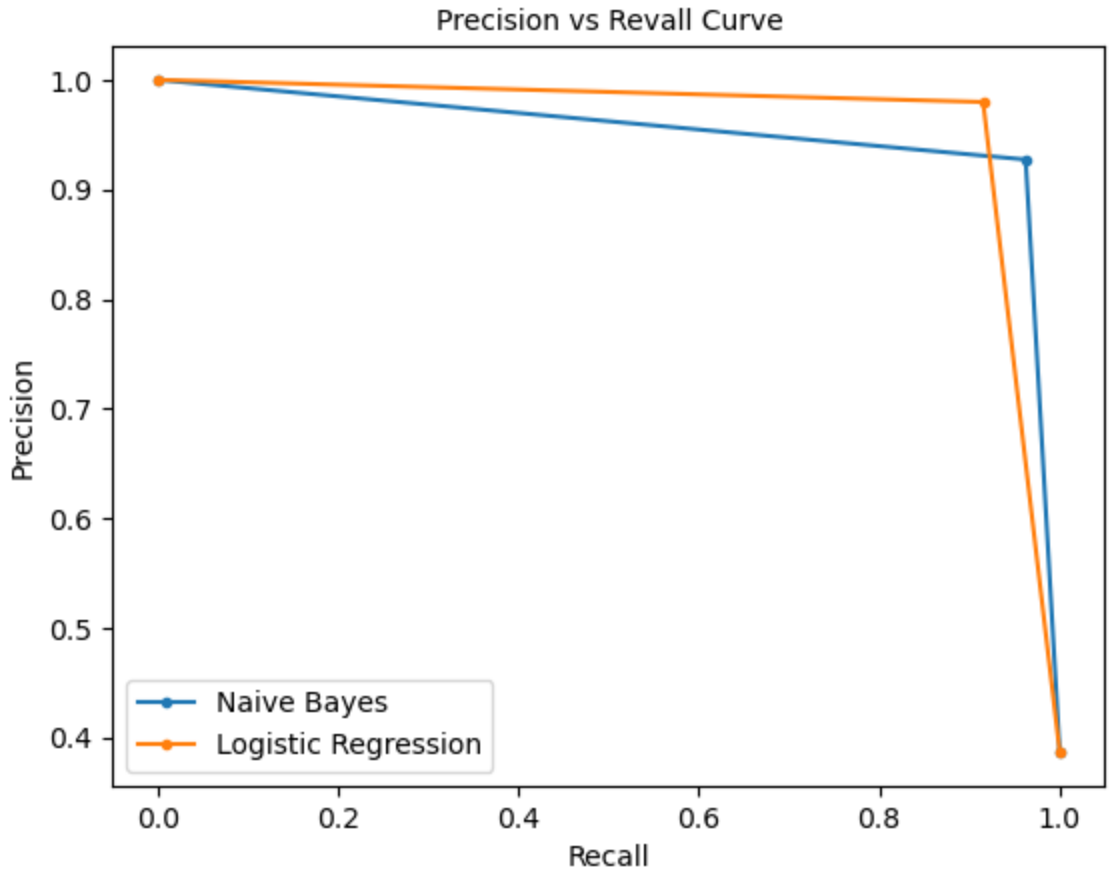


The Roc Curve visually shows high performance with the curve very close to the left-hand side this can be further backed up with the corresponding Auc value calculated showing both models had very high performance when distinguishing between classes with naive bayes having slightly better performance overall

Combined Precision Recall Curve

In [28]:
```
lr_precision, lr_recall, _ = precision_recall_curve(y_test, log_pred)
naive_precision, naive_recall, _ = precision_recall_curve(y_test, naive_pred)
```

In [29]:
```
plt.plot(naive_recall, naive_precision, marker='.', label='Naive Bayes')
plt.plot(lr_recall, lr_precision, marker='.', label='Logistic Regression')
# axis labels
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Precision vs Revall Curve', size =10)
# show legend
plt.legend()
# show plot
plt.show()
```



The precision-recall curve visually shows very good precision and recall performance at all thresholds. Hence we can conclude both models were good classifiers for the given data

Model Evaluation Conclusion

Both models had a very high performance with logistic regression having a slightly higher accuracy of 0.959 compared to the accuracy score naive bayes achieved of 0.956. In terms of precision naive bayes also performed better with a precision score of 0.979 compared to the precision score of 0.927 achieved by the naive bayes model. In terms of recall, our almost important naive bayes performed significantly better with a recall score of 0.962 compares to the recall score of 0.915 achieved by the logistic regression model

Model Validation Overview

To ensure optimal model performance with little or no bias the following validation steps were taken :

1. Train/Test Splits - This is a validation method used to test how our model reacts to truly unseen data. Validation sets also help with this but were not considered as the data set is already quite small. For this project, a 60/40 train test split was used
2. Kfold cross-validation - This is a cross-validation technique that splits the data into folds. For each fold, the model is trained on k-1 folds of the data and then evaluated on the remaining data. It is a useful validation technique as it helps check our model didn't overfit
3. McNemar's test - This is a model comparison technique used to check how well predictions made between the two models(Logistic Regression and Naive Bayes) match up or differ. A contingency table is built with the values :

- A- number of misclassifications made by both models

- B- number of misclassifications made by Logistic Regression but not by Naive Bayes
- C- number of misclassifications made by Naive Bayes but not by Logistic Regression
- D- number of correct classifications made by both models

From the contingency table, x^2 can then be calculated using the formula (B-C)^2/B+C which can then be used to compute the p-value. A p-value lower than 0.05 would mean the null hypothesis(one model isn't significantly better than the other) can be rejected.

Model Validation

In [30]:
```
from sklearn.model_selection import cross_validate , cross_val_score ,KFold
from mlxtend.evaluate import mcnemar_table, mcnemar
from sklearn import model_selection
from sklearn.metrics import SCORERS
```

In [31]:
```
# set k fold with 10 splits
kf = model_selection.KFold(n_splits=10)
```

Naive Bayes Model Validation

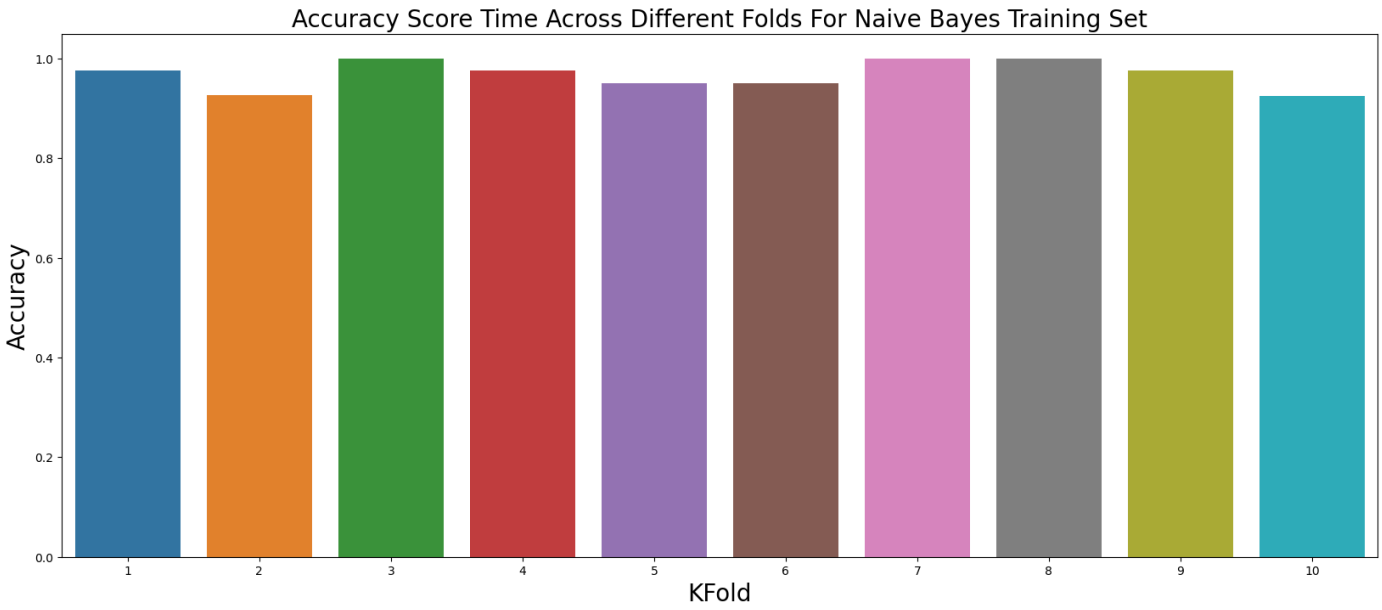
K-fold Cross-Validation Using Cross Validate

In [32]:
```
#kfold cross validation on naive bayes model train and test set
train_score_naive = cross_validate(naive_model  , scaled_X_train, y_train , scoring =['a
test_score_naive = cross_validate(naive_model  , scaled_X_test, y_test , scoring =['accu
```

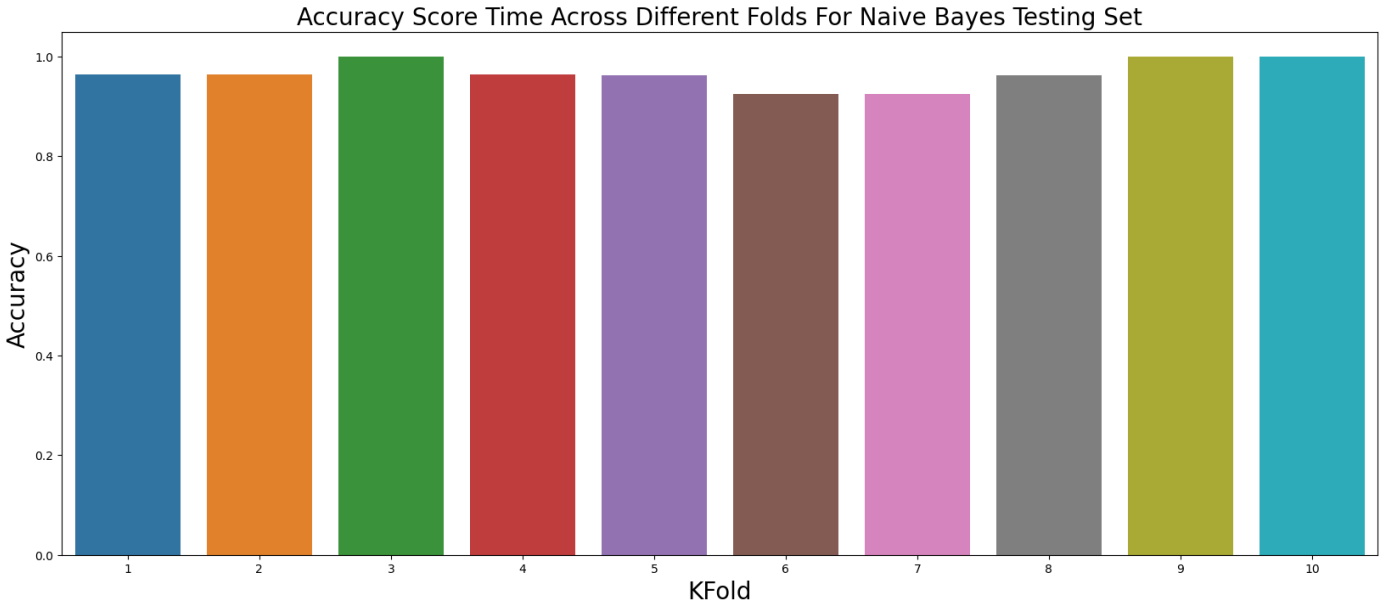
In [33]:
```
#create dataframe of results from cross validation
train_score_naiveDf = pd.DataFrame(train_score_naive)
test_score_naiveDf = pd.DataFrame(test_score_naive)
```

In [34]:
```
#create new column containing folds
train_score_naiveDf['fold'] = list(range(1, 11))
test_score_naiveDf['fold'] = list(range(1, 11))
```

In [35]:
```
#barplot of accuracy per fold fo train set
plt.figure(figsize=(20,8),dpi=100)
sns.barplot(x = 'fold',
            y = 'test_accuracy',
            data = train_score_naiveDf)

plt.xlabel('KFold' , size=20)
plt.ylabel('Accuracy' , size=20)
plt.title('Accuracy Score Time Across Different Folds For Naive Bayes Training Set' , si
plt.show()
```



In [36]:
```
#bar plot of accuracy per fold for test set
plt.figure(figsize=(20,8),dpi=100)
sns.barplot(x = 'fold',
            y = 'test_accuracy',
            data = test_score_naiveDf)

plt.xlabel('KFold' , size=20)
plt.ylabel('Accuracy' , size=20)
plt.title('Accuracy Score Time Across Different Folds For Naive Bayes Testing Set' , siz
plt.show()
```



In [37]:
```
train_score_naiveDf['test_accuracy'].mean()
```

Out[37]:
```
0.968109756097561
```

In [38]:
```
test_score_naiveDf['test_accuracy'].mean()
```

Out[38]:
```
0.9670634920634921
```

Similar mean accuracy score for both testing and training sets across different cross-validation folds confirms the naive bayes model didn't overfit despite the high accuracy scores gotten

Logistic Regression Model Validation

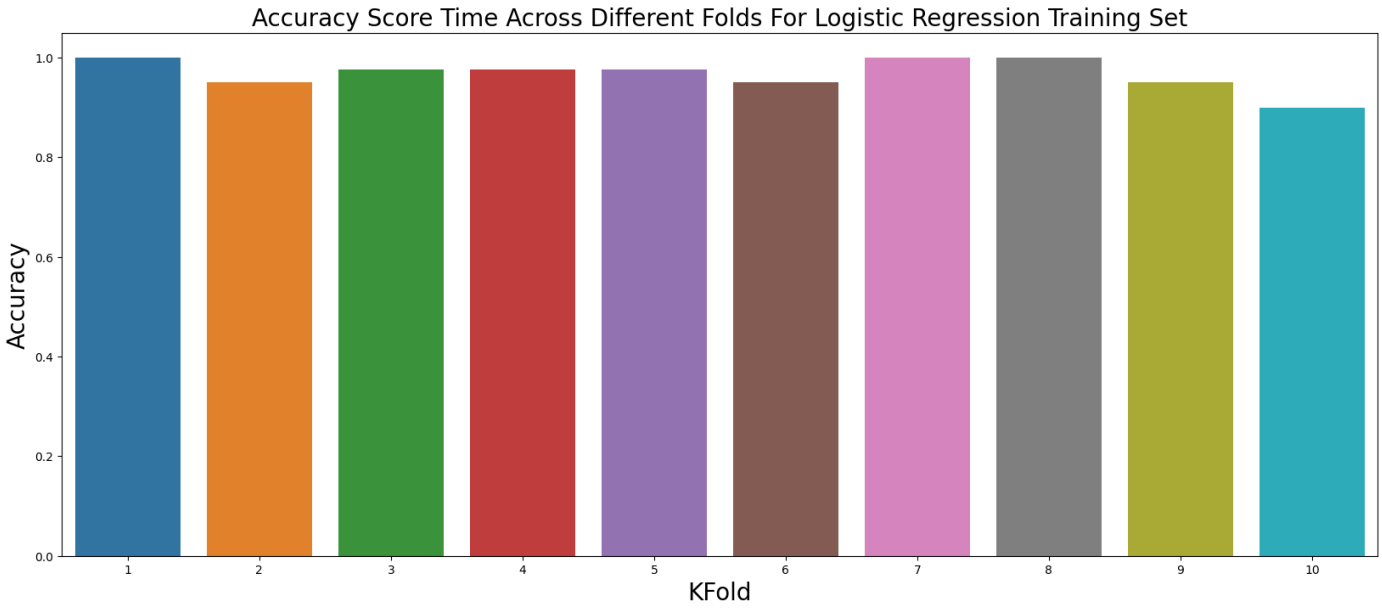
K-fold Cross-Validation Using Cross Validate

In [39]:
```
#kfold cross validation on logistic regression model train and test set
train_score_logreg = cross_validate(log_model  , scaled_X_train, y_train , scoring =['ac
test_score_logreg = cross_validate(log_model  , scaled_X_test, y_test , scoring =['accur
```

In [40]:
```
#create dataframe of results from cross validation
train_score_logregDf = pd.DataFrame(train_score_logreg)
test_score_logregDf = pd.DataFrame(test_score_logreg)
```

In [41]:
```
#create new column containing folds
train_score_logregDf['fold'] = list(range(1, 11))
test_score_logregDf['fold'] = list(range(1, 11))
```

In [42]:
```
#bar plot of accuracy per fold for train set
plt.figure(figsize=(20,8),dpi=100)
sns.barplot(x = 'fold',
            y = 'test_accuracy',
            data = train_score_logregDf)

plt.xlabel('KFold' , size=20)
plt.ylabel('Accuracy' , size=20)
plt.title('Accuracy Score Time Across Different Folds For Logistic Regression Training S
plt.show()
```



In [43]:
```
#bar plot of accuracy per fold for test set
plt.figure(figsize=(20,8),dpi=100)
bar= sns.barplot(x = 'fold',
            y = 'test_accuracy',
            data = test_score_logregDf)
plt.xlabel('KFold' , size=20)
plt.ylabel('Accuracy' , size=20)
plt.title('Accuracy Score Time Across Different Folds For Logistic Regression Testing Se
plt.show()
```

In [44]:
```
train_score_logregDf['test_accuracy'].mean()
```

Out[44]:
```
0.968048780487805
```

In [45]:
```
test_score_logregDf['test_accuracy'].mean()
```

Out[45]:
```
0.9597883597883599
```

Similar mean accuracy score for both testing and training sets across different cross-validation folds confirms the logistic regression model didn't overfit despite the high accuracy scores gotten

Model Comparison

MCNERS Test

In [46]:
```
# Set up contingency table to be used
contingency_table = mcnemar_table(y_target=y_test,
                   y_model1=log_pred,
                   y_model2=naive_pred)
#calculate chi squared and p values from contingency table
chivalue, pvalue = mcnemar(ary=contingency_table, exact=True)

print('chi-squared:', chivalue)
print('p-value:', pvalue)
```

```
chi-squared: 5
p-value: 1.0
```

From Mcner's test we get a p-value of 1 which is above the threshold of 0.05 hence we can reject the null hypothesis (neither the logistic regression nor naive bayes models performed significantly better than the other) and therefore conclude one model performed significantly better than the other

Metrics Barplot

In [47]:
```
#define new variable results
results = {'Model': ['LogisticRegression', 'NaiveBayes', 'LogisticRegression', 'NaiveBay
        'Metric': ['Accuracy', 'Accuracy', 'Precision', 'Precision', 'Recall','Recall'],
          'Score': [accuracy_score(y_test,log_pred), accuracy_score(y_test,naive_pred),
```

In [48]:
```
recall_score(y_test,log_pred)
```

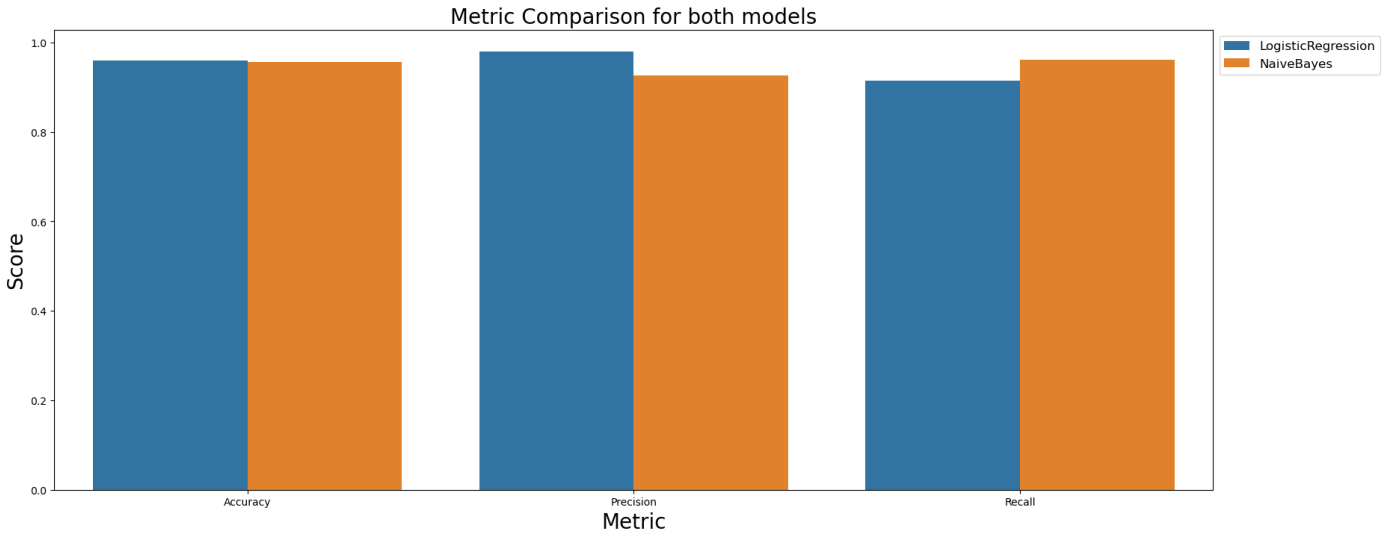
Out[48]:
```
0.9150943396226415
```

In [49]:
```
resultDF= pd.DataFrame(results)
resultDF
```

Out[49]:

	Model	Metric	Score
0	LogisticRegression	Accuracy	0.959854
1	NaiveBayes	Accuracy	0.956204
2	LogisticRegression	Precision	0.979798
3	NaiveBayes	Precision	0.927273
4	LogisticRegression	Recall	0.915094
5	NaiveBayes	Recall	0.962264

In [50]:
```
#barplot of accuracy precision and recall scores for both models
plt.figure(figsize=(20,8),dpi=100)
sns.barplot(x = 'Metric',
            y = 'Score', hue='Model',
            data = resultDF)

plt.legend(bbox_to_anchor=(1, 1), loc='upper left' ,fontsize=12)
plt.xlabel('Metric' , size=20)
plt.ylabel('Score' , size=20)
plt.title("Metric Comparison for both models", size=20)
plt.show()
```



Visually we see logistic regression performed better for accuracy ad precision metrics ad naive bayes performed better for recall

Conclusion

Both models gave a very high performance for all metrics above. However naive bayes performed better when it came to the recall i.e minimizing false negatives. The slightly lower performance on logistic regression could be due to the regularization penalties applied to it. So although it performed worse than naive bayes , in a real-world setting it might be a more appropriate model to use considering the regularization penalty applied to it reduced the variance in the data while the naive bayes model made assumptions when making classifications that all features were independent but as we know from

domain experience features such as cell shape and cell size aren't independent of each other. However, for the scope of this project naive bayes would be the optimal model choice as it managed to minimize false negatives and hence improve recall which was the most important metric considered as stated in the model evaluation overview

Image References

https://www.udemy.com/course/python-for-machine-learning-data-science-masterclass/learn/lecture/23309126#overview https://www.upgrad.com/blog/gaussian-naive-bayes/