Predicting Malignant Breast Cancer Tumors

Overview:

According to healthline, breast imaging techniques such as mammograms, ultrasounds and MRIs can detect suspicious areas, but can't tell a patient whether they have cancer or not – only a biopsy can fully diagnose cancer*.

An estimated 297,790 women are expected to be diagnosed with breast cancer in 2023, which the American Cancer Society notes is the second leading cause of cancer death in women**.

What if there was a way, through existing imaging techniques, to get tumor measurements and be able to predict if it was malignant or benign? That is exactly what the purpose of this project is – creating a model that accurately predicts malignant tumors given existing measurements.

Sources: https://www.healthline.com/health/breast-cancer/can-a-radiologist-tell-if-it-is-breast-cancer (https://www.healthline.com/health/breast-cancer/can-a-radiologist-tell-if-it-is-breast-cancer) , https://www.cancer.org/cancer/breast-cancer/about/how-common-is-breast-cancer.html (https://www.cancer.org/cancer/breast-cancer/about/how-common-is-breast-cancer.html)

Using data from the State of Wisconsin on Breast Cancer tumors, can we predict whether a tumor is malignant? Getting an accurate model with a very small level of False Negatives is imperative, and could allow for malignant tumors to be identified faster than they normally would, which could mean better treatment results for the patients.

We want a high accuracy rate, but ensuring that there are minimal amounts of False Negatives is equally or potentially even more important. If we have false negatives, then those patients could end up having a delayed diagnosis, and delayed treatment, which in some cases can be fatal.

Data Source:

The data was pulled from Kaggle.com, an AirBnb for data scientists. Kaggle is a crowd-sourced platform to attract, nurture, train and challenge data scientists.

From there, I pulled a dataset containing measurements and characteristics from breast tumors coming out of the State of Wisconsin, which were classified as malignant or benign.

Import Libraries:

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

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import confusion_matrix, precision_score, recall_score,
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier

import warnings
#use to ignore warnings
warnings.filterwarnings('ignore')

%matplotlib inline

#from sklearn import linear_model, decomposition, datasets
#from sklearn.pipeline import Pipeline
```

Import and Summarize Data:

Reading csv containing data into dataframe using pandas:

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

Showing and Describing the Data:

Showing first five rows of dataframe:

In [3]:
```
#first five rows
df.head()
```

Out[3]:

	id	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	smoothr
0	842302	M	17.99	10.38	122.80	1001.0	
1	842517	M	20.57	17.77	132.90	1326.0	
2	84300903	M	19.69	21.25	130.00	1203.0	
3	84348301	M	11.42	20.38	77.58	386.1	
4	84358402	M	20.29	14.34	135.10	1297.0	

5 rows × 33 columns

Showing last five rows of dataframe:

In [4]:
```
#last five rows
df.tail()
```

Out[4]:

	id	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_
564	926424	M	21.56	22.39	142.00	1479.0	0
565	926682	M	20.13	28.25	131.20	1261.0	0
566	926954	M	16.60	28.08	108.30	858.1	0
567	927241	M	20.60	29.33	140.10	1265.0	0
568	92751	B	7.76	24.54	47.92	181.0	0

5 rows × 33 columns

Getting the shape of the dataframe:

In [5]:
```
#Show the shape of the data
df.shape
```

Out[5]: (569, 33)

From this, we can see that there are: 33 columns, with 569 total rows.

Getting summary statistics:

In [6]:
```
#summary stats
df.describe().T
```

Out[6]:

	count	mean	std	min	25%	
id	569.0	3.037183e+07	1.250206e+08	8670.000000	869218.000000	9060
radius_mean	569.0	1.412729e+01	3.524049e+00	6.981000	11.700000	
texture_mean	569.0	1.928965e+01	4.301036e+00	9.710000	16.170000	
perimeter_mean	569.0	9.196903e+01	2.429898e+01	43.790000	75.170000	
area_mean	569.0	6.548891e+02	3.519141e+02	143.500000	420.300000	5
smoothness_mean	569.0	9.636028e-02	1.406413e-02	0.052630	0.086370	
compactness_mean	569.0	1.043410e-01	5.281276e-02	0.019380	0.064920	
concavity_mean	569.0	8.879932e-02	7.971981e-02	0.000000	0.029560	
concave points_mean	569.0	4.891915e-02	3.880284e-02	0.000000	0.020310	
symmetry_mean	569.0	1.811619e-01	2.741428e-02	0.106000	0.161900	
fractal_dimension_mean	569.0	6.279761e-02	7.060363e-03	0.049960	0.057700	
radius_se	569.0	4.051721e-01	2.773127e-01	0.111500	0.232400	
texture_se	569.0	1.216853e+00	5.516484e-01	0.360200	0.833900	
perimeter_se	569.0	2.866059e+00	2.021855e+00	0.757000	1.606000	
area_se	569.0	4.033708e+01	4.549101e+01	6.802000	17.850000	
smoothness_se	569.0	7.040979e-03	3.002518e-03	0.001713	0.005169	
compactness_se	569.0	2.547814e-02	1.790818e-02	0.002252	0.013080	
concavity_se	569.0	3.189372e-02	3.018606e-02	0.000000	0.015090	
concave points_se	569.0	1.179614e-02	6.170285e-03	0.000000	0.007638	
symmetry_se	569.0	2.054230e-02	8.266372e-03	0.007882	0.015160	
fractal_dimension_se	569.0	3.794904e-03	2.646071e-03	0.000895	0.002248	
radius_worst	569.0	1.626919e+01	4.833242e+00	7.930000	13.010000	
texture_worst	569.0	2.567722e+01	6.146258e+00	12.020000	21.080000	
perimeter_worst	569.0	1.072612e+02	3.360254e+01	50.410000	84.110000	
area_worst	569.0	8.805831e+02	5.693570e+02	185.200000	515.300000	6
smoothness_worst	569.0	1.323686e-01	2.283243e-02	0.071170	0.116600	
compactness_worst	569.0	2.542650e-01	1.573365e-01	0.027290	0.147200	
concavity_worst	569.0	2.721885e-01	2.086243e-01	0.000000	0.114500	
concave points_worst	569.0	1.146062e-01	6.573234e-02	0.000000	0.064930	
symmetry_worst	569.0	2.900756e-01	6.186747e-02	0.156500	0.250400	
fractal_dimension_worst	569.0	8.394582e-02	1.806127e-02	0.055040	0.071460	
Unnamed: 32	0.0	NaN	NaN	NaN	NaN	

Getting Unique Values of our Target Variable (Diagnosis):

In [7]:
```
#unique values
df.diagnosis.unique()
```

Out[7]:
```
array(['M', 'B'], dtype=object)
```

We can see that tumors are either classified as M-Malignant, or B-Benign.

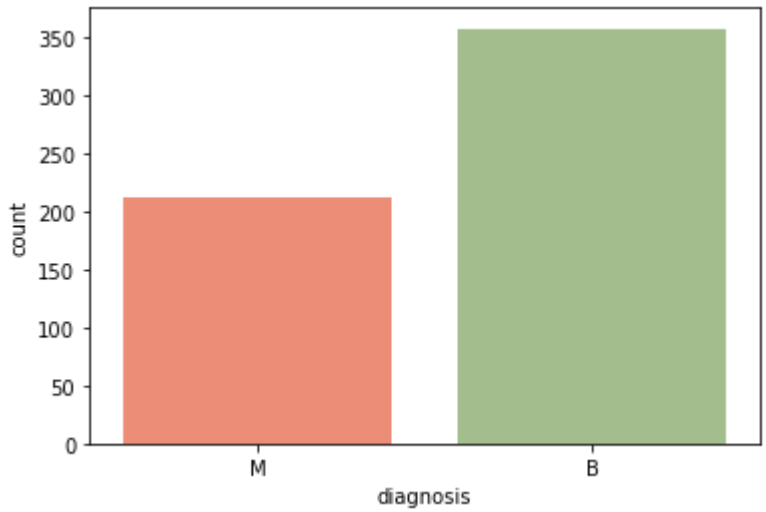
Getting the number of benign vs. malignant classification:

In [8]:
```
#number of malignant vs. benign tumors in the dataframe
df['diagnosis'].value_counts()
```

Out[8]:
```
B    357
M    212
Name: diagnosis, dtype: int64
```

Plotting number of malignant and benign tumors for visual representation:

In [9]:
```
#visual of number of malignant and benign tumors
sns.countplot(df['diagnosis'], palette= ['#FF8164', '#a3c585']);
```



Cleaning and Preparing the Data:

To begin, I am removing the "id" and "Unnamed: 32" columns, as they are not needed for the purposes of this model.

In [10]:
```
#remove column "id" and "Unnamed: 32"
df.drop('id',axis=1,inplace=True)
df.drop('Unnamed: 32',axis=1,inplace=True)
df.head()
```

Out[10]:

	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	cor
0	M	17.99	10.38	122.80	1001.0	0.11840	
1	M	20.57	17.77	132.90	1326.0	0.08474	
2	M	19.69	21.25	130.00	1203.0	0.10960	
3	M	11.42	20.38	77.58	386.1	0.14250	
4	M	20.29	14.34	135.10	1297.0	0.10030	

5 rows × 31 columns

Since Diagnosis is a categorical variable, I will be mapping malignant tumors to a value of 1 and benign tumors to a value of 0 so that we are able to work with the model.

In [11]:
```
#mapping malignant to 1, and benign to 0
df['diagnosis'] = df['diagnosis'].map({'M':1,'B':0})
df.head()
```

Out[11]:

	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	cor
0	1	17.99	10.38	122.80	1001.0	0.11840	
1	1	20.57	17.77	132.90	1326.0	0.08474	
2	1	19.69	21.25	130.00	1203.0	0.10960	
3	1	11.42	20.38	77.58	386.1	0.14250	
4	1	20.29	14.34	135.10	1297.0	0.10030	

5 rows × 31 columns

Checking for any null values:

In [12]:
```
df.isnull().sum()
```

Out[12]:
```
diagnosis                 0
radius_mean               0
texture_mean              0
perimeter_mean            0
area_mean                 0
smoothness_mean           0
compactness_mean          0
concavity_mean            0
concave points_mean       0
symmetry_mean             0
fractal_dimension_mean    0
radius_se                 0
texture_se                0
perimeter_se              0
area_se                   0
smoothness_se             0
compactness_se            0
concavity_se              0
concave points_se         0
symmetry_se               0
fractal_dimension_se      0
radius_worst              0
texture_worst             0
perimeter_worst           0
area_worst                0
smoothness_worst          0
compactness_worst         0
concavity_worst           0
concave points_worst      0
symmetry_worst            0
fractal_dimension_worst   0
dtype: int64
```

There are no null values to deal with, so now I will be checking for collinearity.

Multicollinearity:

Multicollinearity is a problem as it undermines the significance of independent variables. I had a suspicion that it might be present within the variables I had, so I did some additional mapping to see if we needed to exclude any variables from the model.

In [13]:
```
#showing table of collinearity between variables
df.corr()
```

Out[13]:

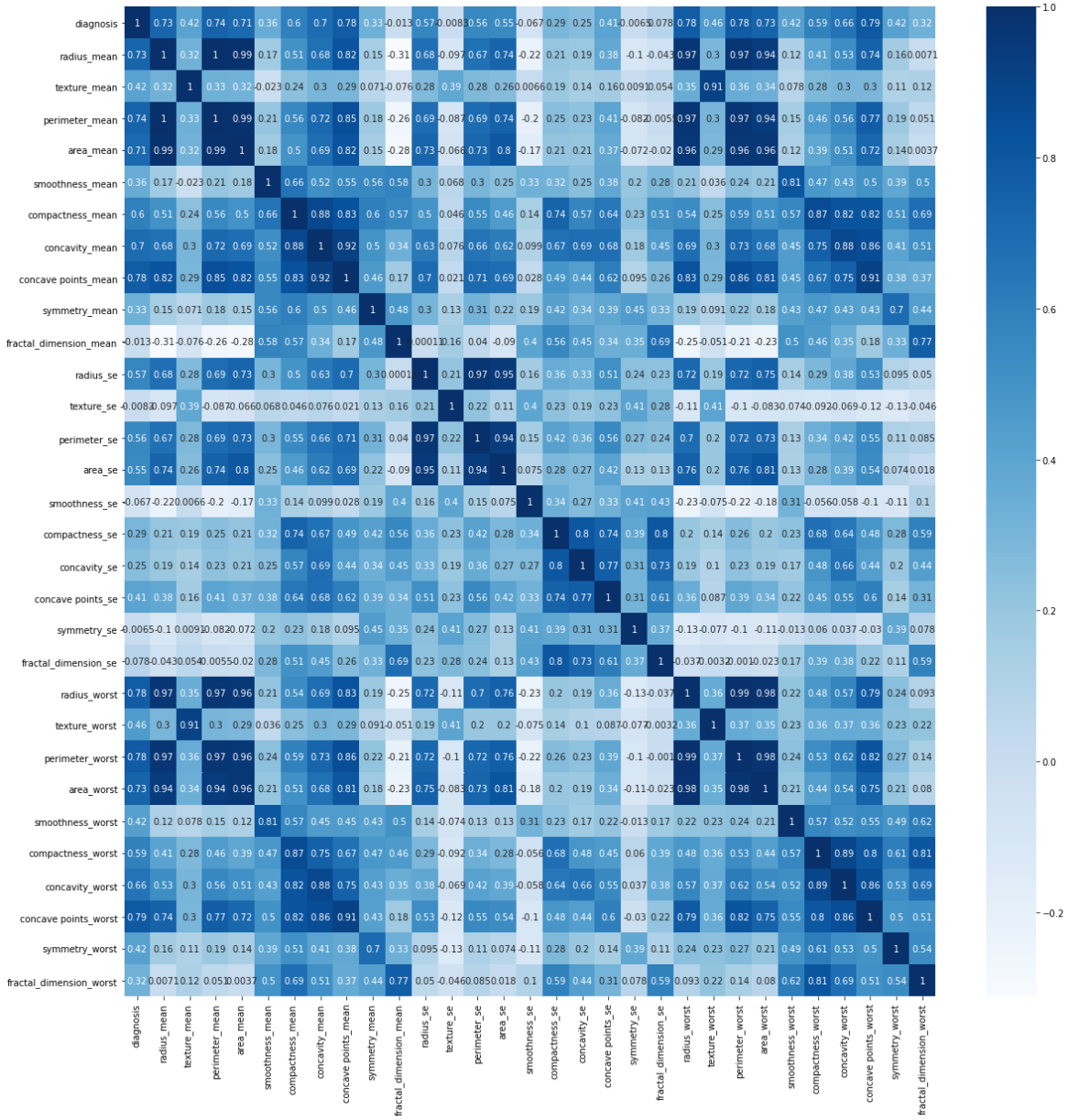
	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	s
diagnosis	1.000000	0.730029	0.415185	0.742636	0.708984	
radius_mean	0.730029	1.000000	0.323782	0.997855	0.987357	
texture_mean	0.415185	0.323782	1.000000	0.329533	0.321086	
perimeter_mean	0.742636	0.997855	0.329533	1.000000	0.986507	
area_mean	0.708984	0.987357	0.321086	0.986507	1.000000	
smoothness_mean	0.358560	0.170581	-0.023389	0.207278	0.177028	
compactness_mean	0.596534	0.506124	0.236702	0.556936	0.498502	
concavity_mean	0.696360	0.676764	0.302418	0.716136	0.685983	
concave points_mean	0.776614	0.822529	0.293464	0.850977	0.823269	
symmetry_mean	0.330499	0.147741	0.071401	0.183027	0.151293	
fractal_dimension_mean	-0.012838	-0.311631	-0.076437	-0.261477	-0.283110	
radius_se	0.567134	0.679090	0.275869	0.691765	0.732562	
texture_se	-0.008303	-0.097317	0.386358	-0.086761	-0.066280	
perimeter_se	0.556141	0.674172	0.281673	0.693135	0.726628	
area_se	0.548236	0.735864	0.259845	0.744983	0.800086	
smoothness_se	-0.067016	-0.222600	0.006614	-0.202694	-0.166777	
compactness_se	0.292999	0.206000	0.191975	0.250744	0.212583	
concavity_se	0.253730	0.194204	0.143293	0.228082	0.207660	
concave points_se	0.408042	0.376169	0.163851	0.407217	0.372320	
symmetry_se	-0.006522	-0.104321	0.009127	-0.081629	-0.072497	
fractal_dimension_se	0.077972	-0.042641	0.054458	-0.005523	-0.019887	
radius_worst	0.776454	0.969539	0.352573	0.969476	0.962746	
texture_worst	0.456903	0.297008	0.912045	0.303038	0.287489	
perimeter_worst	0.782914	0.965137	0.358040	0.970387	0.959120	
area_worst	0.733825	0.941082	0.343546	0.941550	0.959213	
smoothness_worst	0.421465	0.119616	0.077503	0.150549	0.123523	
compactness_worst	0.590998	0.413463	0.277830	0.455774	0.390410	
concavity_worst	0.659610	0.526911	0.301025	0.563879	0.512606	
concave points_worst	0.793566	0.744214	0.295316	0.771241	0.722017	
symmetry_worst	0.416294	0.163953	0.105008	0.189115	0.143570	
fractal_dimension_worst	0.323872	0.007066	0.119205	0.051019	0.003738	

31 rows × 31 columns

Plotting collinearity as a heatmap for better visibility:

In [14]:
```
#heatmap of collinearity
plt.figure(figsize=(20,20))
sns.heatmap(df.corr(), annot=True, cmap="Blues" )
```

Out[14]: <AxesSubplot:>

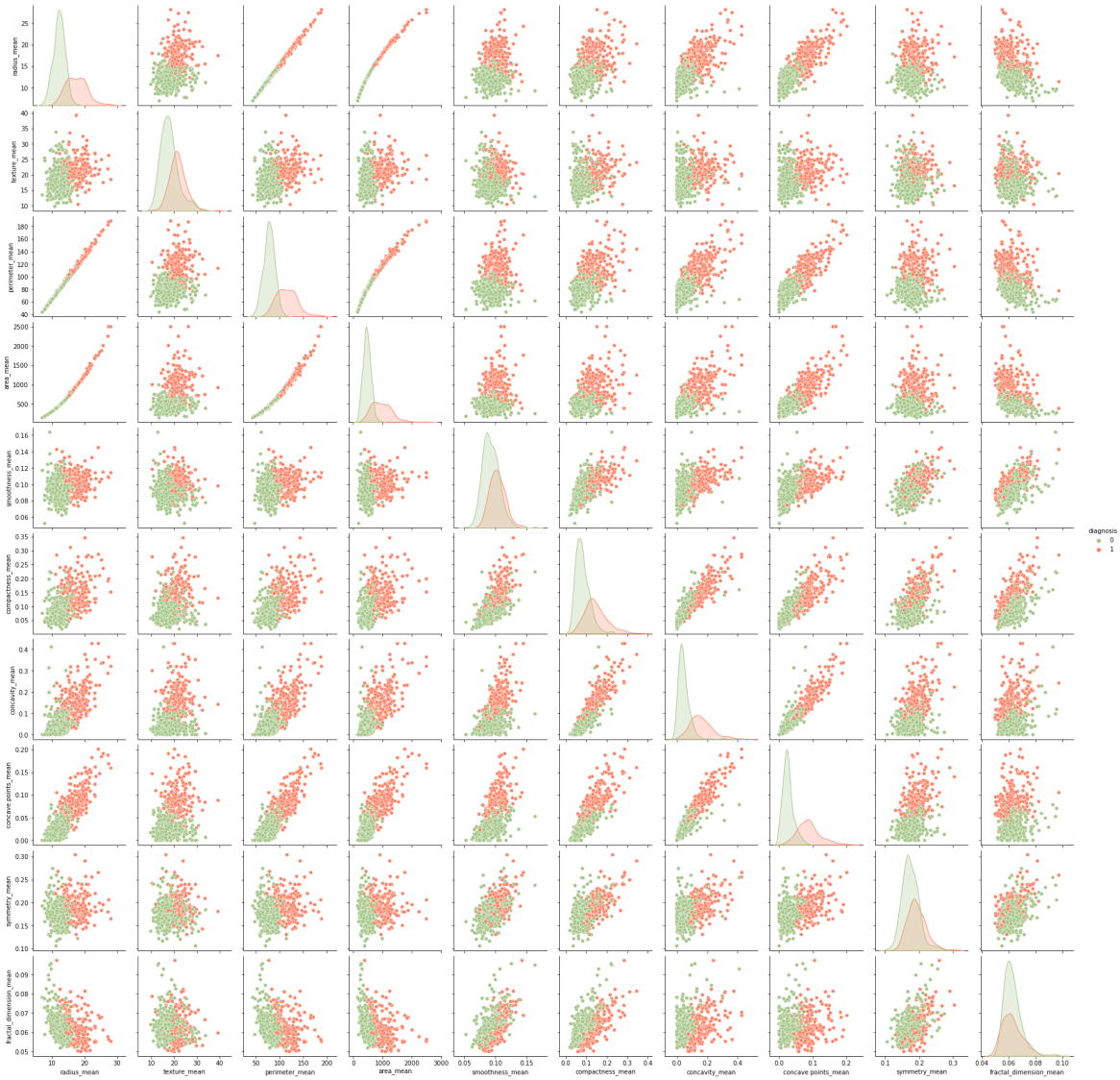


From this, I am noticing that a few of the columns appear to have collinearity. Multicollinearity is a problem as it undermines the significance of independent variables, so I will do additional mapping to see if we need to exclude any of these variables from the model.

Checking the "mean" columns:

In [15]:
```
# Generate a scatter plot matrix with the "mean" columns
means = ['diagnosis',
         'radius_mean',
         'texture_mean',
         'perimeter_mean',
         'area_mean',
         'smoothness_mean',
         'compactness_mean',
         'concavity_mean',
         'concave points_mean',
         'symmetry_mean',
         'fractal_dimension_mean']

sns.pairplot(data=df[means], hue='diagnosis', palette= ['#a3c585','#FF8164'])
```

Out[15]: <seaborn.axisgrid.PairGrid at 0x1981ad8b820>



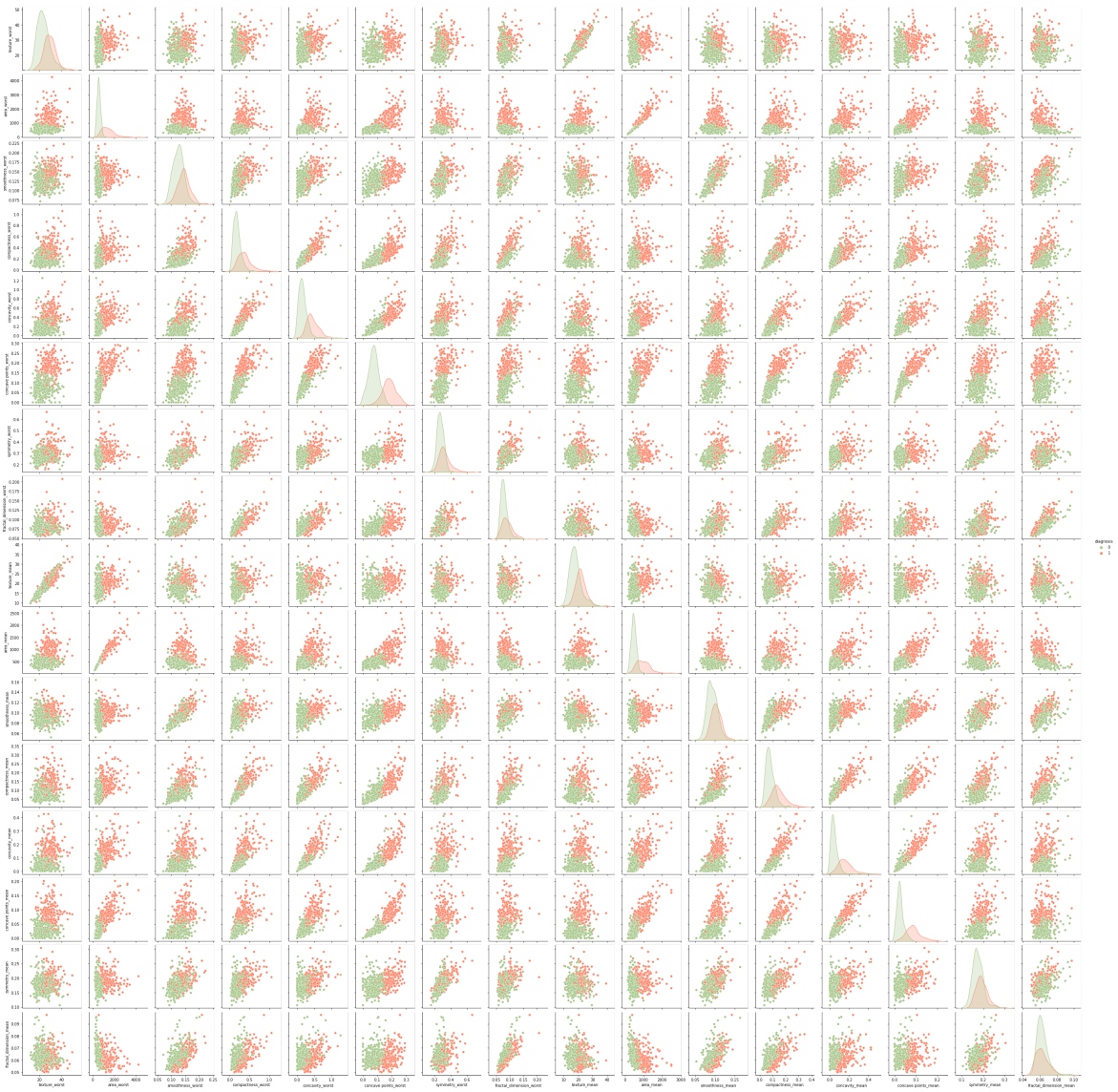
There is a close correlation between the radius_mean column with perimeter_mean and area_mean columns. This is likely because the three columns contain very similar information: the physical size of the the tumor/cell. This means we should only pick one of the columns for analysis,

and I will be removing all columns related to perimeter or radius (and just keeping the area columns).

Checking the "mean" vs. "worst" columns:

In [16]:
```
# Generate a scatter plot matrix with the "mean" and "worst" columns
meansWorst = ['diagnosis',
        'texture_worst',
        'area_worst',
        'smoothness_worst',
        'compactness_worst',
        'concavity_worst',
        'concave points_worst',
        'symmetry_worst',
        'fractal_dimension_worst',
        'texture_mean',
        'area_mean',
        'smoothness_mean',
        'compactness_mean',
        'concavity_mean',
        'concave points_mean',
        'symmetry_mean',
        'fractal_dimension_mean']

sns.pairplot(data=df[meansWorst], hue='diagnosis', palette= ['#a3c585','#FF81
```

Out[16]: <seaborn.axisgrid.PairGrid at 0x1981f654f40>



From this plot, we can see there is a strong correlation between the "worst" and the "mean" columns. Because of that, I will be removing the "worst" columns and keeping the "mean" columns.

Checking the remaining columns:

In [17]:
```
# Generate a scatter plot matrix with the remaining columns
meansFinal= ['diagnosis',
        'texture_mean',
        'area_mean',
        'smoothness_mean',
        'compactness_mean',
        'concavity_mean',
        'concave points_mean',
        'symmetry_mean',
        'fractal_dimension_mean',
        'radius_se',
        'texture_se',
        'smoothness_se',
        'compactness_se',
       'symmetry_se',
        'fractal_dimension_se']

sns.pairplot(data=df[meansFinal], hue='diagnosis', palette= ['#a3c585','#FF81
```

Out[17]: <seaborn.axisgrid.PairGrid at 0x1982b097e20>

From this, I am noticing that compactness, concave_points, and and concavity all seem to be

similar as well. I will be keeping compactness and removing the other two columns.

Cleaning up the Columns:

Dropping all the columns that were mentioned above:

In [18]:
```
#dropping "worst" columns
columns = [
        'texture_worst',
        'area_worst',
        'radius_worst',
        'smoothness_worst',
        'perimeter_worst',
        'compactness_worst',
        'concavity_worst',
        'concave points_worst',
        'symmetry_worst',
        'fractal_dimension_worst']
df = df.drop(columns, axis=1)
```

In [19]:
```
#dropping "perimeter" and "radius" columns
columns = [
        'perimeter_mean',
        'perimeter_se',
        'radius_mean',
        'radius_se']
df = df.drop(columns, axis=1)
```

In [20]:
```
#dropping "concave points" and "concavity" columns
columns = [
        'concavity_mean',
        'concavity_se',
        'concave points_mean',
        'concave points_se']
df = df.drop(columns, axis=1)
```

Final Columns:

In [21]:
```
#show finalized column selection
df.columns
```

Out[21]:
```
Index(['diagnosis', 'texture_mean', 'area_mean', 'smoothness_mean',
       'compactness_mean', 'symmetry_mean', 'fractal_dimension_mean',
       'texture_se', 'area_se', 'smoothness_se', 'compactness_se',
       'symmetry_se', 'fractal_dimension_se'],
      dtype='object')
```

Additional Preprocessing:

Splitting the target variable:

In [22]:
```
#splitting the target variable from the rest of the variables
X = df.drop(['diagnosis'],axis=1)
y = df['diagnosis']
```

Splitting into test and training data:

In [23]:
```
#splitting data into test and train data, establishing a randomized state
SEED = 5
X_train, X_test,y_train, y_test = train_test_split(X, y, random_state=SEED)
```

Feature Scaling:

Scaling features can improve the optimization process by making the flow of gradient descent smoother and helping algorithms reach the minimum of the cost function faster. We scale so that the algorithm is not biased towards the features with values higher in magnitude. I played around with the MinMax Scaler, but found that the Standard Scaler helped the model perform more accurately.

In [24]:
```
#scaling the data
ss = StandardScaler()
X_train = ss.fit_transform(X_train)
X_test = ss.transform(X_test)
```

Building the Model

Baseline Model:

Training the data to a logistic regression, and predicting on the test data

In [25]:
```
blr = LogisticRegression()
modelBLR=blr.fit(X_train,y_train)
predictionBLR=modelBLR.predict(X_test)
```

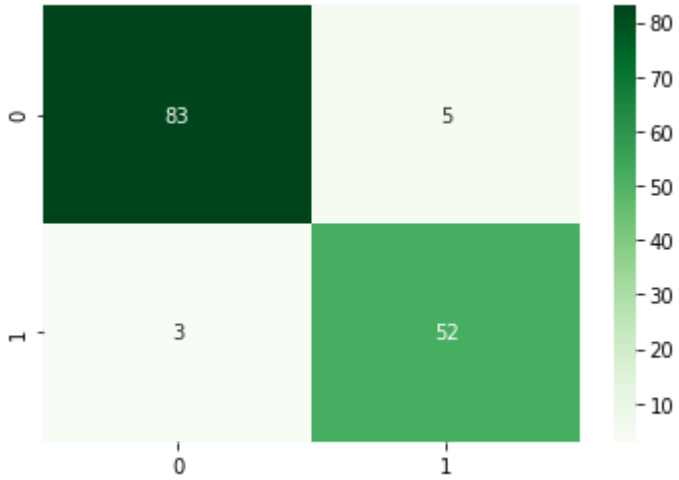
Show in Confusion Matrix:

In [26]:
```
blrmatrix = confusion_matrix(y_test, predictionBLR)
blrmatrix
```

Out[26]:
```
array([[83,  5],
       [ 3, 52]], dtype=int64)
```

In [27]:
```
#Heatmapping
sns.heatmap(blrmatrix, annot=True, cmap="Greens")
```

Out[27]: <AxesSubplot:>



This shows us that out of 143 results, only 3 were classified as a false negative, which is approximately 2.1%.

Next, we will get a classification report of the trained model, so we can better see how it performed. As mentioned above, we are trying to get a high accuracy rate for the model, and more importantly trying to minimize the number of false negatives. Recall looks at the number of false negatives that were thrown into the prediction mix. The recall rate is penalized whenever a false negative is predicted.

In [29]:
```
y_hat_train = modelBLR.predict(X_train)
y_hat_test = modelBLR.predict(X_test)
```

In [30]:
```
cr = classification_report(y_test,y_hat_test)
print(cr)
```

```
              precision    recall  f1-score   support

           0       0.97      0.94      0.95        88
           1       0.91      0.95      0.93        55

    accuracy                           0.94       143
   macro avg       0.94      0.94      0.94       143
weighted avg       0.94      0.94      0.94       143
```

As we can see from the classification report, the recall rate for malignant tumors was 95%, which is overall a pretty good score (the closer to 100% the better the model performs). I will try and look at additional models to see if we can improve the recall and accuracy of the model.

I'm going to look at the training vs testing performance now:

In [31]:
```
print('Training Precision: ', precision_score(y_train, y_hat_train))
print('Testing Precision: ', precision_score(y_test, y_hat_test))
print('\n')

print('Training Recall: ', recall_score(y_train, y_hat_train))
print('Testing Recall: ', recall_score(y_test, y_hat_test))
print('\n')

print('Training Accuracy: ', accuracy_score(y_train, y_hat_train))
print('Testing Accuracy: ', accuracy_score(y_test, y_hat_test))
print('\n')

print('Training F1-Score: ', f1_score(y_train, y_hat_train))
print('Testing F1-Score: ', f1_score(y_test, y_hat_test))
```

```
Training Precision:  0.9403973509933775
Testing Precision:  0.9122807017543859


Training Recall:  0.9044585987261147
Testing Recall:  0.9454545454545454


Training Accuracy:  0.9436619718309859
Testing Accuracy:  0.9440559440559441


Training F1-Score:  0.922077922077922
Testing F1-Score:  0.9285714285714285
```

It does not appear like there is any clear overfitting in the model, which is a good sign.

This model actually performed very well! It had a relatively high accuracy rate, and there were a low level of false negatives (recall). I will continue to look at other models to see if can improve the performance.

Logistic Regression Model:

Performing a GridSearch to tune the hyperparameters for optimal model performance.

In [32]:
```
# Grid search cross validation

grid = {"C":np.logspace(-3,3,7), "penalty":["l1","l2"]}# l1 lasso l2 ridge
logreg = LogisticRegression()
logreg_cv = GridSearchCV(logreg,grid,cv=10)
logreg_cv.fit(X_train,y_train)

print("Tuned hyperparameters: ",logreg_cv.best_params_)
print("Accuracy :",logreg_cv.best_score_)
```

```
Tuned hyperparameters:  {'C': 100.0, 'penalty': 'l2'}
Accuracy : 0.93421926910299
```

Training the data to a logistic regression with the tuned hyperparameters, and predicting on the test data

In [33]:
```
lr = LogisticRegression(C=100, penalty="l2")
modelLR=lr.fit(X_train,y_train)
predictionLR=modelLR.predict(X_test)
```

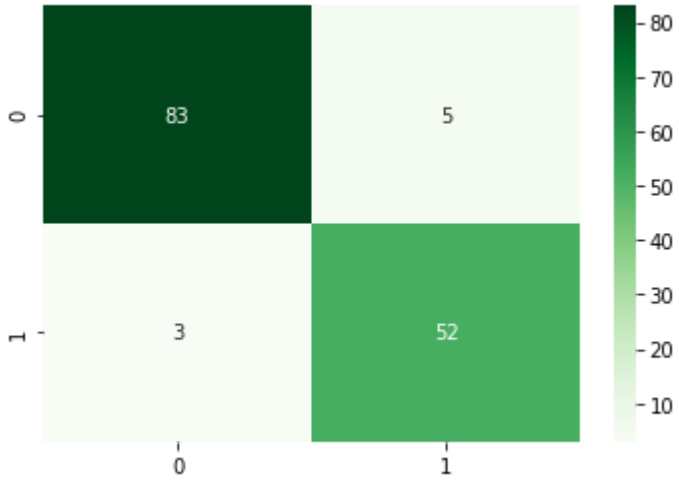
Show in Confusion Matrix:

In [34]:
```
lrmatrix = confusion_matrix(y_test, predictionLR)
lrmatrix
```

Out[34]:
```
array([[83,  5],
       [ 3, 52]], dtype=int64)
```

In [35]:
```
#Heatmapping
sns.heatmap(lrmatrix, annot=True, cmap="Greens")
```

Out[35]:
```
<AxesSubplot:>
```



In [36]:
```
y_hat_train = modelLR.predict(X_train)
y_hat_test = modelLR.predict(X_test)
```

Getting the Classification Report:

In [37]:
```
cr = classification_report(y_test,y_hat_test)
print(cr)
```

```
              precision    recall  f1-score   support

           0       0.97      0.94      0.95        88
           1       0.91      0.95      0.93        55

    accuracy                           0.94       143
   macro avg       0.94      0.94      0.94       143
weighted avg       0.94      0.94      0.94       143
```

I'm going to look at the training vs testing performance now:

In [38]:
```
print('Training Precision: ', precision_score(y_train, y_hat_train))
print('Testing Precision: ', precision_score(y_test, y_hat_test))
print('\n')

print('Training Recall: ', recall_score(y_train, y_hat_train))
print('Testing Recall: ', recall_score(y_test, y_hat_test))
print('\n')

print('Training Accuracy: ', accuracy_score(y_train, y_hat_train))
print('Testing Accuracy: ', accuracy_score(y_test, y_hat_test))
print('\n')

print('Training F1-Score: ', f1_score(y_train, y_hat_train))
print('Testing F1-Score: ', f1_score(y_test, y_hat_test))
```

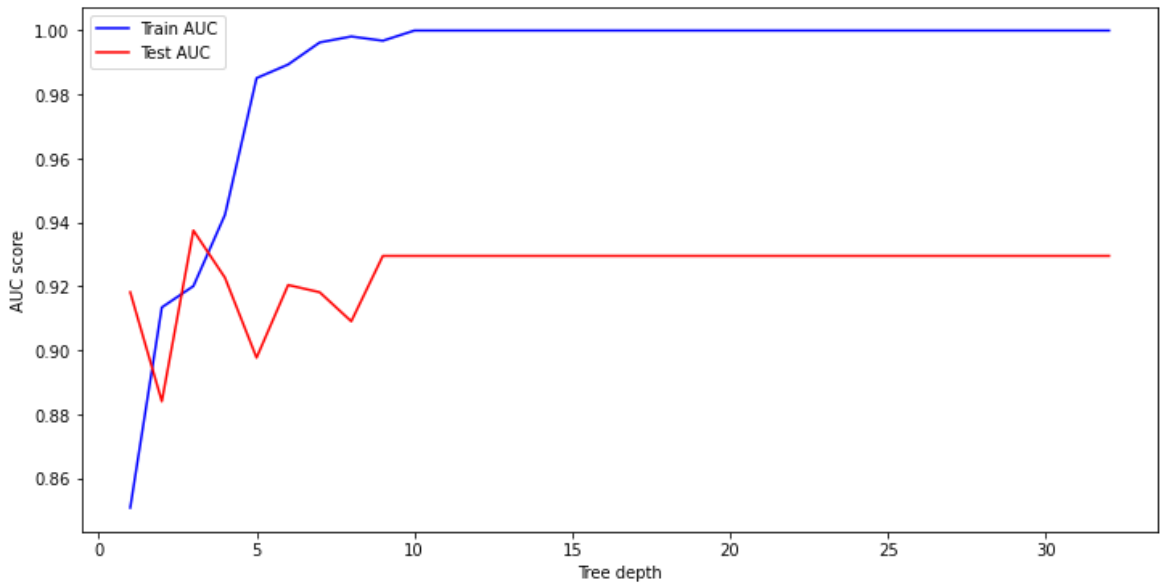
```
Training Precision:  0.934640522875817
Testing Precision:  0.9122807017543859


Training Recall:  0.910828025477707
Testing Recall:  0.9454545454545454


Training Accuracy:  0.9436619718309859
Testing Accuracy:  0.9440559440559441


Training F1-Score:  0.9225806451612903
Testing F1-Score:  0.9285714285714285
```

From this, we can see that the tuned model actually fit just the same as the baseline model. So for simplicity's sake, I would argue that the baseline model would be better to use. I will keep looking at other models to see if we can improve the performance.

Decision Tree:

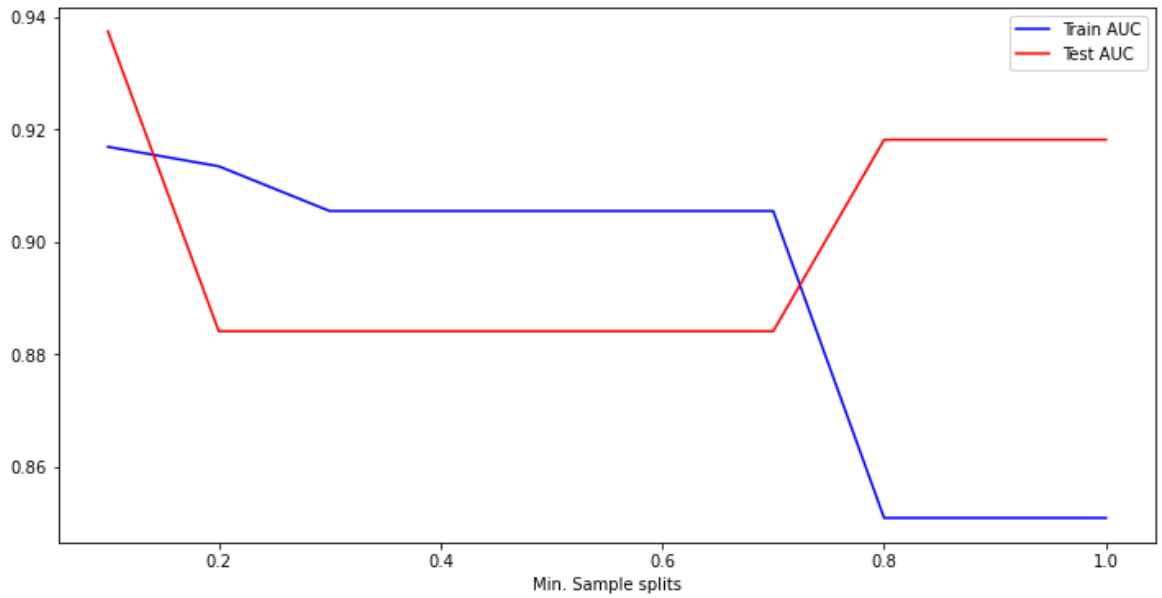
Next, I will try running a decision tree model to see if that will outperform the linear regression model. I will be performing a series of tests to try and tune the parameters. I will be looking at optimal tree depth, min-samples-split, minimum sample leafs, and maximum feature size.

In [39]:
```
# Identify the optimal tree depth for given data
max_depths = list(range(1, 33))
train_results = []
test_results = []
for max_depth in max_depths:
    dt = DecisionTreeClassifier(criterion='entropy', max_depth=max_depth, ran
    dt.fit(X_train, y_train)
    train_pred = dt.predict(X_train)
    false_positive_rate, true_positive_rate, thresholds = roc_curve(y_train,
    roc_auc = auc(false_positive_rate, true_positive_rate)
    # Add auc score to previous train results
    train_results.append(roc_auc)
    y_pred = dt.predict(X_test)
    false_positive_rate, true_positive_rate, thresholds = roc_curve(y_test, y
    roc_auc = auc(false_positive_rate, true_positive_rate)
    # Add auc score to previous test results
    test_results.append(roc_auc)

plt.figure(figsize=(12,6))
plt.plot(max_depths, train_results, 'b', label='Train AUC')
plt.plot(max_depths, test_results, 'r', label='Test AUC')
plt.ylabel('AUC score')
plt.xlabel('Tree depth')
plt.legend()
plt.show()
```



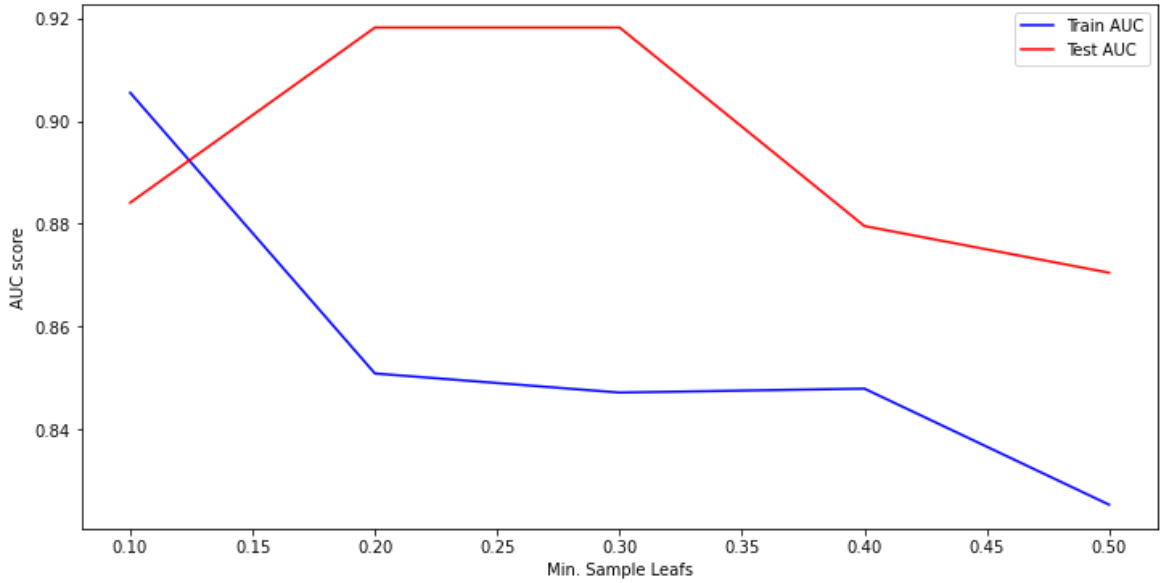
In [40]:

```
# Identify the optimal min-samples-split for given data
min_samples_splits = np.linspace(0.1, 1.0, 10, endpoint=True)
train_results = []
test_results = []
for min_samples_split in min_samples_splits:
    dt = DecisionTreeClassifier(criterion='entropy', min_samples_split=min_sa
    dt.fit(X_train, y_train)
    train_pred = dt.predict(X_train)
    false_positive_rate, true_positive_rate, thresholds =    roc_curve(y_trai
    roc_auc = auc(false_positive_rate, true_positive_rate)
    train_results.append(roc_auc)
    y_pred = dt.predict(X_test)
    false_positive_rate, true_positive_rate, thresholds = roc_curve(y_test, y
    roc_auc = auc(false_positive_rate, true_positive_rate)
    test_results.append(roc_auc)

plt.figure(figsize=(12,6))
plt.plot(min_samples_splits, train_results, 'b', label='Train AUC')
plt.plot(min_samples_splits, test_results, 'r', label='Test AUC')
plt.xlabel('Min. Sample splits')
plt.legend()
plt.show()
```

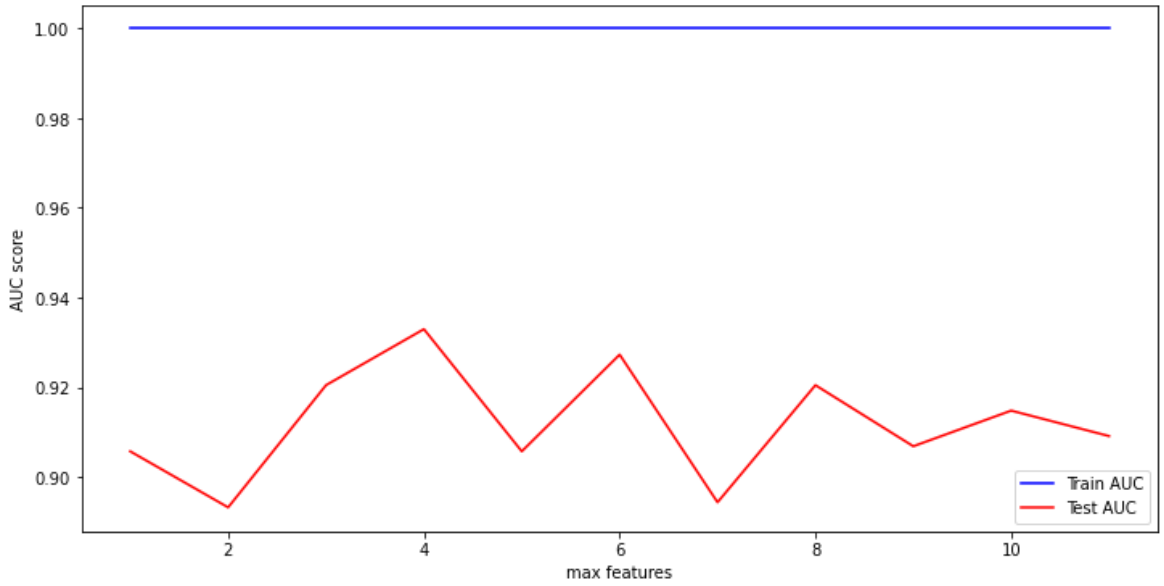


In [41]:

```
# Calculate the optimal value for minimum sample leafs
min_samples_leafs = np.linspace(0.1, 0.5, 5, endpoint=True)
train_results = []
test_results = []
for min_samples_leaf in min_samples_leafs:
    dt = DecisionTreeClassifier(criterion='entropy', min_samples_leaf=min_sam
    dt.fit(X_train, y_train)
    train_pred = dt.predict(X_train)
    false_positive_rate, true_positive_rate, thresholds = roc_curve(y_train,
    roc_auc = auc(false_positive_rate, true_positive_rate)
    train_results.append(roc_auc)
    y_pred = dt.predict(X_test)
    false_positive_rate, true_positive_rate, thresholds = roc_curve(y_test, y
    roc_auc = auc(false_positive_rate, true_positive_rate)
    test_results.append(roc_auc)

plt.figure(figsize=(12,6))
plt.plot(min_samples_leafs, train_results, 'b', label='Train AUC')
plt.plot(min_samples_leafs, test_results, 'r', label='Test AUC')
plt.ylabel('AUC score')
plt.xlabel('Min. Sample Leafs')
plt.legend()
plt.show()
```



In [42]:
```
# Find the best value for optimal maximum feature size
max_features = list(range(1, X_train.shape[1]))
train_results = []
test_results = []
for max_feature in max_features:
    dt = DecisionTreeClassifier(criterion='entropy', max_features=max_feature
    dt.fit(X_train, y_train)
    train_pred = dt.predict(X_train)
    false_positive_rate, true_positive_rate, thresholds = roc_curve(y_train,
    roc_auc = auc(false_positive_rate, true_positive_rate)
    train_results.append(roc_auc)
    y_pred = dt.predict(X_test)
    false_positive_rate, true_positive_rate, thresholds = roc_curve(y_test, y
    roc_auc = auc(false_positive_rate, true_positive_rate)
    test_results.append(roc_auc)

plt.figure(figsize=(12,6))
plt.plot(max_features, train_results, 'b', label='Train AUC')
plt.plot(max_features, test_results, 'r', label='Test AUC')
plt.ylabel('AUC score')
plt.xlabel('max features')
plt.legend()
plt.show()
```



From these tests, I estimated that the optimal values were as seen below. I will then train a classifier:

In [43]:
```
# Train a classifier with optimal values identified above
dt = DecisionTreeClassifier(criterion='entropy',
                            max_features=3,
                            max_depth=2,
                            min_samples_split=0.35,
                            min_samples_leaf=0.11,
                            random_state=SEED)
dt.fit(X_train, y_train)
y_pred = dt.predict(X_test)
false_positive_rate, true_positive_rate, thresholds = roc_curve(y_test, y_pre
roc_auc = auc(false_positive_rate, true_positive_rate)
roc_auc
```

Out[43]:
```
0.7590909090909091
```

Using the trained model, I will then make predictions:

In [44]:
```
# Make predictions using test set
y_pred = dt.predict(X_test)
```

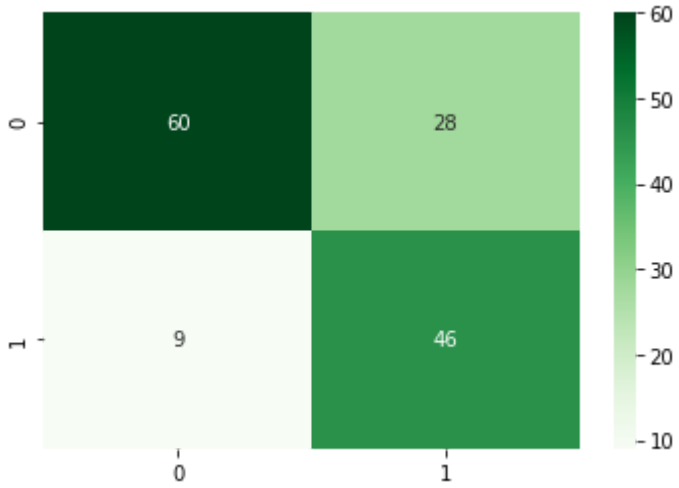
Showing the predictions in a confusion matrix:

In [45]:
```
#show decision tree prediction in matrix
dtmatrix= confusion_matrix(y_test,y_pred)
dtmatrix
```

Out[45]:
```
array([[60, 28],
       [ 9, 46]], dtype=int64)
```

In [46]:
```
sns.heatmap(dtmatrix, annot=True, cmap="Greens")
```

Out[46]:
```
<AxesSubplot:>
```



As we can see here, the decision tree model performed considerably worse than the logistic regression model. The number of false negatives is up 3 times that of the logistic regression model.

In [47]:
```
y_hat_train = dt.predict(X_train)
y_hat_test = dt.predict(X_test)
```

In [48]:
```
print('Training Precision: ', precision_score(y_train, y_hat_train))
print('Testing Precision: ', precision_score(y_test, y_hat_test))
print('\n')

print('Training Recall: ', recall_score(y_train, y_hat_train))
print('Testing Recall: ', recall_score(y_test, y_hat_test))
print('\n')

print('Training Accuracy: ', accuracy_score(y_train, y_hat_train))
print('Testing Accuracy: ', accuracy_score(y_test, y_hat_test))
print('\n')

print('Training F1-Score: ', f1_score(y_train, y_hat_train))
print('Testing F1-Score: ', f1_score(y_test, y_hat_test))
```

```
Training Precision:  0.689119170984456
Testing Precision:  0.6216216216216216


Training Recall:  0.8471337579617835
Testing Recall:  0.8363636363636363


Training Accuracy:  0.8028169014084507
Testing Accuracy:  0.7412587412587412


Training F1-Score:  0.76
Testing F1-Score:  0.7131782945736433
```

Again, we can see that this model performed considerably worse, with recall only at 83.6%.

I'm now going to run a decision tree model without doing any tuning to see if that performs better. First I am training the classifier:

In [49]:
```
# Train the classifier using training data
dte = DecisionTreeClassifier(criterion='entropy', random_state=SEED)
dte.fit(X_train, y_train)
```

Out[49]:
```
DecisionTreeClassifier(criterion='entropy', random_state=5)
```

Next I will make predictions using the test set:

In [50]:
```
# Make predictions using test set
y_predi = dte.predict(X_test)
```

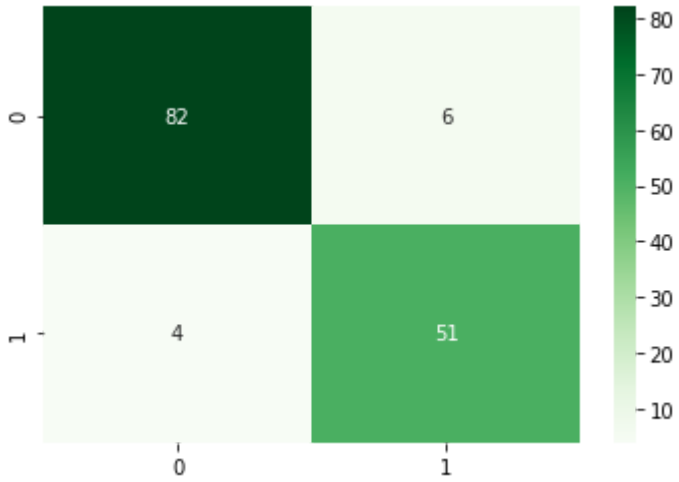
Now I will show the results in a confusion matrix:

In [51]:
```
#show decision tree prediction in matrix
dtematrix= confusion_matrix(y_test,y_predi)
dtematrix
```

Out[51]:
```
array([[82,  6],
       [ 4, 51]], dtype=int64)
```

In [52]:
```
sns.heatmap(dtematrix, annot=True, cmap = "Greens")
```

Out[52]:
```
<AxesSubplot:>
```



This model performed better than the tuned decision tree model, so this means I must have not tuned the model correctly, there is more work that could be done there to get a better performing model. Although this model performed better than my first decision tree model, it still did not outperform the logistic regression model I initially performed.

In [53]:
```
y_hat_train = dte.predict(X_train)
y_hat_test = dte.predict(X_test)
```

In [54]:
```
print('Training Precision: ', precision_score(y_train, y_hat_train))
print('Testing Precision: ', precision_score(y_test, y_hat_test))
print('\n')

print('Training Recall: ', recall_score(y_train, y_hat_train))
print('Testing Recall: ', recall_score(y_test, y_hat_test))
print('\n')

print('Training Accuracy: ', accuracy_score(y_train, y_hat_train))
print('Testing Accuracy: ', accuracy_score(y_test, y_hat_test))
print('\n')

print('Training F1-Score: ', f1_score(y_train, y_hat_train))
print('Testing F1-Score: ', f1_score(y_test, y_hat_test))
```

```
Training Precision:  1.0
Testing Precision:  0.8947368421052632


Training Recall:  1.0
Testing Recall:  0.9272727272727272


Training Accuracy:  1.0
Testing Accuracy:  0.9300699300699301


Training F1-Score:  1.0
Testing F1-Score:  0.9107142857142856
```

We can see that this model is overfit, because it is performed perfectly on the training data, but not nearly as well in the testing data.

Random Forest Model:

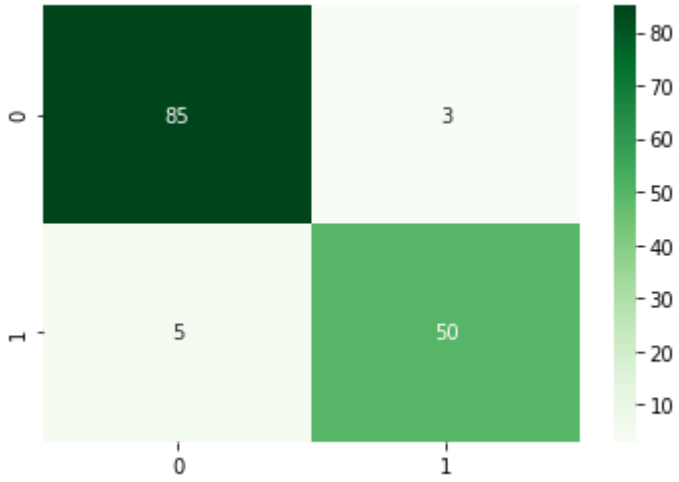
Next, I will run a basic random forest model. I trained the classifier and ran my predictions as I did for the above models.

In [55]:
```
randomForest=RandomForestClassifier()
randomForestModel = randomForest.fit(X_train, y_train)
predictionRM = randomForestModel.predict(X_test)
rfmatrix = confusion_matrix(y_test, predictionRM)
```

In [56]:
```
y_hat_train = randomForestModel.predict(X_train)
y_hat_test = randomForestModel.predict(X_test)
```

In [57]:
```
sns.heatmap(rfmatrix, annot=True, cmap = "Greens")
```

Out[57]: <AxesSubplot:>



We can see that this model still did not outperform the logistic regression model.

In [58]:
```
print('Training Precision: ', precision_score(y_train, y_hat_train))
print('Testing Precision: ', precision_score(y_test, y_hat_test))
print('\n')

print('Training Recall: ', recall_score(y_train, y_hat_train))
print('Testing Recall: ', recall_score(y_test, y_hat_test))
print('\n')

print('Training Accuracy: ', accuracy_score(y_train, y_hat_train))
print('Testing Accuracy: ', accuracy_score(y_test, y_hat_test))
print('\n')

print('Training F1-Score: ', f1_score(y_train, y_hat_train))
print('Testing F1-Score: ', f1_score(y_test, y_hat_test))
```

```
Training Precision:  1.0
Testing Precision:  0.9433962264150944


Training Recall:  1.0
Testing Recall:  0.9090909090909091


Training Accuracy:  1.0
Testing Accuracy:  0.9440559440559441


Training F1-Score:  1.0
Testing F1-Score:  0.9259259259259259
```

Same as the last model, we can see that this model is overfit because it is performed perfectly on the training data, but not nearly as well in the testing data.

Conclusion:

In the end, our best performing model was the Logistic Regression model in the beggining with a recall of 95%. Next steps to improve this research would be to create pipelines to streamline the model creations. We would need to do more research on each model and tune the hyperparameters so that we can minimize the amount of false negatives, and improve the overall accuracy.