Feature selection and data visualization

Content

Import Libraries

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

import time
from subprocess import check_output
print(check_output(["ls", "../input"]).decode("utf8"))
#import warnings library
import warnings
# ignore all warnings
warnings.filterwarnings('ignore')
```

```
data.csv
```

Read Data

In [2]:
```
data = pd.read_csv('../input/data.csv')
```

look at features of data.

In [3]:
```
data.head()
```

Out[3]:

	id	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean	concavity_mean	poi
0	842302	M	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.3001	
1	842517	M	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.0869	
2	84300903	M	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.1974	
3	84348301	M	11.42	20.38	77.58	386.1	0.14250	0.28390	0.2414	
4	84358402	M	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.1980	

5 rows × 33 columns

1) There is an **id** that cannot be used for classificaiton

2) **Diagnosis** is class label

3) **Unnamed: 32** feature includes NaN

Therefore, drop these unnecessary features.

In [4]:
```
# feature names as a list
col = data.columns       # .columns gives columns names in data
print(col)
```

```
Index(['id', 'diagnosis', 'radius_mean', 'texture_mean', 'perimeter_mean',
       'area_mean', 'smoothness_mean', 'compactness_mean', 'concavity_mean',
       'concave points_mean', 'symmetry_mean', 'fractal_dimension_mean',
       'radius_se', 'texture_se', 'perimeter_se', 'area_se', 'smoothness_se',
       'compactness_se', 'concavity_se', 'concave points_se', 'symmetry_se',
       'fractal_dimension_se', 'radius_worst', 'texture_worst',
       'perimeter_worst', 'area_worst', 'smoothness_worst',
       'compactness_worst', 'concavity_worst', 'concave points_worst',
       'symmetry_worst', 'fractal_dimension_worst', 'Unnamed: 32'],
      dtype='object')
```

In [5]:
```
# y includes our labels and x includes our features
y = data.diagnosis                          # M or B
list = ['Unnamed: 32','id','diagnosis']
x = data.drop(list,axis = 1 )
x.head()
```

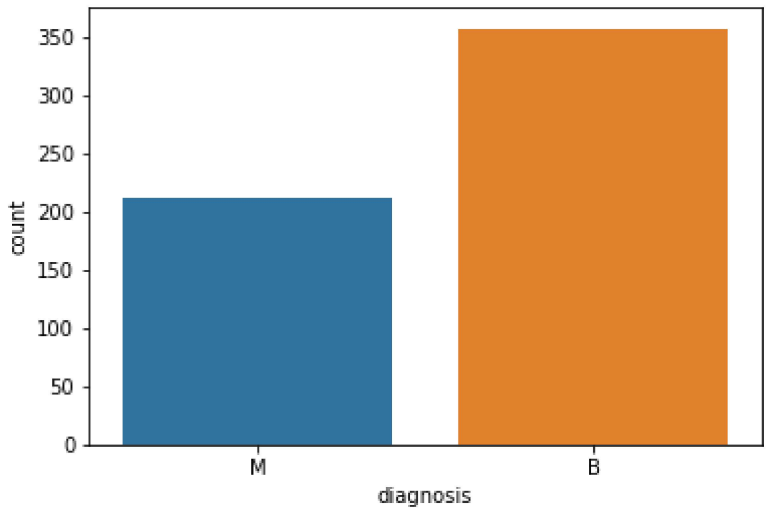
Out[5]:

	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean	concavity_mean	concave points_mean	symmetry_m
0	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.3001	0.14710	0
1	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.0869	0.07017	0
2	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.1974	0.12790	0
3	11.42	20.38	77.58	386.1	0.14250	0.28390	0.2414	0.10520	0
4	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.1980	0.10430	0

5 rows × 30 columns

In [6]:
```
ax = sns.countplot(y,label="Count")       # M = 212, B = 357
B, M = y.value_counts()
print('Number of Benign: ',B)
print('Number of Malignant : ',M)
```

```
Number of Benign:  357
Number of Malignant :  212
```



We have features, but what do they mean? Or, actually, how much do we need to know about these features? The answer is that we do not necessarily need to know the specific meaning of these features. However, to conceptualize them, it's helpful to understand certain attributes like variance, standard deviation, the number of samples (count), or the maximum and minimum values. This type of information aids in grasping the data's characteristics.

For instance, a question might arise: the 'area_mean' feature has a maximum value of 2500, and the 'smoothness_mean' feature has a maximum of 0.16340. Therefore, do we need standardization or normalization before visualization, feature selection, feature extraction, or classification? The answer is both yes and no – not surprising, right?

Anyway, let's progress step by step, beginning with visualization.

In [7]:
```
x.describe()
```

Out[7]:

	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean	concavity_mean	concave points_mean	symm
count	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	
mean	14.127292	19.289649	91.969033	654.889104	0.096360	0.104341	0.088799	0.048919	
std	3.524049	4.301036	24.298981	351.914129	0.014064	0.052813	0.079720	0.038803	
min	6.981000	9.710000	43.790000	143.500000	0.052630	0.019380	0.000000	0.000000	
25%	11.700000	16.170000	75.170000	420.300000	0.086370	0.064920	0.029560	0.020310	
50%	13.370000	18.840000	86.240000	551.100000	0.095870	0.092630	0.061540	0.033500	
75%	15.780000	21.800000	104.100000	782.700000	0.105300	0.130400	0.130700	0.074000	
max	28.110000	39.280000	188.500000	2501.000000	0.163400	0.345400	0.426800	0.201200	

8 rows × 30 columns

Visualization

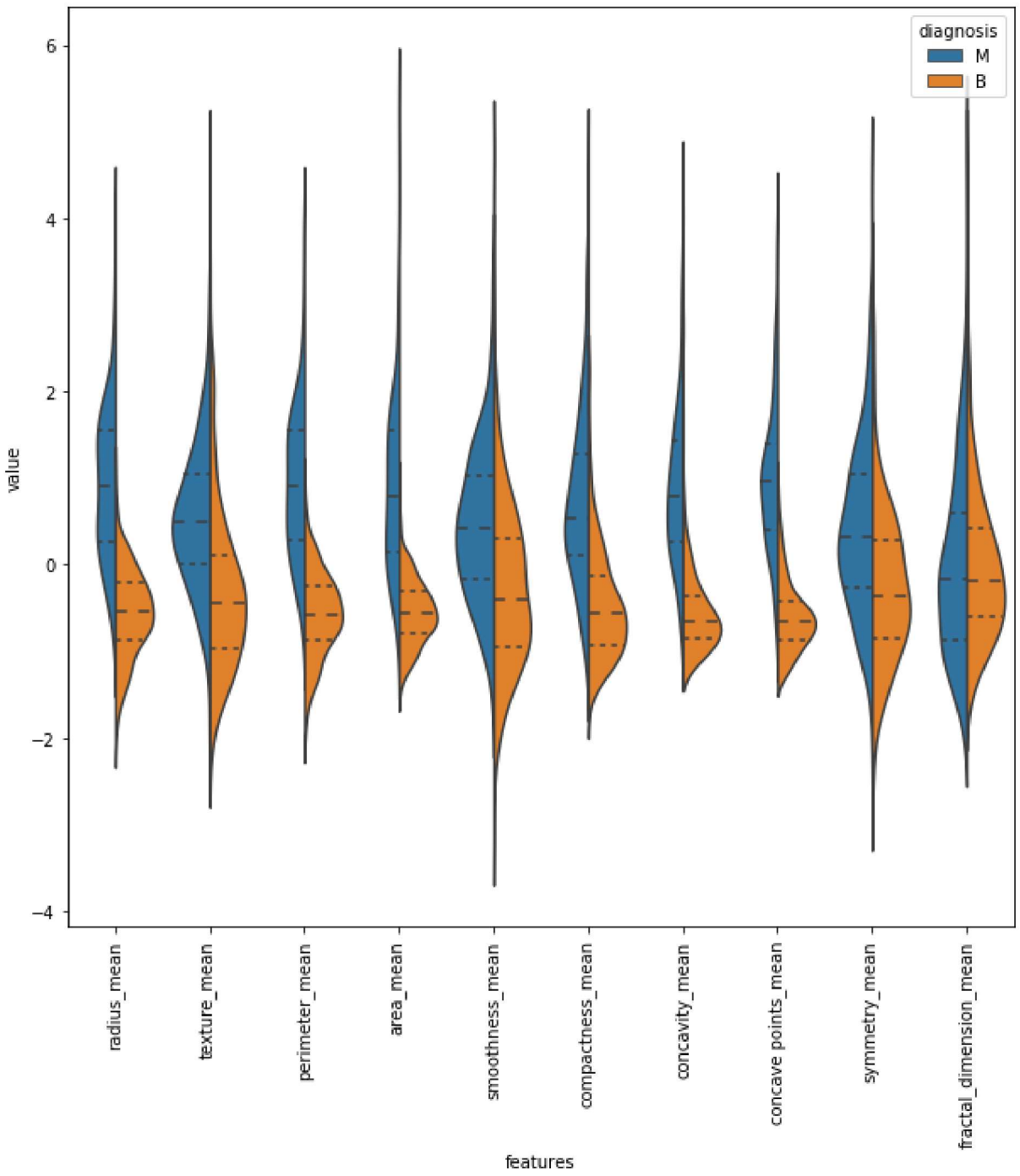
To visualize the data, we'll utilize seaborn plots, a choice not commonly seen in other kernels, aiming to provide diversity in our visualizations. In my real-life applications, I often rely on violin plots and swarm plots. It's important to note that we're not selecting features here; rather, we're endeavoring to understand the data, akin to perusing the drink list displayed at the entrance of a pub.

Prior to creating violin and swarm plots, normalization or standardization is necessary. The wide-ranging differences between feature values make it challenging to observe patterns on the plot. I'll be plotting features in three groups, each containing ten features, to

enhance visibility and facilitate better observation.

In [8]:
```
# first ten features
data_dia = y
data = x
data_n_2 = (data - data.mean()) / (data.std())              # standardization
data = pd.concat([y,data_n_2.iloc[:,0:10]],axis=1)
data = pd.melt(data,id_vars="diagnosis",
                    var_name="features",
                    value_name='value')
plt.figure(figsize=(10,10))
sns.violinplot(x="features", y="value", hue="diagnosis", data=data,split=True, inner="quart")
plt.xticks(rotation=90)
```

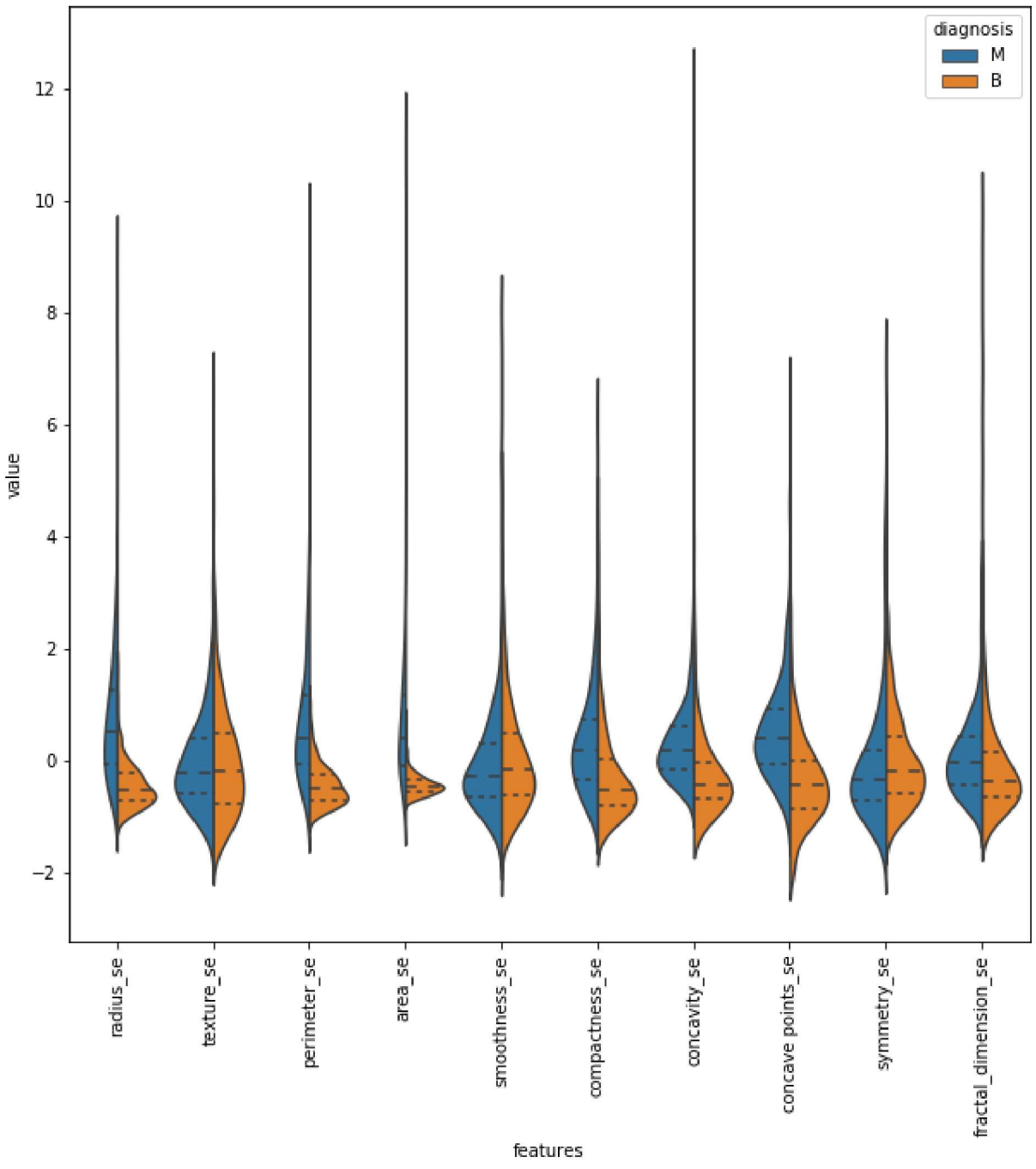
Out[8]:
```
(array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]), <a list of 10 Text xticklabel objects>)
```



Let's interpret the plot together. For instance, in the 'texture_mean' feature, the median values between Malignant and Benign cases appear distinct, suggesting potential usefulness for classification purposes. However, in the 'fractal_dimension_mean' feature, the medians between Malignant and Benign cases don't exhibit clear separation, indicating that this feature may not provide significant information for classification.

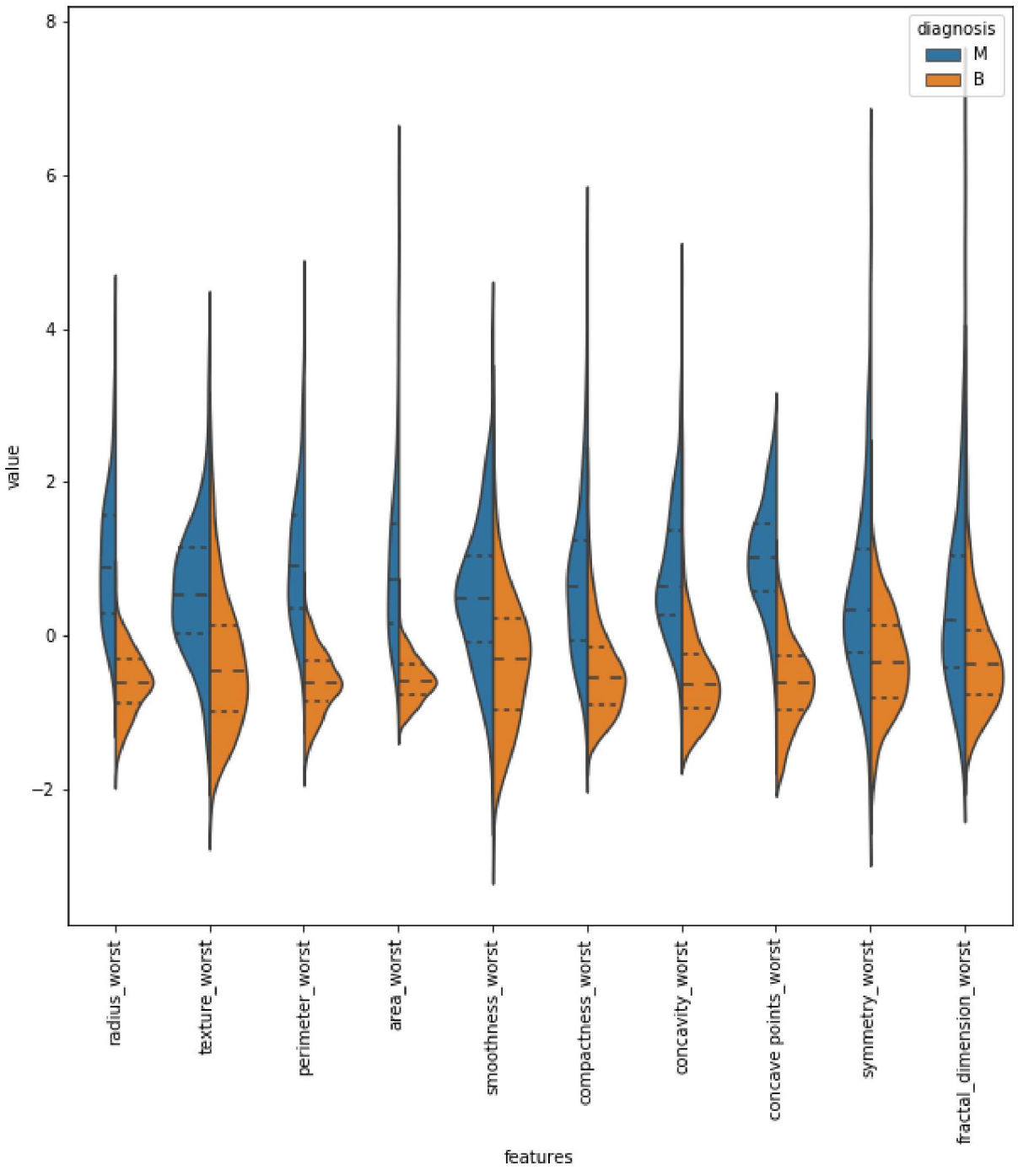
In [9]:
```
# Second ten features
data = pd.concat([y,data_n_2.iloc[:,10:20]],axis=1)
data = pd.melt(data,id_vars="diagnosis",
                    var_name="features",
                    value_name='value')
plt.figure(figsize=(10,10))
sns.violinplot(x="features", y="value", hue="diagnosis", data=data,split=True, inner="quart")
plt.xticks(rotation=90)
```

Out[9]:
```
(array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]), <a list of 10 Text xticklabel objects>)
```



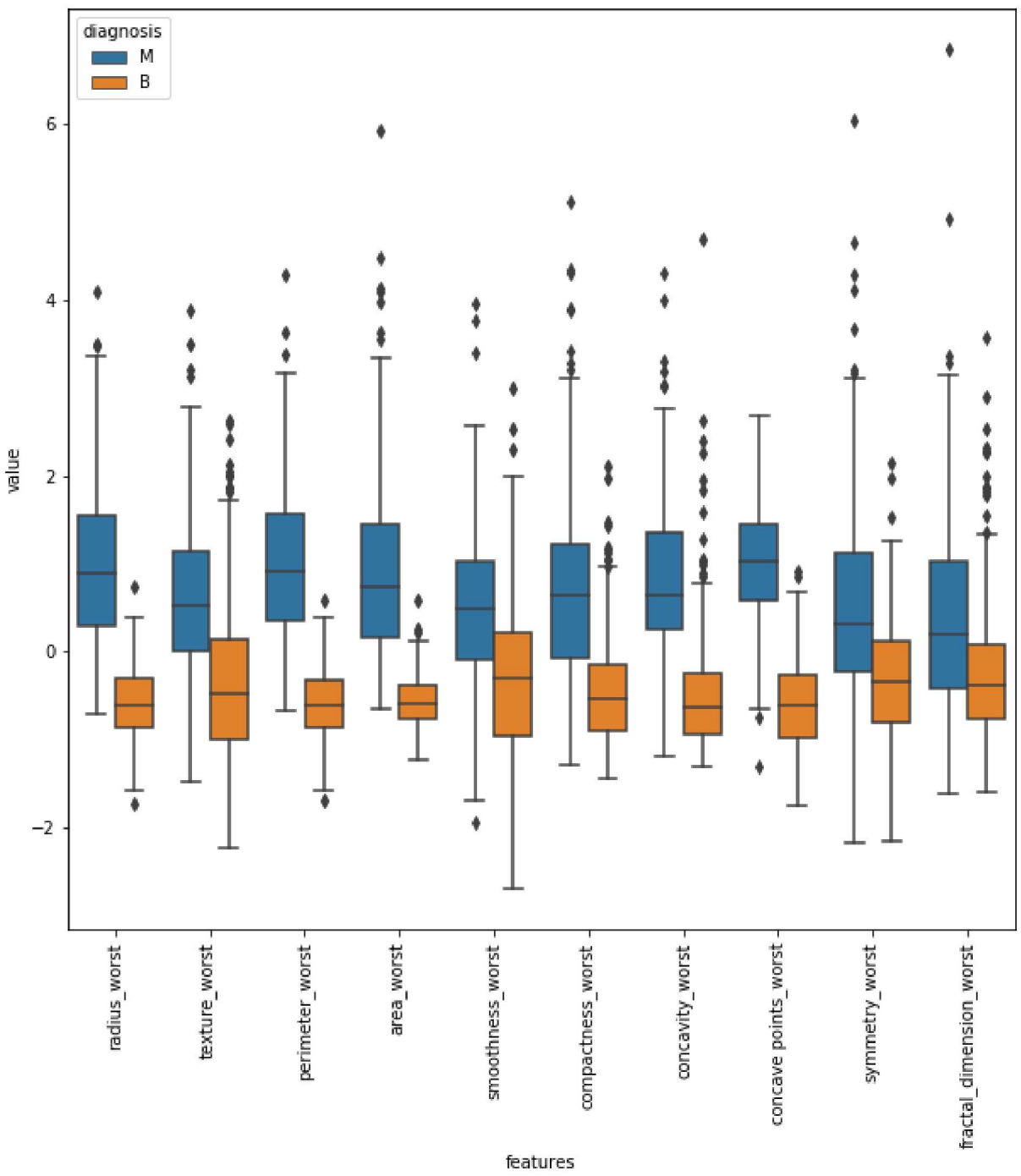
In [10]:
```
# Second ten features
data = pd.concat([y,data_n_2.iloc[:,20:31]],axis=1)
data = pd.melt(data,id_vars="diagnosis",
                    var_name="features",
                    value_name='value')
plt.figure(figsize=(10,10))
sns.violinplot(x="features", y="value", hue="diagnosis", data=data,split=True, inner="quart")
plt.xticks(rotation=90)
```

Out[10]:
```
(array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]), <a list of 10 Text xticklabel objects>)
```



In [11]:
```
# As an alternative of violin plot, box plot can be used
# box plots are also useful in terms of seeing outliers

plt.figure(figsize=(10,10))
sns.boxplot(x="features", y="value", hue="diagnosis", data=data)
plt.xticks(rotation=90)
```

Out[11]:
```
(array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]), <a list of 10 Text xticklabel objects>)
```

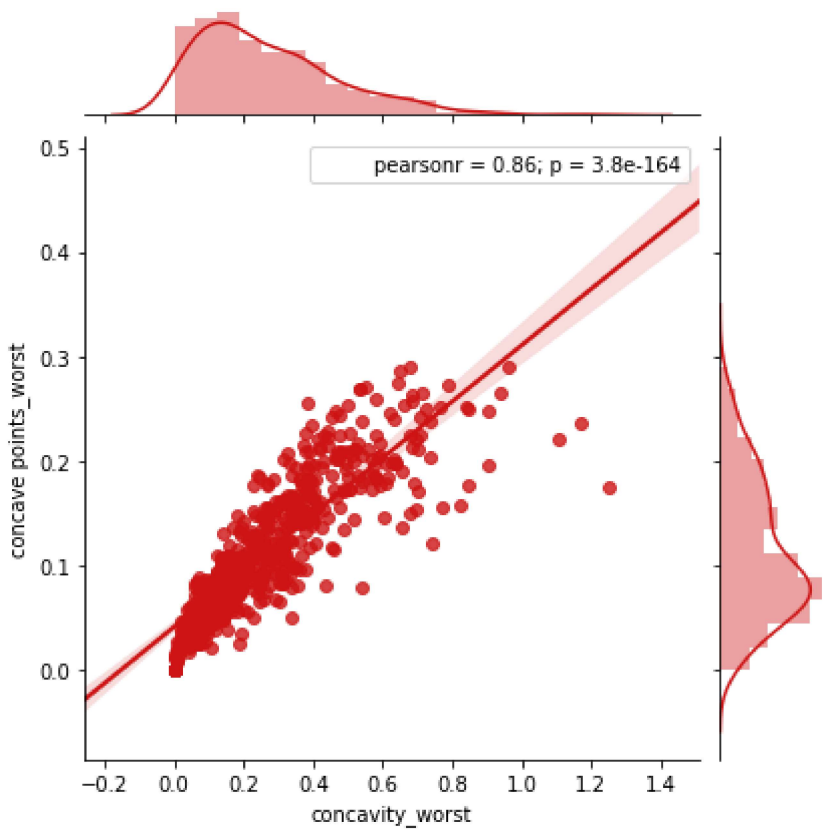


Let's interpret another aspect of the plot above. The variables 'concavity_worst' and 'concave point_worst' seem similar. However, how do we determine if they are correlated with each other? (Although not always the case, if features are correlated, it's generally acceptable to drop one of them.)

To delve deeper into comparing these two features, let's utilize a joint plot. As observed in the joint plot below, there's a strong correlation between them. The Pearson correlation coefficient (Pearsonr) represents the correlation value, with 1 being the highest possible correlation. Hence, a value of 0.86 is significant enough to indicate a strong correlation between these features. It's crucial to note that at this stage, we're not yet selecting features; rather, we're simply gaining insights about them.

In [12]:
```
sns.jointplot(x.loc[:,'concavity_worst'], x.loc[:,'concave points_worst'], kind="regg", color="#ce1414")
```

Out[12]:
```
<seaborn.axisgrid.JointGrid at 0x79eb9d8f98d0>
```

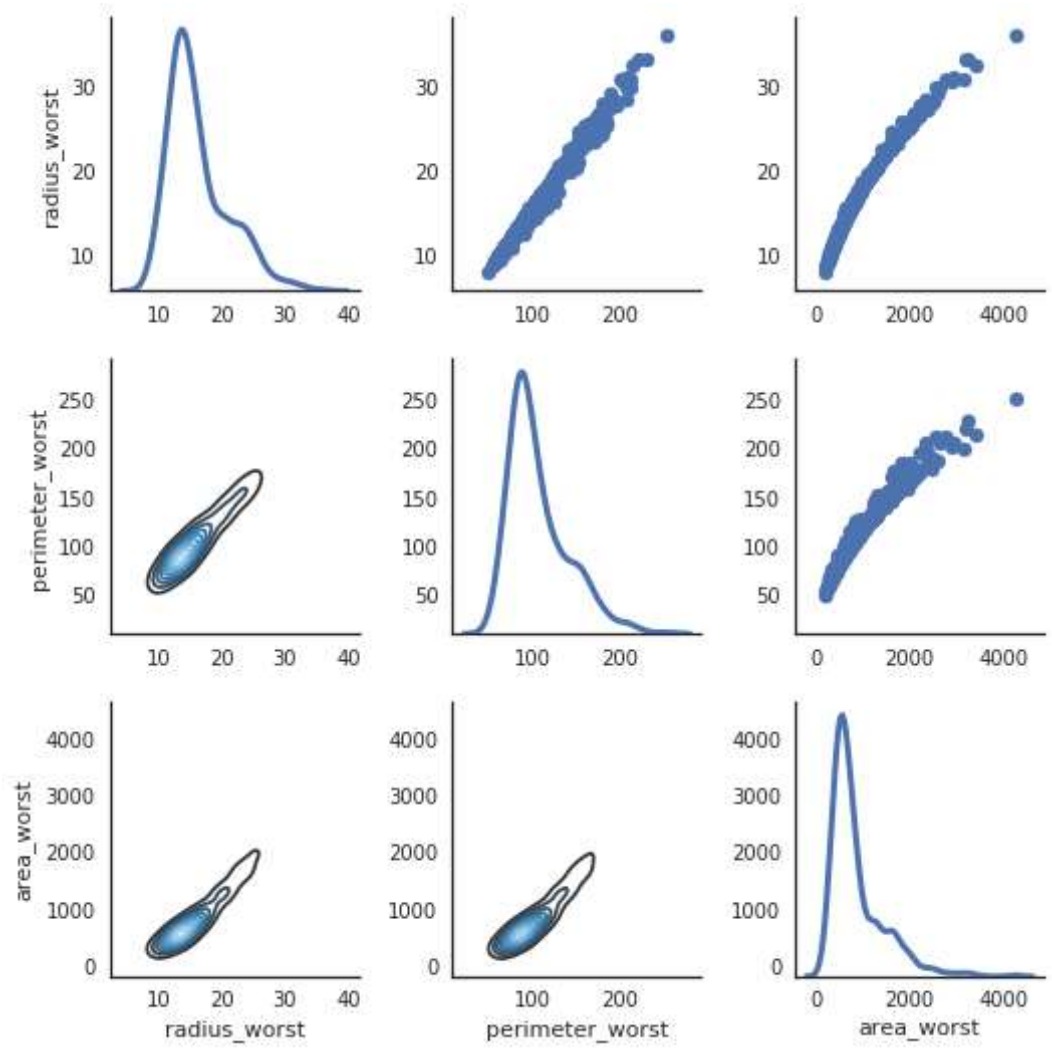


What about comparisons involving three or more features? We can utilize a pair grid plot for this purpose, which also looks quite impressive! Through this visualization, we've discovered another significant finding: 'radius_worst,' 'perimeter_worst,' and 'area_worst' exhibit correlation, as evident in the pair grid plot. These insights will undoubtedly guide our feature selection process.

In [13]:
```
sns.set(style="white")
df = x.loc[:,['radius_worst','perimeter_worst','area_worst']]
g = sns.PairGrid(df, diag_sharey=False)
```

```
g.map_lower(sns.kdeplot, cmap="Blues_d")
g.map_upper(plt.scatter)
g.map_diag(sns.kdeplot, lw=3)
```

Out[13]: <seaborn.axisgrid.PairGrid at 0x79eb9d8427b8>

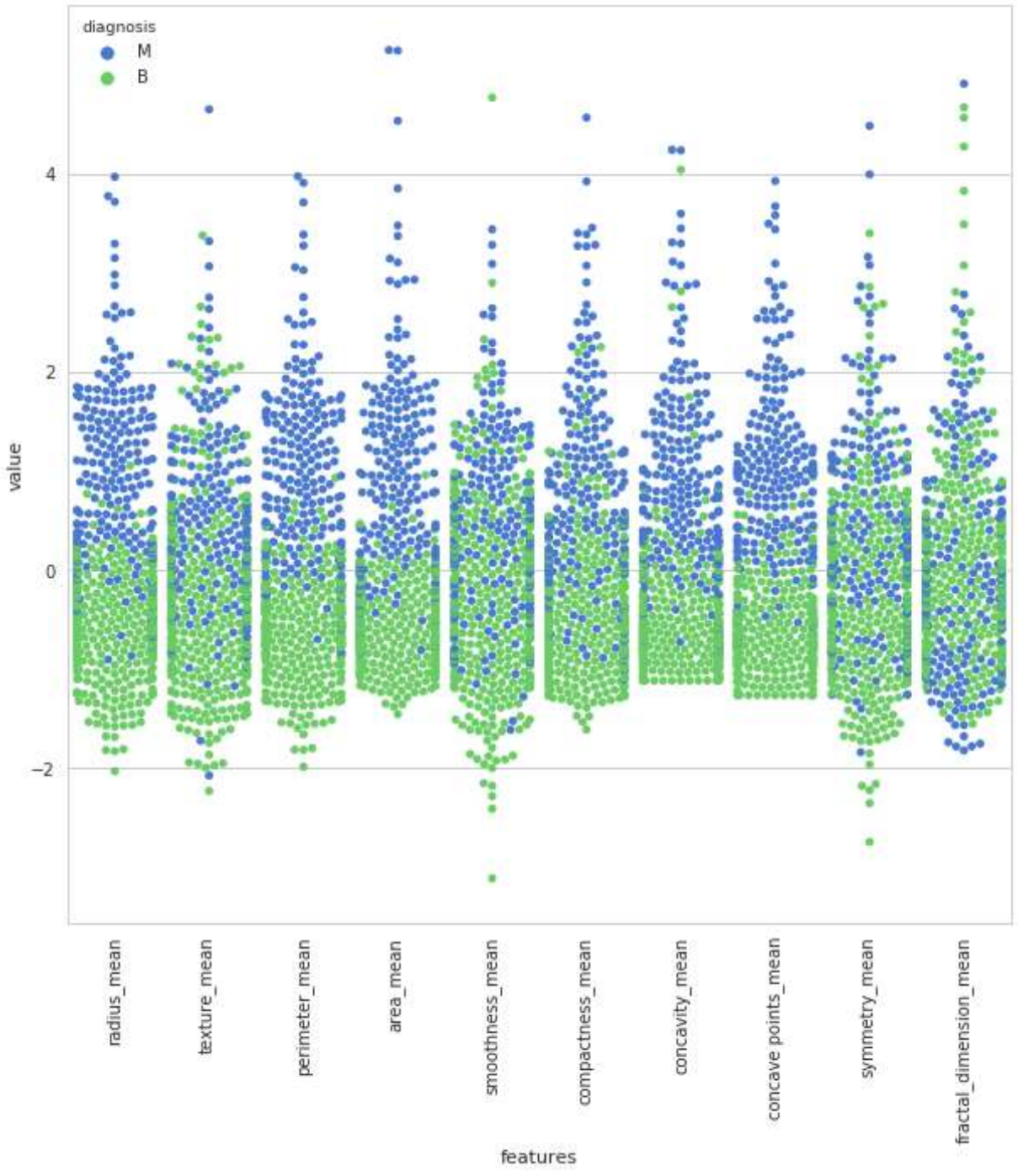


Up to this point, we've made some comments and discoveries regarding the data. If you've enjoyed our progress so far, I'm confident that the swarm plot will reveal even more as it opens the door to the pub.

Similar to the violin plot, I'll split the swarm plot into three parts to avoid overwhelming complexity in the visualization.

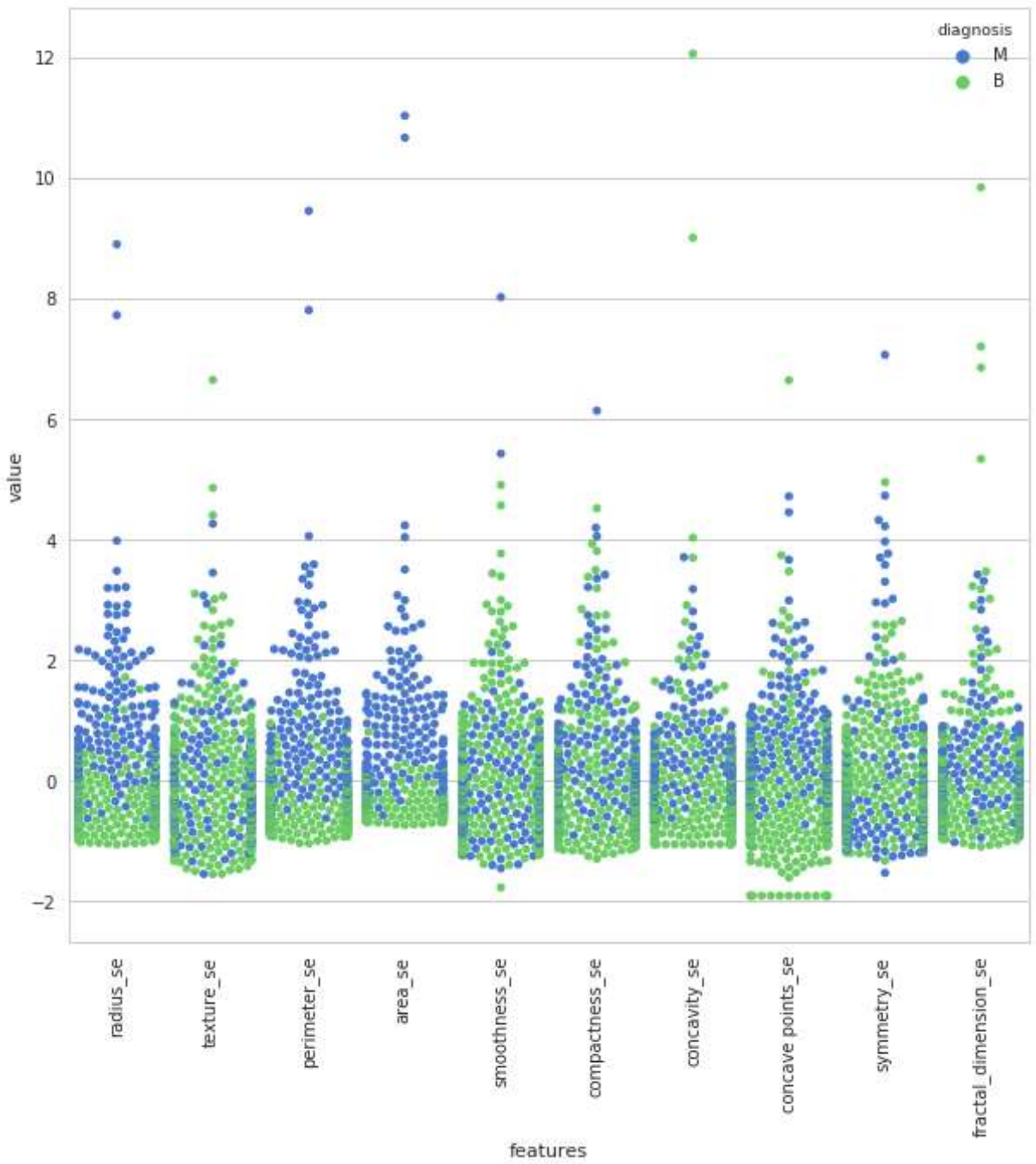
In [14]:
```
sns.set(style="whitegrid", palette="muted")
data_dia = y
data = x
data_n_2 = (data - data.mean()) / (data.std())              # standardization
data = pd.concat([y,data_n_2.iloc[:,0:10]],axis=1)
data = pd.melt(data,id_vars="diagnosis",
                    var_name="features",
                    value_name='value')
plt.figure(figsize=(10,10))
tic = time.time()
sns.swarmplot(x="features", y="value", hue="diagnosis", data=data)

plt.xticks(rotation=90)
```

Out[14]: (array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]), <a list of 10 Text xticklabel objects>)



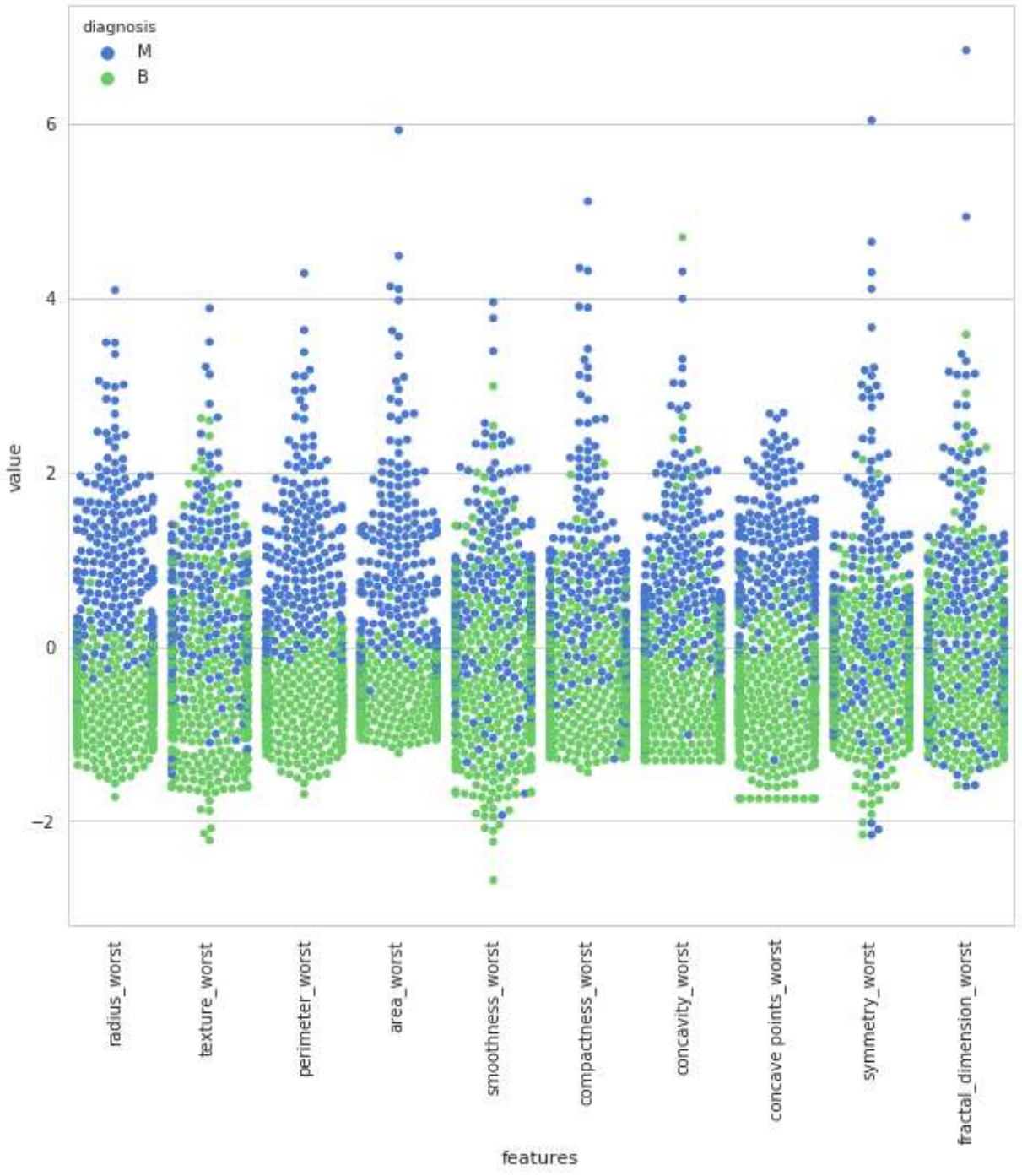
In [15]:
```
data = pd.concat([y,data_n_2.iloc[:,10:20]],axis=1)
data = pd.melt(data,id_vars="diagnosis",
                    var_name="features",
                    value_name='value')
plt.figure(figsize=(10,10))
sns.swarmplot(x="features", y="value", hue="diagnosis", data=data)
plt.xticks(rotation=90)
```

Out[15]:
```
(array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]), <a list of 10 Text xticklabel objects>)
```



In [16]:
```
data = pd.concat([y,data_n_2.iloc[:,20:31]],axis=1)
data = pd.melt(data,id_vars="diagnosis",
                    var_name="features",
                    value_name='value')
plt.figure(figsize=(10,10))
sns.swarmplot(x="features", y="value", hue="diagnosis", data=data)
toc = time.time()
plt.xticks(rotation=90)
print("swarm plot time: ", toc-tic ," s")
```

```
swarm plot time:  16.597289323806763  s
```

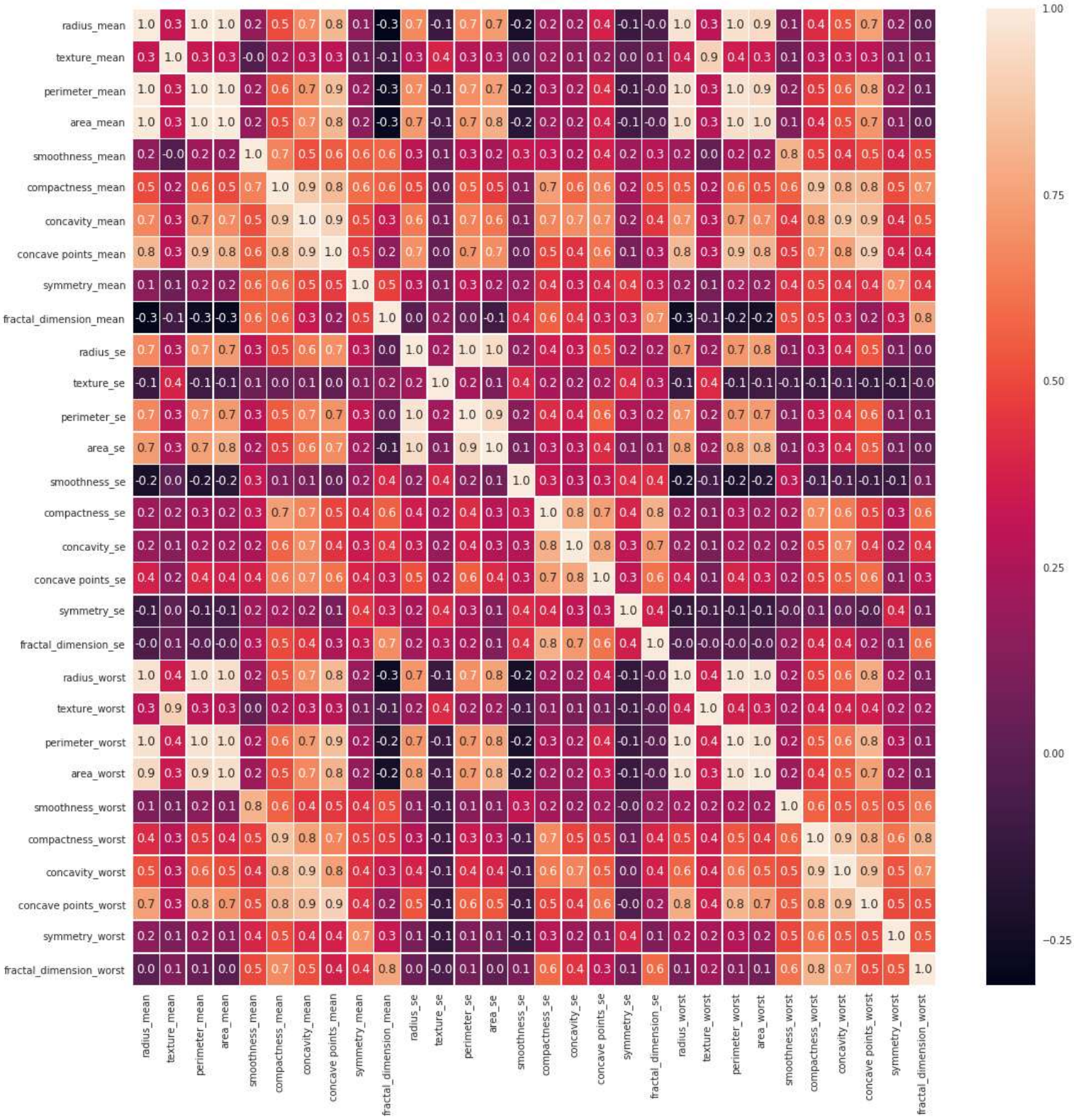


You can clearly observe the variance. Let me pose a question: among these three plots, which feature appears more distinct in terms of classification? In my view, in the last swarm plot, 'area_worst' seems to exhibit a clearer separation between malignant and benign cases, albeit not entirely, but mostly. On the other hand, 'smoothness_se' in the second swarm plot appears quite mixed between malignant and benign, making classification challenging.

Now, if we aim to observe the correlations among all features, what would be the method? Yes, you're right—the answer is the heatmap, an old yet powerful plotting method.

In [17]:
```
#correlation map
f,ax = plt.subplots(figsize=(18, 18))
sns.heatmap(x.corr(), annot=True, linewidths=.5, fmt= '.1f',ax=ax)
```

Out[17]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x79eb9c9d9c88>
```



Finally, we've arrived at the pub, and now it's time to select our drinks—that is, our features—using the heatmap, also known as the correlation matrix.

Feature Selection and Random Forest Classification

In this part we will select feature with different methods that are feature selection with correlation, univariate feature selection, recursive feature elimination (RFE), recursive feature elimination with cross validation (RFECV) and tree based feature selection. We will use random forest classification in order to train our model and predict.

1) Feature selection with correlation and random forest classification

In the heatmap figure, it's evident that 'radius_mean,' 'perimeter_mean,' and 'area_mean' are correlated with each other, prompting the decision to solely utilize 'area_mean.' You might wonder how I selected 'area_mean' as the feature to use. Well, there's no definitive answer—I simply observed the swarm plots, and 'area_mean' appeared clearer to me. However, distinguishing among other correlated features without experimentation isn't exact. So, let's identify other correlated features and assess their accuracy using a random forest classifier.

Observing 'compactness_mean,' 'concavity_mean,' and 'concave points_mean,' which are correlated, I opt for 'concavity_mean.' Similarly, 'radius_se,' 'perimeter_se,' and 'area_se' exhibit correlation, leading me to choose 'area_se.' Likewise, 'radius_worst,' 'perimeter_worst,' and 'area_worst' show correlation, and I select 'area_worst.' 'Compactness_worst,' 'concavity_worst,' and 'concave points_worst' are correlated, hence I choose 'concavity_worst.' 'Compactness_se,' 'concavity_se,' and 'concave points_se' exhibit correlation, so I opt for 'concavity_se.' Additionally, 'texture_mean' and 'texture_worst' show correlation, and I use 'texture_mean.' Lastly, considering the correlation between 'area_worst' and 'area_mean,' I decide to use 'area_mean.'

In [18]:
```
drop_list1 = ['perimeter_mean','radius_mean','compactness_mean','concave points_mean','radius_se','perimeter_se','radiu
x_1 = x.drop(drop_list1,axis = 1 )        # do not modify x, we will use it later
x_1.head()
```

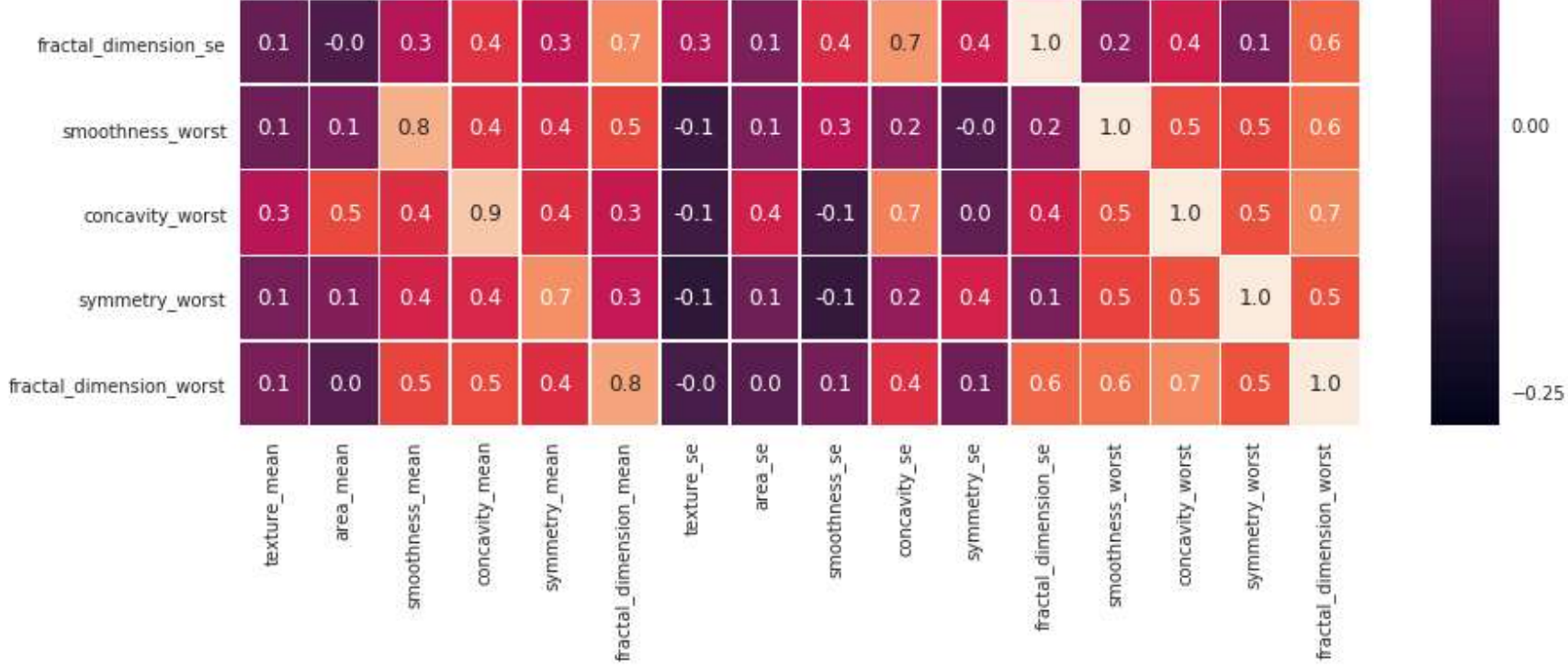
Out[18]:

	texture_mean	area_mean	smoothness_mean	concavity_mean	symmetry_mean	fractal_dimension_mean	texture_se	area_se	smoothness_s
0	10.38	1001.0	0.11840	0.3001	0.2419	0.07871	0.9053	153.40	0.00639
1	17.77	1326.0	0.08474	0.0869	0.1812	0.05667	0.7339	74.08	0.00522
2	21.25	1203.0	0.10960	0.1974	0.2069	0.05999	0.7869	94.03	0.00615
3	20.38	386.1	0.14250	0.2414	0.2597	0.09744	1.1560	27.23	0.00911
4	14.34	1297.0	0.10030	0.1980	0.1809	0.05883	0.7813	94.44	0.01149

After dropping the correlated features, as depicted in the following correlation matrix, no further correlated features remain. Admittedly, I'm aware and you might notice there's a correlation value of 0.9. However, let's explore together what occurs if we don't drop it.

In [19]:
```
#correlation map
f,ax = plt.subplots(figsize=(14, 14))
sns.heatmap(x_1.corr(), annot=True, linewidths=.5, fmt= '.1f',ax=ax)
```

Out[19]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x79eb9ca04630>
```

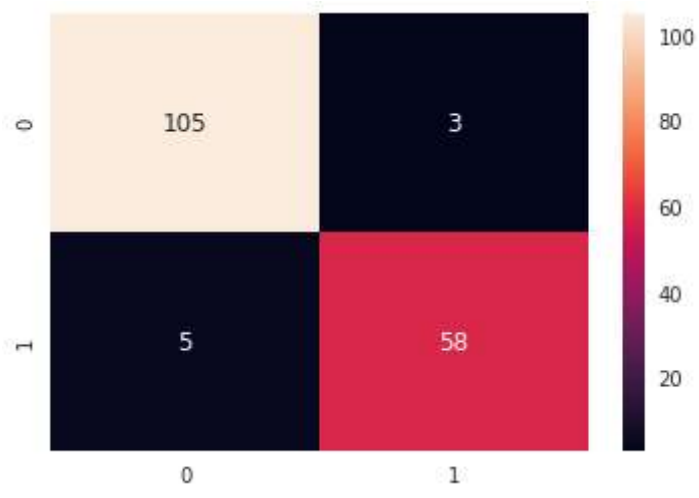


Lets use random forest and find accuracy according to chosen features.

In [20]:
```
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import f1_score,confusion_matrix
from sklearn.metrics import accuracy_score

# split data train 70 % and test 30 %
x_train, x_test, y_train, y_test = train_test_split(x_1, y, test_size=0.3, random_state=42)

#random forest classifier with n_estimators=10 (default)
clf_rf = RandomForestClassifier(random_state=43)
clr_rf = clf_rf.fit(x_train,y_train)

ac = accuracy_score(y_test,clf_rf.predict(x_test))
print('Accuracy is: ',ac)
```

```
cm = confusion_matrix(y_test,clf_rf.predict(x_test))
sns.heatmap(cm,annot=True,fmt="d")
```

Out[20]:
```
Accuracy is:  0.9532163742690059
<matplotlib.axes._subplots.AxesSubplot at 0x79eba3693080>
```



The accuracy stands at almost 95%, and as evident in the confusion matrix, we've made a few incorrect predictions. Now, let's explore other feature selection methods to potentially achieve better results.

2) Univariate feature selection and random forest classification

In univariate feature selection, we will use SelectKBest that removes all but the k highest scoring features. http://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.SelectKBest.html#sklearn.feature_selection.SelectKBest

In this method we need to choose how many features we will use. For example, will k (number of features) be 5 or 10 or 15? The answer is only trying or intuitively. I do not try all combinations but I only choose k = 5 and find best 5 features.

In [21]:
```
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import chi2
# find best scored 5 features
select_feature = SelectKBest(chi2, k=5).fit(x_train, y_train)
```

In [22]:
```
print('Score list:', select_feature.scores_)
print('Feature list:', x_train.columns)
```

```
Score list: [6.06916433e+01 3.66899557e+04 1.00015175e-01 1.30547650e+01
 1.95982847e-01 3.42575072e-04 4.07131026e-02 6.12741067e+03
 1.32470372e-03 6.92896719e-01 1.39557806e-03 2.65927071e-03
 2.63226314e-01 2.58858117e+01 1.00635138e+00 1.23087347e-01]
Feature list: Index(['texture_mean', 'area_mean', 'smoothness_mean', 'concavity_mean',
       'symmetry_mean', 'fractal_dimension_mean', 'texture_se', 'area_se',
       'smoothness_se', 'concavity_se', 'symmetry_se', 'fractal_dimension_se',
       'smoothness_worst', 'concavity_worst', 'symmetry_worst',
       'fractal_dimension_worst'],
      dtype='object')
```

Best 5 feature to classify is that **area_mean, area_se, texture_mean, concavity_worst and concavity_mean**. So lets see what happens if we use only these best scored 5 feature.

In [23]:
```
x_train_2 = select_feature.transform(x_train)
x_test_2 = select_feature.transform(x_test)
#random forest classifier with n_estimators=10 (default)
clf_rf_2 = RandomForestClassifier()
clr_rf_2 = clf_rf_2.fit(x_train_2,y_train)
ac_2 = accuracy_score(y_test,clf_rf_2.predict(x_test_2))
print('Accuracy is: ',ac_2)
cm_2 = confusion_matrix(y_test,clf_rf_2.predict(x_test_2))
sns.heatmap(cm_2,annot=True,fmt="d")
```

Out[23]:
```
Accuracy is:  0.9590643274853801
<matplotlib.axes._subplots.AxesSubplot at 0x79eb7d1b18d0>
```

The accuracy stands at almost 96%, and as evident in the confusion matrix, we've made a few incorrect predictions. Up to this point, we've selected features based on the correlation matrix and utilized the selectkBest method, using five features. Despite this, the accuracies appear similar. Now, let's explore additional feature selection methods to potentially achieve improved results.

3) Recursive feature elimination (RFE) with random forest

http://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.RFE.html

Essentially, it employs a classification method (such as random forest in our example) to assign weights to each feature. Those with the smallest absolute weights are pruned from the current feature set. This procedure is then recursively repeated on the pruned set until the desired number of features is reached.

Similar to the previous method, we will use five features. However, which five features will we select? We will choose them using the RFE (Recursive Feature Elimination) method.

In [24]:
```
from sklearn.feature_selection import RFE
# Create the RFE object and rank each pixel
clf_rf_3 = RandomForestClassifier()
rfe = RFE(estimator=clf_rf_3, n_features_to_select=5, step=1)
rfe = rfe.fit(x_train, y_train)
```

In [25]:
```
print('Chosen best 5 feature by rfe:',x_train.columns[rfe.support_])
```

```
Chosen best 5 feature by rfe: Index(['texture_mean', 'area_mean', 'concavity_mean', 'area_se',
       'concavity_se'],
      dtype='object')
```

The five best features selected by RFE are 'texture_mean,' 'area_mean,' 'concavity_mean,' 'area_se,' and 'concavity_worst,' mirroring the selection from the previous selectkBest method. Hence, there's no need to recalculate the accuracy.

In summary, we've achieved effective feature selection using both the RFE and selectkBest methods. However, there seems to be a lingering question: why precisely five features? Perhaps exploring the use of fewer or more features—say, the best two or fifteen—might yield better accuracy. Therefore, let's employ the RFECV method to determine the optimal number of features needed.

4) Recursive feature elimination with cross validation and random forest classification

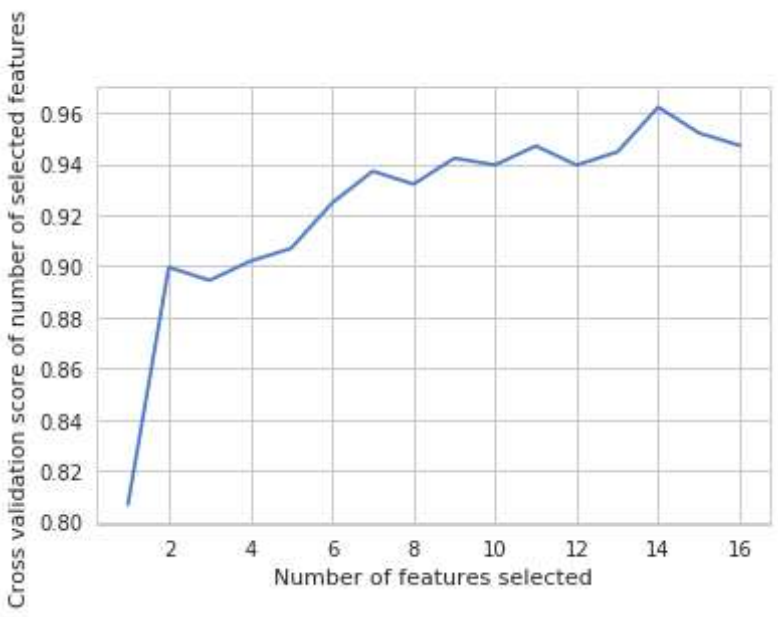
http://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.RFECV.html Now we will not only **find best features** but we also find **how many features do we need** for best accuracy.

In [26]:
```
from sklearn.feature_selection import RFECV

# The "accuracy" scoring is proportional to the number of correct classifications
clf_rf_4 = RandomForestClassifier()
rfecv = RFECV(estimator=clf_rf_4, step=1, cv=5,scoring='accuracy')   #5-fold cross-validation
rfecv = rfecv.fit(x_train, y_train)

print('Optimal number of features :', rfecv.n_features_)
print('Best features :', x_train.columns[rfecv.support_])
```

```
Optimal number of features : 14
Best features : Index(['texture_mean', 'area_mean', 'smoothness_mean', 'concavity_mean',
       'fractal_dimension_mean', 'area_se', 'smoothness_se', 'concavity_se',
       'symmetry_se', 'fractal_dimension_se', 'smoothness_worst',
       'concavity_worst', 'symmetry_worst', 'fractal_dimension_worst'],
      dtype='object')
```

At last, we've identified the best 11 features for optimal classification: 'texture_mean,' 'area_mean,' 'concavity_mean,' 'texture_se,' 'area_se,' 'concavity_se,' 'symmetry_se,' 'smoothness_worst,' 'concavity_worst,' 'symmetry_worst,' and 'fractal_dimension_worst.' Now, let's visualize the highest accuracy achieved using these features.

In [27]:
```
# Plot number of features VS. cross-validation scores
import matplotlib.pyplot as plt
plt.figure()
plt.xlabel("Number of features selected")
plt.ylabel("Cross validation score of number of selected features")
plt.plot(range(1, len(rfecv.grid_scores_) + 1), rfecv.grid_scores_)
plt.show()
```



Let's take a moment to recap our progress thus far. It's safe to assume that this dataset is relatively easy to classify. However, our primary objective isn't solely achieving high accuracy. Our main goal is to comprehend and learn the process of feature selection while gaining a deeper understanding of the data. So, let's proceed with our final feature selection method.

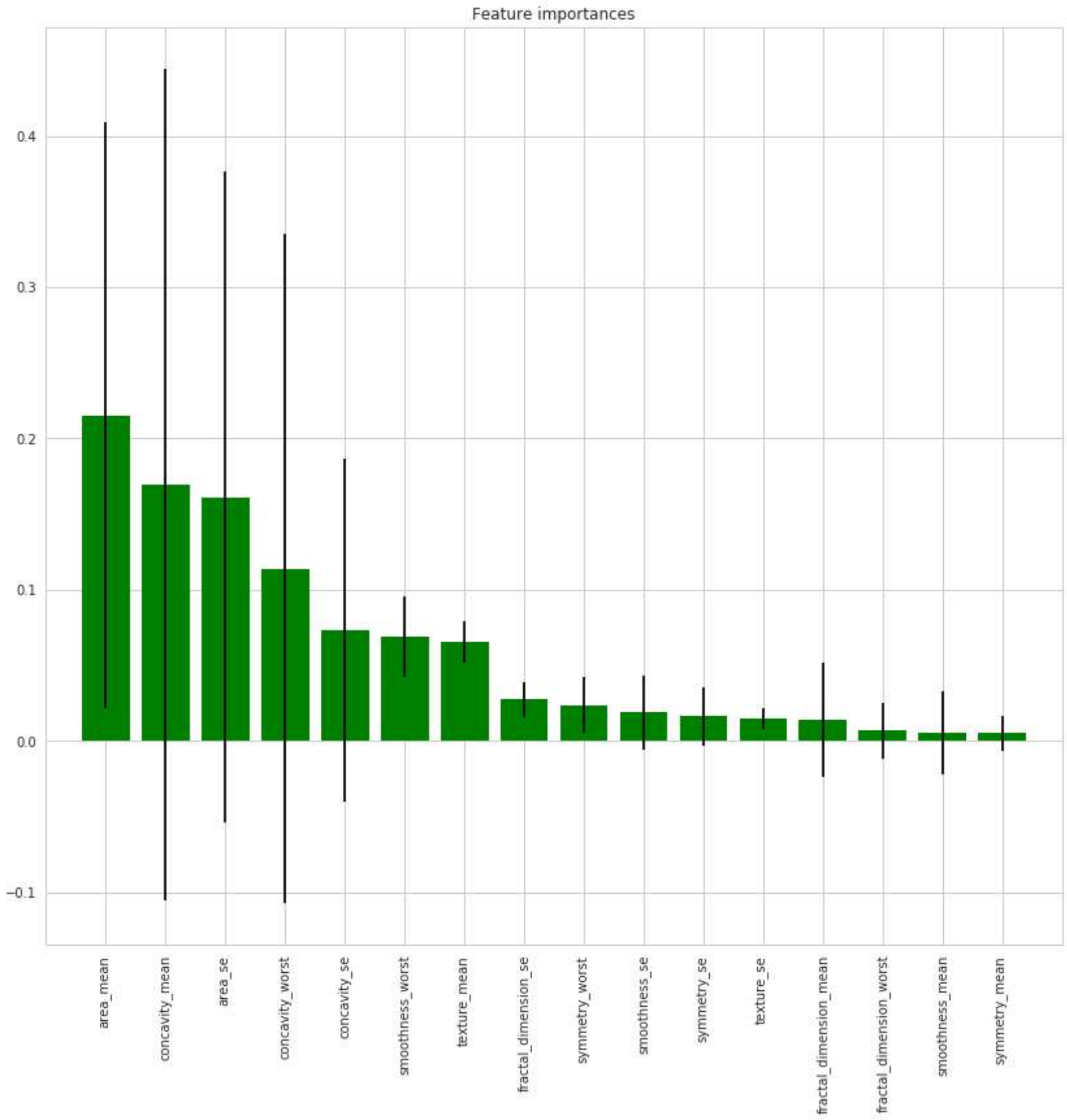
5) Tree based feature selection and random forest classification

http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

In the random forest classification method, there's a feature*importances* attribute that signifies the importance of each feature—the higher the value, the more significant the feature. To properly utilize the feature*importances* method, it's crucial to ensure that the training data doesn't contain correlated features. Additionally, due to the random nature of random forests, the sequence of the feature importance list can vary between iterations.

In [28]:
```
clf_rf_5 = RandomForestClassifier()
clr_rf_5 = clf_rf_5.fit(x_train,y_train)
importances = clr_rf_5.feature_importances_
std = np.std([tree.feature_importances_ for tree in clf_rf.estimators_],
             axis=0)
indices = np.argsort(importances)[::-1]

# Print the feature ranking
print("Feature ranking:")

for f in range(x_train.shape[1]):
    print("%d. feature %d (%f)" % (f + 1, indices[f], importances[indices[f]]))

# Plot the feature importances of the forest

plt.figure(1, figsize=(14, 13))
plt.title("Feature importances")
plt.bar(range(x_train.shape[1]), importances[indices],
       color="g", yerr=std[indices], align="center")
plt.xticks(range(x_train.shape[1]), x_train.columns[indices],rotation=90)
plt.xlim([-1, x_train.shape[1]])
plt.show()
```

```
Feature ranking:
1. feature 1 (0.215167)
2. feature 3 (0.169610)
3. feature 7 (0.161023)
4. feature 13 (0.113699)
5. feature 9 (0.073444)
6. feature 12 (0.068858)
7. feature 0 (0.065733)
8. feature 11 (0.027633)
9. feature 14 (0.023781)
10. feature 8 (0.018790)
11. feature 10 (0.016341)
12. feature 6 (0.014515)
13. feature 5 (0.014098)
14. feature 15 (0.006862)
15. feature 2 (0.005264)
16. feature 4 (0.005181)
```



As evident in the plot above, the importance of features decreases after the top 5 best features. Hence, we can concentrate on these 5 features. As I mentioned earlier, I prioritize understanding the features and identifying the best among them.

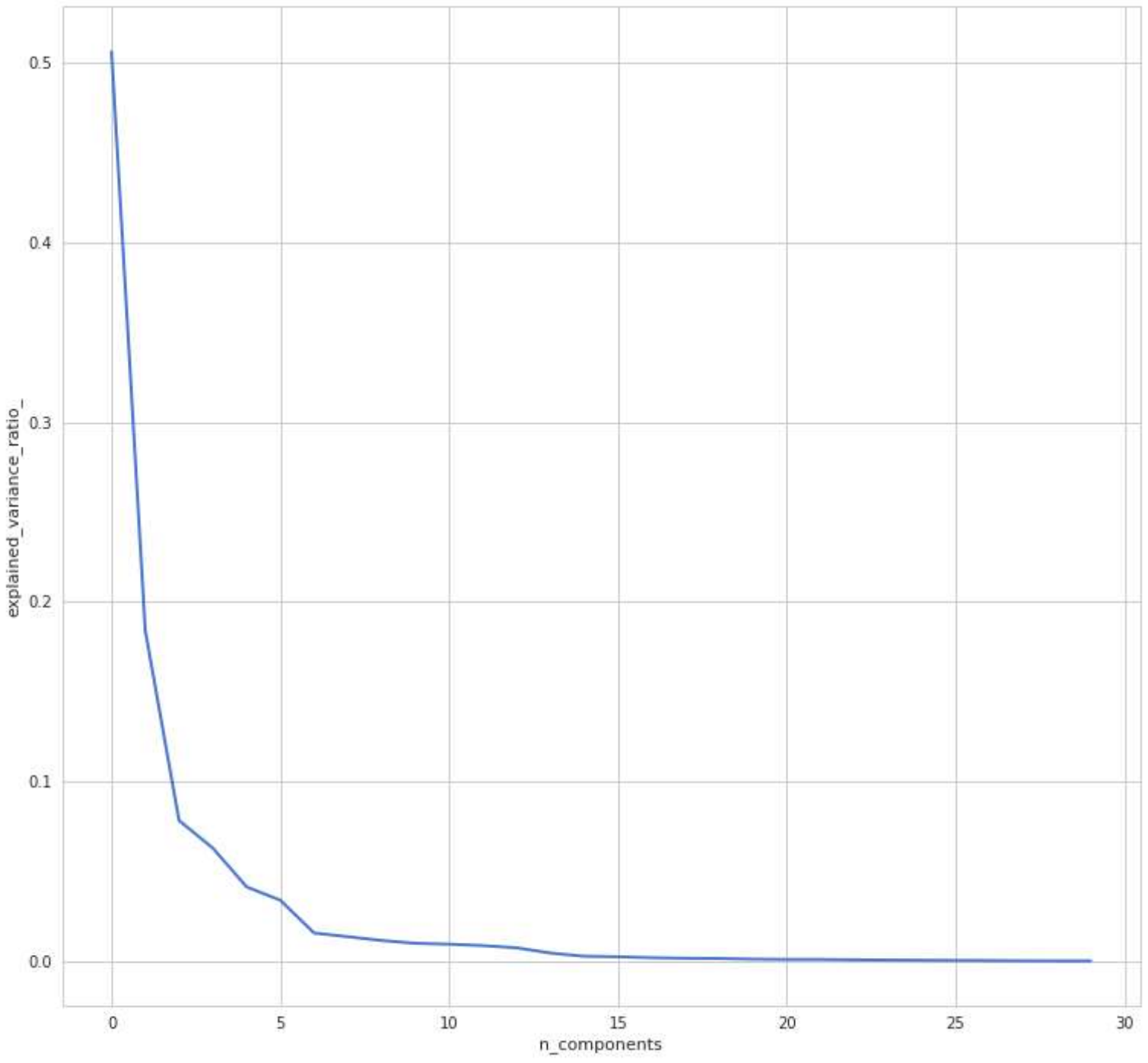
Feature Extraction with PCA

http://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html

We'll employ Principal Component Analysis (PCA) for feature extraction. However, prior to PCA, it's essential to normalize the data to enhance PCA's performance.

In [29]:
```
# split data train 70 % and test 30 %
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.3, random_state=42)
#normalization
x_train_N = (x_train-x_train.mean())/(x_train.max()-x_train.min())
x_test_N = (x_test-x_test.mean())/(x_test.max()-x_test.min())

from sklearn.decomposition import PCA
pca = PCA()
pca.fit(x_train_N)

plt.figure(1, figsize=(14, 13))
plt.clf()
plt.axes([.2, .2, .7, .7])
plt.plot(pca.explained_variance_ratio_, linewidth=2)
plt.axis('tight')
plt.xlabel('n_components')
plt.ylabel('explained_variance_ratio_')
```

Out[29]:
```
Text(0,0.5,'explained_variance_ratio_')
```



- According to variance ration, 3 component can be chosen.

Conclusion

I attempted to highlight the importance of feature selection and data visualization. The default dataset contains 33 features, but through feature selection, we reduced this number to 5 while achieving a 95% accuracy rate. In this kernel, we only explored basic techniques. I am confident that by employing these data visualization and feature selection methods, you can easily surpass the 95% accuracy threshold.