Breast Cancer Diagnostic

Motivation and Goals

Determining whether breast cancer is malignant or benign is crucial for medical treatment of a patient. Failing to predict it correctly may have severe consequences: patients with benign caner but are classified as with malignant cancer may suffer unnecessary treatment like chemotherapy, which is both expensive and harmful to health. More seriously, patients with malignant cancer but are classified improperly may not get proper treatment in time, which results in terrible consequence. The objective of this research experiment is to identify the most important factors affecting prediction of the type of cancer, and once the factors are determined, multiple models will be tested to identify the type of cancer with best performance.

Project Outline

The experiment was performed in three separate steps:

- Exploring and cleaning the dataset
- Reducing number of features
- Applying machine learning models

Overview of the Dataset

Features are computed from a digitized image of a fine needle aspirate (FNA) of a breast mass, which describe characteristics of the cell nuclei present in the image. The attribute information is given below:

ID number. 2) Diagnosis (M: malignant; B: benign). 3-32) Ten features computed for each cell nucleus: a) radius (mean of distances from center to points on the perimeter). b) texture (standard deviation of gray-scale values). c) perimeter. d) area. e) smoothness (local variation in radius lengths). f) compactness (perimeter^2 / area - 1.0). g) concavity (severity of concave portions of the contour). h) concave points (number of concave portions of the contour). i) symmetry. j) fractal dimension ("coastline approximation" - 1)

The mean, standard deviation and “worst” or largest (mean of three largest values) of the features are presented, resulting in 30 features.

Importing of Packages

```
import numpy as np

import pandas as pd

import seaborn as sns

import matplotlib.pyplot as plt


from sklearn.preprocessing import StandardScaler, scale

from sklearn.decompostion import PCA

from sklearn.neighbors import KneighborsClassifier

from sklearn.linear_model import LogisticRegression

from sklearn.model_selection import train_test_split, cross_val_score, GridSearchCV

from sklearn.metrics import confusion_matrix, classification_report, roc_curve, roc_auc_score
```

Exploring and Cleaning the Dataset

The data is explored as follows:

```
df = pd.read_csv('data.csv')
```

```
print(df.head(2))
```

```
       id diagnosis  radius_mean  texture_mean  perimeter_mean  area_mean  \
0  842302         M        17.99         10.38          122.8     1001.0
1  842517         M        20.57         17.77          132.9     1326.0

   smoothness_mean  compactness_mean  concavity_mean  concave points_mean  \
0          0.11840           0.27760          0.3001              0.14710
1          0.08474           0.07864          0.0869              0.07017

       ...       texture_worst  perimeter_worst  area_worst  smoothness_worst  \
0      ...               17.33            184.6      2019.0            0.1622
1      ...               23.41            158.8      1956.0            0.1238

   compactness_worst  concavity_worst  concave points_worst  symmetry_worst  \
0             0.6656           0.7119                0.2654          0.4601
1             0.1866           0.2416                0.1860          0.2750

   fractal_dimension_worst  Unnamed: 32
0                  0.11890          NaN
1                  0.08902          NaN

[2 rows x 33 columns]
```

There are 33 columns in total. The 1st column is the patient id, which is not a useful information in this case. The 2nd column is our class label. There are 30 features from the 3rd column to the 32nd column. The 33rd column has NaN values and seems to be useless. Let's look at the datatype of each column:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 569 entries, 0 to 568
Data columns (total 33 columns):
id                         569 non-null int64
diagnosis                  569 non-null object
radius_mean                569 non-null float64
texture_mean               569 non-null float64
perimeter_mean             569 non-null float64
area_mean                  569 non-null float64
smoothness_mean            569 non-null float64
compactness_mean           569 non-null float64
concavity_mean             569 non-null float64
concave points_mean        569 non-null float64
symmetry_mean              569 non-null float64
fractal_dimension_mean     569 non-null float64
radius_se                  569 non-null float64
texture_se                 569 non-null float64
perimeter_se               569 non-null float64
area_se                    569 non-null float64
smoothness_se              569 non-null float64
compactness_se             569 non-null float64
concavity_se               569 non-null float64
concave points_se          569 non-null float64
symmetry_se                569 non-null float64
fractal_dimension_se       569 non-null float64
radius_worst               569 non-null float64
texture_worst              569 non-null float64
perimeter_worst            569 non-null float64
area_worst                 569 non-null float64
smoothness_worst           569 non-null float64
compactness_worst          569 non-null float64
concavity_worst            569 non-null float64
concave points_worst       569 non-null float64
symmetry_worst             569 non-null float64
fractal_dimension_worst    569 non-null float64
Unnamed: 32                0 non-null float64
```

There are 569 observations in total, and in the 33rd column, the data is full of NaN. In other columns, there is no NaN. Also, all features are numerical, thus we don't need to do any following dummy variable transformation. As a result, we can simply delete the 1st and the 33rd columns, and also get the features as well as classes separately:

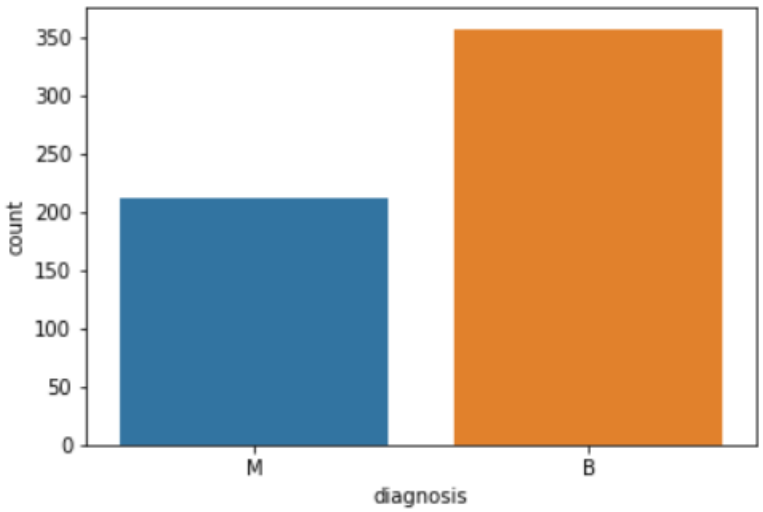
```
y = df.diagnosis
x = df.drop(['id', 'diagnosis', 'Unnamed: 32'], axis=1)
print(x.head(2))
```

```
   radius_mean  texture_mean  perimeter_mean  area_mean  smoothness_mean  \
0        17.99         10.38           122.8     1001.0          0.11840
1        20.57         17.77           132.9     1326.0          0.08474

   compactness_mean  concavity_mean  concave points_mean  symmetry_mean  \
0           0.27760          0.3001              0.14710         0.2419
1           0.07864          0.0869              0.07017         0.1812

   fractal_dimension_mean           ...             radius_worst  \
0                 0.07871           ...                    25.38
1                 0.05667           ...                    24.99

   texture_worst  perimeter_worst  area_worst  smoothness_worst  \
0          17.33            184.6      2019.0            0.1622
1          23.41            158.8      1956.0            0.1238

   compactness_worst  concavity_worst  concave points_worst  symmetry_worst  \
0             0.6656           0.7119                0.2654          0.4601
1             0.1866           0.2416                0.1860          0.2750

   fractal_dimension_worst
0                  0.11890
1                  0.08902

[2 rows x 30 columns]
```

The we take a look at the label distribution:

```
sns.countplot(y)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x11388a3ffd0>
```



Neither of the two classes have too few observations, so the dataset is quite balanced. We can just use normal predictive methods.

To get an idea of the magnitude of the features, we calculated the statistics of each feature:

```
x.describe()
```

	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean	concavity_mean	concave points_mean	symmetry_mean	fracta
count	569.000000	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	0.181162	
std	3.524049	4.301036	24.298981	351.914129	0.014064	0.052813	0.079720	0.038803	0.027414	
min	6.981000	9.710000	43.790000	143.500000	0.052630	0.019380	0.000000	0.000000	0.106000	
25%	11.700000	16.170000	75.170000	420.300000	0.086370	0.064920	0.029560	0.020310	0.161900	
50%	13.370000	18.840000	86.240000	551.100000	0.095870	0.092630	0.061540	0.033500	0.179200	
75%	15.780000	21.800000	104.100000	782.700000	0.105300	0.130400	0.130700	0.074000	0.195700	
max	28.110000	39.280000	188.500000	2501.000000	0.163400	0.345400	0.426800	0.201200	0.304000	

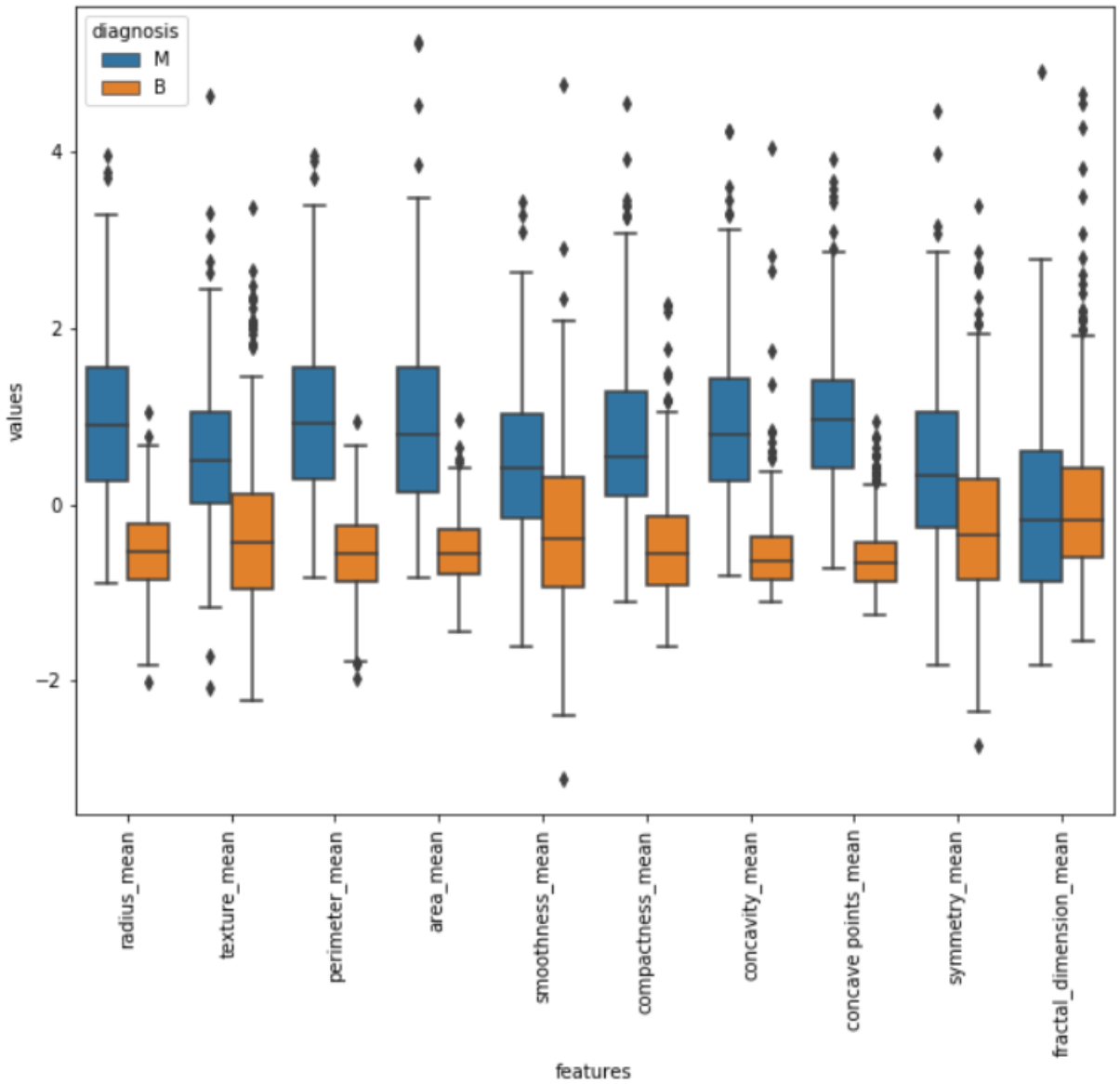
8 rows × 30 columns

There is great discrepancy on the magnitude of each feature. As a result, before we are going to have more visualizations, we first standardized the variables, and then visualized the data using boxplot. The features are divided into three groups (i.e., mean, standard deviation and largest).

```
x_st = (x - x.mean())/x.std()
```

```
data = pd.concat([y,x_st.iloc[:,0:10]],axis=1)
data = pd.melt(data, id_vars='diagnosis', var_name='features', value_name='values')
plt.figure(figsize=(10,8))
sns.boxplot(x='features', y='values', hue='diagnosis', data=data)
plt.xticks(rotation=90)
```

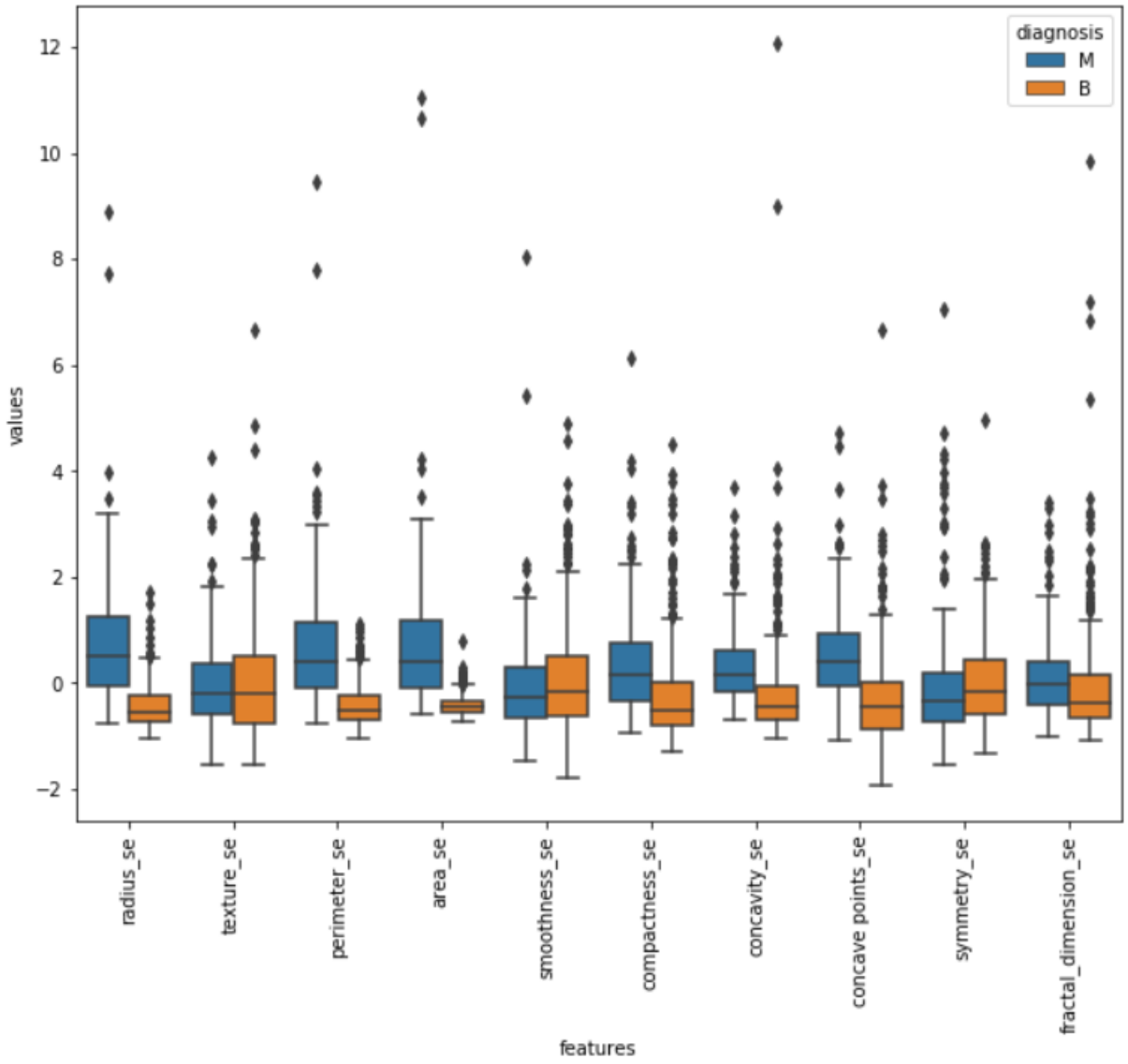
The boxplot for the first 10 mean features is shown below:



The cell nuclei for malignant cancer has significantly larger radius, perimeter, area, compactness, concavity, concave points. Slightly larger texture, smoothness and symmetry can also be observed. There seems to be little difference on the average fractal dimension.

For next ten features (standard deviation):

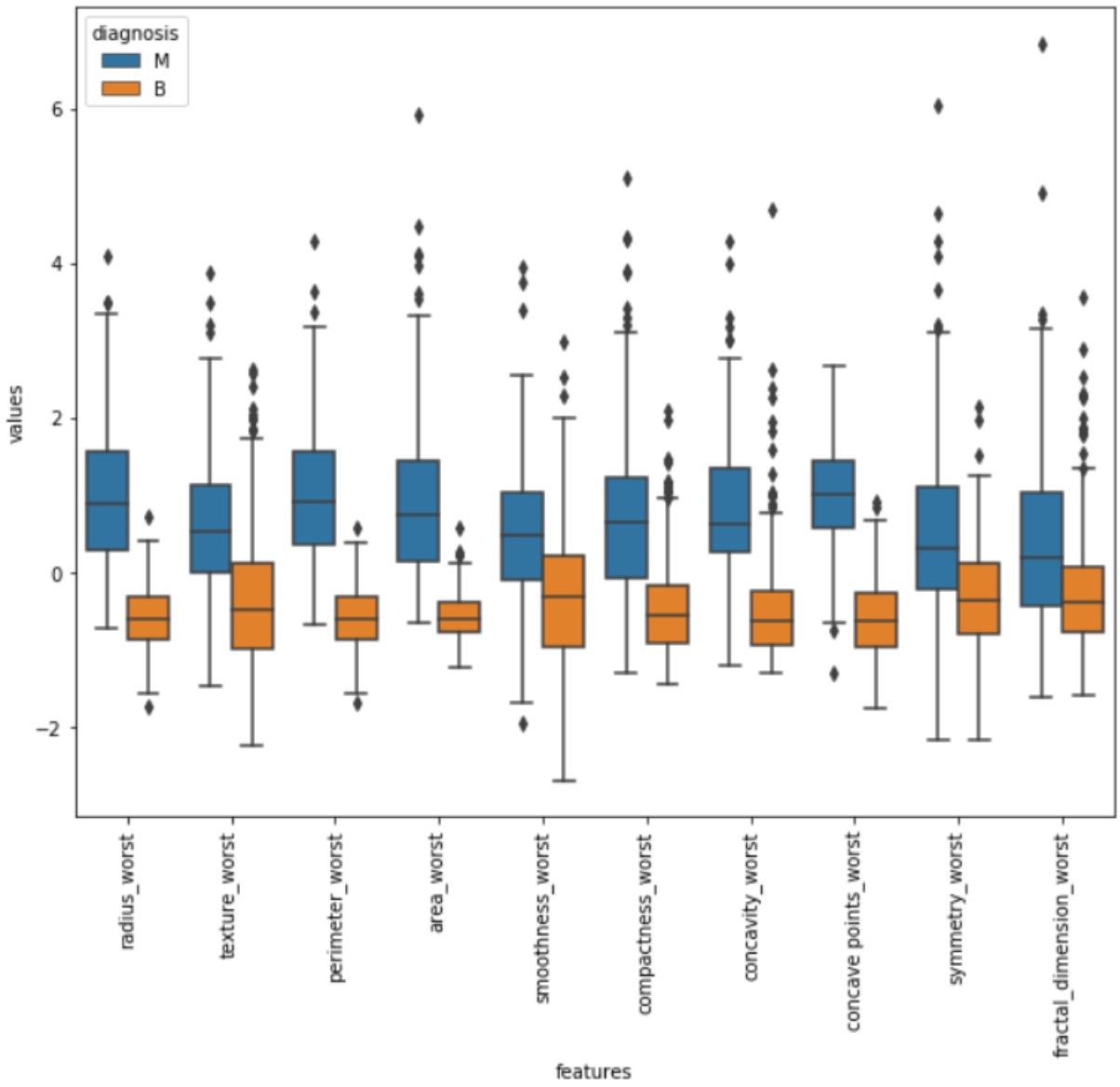
```
data = pd.concat([y,x_st.iloc[:,10:20]],axis=1)
data = pd.melt(data, id_vars='diagnosis', var_name='features', value_name='values')
plt.figure(figsize=(10,8))
sns.boxplot(x='features', y='values', hue='diagnosis', data=data)
plt.xticks(rotation=90)
```



The standard deviation of the radius, perimeter, area (although they are highly correlated) of nuclei of the malignant cancer is significantly larger than that of the benign cancer. Observable differences exist on that of compactness, concavity and concave points while the two types of

nuclei are pretty comparable on the standard deviation of texture, smoothness, symmetry and fractal dimension.

```
data = pd.concat([y,x_st.iloc[:,20:30]],axis=1)
data = pd.melt(data, id_vars='diagnosis', var_name='features', value_name='values')
plt.figure(figsize=(10,8))
sns.boxplot(x='features', y='values', hue='diagnosis', data=data)
plt.xticks(rotation=90)
```

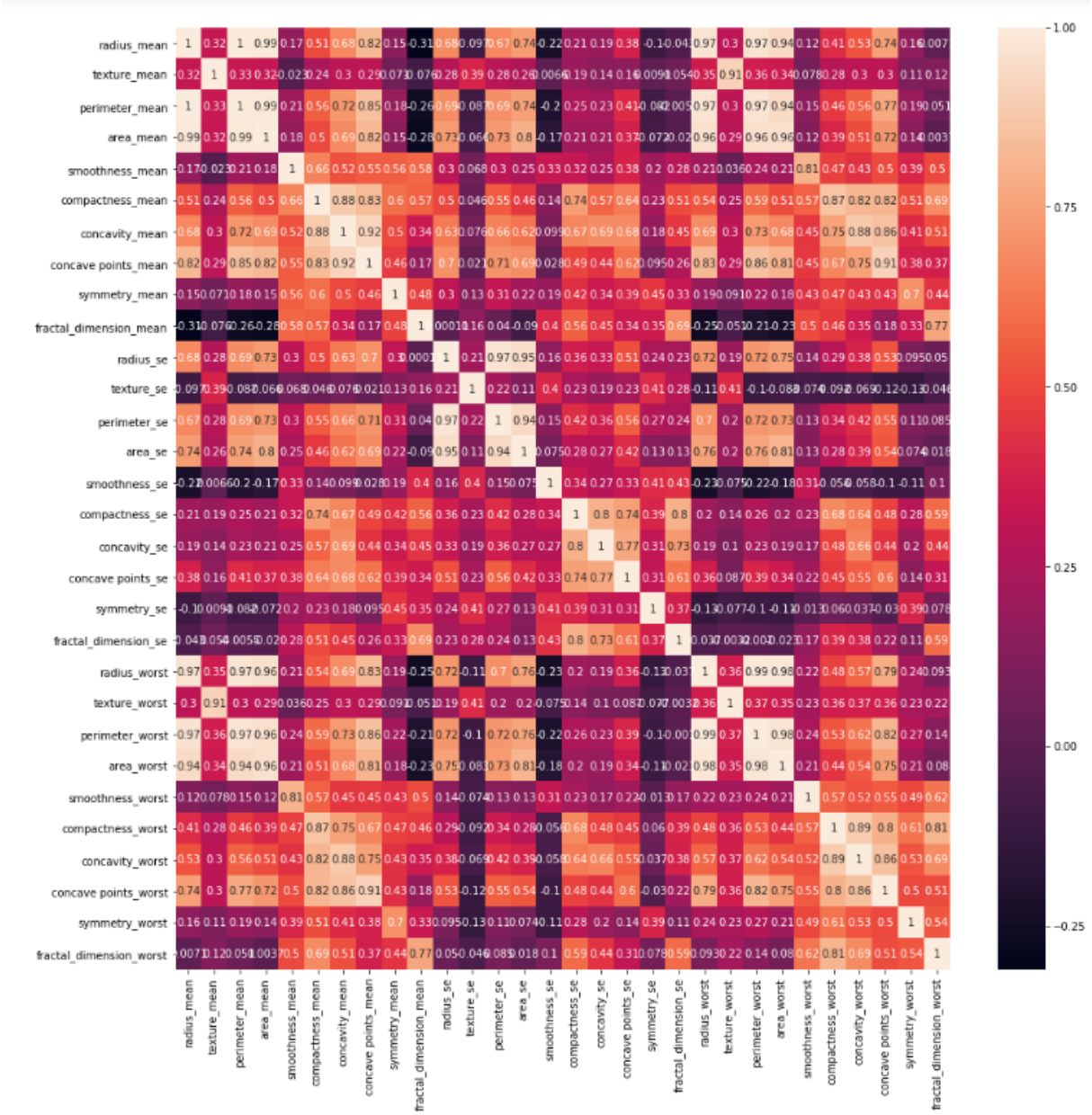


For nearly all features, nuclei malignant cancer cells have higher “worst” values, although the discrepancy is less for smoothness, symmetry and fractal dimension.

Reducing Number of Features

The number of features can be reduced by exploring the correlation between given features. Highly correlated features can be deducted to one and the others are regarded to be redundant. To do this, a heatmap of the correlation coefficient was plotted of the thirty given features, as shown below:

```
plt.figure(figsize=(16,16))
sns.heatmap(x.corr(), annot=True)
```



As expected, there are lots of strongly correlated features, so we have to reduce the number of features. We applied Principal Component Analysis to reduce dimensions. Before applying PCA, the first step is to split the data set into a training set and a testing set. A test size of 20% was applied:

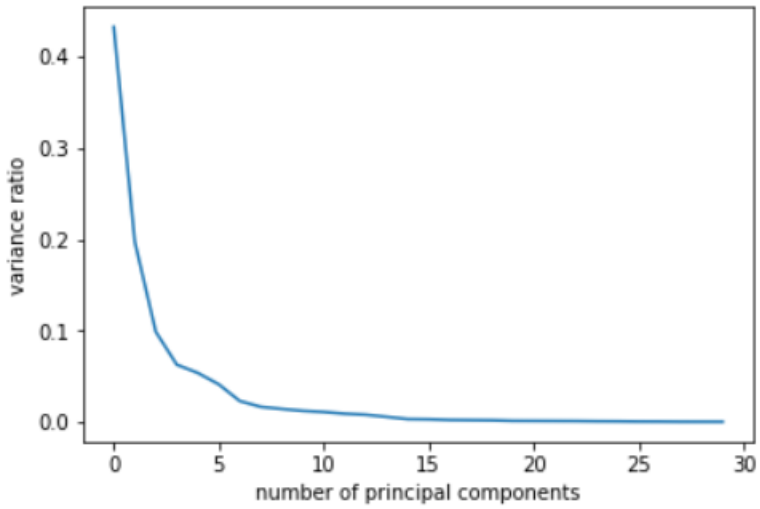
```
X_train, X_test, y_train, y_test = train_test_split(X_st, y, test_size = 0.2, random_state=42)
```

Then we tried to find the optimum number of principal components for PCA:

```
pca = PCA()
pca.fit(X_train)
```

```
PCA(copy=True, iterated_power='auto', n_components=None, random_state=None,
  svd_solver='auto', tol=0.0, whiten=False)
```

```
plt.plot(pca.explained_variance_ratio_)
plt.xlabel('number of principal components')
plt.ylabel('variance ratio')
plt.show()
```



With plotting variance ratio as a function of number of principal components, an optimum N=6 was selected. The PCA was then fit to training data, and applied to both training data and testing data:

```
pca2 = PCA(n_components=6)
pca2.fit(X_train)
X_train_pca = pca2.transform(X_train)
X_test_pca = pca2.transform(X_test)
```

Applying Machine Learning Models

1. KNN model

   When implementing a KNN model, we need to firstly pick the optimum number of neighbors, k. To do this, we applied a cross validation on the training set, aiming to find the optimum k value. As shown below, k=1 was selected:

```
neighbors = np.arange(1, 30)
param_grid = {'n_neighbors':neighbors}
knn_cv = GridSearchCV(KNeighborsClassifier(), param_grid, cv=5)
knn_cv.fit(X_train_pca, y_train)
print(knn_cv.best_params_)
print(svmc_cv.best_score_)
```

```
{'n_neighbors': 1}
0.9692307692307692
```

   Then, we applied the KNN model on the test set, and evaluated the result through confusion matrix and classification report:

```
knn = KNeighborsClassifier(n_neighbors=knn_cv.best_params_['n_neighbors'])
knn.fit(X_train_pca, y_train)
y_pred = knn.predict(X_test_pca)
c = confusion_matrix(y_test, y_pred)
print(c)
print(classification_report(y_test, y_pred))
```

```
[[67  4]
 [ 2 41]]
             precision    recall  f1-score   support

          B       0.97      0.94      0.96        71
          M       0.91      0.95      0.93        43

avg / total       0.95      0.95      0.95       114
```

   An accuracy of ~0.95 is achieved. KNN did a good job, but can other models outperform it? In medical care, 95% is still not high enough for patient concern.

2. Logistic Regression

In Logistic Regression, we also need to find the optimum hyper parameter C.

```
c_space = np.logspace(-5, 8, 20)
param_grid = {'C': c_space}
logreg_cv = GridSearchCV(LogisticRegression(), param_grid, cv=5)
logreg_cv.fit(X_train_pca, y_train)
print("Tuned Logistic Regression Parameters: {}".format(logreg_cv.best_params_))
print("Best score is {}".format(logreg_cv.best_score_))
```

```
Tuned Logistic Regression Parameters: {'C': 0.12742749857031346}
Best score is 0.9714285714285714
```

After C was determined, the model was applied to the testing set:

```
logreg = LogisticRegression(C=logreg_cv.best_params_['C'])
logreg.fit(X_train_pca, y_train)
y_pred = logreg.predict(X_test_pca)
c = confusion_matrix(y_test, y_pred)
print(c)
print(classification_report(y_test, y_pred))
```

```
[[70  1]
 [ 1 42]]
             precision    recall  f1-score   support

          B       0.99      0.99      0.99        71
          M       0.98      0.98      0.98        43

avg / total       0.98      0.98      0.98       114
```

Logistic Regression outperformed KNN model, reaching an accuracy of 98%, which is a pretty good result!

3. SVM

Like the previous two models, we tried to find the optimum C and kernel for SVM:

```
c_space = np.logspace(-1, 10, 1)
kernel_space = ['linear', 'poly', 'rbf', 'sigmoid']
param_grid = {'C': c_space, 'kernel': kernel_space}
svmc_cv = GridSearchCV(SVC(), param_grid, cv=5)
svmc_cv.fit(X_train_pca, y_train)
print(svmc_cv.best_params_)
print(svmc_cv.best_score_)
```

```
{'C': 0.1, 'kernel': 'linear'}
0.9692307692307692
```

A linear kernel with C=0.1 was selected, and applied to the test set:

```
svmc = SVC(C=svmc_cv.best_params_['C'], kernel=svmc_cv.best_params_['kernel'])
svmc.fit(X_train_pca, y_train)
y_pred = svmc.predict(X_test_pca)
c = confusion_matrix(y_test, y_pred)
print(c)
print(classification_report(y_test, y_pred))
```

```
[[70  1]
 [ 2 41]]
             precision    recall  f1-score   support

          B       0.97      0.99      0.98        71
          M       0.98      0.95      0.96        43

avg / total       0.97      0.97      0.97       114
```

An accuracy of 0.97 was reached, slightly higher than KNN, but lower than logistic regression model.