Kaggle Find the perfect data ✨ Notebook ? Is stuck? Ask for help on the forums and the wiki to find a fix or a workaround. ? Please note that updating to Notebook 7 might break some of your extensions.

Hide Extension

Data Processing And building machine learning Model

Steps of Preprocessing of data

Step 1: Import Necessary Libraries, Step 2: Read Data Set, Step 3: Sanity Check of Data Step 4: Exploratory Data Analysis Step 5: Missing Value Treatments Step 6: Outliers Treatment Step 7: Transformation Step 8: Normalization Step 9: Scaling Step 10: Dummy Variable Creation

breast_cancer Data Set

Dataset link: https://www.kaggle.com/datasets/yasserh/breast-cancer-dataset

Breast cancer is the most common cancer amongst women worldwide, representing more than one million new cases annually. In the majority of instances, breast cancer is a disease affecting women aged 50 and older, but it can also occur in men, young women, and those with a family history of breast cancer. Breast cancer is a multifaceted illness with various subtypes, each with its unique characteristics and treatment approaches. Early detection and accurate diagnosis are crucial in the fight against breast cancer, as they significantly impact the success of treatment and the overall prognosis for patients. This comprehensive dataset provides valuable information on various factors and features associated with breast cancer, including tumor characteristics, patient demographics, and other relevant clinical data. Researchers and healthcare professionals can utilize this dataset to study the disease, develop predictive models, and improve understanding of breast cancer risk factors and progression.

Step 1: Import Necessary Library

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib as mt
import matplotlib.pyplot as plt
from sklearn.datasets import load_breast_cancer
import seaborn as sns
pd.pd.pandas.set_option('')
```

In [2]: 0

In [3]: 0

Step 2: Read DataSet

In [4]:
```
breast_cancer = load_breast_cancer()
```

In [5]:
```
# create dataframe
df = pd.DataFrame(breast_cancer.data, columns=breast_cancer.feature_names)
df['target'] = breast_cancer.target

# Preview data.
print("Shape: {shape}".format(shape=df.shape))
print("Features: ")
print(breast_cancer.feature_names)
print("Targets:")
print(breast_cancer.target_names)
```

```
Shape: (569, 31)
Features:
['mean radius' 'mean texture' 'mean perimeter' 'mean area'
 'mean smoothness' 'mean compactness' 'mean concavity'
 'mean concave points' 'mean symmetry' 'mean fractal dimension'
 'radius error' 'texture error' 'perimeter error' 'area error'
 'smoothness error' 'compactness error' 'concavity error'
 'concave points error' 'symmetry error' 'fractal dimension error'
 'worst radius' 'worst texture' 'worst perimeter' 'worst area'
 'worst smoothness' 'worst compactness' 'worst concavity'
 'worst concave points' 'worst symmetry' 'worst fractal dimension']
Targets:
['malignant' 'benign']
```

In [6]:
```
df.columns
```

```
Index(['mean radius', 'mean texture', 'mean perimeter', 'mean area',
       'mean smoothness', 'mean compactness', 'mean concavity',
       'mean concave points', 'mean symmetry', 'mean fractal dimension',
       'radius error', 'texture error', 'perimeter error', 'area error',
       'smoothness error', 'compactness error', 'concavity error',
       'concave points error', 'symmetry error', 'fractal dimension error',
       'worst radius', 'worst texture', 'worst perimeter', 'worst area',
       'worst smoothness', 'worst compactness', 'worst concavity',
       'worst concave points', 'worst symmetry', 'worst fractal dimension',
       'target'],
      dtype='object')
```

In [7]:
```
df.head()
```

Out[7]:

mean radius	mean texture	mean perimeter	mean area	mean smoothness	mean compactness	mean concavity	mean concave points	mean symmetry	mean fractal dimension	radius error	texture error	perimeter error	area error	smoothness error	compactness error	concavity error	concave points error	symmetry error	fractal dimension error	worst radius	worst texture	worst perimeter	worst area	worst smoothness	worst compactness	worst concavity	worst concave points	worst symmetry	worst fractal dimension	target
0	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.3001	0.14710	0.2419	0.07871	1.095	0.9053	8.589	153.40	0.006399	0.04904	0.05373	0.01587	0.03003	0.006193	25.38	17.33	184.60	2019.0	0.16200	0.66560	0.7119	0.2654	0.4601	0.11890	0
1	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.0869	0.07017	0.1812	0.05667	0.5435	0.7339	3.398	74.08	0.005225	0.01308	0.0186	0.00134	0.01389	0.003532	24.99	23.41	158.80	1956.0	0.12380	0.18660	0.2416	0.1860	0.2750	0.08902	0
2	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.1974	0.12790	0.2069	0.05999	0.7456	0.7869	4.585	94.03	0.006150	0.04006	0.03832	0.02058	0.0225	0.004571	23.57	25.53	152.50	1709.0	0.14440	0.42450	0.4504	0.2430	0.3613	0.08758	0
3	11.42	20.38	77.58	386.1	0.14250	0.28390	0.2414	0.10520	0.2597	0.09744	0.4956	1.156	3.445	27.23	0.009110	0.07458	0.05661	0.01867	0.0596	0.009208	14.91	26.50	98.87	567.7	0.20980	0.86663	0.6869	0.2575	0.6638	0.17300	0
4	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.1980	0.10430	0.1809	0.05883	0.7572	0.7813	5.438	94.44	0.011490	0.02461	0.05688	0.01885	0.0176	0.005115	22.54	16.67	152.20	1575.0	0.13740	0.20500	0.4000	0.1625	0.2364	0.07678	0

5 rows × 31 columns

In [8]:
```
df.tail()
```

Out[8]:

mean radius	mean texture	mean perimeter	mean area	mean smoothness	mean compactness	mean concavity	mean concave points	mean symmetry	mean fractal dimension	radius error	texture error	perimeter error	area error	smoothness error	compactness error	concavity error	concave points error	symmetry error	fractal dimension error	worst radius	worst texture	worst perimeter	worst area	worst smoothness	worst compactness	worst concavity	worst concave points	worst symmetry	worst fractal dimension	target
564	21.56	22.39	142.00	1479.0	0.11100	0.11590	0.2439	0.13890	0.1726	0.05623	1.176	1.256	7.673	158.70	0.010300	0.02891	0.05198	0.02454	0.0114	0.004239	25.45	26.40	166.10	2027.0	0.14100	0.21100	0.4264	0.1710	0.3048	0.07851	0
565	20.13	28.25	131.20	1261.0	0.09780	0.10340	0.1400	0.09791	0.1752	0.05533	0.7655	2.463	5.203	99.04	0.005769	0.02423	0.03950	0.01678	0.0127	0.003189	23.690	38.25	155.00	1731.0	0.11660	0.19922	0.3219	0.1628	0.2572	0.06954	0
566	16.60	28.08	108.30	858.1	0.08455	0.10023	0.0973	0.06592	0.1889	0.05766	0.4564	1.075	3.425	48.55	0.005958	0.02274	0.03731	0.00869	0.0188	0.002305	18.980	34.12	126.70	1124.0	0.11390	0.30494	0.4488	0.1418	0.3369	0.08261	0
567	20.60	29.33	140.10	1265.0	0.11780	0.27700	0.3514	0.15200	0.2397	0.07016	0.726	1.595	5.772	86.22	0.006548	0.07431	0.04774	0.02574	0.0268	0.006585	23.160	34.65	162.10	1872.0	0.12390	0.36775	0.4358	0.1980	0.3251	0.10580	0
568	7.76	24.54	47.92	181.0	0.05263	0.04362	0.0000	0.00000	0.1587	0.05884	0.3857	1.428	2.548	19.15	0.007189	0.00466	0.00000	0.00000	0.0205	0.003026	9.456	30.37	59.16	268.6	0.08996	0.06444	0.0000	0.0000	0.2871	0.07039	1

5 rows × 31 columns

Step 3: Sanity Check of Data

In [9]:
```
df.shape
```

Out[9]:
```
(569, 31)
```

In [10]:
```
df.info()
```

```
<class pandas.core.frame.DataFrame>
RangeIndex: 569 entries, 0 to 568
Data columns (total 31 columns):
 #   Column                   Non-Null Count  Dtype
---  ------                   --------------  -----
 0   mean radius              569 non-null    float64
 1   mean texture             569 non-null    float64
 2   mean perimeter           569 non-null    float64
 3   mean area                569 non-null    float64
 4   mean smoothness          569 non-null    float64
 5   mean compactness         569 non-null    float64
 6   mean concavity           569 non-null    float64
 7   mean concave points      569 non-null    float64
 8   mean symmetry            569 non-null    float64
 9   mean fractal dimension   569 non-null    float64
 10  radius error             569 non-null    float64
 11  texture error            569 non-null    float64
 12  perimeter error          569 non-null    float64
 13  area error               569 non-null    float64
 14  smoothness error         569 non-null    float64
 15  compactness error        569 non-null    float64
 16  concavity error          569 non-null    float64
 17  concave points error     569 non-null    float64
 18  symmetry error           569 non-null    float64
 19  fractal dimension error  569 non-null    float64
 20  worst radius             569 non-null    float64
 21  worst texture            569 non-null    float64
 22  worst perimeter          569 non-null    float64
 23  worst area               569 non-null    float64
 24  worst smoothness         569 non-null    float64
 25  worst compactness        569 non-null    float64
 26  worst concavity          569 non-null    float64
 27  worst concave points     569 non-null    float64
 28  worst symmetry           569 non-null    float64
 29  worst fractal dimension  569 non-null    float64
 30  target                   569 non-null    int64
dtype: float64
```

In [11]:
```
df.isnull()
```

Out[11]:
```
mean radius                0
mean texture               0
mean perimeter             0
mean area                  0
mean smoothness            0
mean compactness           0
mean concavity             0
mean concave points        0
mean symmetry              0
mean fractal dimension     0
radius error               0
texture error              0
perimeter error            0
area error                 0
smoothness error           0
compactness error          0
concavity error            0
concave points error       0
symmetry error             0
fractal dimension error    0
worst radius               0
worst texture              0
worst perimeter            0
worst area                 0
worst smoothness           0
worst compactness          0
worst concavity            0
worst concave points       0
worst symmetry             0
worst fractal dimension    0
target                     0
dtype: int64
```

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

Out[12]:
```
mean radius                0
mean texture               0
mean perimeter             0
mean area                  0
mean smoothness            0
mean compactness           0
mean concavity             0
mean concave points        0
mean symmetry              0
mean fractal dimension     0
radius error               0
texture error              0
perimeter error            0
area error                 0
smoothness error           0
compactness error          0
concavity error            0
concave points error       0
symmetry error             0
fractal dimension error    0
worst radius               0
worst texture              0
worst perimeter            0
worst area                 0
worst smoothness           0
worst compactness          0
worst concavity            0
worst concave points       0
worst symmetry             0
worst fractal dimension    0
target                     0
dtype: int64
```

No missing values found. No need to back missing values.

In [13]:
```
df.sanity_duplicate()
```

Out[13]:
```
AttributeError                            Traceback (most recent call last)
Cell In[13], line 1
----> 1 df.sanity_duplicate()

AttributeError: 'DataFrame' object has no attribute 'sanity_duplicate'
```

```
# No duplicates find present there.
df.drop_duplicates(inplace=True)
# drop duplicates and save the dataframe and update df
df.shape(inplace=True) # drop duplicates
```

In [14]:
```
# Identifying garbage value
for c in df.select_dtypes(include=['object'],).columns:
    print(df[c].value_counts())
```

```
Traceback (most recent call last)
AttributeError: 'DataFrame' object has no attribute 'select_dtypes'
```

In [15]: 0 No garbage value

In [16]: 0

Step 4: Exploratory Data Analysis

In [17]:
```
# Descriptive Statistics
df.describe(), #Numerical values
```

Out[17]:

	mean radius	mean texture	mean perimeter	mean area	mean smoothness	mean compactness	mean concavity	mean concave points	mean symmetry	mean fractal dimension	radius error	texture error	perimeter error	area error	smoothness error	compactness error	concavity error	concave points error	symmetry error	fractal dimension error	worst radius	worst texture	worst perimeter	worst area	worst smoothness	worst compactness	worst concavity	worst concave points	worst symmetry	worst fractal dimension	target
count	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000
mean	14.127291	19.289649	91.969033	654.889104	0.096360	0.104341	0.088799	0.048919	0.181162	0.062798	0.405172	1.216853	2.866059	40.337079	0.007041	0.025478	0.031894	0.011796	0.020542	0.003795	16.269190	25.677223	107.261213	880.583128	0.132369	0.254265	0.272188	0.114606	0.290076	0.083946	0.627417
std	3.524049	4.301036	24.298981	351.914129	0.014064	0.052813	0.079720	0.038803	0.027414	0.007060	0.277313	0.551648	2.021855	45.491006	0.003000	0.017908	0.030186	0.006170	0.008266	0.002646	4.833242	6.146258	33.602542	569.356993	0.022832	0.157336	0.208624	0.065732	0.061867	0.018061	0.483918
min	6.981000	9.710000	43.790000	143.500000	0.052630	0.019380	0.000000	0.000000	0.106000	0.049960	0.111500	0.360200	0.757000	6.802000	0.001713	0.002252	0.000000	0.000000	0.007882	0.000895	7.930000	12.020000	50.410000	185.200000	0.071117	0.027290	0.000000	0.000000	0.156500	0.055040	0.000000
max	28.110000	39.280000	188.500000	2501.000000	0.163400	0.345400	0.426800	0.201200	0.304000	0.097440	2.873000	4.885000	21.980000	542.200000	0.031113	0.135400	0.396000	0.052790	0.078950	0.029984	36.040000	49.540000	251.200000	4254.000000	0.222600	1.058000	1.252000	0.291000	0.663800	0.207500	1.000000

5 rows × 31 columns

In [18]:
```
# Need to display the object type data, so object
df.describe(include=['object'])
```

In [19]:
```
# Categorical Descriptive Statistics
df.describe(), #Numerical values

for c in df.select_dtypes(include=['object'],).columns:
    print(df[c].value_counts())
```

In [20]:
```
df.plot(figsize=(,20,20))
```

Out[20]:
```
matplotlib.axes._subplots.AxesSubplot(
<Axes: Column('mean radius'),>,
<Axes: Column('mean texture'),>,
<Axes: Column('mean perimeter'),>,
<Axes: Column('mean area'),>,
<Axes: Column('mean smoothness'),>,
<Axes: Column('mean compactness'),>,
<Axes: Column('mean concavity'),>,
<Axes: Column('mean concave points'),>,
<Axes: Column('mean symmetry'),>,
<Axes: Column('mean fractal dimension'),>,
<Axes: Column('radius error'),>,
<Axes: Column('texture error'),>,
<Axes: Column('perimeter error'),>,
<Axes: Column('area error'),>,
<Axes: Column('smoothness error'),>,
<Axes: Column('compactness error'),>,
<Axes: Column('concavity error'),>,
<Axes: Column('concave points error'),>,
<Axes: Column('symmetry error'),>,
<Axes: Column('fractal dimension error'),>,
<Axes: Column('worst radius'),>,
<Axes: Column('worst texture'),>,
<Axes: Column('worst perimeter'),>,
<Axes: Column('worst area'),>,
<Axes: Column('worst smoothness'),>,
<Axes: Column('worst compactness'),>,
<Axes: Column('worst concavity'),>,
<Axes: Column('worst concave points'),>,
<Axes: Column('worst symmetry'),>,
<Axes: Column('worst fractal dimension'),>,
<Axes: Column('target'),>,
<Axes: Column('target'),>,
<Axes: Column('target
```

```
Outliers in mean area: 0
Outliers in mean perim: 13
Outliers in mean compactness: 16
Outliers in mean concavity: 18
Outliers in mean concave points: 0
Outliers in mean symmetry: 14
Outliers in mean fractal dimension: 15
Outliers in texture error: 20
Outliers in area error: 38
Outliers in smoothness error: 7
Outliers in compactness error: 25
Outliers in concavity error: 19
Outliers in concave points error: 11
Outliers in symmetry error: 14
Outliers in fractal dimension error: 29
Outliers in worst radius: 12
Outliers in worst perim: 5
Outliers in worst area: 35
Outliers in worst compactness: 1
Outliers in worst concavity: 10
Outliers in worst symmetry: 5
Outliers in worst fractal dimension: 12
```

[0]:
```
plot_outliers_boxplot(df,df.shape[1])
plot_outliers_boxplot(df,df.shape[1],hue='target')
```

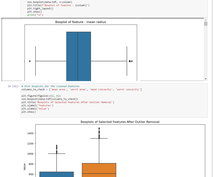
```
# earlier trimmed dataframe
```

Before trimming (old, 5):
After trimming (new, 5):

Outliers are removed successfully from all methods

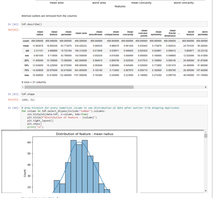
Visualization after outlier trim and dropping duplicates

[0]:
```
# Further detection using Boxplot after outlier trim and dropping duplicates
df = df[df[df.columns[-1]]==df[df.columns[-1]].value_counts()[1]]
df.shape, df.shape[1]
sns.boxplot(df[df.columns[-1]],
            df[df.columns[0]],palette='Set1')
plt.title('Boxplot of Feature - mean radius')
plt.xlabel('Target')
plt.ylabel('Feature')
plt.show()
```



[0]:
```
# Plot explorer for the cleaned dataset
plot_explorer(df,df.shape[1],'worst area','mean concavity','worst concavity')
```

```
((569, 31), 31)
sns.boxplot(data=df[df[df.columns[-1]]==0],
            x='mean radius',y='mean perimeter',hue='target')
plt.title('Boxplot of Selected Features After Outlier Removal')
plt.show()
plt.clf()
plt.close()
```



Minimal outliers are removed from the columns.

[0]:
```
df.describe()
```

[0]:

	mean radius	mean texture	mean perimeter	mean area	mean smoothness	mean compactness	mean concavity	mean concave points	mean symmetry	mean fractal dimension	radius error	texture error	perimeter error	area error	smoothness error	compactness error	concavity error	concave points error	symmetry error	fractal dimension error	worst radius	worst texture	worst perimeter	worst area	worst smoothness	worst compactness	worst concavity	worst concave points	worst symmetry	worst fractal dimension	target
count	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000
mean	14.127291	19.289649	91.969033	654.889104	0.096360	0.104341	0.088799	0.048919	0.181162	0.062798	0.405172	1.216853	2.866059	40.337079	0.007041	0.025478	0.031894	0.011796	0.020542	0.003795	16.269190	25.677223	107.261213	880.583128	0.132369	0.254265	0.272188	0.114606	0.290076	0.083946	0.627417
std	3.524049	4.301036	24.298981	351.914129	0.014064	0.052813	0.079720	0.038803	0.027414	0.007060	0.277313	0.551648	2.021855	45.491006	0.003000	0.017908	0.030186	0.006170	0.008266	0.002646	4.833242	6.146258	33.602542	569.356993	0.022832	0.157336	0.208624	0.065732	0.061867	0.018061	0.483918
min	6.981000	9.710000	43.579000	143.500000	0.052630	0.019380	0.000000	0.000000	0.106000	0.049960	0.111500	0.360200	0.757000	6.802000	0.001713	0.002252	0.000000	0.000000	0.007882	0.000895	7.930000	12.020000	50.410000	185.200000	0.071117	0.027290	0.000000	0.000000	0.156500	0.055040	0.000000
25%	11.700000	16.170000	75.170000	420.300000	0.086370	0.064891	0.029561	0.020310	0.161900	0.057700	0.232400	0.833900	1.606000	17.850000	0.005169	0.013080	0.015090	0.007638	0.018846	0.001954	13.010000	21.080000	84.110000	515.300000	0.116600	0.147200	0.114500	0.064930	0.250400	0.071772	0.000000
50%	13.370000	18.840000	86.240000	551.100000	0.095870	0.092630	0.061540	0.033500	0.179200	0.061540	0.324200	1.108000	2.287000	24.530000	0.006380	0.020450	0.025890	0.010930	0.020050	0.003187	14.970000	25.410000	97.660000	686.500000	0.131300	0.211900	0.226700	0.099930	0.282200	0.080040	1.000000
75%	15.780000	21.080000	104.100000	782.700000	0.105300	0.130400	0.130700	0.074000	0.195700	0.066120	0.478900	1.474000	3.357000	45.190000	0.008146	0.032400	0.042050	0.014710	0.023480	0.004558	18.789000	29.720000	125.400000	1084.000000	0.146000	0.339100	0.382900	0.161400	0.317100	0.092080	1.000000
max	28.110000	39.280000	188.500000	2501.000000	0.163400	0.345400	0.426800	0.201200	0.304000	0.097440	2.873000	4.885000	21.980000	542.200000	0.031113	0.135400	0.396000	0.052790	0.078950	0.029840	36.040000	49.540000	251.200000	4254.000000	0.222600	1.058000	1.252000	0.291000	0.663800	0.207500	1.000000

8 rows × 31 columns

[0]:
```
df.shape
```

[0]:
```
(569, 31)
```

[0]:
```
# Plot explorer for mean radius to see distribution of data after outlier trim dropping duplicates
plot_explorer(df,df.shape[1],'mean radius')
    sns.boxplot(df[df[df.columns[-1]]==0],
                x='mean radius',hue='target')
    plt.title('Boxplot of Selected Features')
    plt.show()
    plt.clf()
    plt.close()
```

Type Numbers and LabelEncoder

[0]:
```
df.dtypes
```

Encoding

[0]:
```
df[df.select_dtypes(bool).columns] = df[df.select_dtypes(bool).columns].astype(int)
```

[0]:
```
df[df.dtypes[df.dtypes.isin(['object'])].index].head()
```

[0]:

	mean radius	mean texture	mean perimeter	mean area	mean smoothness	mean compactness	mean concavity	mean concave points	mean symmetry	mean fractal dimension	radius error	texture error	perimeter error	area error	smoothness error	compactness error	concavity error	concave points error	symmetry error	fractal dimension error	worst radius	worst texture	worst perimeter	worst area	worst smoothness	worst compactness	worst concavity	worst concave points	worst symmetry	worst fractal dimension	target
0	17.99	10.38	122.8	1001.0	0.11840	0.27760	0.3001	0.14710	0.2419	0.07871	1.095	0.9053	8.589	153.4	0.006399	0.04904	0.05373	0.01587	0.03003	0.006193	25.38	17.33	184.6	2019.0	0.1622	0.6656	0.7119	0.2654	0.4601	0.1189	0
1	20.57	17.77	132.9	1326.0	0.08474	0.07864	0.0869	0.07017	0.1812	0.05667	0.5435	0.7339	3.398	74.08	0.005225	0.01308	0.0186	0.0134	0.0139	0.003532	24.99	23.41	158.8	1956.0	0.1238	0.1866	0.2416	0.186	0.275	0.08902	0
2	19.69	21.25	130.0	1203.0	0.10960	0.15990	0.1974	0.12790	0.2069	0.05999	0.7456	0.7869	4.585	94.03	0.006150	0.04006	0.03832	0.02058	0.0225	0.004571	23.57	25.53	152.1	1709.0	0.1444	0.4245	0.4504	0.2243	0.3643	0.08369	0
3	11.42	20.38	77.58	386.1	0.14250	0.28390	0.2414	0.10520	0.2597	0.09744	0.4956	1.156	3.445	27.23	0.009110	0.07458	0.05661	0.01867	0.0596	0.009208	14.91	26.50	98.87	567.7	0.2098	0.8663	0.6869	0.2575	0.6638	0.1730	0
4	20.29	14.34	135.1	1297.0	0.10030	0.13280	0.1980	0.10430	0.1809	0.05883	0.7572	0.7813	5.438	94.44	0.011490	0.02461	0.05688	0.01885	0.0176	0.005115	22.54	16.67	152.1	1575.0	0.1374	0.2050	0.4000	0.1625	0.2364	0.07678	0

```
df[df.dtypes[df.dtypes.isin(['object'])].index].shape
```

```
(569, 0)
```

Here is no need of encoding because no categorical data found.

6. Feature Selection

Common Methods for Feature Selection

Filter Methods:

Statistical Tests: Use statistical tests to evaluate the relationship between each feature and the target variable (e.g., chi-square test for categorical features, ANOVA for continuous features). Correlation Coefficients: Calculate the correlation coefficient between each feature and the target variable and select features with strong correlations.

Wrapper Methods:Embedded Methods are used to. algorithms

[0]:
```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.feature_selection import RFE
from sklearn.ensemble import RandomForestClassifier

# Separate features and target
X = df.drop('target', axis=1)
y = df['target']

# Calculate correlation matrix
correlation_matrix = df.corr()
print("Correlation matrix:")
sns.heatmap(correlation_matrix, annot=True, fmt='.2f', cmap='coolwarm')
plt.show()
```

```
Selected features using RFE:
['mean radius', 'mean perimeter', 'mean concavity', 'mean concave points', 'worst radius', 'worst smoothness', 'worst compac
Feature ranking: [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
Selected features using Random Forest Feature Importance:
['mean radius', 'mean perimeter', 'mean area', 'mean concavity', 'mean concave points', 'worst radius', 'worst perimeter', '
Feature importances: [0.05 0.02 0.04 0.05 0.01 0.02 0.07 0.04 0.01 0.00 0.01 0.00 0.01 0.02 0.00 0.00 0.00 0.00 0.00 0.00 0.06
```

[0]:
```
# Create a DataFrame to store feature importance scores
feature_importance = pd.DataFrame(
    {'Feature': X.columns,
    'Importance': rfe_model.ranking_,
    'Selected': rfe_model.support_,
    'Correlation': correlation_matrix['target'].drop('target')
    ])
feature_importance = feature_importance[feature_importance['Selected'] == True]
feature_importance.sort_values('Importance', ascending=False)
```

[0]:

	Feature	Importance	Selected	Correlation
0	mean radius	1	True	-0.73203
1	mean texture	1	True	-0.415185
2	mean perimeter	1	True	-0.742636
3	mean area	1	True	-0.708984
4	mean smoothness	1	True	-0.358560
5	mean compactness	1	True	-0.596534
6	mean concavity	1	True	-0.696360
7	mean concave points	1	True	-0.776614
8	mean symmetry	1	True	-0.330499
9	mean fractal dimension	1	True	-0.012838
10	radius error	1	True	-0.567134
11	texture error	1	True	-0.008303
12	perimeter error	1	True	-0.557775
13	area error	1	True	-0.548236
14	smoothness error	1	True	-0.067016
15	compactness error	1	True	-0.292992
16	concavity error	1	True	-0.253730
17	concave points error	1	True	-0.408042
18	symmetry error	1	True	-0.006636
19	fractal dimension error	1	True	0.077972
20	worst radius	1	True	-0.776454
21	worst texture	1	True	-0.456903
22	worst perimeter	1	True	-0.782914
23	worst area	1	True	-0.733825
24	worst smoothness	1	True	-0.421465
25	worst compactness	1	True	-0.590998
26	worst concavity	1	True	-0.659610
27	worst concave points	1	True	-0.793566
28	worst symmetry	1	True	-0.416294
29	worst fractal dimension	1	True	-0.323872

Here is no need of encoding because no categorical data found.

6. Feature Selection

Common Methods for Feature Selection

Filter Methods:

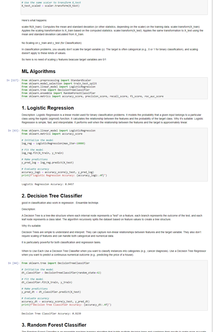
Statistical Tests: Use statistical tests to evaluate the relationship between each feature and the target variable (e.g., chi-square test for categorical features, ANOVA for continuous features). Correlation Coefficients: Calculate the correlation coefficient between each feature and the target variable and select features with strong correlations.

Wrapper Methods:Embedded Methods are used to. algorithms

[0]:
```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.feature_selection import RFE
from sklearn.ensemble import RandomForestClassifier

# Separate features and target
X = df.drop('target', axis=1)
y = df['target']

# Calculate correlation matrix
correlation_matrix = df.corr()
print("Correlation matrix:")
sns.heatmap(correlation_matrix, annot=True, fmt='.2f', cmap='coolwarm')
plt.show()
```




```
Selected features using RFE:
['mean radius', 'mean perimeter', 'mean concavity', 'mean concave points', 'worst radius', 'worst smoothness', 'worst compac
Feature ranking: [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
Selected features using Random Forest Feature Importance:
['mean radius', 'mean perimeter', 'mean area', 'mean concavity', 'mean concave points', 'worst radius', 'worst perimeter', '
Feature importances: [0.05 0.02 0.04 0.05 0.01 0.02 0.07 0.04 0.01 0.00 0.01 0.00 0.01 0.02 0.00 0.00 0.00 0.00 0.00 0.00 0.06
```

[0]:
```
# Create a DataFrame to store feature importance scores
feature_importance = pd.DataFrame(
    {'Feature': X.columns,
    'Importance': rfe_model.ranking_,
    'Selected': rfe_model.support_,
    'Correlation': correlation_matrix['target'].drop('target')
    ])
feature_importance = feature_importance[feature_importance['Selected'] == True]
feature_importance.sort_values('Importance', ascending=False)
```

[0]:

	Feature	Importance	Selected	Correlation
0	mean radius	1	True	-0.73203
1	mean texture	1	True	-0.415185
2	mean perimeter	1	True	-0.742636
3	mean area	1	True	-0.708984
4	mean smoothness	1	True	-0.358560
5	mean compactness	1	True	-0.596534
6	mean concavity	1	True	-0.696360
7	mean concave points	1	True	-0.776614
8	mean symmetry	1	True	-0.330499
9	mean fractal dimension	1	True	-0.012838
10	radius error	1	True	-0.567134
11	texture error	1	True	-0.008303
12	perimeter error	1	True	-0.557775
13	area error	1	True	-0.548236
14	smoothness error	1	True	-0.067016
15	compactness error	1	True	-0.292992
16	concavity error	1	True	-0.253730
17	concave points error	1	True	-0.408042
18	symmetry error	1	True	-0.006636
19	fractal dimension error	1	True	0.077972
20	worst radius	1	True	-0.776454
21	worst texture	1	True	-0.456903
22	worst perimeter	1	True	-0.782914
23	worst area	1	True	-0.733825
24	worst smoothness	1	True	-0.421465
25	worst compactness	1	True	-0.590998
26	worst concavity	1	True	-0.659610
27	worst concave points	1	True	-0.793566
28	worst symmetry	1	True	-0.416294
29	worst fractal dimension	1	True	-0.323872

7. Scaling

Scaling is a crucial preprocessing technique used in data preparation for machine learning and statistical analysis. The main purpose of scaling is to standardize the range of numerical variables in a dataset, so that no single feature dominates the others. This ensures that all features contribute equally and prevents any one feature from exerting undue influence on the model's predictions.

The "StandardScaler" scaling technique transforms the features in a dataset so that they have a mean of zero and a standard deviation of one. This technique is useful when the features have different units or scales, as it ensures that all features are on the same scale and have the same variance.

Why is Scaling Important? Equal Contribution: Features with larger magnitudes can disproportionately influence the results of certain algorithms. Scaling ensures that each feature contributes equally to the analysis.

Improved Model Performance: Many machine learning algorithms converge faster and perform better when the data is scaled. For instance, gradient descent-based optimization algorithms converge more quickly when the features are on a similar scale, leading to faster training times and better model performance.

Distance-based Algorithms: Algorithms like k-Nearest Neighbors (KNN) and Support Vector Machines (SVM) rely on distance calculations. If features are on different scales, the distance metrics can become misleading.

Regularization: Regularization techniques like Lasso and Ridge regression assume that the data is normally distributed. Scaling can help to meet this assumption and improve the effectiveness of the regularization.

Common Scaling Techniques Min-Max Scaling (Normalization):

Range: Rescales the data to a fixed range, typically [0, 1] or [-1, 1]. Formula: Suitable for algorithms that require bounded input values, such as neural networks.

Standardization (Z-score Scaling):

Range: Transforms the data to have a mean of zero and a standard deviation of 1. Use Case: Suitable for algorithms that assume normally distributed data, such as linear regression. Robust Scaling:

IQR

Characteristics: Uses the median and the interquartile range (IQR) to scale features. Handling Outliers: Robust to outliers. Use Case: Effective when the data contains outliers that cannot be removed. Conclusion Scaling is a crucial step in the data preprocessing pipeline, especially when working with machine learning algorithms that are sensitive to the scale of the input features. By applying the appropriate scaling technique, you can improve the performance of your models and ensure more accurate and reliable results.

[0]:
```
# Scale the data using StandardScaler
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Fit the model on the scaled data
model.fit(X_scaled, y)

# Print the coefficients of the model
print("Coefficients:", model.coef_)

# Print the cross-validation scores
cv_scores = cross_val_score(model, X_scaled, y, cv=5)
print("Cross-validation scores:", cv_scores)
```

Here's what happens:

StandardScaler(): Creates the scaler and standard deviation (or other statistics, depending on the scaler) on the training data. scaler.transform(): Applies the scaling transformation to X_train based on the computed statistics. scaler.transform(X_test): Applies the same transformation to X_test, using the statistics from the training data.

For Scaling y_train and y_test (for classification):

For classification problems, you usually don't scale the target variable (y). The target is often categorical (e.g., 0 or 1 for binary classification), and scaling doesn't apply in these cases.

So there's no need to scale y_train because target variables are 0/1.

8. Algorithms

[0]:
```
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report
from sklearn.metrics import accuracy_score,precision_score,recall_score,f1_score,roc_auc_score
```

1. Logistic Regression

Description: Logistic Regression is a linear model used for binary classification problems. It models the probability that a given input belongs to a particular class using the logistic function. It is a simple and interpretable model that works well with linearly separable data. Pros: Simple and interpretable. Fast to train and predict. Cons: Assumes linearity between the features and the log-odds of the target. Not suitable for complex, non-linear relationships. Conclusion: Logistic Regression is a good baseline model for binary classification problems. It provides a simple and interpretable model that can be quickly trained and used for prediction.

[0]:
```
from sklearn.linear_model import LogisticRegression
model = LogisticRegression(max_iter=1000)

# Initialize the model
log_reg = LogisticRegression(max_iter=1000)

# Fit the model on the training data
log_reg.fit(X_train, y_train)

# Make predictions
y_pred_log_reg = log_reg.predict(X_test)

# Evaluate the model
print("Logistic Regression:")
print(classification_report(y_test, y_pred_log_reg))
print("Accuracy:", accuracy_score(y_test, y_pred_log_reg))
```

```
Logistic Regression:
              precision    recall  f1-score   support

           0       0.98       0.95       0.96       106
           1       0.97       0.99       0.98       208

    accuracy                            0.97       314
   macro avg       0.97       0.97       0.97       314
weighted avg       0.97       0.97       0.97       314

Accuracy: 0.9713375796178344
```

2. Decision Tree Classifier

good in classification and cons in regression - Example technique

Description:

A Decision Tree is a tree-like structure where each internal node represents a "test" on a feature, each branch represents the outcome of the test, and each leaf node represents a class label. The algorithm recursively splits the data based on feature values to create a tree structure.

Pros: Easy to understand and interpret. Can handle both numerical and categorical data. Cons: Prone to overfitting. Sensitive to small changes in the data. Can create biased trees if some classes dominate. Conclusion: Decision Trees are a powerful and versatile algorithm that can be used for both classification and regression tasks. It is particularly powerful for tasks where interpretability is important.

[0]:
```
from sklearn.tree import DecisionTreeClassifier

# Initialize the model
dt = DecisionTreeClassifier(random_state=42)

# Fit the model
dt.fit(X_train, y_train)

# Make predictions
y_pred_dt = dt.predict(X_test)

# Evaluate the model
print("Decision Tree:")
print(classification_report(y_test, y_pred_dt))
print("Accuracy:", accuracy_score(y_test, y_pred_dt))
```

```
Decision Tree:
              precision    recall  f1-score   support

           0       0.91       0.91       0.91       106
           1       0.95       0.95       0.95       208

    accuracy                            0.94       314
   macro avg       0.93       0.93       0.93       314
weighted avg       0.94       0.94       0.94       314

Accuracy: 0.9363057324840764
```

3. Random Forest Classifier

Description: Random Forest is an ensemble learning algorithm that builds multiple decision trees and combines their results to make more accurate and robust predictions. It is particularly powerful for both classification and regression tasks.

How Random Forest Works: Multiple Decision Trees:

A Random Forest is essentially a collection of decision trees, each of which is a tree-like structure. Each tree is trained on a random subset of the data. This is a classification task, where the goal is to predict whether a patient has a disease or not (0 for no disease, 1 for disease).

Random Forest uses a technique called bagging (bootstrap aggregating). Bootstrap Sampling: Each tree in the Random Forest is trained on a bootstrap sample of the training data. Sample Variation: Bootstrap sampling means that some data points may be used multiple times in a single tree, while others may not be used at all. This introduces variation in the training data for each tree.

At each node of the tree, only a random subset of features is considered for splitting. Rather than all features. This ensures that each tree is different and reduces the correlation between trees, which improves generalization. Voting:

For classification, after all trees in the forest make their predictions, the Random Forest performs a majority vote to decide the final class label (i.e., the class that most trees predict). The algorithm calculates the average of the predictions from all trees.

Key Parameters:

n_estimators=100: The number of decision trees in the forest. More trees typically lead to better performance, but there is a trade-off with computation time.

random_state=42: Ensures the reproducibility of the results.

max_features: The number of features to consider when splitting a node. Setting it to a square root or log2 reduces correlation between trees and improves model robustness.

[0]:
```
from sklearn.ensemble import RandomForestClassifier

# Initialize the model
rf = RandomForestClassifier(n_estimators=100, random_state=42)

# Fit the model
rf.fit(X_train, y_train)

# Make predictions
y_pred_rf = rf.predict(X_test)

# Evaluate the model
print("Random Forest:")
print(classification_report(y_test, y_pred_rf))
print("Accuracy:", accuracy_score(y_test, y_pred_rf))
```

```
Random Forest:
              precision    recall  f1-score   support

           0       0.97       0.97       0.97       106
           1       0.98       0.98       0.98       208

    accuracy                            0.98       314
   macro avg       0.98       0.98       0.98       314
weighted avg       0.98       0.98       0.98       314

Accuracy: 0.9777777777777777
```

4. Support Vector Machine (SVM)

Support vector machines (SVM) is a powerful and versatile supervised machine learning algorithm used for both classification and regression tasks. It is particularly well-suited for binary classification problems but can also be extended to handle multi-class classification.

Short goal for classification problem is etc.

Key Properties:

Kernel Trick: Specifies the type of kernel to use. For linearly separable data, a linear kernel is appropriate. For non-linear data, kernels like 'rbf' (radial basis function) or 'poly' (polynomial) can be used to map the data into a higher-dimensional space where it becomes linearly separable. Regularization (C): Controls the trade-off between maximizing the decision boundary margin and minimizing classification errors. A smaller C value allows more errors in the training data but results in a smoother decision boundary. A larger C value penalizes misclassifications more heavily, leading to a more complex boundary that fits the training data more closely. This is a hyperparameter that you'll need to tune to find the optimal value for your specific dataset. A larger value of C means that the model will try to classify all the training points correctly, even if it means creating a more complex boundary that may not generalize well to new data. This can lead to overfitting. Conclusion: SVM is a powerful and versatile algorithm that can be used for both classification and regression tasks. Its key properties include the kernel trick, regularization (C), and various other parameters that can be tuned to optimize performance. The choice of kernel is crucial for achieving good results, and it depends on the nature of the data. The regularization parameter C helps to control the trade-off between maximizing the decision boundary margin and minimizing classification errors.

[0]:
```
from sklearn.svm import SVC

# Initialize the model
svm = SVC(random_state=42)

# Fit the model
svm.fit(X_train, y_train)

# Make predictions
y_pred_svm = svm.predict(X_test)

# Evaluate the model
print("Support Vector Machine:")
print(classification_report(y_test, y_pred_svm))
print("Accuracy:", accuracy_score(y_test, y_pred_svm))
```

```
Support Vector Machine:
              precision    recall  f1-score   support

           0       0.98       0.94       0.96       106
           1       0.97       0.99       0.98       208

    accuracy                            0.97       314
   macro avg       0.97       0.97       0.97       314
weighted avg       0.97       0.97       0.97       314

Accuracy: 0.9713375796178344
```

5. k-Nearest Neighbors (k-NN)

This is a supervised learning algorithm used for classification and regression tasks. It's a non-parametric, instance-based learning algorithm. It stores all the training data points and classifies new data points based on the majority vote of their k nearest neighbors in the feature space. Key Properties: Instance-based learning: Instead of making predictions based on learned model parameters, k-NN stores the training data and uses it to make predictions. Distance-based: The algorithm calculates the distance between the new data point and the stored training data points to find the nearest neighbors. The choice of distance metric (e.g., Euclidean distance, Manhattan distance) can significantly impact the performance. Non-parametric: It doesn't make any assumptions about the underlying data distribution. Lazy learning: The algorithm doesn't actually learn a model during the training phase. It simply stores the training data. The computation happens when a prediction is made. Hyperparameters: n_neighbors (k): The number of nearest neighbors to consider when making a prediction. The optimal value of k depends on the dataset and can be tuned using cross-validation. weights: Specifies how to weight the contribution of the neighbors. 'uniform' gives all neighbors equal weight, 'distance' gives more weight to closer neighbors. metric: The distance metric to use. Euclidean distance is the default. Conclusion: k-NN is a simple and intuitive algorithm that can be effective for classification and regression tasks. The choice of k and the distance metric are important hyperparameters that need to be tuned to optimize performance. The algorithm's performance can be sensitive to the scale of the features, so feature scaling is often a good idea.

[0]:
```
from sklearn.neighbors import KNeighborsClassifier

# Initialize the model
knn = KNeighborsClassifier(n_neighbors=5)

# Fit the model
knn.fit(X_train, y_train)

# Make predictions
y_pred_knn = knn.predict(X_test)

# Evaluate the model
print("K-Nearest Neighbors:")
print(classification_report(y_test, y_pred_knn))
print("Accuracy:", accuracy_score(y_test, y_pred_knn))
```

```
K-Nearest Neighbors:
              precision    recall  f1-score   support

           0       0.95       0.92       0.93       106
           1       0.96       0.97       0.97       208

    accuracy                            0.96       314
   macro avg       0.96       0.95       0.95       314
weighted avg       0.96       0.96       0.96       314

Accuracy: 0.9554140127388535
```

Step 4: Summary of Results / Model Comparison

[0]:
```
print("Logistic Regression:", accuracy_score(y_test, y_pred_log_reg))
print("Decision Tree:", accuracy_score(y_test, y_pred_dt))
print("Random Forest Classifier Accuracy:", accuracy_score(y_test, y_pred_rf))
print("Support Vector Machine:", accuracy_score(y_test, y_pred_svm))
print("K-Nearest Neighbors Accuracy:", accuracy_score(y_test, y_pred_knn))
```

```
Logistic Regression: 0.9713375796178344
Decision Tree: 0.9363057324840764
Random Forest Classifier Accuracy: 0.9777777777777777
Support Vector Machine: 0.9713375796178344
K-Nearest Neighbors Accuracy: 0.9554140127388535
```

The classification_report function in scikit-learn provides a detailed report on the precision, recall, F1-score, and support for each class in your dataset. You can use it after making predictions with a classification model.

[0]:
```
from sklearn.metrics import classification_report
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.tree import DecisionTreeClassifier

# Generate and print the classification report
print("Logistic Regression:")
print(classification_report(y_test, y_pred_log_reg))
```

```
              precision    recall  f1-score   support

           0       0.98       0.95       0.96       106
           1       0.97       0.99       0.98       208

    accuracy                            0.97       314
   macro avg       0.97       0.97       0.97       314
weighted avg       0.97       0.97       0.97       314
```

Precision: The proportion of correctly predicted positive instances (true positives) among all predicted positives. Recall (Sensitivity): The proportion of correctly predicted positive instances (true positives) among all actual positives. F1-score: The weighted average of precision and recall, useful for balancing precision and recall. Support: The number of instances in each class in the test dataset. Class-specific scores: Provides scores for each class, which can be crucial for understanding how well the model performs for each class.

Interpretation of Results:

Precision: 0.98: When the model predicts a given class in a category, it is correct 97% of the time. Recall: 0.99: The model correctly identifies 99% of the actual target cases in the test set. The model is slightly better at identifying instances of the target class (class 1). F1-score: 0.98: The model achieves a high balance between precision and recall. Support: The model was tested on 106 instances of class 0 and 208 instances of class 1. These are all important metrics to consider when evaluating a classification model.

[0]:
```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, roc_auc_score, classification_report
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.datasets import load_breast_cancer

# Load the dataset
data = load_breast_cancer()
X, y = data.data, data.target

# Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Standardize the data (scaling)
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Define the models
models = {
    "Logistic Regression": LogisticRegression(max_iter=1000),
    "Decision Tree Classifier": DecisionTreeClassifier(random_state=42),
    "Random Forest Classifier": RandomForestClassifier(random_state=42),
    "Support Vector Machine": SVC(random_state=42),
    "K-Nearest Neighbors": KNeighborsClassifier(n_neighbors=5)
}

# Evaluate each model
for model_name, model in models.items():
    model.fit(X_train_scaled, y_train)
    y_pred = model.predict(X_test_scaled)
    print(f"{model_name}:")
    print(classification_report(y_test, y_pred))
    print("Accuracy:", accuracy_score(y_test, y_pred))
    print("Precision:", precision_score(y_test, y_pred))
    print("Recall:", recall_score(y_test, y_pred))
    print("F1-score:", f1_score(y_test, y_pred))
    print("ROC AUC:", roc_auc_score(y_test, y_pred))
    print("\n")
```

```
Logistic Regression:
              precision    recall  f1-score   support

           0       0.98       0.95       0.96       106
           1       0.97       0.99       0.98       208

    accuracy                            0.97       314
   macro avg       0.97       0.97       0.97       314
weighted avg       0.97       0.97       0.97       314

Accuracy: 0.9713375796178344
Precision: 0.9717514124293785
Recall: 0.9855
F1-score: 0.9786
ROC AUC: 0.9704


Decision Tree Classifier:
              precision    recall  f1-score   support

           0       0.91       0.91       0.91       106
           1       0.95       0.95       0.95       208

    accuracy                            0.94       314
   macro avg       0.93       0.93       0.93       314
weighted avg       0.94       0.94       0.94       314

Accuracy: 0.9363057324840764
Precision: 0.9471153846153846
Recall: 0.9471
F1-score: 0.9471
ROC AUC: 0.9363


Random Forest Classifier:
              precision    recall  f1-score   support

           0       0.97       0.97       0.97       106
           1       0.98       0.98       0.98       208

    accuracy                            0.98       314
   macro avg       0.98       0.98       0.98       314
weighted avg       0.98       0.98       0.98       314

Accuracy: 0.9777777777777777
Precision: 0.9811320754716981
Recall: 0.9811
F1-score: 0.9811
ROC AUC: 0.9777


Support Vector Machine:
              precision    recall  f1-score   support

           0       0.98       0.94       0.96       106
           1       0.97       0.99       0.98       208

    accuracy                            0.97       314
   macro avg       0.97       0.97       0.97       314
weighted avg       0.97       0.97       0.97       314

Accuracy: 0.9713375796178344
Precision: 0.9717514124293785
Recall: 0.9855
F1-score: 0.9786
ROC AU
```