Support Vector Machine

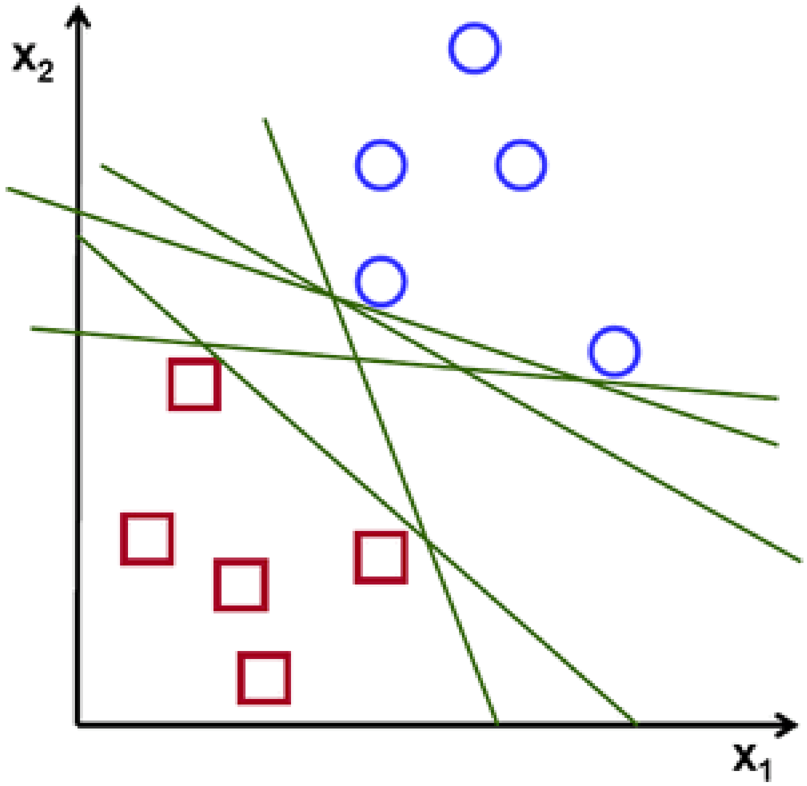
A Support Vector Machine (SVM) is a very powerful and versatile Machine Learning model, capable of performing linear or nonliner classification, regression, and even outlier detection. In this notebook, we will discover the support vector machine algorithm as well as it implementation in scikit-learn. We will also discover the Principal Component Analysis and its implementation with scikit-learn.

1. Support Vector Machine — (SVM)

Support vector machine is another simple algorithm that every machine learning expert should have in his/her arsenal. Support vector machine is highly preferred by many as it produces significant accuracy with less computation power. Support Vector Machine, abbreviated as SVM can be used for both regression and classification tasks. But, it is widely used in classification objectives.

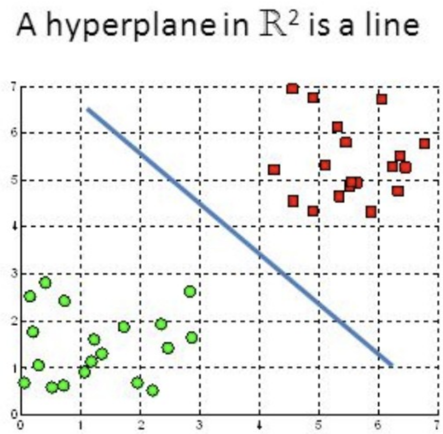
1. 1. What is Support Vector Machine?

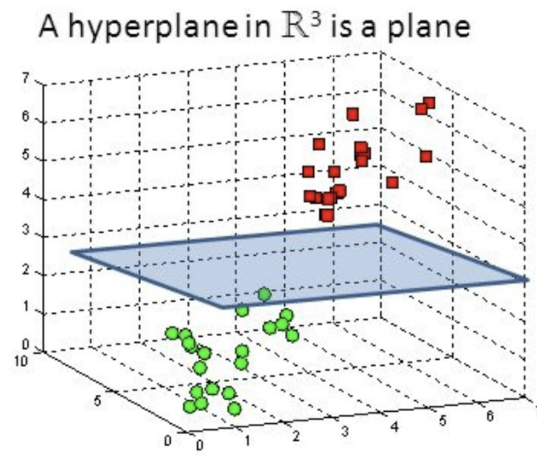
The objective of the support vector machine algorithm is to find a hyperplane in an N-dimensional space(N — the number of features) that distinctly classifies the data points.



To separate the two classes of data points, there are many possible hyperplanes that could be chosen. Our objective is to find a plane that has the maximum margin, i.e the maximum distance between data points of both classes. Maximizing the margin distance provides some reinforcement so that future data points can be classified with more confidence.

1. 2. Hyperplanes and Support Vectors





Hyperplanes are decision boundaries that help classify the data points. Data points falling on either side of the hyperplane can be attributed to different classes. Also, the dimension of the hyperplane depends upon the number of features. If the number of input features is 2, then the hyperplane is just a line. If the number of input features is 3, then the hyperplane becomes a two-dimensional plane. It becomes difficult to imagine when the number of features exceeds 3.

Support vectors are data points that are closer to the hyperplane and influence the position and orientation of the hyperplane. Using these support vectors, we maximize the margin of the classifier. Deleting the support vectors will change the position of the hyperplane. These are the points that help us build our SVM.

1. 3. Large Margin Intuition

In logistic regression, we take the output of the linear function and squash the value within the range of [0,1] using the sigmoid function. If the squashed value is greater than a threshold value(0.5) we assign it a label 1, else we assign it a label 0. In SVM, we take the output of the linear function and if that output is greater than 1, we identify it with one class and if the output is -1, we identify is with another class. Since the threshold values are changed to 1 and -1 in SVM, we obtain this reinforcement range of values([-1,1]) which acts as margin.

SVM Implementation in Python

We will use support vector machine in Predicting if the cancer diagnosis is benign or malignant based on several observations/features.

- 30 features are used, examples: - radius (mean of distances from center to points on the perimeter) - texture (standard deviation of gray-scale values) - perimeter - area - smoothness (local variation in radius lengths) - compactness (perimeter^2 / area - 1.0) - concavity (severity of concave portions of the contour) - concave points (number of concave portions of the contour) - symmetry - fractal dimension ("coastline approximation" - 1)
- Datasets are linearly separable using all 30 input features
- Number of Instances: 569
- Class Distribution: 212 Malignant, 357 Benign
- Target class: - Malignant - Benign

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
```

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import load_breast_cancer

%matplotlib inline
sns.set_style('whitegrid')
```

df = pd.DataFrame(np.c_[cancer.data, cancer.target], columns=col_names): This creates a pandas DataFrame (df) by concatenating the feature data and target variable together. cancer.data contains the feature values, and cancer.target contains the target labels (0 for malignant, 1 for benign).

np.c_: This is a way to concatenate the data and target arrays column-wise.

columns=col_names: This assigns the list of column names (features + target) as the column headers for the DataFrame.

```
cancer.target
```

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1,
1,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0,
0,
       0, 0, 1, 0, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 1, 0,
0,
       1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0,
0,
       1, 1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1, 0,
1,
       1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 0, 1,
0,
       0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1,
1,
       1, 1, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1,
1,
       1, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0, 1, 0,
0,
       0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 1, 0,
0,
       1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 1, 0, 1,
1,
       1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0,
       0, 0, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 1,
1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1,
1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0,
0,
       0, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1,
0,
       0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0,
0,
       1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0, 1,
1,
       1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1,
0,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 1,
1,
       1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0,
0,
       1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1,
1,
       1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 0, 1, 0, 1, 0, 1,
1,
       1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1,
1,
```

```
       1, 1, 1, 1, 1, 0, 1, 0, 0, 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, 0, 0, 0, 0, 0, 0, 1])

cancer = load_breast_cancer()

col_names = list(cancer.feature_names)
col_names.append('target')
df = pd.DataFrame(np.c_[cancer.data, cancer.target],
columns=col_names)
df.head()

   mean radius  mean texture  mean perimeter  mean area  mean
smoothness  \
0        17.99         10.38          122.80     1001.0
0.11840
1        20.57         17.77          132.90     1326.0
0.08474
2        19.69         21.25          130.00     1203.0
0.10960
3        11.42         20.38           77.58      386.1
0.14250
4        20.29         14.34          135.10     1297.0
0.10030

   mean compactness  mean concavity  mean concave points  mean
symmetry  \
0           0.27760          0.3001              0.14710
0.2419
1           0.07864          0.0869              0.07017
0.1812
2           0.15990          0.1974              0.12790
0.2069
3           0.28390          0.2414              0.10520
0.2597
4           0.13280          0.1980              0.10430
0.1809

   mean fractal dimension  ...  worst texture  worst perimeter  worst
area  \
0                 0.07871  ...          17.33           184.60
2019.0
1                 0.05667  ...          23.41           158.80
1956.0
2                 0.05999  ...          25.53           152.50
1709.0
3                 0.09744  ...          26.50            98.87
567.7
4                 0.05883  ...          16.67           152.20
1575.0
```

```
   worst smoothness  worst compactness  worst concavity  worst concave
points  \
0            0.1622             0.6656           0.7119
0.2654
1            0.1238             0.1866           0.2416
0.1860
2            0.1444             0.4245           0.4504
0.2430
3            0.2098             0.8663           0.6869
0.2575
4            0.1374             0.2050           0.4000
0.1625

   worst symmetry  worst fractal dimension  target
0          0.4601                  0.11890     0.0
1          0.2750                  0.08902     0.0
2          0.3613                  0.08758     0.0
3          0.6638                  0.17300     0.0
4          0.2364                  0.07678     0.0

[5 rows x 31 columns]
```

```
print(cancer.target_names)
```

```
['malignant' 'benign']
```

```
df.describe()
```

```
       mean radius  mean texture  mean perimeter    mean area  \
count   569.000000    569.000000      569.000000   569.000000
mean     14.127292     19.289649       91.969033   654.889104
std       3.524049      4.301036       24.298981   351.914129
min       6.981000      9.710000       43.790000   143.500000
25%      11.700000     16.170000       75.170000   420.300000
50%      13.370000     18.840000       86.240000   551.100000
75%      15.780000     21.800000      104.100000   782.700000
max      28.110000     39.280000      188.500000  2501.000000

       mean smoothness  mean compactness  mean concavity  mean concave
points  \
count       569.000000        569.000000      569.000000
569.000000
mean          0.096360          0.104341        0.088799
0.048919
std           0.014064          0.052813        0.079720
0.038803
min           0.052630          0.019380        0.000000
0.000000
25%           0.086370          0.064920        0.029560
0.020310
```

```
50%            0.095870          0.092630        0.061540
0.033500
75%            0.105300          0.130400        0.130700
0.074000
max            0.163400          0.345400        0.426800
0.201200

       mean symmetry  mean fractal dimension  ...  worst texture  \
count     569.000000              569.000000  ...     569.000000
mean        0.181162                0.062798  ...      25.677223
std         0.027414                0.007060  ...       6.146258
min         0.106000                0.049960  ...      12.020000
25%         0.161900                0.057700  ...      21.080000
50%         0.179200                0.061540  ...      25.410000
75%         0.195700                0.066120  ...      29.720000
max         0.304000                0.097440  ...      49.540000

       worst perimeter   worst area  worst smoothness  worst
compactness  \
count       569.000000   569.000000        569.000000
569.000000
mean        107.261213   880.583128          0.132369
0.254265
std          33.602542   569.356993          0.022832
0.157336
min          50.410000   185.200000          0.071170
0.027290
25%          84.110000   515.300000          0.116600
0.147200
50%          97.660000   686.500000          0.131300
0.211900
75%         125.400000  1084.000000          0.146000
0.339100
max         251.200000  4254.000000          0.222600
1.058000

       worst concavity  worst concave points  worst symmetry  \
count       569.000000            569.000000      569.000000
mean          0.272188              0.114606        0.290076
std           0.208624              0.065732        0.061867
min           0.000000              0.000000        0.156500
25%           0.114500              0.064930        0.250400
50%           0.226700              0.099930        0.282200
75%           0.382900              0.161400        0.317900
max           1.252000              0.291000        0.663800

       worst fractal dimension      target
count               569.000000  569.000000
mean                  0.083946    0.627417
std                   0.018061    0.483918
```

```
min                      0.055040    0.000000
25%                      0.071460    0.000000
50%                      0.080040    1.000000
75%                      0.092080    1.000000
max                      0.207500    1.000000

[8 rows x 31 columns]
```

```
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    float64
dtypes: float64(31)
memory usage: 137.9 KB
```

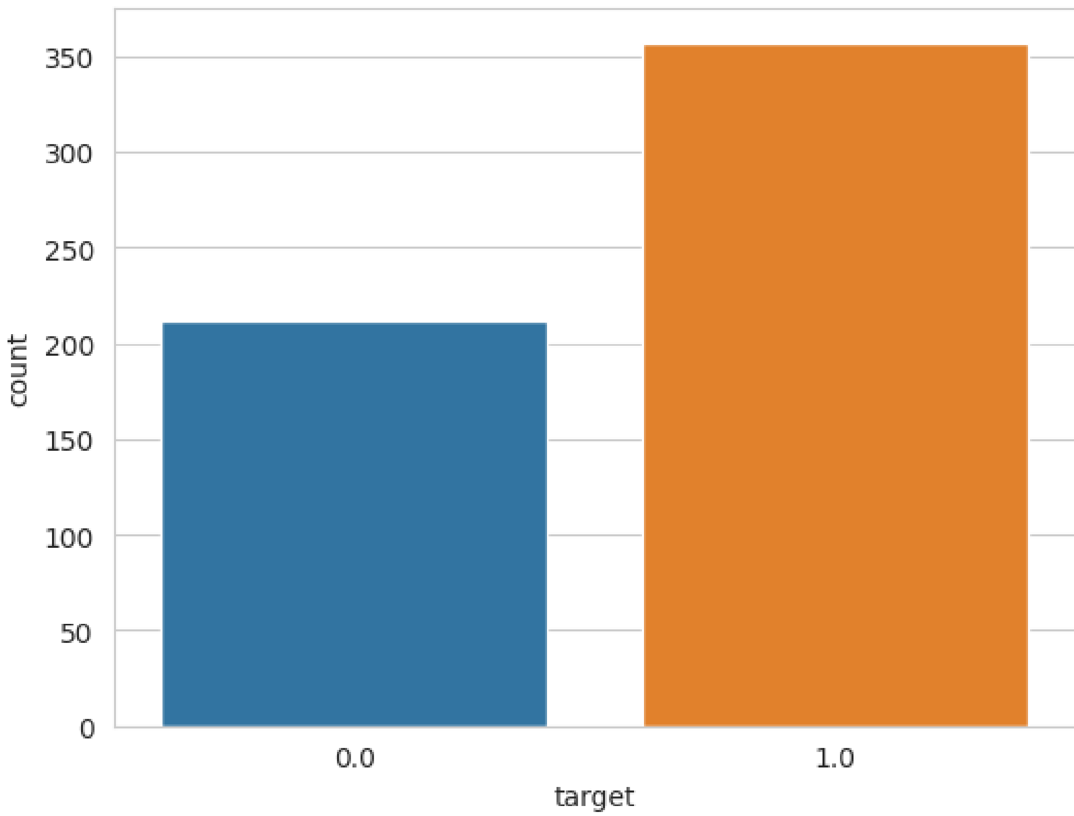
2. 1. VISUALIZING THE DATA

```
df.columns
```

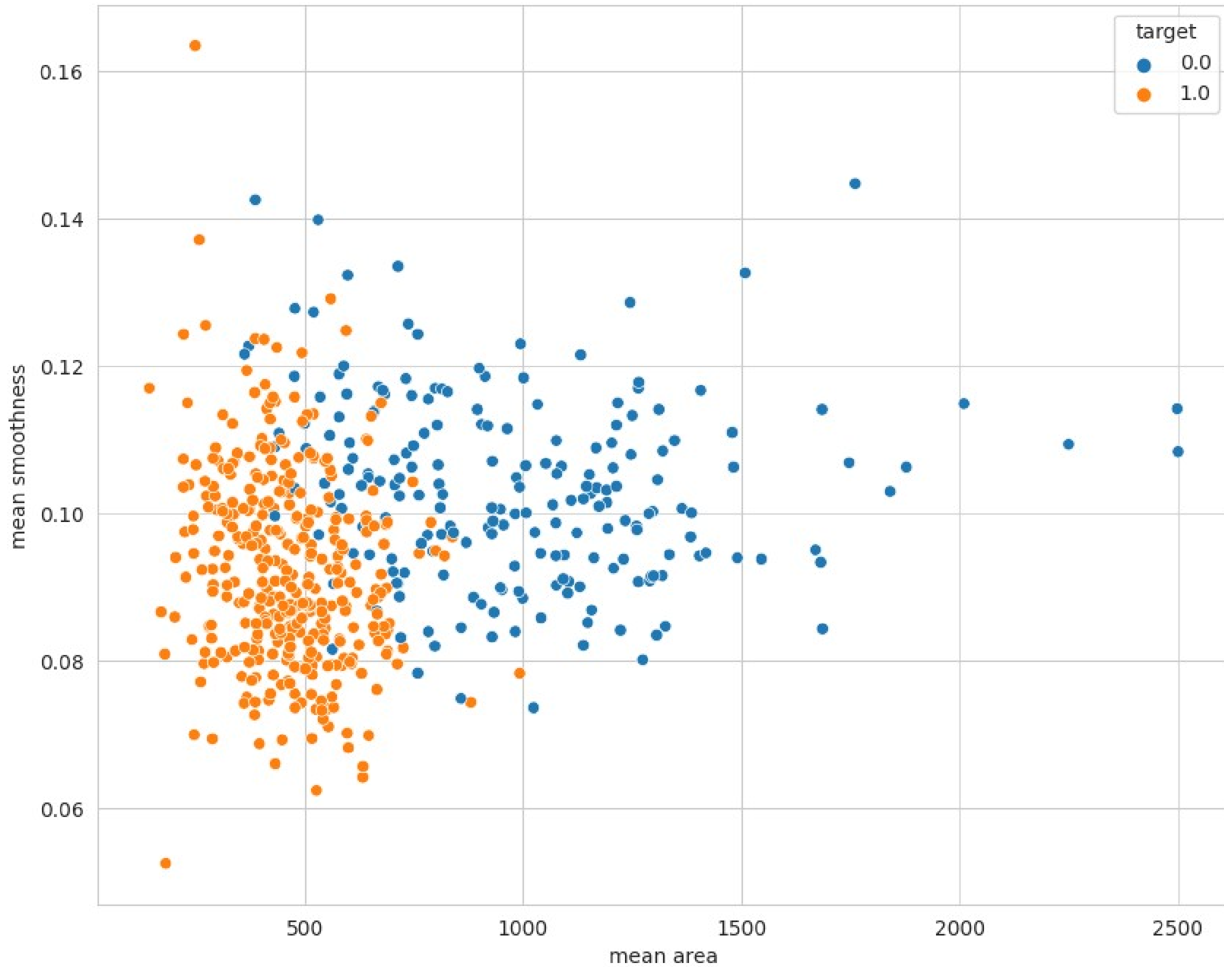
```
Index(['mean radius', 'mean texture', 'mean perimeter', 'mean area',
       'mean smoothness', 'mean compactness', 'mean concavity',
       'mean concave points', 'mean symmetry', 'mean fractal
dimension',
       'radius error', 'texture error', 'perimeter error', 'area
error',
       'smoothness 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')
```

```
#sns.pairplot(df, hue='target', vars=['mean radius', 'mean texture',
'mean perimeter', 'mean area',
                                     #'mean smoothness', 'mean
compactness', 'mean concavity',
                                     #'mean concave points', 'mean
symmetry', 'mean fractal dimension'])

sns.countplot(x=df['target'], label = "Count")
```

```
<Axes: xlabel='target', ylabel='count'>
```

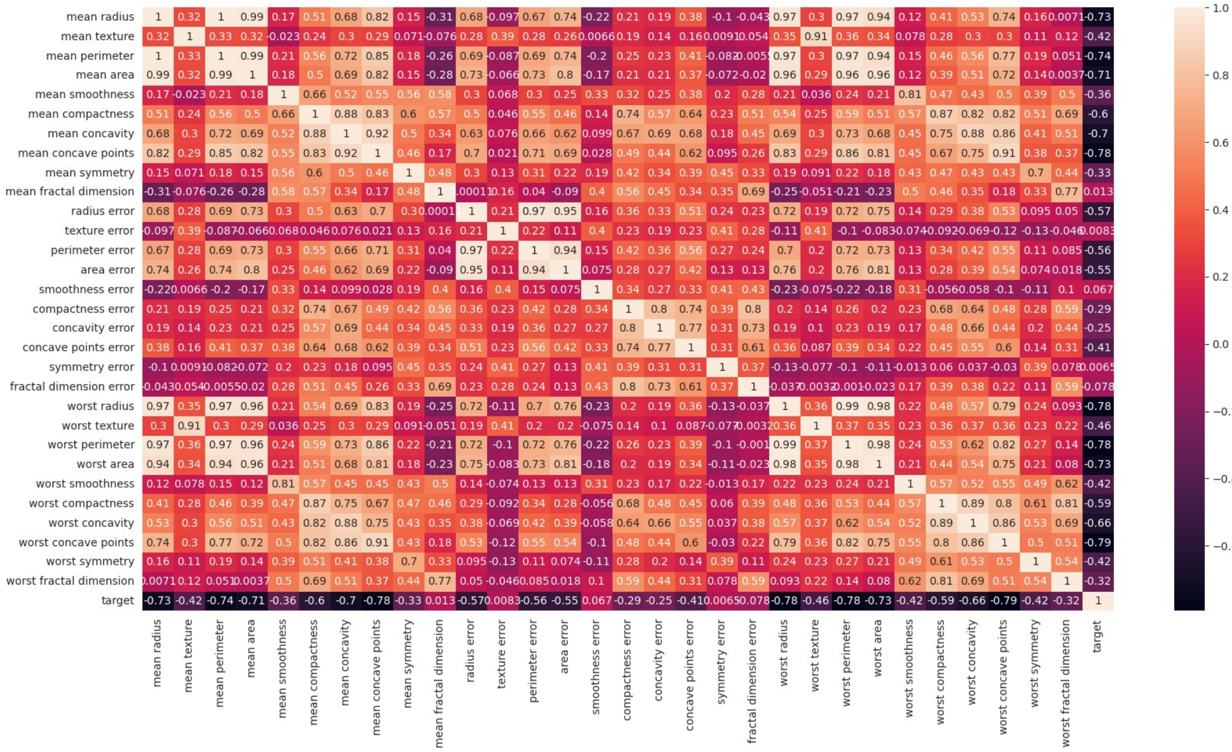



```
plt.figure(figsize=(10, 8))
sns.scatterplot(x = 'mean area', y = 'mean smoothness', hue =
'target', data = df)

<Axes: xlabel='mean area', ylabel='mean smoothness'>
```




```
# Let's check the correlation between the variables
# Strong correlation between the mean radius and mean perimeter, mean
area and mean primeter
plt.figure(figsize=(20,10))
sns.heatmap(df.corr(), annot=True)
```

```
<Axes: >
```



2. 2. MODEL TRAINING (FINDING A PROBLEM SOLUTION)

1. **`X = df.drop('target', axis=1)`:** This creates a new DataFrame `X` by dropping the column named `'target'` from the original DataFrame `df`. This will be your feature set, which includes all columns except the target variable. If you want to remove columns, set axis to 1.

2. **`y = df.target`:** This creates a Series `y` containing the target variable. In this case, it represents the labels (0 for malignant, 1 for benign).

3. **`pipeline = Pipeline([ ('min_max_scaler', MinMaxScaler()), ('std_scaler', StandardScaler()) ])`:** This creates a data preprocessing pipeline. It applies two transformations in sequence: first, it applies Min-Max scaling (`MinMaxScaler()`), and then it applies Standard Scaling (`StandardScaler()`). This pipeline can be used to transform your data before feeding it into a machine learning model.

After running this code, you'll have your preprocessed feature set `X` and corresponding labels `y`. Additionally, you'll have split your data into training and testing sets, with 70% of the data used for training and 30% for testing. The preprocessing pipeline (`pipeline`) is also ready to be applied to your data.

```
from sklearn.model_selection import cross_val_score, train_test_split
#from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler, MinMaxScaler
```

```
X = df.drop('target', axis=1)
y = df.target

#print(f"'X' shape: {X.shape}")
#print(f"'y' shape: {y.shape}")

#pipeline = Pipeline([
#    ('min_max_scaler', MinMaxScaler()),
 #   ('std_scaler', StandardScaler())
#])
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.3, random_state=42)

from sklearn.metrics import accuracy_score, confusion_matrix,
classification_report

def print_score(clf, X_train, y_train, X_test, y_test, train=True):
    if train:
        pred = clf.predict(X_train)
        clf_report = pd.DataFrame(classification_report(y_train, pred,
output_dict=True))
        print("Train Result:\
n================================================")
        print(f"Accuracy Score: {accuracy_score(y_train, pred) *
100:.2f}%")
        print("_______________________________________________")
        print(f"CLASSIFICATION REPORT:\n{clf_report}")
        print("_______________________________________________")
        print(f"Confusion Matrix: \n {confusion_matrix(y_train,
pred)}\n")

    elif train==False:
        pred = clf.predict(X_test)
        clf_report = pd.DataFrame(classification_report(y_test, pred,
output_dict=True))
        print("Test Result:\
n================================================")
        print(f"Accuracy Score: {accuracy_score(y_test, pred) *
100:.2f}%")
        print("_______________________________________________")
        print(f"CLASSIFICATION REPORT:\n{clf_report}")
        print("_______________________________________________")
        print(f"Confusion Matrix: \n {confusion_matrix(y_test, pred)}\
n")
```

2. 3. Support Vector Machines (Kernels)

- `C parameter`: Controlls trade-off between classifying training points correctly and having a smooth decision boundary.
  - Small C (loose) makes cost (penalty) of misclassification low (soft margin)

    - Large C (strict) makes cost of misclassification high (hard margin), forcing the model to explain input data stricter and potentially over it.
- `gamma parameter`: Controlls how far the influence of a single training set reaches.
    - Large gamma: close reach (closer data points have high weight)
    - Small gamma: far reach (more generalized solution)
- `degree parameter` : Degree of the polynomial kernel function (`'poly'`). Ignored by all other kernels.

A common approach to find the right hyperparameter values is to use grid search. It is often faster to first do a very coarse grid search, then a finer grid search around the best values found. Having a good sence of the what each hyperparameter actually does can also help you search in the right part of the hyperparameter space. ****

2. 3. 1. Linear Kernel SVM

1. **`from sklearn.svm import LinearSVC`**: This imports the Linear Support Vector Classifier (LinearSVC) from scikit-learn. This is a linear SVM model used for binary classification.

2. **`model = LinearSVC(loss='hinge')`**: This creates an instance of the LinearSVC model.

    - `loss='hinge'`: This specifies the loss function used by the SVM. The `'hinge'` loss is commonly used for SVMs.
3. **`model.fit(X_train, y_train)`**: This trains the SVM model using the training data (`X_train` for features and `y_train` for labels).

```
from sklearn.svm import LinearSVC

model = LinearSVC(loss='hinge')
model.fit(X_train, y_train)
pred = model.predict(X_test)
accuracy_score(y_test, pred)
clf_report = pd.DataFrame(classification_report(y_test, pred,
output_dict=True))
clf_report
#print_score(model, X_train, y_train, X_test, y_test, train=True)
#print_score(model, X_train, y_train, X_test, y_test, train=False)
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/svm/_base.py:1244:
ConvergenceWarning: Liblinear failed to converge, increase the number
of iterations.
  warnings.warn(
```

	0.0	1.0	accuracy	macro avg	weighted avg
precision	0.684783	1.000000	0.830409	0.842391	0.883867
recall	1.000000	0.731481	0.830409	0.865741	0.830409
f1-score	0.812903	0.844920	0.830409	0.828912	0.833124
support	63.000000	108.000000	0.830409	171.000000	171.000000

```
accuracy_score(y_test, pred)
```

```
0.8304093567251462
```

2. 3. 2. Polynomial Kernel SVM

This code trains a SVM classifier using 2rd degree ploynomial kernel.

`model = SVC(kernel='poly', degree=2, gamma='auto', coef0=1, C=5)`: This creates an instance of the SVC model with the following parameters:

- `kernel='poly'`: This specifies that you're using a polynomial kernel. This kernel is capable of capturing non-linear relationships in the data.
- `degree=2`: This sets the degree of the polynomial. In this case, you're using a polynomial of degree 2.
- `gamma='auto'`: The 'auto' setting means that the value of gamma will be set to 1/n_features. Gamma is a parameter for non-linear kernels (like the polynomial kernel) and controls the influence of individual training samples.
- `coef0=1`: This parameter controls how much the model is influenced by high-degree polynomials. It's particularly relevant for the polynomial kernel.
- `C=5`: This parameter is the regularization parameter. A smaller `C` encourages a larger margin, while a larger `C` encourages a smaller margin but fewer misclassifications.

You've set up your SVM model with a polynomial kernel of degree 2, and you've also specified certain hyperparameters (`gamma`, `coef0`, and `C`) to fine-tune the model's behavior.

```
from sklearn.svm import SVC

# The hyperparameter coef0 controls how much the model is influenced
by high degree ploynomials
model = SVC(kernel='poly', degree=2, coef0 = 1, gamma='auto', C=5)
model.fit(X_train, y_train)
pred = model.predict(X_test)
accuracy_score(y_test, pred)
#print_score(model, X_train, y_train, X_test, y_test, train=True)
#print_score(model, X_train, y_train, X_test, y_test, train=False)
```

```
0.9707602339181286
```

2. 3. 3. Radial Kernel SVM

Just like the polynomial features method, the similarity features can be useful with any

```python
model = SVC(kernel='rbf', gamma=0.5, C=0.1)
model.fit(X_train, y_train)

print_score(model, X_train, y_train, X_test, y_test, train=True)
print_score(model, X_train, y_train, X_test, y_test, train=False)
```

```
Train Result:
================================================
Accuracy Score: 62.56%
_______________________________________________
CLASSIFICATION REPORT:
             0.0         1.0  accuracy   macro avg  weighted avg
precision    0.0    0.625628  0.625628    0.312814      0.391411
recall       0.0    1.000000  0.625628    0.500000      0.625628
f1-score     0.0    0.769706  0.625628    0.384853      0.481550
support    149.0  249.000000  0.625628  398.000000    398.000000
_______________________________________________
Confusion Matrix:
 [[  0 149]
 [  0 249]]

Test Result:
================================================
Accuracy Score: 63.16%
_______________________________________________
CLASSIFICATION REPORT:
            0.0         1.0  accuracy   macro avg  weighted avg
precision   0.0    0.631579  0.631579    0.315789      0.398892
recall      0.0    1.000000  0.631579    0.500000      0.631579
f1-score    0.0    0.774194  0.631579    0.387097      0.488964
support    63.0  108.000000  0.631579  171.000000    171.000000
_______________________________________________
Confusion Matrix:
 [[  0  63]
 [  0 108]]


/usr/local/lib/python3.10/dist-packages/sklearn/metrics/
_classification.py:1344: UndefinedMetricWarning: Precision and F-score
are ill-defined and being set to 0.0 in labels with no predicted
samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classificatio
n.py:1344: UndefinedMetricWarning: Precision and F-score are ill-
defined and being set to 0.0 in labels with no predicted samples. Use
`zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classificatio
n.py:1344: UndefinedMetricWarning: Precision and F-score are ill-
defined and being set to 0.0 in labels with no predicted samples. Use
```

```
`zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classificatio
n.py:1344: UndefinedMetricWarning: Precision and F-score are ill-
defined and being set to 0.0 in labels with no predicted samples. Use
`zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classificatio
n.py:1344: UndefinedMetricWarning: Precision and F-score are ill-
defined and being set to 0.0 in labels with no predicted samples. Use
`zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classificatio
n.py:1344: UndefinedMetricWarning: Precision and F-score are ill-
defined and being set to 0.0 in labels with no predicted samples. Use
`zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
```

Other kernels exist but are not used much more rarely. For example, some kernels are specialized for specific data structures. string kernels are sometimes used when classifying text document on DNA sequences.

With so many kernels to choose from, how can you decide which one to use? As a rule of thumb, you should always try the linear kernel first, especially if the training set is very large or if it has plenty of features. If the training set is not too large, you should try the Gaussian RBF kernel as well.

2. 4. Data Preparation for SVM

This section lists some suggestions for how to best prepare your training data when learning an SVM model.

- **Numerical Inputs:** SVM assumes that your inputs are numeric. If you have categorical inputs you may need to covert them to binary dummy variables (one variable for each category).
- **Binary Classification:** Basic SVM as described in this post is intended for binary (two-class) classification problems. Although, extensions have been developed for regression and multi-class classification.

```
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

print("=======================Linear Kernel
SVM=========================")
model = SVC(kernel='linear')
model.fit(X_train, y_train)

print_score(model, X_train, y_train, X_test, y_test, train=True)
print_score(model, X_train, y_train, X_test, y_test, train=False)
```

```python
print("=======================Polynomial Kernel SVM==========================")
from sklearn.svm import SVC

model = SVC(kernel='poly', degree=2, gamma='auto')
model.fit(X_train, y_train)

print_score(model, X_train, y_train, X_test, y_test, train=True)
print_score(model, X_train, y_train, X_test, y_test, train=False)

print("=======================Radial Kernel SVM==========================")
from sklearn.svm import SVC

model = SVC(kernel='rbf', gamma=1)
model.fit(X_train, y_train)

print_score(model, X_train, y_train, X_test, y_test, train=True)
print_score(model, X_train, y_train, X_test, y_test, train=False)
```

```
=======================Linear Kernel SVM==========================
Train Result:
================================================
Accuracy Score: 98.99%
_______________________________________________
CLASSIFICATION REPORT:
                  0.0         1.0  accuracy   macro avg  weighted avg
precision    1.000000    0.984190   0.98995    0.992095      0.990109
recall       0.973154    1.000000   0.98995    0.986577      0.989950
f1-score     0.986395    0.992032   0.98995    0.989213      0.989921
support    149.000000  249.000000   0.98995  398.000000    398.000000
_______________________________________________
Confusion Matrix:
 [[145   4]
 [  0 249]]

Test Result:
================================================
Accuracy Score: 97.66%
_______________________________________________
CLASSIFICATION REPORT:
                 0.0         1.0  accuracy   macro avg  weighted avg
precision   0.968254    0.981481  0.976608    0.974868      0.976608
recall      0.968254    0.981481  0.976608    0.974868      0.976608
f1-score    0.968254    0.981481  0.976608    0.974868      0.976608
support    63.000000  108.000000  0.976608  171.000000    171.000000
_______________________________________________
Confusion Matrix:
 [[ 61   2]
```

```
 [  2 106]]

=======================Polynomial Kernel SVM==========================
Train Result:
================================================
Accuracy Score: 85.18%
_______________________________________________
CLASSIFICATION REPORT:
                  0.0         1.0  accuracy   macro avg  weighted avg
precision    0.978723    0.812500  0.851759    0.895612      0.874729
recall       0.617450    0.991968  0.851759    0.804709      0.851759
f1-score     0.757202    0.893309  0.851759    0.825255      0.842354
support    149.000000  249.000000  0.851759  398.000000    398.000000
_______________________________________________
Confusion Matrix:
 [[ 92  57]
 [  2 247]]

Test Result:
================================================
Accuracy Score: 82.46%
_______________________________________________
CLASSIFICATION REPORT:
                 0.0         1.0  accuracy   macro avg  weighted avg
precision   0.923077    0.795455  0.824561    0.859266      0.842473
recall      0.571429    0.972222  0.824561    0.771825      0.824561
f1-score    0.705882    0.875000  0.824561    0.790441      0.812693
support    63.000000  108.000000  0.824561  171.000000    171.000000
_______________________________________________
Confusion Matrix:
 [[ 36  27]
 [  3 105]]

=======================Radial Kernel SVM==========================
Train Result:
================================================
Accuracy Score: 100.00%
_______________________________________________
CLASSIFICATION REPORT:
             0.0    1.0  accuracy  macro avg  weighted avg
precision    1.0    1.0       1.0        1.0           1.0
recall       1.0    1.0       1.0        1.0           1.0
f1-score     1.0    1.0       1.0        1.0           1.0
support    149.0  249.0       1.0      398.0         398.0
_______________________________________________
Confusion Matrix:
 [[149   0]
 [  0 249]]

Test Result:
```

```
============================================
Accuracy Score: 63.74%
_______________________________________________
CLASSIFICATION REPORT:
                0.0         1.0  accuracy   macro avg  weighted avg
precision   1.000000    0.635294  0.637427    0.817647      0.769659
recall      0.015873    1.000000  0.637427    0.507937      0.637427
f1-score    0.031250    0.776978  0.637427    0.404114      0.502236
support    63.000000  108.000000  0.637427  171.000000    171.000000
_______________________________________________
Confusion Matrix:
 [[  1  62]
 [  0 108]]
```

3. Support Vector Machine Hyperparameter tuning

1. **`from sklearn.model_selection import GridSearchCV`:** This imports the `GridSearchCV` class from scikit-learn. `GridSearchCV` is a way to systematically search through a grid of hyperparameters and find the best combination based on cross-validated performance.

2. **`param_grid`:** This is a dictionary containing the hyperparameters you want to tune. For each hyperparameter, you've provided a list of possible values to try.

   - `C`: Regularization parameter.
   - `gamma`: Kernel coefficient for 'rbf', 'poly', and 'sigmoid'.
   - `kernel`: Specifies the kernel type to be used in the algorithm.

3. **`grid = GridSearchCV(SVC(), param_grid, refit=True, verbose=1, cv=5)`:** This creates an instance of `GridSearchCV`.

   - `SVC()`: This is the estimator, in this case, an SVC model.
   - `param_grid`: The dictionary of hyperparameters and their possible values.
   - `refit=True`: This means that after finding the best hyperparameters, it will refit the model on the full dataset.
   - `verbose=1`: It provides some progress information during the search.
   - `cv=5`: This is the number of cross-validation folds to use during the search. In this case, it's using 5-fold cross-validation.

4. **`grid.fit(X_train, y_train)`:** This fits the grid search to the training data. It will train and evaluate the model for all combinations of hyperparameters in the `param_grid`.

5. **`best_params = grid.best_params_`**: This retrieves the best hyperparameters found during the grid search.

6. **`svm_clf = SVC(**best_params)`**: This creates an instance of the SVC model using the best hyperparameters found by the grid search.

After running this code, `svm_clf` will be an SVC model with the best hyperparameters found through the grid search. You can then use this model for predictions on new data. This approach helps you find the optimal combination of hyperparameters for your SVM model, which can lead to improved performance.

```
from sklearn.model_selection import GridSearchCV

param_grid = {'C': [0.01, 0.1, 0.5, 1, 10, 100],
              'gamma': [1, 0.75, 0.5, 0.25, 0.1, 0.01, 0.001],
              'kernel': ['rbf', 'poly', 'linear']}

grid = GridSearchCV(SVC(), param_grid, refit=True, verbose=1, cv=5)
grid.fit(X_train, y_train)

best_params = grid.best_params_
print(f"Best params: {best_params}")

svm_clf = SVC(**best_params)
svm_clf.fit(X_train, y_train)
print_score(svm_clf, X_train, y_train, X_test, y_test, train=True)
print_score(svm_clf, X_train, y_train, X_test, y_test, train=False)
```

```
Fitting 5 folds for each of 126 candidates, totalling 630 fits
Best params: {'C': 0.1, 'gamma': 1, 'kernel': 'linear'}
Train Result:
================================================
Accuracy Score: 98.24%
_______________________________________________
CLASSIFICATION REPORT:
                  0.0         1.0  accuracy   macro avg  weighted avg
precision    0.986301    0.980159  0.982412    0.983230      0.982458
recall       0.966443    0.991968  0.982412    0.979205      0.982412
f1-score     0.976271    0.986028  0.982412    0.981150      0.982375
support    149.000000  249.000000  0.982412  398.000000    398.000000
_______________________________________________
Confusion Matrix:
 [[144   5]
 [  2 247]]

Test Result:
================================================
Accuracy Score: 98.25%
_______________________________________________
CLASSIFICATION REPORT:
```

```
                  0.0         1.0  accuracy   macro avg  weighted avg
precision   0.983871    0.981651  0.982456    0.982761      0.982469
recall      0.968254    0.990741  0.982456    0.979497      0.982456
f1-score    0.976000    0.986175  0.982456    0.981088      0.982426
support    63.000000  108.000000  0.982456  171.000000    171.000000
_______________________________________________
Confusion Matrix:
 [[ 61   2]
 [  1 107]]
```

5. Summary

In this notebook you discovered the Support Vector Machine Algorithm for machine learning. You learned about:

- What is support vector machine?.
- Support vector machine implementation in Python.
- Support Vector Machine kernels (Linear, Polynomial, Radial).
- How to prepare the data for support vector machine algorithm.
- Support vector machine hyperparameter tuning.

References:

- Support Vector Machine — Introduction to Machine Learning Algorithms
- Support Vector Machines for Machine Learning
- Support Vector Machines documentations