

logistic-regression

January 21, 2024

0.1 Logistic Regression Tutorial By Neuraldemy - Amritesh Kumar

In this tutorial, we are going to implement what we learnt in theory in our tutorial. Sklearn has two classes `LogisticRegression` and `LogisticRegressionCV` for logistic regression and we are going to see how we can implement them on some datasets.

```
[11]: from sklearn.datasets import make_classification
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# Generate dummy data
X, y = make_classification(n_samples=1000, n_features=5, n_classes=2,
↳random_state=42)
```

We are using `make_classification` class to generate a random 2-class classification problem. The params:

```
n_samples=100, n_features=20, *, n_informative=2, n_redundant=2, n_repeated=0,
n_classes=2, n_clusters_per_class=2, weights=None, flip_y=0.01, class_sep=1.0,
hypercube=True, shift=0.0, scale=1.0, shuffle=True, random_state=None
```

This initially creates clusters of points normally distributed ($\text{std}=1$) about vertices of an $n_{\text{informative}}$ -dimensional hypercube with sides of length $2 \times \text{class_sep}$ and assigns an equal number of clusters to each class. It introduces interdependence between these features and adds various types of further noise to the data.

Without shuffling, X horizontally stacks features in the following order: the primary $n_{\text{informative}}$ features, followed by $n_{\text{redundant}}$ linear combinations of the informative features, followed by n_{repeated} duplicates, drawn randomly with replacement from the informative and redundant features. The remaining features are filled with random noise. Thus, without shuffling, all useful features are contained in the columns $X[:, :n_{\text{informative}} + n_{\text{redundant}} + n_{\text{repeated}}]$.

```
[6]: X.shape
```

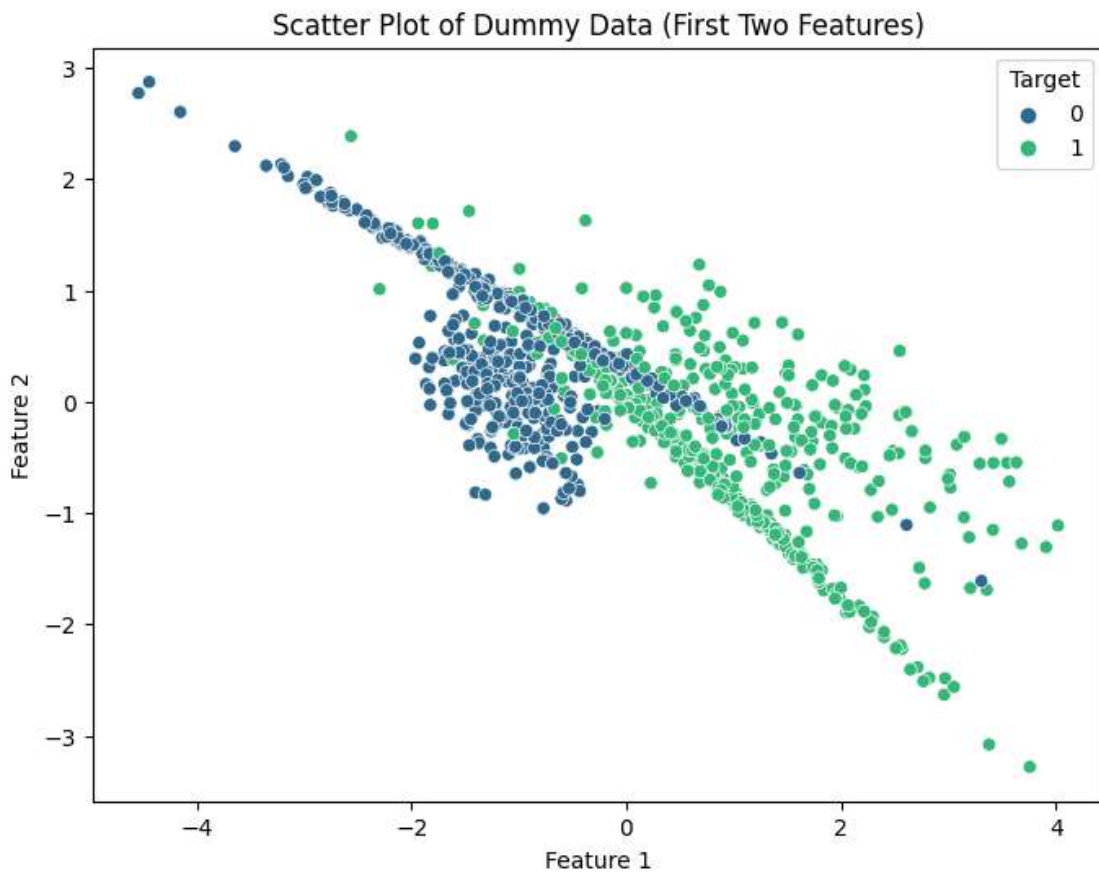
```
[6]: (1000, 5)
```

```
[7]: y.shape
```

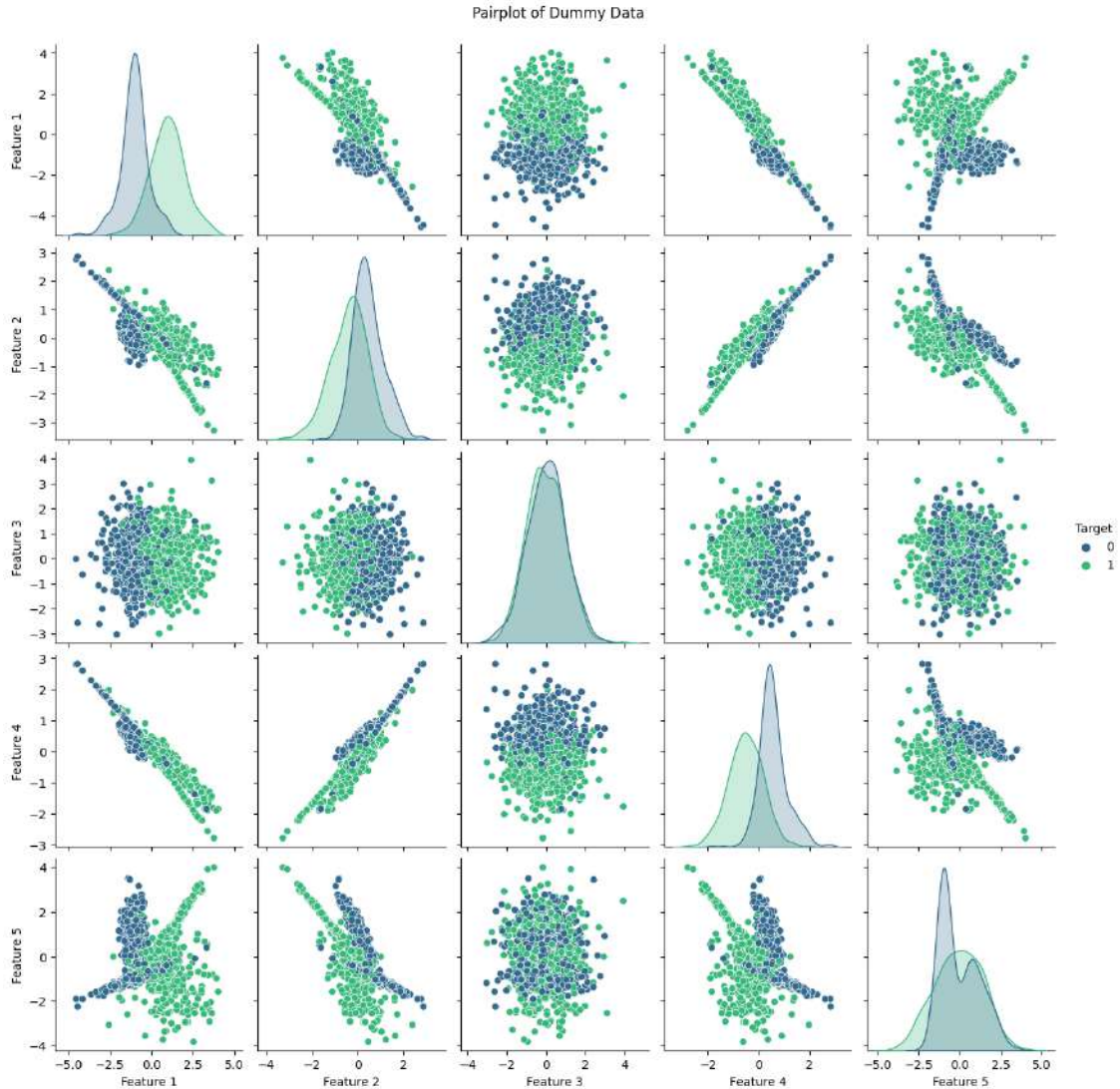
[7]: (1000,)

```
[12]: # convert our dataset into a dataframe for visualization
df = pd.DataFrame(X, columns=['Feature 1', 'Feature 2', 'Feature 3', 'Feature_
↳4', 'Feature 5'])
df['Target'] = y

# Plot the data using the first two features
plt.figure(figsize=(8, 6))
sns.scatterplot(x='Feature 1', y='Feature 2', hue='Target', data=df,
↳palette='viridis')
plt.title('Scatter Plot of Dummy Data (First Two Features)')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.show()
```



```
[13]: sns.pairplot(df, hue='Target', palette='viridis')
plt.suptitle('Pairplot of Dummy Data', y=1.02)
plt.show()
```



The diagonal plots use KDE to estimate the probability density function of each individual feature. The diagonal plots provide a quick visual assessment of the distribution of each feature. If a feature follows a Gaussian-like distribution, it suggests that the data values are concentrated around the mean, and the distribution is symmetric. The appearance of a Gaussian-like shape in these plots does not imply that the actual distribution of the data is exactly Gaussian.

On a real dataset, you will have to perform data analysis, feature selection, feature engineering etc but for the sake of tutorial, we are not going to include that part yet, you will learn those steps by practicing projects. let's talk about Logistic Regression class.

0.1.1 LogisticRegression

Logistic Regression (aka logit, MaxEnt) classifier.

Parameters:

1. **penalty:**

- Determines the type of regularization applied to the model.
- Options:
 - 'l1': Adds an L1 penalty term (encourages sparsity).
 - 'l2': Adds an L2 penalty term (default choice).
 - 'elasticnet': Combines both L1 and L2 penalty terms.
 - None: No penalty is added.

Some penalties may not work with some solvers

2. **dual:**

- Specifies whether to use the dual formulation (constrained) or primal formulation (regularized).
- **False** is recommended when the number of samples is greater than the number of features.

3. **tol:**

- Tolerance for stopping criteria during optimization. It determines when to stop the optimization process.

4. **C:**

- Inverse of the regularization strength.
- Smaller values of **C** result in stronger regularization, preventing overfitting.

5. **fit_intercept:**

- Specifies if a constant (bias or intercept) should be added to the decision function.
- If **True**, a constant term is added; if **False**, no intercept is added.

6. **intercept_scaling:**

- Used when the solver 'liblinear' is used and **fit_intercept** is set to **True**.
- It scales the synthetic feature weight, which is added when intercept is included.

7. **class_weight:**

- Weights associated with classes.
- {**class_label**: **weight**} specifies weights for each class.
- 'balanced' adjusts weights inversely proportional to class frequencies in the input data.

8. **random_state:**

- Seed for random number generation. Ensures reproducibility when solver is 'sag', 'saga', or 'liblinear'.

9. **solver:**

- Algorithm used in the optimization problem.
- Choices:
 - 'lbfgs': Limited-memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS) optimization.
 - 'liblinear': Library for large linear classification.
 - 'newton-cg': Newton Conjugate Gradient optimization.
 - 'sag': Stochastic Average Gradient descent.
 - 'saga': Stochastic Average Gradient descent with regularization.
- Selection depends on the dataset size and structure.
 - For small datasets, 'liblinear' is a good choice, whereas 'sag' and 'saga' are faster for large ones;
 - For multiclass problems, only 'newton-cg', 'sag', 'saga' and 'lbfgs' handle multinomial loss;
 - 'liblinear' is limited to one-versus-rest schemes.
 - 'newton-cholesky' is a good choice for $n_samples \gg n_features$, especially with

one-hot encoded categorical features with rare categories. Note that it is limited to binary classification and the one-versus-rest reduction for multiclass classification. Be aware that the memory usage of this solver has a quadratic dependency on `n_features` because it explicitly computes the Hessian matrix.

10. **max_iter:**
 - Maximum number of iterations for the solvers to converge.
11. **multi_class:**
 - Approach to handling multiple classes.
 - Choices:
 - 'ovr': One-vs-rest (binary problem for each label).
 - 'multinomial': Minimizes multinomial loss across entire probability distribution.
 - 'auto' selects 'ovr' for binary data and 'multinomial' otherwise.
12. **verbose:**
 - Controls the verbosity of the output. Positive values give more detailed output.
13. **warm_start:**
 - When set to `True`, reuses the solution of the previous call to `fit` as initialization.
14. **n_jobs:**
 - Number of CPU cores used when parallelizing over classes (ignored for 'liblinear').
 - Set to `-1` to use all processors.
15. **l1_ratio:**
 - Elastic-Net mixing parameter for `penalty='elasticnet'`.
 - Controls the balance between L1 and L2 penalties.
 - When `l1_ratio=0`, it's equivalent to using L2 penalty ('l2').
 - When `l1_ratio=1`, it's equivalent to using L1 penalty ('l1').
 - For $0 < \text{l1_ratio} < 1$, it's a combination of L1 and L2 penalties.

Please read and if you feel stuck check Sklearn documentation or ask in our community forum. Now, let's apply it on our dataset above but first we need to divide it into train and test set so that we can see how our model performs.

```
[23]: df.head()
```

```
[23]:
```

	Feature 1	Feature 2	Feature 3	Feature 4	Feature 5	Target
0	-0.439643	0.542547	-0.822420	0.401366	-0.854840	0
1	2.822231	-2.480859	-1.147691	-2.101131	3.040278	1
2	1.618386	-1.369478	-2.084113	-1.179659	1.613602	1
3	1.659048	-0.615202	1.112688	-0.835098	-0.272205	1
4	1.849824	-1.679456	-0.926698	-1.402509	2.123129	1

```
[24]: df["Target"].value_counts()
```

```
[24]: Target
0      500
1      500
Name: count, dtype: int64
```

We have a balanced class here.

```
[18]: from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2,
↳random_state = 42)
print(X_train.shape)
print(X_test.shape)
print(y_train.shape)
print(y_test.shape)
```

```
(800, 5)
(200, 5)
(800,)
(200,)
```

```
[26]: from sklearn.linear_model import LogisticRegression

# read sklearn documentation choose your hyperparameters according to the
↳problem
# Don't worry about the solver as we will learn about them in future tutorial.
# While choosing a solver do check for the penalty combination in Sklearn

clf = LogisticRegression(C = 100, random_state = 42)

# fit the training data
clf.fit(X_train, y_train)
```

```
[26]: LogisticRegression(C=100, random_state=42)
```

The underlying C implementation in sklearn uses a random number generator to select features when fitting the model. It is thus not uncommon, to have slightly different results for the same input data. If that happens, try with a smaller tol parameter. Let's check for the class attributes:

1. **classes_ (ndarray of shape (n_classes,)):**
 - A list of class labels known to the classifier.
 - For a binary classification problem, it will have two elements.
 - For a multiclass problem, it will contain all unique class labels.
2. **coef_ (ndarray of shape (1, n_features) or (n_classes, n_features)):**
 - Coefficients of the features in the decision function.
 - If the problem is binary, it is of shape (1, n_features).
 - For a multiclass problem with 'multinomial', it corresponds to outcome 1 (True), and -coef_ corresponds to outcome 0 (False).
3. **intercept_ (ndarray of shape (1,) or (n_classes,)):**
 - Intercept (bias) added to the decision function.
 - If fit_intercept is set to False, the intercept is set to zero.
 - For a binary classification problem, it is of shape (1,).
 - For a multiclass problem with 'multinomial', it corresponds to outcome 1 (True), and -intercept_ corresponds to outcome 0 (False).
4. **n_features_in_ (int):**
 - Number of features seen during fit.

- Introduced in scikit-learn version 0.24.
5. `feature_names_in_ (ndarray of shape (n_features_in_ ,))`:
 - Names of features seen during fit.
 - Defined only when the input data X has feature names that are all strings.
 - Introduced in scikit-learn version 1.0.
 6. `n_iter_ (ndarray of shape (n_classes,) or (1,))`:
 - Actual number of iterations for all classes.
 - If binary or multinomial, it returns only 1 element.
 - For the ‘liblinear’ solver, only the maximum number of iterations across all classes is given.

```
[27]: clf.classes_
```

```
[27]: array([0, 1])
```

```
[29]: clf.coef_
```

```
[29]: array([[ 1.46757679, -0.40302262, -0.01924219, -0.67173496, -0.58561719]])
```

```
[31]: clf.intercept_
```

```
[31]: array([0.16587883])
```

```
[32]: clf.n_features_in_
```

```
[32]: 5
```

```
[34]: clf.n_iter_
```

```
[34]: array([9], dtype=int32)
```

Let’s test it on our test data and see how our model performs.

```
[43]: from sklearn.metrics import accuracy_score, confusion_matrix, \
      ↪ classification_report, log_loss

y_pred = clf.predict(X_test)
print("Accuracy:", accuracy_score(y_test, y_pred))
print("Log Loss:", log_loss(y_test, y_pred))
print("\n")
print("Classification Report:")
print("\n")
print(classification_report(y_test, y_pred))
```

```
Accuracy: 0.88
```

```
Log Loss: 4.325238406694059
```

Classification Report:

	precision	recall	f1-score	support
0	0.85	0.92	0.88	97
1	0.92	0.84	0.88	103
accuracy			0.88	200
macro avg	0.88	0.88	0.88	200
weighted avg	0.88	0.88	0.88	200

Now, that we have our classification report, let see how we can read it:

Precision:

- Precision measures the accuracy of positive predictions.
- Precision for class 0 is 0.85, indicating that out of all instances predicted as class 0, 85% were correct.
- Precision for class 1 is 0.92, suggesting that out of all instances predicted as class 1, 92% were correct.

Recall (Sensitivity or True Positive Rate):

- Recall measures the ability of the model to capture all positive instances.
- Recall for class 0 is 0.92, meaning the model correctly identified 92% of all actual class 0 instances.
- Recall for class 1 is 0.84, indicating that the model captured 84% of all actual class 1 instances.

F1-Score:

- The F1-score is the harmonic mean of precision and recall, providing a balance between the two metrics.
- F1-score for class 0 is 0.88, reflecting a balance between precision and recall for class 0.
- F1-score for class 1 is also 0.88, indicating a balance for class 1.

Support: - Support represents the number of actual occurrences of each class in the specified dataset. - Support for class 0 is 97, and for class 1 is 103.

Accuracy: - Overall accuracy of the model on the entire dataset is 88%. - Accuracy is the ratio of correctly predicted instances to the total instances.

Macro Average: - The macro average is the unweighted average of precision, recall, and F1-score across all classes. - Macro-average precision, recall, and F1-score are all 0.88.

Weighted Average:

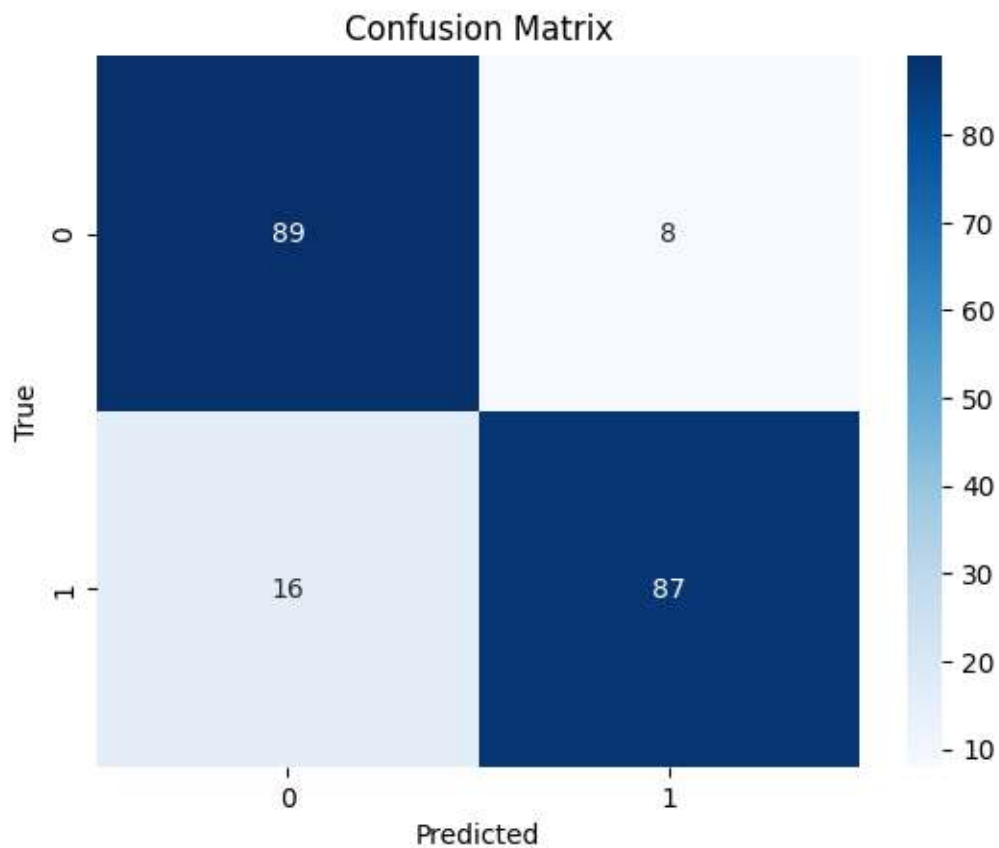
- The weighted average is the average of precision, recall, and F1-score, weighted by the support for each class.
- Weighted average precision, recall, and F1-score are all 0.88.

We will come to this again, let's first see the confusion matrix and learn how to read it.

```
[45]: cm = confusion_matrix(y_test, y_pred)
print(cm)
```

```
[[89  8]
 [16 87]]
```

```
[47]: # Let's beautify it a little bit using sns
sns.heatmap(cm, annot = True, fmt = "d", cmap = "Blues", xticklabels = np.
↳unique(y_test), yticklabels = np.unique(y_pred))
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()
```



The above confusion matrix is of the form:

	Predicted Negative	Predicted Positive
Actual Negative	True Negative	False Positive
Actual Positive	False Negative	True Positive

1. True Negative (TN): These are instances where the model correctly predicted the negative class. The actual class was negative, and the model predicted it as negative.
2. False Positive (FP): These are instances where the model incorrectly predicted the positive class. The actual class was negative, but the model predicted it as positive.
3. False Negative (FN): These are instances where the model incorrectly predicted the negative class. The actual class was positive, but the model predicted it as negative.
4. True Positive (TP): These are instances where the model correctly predicted the positive class. The actual class was positive, and the model predicted it as positive.

Practical Considerations:

1. **Confusion Matrix Representation:** Each row in a confusion matrix corresponds to an actual class, while each column represents a predicted class. The order of the rows typically follows TN (True Negative), FP (False Positive), FN (False Negative), and TP (True Positive), moving from the first row to the next.
2. **Perfect Classifier and Confusion Matrix:** A perfect classifier would only have true positives and true negatives. Therefore, its confusion matrix would only have non-zero values along its main diagonal, which goes from the top left to the bottom right.
3. **Precision Metric:** Precision measures the accuracy of positive predictions. It is calculated as the ratio of true positives (TP) to the sum of false positives (FP) and true positives (TP). Perfect precision (100%) is achieved by making a single positive prediction and ensuring its correctness. However, this extreme scenario is often impractical.
4. **Recall (Sensitivity) Metric:** Recall, also known as sensitivity or the true positive rate (TPR), indicates the ratio of positive instances correctly detected by the classifier. It is calculated as the ratio of true positives (TP) to the sum of true positives (TP) and false negatives (FN). Recall provides insights into the percentage of positive items predicted out of all items.
5. **F1 Score:** Combining precision and recall into a single metric called the F1 score is common, especially for comparing classifiers. The F1 score is the harmonic mean of precision and recall. The harmonic mean gives more weight to lower values, ensuring that a high F1 score is achieved only when both precision and recall are high. The formula is $F1 = 2 * ((\text{precision} * \text{recall}) / (\text{precision} + \text{recall}))$, which can also be expressed as $TP / (TP + ((FN + FP) / 2))$.
6. **Contextual Importance of Precision and Recall:** The importance of precision and recall depends on the context. For the spam email classifier, precision may be more important. You want to ensure that emails classified as spam are indeed spam (high precision) to avoid marking important emails as spam (low false positives). In this case, you might tolerate a lower recall (missing some spam) as long as the emails classified as spam are highly likely to be spam. On the other hand, for a medical test screening for a serious disease, recall becomes crucial. It's more acceptable to have a lower precision (some false alarms) if the test has high recall, ensuring that almost all individuals with the disease are correctly identified. In this scenario, missing a few cases (low false negatives) is more critical than having some false positives.

0.1.2 Precision/Recall trade-off - Precision/Recall Curve

High precision means fewer false positives, but it may result in more false negatives. High recall means fewer false negatives, but it may result in more false positives. The precision-recall tradeoff arises because increasing precision typically leads to a decrease in recall, and vice versa. This tradeoff becomes especially important when making decisions about the classification threshold.

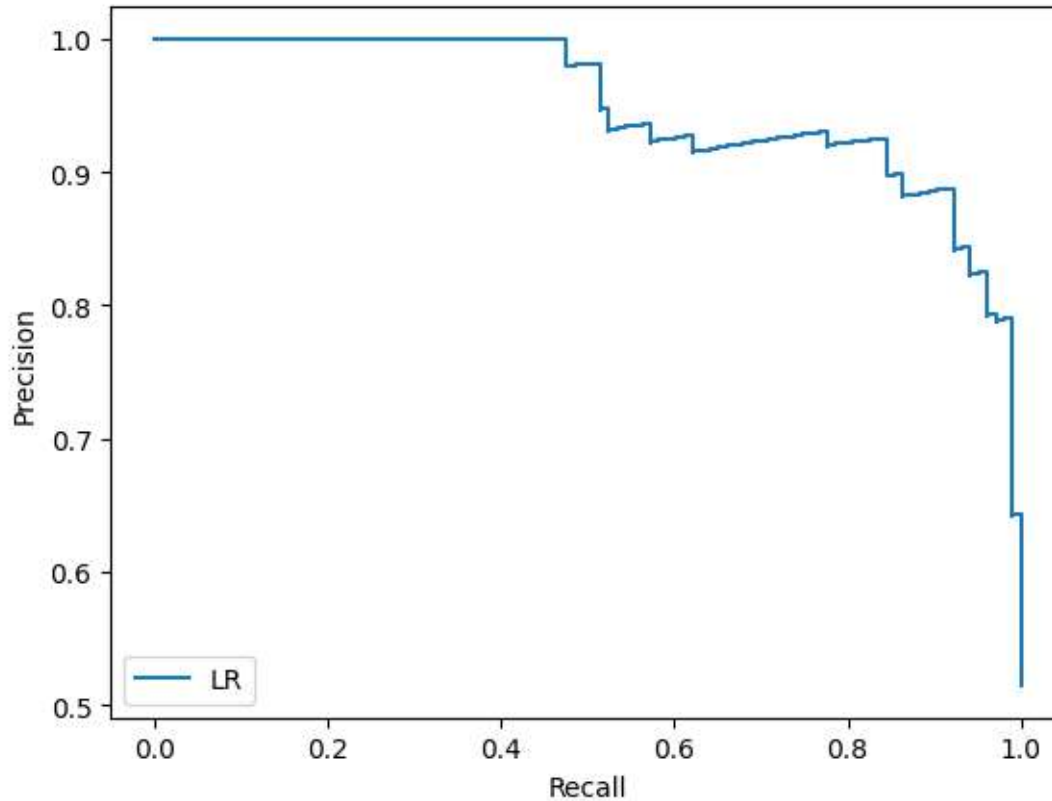
For example, if you have a classifier that predicts whether an email is spam or not, you can adjust the decision threshold to classify an email as spam. If you increase the threshold, you may catch fewer spam emails (lower recall) but have a higher confidence that the ones you catch are indeed spam (higher precision). Conversely, if you lower the threshold, you may catch more spam emails (higher recall) but with a higher chance of including some non-spam emails (lower precision).

let's plot them and see for our trained classifier:

`decision_function(X)`: Predict confidence scores for samples. The confidence score for a sample is proportional to the signed distance of that sample to the hyperplane.

```
[82]: from sklearn.metrics import precision_recall_curve, PrecisionRecallDisplay

y_scores = clf.decision_function(X_test)
precisions, recalls, thresholds = precision_recall_curve(y_test, y_scores)
disp = PrecisionRecallDisplay(precision=precisions, recall=recalls,
    ↪ estimator_name = "LR")
disp.plot()
plt.show()
```



```
[98]: y_scores[:5]
```

```
[98]: array([ 2.34942211, -0.14841633,  6.11015165,  0.36695918, -0.83502965])
```

Based on the graph above you can select the right balance between precision and recall. If precision is more important for your application (e.g., medical diagnosis), focus on the portion of the curve with high precision. If recall is more critical (e.g., identifying all instances of fraud), focus on the portion of the curve with high recall.

0.1.3 ROC-AUC

The Receiver Operating Characteristic (ROC) curve is a visual representation frequently used with binary classifiers. While it's somewhat similar to the precision/recall curve, it focuses on a different set of metrics.

True Positive Rate (Recall) vs. False Positive Rate (FPR):

- 1. True Positive Rate (Recall):**

- Recall is the proportion of actual positive instances that the classifier correctly identifies.
- It's also known as sensitivity or the true positive rate (TPR).

- 2. False Positive Rate (FPR):**

- FPR is the ratio of negative instances that are incorrectly classified as positive.

- It is calculated as 1 minus the true negative rate (TNR), which is the proportion of actual negatives correctly identified as negative.
- TNR is also referred to as specificity.

ROC Curve Construction:

- The ROC curve is constructed by plotting the True Positive Rate (Recall) against the False Positive Rate.
- Each point on the curve represents the performance of the classifier at different classification thresholds.
- The curve helps us understand how well the classifier distinguishes between the positive and negative classes, and it visually illustrates the trade-off between sensitivity and specificity.

Position on the ROC Curve:

- The top-left corner of the ROC curve is considered an ideal position.
- The higher the Recall (True Positive Rate), the more false positives the classifier tends to produce.
- A good classifier aims to be as far away from the diagonal line (random guessing) in the ROC curve, ideally towards the top-left corner.

Area Under the Curve (AUC):

- The Area Under the Curve (AUC) is a numerical measure used to quantify the performance of the classifier.
- A perfect classifier will have a ROC AUC equal to 1, indicating perfect discrimination between positive and negative instances.
- A purely random classifier will have a ROC AUC equal to 0.5.

Choosing Between PR Curve and ROC Curve:

- As a general guideline, if the positive class is rare or if you care more about false positives than false negatives, you might prefer using the Precision/Recall (PR) curve.
- Otherwise, when the class distribution is more balanced, or when you want to understand the trade-off between true positives and false positives, the ROC curve can be a useful tool.

```
[109]: from sklearn.metrics import roc_auc_score, roc_curve

# Compute Area Under the Receiver Operating Characteristic Curve (ROC AUC) from
↳ prediction scores.
y_pred_proba = clf.predict_proba(X_test)
print(y_pred_proba[:5])
```

```
[[0.08711172 0.91288828]
 [0.53703612 0.46296388]
 [0.0022153  0.9977847 ]
 [0.409276   0.590724  ]
 [0.69741737 0.30258263]]
```

Column 0: Predicted probability for the negative class (class 0) Column 1: Predicted probability for the positive class (class 1)

```
[118]: # calculate area under curve for positive class
roc_auc_score(y_test, y_pred_proba[:,1]) #[:,1] because we are interested in
↳ positive class
```

```
[118]: 0.9495545991392252
```

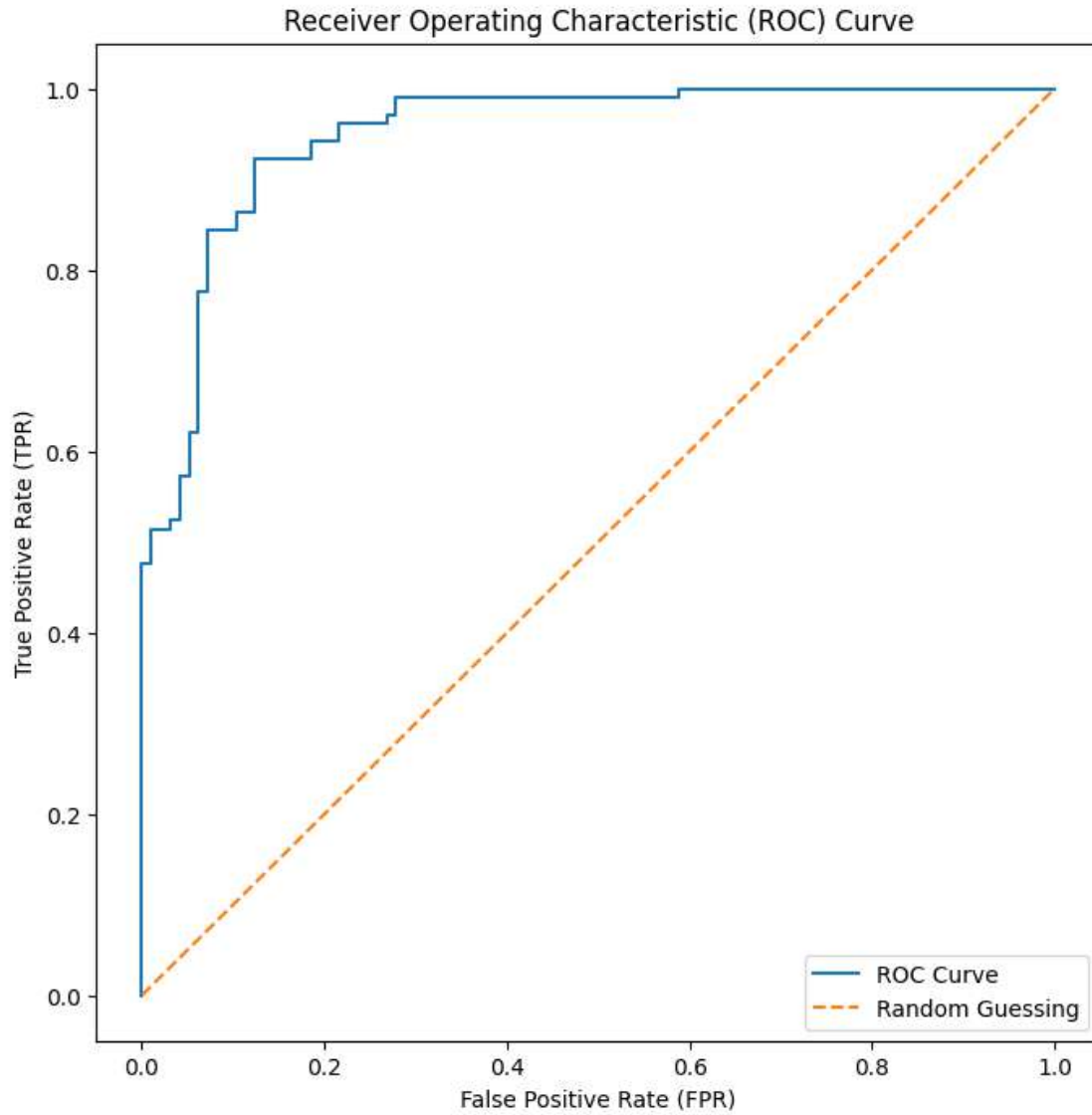
```
[103]: # another way of doing it using
roc_auc_score(y_test, y_scores)
```

```
[103]: 0.9495545991392252
```

The value lies between 0 to 1. Higher the better.

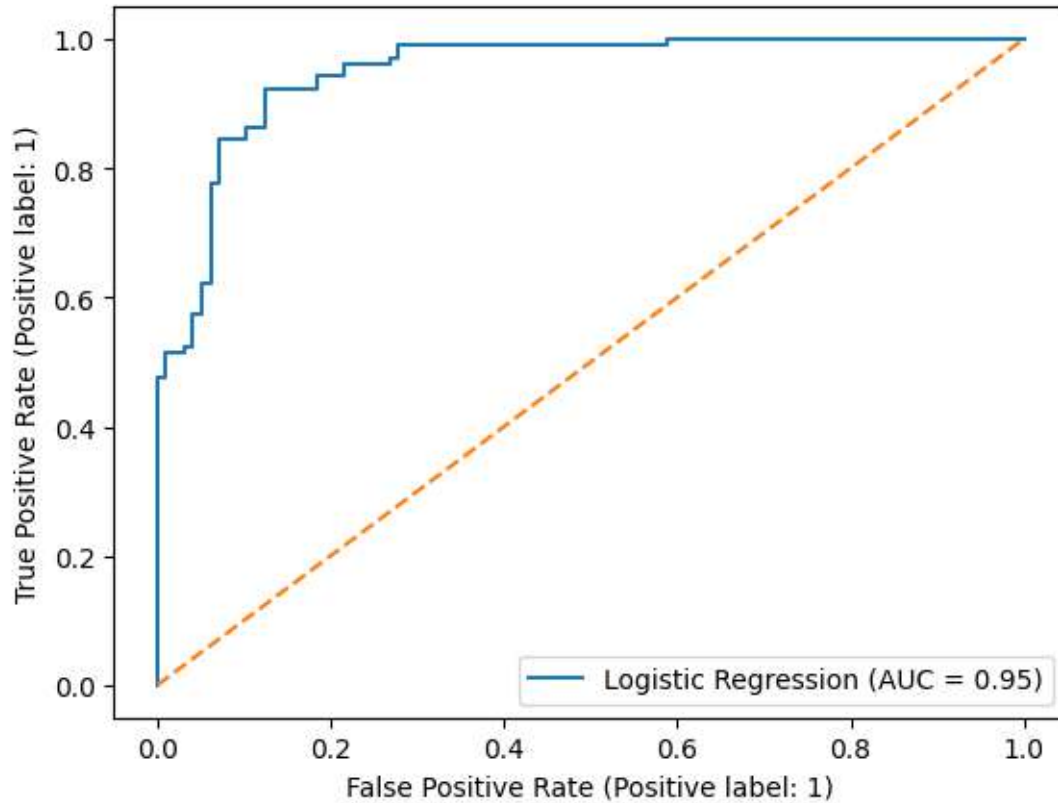
```
[119]: # Compute Receiver operating characteristic (ROC).
fpr, tpr, thresholds = roc_curve(y_test, y_pred_proba[:,1]) # you can also use
↳ y_scores here

# plot ROC curve
plt.figure(figsize=(8, 8))
plt.plot(fpr, tpr, label='ROC Curve')
plt.plot([0, 1], [0, 1], linestyle='--', label='Random Guessing')
plt.xlabel('False Positive Rate (FPR)')
plt.ylabel('True Positive Rate (TPR)')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend()
plt.show()
```

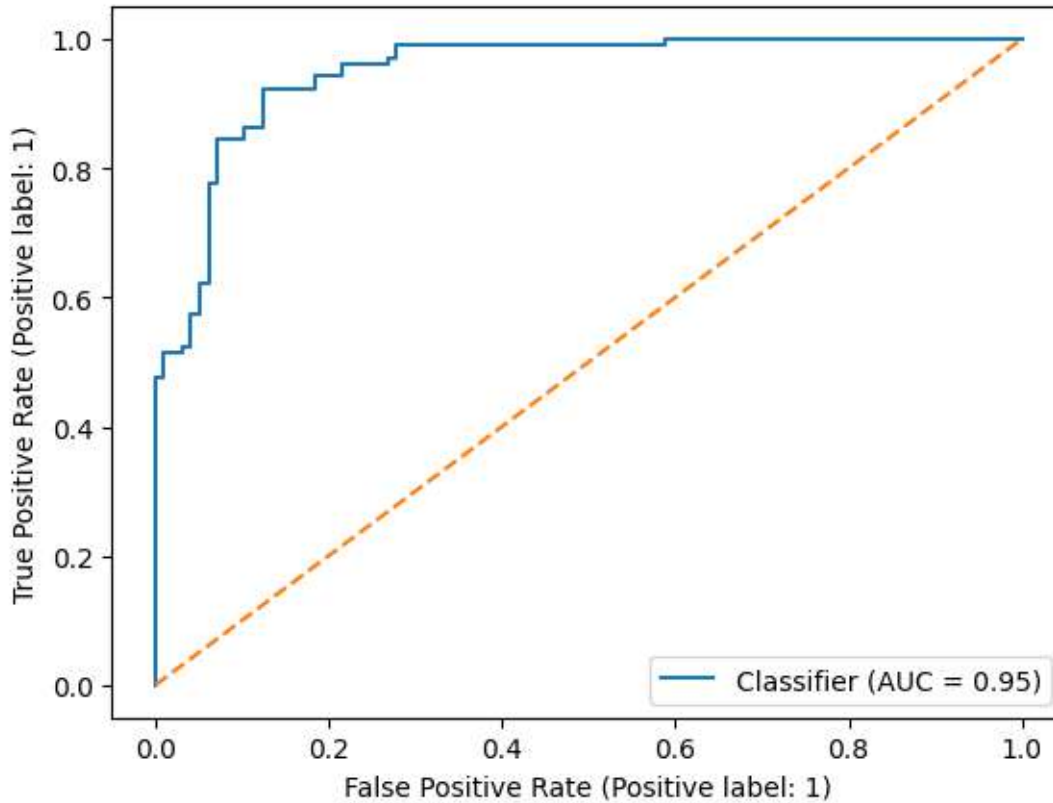


```
[129]: # there is another way to create the ROC curve
from sklearn.metrics import RocCurveDisplay

RocCurveDisplay.from_estimator(clf, X_test, y_test, name = "Logistic_
↳Regression")
plt.plot([0, 1], [0, 1], linestyle='--', label='Random Guessing');
```



```
[130]: # yet there is another way to create ROC curve from prediction directly
RocCurveDisplay.from_predictions(y_test, y_pred_proba[:, 1]) # or use y_scores
plt.plot([0, 1], [0, 1], linestyle='--', label='Random Guessing');
```



For more understanding on ROC curve read this paper here: [ROC Graphs: Notes and Practical Considerations for Researchers](#)

Sklearn also offers you [LogisticRegressionCV](#) you should check it out because it allows you to perform cross validation for multiple values of a particular parameters. `LogisticRegressionCV` implements Logistic Regression with built-in cross-validation support, to find the optimal `C` and `l1_ratio` parameters according to the scoring attribute.

0.2 LR Applied On IRIS Dataset

```
[163]: from sklearn.datasets import load_iris
from sklearn.metrics import roc_auc_score, confusion_matrix, \
↳classification_report
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

iris = load_iris()

df = pd.DataFrame(data = iris.data, columns = iris.feature_names)
df["Target"] = iris.target
```

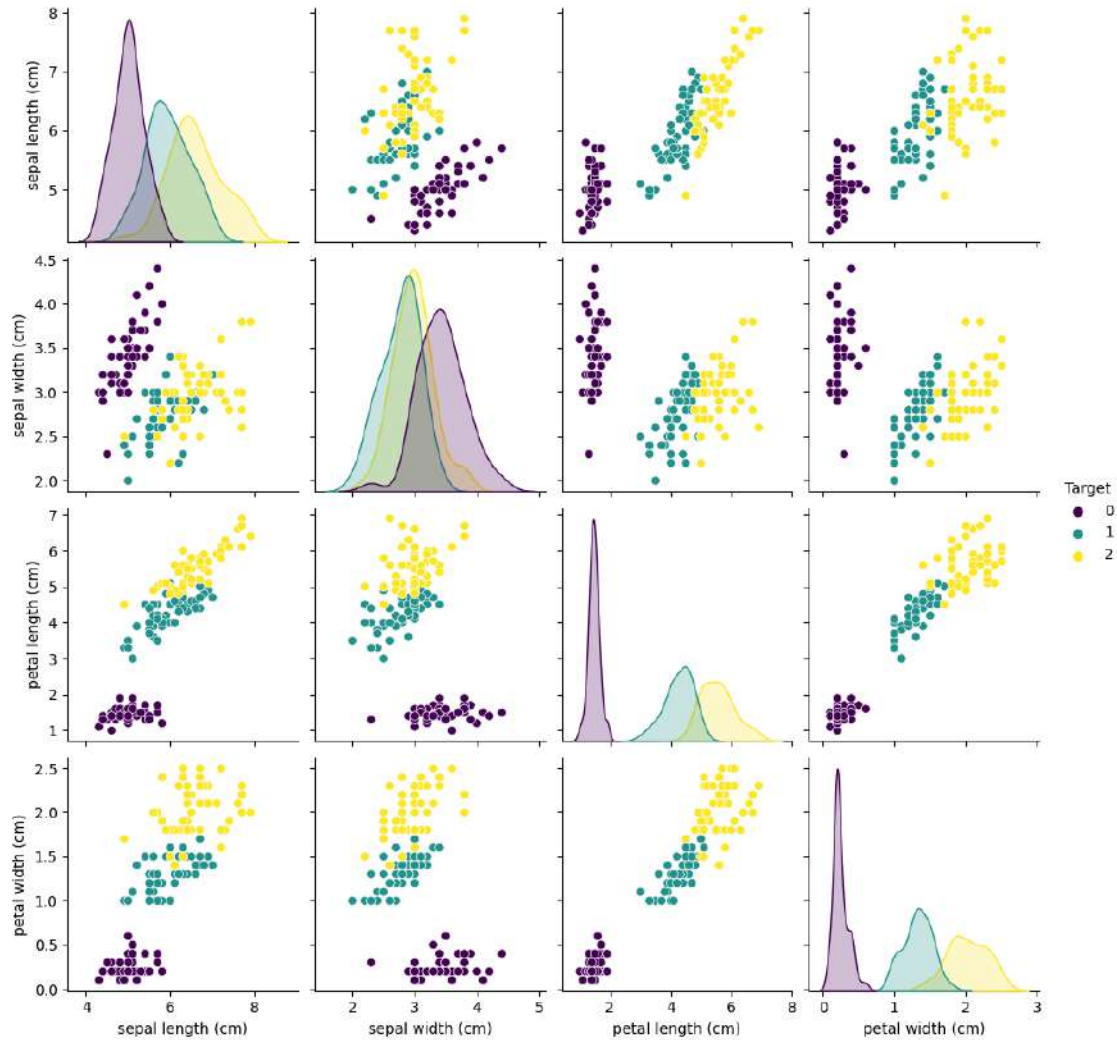
```
[151]: df.head()
```

```
[151]:
```

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	\
0	5.1	3.5	1.4	0.2	
1	4.9	3.0	1.4	0.2	
2	4.7	3.2	1.3	0.2	
3	4.6	3.1	1.5	0.2	
4	5.0	3.6	1.4	0.2	

	Target
0	0
1	0
2	0
3	0
4	0

```
[156]: sns.pairplot(df, hue = "Target", palette = "viridis");
```



```
[157]: df["Target"].value_counts()
```

```
[157]: Target
0     50
1     50
2     50
Name: count, dtype: int64
```

Our class is balanced here

```
[158]: df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 150 entries, 0 to 149
Data columns (total 5 columns):
#   Column          Non-Null Count  Dtype
#   :-----  :-----  :-----  :-----
```

```

-----
 0  sepal length (cm)  150 non-null  float64
 1  sepal width (cm)  150 non-null  float64
 2  petal length (cm) 150 non-null  float64
 3  petal width (cm)  150 non-null  float64
 4  Target            150 non-null  int64
dtypes: float64(4), int64(1)
memory usage: 6.0 KB

```

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

```

[159]:      sepal length (cm)  sepal width (cm)  petal length (cm)  \
count      150.000000      150.000000      150.000000
mean         5.843333         3.057333         3.758000
std          0.828066         0.435866         1.765298
min          4.300000         2.000000         1.000000
25%          5.100000         2.800000         1.600000
50%          5.800000         3.000000         4.350000
75%          6.400000         3.300000         5.100000
max          7.900000         4.400000         6.900000

```

```

      petal width (cm)  Target
count      150.000000  150.000000
mean         1.199333    1.000000
std          0.762238    0.819232
min          0.100000    0.000000
25%          0.300000    0.000000
50%          1.300000    1.000000
75%          1.800000    2.000000
max          2.500000    2.000000

```

```
[171]: df.corr()
```

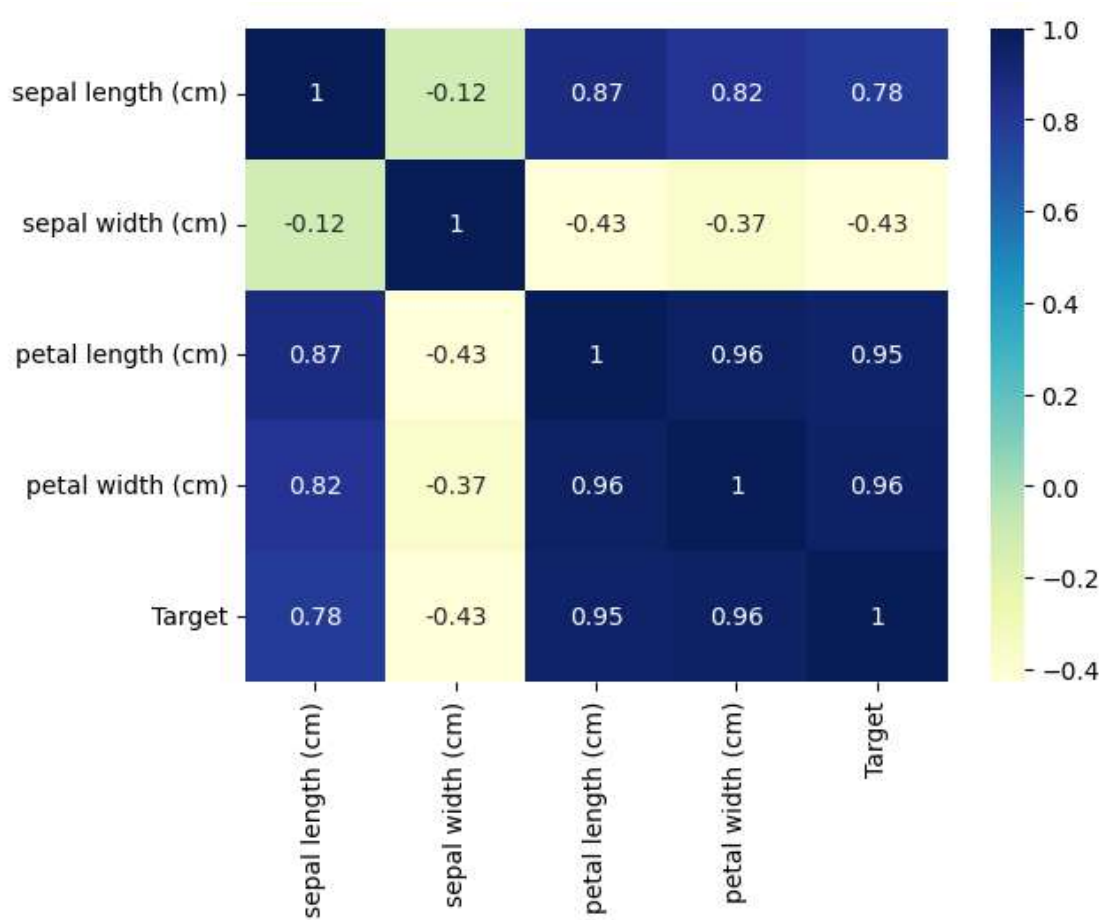
```

[171]:      sepal length (cm)  sepal width (cm)  petal length (cm)  \
sepal length (cm)      1.000000      -0.117570      0.871754
sepal width (cm)      -0.117570      1.000000      -0.428440
petal length (cm)      0.871754      -0.428440      1.000000
petal width (cm)      0.817941      -0.366126      0.962865
Target                0.782561      -0.426658      0.949035

      petal width (cm)  Target
sepal length (cm)      0.817941  0.782561
sepal width (cm)      -0.366126 -0.426658
petal length (cm)      0.962865  0.949035
petal width (cm)      1.000000  0.956547
Target                0.956547  1.000000

```

```
[170]: sns.heatmap(df.corr(), cmap="YlGnBu", annot = True);
```



```
[179]: # training test set
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(df.drop("Target", axis = 1), df["Target"], test_size = 0.2, random_state = 42)
X_train.shape, y_train.shape
```

```
[179]: ((120, 4), (120,))
```

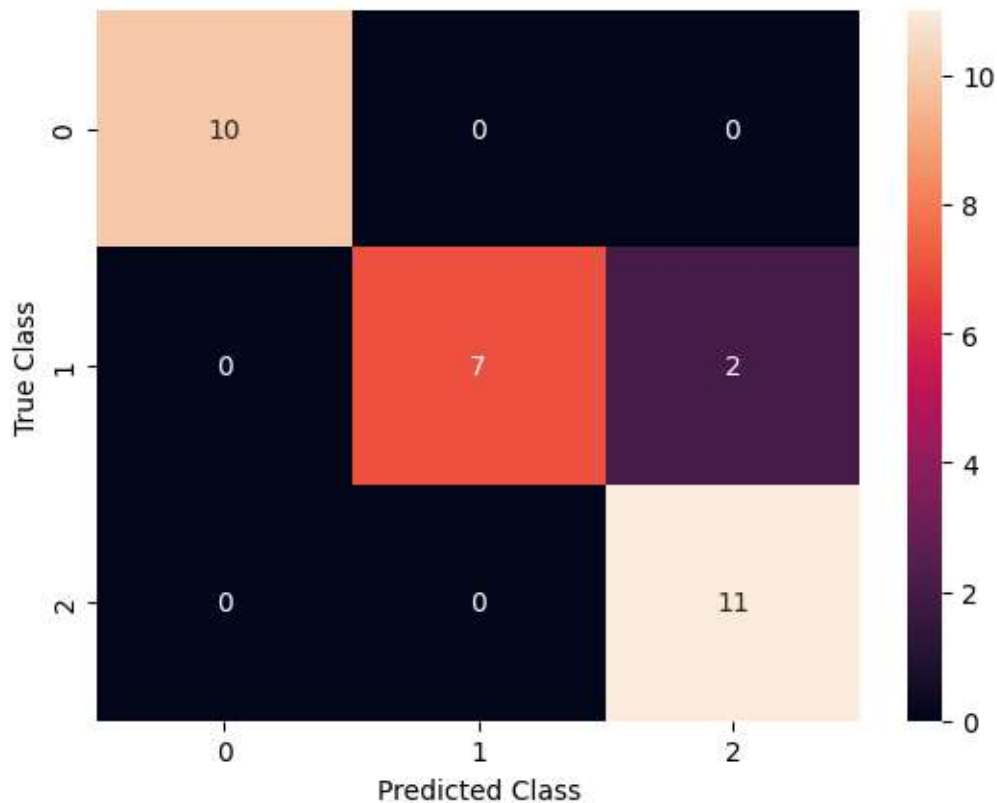
```
[192]: from sklearn.linear_model import LogisticRegression

clf = LogisticRegression(C = 0.001, max_iter = 500)
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)

print(classification_report(y_test, y_pred))
```

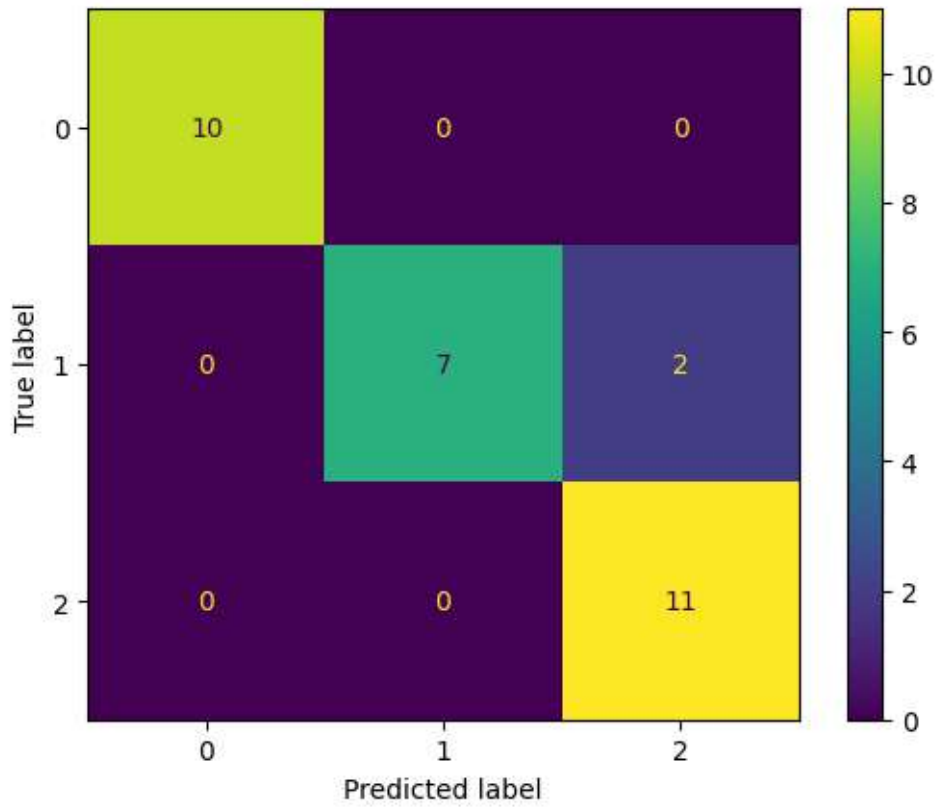
	precision	recall	f1-score	support
0	1.00	1.00	1.00	10
1	1.00	0.78	0.88	9
2	0.85	1.00	0.92	11
accuracy			0.93	30
macro avg	0.95	0.93	0.93	30
weighted avg	0.94	0.93	0.93	30

```
[197]: cm = confusion_matrix(y_test, y_pred)
sns.heatmap(cm, annot = True);
plt.xlabel("Predicted Class")
plt.ylabel("True Class");
```



Our model is confusing with class 1 for 2 datapoints

```
[201]: # another way of creating confusion matrix
from sklearn.metrics import ConfusionMatrixDisplay
ConfusionMatrixDisplay.from_predictions(y_test, y_pred);
```



ROC curves are typically used in binary classification only but you can use the sklearn approach to extend it to Multiclass as well. Read this here: [Multiclass Receiver Operating Characteristic](#)