decision-trees-in-sklearn

January 1, 2024

1 Decision Trees Tutorial - Neuraldemy by Amritesh Kumar

1.0.1 Disclaimer:

- Please note that this notebook is part of the Neuraldemy tutorial on decision trees and is made available exclusively to students who have paid for the tutorial. The contents of this notebook, including code, explanations, and any accompanying materials, are intended for educational purposes within the context of the tutorial.
- Redistribution and sharing of this notebook outside the authorized tutorial context, whether for commercial or non-commercial purposes, are strictly prohibited. Any unauthorized use, reproduction, or distribution may violate intellectual property rights and result in legal consequences.
- If you are not a registered student of the Neuraldemy tutorial on decision trees, please refrain from accessing or using this notebook. For inquiries or information on how to enroll in the tutorial, please contact Neuraldemy directly.
- Neuraldemy reserves the right to take appropriate action against individuals or entities found to be in violation of these terms.
- Thank you for your understanding and cooperation. visit https://neuraldemy.com/ for more.

1.1 Classification On Iris Dataset - Sklearn DecisionTreeClassifier On Default Mode

[2]:
```
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn import tree
import matplotlib.pyplot as plt

# Load the Iris dataset
iris = load_iris()
X = pd.DataFrame(iris.data, columns=iris.feature_names)
y = pd.Series(iris.target, name='target')

# Split the dataset into training and testing sets
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
↪random_state=42)
```

[3]:
```
X
```

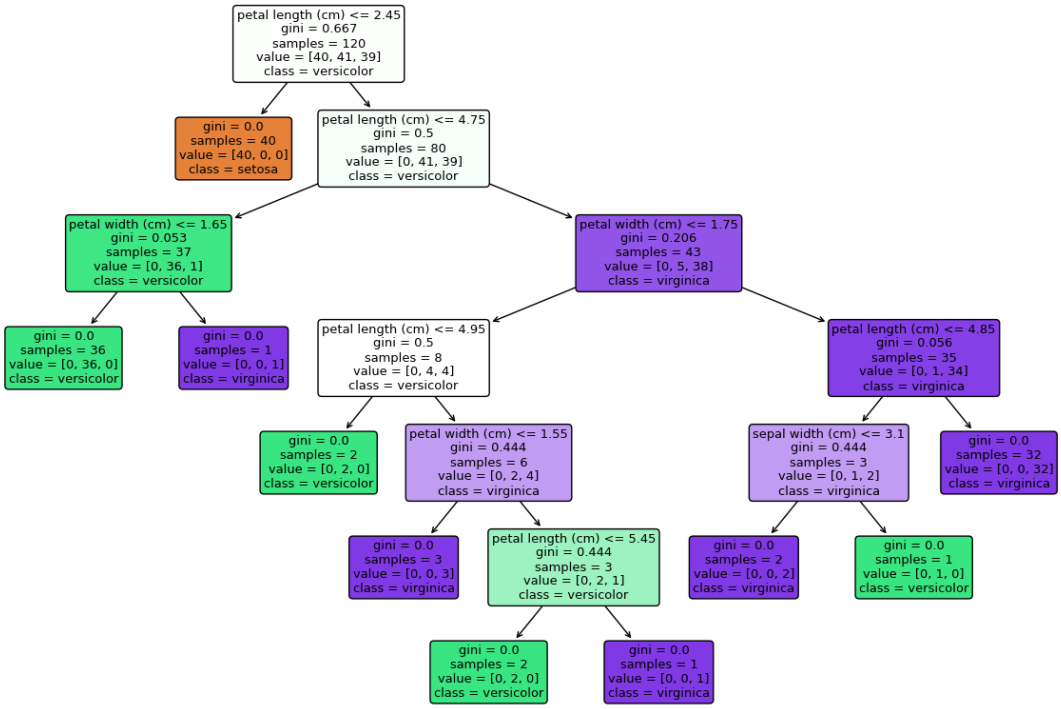
[3]:
```
     sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)
0                  5.1               3.5                1.4               0.2
1                  4.9               3.0                1.4               0.2
2                  4.7               3.2                1.3               0.2
3                  4.6               3.1                1.5               0.2
4                  5.0               3.6                1.4               0.2
..                 …                 …                  …                 …
145                6.7               3.0                5.2               2.3
146                6.3               2.5                5.0               1.9
147                6.5               3.0                5.2               2.0
148                6.2               3.4                5.4               2.3
149                5.9               3.0                5.1               1.8

[150 rows x 4 columns]
```

[4]:
```
iris.target
```

[4]:
```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 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, 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, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

In this problem we have four features in our dataset and 3 classes. We are going to use decision trees to classify the iris dataset. First using gini index and then entropy. We will be using sklearn implementation of decision tree algorithm.

[5]:
```
# Create a decision tree classifier
clf = DecisionTreeClassifier(random_state=42)

# Fit the classifier on the training data
clf.fit(X_train, y_train)

# Visualize the decision tree
plt.figure(figsize=(15, 10))
tree.plot_tree(clf, feature_names=iris.feature_names, class_names=iris.
↪target_names, filled=True, rounded=True)
plt.show()
```



1.1.1 Explaination:

The output above is a decision tree for our iris dataset based on gini impurity. You must be wondering why it is choosing petal length <= 2.45 in the root node and not any other feature? Well, the answer is: > The decision tree algorithm, when faced with a choice of which feature to split on first, evaluates all features and selects the one that provides the best split based on certain criteria. The criteria are measures of impurity, and common ones include Gini impurity, entropy, or classification error. In the case of the Iris dataset, the decision tree algorithm likely found that "petal length" provided the best initial split for distinguishing between the three classes (setosa, versicolor, virginica). This decision is based on the algorithm's assessment of the impurity reduction achieved by splitting on "petal length" compared to other features. The choice of the initial feature depends on the dataset and the relationships between features and classes. If you rerun the training process or use a different dataset, the algorithm might choose a different feature for the first split.

So, you can say the algorithm finds the features that is expected to give the best result. I will talk about the training algorithms used in decision trees later in the tutorial. Now let's look at the other values in the root node and understand their meaning:

Root Node:

- Petal length < = 2.45
- Gini = 0.667
- Sample = 120, It is 120 because you are selecting sample with petal length less than 2.45
- value = [40, 41, 39], the classes are arranged in alphabetical order, the ones with majority

number is chosen ['setosa', 'versicolor', 'virginica']. If you have equal value for each class then first one in alphabetical order will be chosen.
- class = versicolor, we have this chosen class because it is majority

first left-child internal node: - Gini = 0, when the gini reaches 0, we stop the calculation because it becomes final leaf node - Sample = 40 - value = [40, 0, 0] since majority class is 40 which is setosa so we have setosa - class = setosa

first right-child internal node Once the algorithm has performed the inital calculation, it looks for next best features that will give the best result and shows us the tree for that node and it goes on until we reach gini = 0 for each sub-trees, or until they become leaf nodes.

This is what happens when you run the sklearn decision tree on default mode.

The algorithm keeps on going until it classifies each class.

1.2 Hyperparameters in Sklearn DecisionTreeClassifier

There are various hyperparameters that you an tune and modify according your need and problem. You don't need to learn each and every one of them but it's better to go through them so that you can know what they do. It's not always a good idea to let the decision tree to go on forever. With practice you will be able to use them:

- criterion='gini': The criterion used to measure the quality of a split. 'gini' represents the Gini impurity, a measure of node impurity. Other options include 'entropy', which measures information gain.
- splitter='best':The strategy used to choose the split at each node. 'best' selects the best split based on the chosen criterion. Other options include 'random', which selects a random split.
- max_depth=None:The maximum depth of the decision tree. If None, nodes are expanded until they contain fewer than min_samples_split samples per split.
- min_samples_split=2:The minimum number of samples required to split an internal node. If a node has fewer samples than min_samples_split, it will not be split.
- min_samples_leaf=1:The minimum number of samples required to be in a leaf node. If a leaf node would have fewer samples than min_samples_leaf, it will not be created, and the node's samples will be distributed among its ancestors.
- min_weight_fraction_leaf=0.0:The minimum weighted fraction of the total sum of weights (of all input samples) required to be at a leaf node. Useful when working with weighted samples.
- max_features=None:The number of features to consider when looking for the best split. If None, all features will be considered.
- random_state=None:Controls the randomness of the estimator. If an integer, it sets the seed for reproducibility. If None, the randomness is based on the current seed of the random number generator.
- max_leaf_nodes=None:The maximum number of leaf nodes in the decision tree. If None, there is no maximum limit on the number of leaf nodes.
- min_impurity_decrease=0.0:A node will be split if the split induces a decrease in impurity greater than or equal to this value. If 0.0, the split will only occur if it improves impurity.
- class_weight=None:Weights associated with classes. If not provided, all classes are assumed to have equal weight.
- ccp_alpha=0.0:Complexity parameter used for Minimal Cost-Complexity Pruning (MCCP).

A non-negative value that sets the threshold for pruning. Higher values increase pruning, leading to simpler trees.

[6]:
```
# let's evaluate the model on test set
accuracy = clf.score(X_test, y_test)
print("Accuracy on the test set:", accuracy)
```

```
Accuracy on the test set: 1.0
```

When the accuracy is coming out to be 1 chances are higher that our decision tree is overfitting. We will talk about overfitting shortly and how to over come this.

[7]:
```
# how deep is our decision tree
clf.get_depth()
```

[7]:
```
6
```

[8]:
```
clf.get_params()
```

[8]:
```
{'ccp_alpha': 0.0,
 'class_weight': None,
 'criterion': 'gini',
 'max_depth': None,
 'max_features': None,
 'max_leaf_nodes': None,
 'min_impurity_decrease': 0.0,
 'min_samples_leaf': 1,
 'min_samples_split': 2,
 'min_weight_fraction_leaf': 0.0,
 'random_state': 42,
 'splitter': 'best'}
```

[9]:
```
# Predict class
clf.predict(X_test)
```

[9]:
```
array([1, 0, 2, 1, 1, 0, 1, 2, 1, 1, 2, 0, 0, 0, 0, 1, 2, 1, 1, 2, 0, 2,
       0, 2, 2, 2, 2, 2, 0, 0])
```

[10]:
```
clf.predict_log_proba(X_test)
```

```
/opt/conda/lib/python3.10/site-packages/sklearn/tree/_classes.py:965:
RuntimeWarning: divide by zero encountered in log
  return np.log(proba)
```

[10]:
```
array([[-inf,   0., -inf],
       [  0., -inf, -inf],
       [-inf, -inf,   0.],
       [-inf,   0., -inf],
       [-inf,   0., -inf],
```

```
[  0., -inf, -inf],
[-inf,   0., -inf],
[-inf, -inf,   0.],
[-inf,   0., -inf],
[-inf,   0., -inf],
[-inf, -inf,   0.],
[  0., -inf, -inf],
[  0., -inf, -inf],
[  0., -inf, -inf],
[  0., -inf, -inf],
[-inf,   0., -inf],
[-inf, -inf,   0.],
[-inf,   0., -inf],
[-inf,   0., -inf],
[-inf, -inf,   0.],
[  0., -inf, -inf],
[-inf, -inf,   0.],
[  0., -inf, -inf],
[-inf, -inf,   0.],
[-inf, -inf,   0.],
[-inf, -inf,   0.],
[-inf, -inf,   0.],
[-inf, -inf,   0.],
[  0., -inf, -inf],
[  0., -inf, -inf]])
```

If you are getting the above error this means that some of the class probabilities are zero or you can say when you are trying to compute the logarithm of zero or a negative number.

[11]:
```
clf.predict_proba(X_test)
```

[11]:
```
array([[0., 1., 0.],
       [1., 0., 0.],
       [0., 0., 1.],
       [0., 1., 0.],
       [0., 1., 0.],
       [1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.],
       [0., 1., 0.],
       [0., 1., 0.],
       [0., 0., 1.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [0., 1., 0.],
```

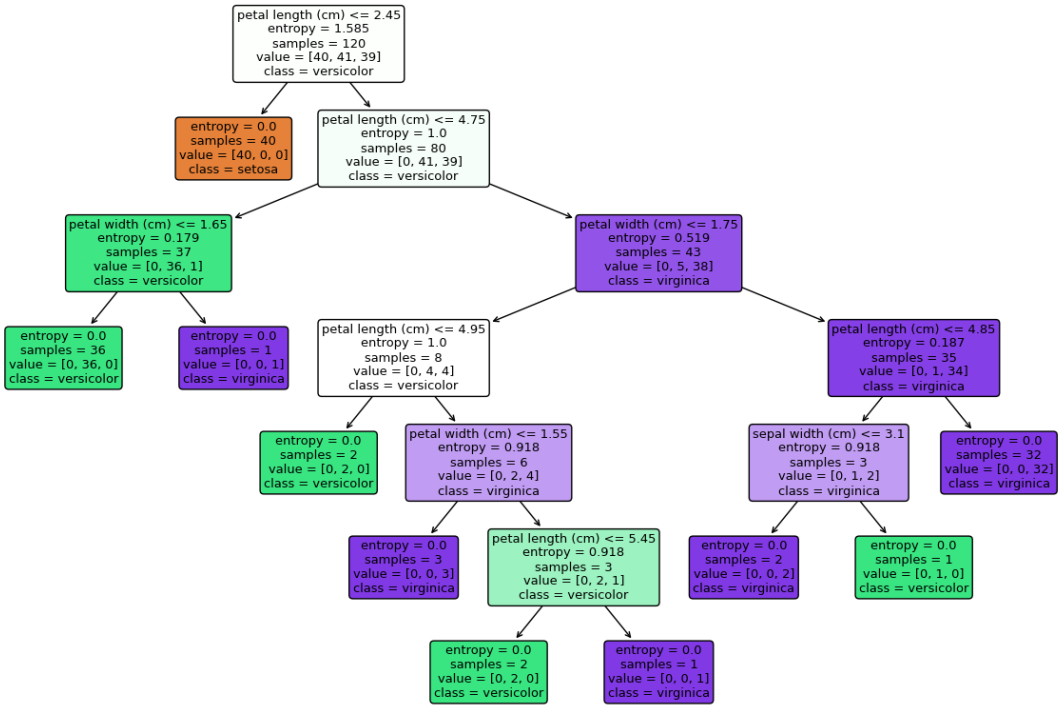
```
       [0., 0., 1.],
       [0., 1., 0.],
       [0., 1., 0.],
       [0., 0., 1.],
       [1., 0., 0.],
       [0., 0., 1.],
       [1., 0., 0.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [1., 0., 0.],
       [1., 0., 0.]])
```

[12]:
```
clf.score(X_test, y_test)
```

[12]:
```
1.0
```

Decision Trees Based on entropy

[15]:
```
# Create a decision tree classifier
clf2 = DecisionTreeClassifier(criterion = "entropy", random_state=42)

# Fit the classifier on the training data
clf2.fit(X_train, y_train)

# Visualize the decision tree
plt.figure(figsize=(15, 10))
tree.plot_tree(clf2, feature_names=iris.feature_names, class_names=iris.
 ↪target_names, filled=True, rounded=True)
plt.show()
```



[16]:
```
clf2.score(X_test, y_test)
```

[16]: 1.0

You can use any criterion. Gini impurity tends to isolate the most frequent class in its own branch of the tree, while entropy tends to produce slightly more balanced trees. Difference:

Gini Impurity:

- Computationally faster than entropy.
- Tends to be a bit more robust to outliers.
- Works well when the classes are well-separated.

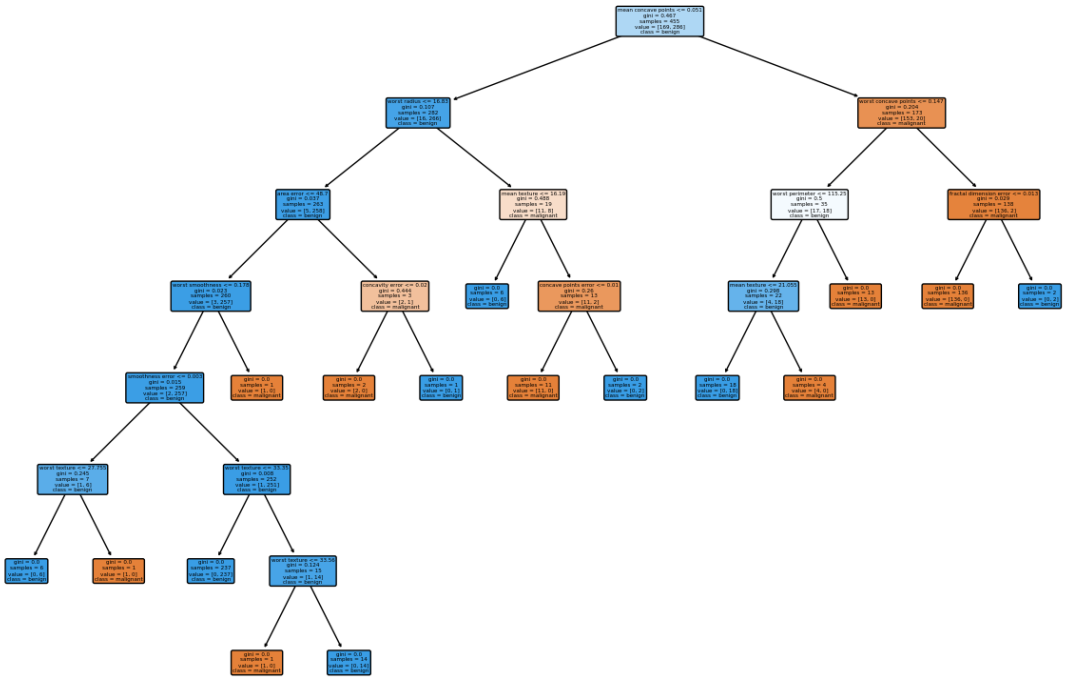
Entropy:

- May be more sensitive to outliers due to its logarithmic nature.
- Tends to create more balanced trees.
- Can be more effective when the classes are not well-separated.

As noticed above our model seems to be overfitting so we need to regularize it a little bit as explained in tutorial. To do that you can restrict some of the parameters and see what happens.

1.3 Decision Trees Classification On Breast Cancer Dataset

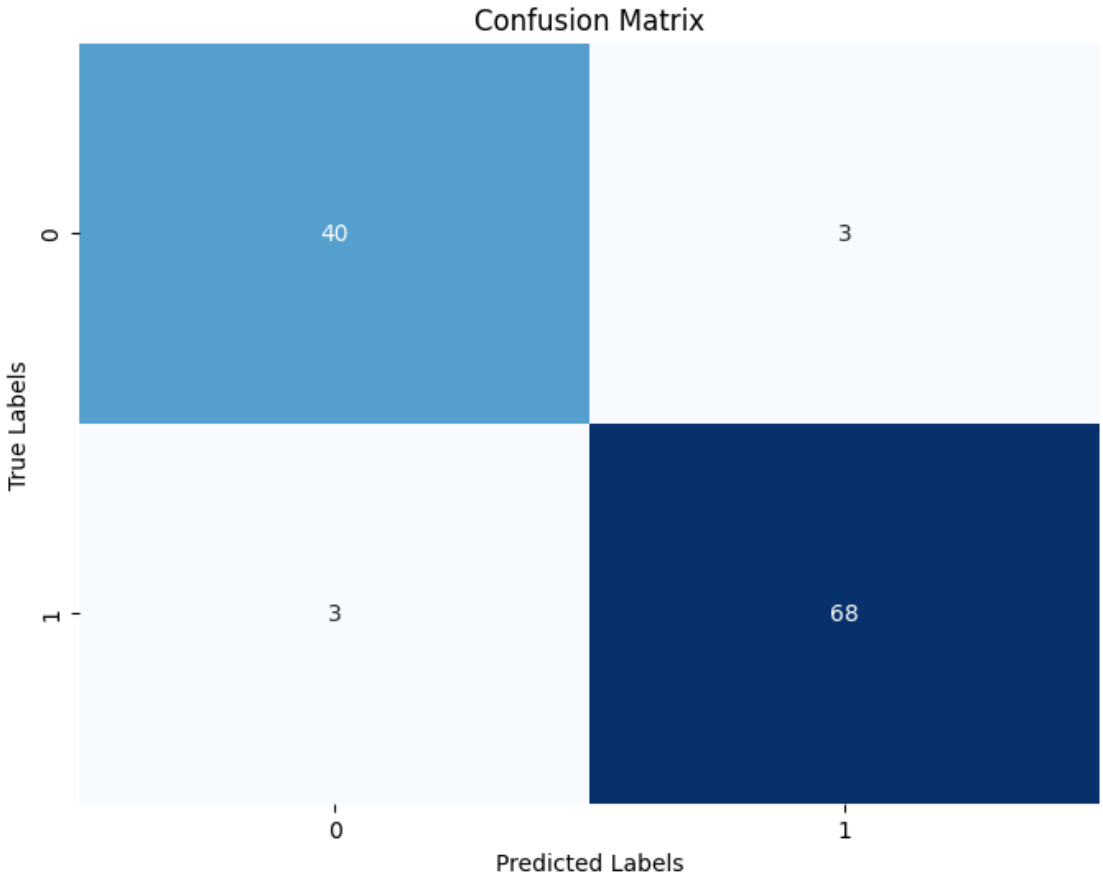
[20]:
```
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, classification_report

# Load the Breast Cancer dataset
cancer = load_breast_cancer()
X = pd.DataFrame(cancer.data, columns=cancer.feature_names)
y = pd.Series(cancer.target, name='target')

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)

# Create a Decision Tree model
dt_model = DecisionTreeClassifier(random_state=42)

# Train the model on the training data
dt_model.fit(X_train, y_train)

# Visualize the decision tree
plt.figure(figsize=(15, 10))
tree.plot_tree(dt_model, feature_names=cancer.feature_names, class_names=cancer.
 ↪target_names, filled=True, rounded=True)
plt.show()
```



[21]:
```
# Make predictions on the testing data
y_pred = dt_model.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
classification_rep = classification_report(y_test, y_pred)

print(f"Accuracy: {accuracy:.2f}")
print("Classification Report:")
print(classification_rep)
```

```
Accuracy: 0.95
Classification Report:
              precision    recall  f1-score   support

           0       0.93      0.93      0.93        43
           1       0.96      0.96      0.96        71

    accuracy                           0.95       114
   macro avg       0.94      0.94      0.94       114
weighted avg       0.95      0.95      0.95       114
```

[23]:
```
from sklearn.metrics import confusion_matrix
import seaborn as sns
# Create a confusion matrix
cm = confusion_matrix(y_test, y_pred)

# Plot the confusion matrix using seaborn
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", cbar=False)
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.title('Confusion Matrix')
plt.show()
```



[27]:
```
dt_model.get_depth()
```

[27]: 7

Let's reguralize the model a little bit

[29]:
```
# Create a Decision Tree model
dt2_model = DecisionTreeClassifier(min_samples_leaf = 4, max_depth = 4,␣
 ↪random_state=42)

# Train the model on the training data
dt2_model.fit(X_train, y_train)
```

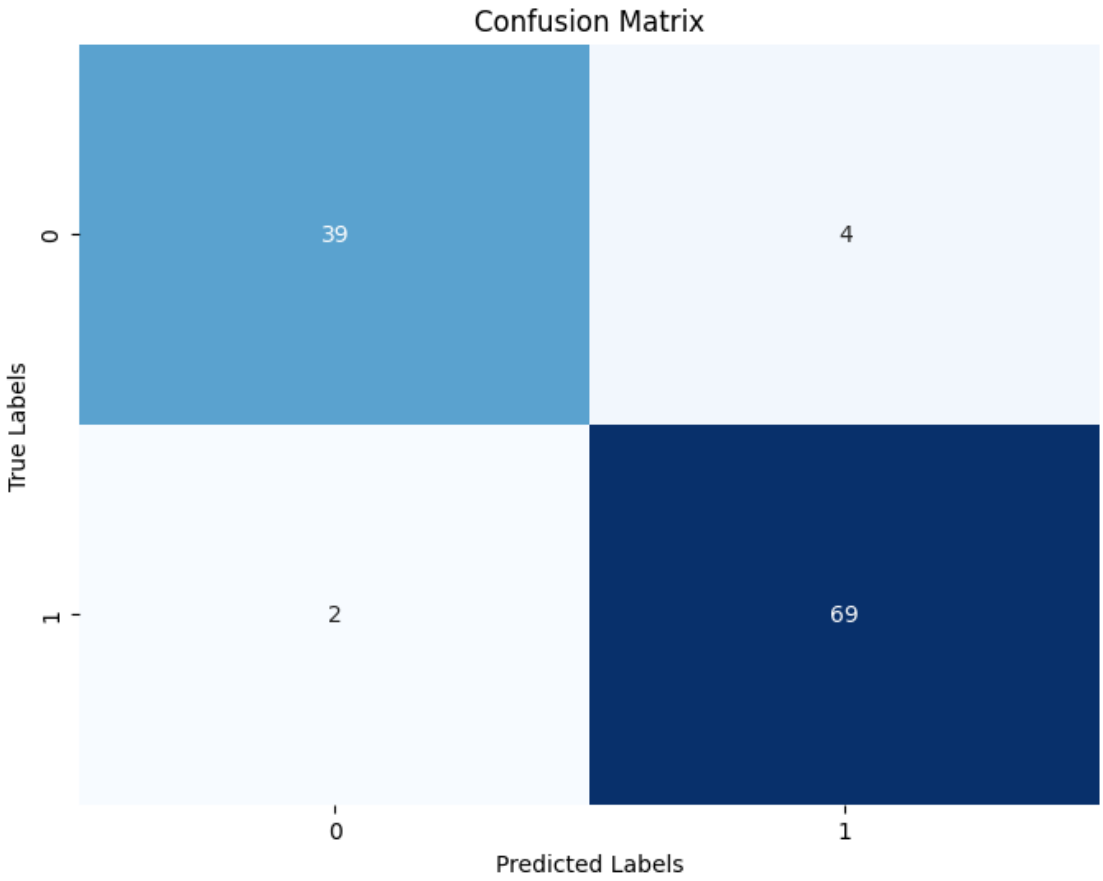
[29]:
```
DecisionTreeClassifier(max_depth=4, min_samples_leaf=4, random_state=42)
```

[30]:
```
y2_pred = dt2_model.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y2_pred)
```

[31]:
```
accuracy
```

[31]:
```
0.9473684210526315
```

[32]:
```
cm2 = confusion_matrix(y_test, y2_pred)

# Plot the confusion matrix using seaborn
plt.figure(figsize=(8, 6))
sns.heatmap(cm2, annot=True, fmt="d", cmap="Blues", cbar=False)
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.title('Confusion Matrix')
plt.show()
```



This is how we can regularize our model by limiting certain parameters. We will see more examples while working with real world datasets. The second example is working as intended. If you are not familiar with the confusion matrix and other evaluation metrics then please check our tutorial on them.

1.4 What If Model Is Overfitting Badly?

Whenever you are getting 100% score on testing dataset chances are higher that it is overfitting if the model parameters are set to default values. Other than tuning depth and leafs, we can also prune some nodes to improve the performance of our model further.

1.5 Minimal Cost-Complexity Pruning

Minimal Cost-Complexity Pruning is a technique used to prune decision trees in order to optimize their complexity while considering the associated costs. This approach aims to find a trade-off between model simplicity and accuracy.

In DecisionTreeClassifier, this pruning technique is parameterized by the cost complexity parameter, ccp_alpha. Greater values of ccp_alpha increase the number of nodes pruned. Here we only show the effect of ccp_alpha on regularizing the trees and how to choose a ccp_alpha based on validation scores.

Minimal cost complexity pruning recursively finds the node with the "weakest link". The weakest link is characterized by an effective alpha, where the nodes with the smallest effective alpha are pruned first. To get an idea of what values of ccp_alpha could be appropriate, scikit-learn provides DecisionTreeClassifier.cost_complexity_pruning_path that returns the effective alphas and the corresponding total leaf impurities at each step of the pruning process. As alpha increases, more of the tree is pruned, which increases the total impurity of its leaves.

To regularize your model using ccp_alpha parameter, simply follow the following steps:

1. Create a decision tree (clf) using Sklearn class
2. find path = clf.cost_complexity_pruning_path(X_train, y_train) and calculate ccp_alphas, impurities = path.ccp_alphas, path.impurities
3. calculate clfs = [ ] for different values of alphas
4. Plot the train and test score for different values of alpha and select the best alpha where your model is generalizing well on training and test data.

[35]:
```
import matplotlib.pyplot as plt
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
```

[36]:
```
X, y = load_breast_cancer(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)

clf = DecisionTreeClassifier(random_state=0)
path = clf.cost_complexity_pruning_path(X_train, y_train)
ccp_alphas, impurities = path.ccp_alphas, path.impurities
```

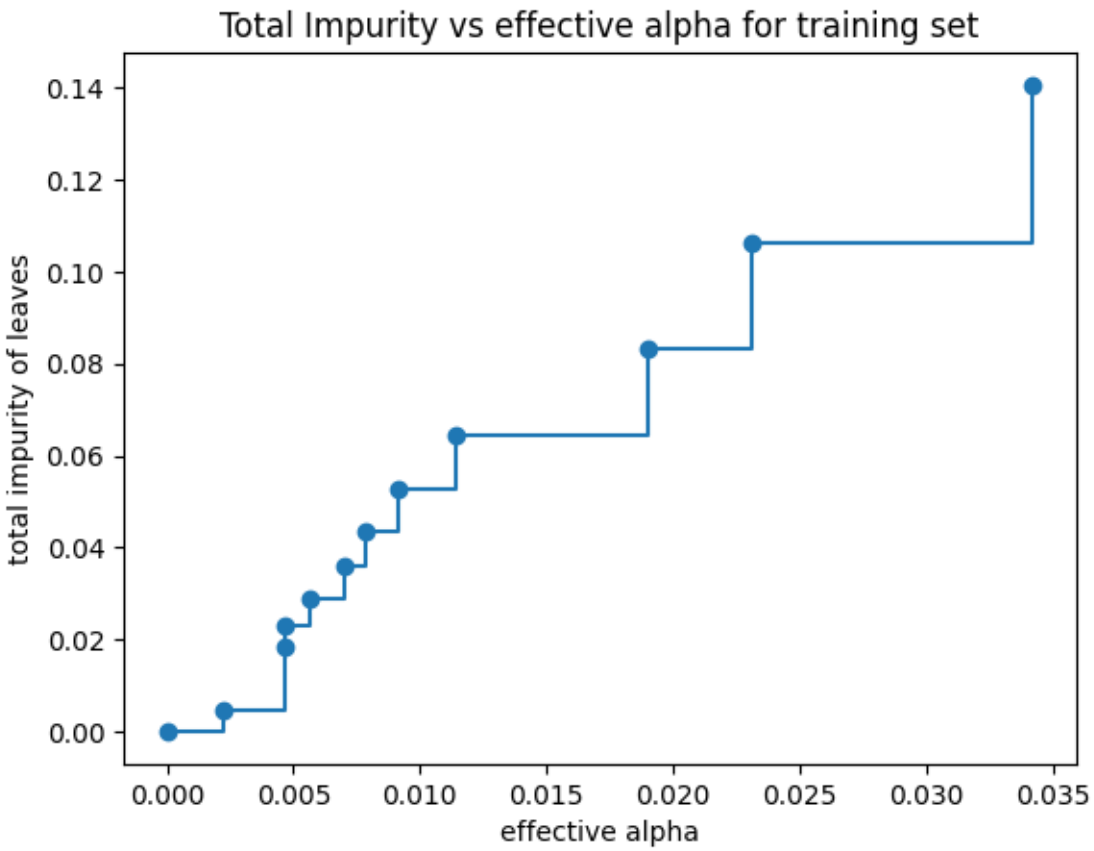
[38]:
```
ccp_alphas, impurities
```

[38]:
```
(array([0.        , 0.00226647, 0.00464743, 0.0046598 , 0.0056338 ,
        0.00704225, 0.00784194, 0.00911402, 0.01144366, 0.018988  ,
        0.02314163, 0.03422475, 0.32729844]),
 array([0.        , 0.00453294, 0.01847522, 0.02313502, 0.02876883,
        0.03581108, 0.04365302, 0.05276704, 0.0642107 , 0.0831987 ,
        0.10634033, 0.14056508, 0.46786352]))
```

In the following plot, the maximum effective alpha value is removed, because it is the trivial tree with only one node. We will only focuse on intermedidate nodes as they are more meangiful.

[41]:
```
fig, ax = plt.subplots()
ax.plot(ccp_alphas[:-1], impurities[:-1], marker="o", drawstyle="steps-post")
ax.set_xlabel("effective alpha")
ax.set_ylabel("total impurity of leaves")
ax.set_title("Total Impurity vs effective alpha for training set")
```

[41]:
```
Text(0.5, 1.0, 'Total Impurity vs effective alpha for training set')
```



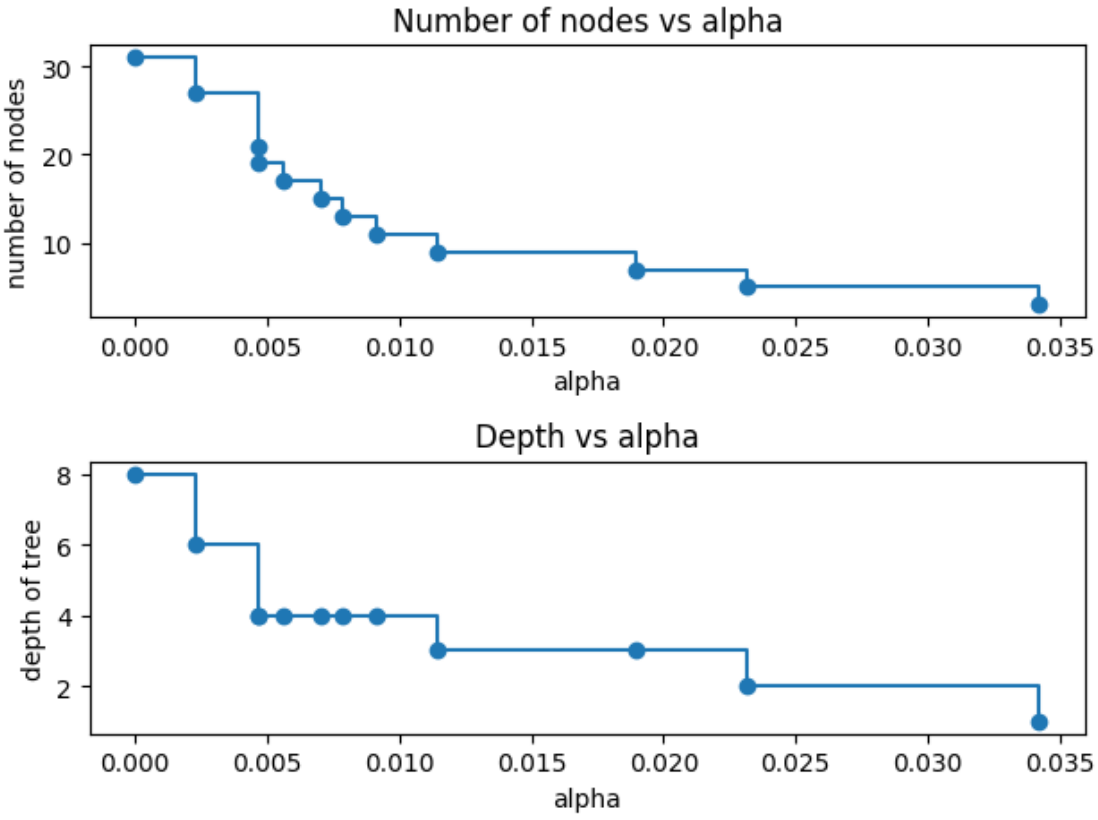
We will now train our decision trees for different values of ccp_alpha

[42]:
```
clfs = []

for ccp_alpha in ccp_alphas:
    clf = DecisionTreeClassifier(random_state=0, ccp_alpha=ccp_alpha)
    clf.fit(X_train, y_train)
    clfs.append(clf)

print(
    "Number of nodes in the last tree is: {} with ccp_alpha: {}".format(
        clfs[-1].tree_.node_count, ccp_alphas[-1]
    )
)
```

```
Number of nodes in the last tree is: 1 with ccp_alpha: 0.3272984419327777
```

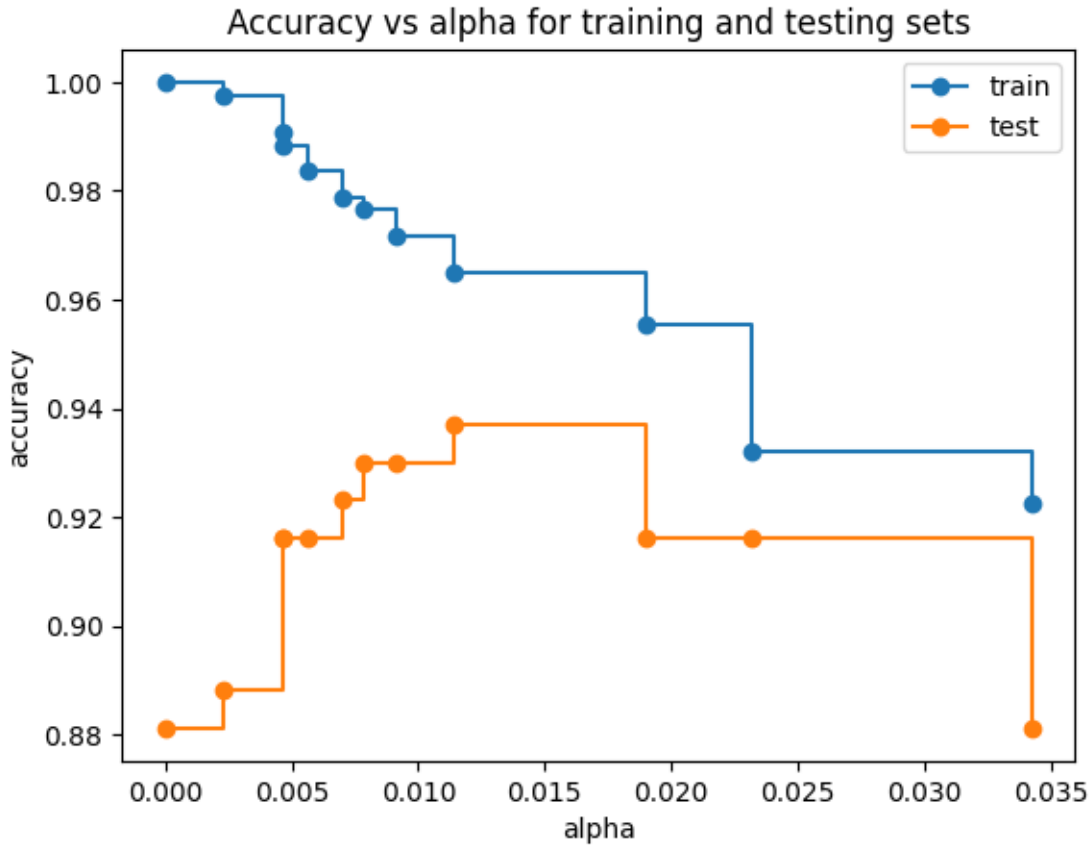
For the remainder of this example, we remove the last element in clfs and ccp_alphas, because it is the trivial tree with only one node. Here we show that the number of nodes and tree depth decreases as alpha increases.

[43]:
```
clfs = clfs[:-1]
ccp_alphas = ccp_alphas[:-1]

node_counts = [clf.tree_.node_count for clf in clfs]
depth = [clf.tree_.max_depth for clf in clfs]

fig, ax = plt.subplots(2, 1)
ax[0].plot(ccp_alphas, node_counts, marker="o", drawstyle="steps-post")
ax[0].set_xlabel("alpha")
ax[0].set_ylabel("number of nodes")
ax[0].set_title("Number of nodes vs alpha")
ax[1].plot(ccp_alphas, depth, marker="o", drawstyle="steps-post")
ax[1].set_xlabel("alpha")
ax[1].set_ylabel("depth of tree")
ax[1].set_title("Depth vs alpha")
fig.tight_layout()
```



When ccp_alpha is set to zero and keeping the other default parameters of DecisionTreeClassifier, the tree overfits, leading to a 100% training accuracy and 88% testing accuracy. As alpha increases, more of the tree is pruned, thus creating a decision tree that generalizes better. In this example, setting ccp_alpha=0.015 maximizes the testing accuracy.

[44]:
```
train_scores = [clf.score(X_train, y_train) for clf in clfs]
test_scores = [clf.score(X_test, y_test) for clf in clfs]

fig, ax = plt.subplots()
ax.set_xlabel("alpha")
ax.set_ylabel("accuracy")
ax.set_title("Accuracy vs alpha for training and testing sets")
ax.plot(ccp_alphas, train_scores, marker="o", label="train",␣
 ↪drawstyle="steps-post")
ax.plot(ccp_alphas, test_scores, marker="o", label="test",␣
 ↪drawstyle="steps-post")
ax.legend()
plt.show()
```



As you can see that at 0.015 value of alpha we are getting better result. This is how you can choose the value of alpha and regularize the decision trees.

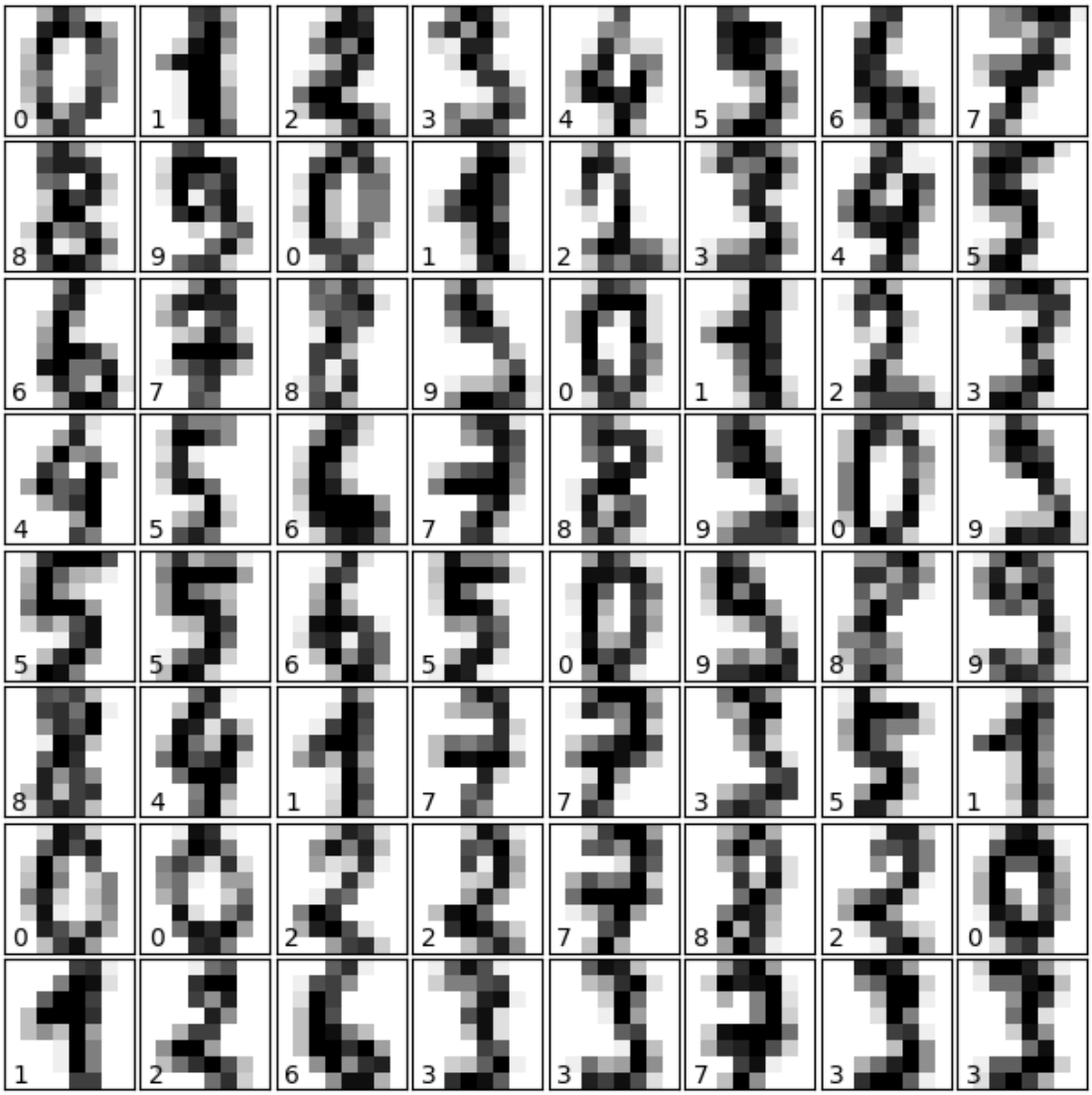
If you further wish to reguralize model on other hyperparameters try using Grid search or random search for different values and see how your model performs. Experiment using these methods to reach a final list of hyperparameters.

1.6 Decision Tree For Digits Classification

[20]:
```
from sklearn.datasets import load_digits
digits = load_digits()
digits.keys()
```

[20]:
```
dict_keys(['data', 'target', 'frame', 'feature_names', 'target_names', 'images',
'DESCR'])
```

[21]:
```
fig = plt.figure(figsize=(6, 6))
fig.subplots_adjust(left=0, right=1, bottom=0, top=1, hspace=0.05, wspace=0.05)

# plot the digits
for i in range(64):
    ax = fig.add_subplot(8, 8, i + 1, xticks=[], yticks=[])
    ax.imshow(digits.images[i], cmap=plt.cm.binary, interpolation='nearest')

    # label the image with the target value
    ax.text(0, 7, str(digits.target[i]))
```



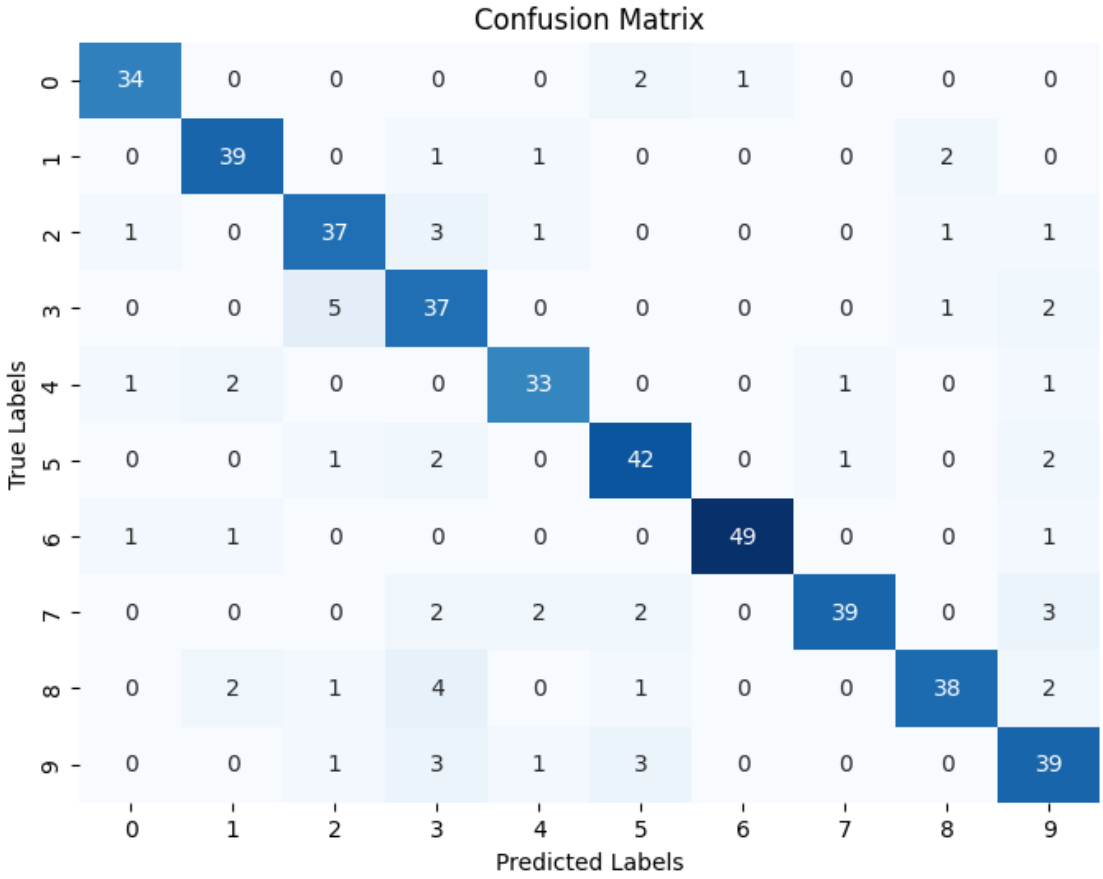
[25]:
```
from sklearn.tree import DecisionTreeClassifier
X_train, X_test, y_train, y_test = train_test_split(digits.data, digits.target,
                                                    random_state=0)
model = DecisionTreeClassifier()
model.fit(X_train, y_train)
y_pred = model.predict(X_test)
```

[26]:
```
model.score(X_test, y_test)
```

[26]: 0.86

[28]:
```
from sklearn.metrics import confusion_matrix
import seaborn as sns
```

```
cm3 = confusion_matrix(y_test, y_pred)

# Plot the confusion matrix using seaborn
plt.figure(figsize=(8, 6))
sns.heatmap(cm3, annot=True, fmt="d", cmap="Blues", cbar=False)
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.title('Confusion Matrix')
plt.show()
```



1.7 Decision Tree For Regression Datasets - California Housing Price

Just like classification, we can also use decision trees for regression tasks as discussed in the theory. The implementation is same as classification but you should check out the hyperparameters of regression task in Sklearn on your own. https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeRegressor.html

[14]:
```
import numpy as np
import matplotlib.pyplot as plt
```

```
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeRegressor
from sklearn.metrics import mean_squared_error, r2_score

# Load the California Housing Prices dataset
california_housing = fetch_california_housing()
X = california_housing.data
y = california_housing.target

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)

# Initialize the Decision Tree Regressor
regressor = DecisionTreeRegressor(random_state=42)

# Fit the model to the training data
regressor.fit(X_train, y_train)

# Make predictions on the test data
y_pred = regressor.predict(X_test)
```

[15]:
```
# Evaluate the model
mse = mean_squared_error(y_test, y_pred)
r_squared = r2_score(y_test, y_pred)

# Print evaluation metrics
print("Mean Squared Error:", mse)
print("R-squared Score:", r_squared)
```

```
Mean Squared Error: 0.495235205629094
R-squared Score: 0.622075845135081
```

[16]:
```
regressor.score(X_test, y_test)
```

[16]:
```
0.622075845135081
```

Here the score is too bad simply because we have not done any pre-processing with the dataset. I have just shown you how you can use dcision trees for regression. To learn more about regression, check out our regression tutorial.