lab4

February 18, 2024

1 Lab 4 - Tree-based Learning: Classifiying Particle Physics Data

In this lab, you will learn to: * Build a decision tree manually * Build a classifer using random forest with sklearn * Visualize decision trees * Build regression model using random forest * Learn about additional performance metrics and hyper-parameter tuning. * Build a classifier for particle physics data

Authors: Tuan Do & Bernie Boscoe

Latest Revision: Tuan Do

```
[1]: # put your imports here
     import numpy as np
     import pandas as pd
     import pylab as plt
```

2 Part A - Building a decision tree by hand

From Question 2 from Chapter 4 of Kelleher

A convicted criminal who reoffends after release is known as a recidivist. The table below lists a dataset that describes prisoners released on parole, and whether they reoffended within two years of release. This dataset lists six instances where prisoners were granted parole. Each of these instances are described in terms of three binary descriptive features. (GOOD BEHAVIOR, AGE ă 30, DRUG DEPENDENT) and a binary target feature, RECIDIVIST. The GOOD BEHAVIOR feature has a value of true if the prisoner had not committed any infringements during incarceration, the AGE ă 30 has a value of true if the prisoner was under 30 years of age when granted parole, and the DRUG DEPENDENT feature is true if the prisoner had a drug addiction at the time of parole. The target feature, RECIDIVIST, has a true value if the prisoner was arrested within two years of being released; otherwise it has a value of false.

Note: This example of predicting recidivism is based on a real application of machine learning: parole boards do rely on machine learning prediction models to help them when they are making their decisions. See Berk and Bleich (2013) for a recent comparison of different machine learning models used for this task. Datasets dealing with prisoner recidivism are available online, for example: catalog.data.gov/dataset/prisoner-recidivism/. The dataset presented here is not based on real data.

2.0.1 Part A1

2.0.2 Question 1

(5 pts)

Using this dataset, construct the decision tree that would be generated by the ID3 algorithm, using entropy-based information gain. Show your calculations for information gain and draw your final tree. Hint: follow the example in 4.3.1 in Kelleher. Note, you can upload a separate PDF for this question if you would like to draw or calculate on paper.

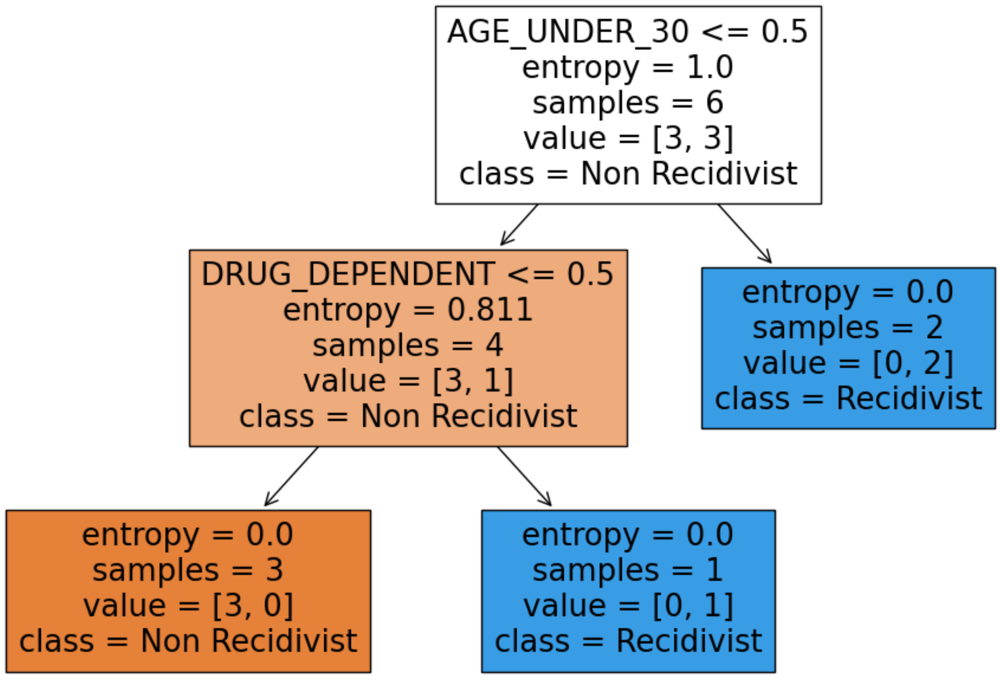
Note: I wanted to answer this in code and build the manual decision tree for the practice, I followed the logic of this Play Golf example https://github.com/milaan9/Python_Decision_Tree_and_Random_Forest/blob/main/001_Decision_Tree_Pla on github and modeled after this ID3 from scratch tutorial, https://www.geeksforgeeks.org/iterative-dichotomiser-3-id3-algorithm-from-scratch/ I wrote the code manually as recomended and tried to make changes and restructures where I could see how.

[2]:
```
from math import log2

def entropy(target_col):
    elements, counts = np.unique(target_col, return_counts=True)
    return sum((-counts[i] / np.sum(counts)) * log2(counts[i] / np.sum(counts))␣
 ↪for i in range(len(elements)))

def InfoGain(data, split_attribute_name, target_name="RECIDIVIST"):
    total_entropy = entropy(data[target_name])
    vals, counts = np.unique(data[split_attribute_name], return_counts=True)
    Weighted_Entropy = sum((counts[i] / np.sum(counts)) * entropy(data.
 ↪where(data[split_attribute_name] == vals[i]).dropna()[target_name]) for i in␣
 ↪range(len(vals)))
    return total_entropy - Weighted_Entropy

def ID3(data, originaldata, features, target_attribute_name="RECIDIVIST",␣
 ↪parent_node_class=None):
    # If all target_values have the same value, return this value
    if len(np.unique(data[target_attribute_name])) <= 1:
        return np.unique(data[target_attribute_name])[0]
    # If the dataset is empty, return the mode target feature value in the␣
 ↪original dataset
    elif len(data) == 0:
        return np.unique(originaldata[target_attribute_name])[np.argmax(np.
 ↪unique(originaldata[target_attribute_name], return_counts=True)[1])]
    # If the feature space is empty, return the mode target feature value of␣
 ↪the direct parent node
    elif len(features) == 0:
        return parent_node_class
    else:
        parent_node_class = np.unique(data[target_attribute_name])[np.argmax(np.
↪unique(data[target_attribute_name], return_counts=True)[1])]
        item_values = [InfoGain(data, feature, target_attribute_name) for␣
↪feature in features]
        best_feature_index = np.argmax(item_values)
        best_feature = features[best_feature_index]

        # Print the feature and the corresponding information gain
        print(f"Splitting on feature {best_feature} with gain =␣
↪{item_values[best_feature_index]}")

        tree = {best_feature: {}}
        features = [i for i in features if i != best_feature]

        for value in np.unique(data[best_feature]):
            sub_data = data.where(data[best_feature] == value).dropna()
            subtree = ID3(sub_data, originaldata, features,␣
↪target_attribute_name, parent_node_class)
            tree[best_feature][value] = subtree

        return tree

def build_tree(data, target_name="RECIDIVIST"):
    features = data.columns.tolist()
    features.remove(target_name)
    return ID3(data, data, features, target_name)

# Sample data
data = pd.DataFrame({
    'GOOD_BEHAVIOR': [False, False, False, True, True, True],
    'AGE_UNDER_30': [True, False, True, False, False, False],
    'DRUG_DEPENDENT': [False, False, False, False, True, False],
    'RECIDIVIST': [True, False, True, False, True, False]
})

decision_tree = build_tree(data)
print("Decision Tree:")
print(decision_tree)
```

```
Splitting on feature AGE_UNDER_30 with gain = 0.4591479170272448
Splitting on feature DRUG_DEPENDENT with gain = 0.8112781244591328
Decision Tree:
{'AGE_UNDER_30': {False: {'DRUG_DEPENDENT': {False: False, True: True}}, True:
True}}
```

[3]:
```
# To 'draw' the tree I'll create an identical tree with sklearn and use the␣
↪tree plot
```

```
from sklearn.tree import DecisionTreeClassifier
from sklearn import tree
import matplotlib.pyplot as plt
import pandas as pd

X = data.drop('RECIDIVIST', axis=1)
y = data['RECIDIVIST'].astype(int)  # Target is binary encoded

decision_tree_classifier = DecisionTreeClassifier(criterion='entropy')
decision_tree_classifier.fit(X, y)

# Plot the trained decision tree
plt.figure(figsize=(12, 8))
tree.plot_tree(decision_tree_classifier,
               feature_names=['GOOD_BEHAVIOR', 'AGE_UNDER_30',␣
 ↪'DRUG_DEPENDENT'],
               class_names=['Non Recidivist', 'Recidivist'],
               filled=True)
plt.show()
```



2.0.3 Part A2

2.0.4 Question 2

(1 pt)

What prediction will the decision tree generated in part 1.1 return for the following query? GOOD BEHAVIOR = false,AGE < 30 = false, DRUG DEPENDENT = true

Recidivism = true

2.0.5 Part A3

2.0.6 Question 3

(1 pt)

What prediction will the decision tree generated in part 1.1 predict for the following query?

GOOD BEHAVIOR = true,AGE > 30 = true, DRUG DEPENDENT = false

Recidivism = false

2.0.7 Part A4

2.0.8 Question 4

(1 pt)

Discuss why this decision tree might lead to biases in inferences about recividism.

The data is very limited, according to this data age < 30 almost completely determines recidivism on its own which is certainly not true. With 3 features and just 6 examples any model is sure to have poor general predictions since the dataset cannot be representative. We need much more data and the data must be sourced from a wide variety of individuals in order to mitigate its biases.

3 Part B - Classify irises with a decision tree

Here, we'll repeat the iris classification problem with a decision tree using continuous features. We'll also be using the sklearn decision tree classifier.

[4]:
```python
# load the data and the module to visualize decision trees
from sklearn import datasets
from sklearn import tree
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

3.1 B1 Building a decision tree with 2 features

3.2 Question 5

(1 pt)

Load the iris data set. Let's start by using only **2 features: sepal length and width**

```
[5]: # enter code here, hint: use your code from  your previous labs to load the␣
      ↪iris dataset
     iris = datasets.load_iris()
     iris_df = pd.DataFrame(data= np.c_[iris['data'], iris['target']], columns=␣
      ↪iris['feature_names'] + ['target'])

     data = iris_df[['sepal length (cm)', 'sepal width (cm)']].values
     targets = iris_df[['target']].values
```

Split the data into training and testing data

```
[6]: from sklearn.model_selection import train_test_split

     # enter code here

     # Split up data
     X_train, X_test, y_train, y_test = train_test_split(data, targets, test_size=0.
      ↪2, random_state=42)
```

3.3 Question 6

(2 pts)

Load the classifier

See the sklearn documentation for more info: https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html

```
[7]: # load the classifier
     from sklearn.tree import DecisionTreeClassifier

     IrisTreeCLF = DecisionTreeClassifier(random_state=616)
```

Train the classifier

```
[8]: # enter your code here
```

```
[9]: IrisTreeCLF.fit(X_train,y_train)
```

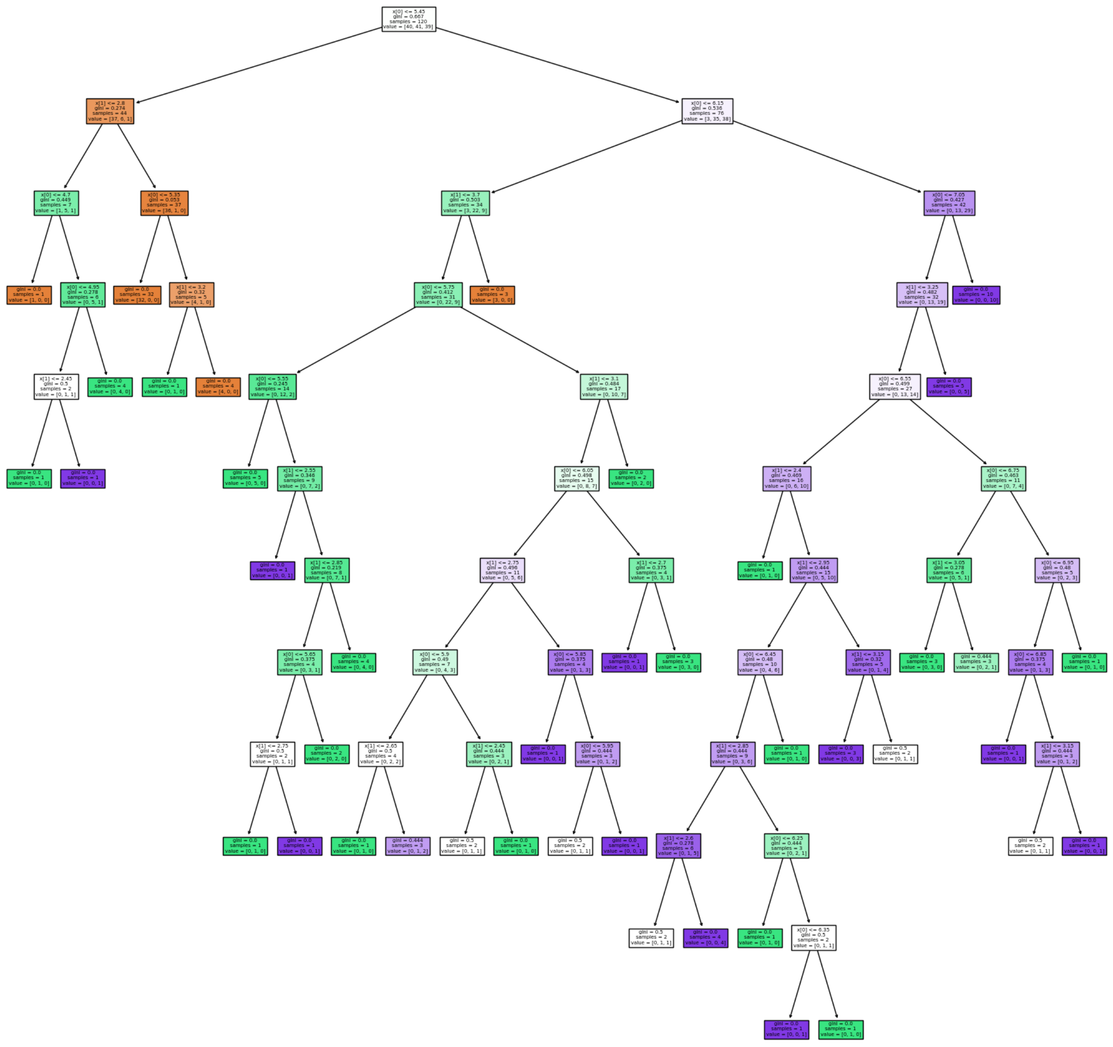
```
[9]: DecisionTreeClassifier(random_state=616)
```

```
[10]: IrisTreeCLF.get_depth()
```

```
[10]: 11
```

Visualize your decision tree. Sklearn has a nice function called tree.plot_tree that will plot the tree you created.

[11]:
```
# visualize the tree that was created
plt.figure(figsize=(20,20))
t = tree.plot_tree(IrisTreeCLF,filled=True)
```



3.4 Question 7

(2 pts)

Describe the tree that you created. **Where which feature did the root node use to split? Why? How deep is it? Why is it that deep?**

It initially split on the feature X[0] having the value less than or equal to 5.45. X[0] is the Sepal length and this was used for the split as this feature had the maximum information gain. The tree has a depth of 11. We can understand the reason for this depth by considering the scatter plot of the data points corresponding to each of the three classes. There is a complex decision boundary so we should expect a non trivial amount of depth to correctly classifiy unlike in the recidivism tree.

Test the classifier using the **accuracy score**.

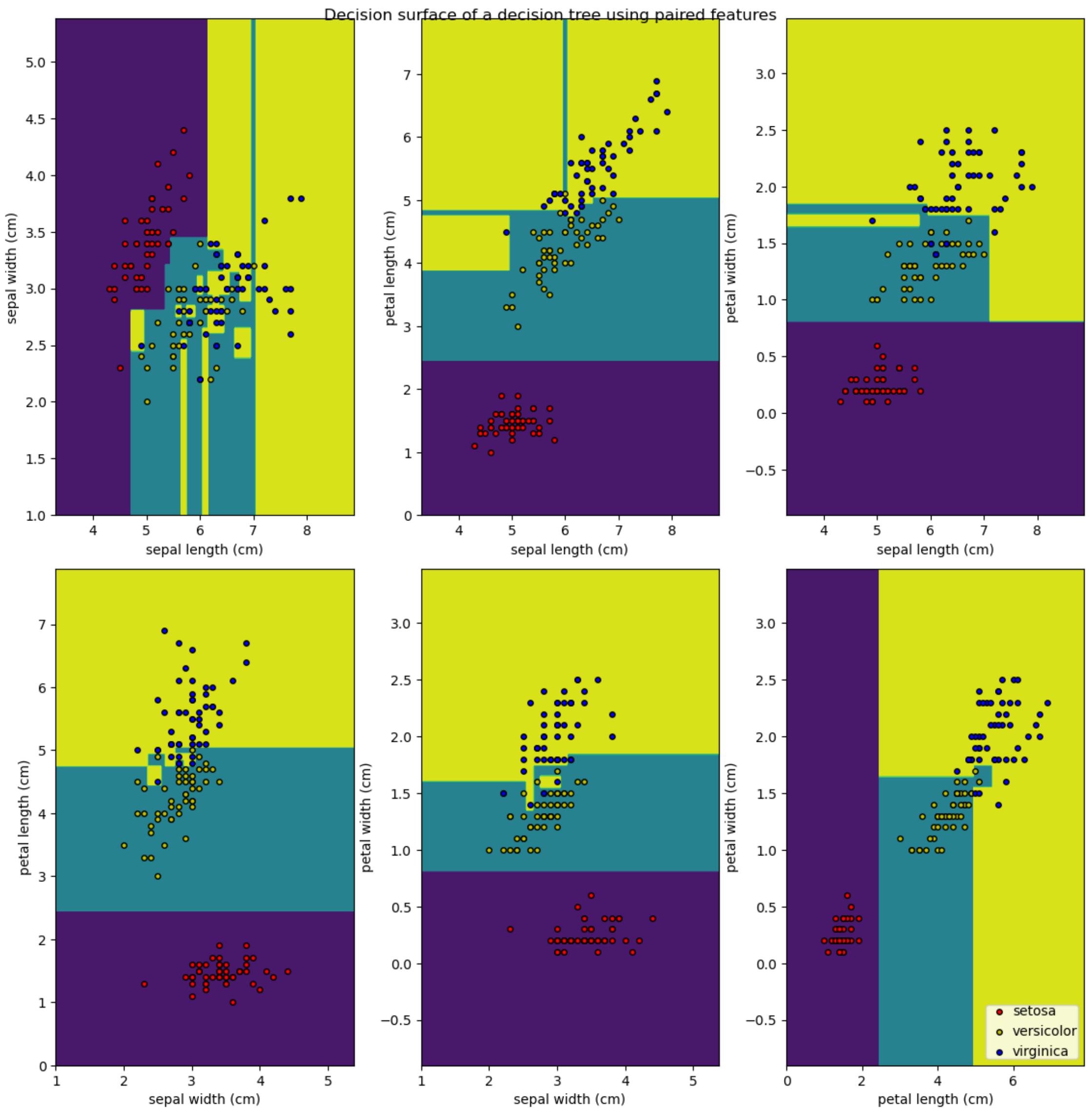
[12]:
```
# enter your code here
from sklearn.metrics import accuracy_score

y_pred = IrisTreeCLF.predict(X_test)
print(100*accuracy_score(y_test, y_pred), '%')
```

```
63.33333333333333 %
```

Visualization: we can visualize how different pairs of iris features would give in terms of their classification in the iris problem. **Run the code cell below to see this decision surface for different pairs of features.**

[13]:
```
# Parameters
n_classes = 3
plot_colors = "ryb"
plot_step = 0.02

# Load data
plt.figure(figsize=(12,12))

for pairidx, pair in enumerate([[0, 1], [0, 2], [0, 3],
                               [1, 2], [1, 3], [2, 3]]):
    # We only take the two corresponding features
    X = iris.data[:, pair]
    y = iris.target

    # Train
    clf = tree.DecisionTreeClassifier().fit(X, y)

    # Plot the decision boundary
    plt.subplot(2, 3, pairidx + 1)

    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, plot_step),
                         np.arange(y_min, y_max, plot_step))
    plt.tight_layout(h_pad=0.5, w_pad=0.5, pad=2.5)

    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    cs = plt.contourf(xx, yy, Z)

    plt.xlabel(iris.feature_names[pair[0]])
    plt.ylabel(iris.feature_names[pair[1]])

    # Plot the training points
    for i, color in zip(range(n_classes), plot_colors):
        idx = np.where(y == i)
        plt.scatter(X[idx, 0], X[idx, 1], c=color, label=iris.target_names[i],␣
↪edgecolor='black', s=15)

plt.suptitle("Decision surface of a decision tree using paired features")
plt.legend(loc='lower right', borderpad=0, handletextpad=0)
plt.axis("tight")
```

[13]: (0.0, 7.88, -0.9, 3.4800000000000044)



3.5 Question 8

(1 pt)

Based on the plot above, **identify some regions where overfitting is occurring.**

In the sepal length vs sepal width plot we see a sort of pockmarked set of boundary conditions which are a fit for this specific dataset but are unlikely to generalize well and are likely overfit. Specifically the region that differentiates veriscolor from virginica is an issue since many of the datapoints for those class intermingle a bit in the scatter plots while setosa is easily classified and distinct from the others.

3.6 Question 9

(2 pt)

By default, the decision tree can be too deep, which can make it overfit. **Recreate the classifier and use the keyword `max_depth` in the `DecisionTreeClassifier` object to specify a depth for your tree. Try refitting the data. Does your accuracy score improve?**

[14]:
```
# your code here
IrisTreeCLF_dep2 = DecisionTreeClassifier(max_depth=2, random_state=616)
IrisTreeCLF_dep2.fit(X_train, y_train)
y_pred_dep2 = IrisTreeCLF_dep2.predict(X_test)
print(100*accuracy_score(y_test, y_pred_dep2), '%')
```

```
80.0 %
```

3.7 B2 Building a decision tree with all features

3.8 Question 10

(4 pts)

Now that you have a sense of how the decision tree works with sklearn, **build a decision tree to classify irises using all the features and test it.**

[15]:
```
# your code here

# Instantiate full dataset
iris = datasets.load_iris()
iris_df = pd.DataFrame(data= np.c_[iris['data'], iris['target']], columns=␣
 ↪iris['feature_names'] + ['target'])

data = iris_df[['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)',␣
 ↪'petal width (cm)']].values
targets = iris_df[['target']].values
```

```
X_train, X_test, y_train, y_test = train_test_split(data, targets, test_size=0.
 ↪2, random_state=818)

# Build Classifier
IrisFullCLF = DecisionTreeClassifier(max_depth=2, random_state=42)
IrisFullCLF.fit(X_train, y_train)

# Test Classifier
y_pred = IrisFullCLF.predict(X_test)
print(100*accuracy_score(y_test, y_pred), '%')
```

```
86.66666666666667 %
```

3.9 Question 11

(4 pts)

Based on your new tree, answer the following questions:

- **What feature does this tree split on? Why?**
- **What is your new accuracy score?**
- **Examine whether your score improves by changing `max_depth`.**
- By default, the classifer uses the Gini index to split the nodes. **What happens if you use information entropy instead?**

[16]:
```
# Examine the feature importances to see what feature the tree splits on␣
 ↪initially
feature_importances = IrisFullCLF.feature_importances_

# Testing different max_depth values
depths = [1, 2, 3, 4, 5, 6, None] # None implies expanding until all leaves are␣
 ↪pure or contain < min_samples_split samples.
accuracy_scores = []

for depth in depths:
    clf = DecisionTreeClassifier(max_depth=depth, random_state=42)
    clf.fit(X_train, y_train)
    y_pred = clf.predict(X_test)
    accuracy_scores.append(100*accuracy_score(y_test, y_pred))

# Test with information entropy
IrisFullCLF_entropy = DecisionTreeClassifier(max_depth=None,␣
 ↪criterion='entropy', random_state=42)
IrisFullCLF_entropy.fit(X_train, y_train)
y_pred_entropy = IrisFullCLF_entropy.predict(X_test)
accuracy_entropy = 100*accuracy_score(y_test, y_pred_entropy)

# Feature importances printout
```

```
features = ['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)',␣
↪'petal width (cm)']
importances = feature_importances
feature_importance_str = "\n".join([f"{feature}: {importance*100:.2f}%" for␣
↪feature, importance in zip(features, importances)])

# Accuracy improvements with different depths
accuracy_improvement_str = "\n".join([f"max_depth={('unlimited' if depth is␣
↪None else depth)}: {accuracy:.2f}%" for depth, accuracy in zip(depths,␣
↪accuracy_scores)])

# Accuracy using information entropy
accuracy_entropy_str = f"Accuracy using Information Entropy (max_depth=None):␣
↪{accuracy_entropy:.2f}%"

# Printouts
printout = f"Feature Importances:\n{feature_importance_str}\n\nAccuracy by Max␣
↪Depth:\n{accuracy_improvement_str}\n\n{accuracy_entropy_str}"
print(printout)
```

```
Feature Importances:
sepal length (cm): 0.00%
sepal width (cm): 0.00%
petal length (cm): 55.09%
petal width (cm): 44.91%

Accuracy by Max Depth:
max_depth=1: 56.67%
max_depth=2: 86.67%
max_depth=3: 90.00%
max_depth=4: 90.00%
max_depth=5: 90.00%
max_depth=6: 90.00%
max_depth=unlimited: 90.00%

Accuracy using Information Entropy (max_depth=None): 90.00%
```

The model found petal length and width to be useful in distinguishing between different iris species, with petal length being slightly more influential than petal width. This makes sense based on the analysis of the iris dataset we preformed in previous labs. The percentages reflect the relative importance of each feature in making splits: the higher the percentage, the more useful the feature is for splitting on.

We see that increasing the max depth to 3 improves accuracy but further increasing beyond 3 does not.

Using information entropy changes the criterion. Both Gini impurity and entropy aim to increase the homogeneity of the nodes after each split, but, they differ in how they penalize the misclassification. In practice, the choice between Gini impurity and entropy might not significantly impact the performance of most models, as we saw in this case.

4 Part C - Classify handwritten digits with random forest

As discussed in lecture and readings, usually we will not make just one decision tree, but rather use an emsemble of them called a random forest. Random forests have many applications and can even be used for image recognition. In this part, we will examine how we could use random forest to classifiy handwritten digits using another classic machine learning dataset.

This dataset is called MNIST and is used to test many machine learning models (for more information, see http://yann.lecun.com/exdb/mnist/). We'll use our usual framework for an ML workflow for this part.

4.1 C1 Big Picture

Optical character recognition (OCR) is a very common machine learning task that is used to digitize handwriting. There is a lot of variation in how people write, but the number of possible characters is fairly small, so it is a well defined task. For this part of the lab, we'll try to identify numbers, which means there will be 10 classes (zero to nine).

4.2 C2 Get the data

[17]:
```
from sklearn.datasets import load_digits
digits = load_digits()
digits.keys() # the data is in the form of a dictionary, this will let you see␣
 ↪the keys
```

[17]:
```
dict_keys(['data', 'target', 'frame', 'feature_names', 'target_names', 'images',
'DESCR'])
```

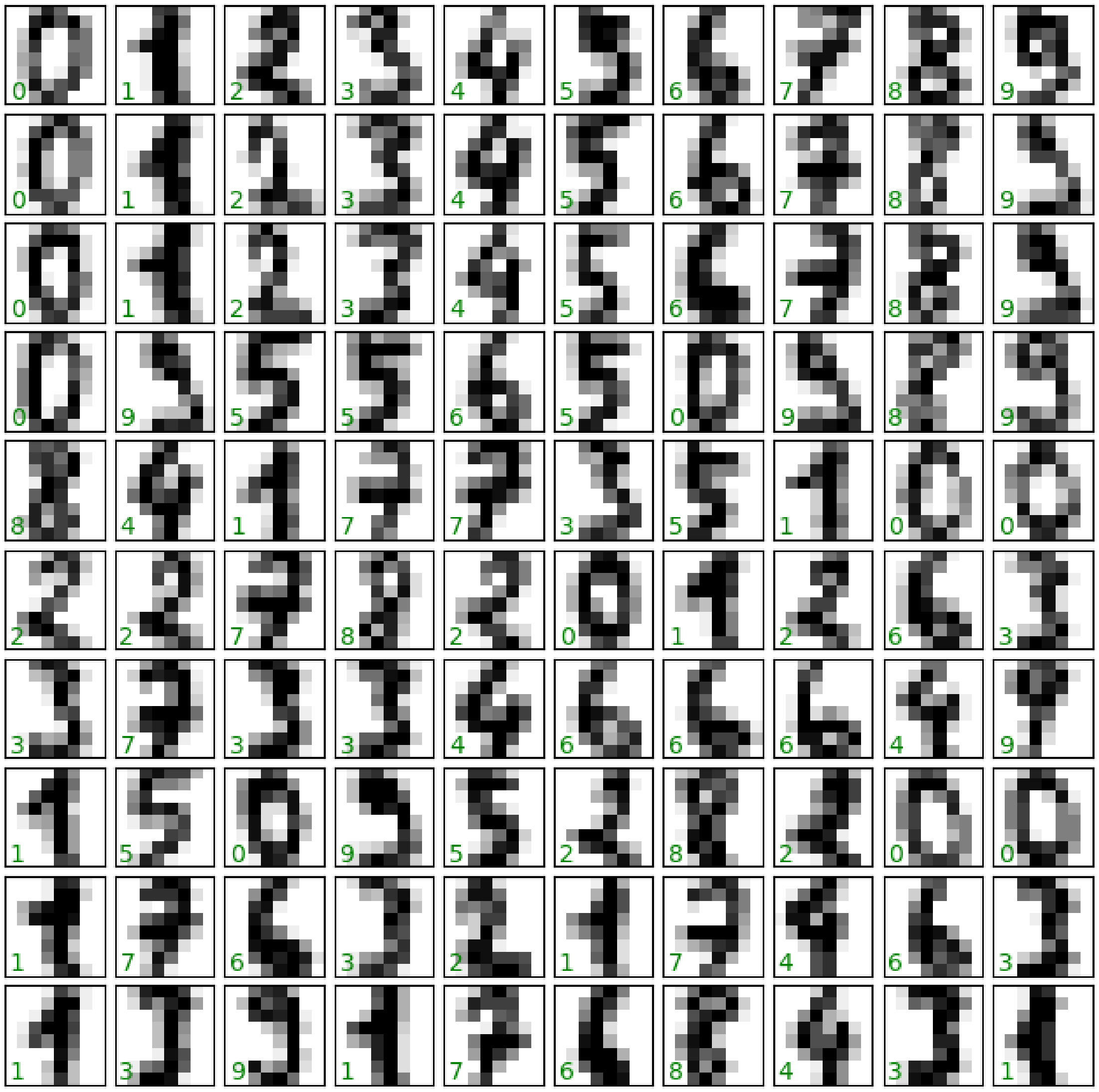
4.3 C3 Explore the data

4.4 Question 12

(2 pts)

Check the shape, and head, plot, visualize, and have a look at the data with whatever ways you think will help your task. Comment throughout on observations. There is some code below to look at some sample of digits

[18]:
```
# We can display some of the digits to see what they look like
fig, axes = plt.subplots(10, 10, figsize=(8, 8),
                         subplot_kw={'xticks':[], 'yticks':[]},
                         gridspec_kw=dict(hspace=0.1, wspace=0.1))

for i, ax in enumerate(axes.flat):
    ax.imshow(digits.images[i], cmap='binary', interpolation='nearest')
    ax.text(0.05, 0.05, str(digits.target[i]),
                  transform=ax.transAxes, color='green')
```



[19]:
```
digits.data[:5]
```

[19]:
```
array([[ 0.,  0.,  5., 13.,  9.,  1.,  0.,  0.,  0.,  0., 13., 15., 10.,
        15.,  5.,  0.,  0.,  3., 15.,  2.,  0., 11.,  8.,  0.,  0.,  4.,
        12.,  0.,  0.,  8.,  8.,  0.,  0.,  5.,  8.,  0.,  0.,  9.,  8.,
         0.,  0.,  4., 11.,  0.,  1., 12.,  7.,  0.,  0.,  2., 14.,  5.,
        10., 12.,  0.,  0.,  0.,  0.,  6., 13., 10.,  0.,  0.,  0.],
       [ 0.,  0.,  0., 12., 13.,  5.,  0.,  0.,  0.,  0.,  0., 11., 16.,
         9.,  0.,  0.,  0.,  0.,  3., 15., 16.,  6.,  0.,  0.,  0.,  7.,
        15., 16., 16.,  2.,  0.,  0.,  0.,  0.,  1., 16., 16.,  3.,  0.,
         0.,  0.,  0.,  1., 16., 16.,  6.,  0.,  0.,  0.,  0.,  1., 16.,
        16.,  6.,  0.,  0.,  0.,  0.,  0., 11., 16., 10.,  0.,  0.],
       [ 0.,  0.,  0.,  4., 15., 12.,  0.,  0.,  0.,  0.,  3., 16., 15.,
        14.,  0.,  0.,  0.,  0.,  8., 13.,  8., 16.,  0.,  0.,  0.,  0.,
         1.,  6., 15., 11.,  0.,  0.,  0.,  1.,  8., 13., 15.,  1.,  0.,
         0.,  0.,  9., 16., 16.,  5.,  0.,  0.,  0.,  0.,  3., 13., 16.,
        16., 11.,  5.,  0.,  0.,  0.,  0.,  3., 11., 16.,  9.,  0.],
       [ 0.,  0.,  7., 15., 13.,  1.,  0.,  0.,  0.,  8., 13.,  6., 15.,
         4.,  0.,  0.,  0.,  2.,  1., 13., 13.,  0.,  0.,  0.,  0.,  0.,
         2., 15., 11.,  1.,  0.,  0.,  0.,  0.,  0.,  1., 12., 12.,  1.,
         0.,  0.,  0.,  0.,  0.,  1., 10.,  8.,  0.,  0.,  0.,  8.,  4.,
         5., 14.,  9.,  0.,  0.,  0.,  7., 13., 13.,  9.,  0.,  0.],
       [ 0.,  0.,  0.,  1., 11.,  0.,  0.,  0.,  0.,  0.,  0.,  7.,  8.,
         0.,  0.,  0.,  0.,  0.,  1., 13.,  6.,  2.,  2.,  0.,  0.,  0.,
         7., 15.,  0.,  9.,  8.,  0.,  0.,  5., 16., 10.,  0., 16.,  6.,
         0.,  0.,  4., 15., 16., 13., 16.,  1.,  0.,  0.,  0.,  0.,  3.,
        15., 10.,  0.,  0.,  0.,  0.,  0.,  2., 16.,  4.,  0.,  0.]])
```

[20]:
```
digits.target.shape
```

[20]:
```
(1797,)
```

[21]:
```
digits.data.shape
```

[21]:
```
(1797, 64)
```

I've heard of MNIST before. The dataset is a collection of 8x8 pixel images of handwritten digits, from 0 to 9, and it's commonly used for practicing image classification tasks. The data array consists of 1797 samples, each with 64 features (8x8 pixels). This indicates we have 1797 images in total. The target array matches this with 1797 labels, indicating the digit each image represents.

Side Note: I recently read a cool paper published in nature by authors from UCLA and University of Sydney. They created a silver nanowire network and set it to the task of classifying MNIST digits. Pretty cool example of neuromorphic computing, wanted to present it in class but didn't find the time.

The paper is, Online Dynamical Learning and Sequence Memory with Neuromorphic Nanowire Networks: https://www.nature.com/articles/s41467-023-42470-5

4.5 C4 Prepare Data

For this assignment, the data is already prepared and is part of the digits dictionary. No need to do anything else for this part.

4.6 C5 Select model and train

4.7 Question 12

(6 pts)

Here, we'll be using the pixels as features and the digit as the target. We will use the random forest classifier from sklearn to model the data.

Remember, in this part, you'll need to :

- Split the data into training, validation, and testing
- Build a random forest classifier
- Train the classifier

[22]:
```
# your code here
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier

# Split data into training, validation, and testing sets
X_train, X_temp, y_train, y_temp = train_test_split(digits.data, digits.target,␣
 ↪test_size=0.2, random_state=42)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5,␣
 ↪random_state=42)

# Build a random forest classifier
rf_clf = RandomForestClassifier(n_estimators=100, random_state=42)

# Train the classifier on the training data
rf_clf.fit(X_train, y_train)

# Output shapes of the splits just to see
X_train.shape, X_val.shape, X_test.shape
```

[22]:
```
((1437, 64), (180, 64), (180, 64))
```

4.7.1 Performance / evaluation metrics

4.8 Question 13

(6 pts)

We have mainly been using **accuracy score** as a metric for classification, but that is just one measure of how well our predictions are. For this part, let's compute some additional metrics.

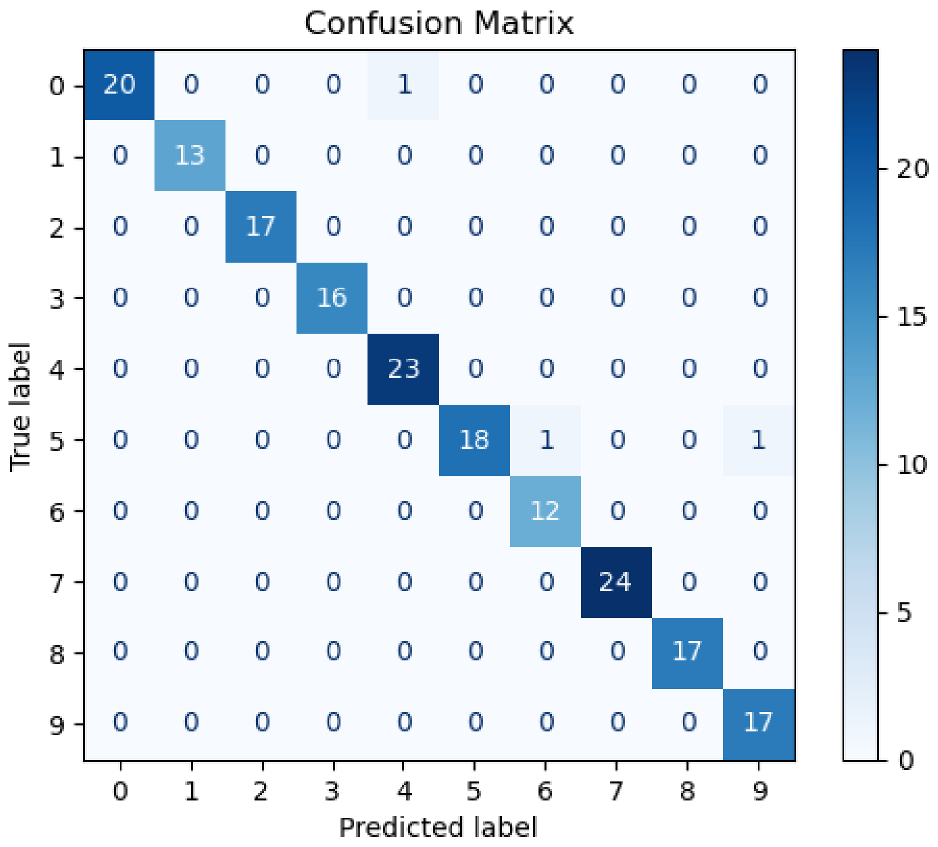
- First, compute the accuracy score of your random forest classifier.
- Compute the confusion matrix and plot it (see lab 1)
- Learn about and compute another classification metric in sklearn: https://scikit-learn.org/stable/modules/model_evaluation.html#classification-metrics

[23]:
```
from sklearn.metrics import f1_score

# Predictions on the test set
y_pred = rf_clf.predict(X_test)

# Compute the accuracy score
```

```
accuracy = accuracy_score(y_test, y_pred)

print(f"Accuracy: {accuracy:.2%}")
```

```
Accuracy: 98.33%
```

[24]:
```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

# Predictions on the test set
y_pred = rf_clf.predict(X_test)

# Compute the confusion matrix
conf_matrix = confusion_matrix(y_test, y_pred)

# Plotting the confusion matrix
disp = ConfusionMatrixDisplay(confusion_matrix=conf_matrix,␣
 ↪display_labels=rf_clf.classes_)
disp.plot(cmap=plt.cm.Blues)
plt.title('Confusion Matrix')
plt.show()
```



	0	1	2	3	4	5	6	7	8	9
0	20	0	0	0	1	0	0	0	0	0
1	0	13	0	0	0	0	0	0	0	0
2	0	0	17	0	0	0	0	0	0	0
3	0	0	0	16	0	0	0	0	0	0
4	0	0	0	0	23	0	0	0	0	0
5	0	0	0	0	0	18	1	0	0	1
6	0	0	0	0	0	0	12	0	0	0
7	0	0	0	0	0	0	0	24	0	0
8	0	0	0	0	0	0	0	0	17	0
9	0	0	0	0	0	0	0	0	0	17

```
[25]: # Compute F1 Score as an additional metric, using the 'weighted' average
f1 = f1_score(y_test, y_pred, average='weighted')
print(f"F1 Score: {f1:.2%}")
```

```
F1 Score: 98.32%
```

4.9 3.6 Fine tune model

4.10 Question 14

(6 pts)

The random forest classifier has many hyper-parameters to tune that can make your classification better. Here, you'll conduct some systematic experiments in tuning hyper-parameters to see how they affect the model performance.

- First, read the documentation for the random forest classifier to see the different hyper-parameters https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html
- Pick one of the hyper-parameters to tune (some examples are: `n_estimators`, `max_depth`, `min_samples_leaf`, `max_leaf_nodes`)
- Systematically vary one of these hyper-parameters and record your performance metric (for example accuracy score). Plot this relationship.
- If you want to search multiple parameters in a grid, sklearn also has a wrapper function that is very useful called `GridSearchCV` https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html
- Discuss the results of your experiments

```
[26]: # your code here
from sklearn.model_selection import GridSearchCV

# Define the parameter grid to search
param_grid = {
    'n_estimators': [50, 100, 200],
    'max_depth': [None, 10, 20, 30],
    'min_samples_leaf': [1, 2, 4]
}

# Create a GridSearchCV object with RandomForestClassifier
grid_search = GridSearchCV(estimator=RandomForestClassifier(random_state=42),␣
 ↪param_grid=param_grid,
                           cv=5, n_jobs=-1, scoring='accuracy', verbose=1)

# Perform the grid search on the training data
grid_search.fit(X_train, y_train)

# Best parameters and best score
best_params = grid_search.best_params_
best_score = grid_search.best_score_
```

```
# Creating a printout
best_params_str = ", ".join([f"{key}={value}" for key, value in best_params.
 ↪items()])
print(f"Best Hyperparameters: {best_params_str}")
print(f"Cross-Validated Accuracy Score: {best_score:.2%}")
```

```
Fitting 5 folds for each of 36 candidates, totalling 180 fits
Best Hyperparameters: max_depth=10, min_samples_leaf=1, n_estimators=200
Cross-Validated Accuracy Score: 97.43%
```

4.11 3.7 Present solution

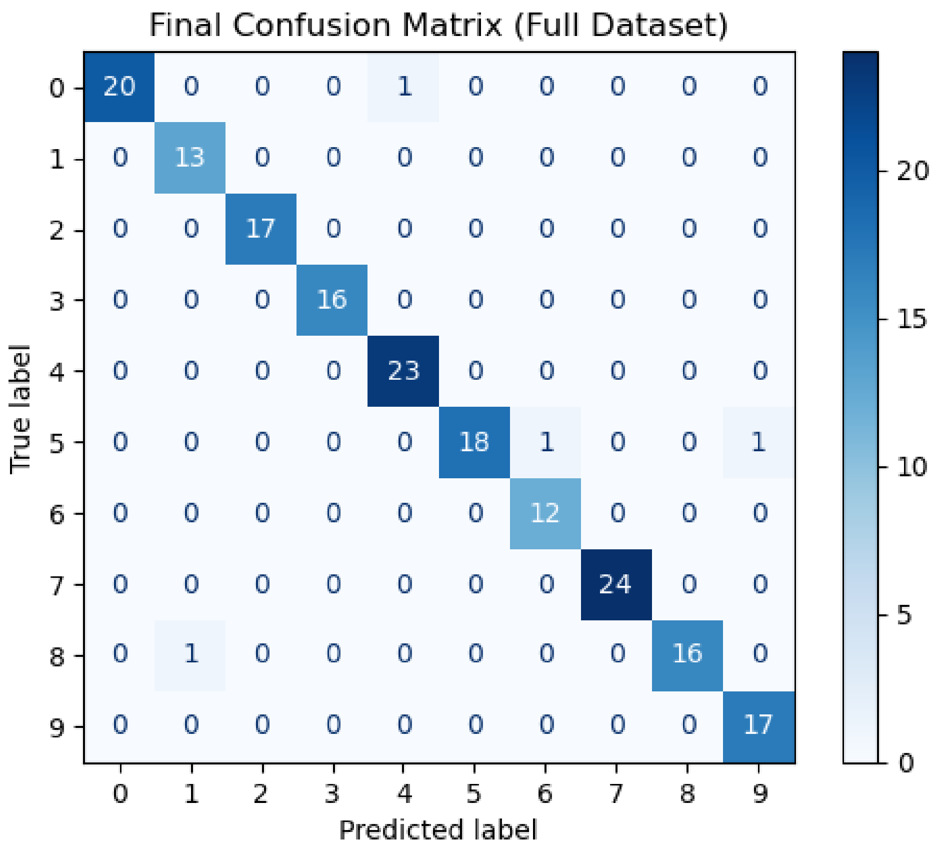
4.12 Question 15

(4 pts)

Using the testing data that you left out in the beginning, compute your final performance metrics. Also plot the final confusion matrix. Discuss where your model is good and where the model can still be improved.

[27]:
```
# Merging the training and validation sets for final training
X_train_full = np.concatenate((X_train, X_val), axis=0)
y_train_full = np.concatenate((y_train, y_val), axis=0)

# Retrain the model with the optimal hyperparameters on the full training␣
 ↪dataset
rf_clf_optimal_full = RandomForestClassifier(n_estimators=200, max_depth=10,␣
 ↪min_samples_leaf=1, random_state=42)
rf_clf_optimal_full.fit(X_train_full, y_train_full)

# Predict on the test set using the model trained on the full training dataset
y_pred_optimal_full = rf_clf_optimal_full.predict(X_test)

# Compute final performance metrics for the model trained on the full dataset
accuracy_optimal_full = accuracy_score(y_test, y_pred_optimal_full)
conf_matrix_optimal_full = confusion_matrix(y_test, y_pred_optimal_full)

# Display the accuracy
print(f"Final Accuracy ('optimal' hyper params trained on full dataset):␣
 ↪{accuracy_optimal_full:.2%}")

# Plot the final confusion matrix for the full dataset model
disp_full = ConfusionMatrixDisplay(confusion_matrix=conf_matrix_optimal_full,␣
 ↪display_labels=digits.target_names)
disp_full.plot(cmap=plt.cm.Blues)
plt.title('Final Confusion Matrix (Full Dataset)')
plt.show()
```

```
Final Accuracy ('optimal' hyper params trained on full dataset): 97.78%
```



Discussion of Results This accuracy is not too bad but classifying digits with MNIST dataset has been solved with greater accuracy to be sure. The nanowire networks achieved 93.4% accuracy which is impresive more for the novelty of the computing paradigm than for its marks. MNIST benchmarks found on https://paperswithcode.com/sota/image-classification-on-mnist show models with accuracy around 99.8+% but the top model used 1.5 million trainable parameters to achieve this. I hadn't played with this datset before but I'm surpised how quick and easy it was to obtain an accuracy of 97.8%

The model, I think is good considering how quick it was to train. It might be improved, we could do a much finer grid search over the hyper parameters if we were willing to spend extra computation time. We might also further experiment with different splits of the dataset to really maximize the accuracy.

4.13 3.8. Launch, Monitor and Maintain

Nothing to here for this lab.

5 Part D - Machine Learning in Particle Physics

5.1 D1 - The Big Picture

In this part, you will build a random forest machine learning model to recongnize types creating of top-quarks in a particle accelarator based on the observed products. We can use the decay products, energy, and momentum vectors to infer the presense of top-quarks. Beyond Standard Model Particle physics models often predict new types of particles so knowing whether the number of top-quarks (which have very high energy compared to other quarks) are as expected can be important for identifying when the Standard Model breaks down.

Before you begin this experiment, read more about the dataset in Chapter 4, Section 4.1 of the Machine Learning for Physics and Astronomy in the PDF in the lab directory (also on Bruinlearn).

5.2 D2 - Get the Data

The dataset is included here in `ParticleID_features.csv`, the target is in `'ParticleID_labels.txt`. Our goal is to differentiate between the `4-top` events (Interactions of 4 top quark) from the `ttbar` events (interaction of top and anti-top quarks background events).

[28]:
```python
## get the data

import pandas as pd
features = pd.read_csv('ParticleID_features.csv')
targets = pd.read_csv('ParticleID_labels.txt',header=None)
```

5.3 D3 - Explore the data

In this lab, you have the freedom to decide what features you would like to use to train your model and how to clean and inpute your data.

To help guide you, be sure to answer the following questions about the data. Read Chapter 4, Section 4.1 and 4.3 of the included PDF to learn more and get ideas of how to deal with cleaning the data and selecting good features to use. This chapter also has helpful pandas and sklearn code that will help make some of these tasks easier. Use the `features.describe()` to see the statistics of the columns of the `features` table.

5.4 Question 16

(4 pts)

1. What are the columns and what do they mean?
2. Why do some columns have more NaN's than others?
3. Examine your target classes. What is the distribution of the number of training samples for the two classes?
4. Make plots to explore some of the features that you are interested in using.

[29]:
```python
features
```

```
[29]:        ID       MET    METphi Type_1        P1        P2        P3        P4  \
      0       0   62803.5 -1.810010      j  137571.0  128444.0 -0.345744 -0.307112
      1       1   57594.2 -0.509253      j  161529.0   80458.3 -1.318010  1.402050
      2       2   82313.3  1.686840      b  167130.0  113078.0  0.937258 -2.068680
      3       3   30610.8  2.617120      j  112267.0   61383.9 -1.211050 -1.457800
      4       4   45153.1 -2.241350      j  178174.0  100164.0  1.166880 -0.018721
      …     …       …        …        …        …         …         …
      4995 4995  269074.0 -1.274730      j  495577.0  362590.0 -0.791914  1.671250
      4996 4996   12385.8  0.986871      j  258932.0  133559.0 -1.276540  2.970100
      4997 4997   32762.8  3.057630      b  122222.0   79947.8  0.983920 -0.399231
      4998 4998  104474.0 -1.875250      b  791028.0  457589.0  1.141530  2.934810
      4999 4999   65623.8 -0.817535      b  145360.0  138746.0  0.236765  2.277120

         Type_2        P5  … Type_12  P45  P46 P47  P48  Type_13  P49  P50 P51  \
      0       j  174209.0  …     NaN  NaN  NaN NaN  NaN      NaN  NaN  NaN NaN
      1       j  291490.0  …     NaN  NaN  NaN NaN  NaN      NaN  NaN  NaN NaN
      2       j  102423.0  …     NaN  NaN  NaN NaN  NaN      NaN  NaN  NaN NaN
      3       b   40647.8  …     NaN  NaN  NaN NaN  NaN      NaN  NaN  NaN NaN
      4       j   92351.3  …     NaN  NaN  NaN NaN  NaN      NaN  NaN  NaN NaN
      …     …       …  …      …  …   …  ..  …       …  …   …  ..
      4995    b  328278.0  …     NaN  NaN  NaN NaN  NaN      NaN  NaN  NaN NaN
      4996    j   87822.2  …     NaN  NaN  NaN NaN  NaN      NaN  NaN  NaN NaN
      4997    j  260623.0  …     NaN  NaN  NaN NaN  NaN      NaN  NaN  NaN NaN
      4998    b  304661.0  …     NaN  NaN  NaN NaN  NaN      NaN  NaN  NaN NaN
      4999    j  159680.0  …     NaN  NaN  NaN NaN  NaN      NaN  NaN  NaN NaN

            P52
      0     NaN
      1     NaN
      2     NaN
      3     NaN
      4     NaN
      …   …
      4995  NaN
      4996  NaN
      4997  NaN
      4998  NaN
      4999  NaN

      [5000 rows x 68 columns]
```

```
[30]: features.describe()
```

```
[30]:                 ID           MET       METphi            P1            P2  \
      count  5000.000000   5000.000000  5000.000000  5.000000e+03  5.000000e+03
      mean   2499.500000  64071.074332    -0.028916  3.301357e+05  1.540486e+05
      std    1443.520003  60525.122480     1.819257  3.068202e+05  1.149469e+05
```

```
min       0.000000     290.756000    -3.141010  3.857940e+04  2.825400e+04
25%    1249.750000   24352.375000    -1.619905  1.369522e+05  8.883690e+04
50%    2499.500000   46814.400000    -0.055612  2.263525e+05  1.182015e+05
75%    3749.250000   83032.350000     1.537323  4.077158e+05  1.771265e+05
max    4999.000000  692674.000000     3.141130  3.186360e+06  1.276710e+06

                P3           P4            P5            P6           P7  …  \
count  5000.000000  5000.000000  4.997000e+03  4.997000e+03  4997.000000  …
mean     -0.039812    -0.003049  2.527799e+05  1.080302e+05    -0.029936  …
std       1.361762     1.814855  2.638580e+05  8.136261e+04     1.439105  …
min      -4.110220    -3.140710  1.087540e+04  1.080000e+04    -4.668790  …
25%      -1.035570    -1.574213  1.007510e+05  6.321840e+04    -1.060500  …
50%      -0.038731    -0.009037  1.659740e+05  8.584360e+04    -0.057428  …
75%       0.943598     1.542370  2.999950e+05  1.238700e+05     1.028340  …
max       4.141410     3.138540  3.587700e+06  1.146330e+06     4.559150  …

              P43         P44           P45            P46         P47  \
count  261.000000  261.000000  1.270000e+02     127.000000  127.000000
mean     0.029455    0.026422  1.631051e+05   34876.849606    0.206978
std      1.884750    1.753017  2.248603e+05   20433.767238    1.998859
min     -4.400470   -3.130690  1.780380e+04   12987.900000   -4.447660
25%     -1.413650   -1.270700  4.365005e+04   24742.500000   -1.259230
50%     -0.088908   -0.041002  8.050910e+04   28262.800000    0.120301
75%      1.416310    1.514030  1.578350e+05   35445.700000    1.727295
max      4.790720    3.120760  1.246080e+06  167840.000000    4.691500

              P48           P49            P50        P51        P52
count  127.000000  5.600000e+01      56.000000  56.000000  56.000000
mean    -0.001085  1.456600e+05   36151.183929  -0.000879   0.219260
std      1.949004  1.943657e+05   25861.883410   1.941707   1.910400
min     -3.139820  2.512510e+04   14836.000000  -4.448760  -2.990730
25%     -1.817600  4.112588e+04   24974.125000  -1.243362  -1.490900
50%     -0.232455  9.553645e+04   27353.550000  -0.121213   0.128103
75%      1.712720  1.754910e+05   33817.950000   1.800682   1.984745
max      3.091510  1.177730e+06  155888.000000   4.151320   3.058890

[8 rows x 55 columns]
```

```
[31]: targets
```

```
[31]:        0
      0     ttbar
      1     ttbar
      2     ttbar
      3     ttbar
      4     ttbar
      …     …
```

```
4995   4top
4996  ttbar
4997  ttbar
4998   4top
4999  ttbar

[5000 rows x 1 columns]
```

Q1: Meaning of Columns The dataset consists of features from collision events in a particle accelerator, with each row representing a different event. The columns represent different aspects of the particles detected in each event, including:

MET (Missing Transverse Energy): Indicates the magnitude of energy missing from the detectors, which is transverse (perpendicular) to the direction of the particle collider beam. This is crucial for detecting particles like neutrinos that do not interact with the detector.

METphi (Azimuthal Angle of MET): The direction of the missing transverse energy in the detector's transverse plane.

obj (Object Type): Type of particle detected (e.g., electron, muon, jet, b-jet).

E (Energy): Energy of the detected particle.

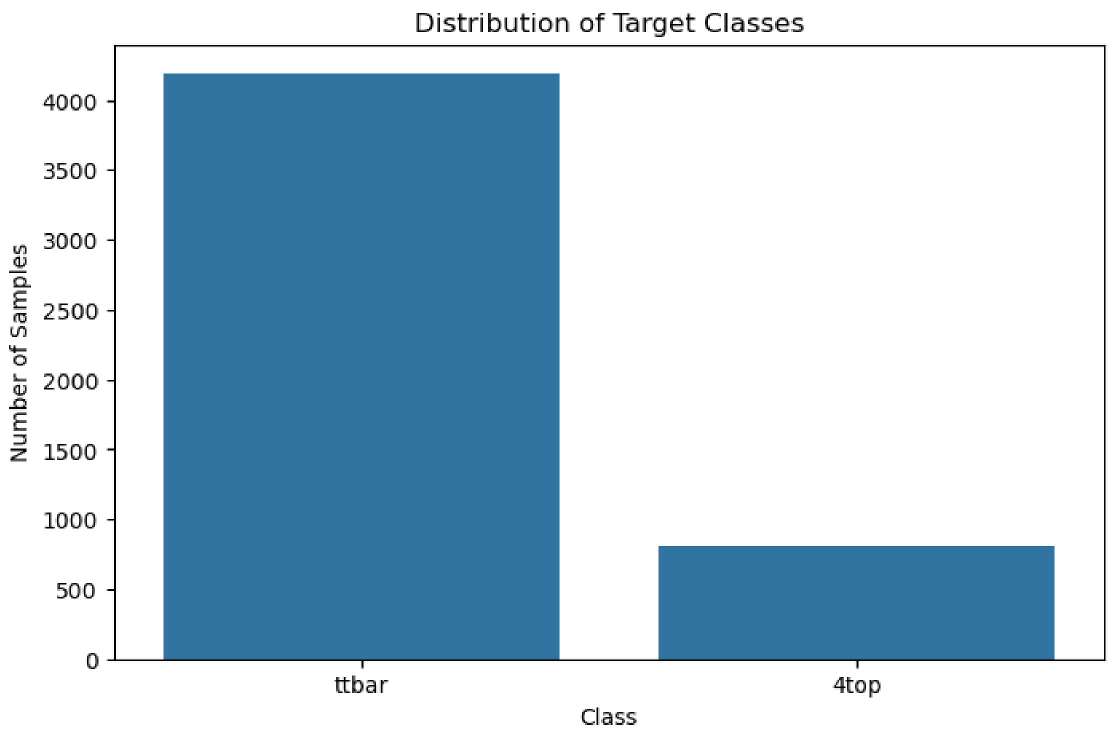
P# (Momentum): Element of the 4 momenta of the i'th particle object of type type. Four elements for each particle detected.

Particles are identified as obj with types including electrons (e-/+), photons (g), jets (j), b-jets (b), and muons (m+/m-), with each type potentially having multiple instances per event (e.g., obj1, obj2, etc.).

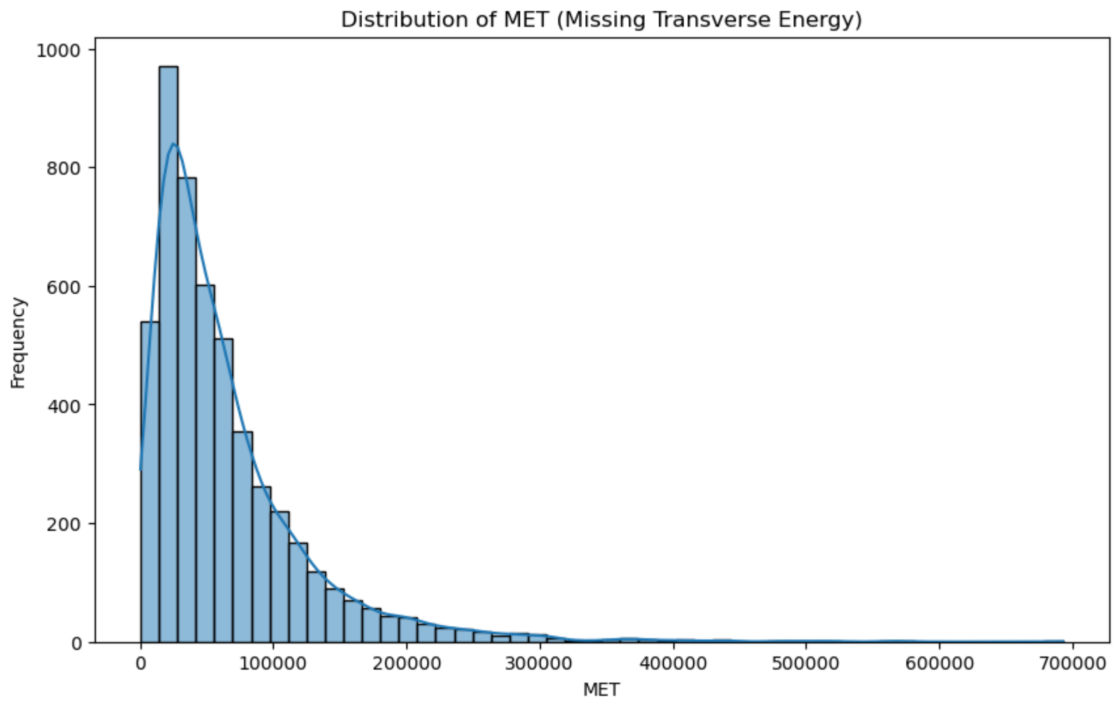
Q2: NaN's In the context of particle collision data, some columns have more NaNs than others due to the variability in collision outcomes. Not every collision event produces the same types or numbers of particles, leading to entries with missing information where a particular type of particle wasn't detected or is not relevant for that event.

[32]:
```python
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# 3. Examine target classes
# Get the distribution of the number of training samples for the two classes
target_distribution = targets[0].value_counts()
print("Target class distribution:\n", target_distribution)

# Plotting the distribution of target classes
plt.figure(figsize=(8, 5))
sns.barplot(x=target_distribution.index, y=target_distribution.values)
plt.title('Distribution of Target Classes')
plt.xlabel('Class')
plt.ylabel('Number of Samples')
plt.show()
```

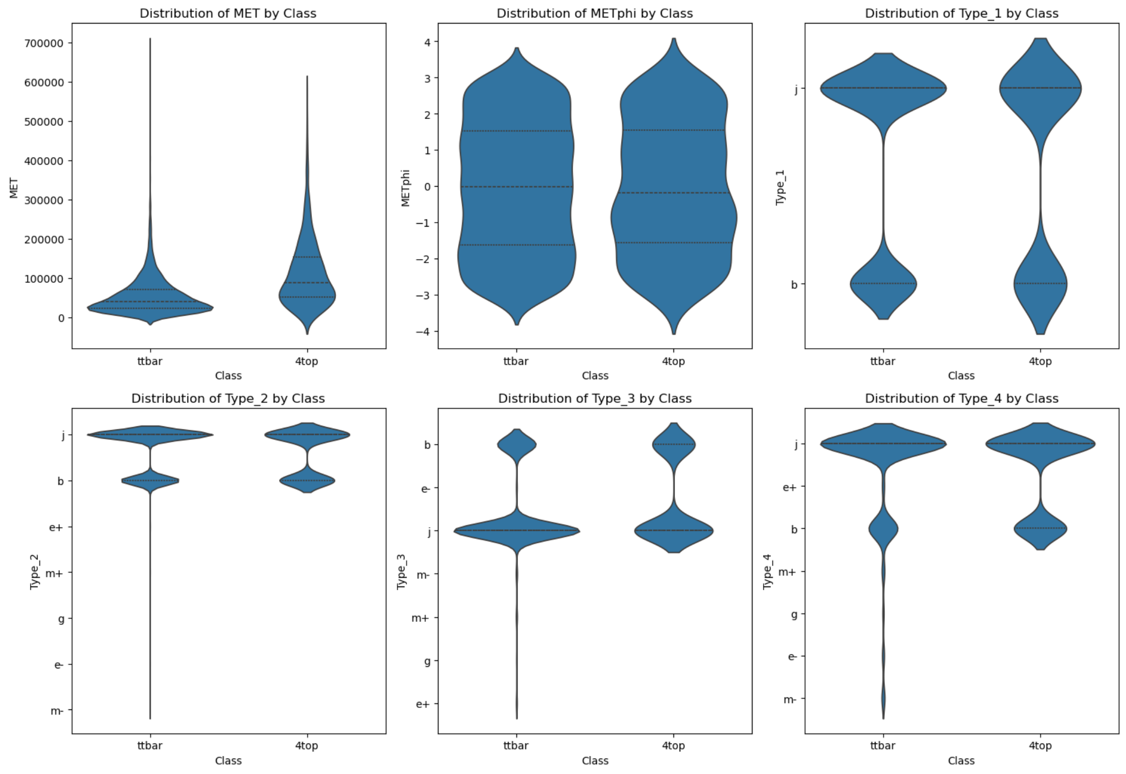
```
Target class distribution:
 0
ttbar    4189
4top      811
Name: count, dtype: int64
```



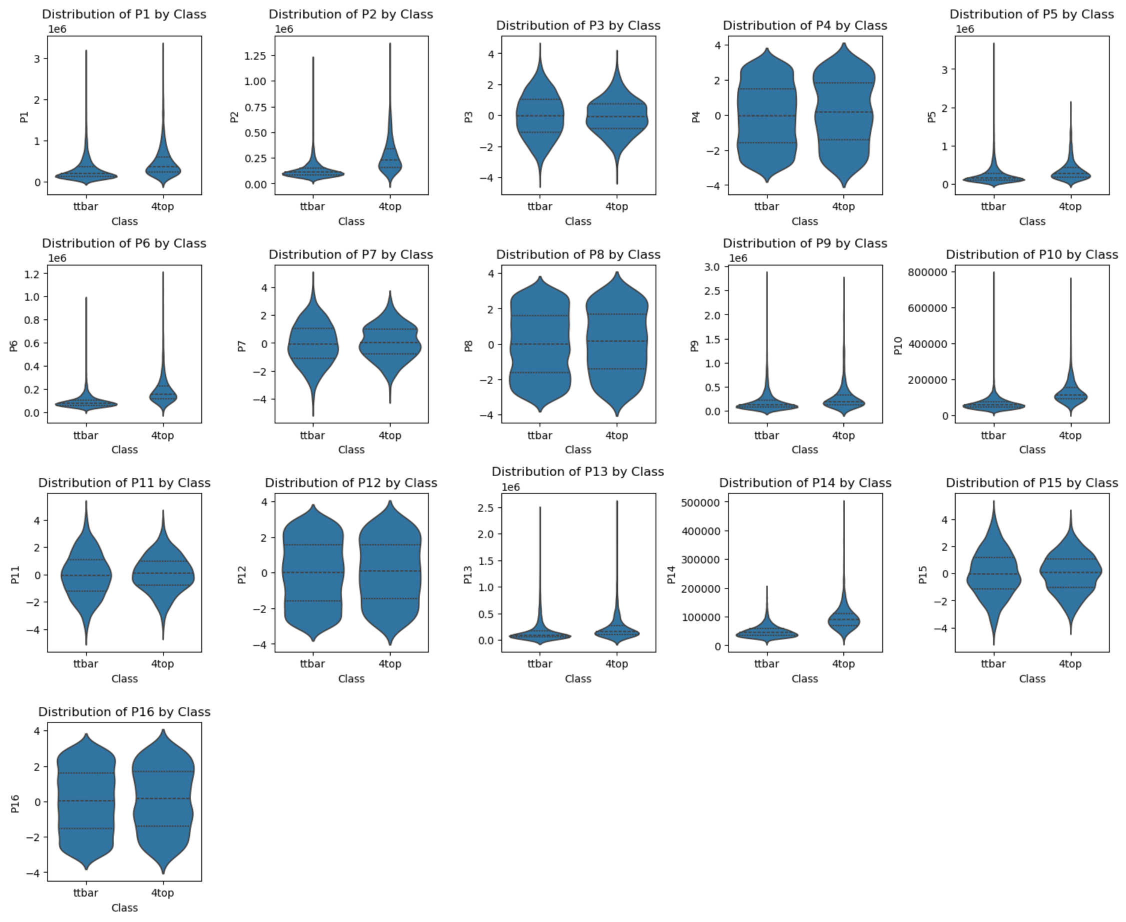
[33]:
```
# 4. Make plots to explore some of the features
# Explore MET Distribution
plt.figure(figsize=(10, 6))
sns.histplot(features['MET'], bins=50, kde=True)
plt.title('Distribution of MET (Missing Transverse Energy)')
plt.xlabel('MET')
plt.ylabel('Frequency')
plt.show()

'''Looked up this violin plot which I think could be useful to get an idea if␣
  ↪the distribution of a feature could
be a good predictor of class'''
features['class'] = targets.iloc[:, 0].values

def plot_features_distribution_grid(features, features_to_examine):
    """
    Creates a grid of violin plots to compare the distributions of specified␣
↪features across different classes.
    Also, checkout this propper docstring :)

    Parameters:
    - features: DataFrame containing the features and class labels.
    - features_to_examine: List of strings representing the feature column␣
↪names to be plotted.
    """
    num_features = len(features_to_examine)
    # Calculate grid size
    grid_size = int(num_features ** 0.5) + 1
    fig, axes = plt.subplots(grid_size, grid_size, figsize=(15, 15))
    axes = axes.flatten()  # Flatten the grid to iterate over it

    for ax, feature_to_examine in zip(axes, features_to_examine):
        sns.violinplot(x='class', y=feature_to_examine, data=features,␣
↪inner='quart', ax=ax)
        ax.set_title(f'Distribution of {feature_to_examine} by Class')
        ax.set_xlabel('Class')
        ax.set_ylabel(feature_to_examine)

    for ax in axes[len(features_to_examine):]:
        ax.set_visible(False)

    plt.tight_layout()
    plt.show()

# Features of interest
features_of_interest = ['MET', 'METphi', 'Type_1', 'Type_2', 'Type_3', 'Type_4']
target_number = 16
momenta_of_interest = [f'P{i}' for i in range(1, target_number + 1)]


# Call the function with the array of features
plot_features_distribution_grid(features, features_of_interest)
plot_features_distribution_grid(features, momenta_of_interest)
```



Distribution of MET (Missing Transverse Energy)
Frequency
MET
0
200
400
600
800
1000
0
100000
200000
300000
400000
500000
600000
700000



Distribution of MET by Class
Distribution of METphi by Class
Distribution of Type_1 by Class
Distribution of Type_2 by Class
Distribution of Type_3 by Class
Distribution of Type_4 by Class
MET
METphi
Type_1
Type_2
Type_3
Type_4
Class
ttbar
4top
j
b
e+
m+
g
e-
m-



[ ]:
```

```

5.5 D4 - Prepare the Data

Here, you'll be selecting the features and creating your training, testing, and validating data. Again, Chapter 4, Section 4.1 and 4.3 will be helpful for this.

5.6 Question 17

(6 pts)

1. Choose at least 5 features to use in your random forest model and justfy why you think these are helpful features.
2. Do you need to clean these features? Are there invalid or missing values that you need to deal with? If so describe your method for dealing with them.
3. Split your dataset into training, testing, and validation. You are free to choose what fraction you would like to use for each.

[34]:
```
#1. Feature Choice
```

I visually inspected the violin plots I created. Features that have very similar looking distributions regardless of class will not be that useful for predicting class and that features with distinct looking distributions depending class will be more useful as predictors. As noted in Ch 4.3 of the text book many values are missing for high values of the momenta since only a minority of interactions involve the greatest number of objects. We can impute or otherwise employ a strategy to fill in missing data but the ammount of data to fill is pretty substantial. As is proposed in the textbook I'll select a subset of features and try to make a compromise between truncating the dataset and overfilling it with estimated values.

To this end, I choose to use P1-P16, since this involves cutting off some 5% of the data. Further based on the violin plot inspection It seems like MET is informative while METphi is less so. Type also seems somewhat informative. For filling missing data, I'll replace NaN's with 0.

My initial attempt will use features: MET, Type 1-4, and P 1-16.

[35]:
```
features = pd.read_csv('ParticleID_features.csv')
targets = pd.read_csv('ParticleID_labels.txt',header=None)
```

[36]:
```
# Data Processing
# List of features to include: MET, Type 1-4, and P 1-16

# Map type labels to floats
unique_labels = set()
for i in range(1, 5):
    unique_labels.update(features[f'Type_{i}'].unique())

label_to_float_map = {label: float(i) for i, label in enumerate(unique_labels,␣
 ↪start=1)}

for i in range(1, 5):
    features[f'Type_{i}'] = features[f'Type_{i}'].map(label_to_float_map)

selected_features = ['MET'] + [f'Type_{i}' for i in range(1, 5)] + [f'P{i}' for␣
 ↪i in range(1, 17)]

# Create a new DataFrame with the selected features
features = features[selected_features].copy()
features = features.fillna(0)
features.head()
```

[36]:
```
       MET  Type_1  Type_2  Type_3  Type_4        P1        P2        P3  \
0  62803.5     3.0     3.0     6.0     3.0  137571.0  128444.0 -0.345744
1  57594.2     3.0     3.0     5.0     1.0  161529.0   80458.3 -1.318010
2  82313.3     6.0     3.0     3.0     3.0  167130.0  113078.0  0.937258
3  30610.8     3.0     6.0     3.0     3.0  112267.0   61383.9 -1.211050
4  45153.1     3.0     3.0     3.0     3.0  178174.0  100164.0  1.166880

         P4        P5  …        P7        P8        P9       P10       P11  \
0 -0.307112  174209.0  …  0.826569  2.332000   86788.9  84554.9 -0.180795
1  1.402050  291490.0  … -2.126740 -2.582310   44270.1  35139.6 -0.706120
2 -2.068680  102423.0  …  1.226850  0.646589   60768.9  36244.3  1.102890
3 -1.457800   40647.8  … -0.024646 -2.222800  201589.0  32978.6 -2.496040
4 -0.018721   92351.3  …  0.774114  2.568740   61625.2  50086.7  0.652572

        P12       P13      P14      P15       P16
0  2.187970  140289.0  76955.8 -1.19933 -1.302800
1 -0.371392   72883.9  26902.2 -1.65386 -3.129630
2 -1.434480   77714.0  27801.5  1.68461  1.389690
3  1.137810   90096.7  26964.5  1.87132  0.817631
4 -3.012800  104193.0  31151.0  1.87641  0.865381

[5 rows x 21 columns]
```

[37]:
```python
# Data Split
X_train_val, X_test, y_train_val, y_test = train_test_split(features, targets.
 ↪squeeze(), test_size=0.2, random_state=42)
X_train, X_val, y_train, y_val = train_test_split(X_train_val, y_train_val,␣
 ↪test_size=0.25, random_state=42)
```

5.7 D5 - Select model and train

5.8 Question 18

(3 pts)

Use a random forest classifier to train a model to classify `4-top` and `ttbar` events.

[38]:
```python
# Instantiate the Random Forest Classifier
rf_classifier = RandomForestClassifier(n_estimators=100, random_state=42)

# Train the model on the training set
rf_classifier.fit(X_train, y_train)

# Predictions can be made on the validation set or test set, e.g.,
y_pred_val = rf_classifier.predict(X_val)
```

5.9 Performance and evaluation metrics

5.10 Question 19

(6 pts)

- First, compute the accuracy score of your random forest classifier. Based on the ratio of `4-top` samples compared to the `ttbar` class, what accuracy would you expect if you always guess `ttbar`?
- Use `sklearn.metrics.classification_report` to compute some other common metrics. Descibe what each metric means.

- Compute the confusion matrix and plot it.

[39]:
```python
# Calculate the accuracy on the validation set
accuracy_val = accuracy_score(y_val, y_pred_val)
print(f"Validation Accuracy: {accuracy_val:.2%}")
```

```
Validation Accuracy: 90.80%
```

[40]:
```python
# Count the frequency of each class in the 'targets' DataFrame
class_counts = targets[0].value_counts()
# Calculate the percentage of the dataset that is class "ttbar"
ttbar_percentage = (class_counts['ttbar'] / targets.shape[0]) * 100
print(f"Percentage of the dataset that is class 'ttbar': {ttbar_percentage:.
 ↪2f}%")
```

```
Percentage of the dataset that is class 'ttbar': 83.78%
```

We'd expect over 83% accuracy from only guessing ttbar since the majority of the data corresponds to that class. Our model should do better than this at the very least.

[41]:
```python
# Use sklearn classification report
from sklearn.metrics import classification_report

# Make predictions on the test set
y_pred = rf_classifier.predict(X_test)

# Compute the classification report
report = classification_report(y_test, y_pred)
print("Classification Report:\n", report)
```

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

        4top       0.67      0.67      0.67       135
       ttbar       0.95      0.95      0.95       865

    accuracy                           0.91      1000
   macro avg       0.81      0.81      0.81      1000
weighted avg       0.91      0.91      0.91      1000
```

For the above report the metrics are defined as:

Precision: The ratio of correctly predicted positive observations to the total predicted positives. It shows how many of the items identified as belonging to a class actually belong to that class.

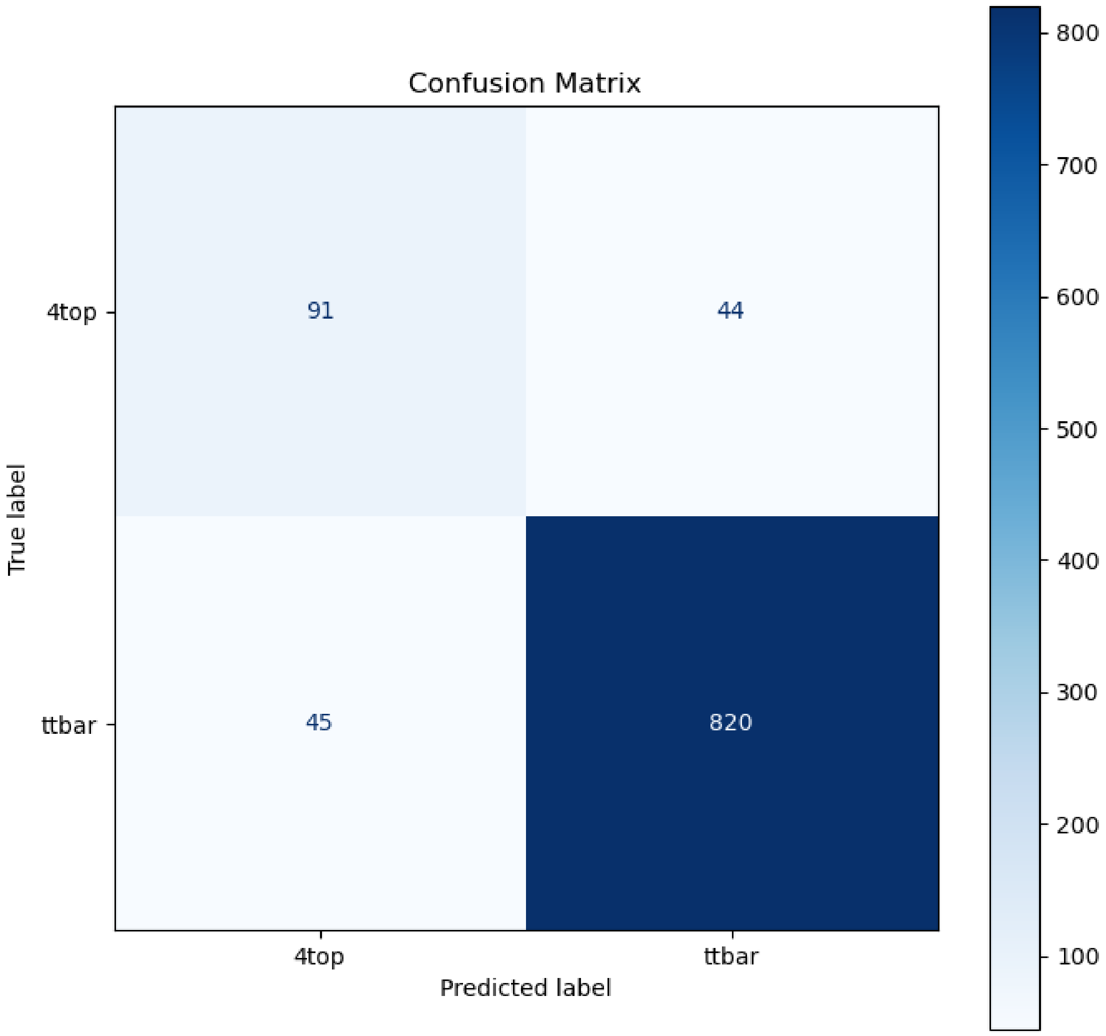
Recall (Sensitivity): The ratio of correctly predicted positive observations to all observations in the actual class. It measures how many of the actual positives our model capture through labeling it as positive.

F1-Score: The weighted average of Precision and Recall. This score takes both false positives and false negatives into account. It is a better measure than accuracy for imbalanced classes.

Support: The number of actual occurrences of the class in the specified dataset. For balanced datasets, this means the number of samples per class.

We notice the model is considerably weaker for 4top, which makes since given the assymetry of the data. We can try to improve by tuning hyperparams.

[42]:
```
# Compute the confusion matrix
conf_mat = confusion_matrix(y_test, y_pred)

# Plot the confusion matrix
fig, ax = plt.subplots(figsize=(8, 8))
ConfusionMatrixDisplay(conf_mat, display_labels=rf_classifier.classes_).
 ↪plot(cmap=plt.cm.Blues, ax=ax)
plt.title('Confusion Matrix')
plt.show()
```



5.11 D6 - Fine tune your model

5.12 Question 20

(6 pts)

Re-run your random forest classifier with other choices for hyper-parameters. Are there better choies of hyper-parameters? Describe your final model and justify your choices.

[43]:
```python
# Define the parameter grid to explore
param_grid = {
    'n_estimators': [100, 200, 300],  # Number of trees
    'max_depth': [None, 10, 20],  # Maximum depth of each tree
    'min_samples_split': [2, 5, 10],  # Corrected: Minimum number of samples␣
 ↪required to split a node
    'min_samples_leaf': [1, 2, 4]  # Minimum number of samples required at each␣
 ↪leaf node
}

# Initialize the classifier
rf = RandomForestClassifier(random_state=42)

# Initialize GridSearchCV
grid_search = GridSearchCV(estimator=rf, param_grid=param_grid, cv=5,␣
 ↪n_jobs=-1, scoring='accuracy')

# Fit GridSearchCV to the training data
grid_search.fit(X_train, y_train)

# Print the best parameters and the corresponding score
print("Best parameters:", grid_search.best_params_)
print("Best score:", grid_search.best_score_)

# Optionally, you can directly use the best estimator to make predictions
best_rf = grid_search.best_estimator_
y_pred = best_rf.predict(X_test)

# Evaluate the best model on the test set
print("Test set accuracy:", accuracy_score(y_test, y_pred))
```

```
Best parameters: {'max_depth': 20, 'min_samples_leaf': 2, 'min_samples_split':
2, 'n_estimators': 300}
Best score: 0.9019999999999999
Test set accuracy: 0.908
```

```
[44]: best_params = {
          'n_estimators': 100,
          'max_depth': 10,
          'min_samples_leaf': 2,
          'min_samples_split': 2
      }

      # Initialize the classifier with the best hyperparameters
      rf_classifier_full = RandomForestClassifier(**best_params, random_state=42)

      # Train the model on the combined training dataset
      rf_classifier_full.fit(X_train_val, y_train_val)

      # Make predictions on the test set
      y_pred_test = rf_classifier_full.predict(X_test)

      # Calculate accuracy on the test set
      test_accuracy = accuracy_score(y_test, y_pred_test)
      print(f"Test Set Accuracy: {test_accuracy:.2%}")
```

```
Test Set Accuracy: 90.70%
```

5.13 D7 - Present your solution

5.14 Question 21

(6 pts)

Describe the performance of your model on the validation data set. Describe two improvements that one could make in this work in the future (for example how to improve: data collection, feature engineering, data cleaning, model buiding, etc.).

To obtain this accuracy I began with a grid search using a wide range of values for the number of estimators, the maximum depth, the minimum samples per split and the minimum samples per leaf node, totalling 405 fits. I further ran several experiments with a narrower search grid around the previous set of optimal parameters but did not find one improving fit any further.

The final accuracy for the model is 90.7% which is comparable to the 89.5% found by the SVM presented in the textbook.

Many improvements might still be made, a less biased dataset could be helpful if obtained, more exploration with features could yield a set of features which improves the classifier, we could also consider alternative schema for cleaning the data.

6 Submit your lab

You're done! Congrats!