COMP47590 Advanced Machine Learning

Basic Machine Learning in Python - MNIST

Imports

To build predictive models in Python we use a set of libraries that are imported here. In particular **pandas** and **sklearn** are particularly important.

```
#import os
#import subprocess
import io
import random

import sys
!{sys.executable} -m pip install joblib==1.1.0
!{sys.executable} -m pip install pandas-profiling[notebook]
!{sys.executable} -m pip install scikit-learn==1.0.2
```

```
torch 2.5.1+cu124 requires nvidia-cuda-cupti-cu12==12.4.127; platform_system == "Linu
torch 2.5.1+cu124 requires nvidia-cuda-nvrtc-cu12==12.4.127; platform_system == "Linu
torch 2.5.1+cu124 requires nvidia-cuda-runtime-cu12==12.4.127; platform_system == "Li
torch 2.5.1+cu124 requires nvidia-cudnn-cu12==9.1.0.70; platform_system == "Linux" an
torch 2.5.1+cu124 requires nvidia-cufft-cu12==11.2.1.3; platform_system == "Linux" an
torch 2.5.1+cu124 requires nvidia-curand-cu12==10.3.5.147; platform_system == "Linux"
torch 2.5.1+cu124 requires nvidia-cusolver-cu12==11.6.1.9; platform_system == "Linux"
torch 2.5.1+cu124 requires nvidia-cusparse-cu12==12.3.1.170; platform_system == "Linu
torch 2.5.1+cu124 requires nvidia-nvjitlink-cu12==12.4.127; platform_system == "Linux
Successfully installed PyWavelets-1.8.0 htmlmin-0.1.12 imagehash-4.3.2 jedi-0.19.2 ma
Collecting scikit-learn==1.0.2
  Downloading scikit-learn-1.0.2.tar.gz (6.7 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 6.7/6.7 MB 37.0 MB/s eta 0:00:00
  Installing build dependencies ... done
  Getting requirements to build wheel ... done
  error: subprocess-exited-with-error

  × Preparing metadata (pyproject.toml) did not run successfully.
  │ exit code: 1
  ╰─> See above for output.

  note: This error originates from a subprocess, and is likely not a problem with pip
  Preparing metadata (pyproject.toml) ... error
error: metadata-generation-failed

× Encountered error while generating package metadata.
╰─> See above for output.

note: This is an issue with the package mentioned above, not pip.
hint: See above for details.
```

```
import pandas as pd # core data handling package
import numpy as np # core data handling package
import matplotlib # core plotting functioanlity
import matplotlib.pyplot as plt
%matplotlib inline

import seaborn as sns # nicer plotting functionlity
#import pandas_profiling # Nice exploratory data analysis package
import missingno # For nice missing number analysis

import sklearn # For basic machine learning functionality
import sklearn.preprocessing
import sklearn.metrics
import sklearn.model_selection
import sklearn.tree
import sklearn.ensemble
import sklearn.svm
import sklearn.linear_model
```

```
import sklearn.neighbors
import sklearn.neural_network
```

Data Prep

Setup

Take only a sample of the dataset for fast testing

```
data_sampling_rate = 0.1
```

Setup the number of folds for all grid searches (should be 5 - 10)

```
cv_folds = 2
```

Set up a dictionary to store simple model performance comparions

```
model_valid_accuracy_comparisons = dict()
model_accuracy_comparisons = dict()
model_tuned_params_list = dict()
```

Load Data

Load the dataset and explore it.

```
file_name = 'fashion-mnist_train.csv'

target_feature = "label"
num_classes = 10
classes = {
    0: "T-shirt/top", 1: "Trouser", 2: "Pullover", 3: "Dress",
    4: "Coat", 5: "Sandal", 6: "Shirt", 7: "Sneaker", 8: "Bag", 9: "Ankle boot"}
```

```
dataset = pd.read_csv(file_name)
dataset = dataset.sample(frac=data_sampling_rate) #take a sample from the dataset so everyht
display(dataset.head())
display(dataset.tail())
```

	label	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	pixel9	...
1189	8	0	0	0	0	0	0	0	0	1	...
1062	3	0	0	0	0	0	0	1	0	7	...
631	7	0	0	0	0	0	0	0	0	0	...
1396	2	0	0	0	0	0	0	0	0	0	...
401	6	0	0	1	0	0	0	0	0	0	...

5 rows × 785 columns

	label	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	pixel9	...
712	4	0	0	0	0	0	0	0	0	1	...
878	2	0	0	0	0	0	0	0	0	0	...
404	4	0	0	0	0	0	0	0	0	0	...
495	9	0	0	0	0	0	0	0	0	0	...
701	1	0	0	0	0	0	0	0	0	7	...

5 rows × 785 columns

Why is the Dataset in CSV Format? Instead of storing images as separate files, the pixel data is flattened and represented as numerical values in a tabular format (CSV), where each row represents an image.

Explore Data

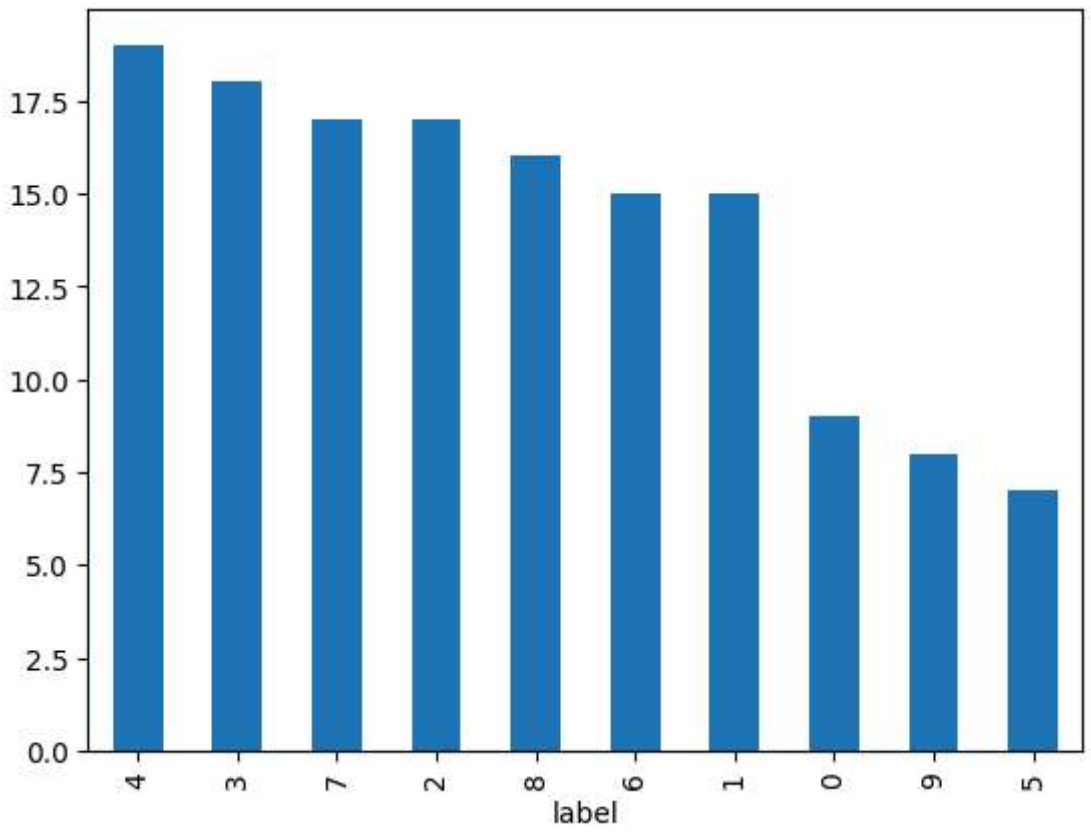
Examine the distribution of the target levels

```
dataset[target_feature].value_counts()
```

label	count
4	19
3	18
7	17
2	17
8	16
6	15
1	15
0	9
9	8
5	7

dtype: int64

```
dataset[target_feature].value_counts().plot(kind = 'bar')
plt.show()
```



Display summary statistics for each feature.

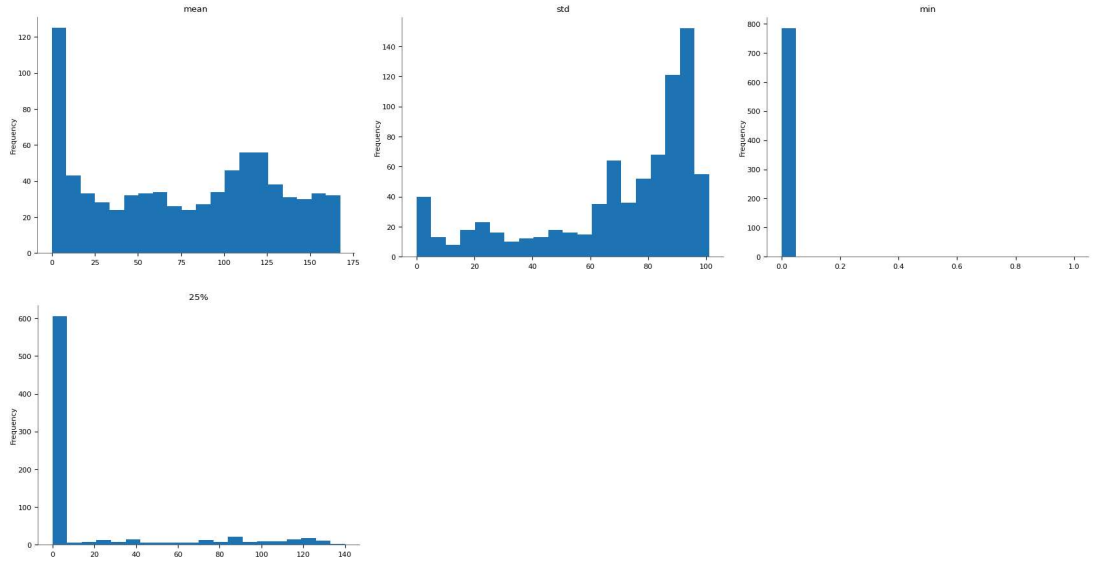
```
# Print descriptive statsitcs for each column
print("Summary Stats")
if dataset.select_dtypes(include=[np.number]).shape[1] > 0:
    display(dataset.select_dtypes(include=[np.number]).describe().transpose())

if dataset.select_dtypes(include=[object]).shape[1] > 0:
    display(dataset.select_dtypes(include=[object]).describe().transpose())
```

Summary Stats

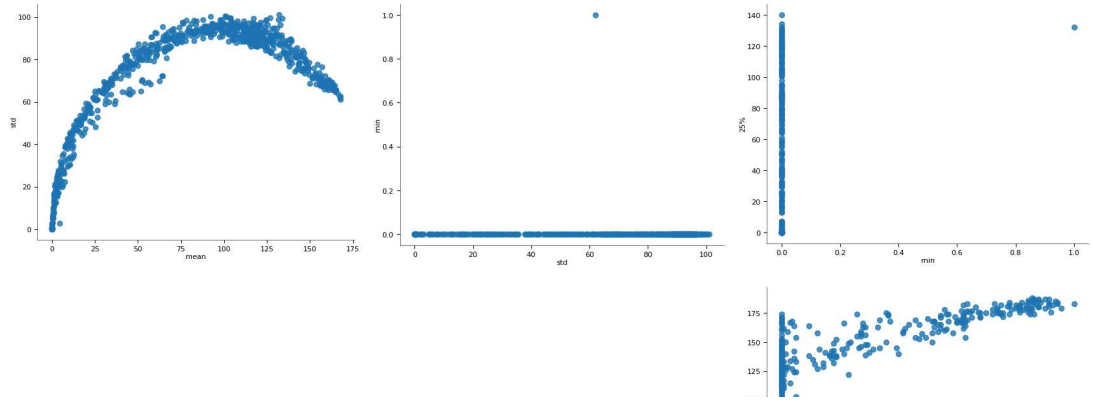
	count	mean	std	min	25%	50%	75%	max
label	141.0	4.418440	2.694322	0.0	2.0	4.0	7.0	9.0
pixel1	141.0	0.000000	0.000000	0.0	0.0	0.0	0.0	0.0
pixel2	141.0	0.000000	0.000000	0.0	0.0	0.0	0.0	0.0
pixel3	141.0	0.056738	0.354019	0.0	0.0	0.0	0.0	3.0
pixel4	141.0	0.007092	0.084215	0.0	0.0	0.0	0.0	1.0
...	...	...	...	...	...	...	...	...
pixel780	141.0	26.468085	55.909819	0.0	0.0	0.0	4.0	216.0
pixel781	141.0	11.382979	33.256720	0.0	0.0	0.0	0.0	221.0
pixel782	141.0	2.021277	11.988368	0.0	0.0	0.0	0.0	92.0
pixel783	141.0	0.765957	7.229541	0.0	0.0	0.0	0.0	82.0
pixel784	141.0	0.049645	0.589506	0.0	0.0	0.0	0.0	7.0

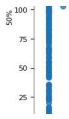
785 rows × 8 columns

Distributions



2-d distributions





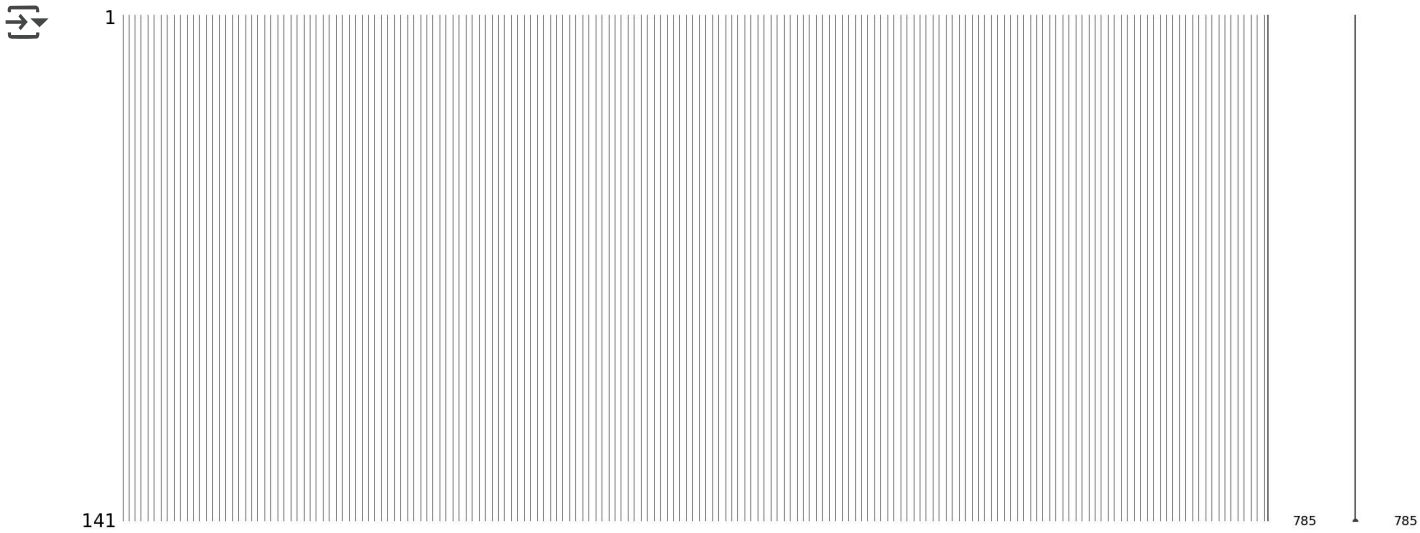
Examine presence of missing values

```
# Check for presence of missing values
print("Missing Values")
print(dataset.isnull().sum().sort_values(ascending = False))
```

```
Missing Values
label       0
pixel516    0
pixel518    0
pixel519    0
pixel520    0
           ..
pixel264    0
pixel265    0
pixel266    0
pixel267    0
pixel784    0
Length: 785, dtype: int64
```

Plot a nice diagram showing missing values - especially usefuol for combined missing values.

```
missingno.matrix(dataset)
plt.show()
```



Display some of the instances in the dataset (only really useful for images).

```
pltsize=4
row_images = 5
col_images = 5
plt.figure(figsize=(col_images*pltsize, row_images*pltsize))

for i in range(row_images * col_images):
    i_rand = random.randint(0, dataset.shape[0] -1 )
    plt.subplot(row_images,col_images,i+1)
    plt.axis('off')
    plt.imshow((dataset.iloc[i_rand, 1:]).values.reshape(28,28), cmap='gray', vmin=0, vmax=2
    plt.title(str(classes[dataset[target_feature].iloc[i_rand]]))
plt.show()
```


T-shirt/top
Trouser
Coat
Pullover
Shirt
Pullover
Pullover
Dress
Shirt
Coat
Bag
Shirt
Bag
Dress
T-shirt/top
Ankle boot
Coat
Ankle boot
T-shirt/top
Bag
Dress
Sneaker
Coat
Sneaker
Coat

Partition Dataset

Isolate the descriptive features we are interested in

```
X = dataset[dataset.columns[1:]]
y = dataset[target_feature] # its "label"
```

Split the data into a **training set** and **validation set** [70 : 30] split

```
X_train, X_valid, y_train, y_valid \
    = sklearn.model_selection.train_test_split(X, y,
                        shuffle=True,
                        stratify = y,
                        train_size = 0.7)
```

Preprocess Dataset

Normalise the data (important for some models but not used in this example.)

```
# Make the min max scalar object
#min_max_scaler = sklearn.preprocessing.MinMaxScaler((-1,1))
#min_max_scaler.fit(X_train)
#
## Train the scalar on the training dataset
#a = min_max_scaler.transform(X_train)
#
#X_train = pd.DataFrame(a, columns = min_max_scaler.feature_names_in_)
#
## Also normalise other partitions
#a = min_max_scaler.transform(X_valid)
#X_valid = pd.DataFrame(a, columns = min_max_scaler.feature_names_in_)
```

Normalise the data (using hardcoded approach based on domain knowledge)

```
X_train = (X_train/255*2) - 1
X_valid = (X_valid/255*2) - 1


display(X_train.shape)
display(X_train.head())
display(X_valid.shape)
display(X_valid.head())
```

(98, 784)

	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	pixel9	pi
89	-1.0	-1.0	-1.000000	-1.0	-1.0	-1.000000	-1.0	-0.992157	-0.992157	
511	-1.0	-1.0	-1.000000	-1.0	-1.0	-1.000000	-1.0	-0.992157	-1.000000	
1324	-1.0	-1.0	-1.000000	-1.0	-1.0	-0.984314	-1.0	-1.000000	-1.000000	
912	-1.0	-1.0	-0.976471	-1.0	-1.0	-1.000000	-1.0	-1.000000	-1.000000	
786	-1.0	-1.0	-1.000000	-1.0	-1.0	-1.000000	-1.0	-1.000000	-1.000000	

5 rows × 784 columns

(43, 784)

	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	pixel9	pixel
947	-1.0	-1.0	-1.0	-1.0	-1.0	-1.000000	-1.0	-1.0	-1.000000	-1.0000
668	-1.0	-1.0	-1.0	-1.0	-1.0	-1.000000	-1.0	-1.0	-1.000000	-0.0039
1006	-1.0	-1.0	-1.0	-1.0	-1.0	-1.000000	-1.0	-1.0	-1.000000	-0.7647
280	-1.0	-1.0	-1.0	-1.0	-1.0	-1.000000	-1.0	-1.0	-1.000000	-1.0000
588	-1.0	-1.0	-1.0	-1.0	-1.0	-0.992157	-1.0	-1.0	-0.984314	-1.0000

5 rows × 784 columns

Check that we haven't messed up the dataset!

```
# Print descriptive statsitcs for each column
print("Summary Stats")
if dataset.select_dtypes(include=[np.number]).shape[1] > 0:
    display(X_train.select_dtypes(include=[np.number]).describe().transpose())

if dataset.select_dtypes(include=[object]).shape[1] > 0:
    display(X_train.select_dtypes(include=[object]).describe().transpose())
```

Summary Stats

	count	mean	std	min	25%	50%	75%	max
pixel1	98.0	-1.000000	0.000000	-1.0	-1.0	-1.0	-1.000000	-1.000000
pixel2	98.0	-1.000000	0.000000	-1.0	-1.0	-1.0	-1.000000	-1.000000
pixel3	98.0	-0.999360	0.003317	-1.0	-1.0	-1.0	-1.000000	-0.976471
pixel4	98.0	-0.999920	0.000792	-1.0	-1.0	-1.0	-1.000000	-0.992157
pixel5	98.0	-1.000000	0.000000	-1.0	-1.0	-1.0	-1.000000	-1.000000
...	...	...	...	...	...	...	...	...
pixel780	98.0	-0.764226	0.463923	-1.0	-1.0	-1.0	-0.809804	0.694118
pixel781	98.0	-0.897879	0.287728	-1.0	-1.0	-1.0	-1.000000	0.733333
pixel782	98.0	-0.977831	0.112198	-1.0	-1.0	-1.0	-1.000000	-0.278431
pixel783	98.0	-0.991357	0.067951	-1.0	-1.0	-1.0	-1.000000	-0.356863
pixel784	98.0	-0.999440	0.005546	-1.0	-1.0	-1.0	-1.000000	-0.945098

784 rows × 8 columns

image plottign after normalization

```
pltsize=4
row_images = 5
col_images = 5
plt.figure(figsize=(col_images*pltsize, row_images*pltsize))

for i in range(row_images * col_images):
    i_rand = random.randint(0, X_train.shape[0] - 1)
    plt.subplot(row_images,col_images,i+1)
    plt.axis('off')
    plt.imshow((X_train.iloc[i_rand]).values.reshape(28,28), cmap='gray', vmin=-1, vmax=1)
   # The second plotting section uses vmin=-1 and vmax=1. This indicates that the pixel valu
    plt.title((str(classes[y_train.iloc[i_rand]])))
plt.show()
```


Coat
Pullover
Shirt
Ankle boot
Bag
Sandal
Dress
Coat
Trouser
Coat
Sneaker
Bag
Trouser
Shirt
Bag
T-shirt/top
Pullover
T-shirt/top
Bag
Dress
Shirt
Pullover
Pullover
Dress
Coat

they look the same as before, normalization is therefore effective

Building Simple Models

Train a decision tree, setting min samples per leaf to a sensible value

```
my_tree = sklearn.tree.DecisionTreeClassifier(min_samples_split = 0.05)
my_tree = my_tree.fit(X_train,y_train)
```

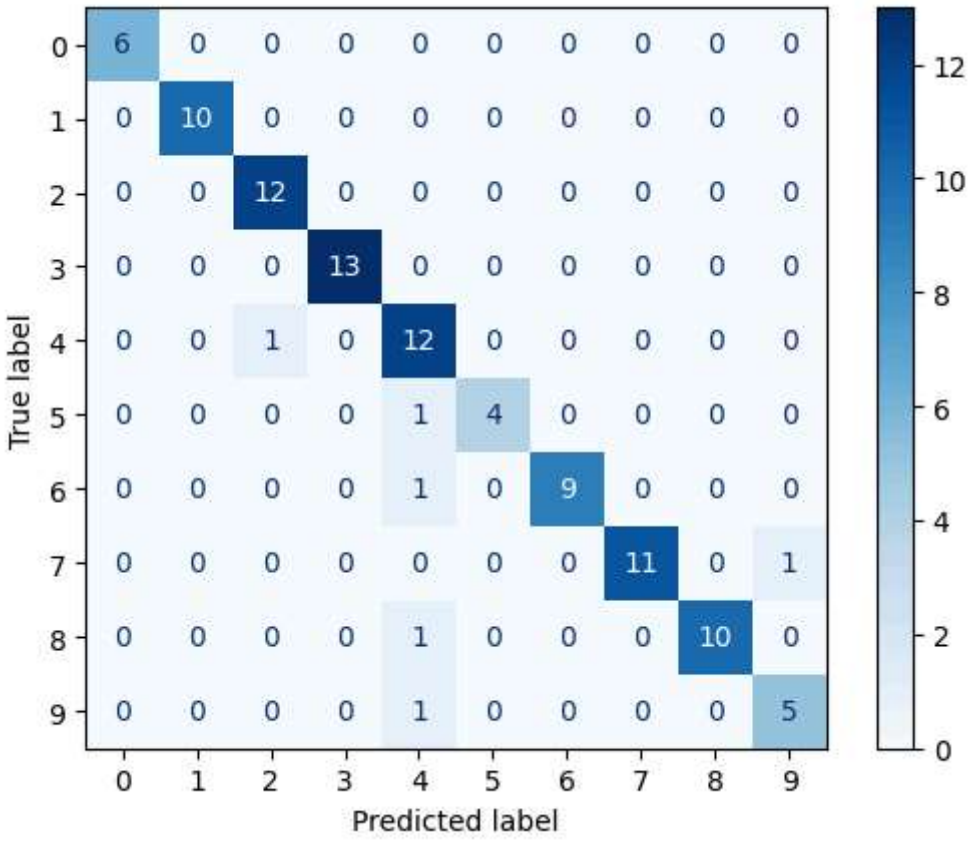
Assess the performance of the decision tree on the **training set**

```
# Make a set of predictions for the training data
y_pred = my_tree.predict(X_train)

# Print performance details
accuracy = sklearn.metrics.accuracy_score(y_train, y_pred) # , normalize=True, sample_weight
print("Accuracy: " +  str(accuracy))
print(sklearn.metrics.classification_report(y_train, y_pred))

# Print confusion matrix
print("Confusion Matrix")
```

```
sklearn.metrics.ConfusionMatrixDisplay.from_predictions(y_train, y_pred, cmap = 'Blues')
plt.show()
```

```
Accuracy: 0.9387755102040817
              precision    recall  f1-score   support

           0       1.00      1.00      1.00         6
           1       1.00      1.00      1.00        10
           2       0.92      1.00      0.96        12
           3       1.00      1.00      1.00        13
           4       0.75      0.92      0.83        13
           5       1.00      0.80      0.89         5
           6       1.00      0.90      0.95        10
           7       1.00      0.92      0.96        12
           8       1.00      0.91      0.95        11
           9       0.83      0.83      0.83         6

    accuracy                           0.94        98
   macro avg       0.95      0.93      0.94        98
weighted avg       0.95      0.94      0.94        98
```

Confusion Matrix



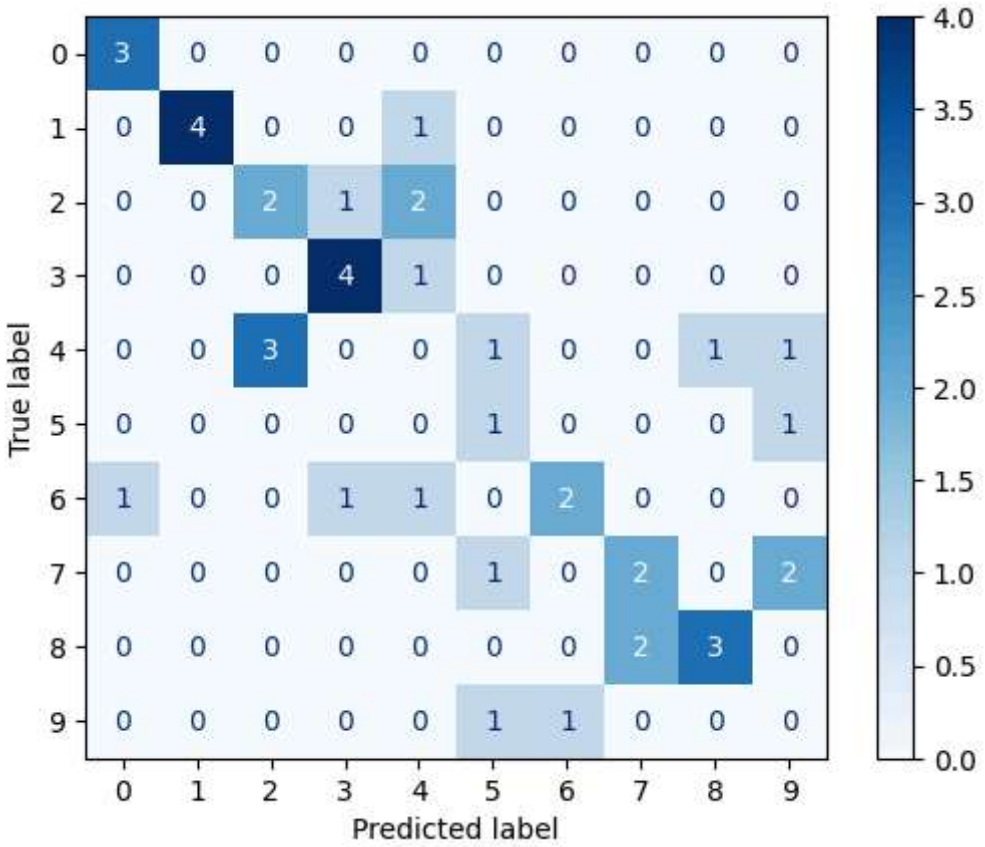
Assess the performance of the decision tree on the **validation set**

```
# Make a set of predictions for the test data
y_pred = my_tree.predict(X_valid)
```

```
# Print performance details
accuracy = sklearn.metrics.accuracy_score(y_valid, y_pred) # , normalize=True, sample_weight
model_valid_accuracy_comparisons["Better Tree"] = accuracy
print("Accuracy: " +  str(accuracy))
print(sklearn.metrics.classification_report(y_valid, y_pred))

# Print confusion matrix
print("Confusion Matrix")
sklearn.metrics.ConfusionMatrixDisplay.from_predictions(y_valid, y_pred, cmap = 'Blues')
plt.show()
```

```
Accuracy: 0.4883720930232558
              precision    recall  f1-score   support

           0       0.75      1.00      0.86         3
           1       1.00      0.80      0.89         5
           2       0.40      0.40      0.40         5
           3       0.67      0.80      0.73         5
           4       0.00      0.00      0.00         6
           5       0.25      0.50      0.33         2
           6       0.67      0.40      0.50         5
           7       0.50      0.40      0.44         5
           8       0.75      0.60      0.67         5
           9       0.00      0.00      0.00         2

    accuracy                           0.49        43
   macro avg       0.50      0.49      0.48        43
weighted avg       0.53      0.49      0.50        43
```

Confusion Matrix



The validation set avg performance are worse than the training set

Evaluating Using Cross Validation

Use a cross validation to perfrom an evaluation

```
my_tree = sklearn.tree.DecisionTreeClassifier(min_samples_split = 0.05)
cv_results = sklearn.model_selection.cross_validate(my_tree, X, y, cv=10)
print(cv_results)
```

```
/usr/local/lib/python3.11/dist-packages/sklearn/model_selection/_split.py:805: UserWarni
  warnings.warn(
{'fit_time': array([0.07122374, 0.04113245, 0.04136992, 0.04554749, 0.04071641,
       0.04074407, 0.03916645, 0.04146647, 0.04380536, 0.03941584]), 'score_time': array
       0.00607634, 0.00570369, 0.00657535, 0.00589514, 0.00542974]), 'test_score': array
       0.71428571, 0.64285714, 0.64285714, 0.5       , 0.42857143])}
```

Choosing Parameters Using a Grid Search

A common way to tune models is to use a grid search through a large set of possible parameters. Here we try depths between 3 and 20 and different limits on the minimum number of samples per split.

```
# Set up the parameter grid to seaerch
param_grid ={'criterion': ['gini', "entropy"], \
             'max_depth': list(range(3, 50, 3)), \
             'min_samples_split': [50]}

# Perform the search
my_tuned_tree = sklearn.model_selection.GridSearchCV(sklearn.tree.DecisionTreeClassifier(),
                                param_grid, cv=cv_folds, verbose = 2, \
                            return_train_score=True, n_jobs = -1)
my_tuned_tree.fit(X, y)

# Print details
print("Best parameters set found on development set:")
display(my_tuned_tree.best_params_)
model_tuned_params_list["Tuned Tree"] = my_tuned_tree.best_params_
display(my_tuned_tree.best_score_)
display(my_tuned_tree.cv_results_)
```

```
Fitting 2 folds for each of 32 candidates, totalling 64 fits
Best parameters set found on development set:
{'criterion': 'gini', 'max_depth': 3, 'min_samples_split': 50}
0.35452716297786724
{'mean_fit_time': array([0.04798818, 0.05045629, 0.04376793, 0.04668736, 0.0475173 ,
        0.04551554, 0.04829848, 0.04145253, 0.04432344, 0.04879487,
        0.04097855, 0.04781365, 0.04786432, 0.03930247, 0.04199505,
        0.04452264, 0.0363251 , 0.02810717, 0.02992165, 0.03363037,
        0.02486312, 0.02636003, 0.02423823, 0.02854407, 0.0245477 ,
        0.02528584, 0.0251534 , 0.02528024, 0.02512252, 0.02452993,
        0.02456677, 0.02328658]),
 'std_fit_time': array([1.16479397e-02, 8.76212120e-03, 7.47084618e-03, 8.94951820e-03,
        1.18651390e-02, 2.01845169e-03, 7.25591183e-03, 7.97426701e-03,
        6.89363480e-03, 5.95581532e-03, 9.46724415e-03, 7.50327110e-03,
        8.04769993e-03, 1.27133131e-02, 5.21922112e-03, 6.53541088e-03,
        7.70843029e-03, 2.49505043e-03, 4.93133068e-03, 8.91447067e-03,
        1.16944313e-04, 1.37305260e-03, 1.91330910e-04, 1.36911869e-03,
        1.77979469e-04, 2.40683556e-04, 7.91788101e-04, 1.04284286e-03,
        1.29580498e-04, 1.66893005e-05, 1.71542168e-04, 1.49941444e-03]),
 'mean_score_time': array([0.02224922, 0.01976788, 0.02185285, 0.02394021, 0.01997256,
        0.01729119, 0.0163126 , 0.02226448, 0.01941442, 0.0162549 ,
        0.02193296, 0.0192951 , 0.01589501, 0.01735258, 0.01783681,
        0.01569903, 0.0208981 , 0.01268542, 0.01109993, 0.00918198,
        0.00916719, 0.01092494, 0.01341748, 0.00908244, 0.01166809,
        0.00959635, 0.01071811, 0.00926375, 0.00919402, 0.00883913,
        0.00904465, 0.00749695]),
 'std_score_time': array([5.10478020e-03, 9.40442085e-04, 2.44021416e-04, 7.94756413e-
03,
        3.52025032e-03, 8.79406929e-04, 9.01222229e-05, 1.20711327e-03,
        3.13758850e-03, 7.62939453e-06, 2.19225883e-04, 3.95596027e-03,
        6.35981560e-04, 2.70748138e-03, 1.44267082e-03, 5.63037395e-03,
        1.36590004e-03, 3.35657597e-03, 2.24006176e-03, 2.74181366e-05,
        1.59263611e-04, 2.11513042e-03, 4.32062149e-03, 8.97645950e-05,
        2.90191174e-03, 3.49521637e-04, 1.63125992e-03, 5.98430634e-05,
        2.56419182e-04, 2.37226486e-04, 2.20775604e-04, 1.77514553e-03]),
 'param_criterion': masked_array(data=['gini', 'gini', 'gini', 'gini', 'gini', 'gini',
'gini',
                    'gini', 'gini', 'gini', 'gini', 'gini', 'gini', 'gini',
                    'gini', 'gini', 'entropy', 'entropy', 'entropy',
                    'entropy', 'entropy', 'entropy', 'entropy', 'entropy',
                    'entropy', 'entropy', 'entropy', 'entropy', 'entropy',
                    'entropy', 'entropy', 'entropy'],
              mask=[False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False],
        fill_value='?',
             dtype=object),
 'param_max_depth': masked_array(data=[3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39,
42,
                    45, 48, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36,
                    39, 42, 45, 48],
              mask=[False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False],
```

```
        fill_value=999999),
 'param_min_samples_split': masked_array(data=[50, 50, 50, 50, 50, 50, 50, 50, 50, 50,
50, 50, 50, 50,
                    50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50,
                    50, 50, 50, 50],
              mask=[False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False],
        fill_value=999999),
 'params': [{'criterion': 'gini', 'max_depth': 3, 'min_samples_split': 50},
  {'criterion': 'gini', 'max_depth': 6, 'min_samples_split': 50},
  {'criterion': 'gini', 'max_depth': 9, 'min_samples_split': 50},
  {'criterion': 'gini', 'max_depth': 12, 'min_samples_split': 50},
  {'criterion': 'gini', 'max_depth': 15, 'min_samples_split': 50},
  {'criterion': 'gini', 'max_depth': 18, 'min_samples_split': 50},
  {'criterion': 'gini', 'max_depth': 21, 'min_samples_split': 50},
  {'criterion': 'gini', 'max_depth': 24, 'min_samples_split': 50},
  {'criterion': 'gini', 'max_depth': 27, 'min_samples_split': 50},
  {'criterion': 'gini', 'max_depth': 30, 'min_samples_split': 50},
  {'criterion': 'gini', 'max_depth': 33, 'min_samples_split': 50},
  {'criterion': 'gini', 'max_depth': 36, 'min_samples_split': 50},
  {'criterion': 'gini', 'max_depth': 39, 'min_samples_split': 50},
  {'criterion': 'gini', 'max_depth': 42, 'min_samples_split': 50},
  {'criterion': 'gini', 'max_depth': 45, 'min_samples_split': 50},
  {'criterion': 'gini', 'max_depth': 48, 'min_samples_split': 50},
  {'criterion': 'entropy', 'max_depth': 3, 'min_samples_split': 50},
  {'criterion': 'entropy', 'max_depth': 6, 'min_samples_split': 50},
  {'criterion': 'entropy', 'max_depth': 9, 'min_samples_split': 50},
  {'criterion': 'entropy', 'max_depth': 12, 'min_samples_split': 50},
  {'criterion': 'entropy', 'max_depth': 15, 'min_samples_split': 50},
  {'criterion': 'entropy', 'max_depth': 18, 'min_samples_split': 50},
  {'criterion': 'entropy', 'max_depth': 21, 'min_samples_split': 50},
  {'criterion': 'entropy', 'max_depth': 24, 'min_samples_split': 50},
  {'criterion': 'entropy', 'max_depth': 27, 'min_samples_split': 50},
  {'criterion': 'entropy', 'max_depth': 30, 'min_samples_split': 50},
  {'criterion': 'entropy', 'max_depth': 33, 'min_samples_split': 50},
  {'criterion': 'entropy', 'max_depth': 36, 'min_samples_split': 50},
  {'criterion': 'entropy', 'max_depth': 39, 'min_samples_split': 50},
  {'criterion': 'entropy', 'max_depth': 42, 'min_samples_split': 50},
  {'criterion': 'entropy', 'max_depth': 45, 'min_samples_split': 50},
  {'criterion': 'entropy', 'max_depth': 48, 'min_samples_split': 50}],
 'split0_test_score': array([0.36619718, 0.36619718, 0.36619718, 0.36619718,
0.36619718,
        0.36619718, 0.36619718, 0.36619718, 0.36619718, 0.36619718,
        0.36619718, 0.36619718, 0.36619718, 0.36619718, 0.36619718,
        0.36619718, 0.21126761, 0.21126761, 0.21126761, 0.21126761,
        0.21126761, 0.21126761, 0.21126761, 0.21126761, 0.21126761,
        0.21126761, 0.21126761, 0.21126761, 0.21126761, 0.21126761,
        0.21126761, 0.21126761]),
 'split1_test_score': array([0.34285714, 0.34285714, 0.34285714, 0.34285714,
0.34285714,
        0.34285714, 0.34285714, 0.34285714, 0.34285714, 0.34285714,
        0.34285714, 0.34285714, 0.34285714, 0.34285714, 0.32857143,
        0.32857143, 0.21428571, 0.21428571, 0.21428571, 0.21428571,
        0.21428571, 0.21428571, 0.21428571, 0.21428571, 0.21428571,
        0.21428571, 0.21428571, 0.21428571, 0.21428571, 0.21428571,
```

```
       0.21428571, 0.21428571, 0.21428571, 0.21428571, 0.21428571,
       0.21428571, 0.21428571]),
 'mean_test_score': array([0.35452716, 0.35452716, 0.35452716, 0.35452716, 0.35452716,
       0.35452716, 0.35452716, 0.35452716, 0.35452716, 0.35452716,
       0.35452716, 0.35452716, 0.35452716, 0.35452716, 0.34738431,
       0.34738431, 0.21277666, 0.21277666, 0.21277666, 0.21277666,
       0.21277666, 0.21277666, 0.21277666, 0.21277666, 0.21277666,
       0.21277666, 0.21277666, 0.21277666, 0.21277666, 0.21277666,
       0.21277666, 0.21277666]),
 'std_test_score': array([0.01167002, 0.01167002, 0.01167002, 0.01167002, 0.01167002,
       0.01167002, 0.01167002, 0.01167002, 0.01167002, 0.01167002,
       0.01167002, 0.01167002, 0.01167002, 0.01167002, 0.01881288,
       0.01881288, 0.00150905, 0.00150905, 0.00150905, 0.00150905,
       0.00150905, 0.00150905, 0.00150905, 0.00150905, 0.00150905,
       0.00150905, 0.00150905, 0.00150905, 0.00150905, 0.00150905,
       0.00150905, 0.00150905]),
 'rank_test_score': array([ 1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1, 15,
15, 17,
       17, 17, 17, 17, 17, 17, 17, 17, 17, 17, 17, 17, 17, 17, 17],
      dtype=int32),
 'split0_train_score': array([0.45714286, 0.45714286, 0.45714286, 0.45714286,
0.45714286,
       0.45714286, 0.45714286, 0.45714286, 0.45714286, 0.45714286,
       0.45714286, 0.45714286, 0.45714286, 0.45714286, 0.45714286,
       0.45714286, 0.25714286, 0.25714286, 0.25714286, 0.25714286,
       0.25714286, 0.25714286, 0.25714286, 0.25714286, 0.25714286,
       0.25714286, 0.25714286, 0.25714286, 0.25714286, 0.25714286,
       0.25714286, 0.25714286]),
 'split1_train_score': array([0.36619718, 0.36619718, 0.36619718, 0.36619718,
0.36619718,
       0.36619718, 0.36619718, 0.36619718, 0.36619718, 0.36619718,
       0.36619718, 0.36619718, 0.36619718, 0.36619718, 0.36619718,
       0.36619718, 0.23943662, 0.23943662, 0.23943662, 0.23943662,
       0.23943662, 0.23943662, 0.23943662, 0.23943662, 0.23943662,
       0.23943662, 0.23943662, 0.23943662, 0.23943662, 0.23943662,
       0.23943662, 0.23943662]),
 'mean_train_score': array([0.41167002, 0.41167002, 0.41167002, 0.41167002, 0.41167002,
       0.41167002, 0.41167002, 0.41167002, 0.41167002, 0.41167002,
       0.41167002, 0.41167002, 0.41167002, 0.41167002, 0.41167002,
       0.41167002, 0.24828974, 0.24828974, 0.24828974, 0.24828974,
       0.24828974, 0.24828974, 0.24828974, 0.24828974, 0.24828974,
       0.24828974, 0.24828974, 0.24828974, 0.24828974, 0.24828974,
       0.24828974, 0.24828974]),
 'std_train_score': array([0.04547284, 0.04547284, 0.04547284, 0.04547284, 0.04547284,
       0.04547284, 0.04547284, 0.04547284, 0.04547284, 0.04547284,
       0.04547284, 0.04547284, 0.04547284, 0.04547284, 0.04547284,
       0.04547284, 0.00885312, 0.00885312, 0.00885312, 0.00885312,
       0.00885312, 0.00885312, 0.00885312, 0.00885312, 0.00885312,
       0.00885312, 0.00885312, 0.00885312, 0.00885312, 0.00885312,
       0.00885312, 0.00885312]),
```

Comparing Models

We can easily use the same patterns to train other types of models.

Random Forests

Train and evaluate a simple model

```
# Do the same job with random forests
my_model = sklearn.ensemble.RandomForestClassifier(n_estimators=300, \
                                           max_features = 3,\
                                           min_samples_split=200)
my_model.fit(X_train,y_train)
```

```
RandomForestClassifier(max_features=3, min_samples_split=200, n_estimators=300)
```

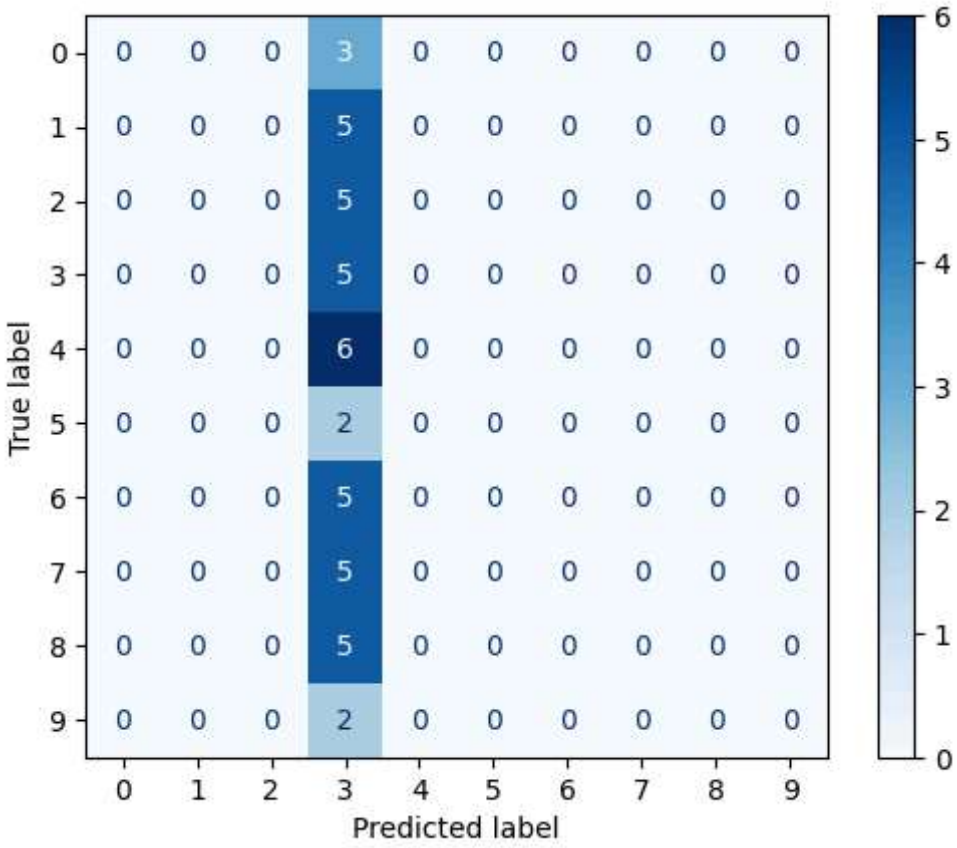
```
# Make a set of predictions for the test data
y_pred = my_model.predict(X_valid)

# Print performance details
accuracy = sklearn.metrics.accuracy_score(y_valid, y_pred) # , normalize=True, sample_weight
model_valid_accuracy_comparisons["Random Forest"] = accuracy
print("Accuracy: " +  str(accuracy))
print(sklearn.metrics.classification_report(y_valid, y_pred))

# Print confusion matrix
print("Confusion Matrix")
sklearn.metrics.ConfusionMatrixDisplay.from_predictions(y_valid, y_pred, cmap = 'Blues')
plt.show()
```

Accuracy: 0.11627906976744186

	precision	recall	f1-score	support
0	0.00	0.00	0.00	3
1	0.00	0.00	0.00	5
2	0.00	0.00	0.00	5
3	0.12	1.00	0.21	5
4	0.00	0.00	0.00	6
5	0.00	0.00	0.00	2
6	0.00	0.00	0.00	5
7	0.00	0.00	0.00	5
8	0.00	0.00	0.00	5
9	0.00	0.00	0.00	2
accuracy			0.12	43
macro avg	0.01	0.10	0.02	43
weighted avg	0.01	0.12	0.02	43

Confusion Matrix

```
/usr/local/lib/python3.11/dist-packages/sklearn/metrics/_classification.py:1565: Undefir
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/lib/python3.11/dist-packages/sklearn/metrics/_classification.py:1565: Undefir
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/lib/python3.11/dist-packages/sklearn/metrics/_classification.py:1565: Undefir
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
```



Choose parameters using a grid search

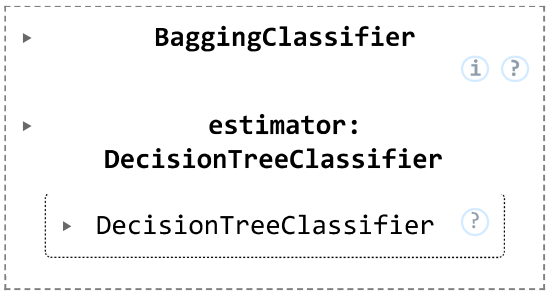
```
# Set up the parameter grid to seaerch
param_grid = [
 {'n_estimators': list(range(100, 501, 50)), 'max_features': list(range(2, 10, 2)), 'min_sam
]

# Perform the search
my_tuned_model = sklearn.model_selection.GridSearchCV(sklearn.ensemble.RandomForestClassifie
my_tuned_model.fit(X, y)

# Print details
print("Best parameters set found on development set:")
print(my_tuned_model.best_params_)
model_tuned_params_list["Tuned Random Forest"] = my_tuned_model.best_params_
print(my_tuned_model.best_score_)
model_accuracy_comparisons["Tuned Random Forest"] = my_tuned_model.best_score_
```

```
Fitting 2 folds for each of 36 candidates, totalling 72 fits
Best parameters set found on development set:
{'max_features': 2, 'min_samples_split': 200, 'n_estimators': 250}
0.13480885311871227
```

Bagging

Train and evaluate a simple model

```
# Do the same job with random forests
my_model = sklearn.ensemble.BaggingClassifier(estimator = sklearn.tree.DecisionTreeClassifie
                                              n_estimators=10)
my_model.fit(X_train,y_train)
```




```
# Make a set of predictions for the validation data
y_pred = my_model.predict(X_valid)

# Print performance details
accuracy = sklearn.metrics.accuracy_score(y_valid, y_pred) # , normalize=True, sample_weight
model_valid_accuracy_comparisons["Bagging"] = accuracy
```

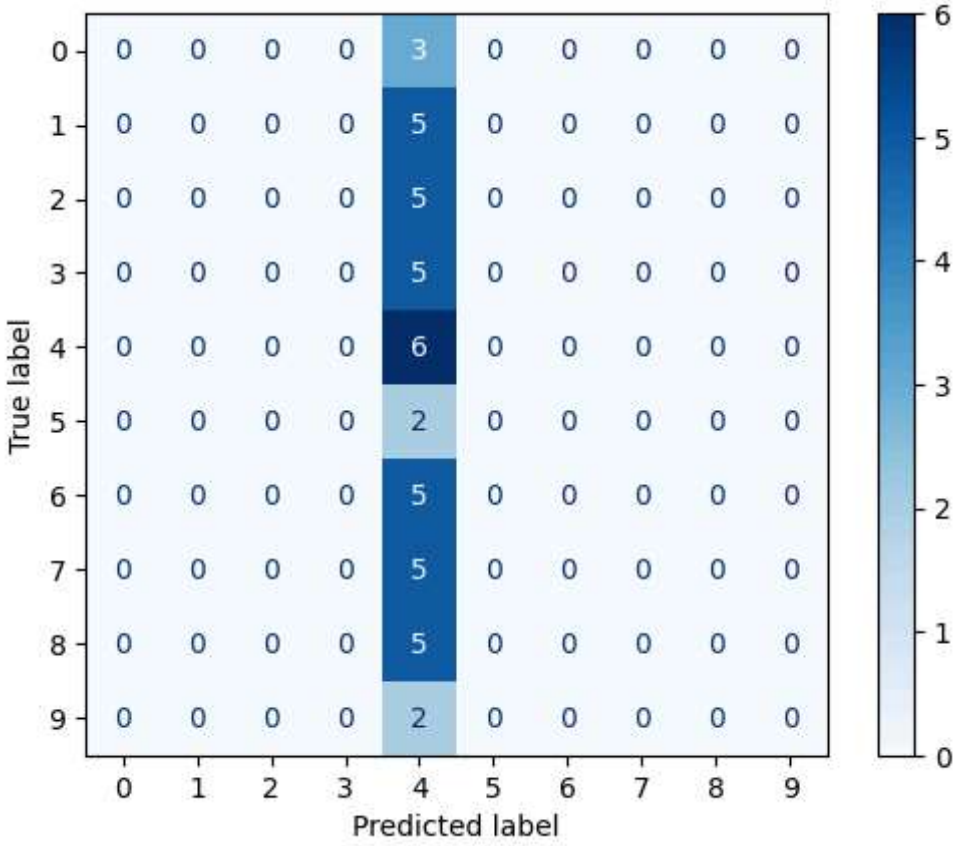
```
print("Accuracy: " +  str(accuracy))
print(sklearn.metrics.classification_report(y_valid, y_pred))

# Print confusion matrix
print("Confusion Matrix")
sklearn.metrics.ConfusionMatrixDisplay.from_predictions(y_valid, y_pred, cmap = 'Blues')
plt.show()
```

Accuracy: 0.13953488372093023

	precision	recall	f1-score	support
0	0.00	0.00	0.00	3
1	0.00	0.00	0.00	5
2	0.00	0.00	0.00	5
3	0.00	0.00	0.00	5
4	0.14	1.00	0.24	6
5	0.00	0.00	0.00	2
6	0.00	0.00	0.00	5
7	0.00	0.00	0.00	5
8	0.00	0.00	0.00	5
9	0.00	0.00	0.00	2
accuracy			0.14	43
macro avg	0.01	0.10	0.02	43
weighted avg	0.02	0.14	0.03	43

Confusion Matrix

```
/usr/local/lib/python3.11/dist-packages/sklearn/metrics/_classification.py:1565: Undefir
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/lib/python3.11/dist-packages/sklearn/metrics/_classification.py:1565: Undefir
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/lib/python3.11/dist-packages/sklearn/metrics/_classification.py:1565: Undefir
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
```



Choose parameters using a grid search

```
# Set up the parameter grid to seaerch
param_grid = [
 {'n_estimators': list(range(50, 501, 50))}
]

# Perform the search
my_tuned_model = sklearn.model_selection.GridSearchCV(sklearn.ensemble.BaggingClassifier(est
my_tuned_model.fit(X, y)

# Print details
print("Best parameters set found on development set:")
print(my_tuned_model.best_params_)
model_tuned_params_list["Tuned Bagging"] = my_tuned_model.best_params_
print(my_tuned_model.best_score_)
model_accuracy_comparisons["Tuned Bagging"] = my_tuned_model.best_score_
```

```
Fitting 2 folds for each of 10 candidates, totalling 20 fits
Best parameters set found on development set:
{'n_estimators': 50}
0.13480885311871227
```

Gradient Boosting

Train and evaluate a simple model

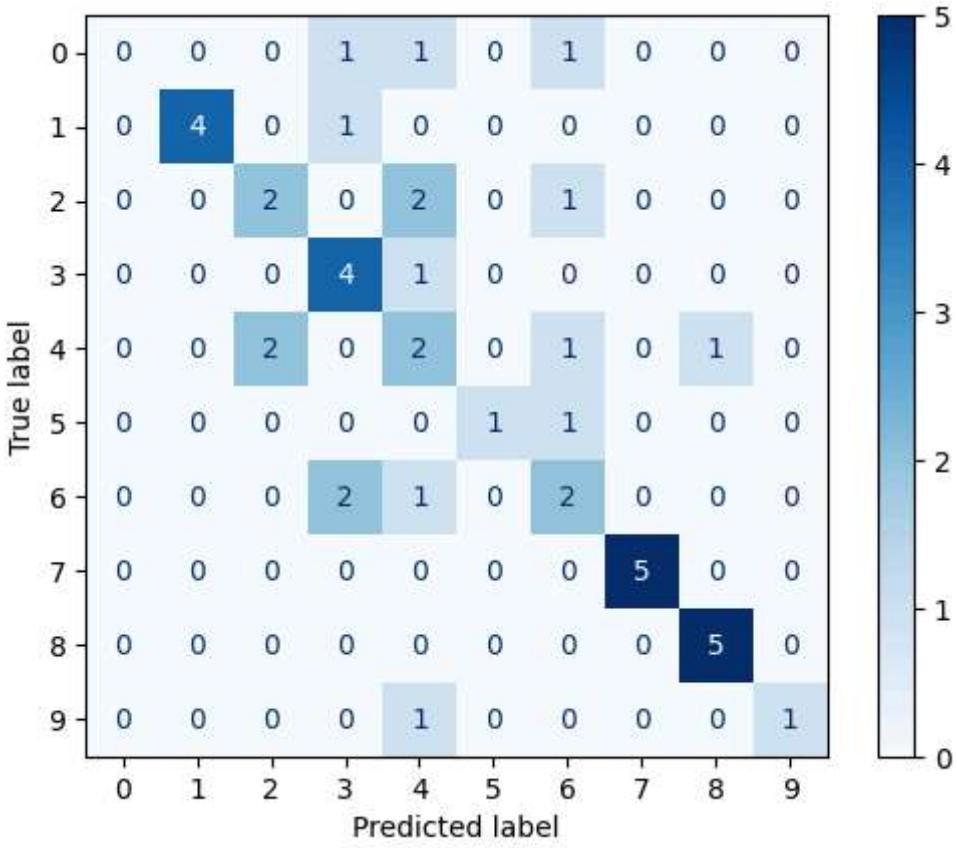
```
# Do the same job with random forests
my_model = sklearn.ensemble.GradientBoostingClassifier()
my_model.fit(X_train,y_train)
```

```
GradientBoostingClassifier
GradientBoostingClassifier()
```

```
# Make a set of predictions for the validation data
y_pred = my_model.predict(X_valid)

# Print performance details
accuracy = sklearn.metrics.accuracy_score(y_valid, y_pred) # , normalize=True, sample_weight
model_valid_accuracy_comparisons["GradBoost"] = accuracy
print("Accuracy: " +  str(accuracy))
print(sklearn.metrics.classification_report(y_valid, y_pred))

# Print confusion matrix
print("Confusion Matrix")
```

```
sklearn.metrics.ConfusionMatrixDisplay.from_predictions(y_valid, y_pred, cmap = 'Blues')
plt.show()
```

```
Accuracy: 0.6046511627906976
              precision    recall  f1-score   support

           0       0.00      0.00      0.00         3
           1       1.00      0.80      0.89         5
           2       0.50      0.40      0.44         5
           3       0.50      0.80      0.62         5
           4       0.25      0.33      0.29         6
           5       1.00      0.50      0.67         2
           6       0.33      0.40      0.36         5
           7       1.00      1.00      1.00         5
           8       0.83      1.00      0.91         5
           9       1.00      0.50      0.67         2

    accuracy                           0.60        43
   macro avg       0.64      0.57      0.58        43
weighted avg       0.61      0.60      0.59        43

Confusion Matrix
/usr/local/lib/python3.11/dist-packages/sklearn/metrics/_classification.py:1565: Undefin
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/lib/python3.11/dist-packages/sklearn/metrics/_classification.py:1565: Undefin
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/lib/python3.11/dist-packages/sklearn/metrics/_classification.py:1565: Undefin
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
```



Choose parameters using a grid search

```
# Set up the parameter grid to seaerch
param_grid = [
 {'n_estimators': list(range(50, 501, 50)),
 'learning_rate': [0.001, 0.01, 0.1]}
]

# Perform the search
my_tuned_model = sklearn.model_selection.GridSearchCV(sklearn.ensemble.GradientBoostingClass
my_tuned_model.fit(X, y)

# Print details
print("Best parameters set found on development set:")
print(my_tuned_model.best_params_)
model_tuned_params_list["Tuned GradBoost"] = my_tuned_model.best_params_
print(my_tuned_model.best_score_)
model_accuracy_comparisons["Tuned GradBoost"] = my_tuned_model.best_score_
```

```
Fitting 2 folds for each of 30 candidates, totalling 60 fits
Best parameters set found on development set:
{'learning_rate': 0.1, 'n_estimators': 200}
0.510764587525151
```

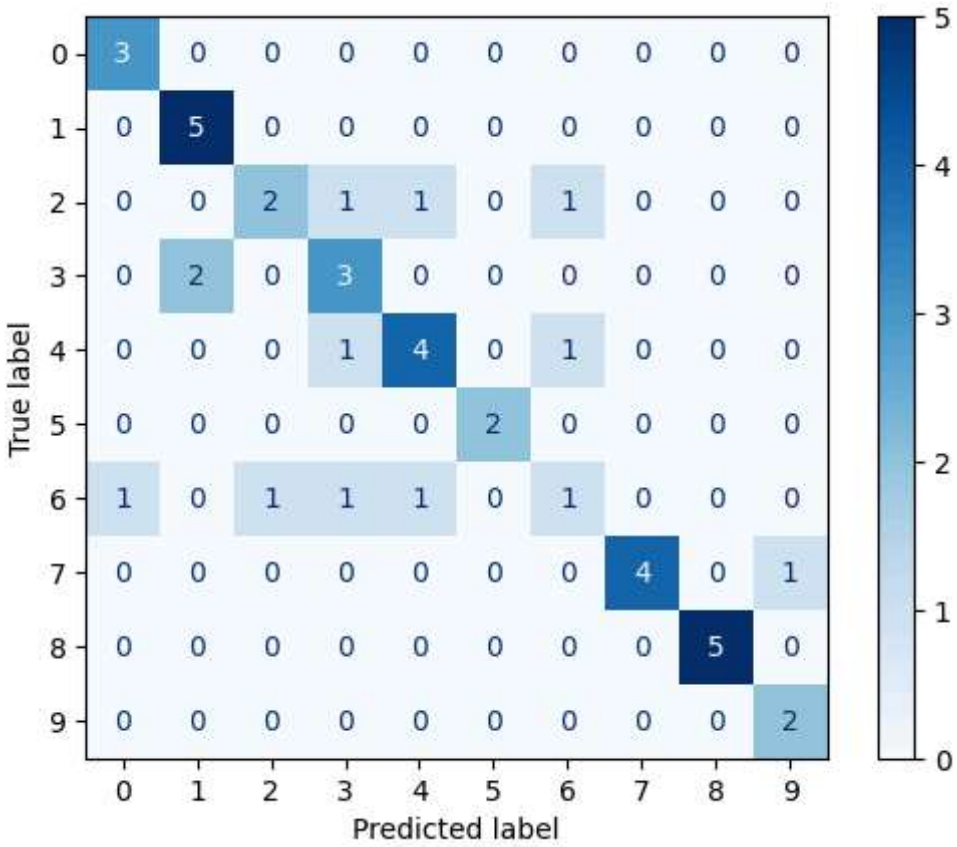
Nearest Neighbour

Train and evaluate a simple model

```
# Do the same job with random forests
my_model = sklearn.neighbors.KNeighborsClassifier()
my_model = my_model.fit(X_train,y_train)


# Make a set of predictions for the test data
y_pred = my_model.predict(X_valid)

# Print performance details
accuracy = sklearn.metrics.accuracy_score(y_valid, y_pred) # , normalize=True, sample_weight
model_valid_accuracy_comparisons["kNN"] = accuracy
print("Accuracy: " +  str(accuracy))
print(sklearn.metrics.classification_report(y_valid, y_pred))

# Print confusion matrix
print("Confusion Matrix")
sklearn.metrics.ConfusionMatrixDisplay.from_predictions(y_valid, y_pred, cmap = 'Blues')
plt.show()
```

```
Accuracy: 0.7209302325581395
              precision    recall  f1-score   support

           0       0.75      1.00      0.86         3
           1       0.71      1.00      0.83         5
           2       0.67      0.40      0.50         5
           3       0.50      0.60      0.55         5
           4       0.67      0.67      0.67         6
           5       1.00      1.00      1.00         2
           6       0.33      0.20      0.25         5
           7       1.00      0.80      0.89         5
           8       1.00      1.00      1.00         5
           9       0.67      1.00      0.80         2

    accuracy                           0.72        43
   macro avg       0.73      0.77      0.73        43
weighted avg       0.71      0.72      0.70        43
```

Confusion Matrix



Choose parameters using a grid search

```
# Set up the parameter grid to seaerch
param_grid = [
              {'n_neighbors': list(range(1, 50, 5))}
]
```