Google Colab Specific Setup

For getting files on Googl Drive and to Open in Colab

First download the files from the lab and uploade them to google drive near the top level. Then go to this folder in your browser and click on column of dots to the right of the *.ipynb file. Select "Open With". If "Google Collaboratory" appears choose it. If it does not appear go to the bottom of the list and choose "Connect more apps" and then install "Google Collaboratory". Close browser, reopen, and try again. Now "Google Collaboratory" should appear when you select "Open With". You should also just be able to double click on the* .ipynb file.

This cell links your Google drive to the Colab session

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

Here, modify the path where you saved this notebook and the associated lab files on your Google drive, if different from below (this is an example Dane Morgan had for class). This can be obtained in colab

```
path = r'C:/Users/jhyang/OneDrive/文档/GitHub_Projects/MSE_760/Lab4-
Assignment'
```

Check the path name is assigned correctly.

```
!echo $path
import os
os.path.isdir(path)
```

```
C:/Users/jhyang/OneDrive/ĵ/GitHub_Projects/MSE_760/Lab4-Assignment
```

```
True
```

```
# This is needed so can import packages from a different path than
standard libraries
import sys
sys.path.append(path)
```

Need to install pymatgen, a python package for materials analysis which is not present on Colab environement by default.

```
# %%capture
#!pip install pymatgen==2020.12.31
#!pip install pymatgen
```

```
# Import some resources from pymatgen and check they worked
from pymatgen.core.composition import Composition
print(Composition('Fe2O3'))
```

```
Fe2 O3
```

With that your colab environment should be compatible with all of the code and lab activities below.

As a note the look and feel of this notebook may be slightly off from some text descriptions (noteably descriptions of "highlight boxes", which are not functioning in colab) this will not change functionality only the looks of some sections. Additionaly some keyboard shortcuts described will be different in colab notebooks. There are also some descriptions of a cloud computing resource "Nanohub", when running on colab we can skip those specifics as they do not apply here.

```
import os                          # OS stands for Operating System and
provides ways for python to interact with files or directories
from collections import Counter  # Collections is a package for
handling data
from pprint import pprint

import pandas as pd                # Pandas is a data analysis library
which we'll primarily use to handle our dataset
import numpy as np                 # Numpy is a package for scientific
computing. We'll use it for some of it's math functions
import pymatgen                    # Pymatgen is a library for materials
analysis which we use to interpret our material compositions

import matplotlib                  # Matplotlib is the plotting package
that we'll use throughout the lab
import matplotlib.pyplot as plt
import seaborn as sns              # Seaborn is a Python data
visualization library based on matplotlib

import sklearn                     # Scikit-learn is a machine learning
package, providing the backbone for the work we'll perform
from sklearn import metrics
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import
cross_validate,GridSearchCV,ParameterGrid
from sklearn.model_selection import KFold,RepeatedKFold

#!pip install graphviz
import graphviz                    # graphviz is a package that helps
visualize decision trees
```

```
#file = path+'helper_functions'
#file
#import file
from helper_functions import *
```

There are a number of steps we'll take which would normally have a random state. In order to have consistent results we'll fix them all by setting a random seed for all those processes.

```
seed = 2345312
```

Content/Exercises/Lessons

```
# Read in the band gap data from our dataset
mastml_df = pd.read_csv(os.path.join(path,"bandgap_data_v2.csv"))

mastml_df
```

```
      index chemicalFormula Clean  Band gap values Clean Band gap
units  \
0         0                 Li1F1                  13.60
eV
1         1                 Li1F1                  12.61
eV
2         2                 Li1F1                  12.60
eV
3         3                 Li1F1                  12.10
eV
4         4                 Li1F1                  12.00
eV
...     ...                   ...                    ...             ..
.
1442   1454                 Th1O2                   3.30
eV
1443   1455                   U0                    1.50
eV
1444   1456                 U1O2                    2.18
eV
1445   1457                   U0                    0.60
eV
1446   1458                 U1O2                    1.30
eV

         Band gap method  Reliability
0             Reflection            1
1             Reflection            1
2              Estimated            2
3             Absorption            2
4             Absorption            2
...                  ...          ...
```

```
1442          Reflection            2
1443  Thermal activation            1
1444          Absorption            1
1445  Thermal activation            2
1446  Thermal activation            2

[1447 rows x 6 columns]
```

Before we dig into too much detail, lets take a second to understand what is included in the dataset column by column:

1) Index
When dealing with large datasets having an explicit index is essential for keeping track of data points. Throughout the lab we'll be making changes to the dataset, and without proper indexing it's easy to make mistakes and lose track of where data came from. By Specifying a unique number to each datapoint we can always track things down to troubleshoot, make later changes, or track where something came from.

2) chemicalFormula Clean
This is the key input parameter for all of the models you'll build. Fundamentally all of the information that the model contains can be represented by the chemical formulas in this column. Take a second to think about how powerful it would be to have a model that only has these simple letters and numbers as input. With an accurate model it would be possible to think of any composition of interest and obtain an almost immediate prediction.

3) Band gap values Clean, Band gap units
Carrying on from the previous thought we have to ask ourselves "what is it that we're predicting?". In this case we have a dataset of band gap values for semiconductors and insulators. Knowledge of a material's bandgap is essential for a whole range of semiconductor applications. If we could predict a new material's band gap we could potentially accelerate discovery and design of materials, contributing to what is already more than a 400 billion dollar industry!

4) Band gap method
We won't dive too deeply into the method information here directly, but notice that in the dataset we have a few different experimental measurement types. This is often the case when putting together large datasets that not all data is exactly equal. The accuracy of a model is often limited by the quality of data availabe so it's always important to understand where are the data comes from, and if it can be combined.

5) Reliability
As a simpler version of the idea above about data quality we have a column labeled "Reliability". The researcher who put the dataset together took time to check each of their sources and come up with a reliability score or 1 or 2. A score of 1 indicates the most reliable data, and a 2 indicates that the samples may have been less pure, or the experimental technique was less accurate. As part of the data cleaning process later on we'll only use the most reliable data we have.

Answer

In the mastml_df["chemicalFormula Clean"], the same chemical formula shows up multiple times. If the same chemical formula appears much more frequently than other formulas, it can

lead to an imbalance in the dataset's class distribution. In certain machine learning tasks, class imbalance can impact the model's performance, causing it to be biased towards the more frequently occurring class. If duplicate data represents errors or redundant information in the dataset, it may be necessary to perform data cleaning to ensure data accuracy and consistency. This is also the reason why the groupby and mean functions are used down below.

```
# Filter for only Reliability 1
mastml_df_filtered = mastml_df[mastml_df["Reliability"]==1]

# Print filtered data
mastml_df_filtered.head(10)
```

```
    index chemicalFormula Clean  Band gap values Clean Band gap units
\
0       0                  Li1F1                  13.60             eV

1       1                  Li1F1                  12.61             eV

6       6                 Li1Cl1                   9.33             eV

7       7                 Li1Br1                   7.95             eV

9       9                 Li3Sb1                   1.00             eV

10     10                  Li1I1                   6.00             eV

15     15                 Li3Bi1                   0.70             eV

16     16                  Be1O1                  10.39             eV

17     17                  Be1O1                  10.57             eV

22     22                  Be1S1                   4.17             eV


       Band gap method  Reliability
0           Reflection            1
1           Reflection            1
6           Reflection            1
7           Absorption            1
9   Thermal activation            1
10          Reflection            1
15  Thermal activation            1
16          Reflection            1
17          Reflection            1
22               SCOPW            1
```

Looking through the filtered data and paying attention to the chemical formula column there are still some formulas for which we have multiple measurements. Because we don't have another way to decide which data points to keep, let's average the values between these multiple measurements.

To do this we'll use a method in Pandas (the dataframe package we are using to handle the data) called groupby which allows us to create groups of all of the identical formulas, and then average within each group.

```
# mastml_df_clean = mastml_df_filtered.groupby("chemicalFormula
Clean", as_index = False).mean()
# mastml_df_clean

# Exclude non-numeric columns for aggregation
numeric_columns = mastml_df_filtered.select_dtypes(include='number')

# Include the grouping column
columns_to_group = ["chemicalFormula Clean"] +
list(numeric_columns.columns)

# Perform the aggregation
mastml_df_clean =
mastml_df_filtered[columns_to_group].groupby("chemicalFormula Clean",
as_index=False).mean()

mastml_df_clean.head(5)
```

	chemicalFormula Clean	index	Band gap values Clean	Reliability
0	Ag1Br1	808.5	3.485	1.0
1	Ag1Cl1	793.5	4.190	1.0
2	Ag1N3	783.0	3.900	1.0
3	Ag1Te1	820.0	0.850	1.0
4	Ag2O1	785.0	1.200	1.0

Answer

```
# Look at the starting dataframe mastml_df, How Many data points did
we start with?
print("There are", len(mastml_df), "data points in the mastml_df.")

# look at the cleaned dataframe mastml_df_clean, how many data points
do we have now?
print("There are", len(mastml_df_clean), "data points in the
mastml_df_clean.")
```

```
There are 1447 data points in the mastml_df.
There are 467 data points in the mastml_df_clean.
```

```
# generate basic statistics on our band gap values
mastml_df_clean["Band gap values Clean"].describe().round(3)
```

```
count    467.000
mean       2.231
std        2.287
min        0.009
```

```
25%         0.695
50%         1.435
75%         3.000
max        13.105
Name: Band gap values Clean, dtype: float64
```

Answer

1. What is the range of band gap values?

   range = max - min = 13.105 - 0.009 = 13.096

2. Would a predicted error of 5 eV be considered small enough to be an accurate or useful prediction?

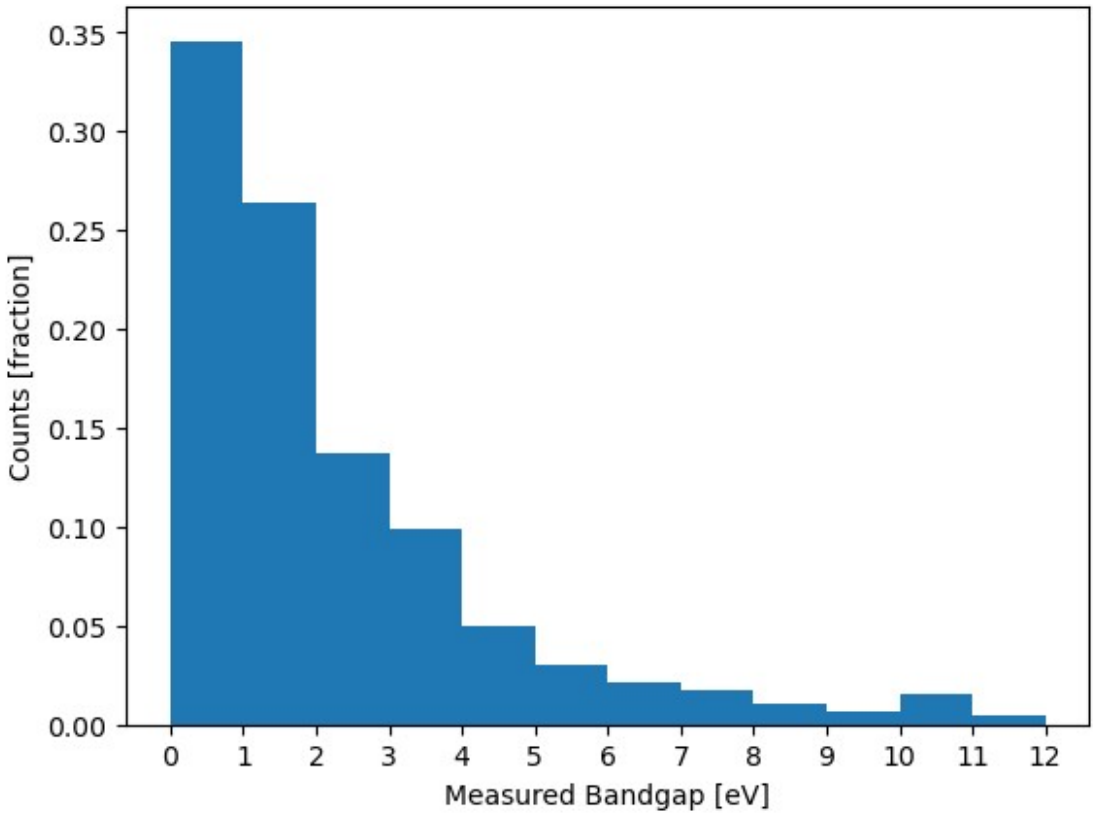
   The dataset's mean band gap is approximately 2.231 eV, with a standard deviation of 2.287 eV, indicating considerable dispersion. A 5 eV error exceeds twice the mean and represents a significant portion of the standard deviation, highlighting its inaccuracy.

3. How about a predicted error of 0.5 eV?

   Considering the point I made in question 2, a predicted error of 0.5 eV would be considered smaller and potentially more useful than a 5 eV error.

Apart from just the ranges of values it is also useful to visualize the distribution of data.
Let's build a simple histogram of the band gap values.

```
# we'll also define a simple histogram plotting function to use later
def histogram_plot(data):
    fig1,ax1 = plt.subplots()
    ax1.hist(data,bins=range(13),density=1)
    ax1.set_xticks(range(13))
    ax1.set_xlabel('Measured Bandgap [eV]')
    ax1.set_ylabel('Counts [fraction]')
    plt.show()

histogram_plot(mastml_df_clean["Band gap values
Clean"].astype("float"))
```



Answer

1. Is our band gap data balanced (i.e. uniformly distributed across its range)?

   According to the histogram polt, we have more data in 0–4 than 5–12. It appears to be right-skewed or positively skewed, meaning that there are more data points at lower band gap values, and the frequency of data points decreases as band gap values increase. Thus, it is not uniformly distributed across its range and our band gap data is not balanced.

2. Would you expect that the model has similar performance between 0-2 eV as between 10-12 eV?

   Given the imbalance in the band gap data distribution, we would not expect the model to have similar performance between the 0-2 eV range and the 10-12 eV range. Since there are more data points in the lower band gap range (0-2 eV), the model would likely have more data to learn from in that range and may perform better there. Conversely, in the 10-12 eV range, where there are fewer data points, the model may have limited examples to learn from, potentially leading to lower performance in that range.

Let's also try to get a feel for the compositions present in our dataset.
Specifically we'll focus on looking at which elements are present in the data, and in what quantity.

```
# parse out individual elements for each formula using pymatgen's
composition parser
element_list = list()
for idx in mastml_df.index:

element_list.extend(pymatgen.core.composition.Composition(mastml_df["c
hemicalFormula Clean"][idx]).elements)

# setup a counter to count each element
temp_counter = Counter(element_list)
element_tuples =
list(zip(list(temp_counter.keys()),list(temp_counter.values())))
element_df = pd.DataFrame(element_tuples,columns=["Element","Count"])
element_df_sorted =
element_df.sort_values(by=["Count"],ascending=False)

element_df_sorted.head(10)
```

	Element	Count
8	O	240
10	Se	196
9	S	191
11	Te	187
15	As	141
4	Sb	129
5	I	96
36	Ga	81
22	Pb	75
47	Cd	69

```
element_df_sorted.tail(10)
```

	Element	Count
68	Re	4
58	Tb	4
44	Rh	4
75	U	4
60	Ho	3
66	Ta	3
39	Y	3
70	Ir	2
62	Tm	1
64	Lu	1

Answer

1. What are the five most common elements in the dataset?

   O, Se, S, Te, As

2. What are the five least common elements in the dataset?

Ta, Y, Ir, Tm, Lu

3. Rank your confidence in the following predictions:

   Scale 0–10, with 10 being very confident and 0 being no hope at all

   - predictions containing Oxygen (oxides): almost 10, very confident, because it is the most common element in the dataset, and its frequent presence suggests it plays a significant role in various compounds;
   - predictions containing Iridium: about 3—with some hope or moderate confidence, but not too much, because it is the third least common element, making predictions involving it less common but still plausible;
   - predictions containing an element that doesn't appear in the dataset at all: Lu, because it is the least common element. As elements not present in the dataset are virtually impossible to predict accurately, given no training data for reference.

```
# Output data to csv - note depending on when you run this the updated
data file may have been pregenerated so this cell isn't technically
necessary.
output_path = "./bandgap_data_v3.csv"

if os.path.isfile(output_path):
    print(output_path," exists, not creating new file")
else:
    mastml_df_clean.to_csv(output_path)
```

```
./bandgap_data_v3.csv  exists, not creating new file
```

```
# get a new dataframe of generated features from the pregenerated
matml run.
generated_features_path = os.path.join(path,"generated_features.csv")
features_df = pd.read_csv(generated_features_path)
```

The raw MASTML output combines the original data and the generated features in one single dataframe, which isn't ideal. To make our next step (feature engineering) easier, We will split it into two dataframes:

1. `target_data_df`: target values (outputs)
2. `features_df`: features (inputs)

```
#split features_df into two dataframes
target_data_df = pd.DataFrame([features_df["chemicalFormula
Clean"],features_df["Band gap values Clean"],
features_df["Band gap
units"],features_df["index"],features_df["Reliability"]]).T
features_df = features_df.drop(columns=['index','Reliability','Band
gap values Clean','Band gap units','chemicalFormula Clean'])
```

Now, let's take a look at our target values first. Note that it still contains other input information (such as chemical formula) to help you contextualize what the bandgap values mean. Later, we will drop these columns as they won't be used in the model training.

```
target_data_df.head(10) # our original dataset with inputs and outputs
```

```
  chemicalFormula Clean Band gap values Clean Band gap units index
Reliability
0                  Li1F1                 13.105              eV     0
1
1                 Li1Cl1                  9.33              eV     6
1
2                 Li1Br1                  7.95              eV     7
1
3                 Li3Sb1                   1.0              eV     9
1
4                  Li1I1                   6.0              eV    10
1
5                 Li3Bi1                   0.7              eV    15
1
6                  Be1O1                 10.48              eV    16
1
7                  Be1S1                  4.17              eV    22
1
8                 Be1Se1                  3.61              eV    23
1
9                 Be3Sb2                  0.67              eV    24
1
```

Let's also take a look at the features generated. Looking at the column names you will notice that each of them follows the pattern of: ElementalProperty_composition_average

Some of these properties may be familiar to you such as AtomicWeight, which can be looked up in the periodic table of the elements. Others may be a bit harder to understand from their shorthand such as BCCefflatcnt, which stands for Body Centered Cubic effective lattice constant. In this case this property is describing information about how long certain bond lengths are within an idealized crystal of the element. Even though they are more complex they have still be tabulated by previous researchers and therefore MAST-ML is able to simply look them up from known resources to calculate the properties shown.

```
features_df.head(10) # features generated
```

```
   AtomicNumber_composition_average
AtomicRadii_composition_average  \
0                               6.0                                1.1350

1                              10.0                                1.2700

2                              19.0                                1.3450
```

```
3                                  15.0                                  1.5600
4                                  28.0                                  1.4400
5                                  23.0                                  1.5875
6                                   6.0                                  0.9250
7                                  10.0                                  1.1950
8                                  19.0                                  1.2600
9                                  22.8                                  1.3080

   AtomicVolume_composition_average  AtomicWeight_composition_average
\
0                       9311.576313                         12.969702
1                       9169.525548                         21.197000
2                         32.035942                         43.422500
3                         23.705899                         35.645750
4                         32.101458                         66.922735
5                         25.028908                         57.450850
6                       9300.147671                         12.505791
7                         17.632361                         20.538591
8                         17.653491                         43.986091
9                         16.935475                         54.111309

   BCCefflatcnt_composition_average  BCCenergy_pa_composition_average
\
0                          5.772386                         -1.346741
1                          6.658641                         -1.410040
2                          6.919518                         -1.432083
3                          6.704252                         -2.371630
4                          7.343549                         -1.459519
5                          6.767748                         -2.372997
```

```
6                            4.946102                            -3.057512

7                            5.648133                            -3.329031

8                            5.928280                            -3.278007

9                            6.025566                            -3.749129


   BCCfermi_composition_average  BCCmagmom_composition_average  \
0                     -0.679877                            0.0
1                      1.219961                            0.0
2                      1.117212                            0.0
3                      2.267697                            0.0
4                      2.360221                            0.0
5                      2.286196                            0.0
6                      5.584821                            0.0
7                      7.930848                            0.0
8                      6.853935                            0.0
9                      8.213330                            0.0

   BCCvolume_pa_composition_average
BCCvolume_padiff_composition_average  \
0                            12.470                            -
0.680417
1                            18.525                            -
2.020417
2                            21.035                            -
2.001667
3                            19.155                            -
1.180000
4                            25.935                            -
3.869167
5                            19.860                            -
0.822500
6                             7.565                            -
0.932500
7                            11.710                            -
5.128438
8                            13.985                            -
2.920000
9                            15.516                            -
1.842000

   ...  SecondIonizationEnergy_composition_average  \
0  ...                                    55.80400
1  ...                                    50.22400
2  ...                                    49.21900
3  ...                                    61.61100
4  ...                                    47.88450
```

```
5  ...                                          61.65025
6  ...                                          26.66400
7  ...                                          20.77050
8  ...                                          19.70050
9  ...                                          17.53860

   ShearModulus_composition_average
SpaceGroupNumber_composition_average  \
0                                2.10
122.00
1                                2.10
146.50
2                                2.10
146.50
3                                8.15
213.25
4                                2.10
146.50
5                                6.15
174.75
6                               66.00
103.00
7                               66.00
132.00
8                               67.85
104.00
9                               87.20
182.80

   SpecificHeatCapacity_composition_average  \
0                                   2.20300
1                                   2.03100
2                                   1.90400
3                                   2.73825
4                                   1.86350
5                                   2.71700
6                                   1.37250
7                                   1.26750
8                                   1.07250
9                                   1.17780

   ThermalConductivity_composition_average  \
0                                  42.36395
1                                  42.35445
2                                  42.41100
3                                  69.60000
4                                  42.57450
5                                  65.49250
6                                 100.13370
7                                 100.13450
```

```
8                                  101.02000
9                                  129.72000

   ThermalExpansionCoefficient_composition_average  \
0                                           923.00
1                                            23.00
2                                            23.00
3                                            37.25
4                                            66.50
5                                            37.85
6                                           395.65
7                                            40.65
8                                            28.15
9                                            11.18

   ThirdIonizationEnergy_composition_average
n_ws^third_composition_average  \
0                                   92.57900
0.490
1                                   81.03100
0.490
2                                   79.22550
0.490
3                                   98.16325
1.050
4                                   77.72550
0.490
5                                   98.22800
1.025
6                                  104.41350
0.835
7                                   94.36150
0.835
8                                   92.35650
0.835
9                                  102.45580
1.506

   phi_composition_average  valence_composition_average
0                   1.4250                          1.0
1                   1.4250                          4.0
2                   1.4250                          4.0
3                   3.2375                          2.0
4                   1.4250                          4.0
5                   3.1750                          2.0
6                   2.5250                          2.0
7                   2.5250                          4.0
8                   2.5250                          4.0
9                   4.7900                          3.2
```

```
[10 rows x 87 columns]
```

Answer

```
print("There are", features_df.shape[1], "features that we
generated.")
```

```
There are 87 features that we generated.
```

Answer

Based in the formula above:

AtomicNumber_CompositionAverage = (3 X 3 + 1 X 51) / (3 + 1) = 15

```
# Remove Constant Columns
features_df_noconstant = features_df.loc[:, (features_df !=
features_df.iloc[0]).any()]

# report number of columns
len(features_df_noconstant.columns)
```

```
86
```

Answer

1. How many features do we have left?

   86 features;

2. Should you worry about having too few useful features?

   Columns with constant values add no information and can negatively impact model performance, so removing them is a common preprocessing step. This reduces model complexity, improves efficiency, and helps prevent overfitting. However, it's crucial to retain enough relevant features for accurate predictions, making careful feature selection essential.

```
# Remove Highly correlated Features
# using notes here for methodology:
https://chrisalbon.com/machine_learning/feature_selection/drop_highly_
correlated_features/

features_corr_df = features_df_noconstant.corr(method="pearson").abs()
features_corr_df.iloc[:5, :5] # Preview the first 5 rows/columns of
the correlation matrix
```

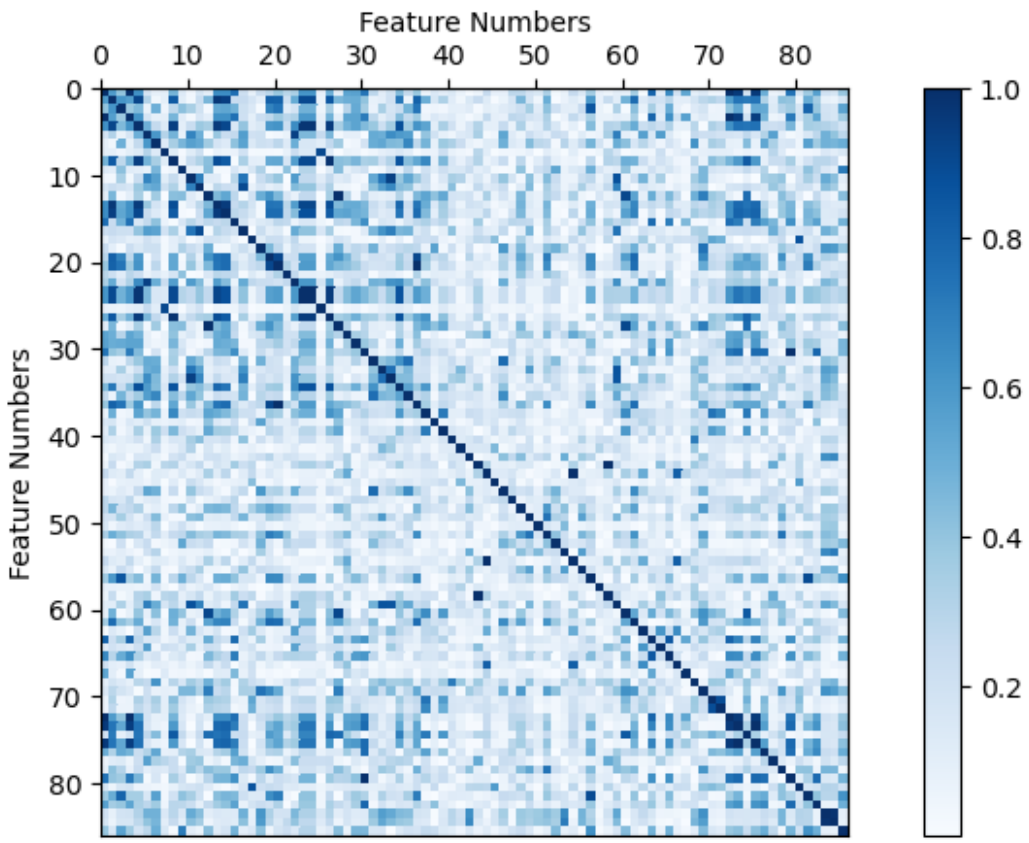
```
                                AtomicNumber_composition_average  \
AtomicNumber_composition_average                          1.000000
```

```
AtomicRadii_composition_average                    0.585580
AtomicVolume_composition_average                   0.405061
AtomicWeight_composition_average                   0.998598
BCCefflatcnt_composition_average                   0.628049
```

```
                                  AtomicRadii_composition_average  \
AtomicNumber_composition_average                         0.585580
AtomicRadii_composition_average                          1.000000
AtomicVolume_composition_average                         0.609457
AtomicWeight_composition_average                         0.571820
BCCefflatcnt_composition_average                         0.918506

                                  AtomicVolume_composition_average  \
AtomicNumber_composition_average                          0.405061
AtomicRadii_composition_average                           0.609457
AtomicVolume_composition_average                          1.000000
AtomicWeight_composition_average                          0.382296
BCCefflatcnt_composition_average                          0.449378

                                  AtomicWeight_composition_average  \
AtomicNumber_composition_average                          0.998598
AtomicRadii_composition_average                           0.571820
AtomicVolume_composition_average                          0.382296
AtomicWeight_composition_average                          1.000000
BCCefflatcnt_composition_average                          0.615523

                                  BCCefflatcnt_composition_average
AtomicNumber_composition_average                          0.628049
AtomicRadii_composition_average                           0.918506
AtomicVolume_composition_average                          0.449378
AtomicWeight_composition_average                          0.615523
BCCefflatcnt_composition_average                          1.000000
```

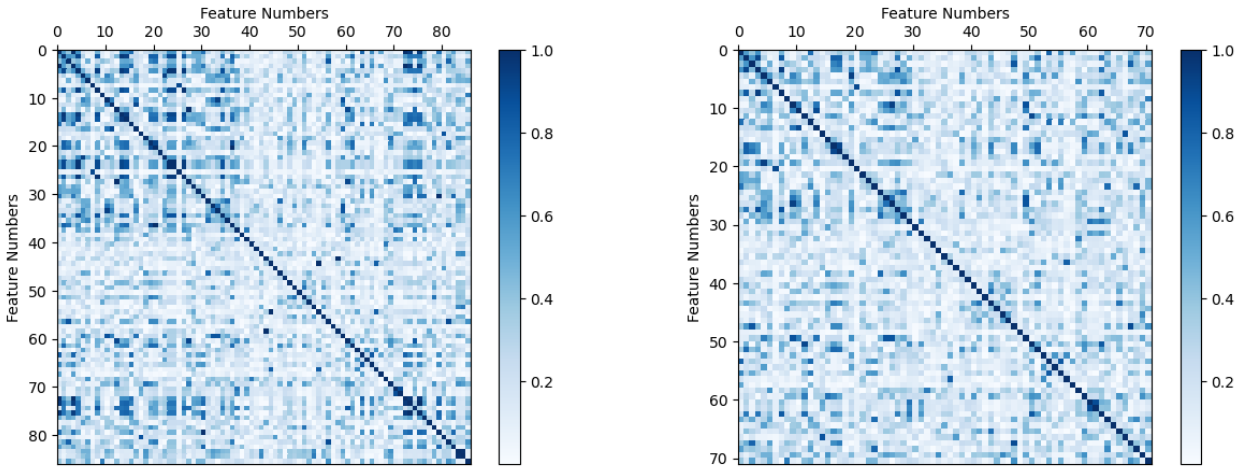
A better way to interpret this correlation matrix is by plotting a heatmap: The darker the color on the plot, the more highly correlated features are.
Note the diagonal line with all 1 values. This is because each feature is by definition perfectly correlated with itself.

```
# before removing correlated features
fig1, ax1 = plt.subplots(figsize=(10,5))
c = ax1.pcolor(features_corr_df,cmap="Blues")
ax1.set_ylim(ax1.get_ylim()[::-1])
ax1.xaxis.set_ticks_position('top')
ax1.xaxis.set_label_position('top')
ax1.set_xlabel('Feature Numbers')
ax1.set_ylabel('Feature Numbers')
ax1.set_aspect('equal')
plt.colorbar(c,ax=ax1)
plt.show()
```




```
# # Filter the features with correlation coefficients above 0.95
# upper =
features_corr_df.where(np.triu(np.ones(features_corr_df.shape),
k=1).astype(np.bool))
# to_drop = [column for column in upper.columns if any(upper[column] >
0.95)]
# features_df_lowcorr = features_df_noconstant.drop(columns=to_drop)
# # recalculate the correlation matrix so we can compare
# features_corr_df_update =
features_df_lowcorr.corr(method="pearson").abs()

# Filter the features with correlation coefficients above 0.95
upper =
features_corr_df.where(np.triu(np.ones(features_corr_df.shape),
k=1).astype(bool))
to_drop = [column for column in upper.columns if any(upper[column] >
0.95)]
features_df_lowcorr = features_df_noconstant.drop(columns=to_drop)

# Recalculate the correlation matrix so we can compare
features_corr_df_update =
features_df_lowcorr.corr(method="pearson").abs()
```

```
# plot correlation after removing highly correlated features

fig1, (ax1,ax2) = plt.subplots(1,2,figsize=(15,5))
c1 = ax1.pcolor(features_corr_df,cmap="Blues")
ax1.set_ylim(ax1.get_ylim()[::-1])
ax1.xaxis.set_ticks_position('top')
ax1.xaxis.set_label_position('top')
ax1.set_xlabel('Feature Numbers')
ax1.set_ylabel('Feature Numbers')
ax1.set_aspect('equal')

plt.colorbar(c1,ax=ax1)

c2 = ax2.pcolor(features_corr_df_update,cmap="Blues")
ax2.set_ylim(ax2.get_ylim()[::-1])
ax2.xaxis.set_ticks_position('top')
ax2.xaxis.set_label_position('top')
ax2.set_xlabel('Feature Numbers')
ax2.set_ylabel('Feature Numbers')
ax2.set_aspect('equal')
plt.colorbar(c2,ax=ax2)
plt.show()
```




```
len(features_df_lowcorr.columns)
```

```
71
```

Answer

1. After filtering for highly correlated features how many features do we have left?

   71;

2. Are we worried about having too few useful features?

Removing highly correlated features reduces multicollinearity and improves model performance but must be done carefully. Excessive removal may result in losing important information and reducing predictive accuracy. Balancing redundancy removal with retaining relevant features, guided by domain knowledge, ensures an effective and informative feature set.

```
features_df_lowcorr.head(10)
```

```
   AtomicNumber_composition_average
AtomicRadii_composition_average  \
0                               6.0                                1.1350

1                              10.0                                1.2700

2                              19.0                                1.3450

3                              15.0                                1.5600

4                              28.0                                1.4400

5                              23.0                                1.5875

6                               6.0                                0.9250

7                              10.0                                1.1950

8                              19.0                                1.2600

9                              22.8                                1.3080


   AtomicVolume_composition_average  BCCefflatcnt_composition_average
\
0                       9311.576313                           5.772386

1                       9169.525548                           6.658641

2                         32.035942                           6.919518

3                         23.705899                           6.704252

4                         32.101458                           7.343549

5                         25.028908                           6.767748

6                       9300.147671                           4.946102

7                         17.632361                           5.648133

8                         17.653491                           5.928280
```

```
9                          16.935475                           6.025566


   BCCenergy_pa_composition_average  BCCfermi_composition_average  \
0                         -1.346741                     -0.679877
1                         -1.410040                      1.219961
2                         -1.432083                      1.117212
3                         -2.371630                      2.267697
4                         -1.459519                      2.360221
5                         -2.372997                      2.286196
6                         -3.057512                      5.584821
7                         -3.329031                      7.930848
8                         -3.278007                      6.853935
9                         -3.749129                      8.213330

   BCCmagmom_composition_average  BCCvolume_pa_composition_average  \
0                            0.0                            12.470
1                            0.0                            18.525
2                            0.0                            21.035
3                            0.0                            19.155
4                            0.0                            25.935
5                            0.0                            19.860
6                            0.0                             7.565
7                            0.0                            11.710
8                            0.0                            13.985
9                            0.0                            15.516

   BCCvolume_padiff_composition_average  BoilingT_composition_average
...  \
0                             -0.680417                        849.94
...
1                             -2.020417                        926.98
...
2                             -2.001667                        973.50
...
3                             -1.180000                       1676.25
...
4                             -3.869167                       1036.15
...
5                             -0.822500                       1670.50
...
6                             -0.932500                       1416.55
...
7                             -5.128438                       1730.36
...
8                             -2.920000                       1850.50
...
9                             -1.842000                       2389.80
...
```

```
   NsValence_composition_average
Polarizability_composition_average  \
0                            1.50                                12.44600

1                            1.50                                13.25750

2                            1.50                                13.69250

3                            1.25                                19.90125

4                            1.50                                14.68000

5                            1.25                                18.35125

6                            2.00                                 3.20100

7                            2.00                                 4.25000

8                            2.00                                 4.68500

9                            2.00                                 6.00000


   SecondIonizationEnergy_composition_average  \
0                                    55.80400
1                                    50.22400
2                                    49.21900
3                                    61.61100
4                                    47.88450
5                                    61.65025
6                                    26.66400
7                                    20.77050
8                                    19.70050
9                                    17.53860

   ShearModulus_composition_average
SpaceGroupNumber_composition_average  \
0                                2.10
122.00
1                                2.10
146.50
2                                2.10
146.50
3                                8.15
213.25
4                                2.10
146.50
5                                6.15
174.75
6                               66.00
```

```
103.00
7                                66.00
132.00
8                                67.85
104.00
9                                87.20
182.80

   ThermalConductivity_composition_average  \
0                                  42.36395
1                                  42.35445
2                                  42.41100
3                                  69.60000
4                                  42.57450
5                                  65.49250
6                                 100.13370
7                                 100.13450
8                                 101.02000
9                                 129.72000

   ThermalExpansionCoefficient_composition_average  \
0                                           923.00
1                                            23.00
2                                            23.00
3                                            37.25
4                                            66.50
5                                            37.85
6                                           395.65
7                                            40.65
8                                            28.15
9                                            11.18

   ThirdIonizationEnergy_composition_average
n_ws^third_composition_average  \
0                                    92.57900
0.490
1                                    81.03100
0.490
2                                    79.22550
0.490
3                                    98.16325
1.050
4                                    77.72550
0.490
5                                    98.22800
1.025
6                                   104.41350
0.835
7                                    94.36150
0.835
```

```
8                                    92.35650
0.835
9                                   102.45580
1.506

   valence_composition_average
0                          1.0
1                          4.0
2                          4.0
3                          2.0
4                          4.0
5                          2.0
6                          2.0
7                          4.0
8                          4.0
9                          3.2

[10 rows x 71 columns]
```

It should be fairly apparent that our features come in many shapes and sizes. Machine Learning algorithms can be very sensitive to these differences.
For example one feature may be several orders of magnitude larger in values and in range of values.
This can make some algorithms significantly biased towards those features so the best practice is usually to perform some alteration to make all the features look similar, while still preserving the information they contain

In our case we're going to linearly rescale the features so that they all have the same minimum and same maximum. If you're interested in checking of the details of how this is done you can check out the documentation for the Scikit-learn method we'll be using: https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html

```
minmax_features = MinMaxScaler().fit_transform(features_df_lowcorr)
minmax_features_df =
pd.DataFrame(minmax_features,columns=features_df_lowcorr.columns)
minmax_features_df.iloc[:5, :5]

   AtomicNumber_composition_average
AtomicRadii_composition_average  \
0                          0.012821                                0.190923

1                          0.064103                                0.275430

2                          0.179487                                0.322379

3                          0.128205                                0.456964

4                          0.294872                                0.381847
```

```
   AtomicVolume_composition_average  BCCefflatcnt_composition_average
\
0                          0.583946                          0.176111

1                          0.575030                          0.310002

2                          0.001553                          0.349415

3                          0.001030                          0.316893

4                          0.001557                          0.413475


   BCCenergy_pa_composition_average
0                          0.893262
1                          0.884705
2                          0.881725
3                          0.754709
4                          0.878016
```

challenge: Turn off the feature normalization. Feature normalization is a common practice to enable models to better learn from multiple features when some are on significantly different scales. Try removing this section to see how the later results are affected. In the case of the decision tree / random forest model being used by default this may not be the case, but what about other model types? Try doing the same thing with a Kernel Ridge Regression Model for example.

Notice how compared to some of the previous sections, performing the scaling only took a few lines of code. This is the power of using existing code packages and tools that are already out there!

First, we store our cleaned and normalized inputs and outputs in new variables `X` and `y` for easier understanding and manipulation.

```
X = minmax_features_df                          # inputs/features
y = target_data_df["Band gap values Clean"] # outputs/targets
```

Normally, if you are using machine learning to predict the bandgap (or other properties) of a novel material, you won't know its real bandgap until you fabricate and measure it in the lab, which is bad news for instructors: What's the point if we weren't able to validate the predictions and show you the power of machine learning?

Therefore for instructional purposes, we will stage our prediction by using 5 common materials with known bandgap values instead - Si, SiO2, C, NaCl, and Sn - and removing them from the dataset.

The following code accomplishes the above and is not important otherwise. Note that you do **NOT** need this step in a real research setting. Note that we will use this modified dataset (named *_predict) later in the lab.

```
# Find prediction compounds and generate inputs for them to make
predictions later.
def extract_predictions(formula="string"):
    index_prediction = target_data_df[target_data_df["chemicalFormula
Clean"]==formula].index
    xpredict = X.loc[index_prediction].copy()
    ypredict = y.loc[index_prediction]

    return (index_prediction,xpredict,ypredict)

index_predict_Si, xpredict_Si, ypredict_Si =
extract_predictions(formula="Si")
index_predict_SiO2, xpredict_SiO2, ypredict_SiO2 =
extract_predictions(formula="Si1O2")
index_predict_C, xpredict_C, ypredict_C =
extract_predictions(formula="C")
index_predict_Sn, xpredict_Sn, ypredict_Sn =
extract_predictions(formula="Sn")
index_predict_NaCl, xpredict_NaCl, ypredict_NaCl =
extract_predictions(formula="Na1Cl1")

X_predict = X.drop(index=index_predict_Si.to_list()
+index_predict_SiO2.to_list()+index_predict_C.to_list()
+index_predict_Sn.to_list()+index_predict_NaCl.to_list())
y_predict = y.drop(index=index_predict_Si.to_list()
+index_predict_SiO2.to_list()+index_predict_C.to_list()
+index_predict_Sn.to_list()+index_predict_NaCl.to_list())
```

Then, we use the `train_test_split()` method from the `scikit-learn` package to generate the split. In this case, our input data `X` and output data `y` are split into 4 parts:

- `X_train`: training set input data
- `X_test`: test set input data
- `y_train`: training set output data
- `y_test`: test set output data

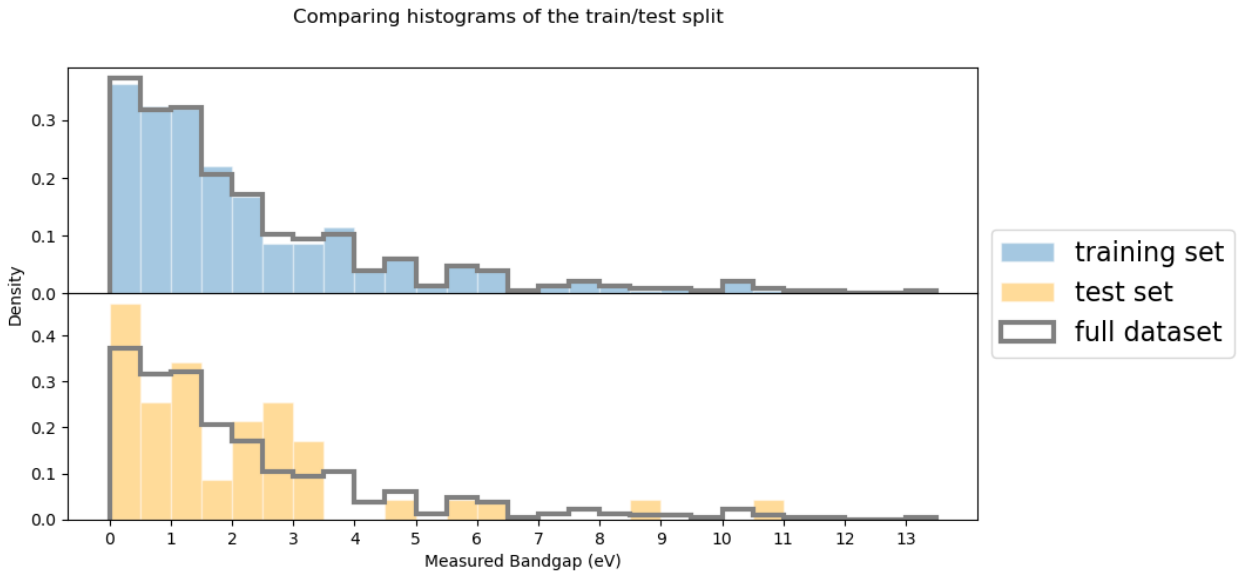
We will continue referencing these 4 objects throughout the rest of this lab.

```
# Generate train/test split by reserving 10% of data as test set

test_fraction = 0.1
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=test_fraction, shuffle=True,random_state=seed)

fig, (ax1, ax2) = plt.subplots(2, figsize=(10,5), sharex = True,
gridspec_kw={'hspace': 0})
```

```
fig.set_tight_layout(False)
myarray = mastml_df_clean["Band gap values Clean"]

bins = np.true_divide(range(28),2)

l1 = sns.distplot(y_train.astype("float"), hist = True, norm_hist =
True, kde = False, bins = bins, hist_kws={"edgecolor": "white"}, label
= 'training set', ax = ax1)
l2 = sns.distplot(y_test.astype("float"), hist = True, norm_hist =
True, kde = False, bins = bins, hist_kws={"edgecolor": "white",
"color": "orange"}, label = 'test set', ax = ax2)
l3 = sns.distplot(myarray, hist = True, norm_hist = True, kde = False,
bins = bins, hist_kws={"histtype": "step","linewidth": 3, "alpha": 1,
"color": "grey"}, ax = ax1)
l4 = sns.distplot(myarray, hist = True, norm_hist = True, kde = False,
bins = bins, hist_kws={"histtype": "step","linewidth": 3, "alpha": 1,
"color": "grey"}, label = 'full dataset', ax = ax2)


ax1.set_xticks(range(14))
ax2.set_xticks(range(14))
ax2.xaxis.label.set_visible(False)
handles, labels = [(a + b) for a, b in
zip(ax1.get_legend_handles_labels(), ax2.get_legend_handles_labels())]
fig.suptitle('Comparing histograms of the train/test split')
fig.add_subplot(111, frame_on=False)
plt.tick_params(labelcolor="none", bottom=False, left=False)
plt.legend(handles, labels, loc = 'center left', bbox_to_anchor=(1,
0.5),prop={'size': 16})
plt.xlabel('Measured Bandgap (eV)')
_ = plt.ylabel('Density')
```

```
C:\Users\jhyang\AppData\Local\Temp\ipykernel_12820\1844405501.py:7:
UserWarning:

`distplot` is a deprecated function and will be removed in seaborn
v0.14.0.

Please adapt your code to use either `displot` (a figure-level
function with
similar flexibility) or `histplot` (an axes-level function for
histograms).

For a guide to updating your code to use the new functions, please see
https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751

  l1 = sns.distplot(y_train.astype("float"), hist = True, norm_hist =
True, kde = False, bins = bins, hist_kws={"edgecolor": "white"}, label
= 'training set', ax = ax1)
C:\Users\jhyang\AppData\Local\Temp\ipykernel_12820\1844405501.py:8:
```

```
UserWarning:

`distplot` is a deprecated function and will be removed in seaborn
v0.14.0.

Please adapt your code to use either `displot` (a figure-level
function with
similar flexibility) or `histplot` (an axes-level function for
histograms).

For a guide to updating your code to use the new functions, please see
https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751

  l2 = sns.distplot(y_test.astype("float"), hist = True, norm_hist =
True, kde = False, bins = bins, hist_kws={"edgecolor": "white",
"color": "orange"}, label = 'test set', ax = ax2)
C:\Users\jhyang\AppData\Local\Temp\ipykernel_12820\1844405501.py:9:
UserWarning:

`distplot` is a deprecated function and will be removed in seaborn
v0.14.0.

Please adapt your code to use either `displot` (a figure-level
function with
similar flexibility) or `histplot` (an axes-level function for
histograms).

For a guide to updating your code to use the new functions, please see
https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751

  l3 = sns.distplot(myarray, hist = True, norm_hist = True, kde =
False, bins = bins, hist_kws={"histtype": "step","linewidth": 3,
"alpha": 1, "color": "grey"}, ax = ax1)
C:\Users\jhyang\AppData\Local\Temp\ipykernel_12820\1844405501.py:10:
UserWarning:

`distplot` is a deprecated function and will be removed in seaborn
v0.14.0.

Please adapt your code to use either `displot` (a figure-level
function with
similar flexibility) or `histplot` (an axes-level function for
histograms).

For a guide to updating your code to use the new functions, please see
https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751

  l4 = sns.distplot(myarray, hist = True, norm_hist = True, kde =
False, bins = bins, hist_kws={"histtype": "step","linewidth": 3,
"alpha": 1, "color": "grey"}, label = 'full dataset', ax = ax2)
```



Answer

The test dataset should share a similar distribution with the training dataset to assess the model's generalization accurately. However, the bar chart indicates a mismatch in distributions, suggesting the test split may not adequately represent the full dataset.

```
Default_model =
RandomForestRegressor(random_state=seed,n_estimators=1,bootstrap=False
).fit(X_train,y_train) # fit the decision tree model
print('Model training complete.')
# print('Tree depth:', [estimator.tree_.max_depth for estimator in
Default_model.estimators_])
# for importance in zip (estimator.feature_importances_ for estimator
in Default_model.estimators_):
#     print (importance)
#print('Leaf nodes:',[estimator.tree_.n_leaves for estimator in
Default_model.estimators_])
```

```
Model training complete.
```

The outputs above describes the hyperparameters selected (in this case, by default) to fit the decision tree model. and the parameters being generated in the training process. You may also wonder what the decision tree *looks* like, and we will visualize the entire tree later.

We'll also go into more detail about what these (hyper)parameters are at a later time when they become more relevant. For now, we're glossing over because simply knowing these (hyper)parameters doesn't help us evaluate model quality until we have seen its performance: How accurately and precisely can our decision tree model predict bandgaps? We will jump into that.

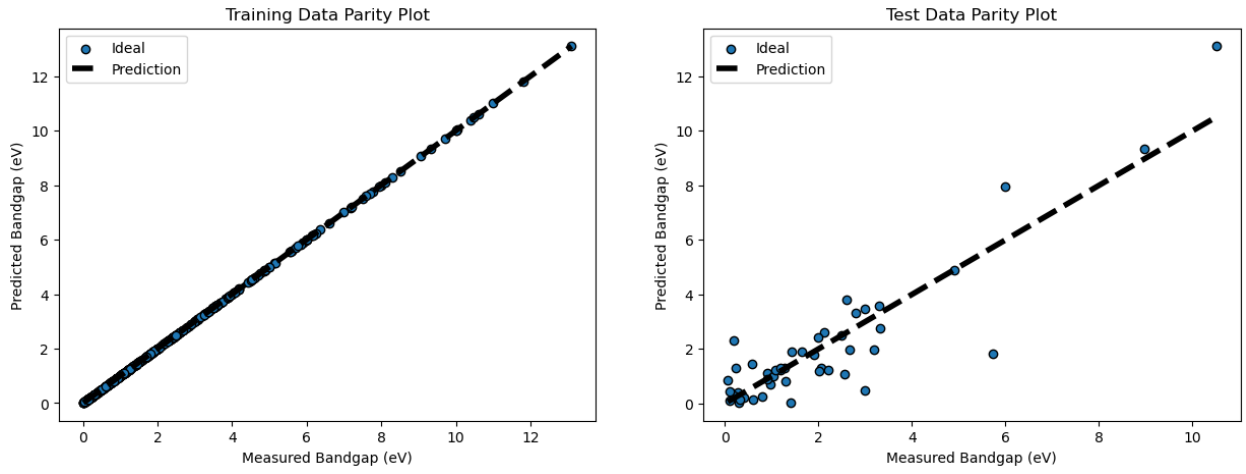
As one last motivation as we start asssessing our model, lets predict two band gaps of materials you're probably familiar with, Silicon and Silica. Silicon is used in practically every electronic device as a semiconductor, and Silica is basic window glass. You can look up the values of their

band gaps fro reference, but look how just in a few lines of code the model can already give us a rough idea of their values. We know Silicon is a semi-conductor and it's bandgap should be fairly low, while the band gap for Silica has to be much higher because window glass shouldn't absorb any light at all. Based on these predictions it seems like the model can already pick up on these trends! But as we've been mentioning, just making a few select predictions is not a good way to measure overall performance, in the next sections we'll dig into more robus ways to measure the performance!

```
Default_model_all_data =
RandomForestRegressor(random_state=seed,n_estimators=1,bootstrap=False
).fit(X_predict,y_predict)

print("Predicting Silicon Band Gap:
",Default_model_all_data.predict(xpredict_Si))

print("Predicting Silica Band Gap:
",Default_model_all_data.predict(xpredict_SiO2))
```

```
Predicting Silicon Band Gap:  [2.]
Predicting Silica Band Gap:  [6.]
```

```
Train_predictions = Default_model.predict(X_train)    # Make
predictions on training data
Test_predictions = Default_model.predict(X_test)      # Make
predictions on testing data

parity_plots_side_by_side(y_train,Train_predictions,y_test,Test_predic
tions,title_left="Training Data Parity Plot",title_right="Test Data
Parity Plot") # build both plots
parity_stats_side_by_side(y_train,Train_predictions,y_test,Test_predic
tions,"Training Data","Test Data")  # print error metrics for training
data
```




```
  Error Metric Training Data    Test Data
Note
```

```
0         RMSE   0.0003 (eV)  1.0492 (eV)  (0.0 for perfect
prediction)
1     RMSE/std        0.0001       0.4884  (0.0 for perfect
prediction)
2          MAE      0.0 (eV)  0.6811 (eV)  (0.0 for perfect
prediction)
3           R2           1.0       0.7614  (1.0 for perfect
prediction)
```

We've generated a few different error metrics which we can use to asses the model's performance. One that we'll focus on throughout the lab is the Root Mean Squared Error (RMSE), which we are going to use as a rough error bar when talking about predictive ability of the model. Meaning when we're analyzing performance and asking questions about how accurate the model is is making predictions this is the statistic we'll reference. It's important to note that this is just one choice we could make for assigning an error bar to the model's predictions. There are other, more complex methods for generating error bars for model predictions, and we're going to ignore those for now in favor of simplicity. So whenever we're asking you to think about the predictive power of the model for now think of model predictions as having a predicted value plus or minus the RMSE.

Answer

1. Is there enough information in the features to make predictions? Do the features model this data (training) well?

   If the training data plot shows predicted values aligning closely with the 45-degree line, it indicates that the features provide sufficient information and the model effectively fits the training data.

2. Are there any outliers?

   Through the training plot, it is possible to observe whether there are outliers that are distinctly different from the other data points. From the graph above, there are no outliers shown. Therefore, there are no outliers.

3. Does it consistently overpredict/underpredict bandgap values in any particular range?

   If the predicted values deviate from the 45-degree line and consistently exhibit overestimation or underestimation within a certain range of bandgap values, then the model may perform poorly within that range. From the training data plot, there is no predicted value deviate from the 45-degree line. There is no overpredict/underpredict bandgap values in any particular range.

4. Can we use this model to predict bandgap values of materials for making single-junction solar cell, which requires a bandgap between 1.1 and 1.7eV?

   The training RMSE is 0.0003 eV, while the test RMSE is 1.0492 eV, indicating a significant accuracy drop. Since the test RMSE exceeds the desired bandgap range

(1.1–1.7 eV) for single-junction solar cells, the model's predictions may not be accurate enough for this application.

5. Can we use this model to predict high bandgap materials above 3 eV?

   The test RMSE of 1.0492 eV, compared to 0.0003 eV in training, shows poor generalization to high bandgap materials, with significant errors. The RMSE/std ratio of 0.4884 further highlights high prediction errors, making the model potentially unreliable for applications requiring high bandgap materials (>3 eV).

Answer

1. Compare both the parity plots and performance statistics for the training and test set. Is the model performing better on one set than the other, or is there no difference? (No calculation needed.)

   In a parity plot, data points aligning along the 45-degree line indicate accurate predictions. If training points closely follow the line but test points deviate, it suggests the model performs better on training data than on testing data, highlighting potential overfitting or poor generalization.

2. Which of the following most accurately describes this model: Underfit, overfit, or neither?

   If the model fits the training data extremely well but performs poorly on the test data, it may be overfitting. If the model performs poorly on both the training and test data, it may be underfitting. If the model performs well on both datasets, it is likely an appropriate model. Based on the graphs, it should be overfitting.

3. Should we use training data or test data to estimate model prediction performance?

   In general, the test dataset is used to assess the model's performance on unseen data, making it suitable for evaluating the model's generalization ability. The training dataset is primarily used for model training and is not a reliable dataset for performance estimation because the model may perform well on the training data but poorly on new, unseen data. Therefore, we should use test data.

```
# # generate an image of the default decision tree
# dot_data =
sklearn.tree.export_graphviz(Default_model.estimators_[0],out_file=Non
e,feature_names=features_df_lowcorr.columns,filled=True,rounded=True,s
pecial_characters=True)
# graph = graphviz.Source(dot_data)
# graph.render(view=True, format="pdf",
filename="./output/decisiontree_pdf")
```

This visualization explicitly constructs each node in the default tree. Using the decision node `Density_composition_average ≤ 0.059` as an example,

It lists which feature the node splits on, gives the mse for the data at that split, how many samples are at the node, and the value of the estimated band gap if it was a leaf node.
It should be immediately apparent that the tree is incredibly large, and and so we'll pick a few things to focus on as we look through it briefly.

challenge (optional): We're currently leaving out a fairly small percentage of the data, 10%. Try going back and changing this to a few different values to see how it changes the results. For example try: 5%, 25%, 50%, 75%.
To do this you can edit the test_fraction parameter at the start of Section 4 and rerun the cells after that.

```
print('Default model uses the following hyperparameters:\n') # print
default hyperparameters used
pprint(Default_model.get_params())
```

```
Default model uses the following hyperparameters:

{'bootstrap': False,
 'ccp_alpha': 0.0,
 'criterion': 'squared_error',
 'max_depth': None,
 'max_features': 1.0,
 'max_leaf_nodes': None,
 'max_samples': None,
 'min_impurity_decrease': 0.0,
 'min_samples_leaf': 1,
 'min_samples_split': 2,
 'min_weight_fraction_leaf': 0.0,
 'monotonic_cst': None,
 'n_estimators': 1,
 'n_jobs': None,
 'oob_score': False,
 'random_state': 2345312,
 'verbose': 0,
 'warm_start': False}
```

```
# set up hyperparameter grid (a dictionary of hyperparameter
candidates that we want the optimization strategy to consider)

# EDIT LIST TO TRY DIFFERENT VALUES!
### MAKE EDITS BELOW HERE ###

# Short
number_of_trees = [1,10,25,50]
# long
number_of_trees = [1,3,5,7,10,15,20,50]


### MAKE EDITS ABOVE HERE ###

opt_dict = {'n_estimators':number_of_trees,'bootstrap':[bool(1)]}
```

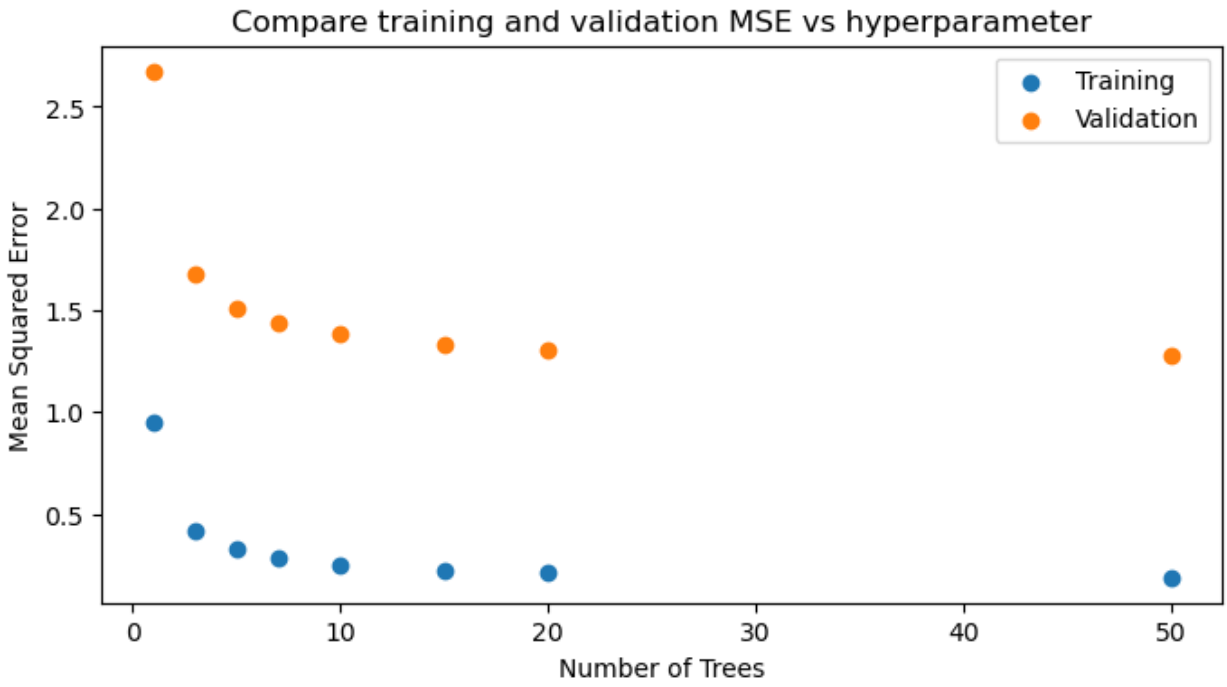
```
# kfold = KFold(n_splits=5,random_state=seed,shuffle=True)
kfold = RepeatedKFold(n_splits=5,
                      random_state=seed,
                      n_repeats=5)

import time

CV = GridSearchCV(Default_model, # 1. the model whose hyperparamter is
being optimized right now
                  opt_dict,   # 2. a dictionary of values that we want
the grid search to use
                  cv=kfold,   # 4. k-fold cross-validation strategy is
used to define training and validation splits (note this is separate
from test splits) to be used for each grid point
                  return_train_score=True,

scoring=['neg_mean_squared_error','r2','neg_mean_absolute_error'], #
5. the performance metrics to be reported at each grid point specified
in opt_dict
                  refit='neg_mean_squared_error')

# perform grid search
tic = time.perf_counter() # start timer

CV = CV.fit(X_train,y_train)

toc = time.perf_counter() # stop timer

# print results
print(f"Grid search completed in {toc - tic:0.3f} seconds.")
print(CV.best_params_)
```

```
Grid search completed in 22.281 seconds.
{'bootstrap': True, 'n_estimators': 50}
```

And just like that we've performed the grid search! to visualize the results see the code blocks below.

```
# plot number of trees vs train and test MSE

opt_dict_array = opt_dict["n_estimators"]                    # array
of grid points (x-axis)
train_mse = CV.cv_results_["mean_train_neg_mean_squared_error"] # MSE
of training set at each grid point (y-axis)
test_mse = CV.cv_results_["mean_test_neg_mean_squared_error"]   # MSE
of test set at each grid point (y-axis)

fig1,ax1 = plt.subplots(figsize=(8,4))
ax1.scatter(opt_dict_array, -train_mse)
ax1.scatter(opt_dict_array, -test_mse)
```

```
# ax1.fill_between(opt_dict_array, -train_mse, -test_mse, alpha=0.1)
ax1.set_xlabel('Number of Trees')
ax1.set_ylabel('Mean Squared Error')
ax1.set_title('Compare training and validation MSE vs hyperparameter')
plt.legend(["Training","Validation","difference"])
plt.show()
print("Minimum Mean Squared Error: ", round(min(-test_mse),4))
print("Number of Trees at minimum: ", opt_dict_array[np.argmin(-
test_mse)])
```




```
Minimum Mean Squared Error:  1.2795
Number of Trees at minimum:  50
```

Answer

1. Looking at the validation data, about how many trees do you think is optimal to get the lowest errors with the simplest model?

   50

```
# check what the best parameters identified in the grid search were
CV.best_params_
```

```
{'bootstrap': True, 'n_estimators': 50}
```

```
# Extract cross validation performance metrics for the optimized model
opt_CV_stats = CV_best_stats(CV,y_train)
```

```
Average test RMSE:  1.1311 (0.0 for perfect prediction)
Average test RMSE/std:  0.4922 (0.0 for perfect prediction)
Average test MAE:  0.7632 (0.0 for perfect prediction)
Average test R2:  0.7488 (1.0 for perfect prediction)
```

To compare back to our default model we can construct another grid search that only uses "1" for `n_estimators`. That way it will still be the best model available.

We do CV on a single grid point ("1"), build 25 different models, and average across them to get the results below.

Now we can directly compare the model's performance on these metrics generated from the Kfold cross validation.

```
default_opt_dict = {'n_estimators':[1]}

default_CV = GridSearchCV(Default_model,
                          default_opt_dict,
                          cv=kfold,
                          return_train_score=True,

scoring=['neg_mean_squared_error','r2','neg_mean_absolute_error'],
                          refit='neg_mean_squared_error')
default_CV = default_CV.fit(X_train,y_train)

default_CV_stats = CV_best_stats(default_CV,y_train)
```

```
Average test RMSE:  1.4691 (0.0 for perfect prediction)
Average test RMSE/std:  0.6393 (0.0 for perfect prediction)
Average test MAE:  0.9779 (0.0 for perfect prediction)
Average test R2:  0.5774 (1.0 for perfect prediction)
```

Answer

1. Do we get improvement in the RMSE between the default and optimized model? What is the percentage improvement ($\left|RMSE_{testopt} - RMSE_{testdefault}\right| / RMSE_{testdefault}$)?

   Percentage Improvement = |1.1311 − 1.4691| / 1.4691 × 100 ≈ 23.01%;

   Yes, we have indeed observed an improvement in RMSE between the default model and the optimized model. The average test RMSE for the optimized model is 1.1311 eV, whereas for the default model, it is 1.4691 eV. Therefore, the RMSE of the optimized model is lower than that of the default model, indicating an enhancement in model performance on the test data.

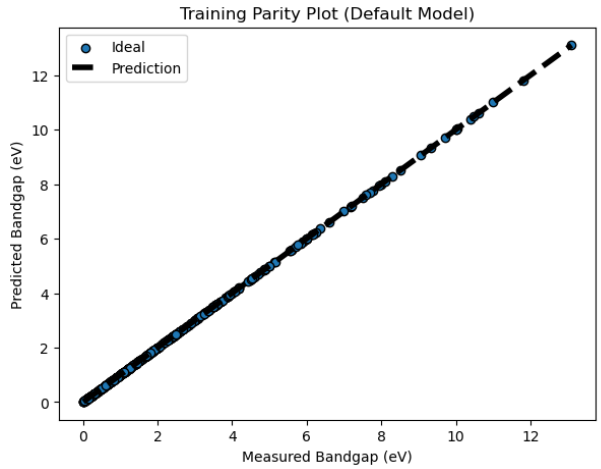
2. Assuming this level of accuraccy from the optimized model. Is our model accurate enough to predict single-junction solar materials? where the key design metric is having a band gap between 1.1 eV and 1.7 eV?

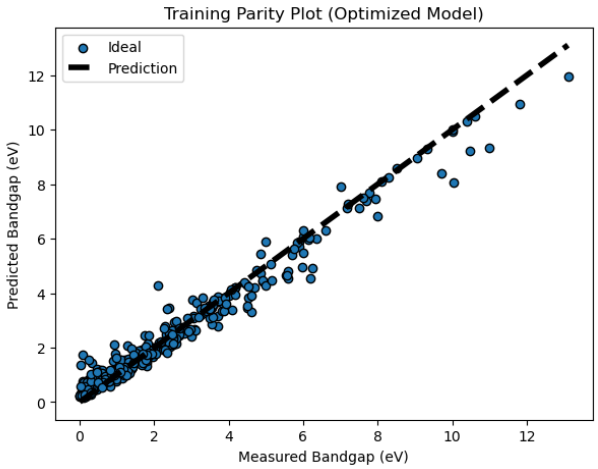
With the optimized model's RMSE of 1.1311, it indicates that, on average, the model's predictions have an error of about 1.1311 eV. While this represents an improvement over the default model, it's still relatively high for predicting single-junction solar materials, where the key design metric is having a band gap between 1.1 eV and 1.7 eV. The model's accuracy may not be sufficient for this task.

3. How about our other task. Is the optimized model accurate enough to predict high bandgap materials? where the key metric is ensuring predictions are above 3 eV?

   Similarly, for predicting high bandgap materials (above 3 eV), the optimized model's performance with an RMSE of 1.1311 may still not be accurate enough. High bandgap materials require precise identification, and the model's average error of about 1.1311 eV may not meet this requirement effectively.

```
# Refit the model using the best hyperparameters
DT2 = CV.best_estimator_.fit(X_train,y_train)

# predict both the train and test data
Train_predictions2 = DT2.predict(X_train)
Test_predictions2 = DT2.predict(X_test)

parity_plots_side_by_side(y_train,Train_predictions,y_train,Train_pred
ictions2,title_left="Training Parity Plot (Default
Model)",title_right="Training Parity Plot (Optimized Model)") # build
both plots
parity_stats_side_by_side(y_train,Train_predictions,y_train,Train_pred
ictions2,"Training Set (Default Model)","Training Set (Optimized
Model)")
```






```
  Error Metric Training Set (Default Model) Training Set (Optimized
Model)  \
0         RMSE                    0.0003 (eV)                     0.4226
(eV)
1     RMSE/std                         0.0001
```

```
0.1839
2          MAE                        0.0 (eV)                        0.274
(eV)
3           R2                         1.0
0.9662

                             Note
0  (0.0 for perfect prediction)
1  (0.0 for perfect prediction)
2  (0.0 for perfect prediction)
3  (1.0 for perfect prediction)
```

Answer

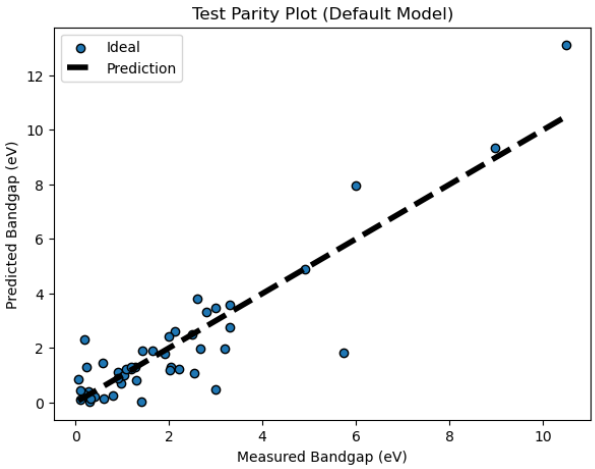
1. Look at both the parity plots and the training data statistics. Does the optimized model do better or worse at predicting the training data than the default model?

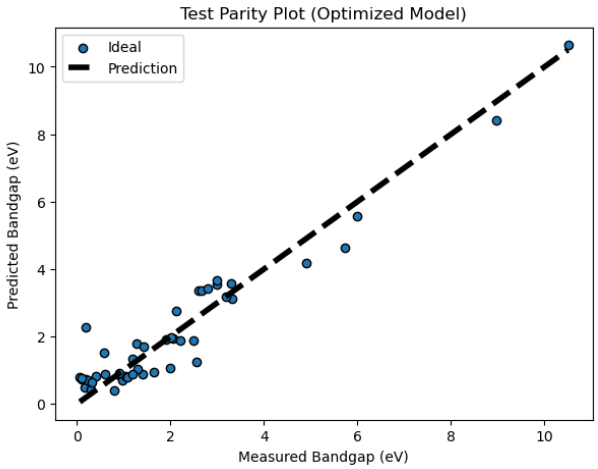
   The default model outperforms the optimized model on training data, with lower RMSE (0.0003 vs. 0.4226 eV), RMSE/std (0.0001 vs. 0.1839), MAE (0.0 vs. 0.274 eV), and higher R^2 (1.0 vs. 0.9662). The optimized model shows larger prediction errors and poorer fit. While the default model's predictions align closely with the 45-degree line, the optimized model's predictions deviate, indicating overestimation or underestimation in certain bandgap ranges.

2. Do prediction performances on training data give you enough information to decide which model is more likely to give better predictions on Si and SiO_2, which are not in the training set or the test set? Another way to ask this is does the training data result tell us anything about the predictive power of the model?

   The results on the training data are primarily used to assess the model's fit to the training data but may not adequately reflect the model's performance on unknown materials such as Si and SiO2, which are not included in the training set or test set. To evaluate the model's predictive ability on materials like Si and SiO2, it is best to directly make predictions on these materials and assess the model's performance. Therefore, the performance of the training data typically cannot serve as a sufficient basis for predicting the model's performance on unknown materials.

```
parity_plots_side_by_side(y_test,Test_predictions,y_test,Test_predicti
ons2, title_left="Test Parity Plot (Default Model)",title_right="Test
Parity Plot (Optimized Model)") # build both plots
parity_stats_side_by_side(y_test,Test_predictions,y_test,Test_predicti
ons2,"Test Set (Default Model)","Test Set (Optimized Model)")
```





	Error Metric	Test Set (Default Model)	Test Set (Optimized Model)	Note
0	RMSE	1.0492 (eV)	0.6062 (eV)	(0.0 for perfect prediction)
1	RMSE/std	0.4884	0.2822	(0.0 for perfect prediction)
2	MAE	0.6811 (eV)	0.4776 (eV)	(0.0 for perfect prediction)
3	R2	0.7614	0.9204	(1.0 for perfect prediction)

Answer

1. Just looking at the testing data statistics, does the optimized model do better or worse at predicting the testing data?

   Looking at the testing data statistics, the optimized model performs better at predicting the testing data. The optimized model has a testing data RMSE of 0.6062 eV, whereas the default model has a testing data RMSE of 1.0492 eV. Additionally, the optimized model outperforms the default model in terms of testing data RMSE/std, MAE, and R2.

2. Compare the difference between train and test RMSE for the default and optimized model. Did the difference between training and test performance increase or decrease after hyperparameter optimization?

   The difference in RMSE between the training and test sets for the default model is 1.0492 - 0.0003 = 1.0489 (eV), while for the optimized model, it is 0.6062 - 0.4226 = 0.1836 (eV). The difference between training and test RMSE decreased after hyperparameter optimization. Specifically, the difference decreased from approximately 1.0489 (eV) for the default model to approximately 0.1836 (eV) for the optimized model.

3. Is this evidence that the optimized model is more overfit or less overfit?

   This decrease in the difference between training and test RMSE suggests that the optimized model is less overfit compared to the default model. The smaller gap between training and test performance indicates improved generalization of the optimized model to unseen data, which is evidence of reduced overfitting.

Answer

- Lower Test RMSE: The optimized model has a lower Test RMSE (0.6062 eV) compared to the default model (1.0492 eV). Lower RMSE indicates that the optimized model provides more accurate predictions on the test data, which is a critical criterion for model selection.
- Reduced Overfitting: The optimized model exhibits a smaller difference between the Training and Test RMSE (0.1836 eV) compared to the default model (1.0489 eV). This suggests that the optimized model is less prone to overfitting and generalizes better to unseen data, which is an essential factor for model reliability.
- Higher Test R2 Score: The optimized model has a higher Test R2 score (0.9204) compared to the default model (0.7614). A higher R2 score indicates that the optimized model explains a larger portion of the variance in the test data, signifying its superior predictive power.

Remember back when we first trained the model and predicted Silicon and Silica? Let do the same thing for fun with the optimized model. The values have likely shifted.

When we fit the DT3 model we use the X_predict and y_predict versions of the dataset in which we removed 5 compounds so that we could predict them now. Note that these predictions are a bit artificial because when we did the model optimization this data was included. In a true research environment this isn't something you'd want to do.

```
# fit model to all data except for the values we want to predict.
DT3 = CV.best_estimator_.fit(X_predict,y_predict)
```

Edit the cell below to change which compound is predicted between: Silicon, Silica, Salt, Diamond, and Tin

Change the Prediction_features object to one of the following:
xpredict_Si
xpredict_SiO2
xpredict_NaCl
xpredict_C
xpredict_Sn

```
### MAKE EDITS BELOW HERE ###

Prediction_features = xpredict_Si
```

```
### MAKE EDITS ABOVE HERE ###

# make a prediction with the trained DT3 model
print("Predicted Band Gap: ",DT3.predict(Prediction_features))
```

```
Predicted Band Gap:  [1.4877]
```

Now for our final test on model performance. We are going to take the individual predictions on our Test data set and quantify how often the model succeeded or failed in making predictions for both the Solar application and Wide Band Gap application. Below we've rearranged the existing data from the parity plots in the previous section and printed it explicitly so we can look in more detail.

In doing this we are viewing the results of this regression model through the lens of classificaiton. Essentially the materials with known values in a certain range will be viewed as one class of materials, and everything else as another class. We'll then assess how well the model does at correctly identifying these classes of materials. If you want you read up on the background related to a few of these metrics you can look into the metrics precision and recall for binary classifiers. During the exercises below we'll walk through the process of calculating the recall for this pseudo-classification model.

```
# combine previous data into one dataframe for visualization
predictions_combined =
pd.DataFrame(list(zip(y_test,Test_predictions2)),columns=['test','pred
ictions'])

# sort on the Test values from low to high
predictions_combined.sort_values("test").head(10)
```

	test	predictions
31	0.064	0.79062
39	0.100	0.76934
10	0.100	0.72560
0	0.170	0.49400
21	0.200	2.28480
20	0.200	0.74369
37	0.230	0.70072
7	0.270	0.66842
9	0.310	0.42176
43	0.332	0.63554

Answer

1. In the Test dataset how many materials do we have with band gaps within the range of being a good solar material? Note in terms of classification we are identifying the number of positive cases in the dataset.

2. Divide this number of true positives by the total number of positive cases (from question 1) to obtain the recall value. What is the recall of our pseudo-classifier to predict single-junction solar materials?

   2 is the number of true positives; recall value = 2 / 7 = 0.2857

3. In the Test dataset how many materials do we have with band gaps at or above 3 eV?

   9

4. Perform the same process from question 2 (remember are classes are now defined differently for this new task) and calculate the recall for predicting high bandgap materials. What is the recall in this case?

   9 is the number of true positives; recall value = 9 / 9 = 1

5. Based on the evidence from questions 1-4 which tasks can the model succeed at?

   The model can identify all high bandgap materials (band gaps at or above 3 eV) with a recall of 1, indicating perfect performance in this task. The model can predict single-junction solar materials (band gaps between 1.1 eV and 1.7 eV) with a recall of approximately 0.2857, which is relatively low but still provides some predictive capability. So, the model succeeds in identifying high bandgap materials but has limited success in predicting single-junction solar materials.

Submitting for MSE 760

To submit the answers to the lab you can save the entire notebook as a pdf by printing the entire webpage to pdf. The shortcut to do this is

ctrl+P or cmd+P

This should capture your answers below as well as all of the outputs and plots above. Then simply upload the pdf file to Canvas.

The End

So, are we officially done? What's next in the machine learning workflow?

- hopefully you have your research problem down at this point. or else, figure out what you need to know, and whether ML can help with that
- go back and reiterate on hyperparameter tuning
- use a different model
  - get uncertainty estimate on your prediction by going from (decision) tree to (random) forest
- redo data cleaning/featurization

- get more data

what should you do next as a student?

if this is only interesting to you, and you don't plan to do ML yourself in the near future: solidify the big ideas and key takeaways.

if you want to get hands-on with ML:

- think about your data
- go through the lab again and figure out each line of code
- change parameters and do all the challenges
- read the docs for software packages such as scikit-learn or mastml that help us perform these machine learning workflows.

Answers

- Back to TOC

Submit answers for the code exercises.

Back to Exercise 1.1

Here my where you put an answer to Q1.1.

Back to Exercise 1.2

Skip this S23.

Back to Exercise 1.3

Back to Exercise 1.4

Back to Exercise 1.5

Skip this S23.

Back to Exercise 2.1

Back to Exercise 2.2

Back to Exercise 3.1

Back to Exercise 3.2

Back to Exercise 4.1

Back to Exercise 5.1

Back to Exercise 5.2

Back to Exercise 5.3

Back to Exercise 6.1

Back to Exercise 6.2

Back to Exercise 6.3

Back to Exercise 6.4

Back to Exercise 6.5

Back to Exercise 7.1