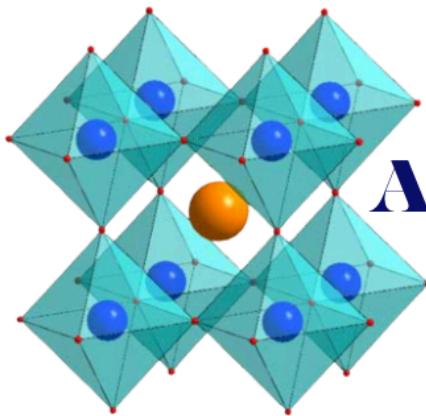


# Machine Learning Molecular Structure Predictor



## ABX<sub>3</sub> Laboratories

### Overview

This project explores different machine learning algorithms to predict the lowest distortion in perovskite structures, which is a critical aspect in ceramic science, materials physics, and solid-state inorganic chemistry. Researchers have identified a total of 73 elements in the A and B cation sites of ABO<sub>3</sub> structures, leading to numerous oxides of the perovskite type.

The objective is to develop models capable of accurately classifying perovskite structures based on features such as electronegativity, ionic radius, valence, and bond lengths of A-O and B-O pairs, thereby aiding in the prediction and optimization of material properties. Model performance is assessed using the Accuracy Score metric.

### Business Problem

Here we will take a look at the [Kaggle Crystal Structure Dataset] (<https://www.kaggle.com/datasets/meetnagadia/crystal-structure-dataset/data> (<https://www.kaggle.com/datasets/meetnagadia/crystal-structure-dataset/data>)). Each record represents a different Perovskite structures based on their characteristics. The data consists of 4,165 ABO<sub>3</sub> perovskite-type oxides. Each observation is described by 13 feature columns and 1 class column which identifies it to be either a cubic, tetragonal, orthorhombic, and rhombohedral structure.

A Research Laboratory has asked for an analysis of a dataset to help understand the relationships between various physical and chemical properties of perovskite materials and their crystal structure types.

This project will determine:

1. If there are any discernable patterns or correlations among the different variables and perovskite structure types?
2. How well features like the Glodschmidt tolerance factor, the electronegativity, and octahedral factor contribute to the predictive accuracy of the classification model?

3. Are there any limitations or challenges in interpreting the model's predictions based on the selected features?



## Data Understanding

The data consists of 4,165  $\text{ABO}_3$  perovskite-type oxides. Each observation is described by 13 feature columns and 1 class column which identifies it to be either a cubic, tetragonal, orthorhombic, and rhombohedral structure.(2)

**Missing Values:** In the case of data quality, missing or inconsistent data were handled on a case by case basis. Some considerations made were the relative importance of the variable in the analysis, how much data was needed, and finally, the possibility of filling data with a central measure of tendency and the statistical implications.

## Data Preparation (EDA)

In [1]:

```
1 # Import the necessary packages
2
3 #Third Party Packages
4 import pandas as pd
5 import plotly.express as px
6 import seaborn as sns
7 import matplotlib.pyplot as plt
8 from sklearn.tree import DecisionTreeClassifier
9 from sklearn.model_selection import *
10 from sklearn.metrics import *
11 from sklearn.model_selection import GridSearchCV
12 from sklearn.metrics import confusion_matrix
13 from sklearn.metrics import accuracy_score
14 from sklearn.metrics import roc_curve, roc_auc_score
15 import numpy as np
16 from sklearn.ensemble import RandomForestClassifier
17 import time
18 from sklearn.svm import SVC
19 from sklearn.model_selection import cross_val_score
20 from sklearn.metrics import classification_report
21 from sklearn.inspection import permutation_importance
22 from sklearn.metrics import ConfusionMatrixDisplay
23
```

In [2]:

```
1 # Create DataFrame
2 prv_train = pd.read_csv("Data/Perovskite_train.csv")
3
4 # Display
5 print("Total number of results:", len(prv_train))
6 display(prv_train.head()) # Using display instead of print leads to a neater output
7
```

Total number of results: 4165

	v(A)	v(B)	r(AXII) (Å)	r(AVI) (Å)	r(BVI) (Å)	EN(A)	EN(B)	I(A-O)(Å)	I(B-O)(Å)	ΔENR	tG	τ
0	0	0	0.52	0.52	0.93	2.18	2.54	2.214685	2.313698	-1.728214	0.582680	0.000000
1	0	0	1.03	0.86	0.60	1.27	1.90	2.500930	0.000000	-1.768643	0.859135	0.000000
2	2	4	0.92	0.67	0.53	1.83	1.88	2.290644	1.930311	-1.468464	0.849994	4.936558
3	1	5	1.64	1.38	0.62	0.82	2.36	3.025719	1.745600	-1.974429	1.064161	3.977376
4	0	0	0.57	0.57	0.71	2.20	1.30	2.300109	2.027412	-1.622357	0.660190	0.000000

```
In [3]: 1 # Rename columns in prv_train DataFrame
2 #(pep8 standard, a line of code should not exceed 79 characters
3 #so each renaming is it's own line)
4 prv_train = prv_train.rename(columns={'v(A)': 'vA'})
5 prv_train = prv_train.rename(columns={'v(B)': 'vB'})
6 prv_train = prv_train.rename(columns={'r(AXII)(Å)': 'r_A12'})
7 prv_train = prv_train.rename(columns={'r(AVI)(Å)': 'r_A6'})
8 prv_train = prv_train.rename(columns={'r(BVI)(Å)': 'r_B6'})
9 prv_train = prv_train.rename(columns={'EN(A)': 'EN_A'})
10 prv_train = prv_train.rename(columns={'EN(B)': 'EN_B'})
11 prv_train = prv_train.rename(columns={'l(A-O)(Å)': 'bond_len_A0'})
12 prv_train = prv_train.rename(columns={'l(B-O)(Å)': 'bond_len_B0'})
13 prv_train = prv_train.rename(columns={'ΔENR': 'ENR_diff'})
14 prv_train = prv_train.rename(columns={'τ': 'tau'})
15 prv_train = prv_train.rename(columns={'μ': 'mu'})
16 prv_train = prv_train.rename(columns={'Lowest distortion': 'lowest_distortion'})
17
```

```
In [4]: 1 # Get summary
2 display(prv_train.info())
#   column          non-null count  dtype
---  -----
0   vA              4165 non-null  int64
1   vB              4165 non-null  int64
2   r_A12             4165 non-null  float64
3   r_A6              4165 non-null  float64
4   r_B6              4165 non-null  float64
5   EN_A              4165 non-null  float64
6   EN_B              4165 non-null  float64
7   bond_len_A0        4165 non-null  float64
8   bond_len_B0        4165 non-null  float64
9   ENR_diff            4165 non-null  float64
10  tG                  4165 non-null  float64
11  tau                 4165 non-null  float64
12  mu                  4165 non-null  float64
13  lowest_distortion    4165 non-null  object
dtypes: float64(11), int64(2), object(1)
memory usage: 455.7+ KB
```

None

List of Feature (or attributes, either work) Descriptions

- vA : Valence of A
- vB : Valence of B
- r\_A6 : ionic radius of A cation (A6)
- r\_A12 : ionic radius of A cation (a second one, A12)
- r\_B6 : ionic radius of B cation
- EN\_A : Average electronegativity value of A cation
- EN\_B : Average electronegativity value of B cation
- bond\_len\_A0 : Bond length of A-O pair
- bond\_len\_B0 : Bond length of B-O pair
- ENR\_diff : Electronegativity difference with radius
- tG : Goldschmidt tolerance factor
- tau : New tolerance factor

- mu : Octahedral factor

This is a multiclass, imbalanced dataset. The cubic structure is far more common than any other dataset.

$$\tau = \frac{r_A + r_B}{\sqrt{2}(r_O) + l_{B-O} - l_{A-O}}$$

- $r_A$  is the ionic radius of the A-site cation.
- $r_B$  is the ionic radius of the B-site cation.
- $r_O$  is the radius of the oxygen ion.
- $l_{A-O}$  is the bond length of the A-O pair.
- $l_{B-O}$  is the bond length of the B-O pair.

Mathematically, the Goldschmidt tolerance factor ( $t_G$ ) is given by:

$$t_G = \frac{r_A + r_B}{2(r_O)}$$

- $(r_A)$  is the ionic radius of the A-site cation.
- $(r_B)$  is the ionic radius of the B-site cation.
- $(r_O)$  is the radius of the oxygen ion.

Side quest (because it is interesting). The above formulas give context, which is extra data that many times won't be available.

Taking a closer look at the variables:

The only empirical data from the crystal structures here are bond lengths and bond angles .

Some may consider valence electron counts, but locking the atoms into formal charges when such things are assigned by convention might disrupt the algorithms (or not, charges probably aren't far off given it's oxygen).

Ionic radii are also, to some degree, just average values without someone/something to determine where those radii actually extend to across different combinations.

Electronegativity could be treated as an empirical parameter provided the different ways of obtaining those numbers don't produce significantly different results (collection methods were not provided) - in which case there would be a nice electronegativity/ionic radius predictor from enough data.

$t_G$  ,  $\tau$  , and  $\mu$  are all things that should be able to be predicted from the structural parameters. It is suspected that  $t_G$  and  $\tau$  should be something that would fall out of a large dataset analysis easily.

```
In [5]: 1 # Select columns with data type 'object'  
2 string_col = prv_train.select_dtypes(include="object").columns  
3  
4 # Convert selected columns to the 'string' data type  
5 prv_train[string_col] = prv_train[string_col].astype("string")  
6
```

```
In [6]: 1 # Get summary  
2 display(prv_train.info())
```

```
<class 'pandas.core.frame.DataFrame'>  
RangeIndex: 4165 entries, 0 to 4164  
Data columns (total 14 columns):  
 #   Column           Non-Null Count  Dtype     
---  --  
 0   vA               4165 non-null   int64    
 1   vB               4165 non-null   int64    
 2   r_A12             4165 non-null   float64  
 3   r_A6               4165 non-null   float64  
 4   r_B6               4165 non-null   float64  
 5   EN_A              4165 non-null   float64  
 6   EN_B              4165 non-null   float64  
 7   bond_len_A0       4165 non-null   float64  
 8   bond_len_B0       4165 non-null   float64  
 9   ENR_diff           4165 non-null   float64  
 10  tG                4165 non-null   float64  
 11  tau               4165 non-null   float64  
 12  mu                4165 non-null   float64  
 13  lowest_distortion 4165 non-null   string  
dtypes: float64(11), int64(2), string(1)  
memory usage: 455.7 KB
```

None

The object data has been converted to string.

It appears there are two measurements for the radius of the A cation, labeled as "r(AXII)(Å)" and "r(AVI)(Å)". This duplication could be due to different conventions or sources of data.

In short, there are two columns due to atomic arrangements; factors such as the types of ions present, their sizes, and their positions within the crystal lattice can influence the overall size of the perovskite structure. These variations in atomic arrangements and compositions can lead to different radii for perovskite structures.

Unfortunately, no atomic data is provided.

```
In [7]: 1 # Create a copy of the DataFrame to avoid altering the original  
2 prv_train_num = prv_train.copy()  
3  
4 # Select columns with data type 'string'  
5 string_col=prv_train_num.select_dtypes("string").columns.to_list()
```

```
In [8]: 1 # Extract column names of prv_train_num DataFrame and store them in num_col
2 num_col = prv_train_num.columns.to_list()
3
4
5 # Remove the column "Lowest distortion" from num_col
6 num_col.remove("lowest_distortion")
7 print(num_col)
8
```

['vA', 'vB', 'r\_A12', 'r\_A6', 'r\_B6', 'EN\_A', 'EN\_B', 'bond\_len\_A0', 'bond\_len\_BO', 'ENR\_diff', 'tG', 'tau', 'mu']

For this project, it doesn't really remove anything, but if there were hundreds of columns then the strings would get in the way of the data describe and so generally they would be removed this way before describing the data.

This line generates a summary statistics table for the DataFrame `prv_train_num`. The `.describe()` method calculates summary statistics for numerical columns by default. The `.T` at the end is a transpose operation, which swaps the rows and columns of the summary statistics table. This operation is performed to make the table more readable, with variables as rows and summary statistics as columns. This is often done to have a better visual representation of the data, especially when there are many columns, as is the case with this data set.

```
In [9]: 1 prv_train_num.describe().T
```

Out[9]:

	count	mean	std	min	25%	50%	75%	max
<b>vA</b>	4165.0	0.882353	1.672111	-1.000000	-1.000000	0.000000	2.000000	5.000000
<b>vB</b>	4165.0	1.385114	2.161707	-1.000000	-1.000000	0.000000	3.000000	5.000000
<b>r_A12</b>	4165.0	1.005570	0.336940	0.270000	0.730000	0.980000	1.240000	1.880000
<b>r_A6</b>	4165.0	0.841789	0.240549	0.270000	0.670000	0.860000	0.960000	1.670000
<b>r_B6</b>	4165.0	0.801999	0.234891	0.270000	0.630000	0.760000	0.940000	1.670000
<b>EN_A</b>	4165.0	1.547309	0.447277	0.790000	1.200000	1.500000	1.910000	2.540000
<b>EN_B</b>	4165.0	1.599059	0.442961	0.790000	1.220000	1.600000	1.960000	2.540000
<b>bond_len_AO</b>	4165.0	2.323011	0.635699	0.000000	2.294004	2.422517	2.582926	3.300176
<b>bond_len_BO</b>	4165.0	2.017219	0.555599	0.000000	1.956808	2.075849	2.284156	3.009747
<b>ENR_diff</b>	4165.0	-2.186093	0.636470	-5.411536	-2.570036	-2.084214	-1.734643	-0.601714
<b>tG</b>	4165.0	0.781040	0.135816	0.384648	0.680809	0.776944	0.873123	1.321062
<b>tau</b>	4165.0	1.332182	20.804362	-480.827696	0.000000	0.000000	3.892903	305.871348
<b>mu</b>	4165.0	0.572857	0.167778	0.192857	0.450000	0.542857	0.671429	1.192857

Looking at Tau, the zeros could be a problem. The max value is also 305 (which may likely be an outlier) with a listed mean of 1.3. Further investigation around this is needed. This column may need to be dropped.

EN(A) and EN(B) have the same values indicating that they are distributed the same.

Some of the bond lengths are displaying zero and will need to be investigated further (domain knowledge says that this is impossible).

If the valence(A) and (B) are both zero, in this dataset, it implies that both cations (A and B) are essentially neutral, meaning they have no charge. This scenario is highly unlikely and not physically meaningful in the context of perovskite compounds because A and B cations typically have non-zero valence states to contribute to the overall charge balance of the compound.

Therefore, in this dataset, the occurrence of both A and B having zero valence is likely an artifact or error rather than a meaningful data point. It's important to review the data and ensure its integrity, an occurrence like this may indicate issues with data collection or processing. These occurrences may have to be dropped.

```
In [10]: 1 # Count the number of zeros in each column of the training dataset
2 zero_counts = prv_train.apply(lambda col: (col == 0).sum())
3 print("Count of zeros in each column:")
4 print(zero_counts)
```

Count of zeros in each column:

vA	859
vB	859
r_A12	0
r_A6	0
r_B6	0
EN_A	0
EN_B	0
bond_len_A0	254
bond_len_B0	243
ENR_diff	0
tG	0
tau	2177
mu	0
lowest_distortion	0
dtype:	int64

The tau column, representing the New tolerance factor, is essential because:

**Structural Stability Assessment:** The New tolerance factor tau provides insight into the structural stability of perovskite compounds. It considers the ionic radii of the A and B cations and the oxygen ion as well as the bond lengths between these ions. By incorporating bond lengths, tau offers a comprehensive assessment of the compound's structural stability compared to the traditional Goldschmidt tolerance factor t\_G .

So it would be worth our time to investigate the effectiveness of the model both with and without Tau.

```

In [11]: 1 # Create a copy of the DataFrame to avoid altering the original
2 prv_train_tau = prv_train.copy()
3
4 # Drop rows with zero values
5 prv_train_tau = prv_train_tau[prv_train_tau['tau'] != 0]
6
7 # Display
8 print("Total number of results:", len(prv_train_tau))
9 display(prv_train_tau.head())

```

Total number of results: 1988

	vA	vB	r_A12	r_A6	r_B6	EN_A	EN_B	bond_len_AO	bond_len_BO	ENR_diff	tG	tau
2	2	4	0.92	0.67	0.53	1.83	1.88	2.290644	1.930311	-1.468464	0.849994	4.936558
3	1	5	1.64	1.38	0.62	0.82	2.36	3.025719	1.745600	-1.974429	1.064161	3.977376
6	1	5	0.92	0.80	0.62	1.78	2.36	2.392362	1.745600	-1.484143	0.812123	5.017992
10	3	3	1.36	1.03	0.90	1.10	1.23	2.689119	2.258656	-2.931286	0.848528	3.536265
11	2	4	1.19	0.99	0.61	1.22	1.54	2.550861	1.927849	-2.040571	0.911148	4.133678

```

In [12]: 1 # Count the number of zeros in each column of the training dataset with tol
2 zero_counts = prv_train_tau.apply(lambda col: (col == 0).sum())
3 print("Count of zeros in each column:")
4 print(zero_counts)

```

Count of zeros in each column:

vA	0
vB	0
r_A12	0
r_A6	0
r_B6	0
EN_A	0
EN_B	0
bond_len_AO	0
bond_len_BO	0
ENR_diff	0
tG	0
tau	0
mu	0
lowest_distortion	0
dtype:	int64

```

In [13]: 1 # Create a copy of the DataFrame to avoid altering the original
2 prv_train_no_tau = prv_train.copy()
3
4 # Drop tau column
5 prv_train_no_tau.drop(['tau'], axis=1, inplace=True)
6
7 # Display
8 print("Total number of results:", len(prv_train_no_tau))
9 display(prv_train_no_tau.head())

```

Total number of results: 4165

	vA	vB	r_A12	r_A6	r_B6	EN_A	EN_B	bond_len_AO	bond_len_BO	ENR_diff	tG	mu	lo
0	0	0	0.52	0.52	0.93	2.18	2.54	2.214685	2.313698	-1.728214	0.582680	0.664286	
1	0	0	1.03	0.86	0.60	1.27	1.90	2.500930	0.000000	-1.768643	0.859135	0.428571	
2	2	4	0.92	0.67	0.53	1.83	1.88	2.290644	1.930311	-1.468464	0.849994	0.378571	
3	1	5	1.64	1.38	0.62	0.82	2.36	3.025719	1.745600	-1.974429	1.064161	0.442857	
4	0	0	0.57	0.57	0.71	2.20	1.30	2.300109	2.027412	-1.622357	0.660190	0.507143	

```

In [14]: 1 # Count the number of zeros in each column of the training dataset without
2 zero_counts = prv_train_no_tau.apply(lambda col: (col == 0).sum())
3
4 # Print a message indicating the count of zeros in each column.
5 print("Count of zeros in each column:")
6
7 # Print the variable zero_counts, which contains the count of zeros in each
8 # of the DataFrame prv_train_no_tau.
9 print(zero_counts)
10
11

```

Count of zeros in each column:

vA	859
vB	859
r_A12	0
r_A6	0
r_B6	0
EN_A	0
EN_B	0
bond_len_AO	254
bond_len_BO	243
ENR_diff	0
tG	0
mu	0
lowest_distortion	0
dtype:	int64

For consideration: Should these zero values be left to skew the data to be incorrect or should they be dropped?

vA and vB are crucial factors so zero values for those features would really not give any meaningful results. The dataset could be split here but then there would be multiple analyses on various datasets and not evaluations of various machine learning models. So these values will be dropped to limit the number of datasets and maximize the number of machine learning models.

```
In [15]: 1 # Drop rows with zero values
2 prv_train_no_tau = prv_train_no_tau[prv_train_no_tau['vA'] != 0]
3
4 # Display
5 print("Total number of results:", len(prv_train_no_tau))
6 display(prv_train_no_tau.head())
```

Total number of results: 3306

	vA	vB	r_A12	r_A6	r_B6	EN_A	EN_B	bond_len_AO	bond_len_BO	ENR_diff	tG	mu	k
2	2	4	0.92	0.67	0.53	1.83	1.88	2.290644	1.930311	-1.468464	0.849994	0.378571	
3	1	5	1.64	1.38	0.62	0.82	2.36	3.025719	1.745600	-1.974429	1.064161	0.442857	
6	1	5	0.92	0.80	0.62	1.78	2.36	2.392362	1.745600	-1.484143	0.812123	0.442857	
10	3	3	1.36	1.03	0.90	1.10	1.23	2.689119	2.258656	-2.931286	0.848528	0.642857	
11	2	4	1.19	0.99	0.61	1.22	1.54	2.550861	1.927849	-2.040571	0.911148	0.435714	

```
In [16]: 1 # Count the number of zeros in each column of the training dataset without
2 zero_counts = prv_train_no_tau.apply(lambda col: (col == 0).sum())
3 print("Count of zeros in each column:")
4 print(zero_counts)
```

Count of zeros in each column:

```
vA          0
vB          0
r_A12       0
r_A6         0
r_B6         0
EN_A         0
EN_B         0
bond_len_AO  0
bond_len_BO  0
ENR_diff     0
tG          0
mu          0
lowest_distortion  0
dtype: int64
```

Where vA was zero, it appears that vB and also bond\_len\_AO and bond\_len\_BO are all also zero. So only the zeros from vA needed to be dropped.

In [17]:

```
1 # Accessing -1 values
2 prv_train_no_tau.loc[prv_train_no_tau['vA'] == -1]
3
```

Out[17]:

	vA	vB	r_A12	r_A6	r_B6	EN_A	EN_B	bond_len_AO	bond_len_BO	ENR_diff	tG	mu
20	-1	-1	1.24	0.96	0.99	1.17	1.22	2.611965	2.255581	-3.002036	0.781072	0.707143
21	-1	-1	1.70	1.20	0.80	1.62	1.78	2.531967	2.147681	-2.510714	0.996378	0.571429
23	-1	-1	0.27	0.27	0.27	2.04	2.04	1.891523	1.624662	-0.601714	0.707107	0.192857
24	-1	-1	0.63	0.63	0.90	2.20	2.02	2.311295	2.215655	-1.823786	0.624099	0.642857
30	-1	-1	0.90	0.63	1.52	1.88	0.82	2.233563	2.825997	-3.556429	0.556968	1.085714
...	...	...	...	...	...	...	...	...	...	...	...	...
4137	-1	-1	0.68	0.68	0.90	2.05	2.02	2.361079	2.215655	-1.897357	0.639470	0.642857
4142	-1	-1	0.45	0.45	1.11	1.57	2.00	2.178342	2.323655	-2.336464	0.521174	0.792857
4143	-1	-1	0.87	0.75	0.62	1.36	1.55	2.417029	2.169994	-1.809536	0.794620	0.442857
4146	-1	-1	1.03	0.86	0.77	1.27	2.10	2.500930	2.115492	-2.035679	0.791829	0.550000
4148	-1	-1	0.54	0.54	0.94	1.61	1.30	2.186789	2.246056	-2.181929	0.586234	0.671429

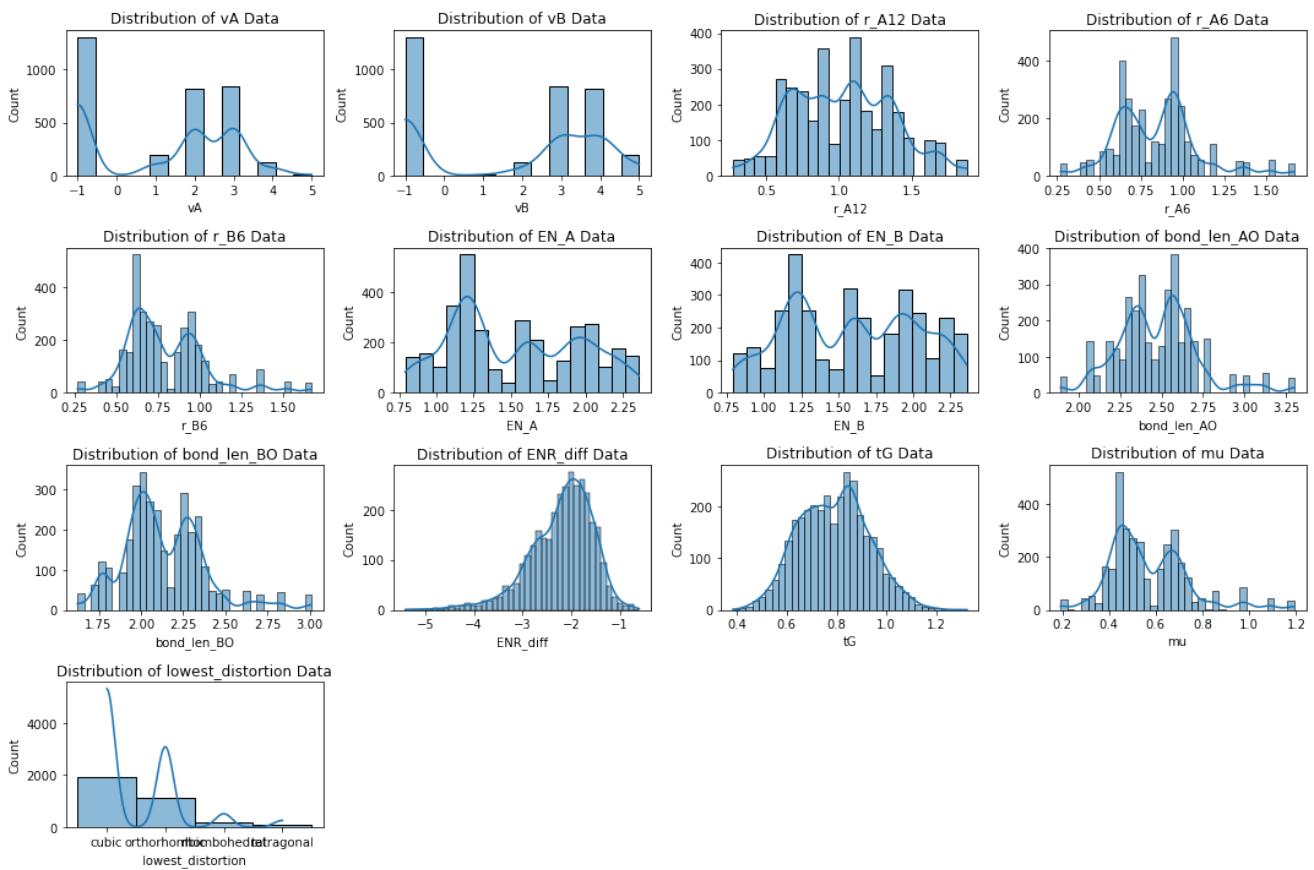
1299 rows × 13 columns

This might be two data sets combined. All perovskites contain two positively charged ions (A and B). Positively charged ions are called cations and are bonded to negatively charged atoms. A negative value for a valence electron count would indicate an overall negative charge. Since the compound will have a balance of charges, this value may indicate some inconsistency.

Data set 1: has valence electrons reported for each entry and a tau calculation. Data set 2: doesn't contain valence electrons or a calculated tau value.

In [18]:

```
1 # Set up the figure size for the plot
2 plt.figure(figsize=(15,10))
3
4 # Iterate over each column in the dataframe
5 for i, col in enumerate(prv_train_no_tau.columns, 1):
6     # Create subplots in a 4x4 grid
7     plt.subplot(4, 4, i)
8
9     # Set title for each subplot indicating the distribution of data for the
10    plt.title(f"Distribution of {col} Data")
11
12    # Plot the histogram with kernel density estimation using seaborn's his-
13    sns.histplot(prv_train_no_tau[col], kde=True)
14
15    # Adjust subplot layout
16    plt.tight_layout()
17
18    # Ensure the plot is shown
19    plt.plot()
20
```



The valence electrons should add up to 6. A '-1' value seems to be used as a placeholder. Drop all rows with a -1 value.

```
In [19]: 1 # Drop rows with -1 values
2 prv_train_no_tau = prv_train_no_tau[prv_train_no_tau['vA'] != -1]
3
4 # Display
5 print("Total number of results:", len(prv_train_no_tau))
6 display(prv_train_no_tau.head())
```

Total number of results: 2007

vA	vB	r_A12	r_A6	r_B6	EN_A	EN_B	bond_len_AO	bond_len_BO	ENR_diff	tG	mu	k
2	2	4	0.92	0.67	0.53	1.83	1.88	2.290644	1.930311	-1.468464	0.849994	0.378571
3	1	5	1.64	1.38	0.62	0.82	2.36	3.025719	1.745600	-1.974429	1.064161	0.442857
6	1	5	0.92	0.80	0.62	1.78	2.36	2.392362	1.745600	-1.484143	0.812123	0.442857
10	3	3	1.36	1.03	0.90	1.10	1.23	2.689119	2.258656	-2.931286	0.848528	0.642857
11	2	4	1.19	0.99	0.61	1.22	1.54	2.550861	1.927849	-2.040571	0.911148	0.435714

Next, let's take a look at maximizing the number of data points. What would the data set look like if the valence electron columns and the tau column were dropped? Would there be any zeros left in the data set?

```
In [20]: 1 # Create a copy of the DataFrame to avoid altering the original
2 prv_train_no_tau_no_v = prv_train.copy()
3
4 # Drop tau column
5 prv_train_no_tau_no_v.drop(['tau', 'vA', 'vB'], axis=1, inplace=True)
6
7 # Display
8 print("Total number of results:", len(prv_train_no_tau_no_v))
9 display(prv_train_no_tau_no_v.head())
```

Total number of results: 4165

r_A12	r_A6	r_B6	EN_A	EN_B	bond_len_AO	bond_len_BO	ENR_diff	tG	mu	lowest_dist
0	0.52	0.52	0.93	2.18	2.54	2.214685	2.313698	-1.728214	0.582680	0.664286
1	1.03	0.86	0.60	1.27	1.90	2.500930	0.000000	-1.768643	0.859135	0.428571
2	0.92	0.67	0.53	1.83	1.88	2.290644	1.930311	-1.468464	0.849994	0.378571
3	1.64	1.38	0.62	0.82	2.36	3.025719	1.745600	-1.974429	1.064161	0.442857
4	0.57	0.57	0.71	2.20	1.30	2.300109	2.027412	-1.622357	0.660190	0.507143

```
In [21]: 1 # Count the number of zeros in each column of the training dataset without
2 zero_counts = prv_train_no_tau_no_v.apply(lambda col: (col == 0).sum())
3 print("Count of zeros in each column:")
4 print(zero_counts)
```

Count of zeros in each column:

r_A12	0
r_A6	0
r_B6	0
EN_A	0
EN_B	0
bond_len_AO	254
bond_len_BO	243
ENR_diff	0
tG	0
mu	0
lowest_distortion	0
dtype:	int64

Domain knowledge says that bond lengths cannot be zero so these values must be dropped. Also note that the bond length counts are different and may not occur in the same rows. The number of zero values is low so it is acceptable to drop these values.

```
In [22]: 1 # Drop rows with 0 values
2 prv_train_no_tau_no_v = prv_train_no_tau_no_v[prv_train_no_tau_no_v['bond_1']
3 prv_train_no_tau_no_v = prv_train_no_tau_no_v[prv_train_no_tau_no_v['bond_1'
4
5 # Display
6 print("Total number of results:", len(prv_train_no_tau_no_v))
7 display(prv_train_no_tau_no_v.head())
8
```

Total number of results: 3682

	r_A12	r_A6	r_B6	EN_A	EN_B	bond_len_AO	bond_len_BO	ENR_diff	tG	mu	lowest_dist
0	0.52	0.52	0.93	2.18	2.54	2.214685	2.313698	-1.728214	0.582680	0.664286	
2	0.92	0.67	0.53	1.83	1.88	2.290644	1.930311	-1.468464	0.849994	0.378571	
3	1.64	1.38	0.62	0.82	2.36	3.025719	1.745600	-1.974429	1.064161	0.442857	orthorh
4	0.57	0.57	0.71	2.20	1.30	2.300109	2.027412	-1.622357	0.660190	0.507143	
6	0.92	0.80	0.62	1.78	2.36	2.392362	1.745600	-1.484143	0.812123	0.442857	

```
In [23]: 1 # Count the number of zeros in each column of the training dataset without  
2 zero_counts = prv_train_no_tau_no_v.apply(lambda col: (col == 0).sum())  
3 print("Count of zeros in each column:")  
4 print(zero_counts)
```

```
Count of zeros in each column:
```

```
r_A12          0  
r_A6           0  
r_B6           0  
EN_A           0  
EN_B           0  
bond_len_A0    0  
bond_len_B0    0  
ENR_diff       0  
tG             0  
mu             0  
lowest_distortion 0  
dtype: int64
```

## Choosing a final data set to train

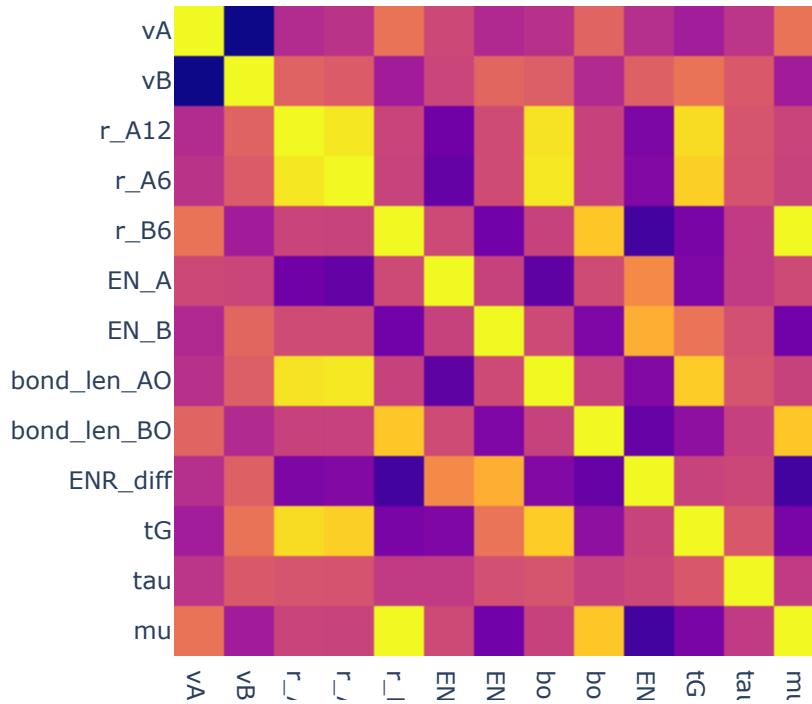
### Comparing the perovskite data with tau and without tau.

In this section, it's really important to both consider removing columns or rows and how that impacts the integrity and completeness of the dataset. Specifically, we are considering different data set combinations that make use of different columns because that changes which rows are included in the dataset.

## 1. Correlation Plots

```
In [24]: 1 # Create a correlation plot of the training dataset with tolerance factor
2 px.imshow(prv_train_tau.corr(),title="Correlation Plot of Perovskite Data v
```

Correlation Plot of Perovskite Data with Tau



Some notes on the correlation matrix:

Strong Correlations:

Some variables exhibit strong correlations. For instance, there is a perfect negative correlation (-1) between vA and vB, indicating that they are inversely related. Similarly, r\_A12 and r\_A6, which likely represent different aspects of the same quantity, show a strong positive correlation (0.917346), suggesting they move in the same direction. Variables such as bond\_len\_AO and bond\_len\_BO also show strong positive correlations (0.923351), implying a strong relationship between the bond lengths of A-O and B-O pairs.

Moderate Correlations:

There are moderate correlations between certain variables, such as tG and bond\_len\_AO (0.785924) or tG and r\_A12 (0.863578). These correlations suggest some degree of relationship between the variables but may not be as strong as the highly correlated ones.

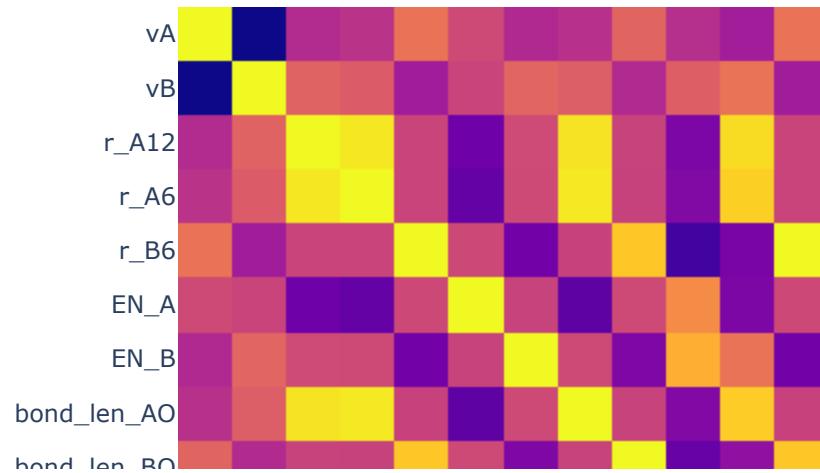
## Weak Correlations:

Some variables exhibit weak correlations or no significant correlation at all. For example, the correlation between EN\_A and EN\_B is relatively low (-0.034935), indicating a weak relationship between the electronegativities of A and B cations.

Correlation does not imply causation. While correlated variables may move together, it does not necessarily mean that changes in one variable cause changes in the other. Understanding the physical or chemical relationship between variables can provide insights into why certain correlations exist and their implications, such as in the case of the valence electrons.

```
In [25]: 1 # Create a correlation plot of the training dataset with tolerance factor
2 px.imshow(prv_train_no_tau.corr(),title="Correlation Plot of Perovskite Dat
```

Correlation Plot of Perovskite Data without Tau



In this correlation matrix, several noteworthy observations can be made:

### Strong Positive Correlations:

Variables such as vA and vB (0.813366), r\_A12 and r\_A6 (0.935433), bond\_len\_AO and bond\_len\_BO (0.938676), and bond\_len\_BO and EN\_diff (0.907513) exhibit strong positive correlations. These correlations suggest that changes in one variable tend to be associated with similar changes in the other variable. It is interesting that the EN\_diff and the bond\_len\_BO are directly related while the EN\_diff and the EN\_A are inversely (see next statement).

### Strong Negative Correlations:

There are also strong negative correlations observed, such as between r\_B6 and tG (-0.842273) or between EN\_diff and EN\_A (-0.842273). These negative correlations suggest an inverse relationship between the variables.

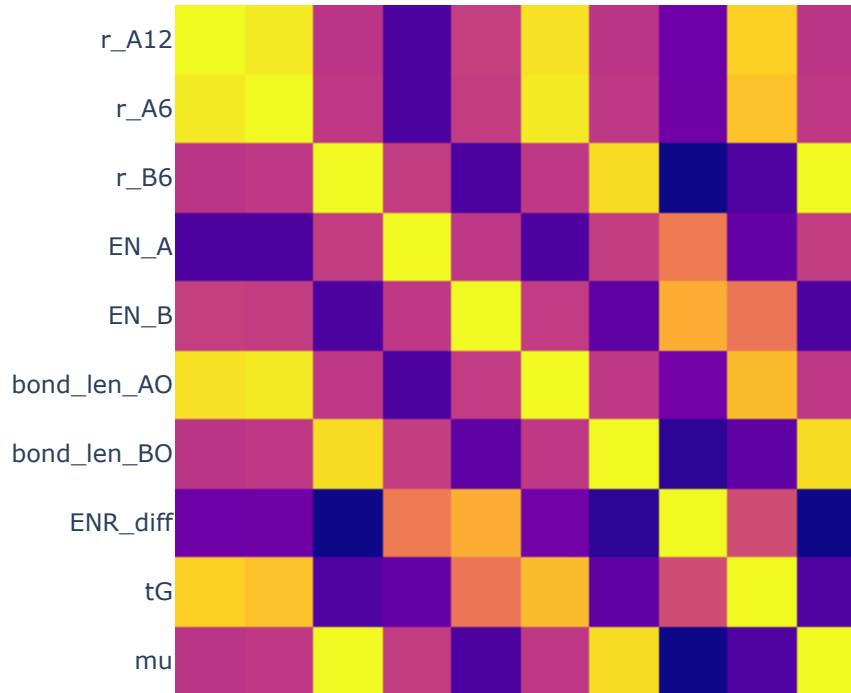
### Moderate Correlations:

Moderate correlations are present between various pairs of variables, such as vA and tG (0.215130), vB and tG (0.342345), bond\_len\_AO and r\_A12 (0.905104), or EN\_B and EN\_diff (0.674963). These moderate correlations indicate a discernible but not excessively strong relationship between the

variables

```
In [26]: te1a correlation plot of the training dataset without tolerance factor (tau) and without valence electrons (v).  
In [26]: corr = prv_train_no_tau_no_v.corr()  
In [26]: corr
```

Correlation Plot of Perovskite Data without Tau and without Valence Electrons



#### Strong Positive Correlations:

Variables such as r\_A12 and r\_A6 (0.934533) exhibit a strong positive correlation. This suggests that changes in the radius at site A with a coordination number of 12 are strongly associated with similar changes in the radius at site A with a coordination number of 6, moving in the same direction. Another strong positive correlation is observed between bond\_len\_AO and bond\_len\_BO (0.894372), indicating that changes in the bond lengths between atoms A and oxygen are strongly associated with changes in the bond lengths between atoms B and oxygen, moving in the same direction.

#### Strong Negative Correlations:

A strong negative correlation is seen between r\_B6 and tG (-0.584644). This suggests an inverse relationship, where increases in the radius at site B with a coordination number of 6 are associated with decreases in the value of parameter tG, and vice versa.

#### Moderate Correlations:

Moderate correlations are present between various pairs of variables, such as between r\_A12 and

## **Summary of the three matrices:**

### Matrix 1 (with Tau):

This matrix includes all the variables along with the Tau variable. Strong positive correlations are observed between variables like vA and vB, bond\_len\_AO and r\_A12, tG and r\_A12, etc. Strong negative correlations are seen between EN\_A and r\_A12, EN\_diff and r\_B6, mu and r\_B6, etc. The Tau variable shows moderate correlations with other variables. Overall, this matrix provides comprehensive information about the correlations, including Tau, with other variables.

### Matrix 2 (without Tau):

Initially, Tau was excluded because of the large number of zeros in the data. This matrix is similar to Matrix 1 but excludes the Tau variable. The correlations remain quite similar to those in Matrix 1, except for the absence of correlations involving Tau. Strong positive and negative correlations between other variables remain intact. The removal of the Tau variable reduces the complexity of the matrix, making it easier to interpret if Tau is not a focus of the analysis.

### Matrix 3 (without Tau and without Valence Electron Data):

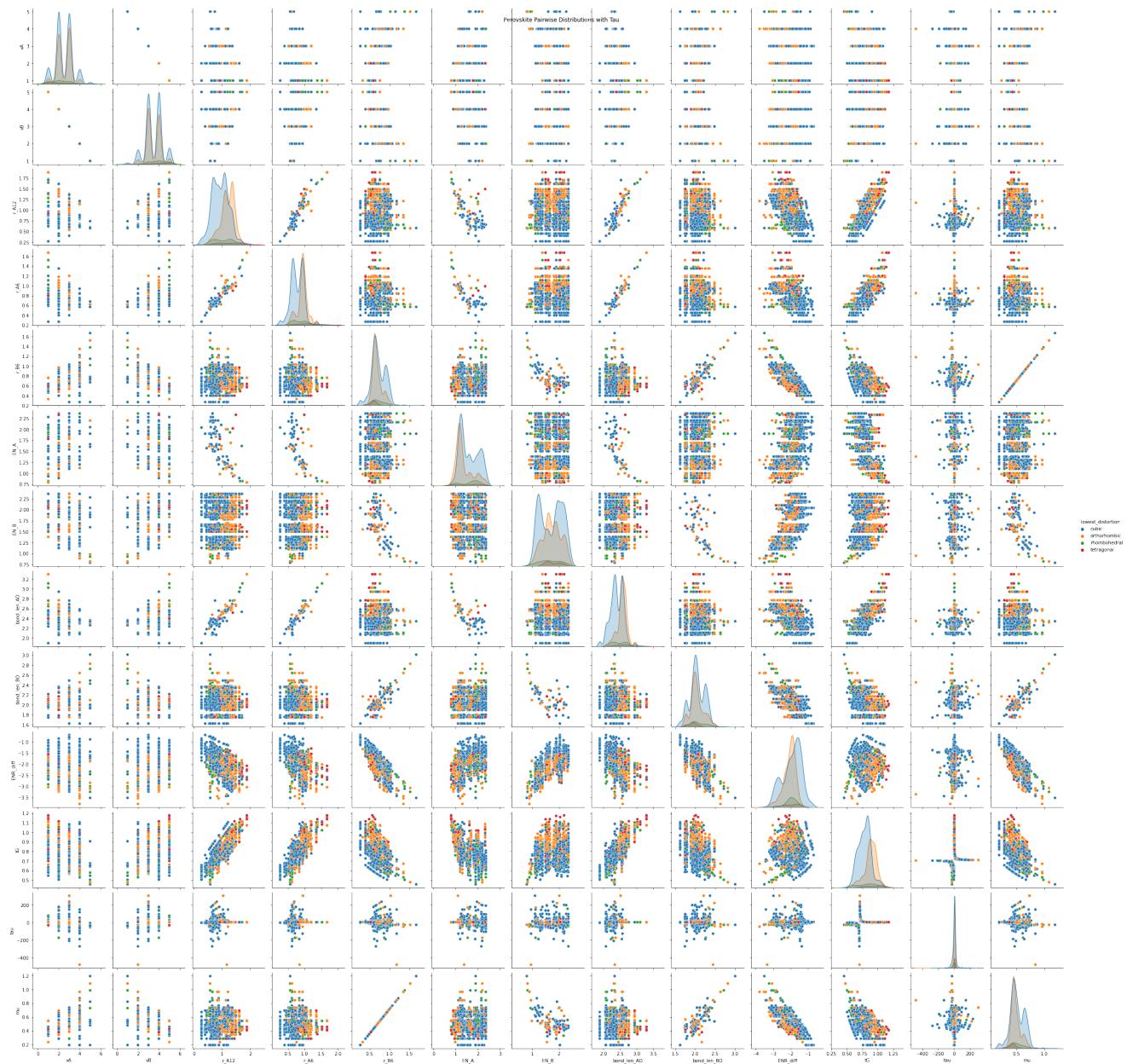
This matrix excludes both the Tau variable and the Valence Electron data in order to preserve more rows of data. The correlations presented here are limited to the remaining variables: r\_A12, r\_A6, r\_B6, EN\_A, EN\_B, bond\_len\_AO, bond\_len\_BO, EN\_diff, tG, and mu. Strong positive and negative correlations observed in the previous matrices persist in this matrix as well. This matrix provides a simplified view of the correlations among the selected variables, excluding additional information such as Tau and Valence Electron data.

Matrix 1 provides the most comprehensive view, Matrix 2 is useful when excluding Tau from consideration, and Matrix 3 offers a simplified view focusing only on the core variables.

## Pair plots

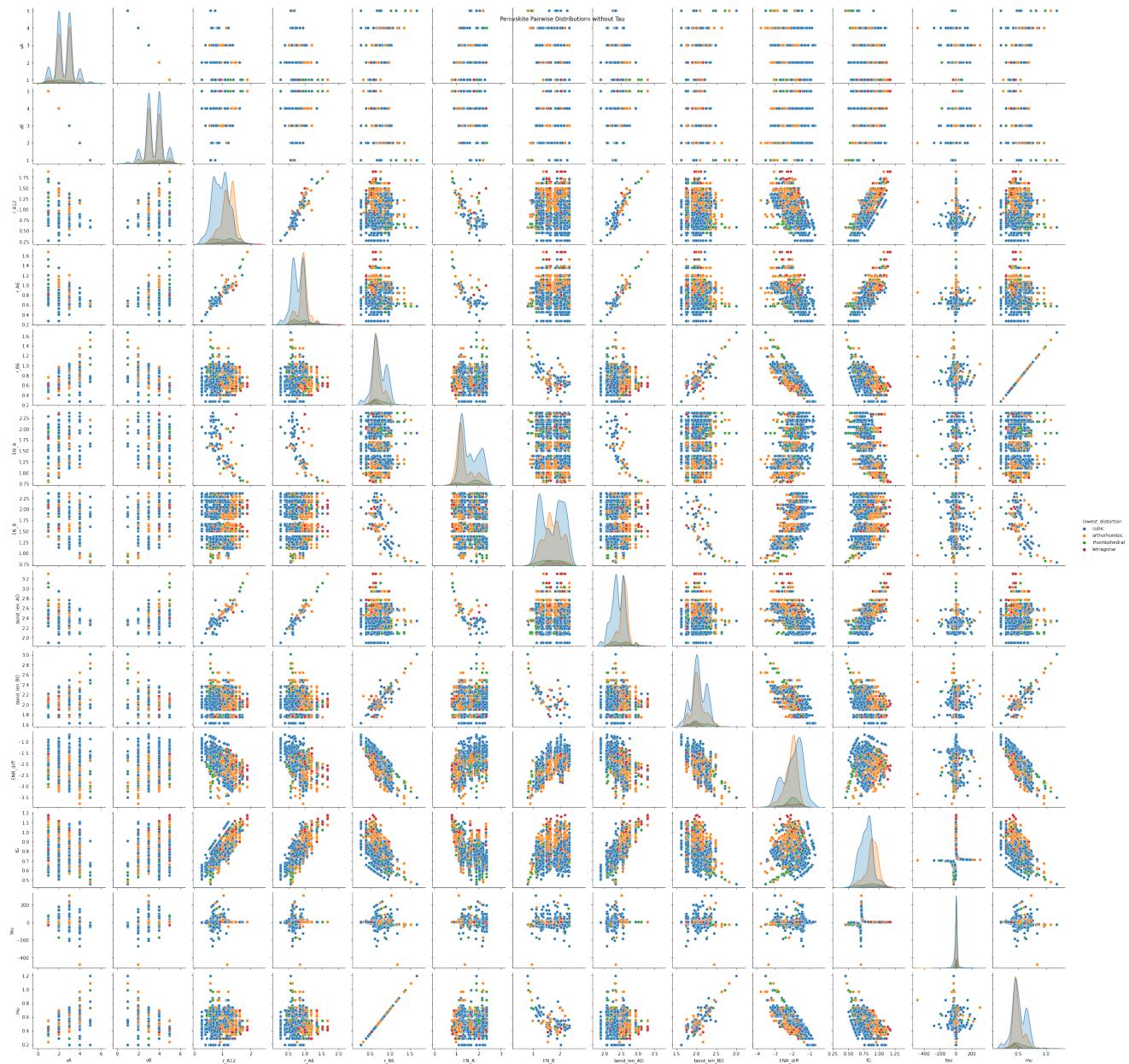
In [27]:

```
1 # Create a pairplot of the training dataset with tolerance factor (tau), di
2 pair_plot = sns.pairplot(prv_train_tau, hue="lowest_distortion")
3
4 # Set the title of the pairplot
5 pair_plot.fig.suptitle("Perovskite Pairwise Distributions with Tau")
6
7 # Save the pairplot as an image file
8 pair_plot.savefig("pairplot_perovskite_train_tau.jpg")
9
10 # Display the pairplot
11 plt.show()
12
```

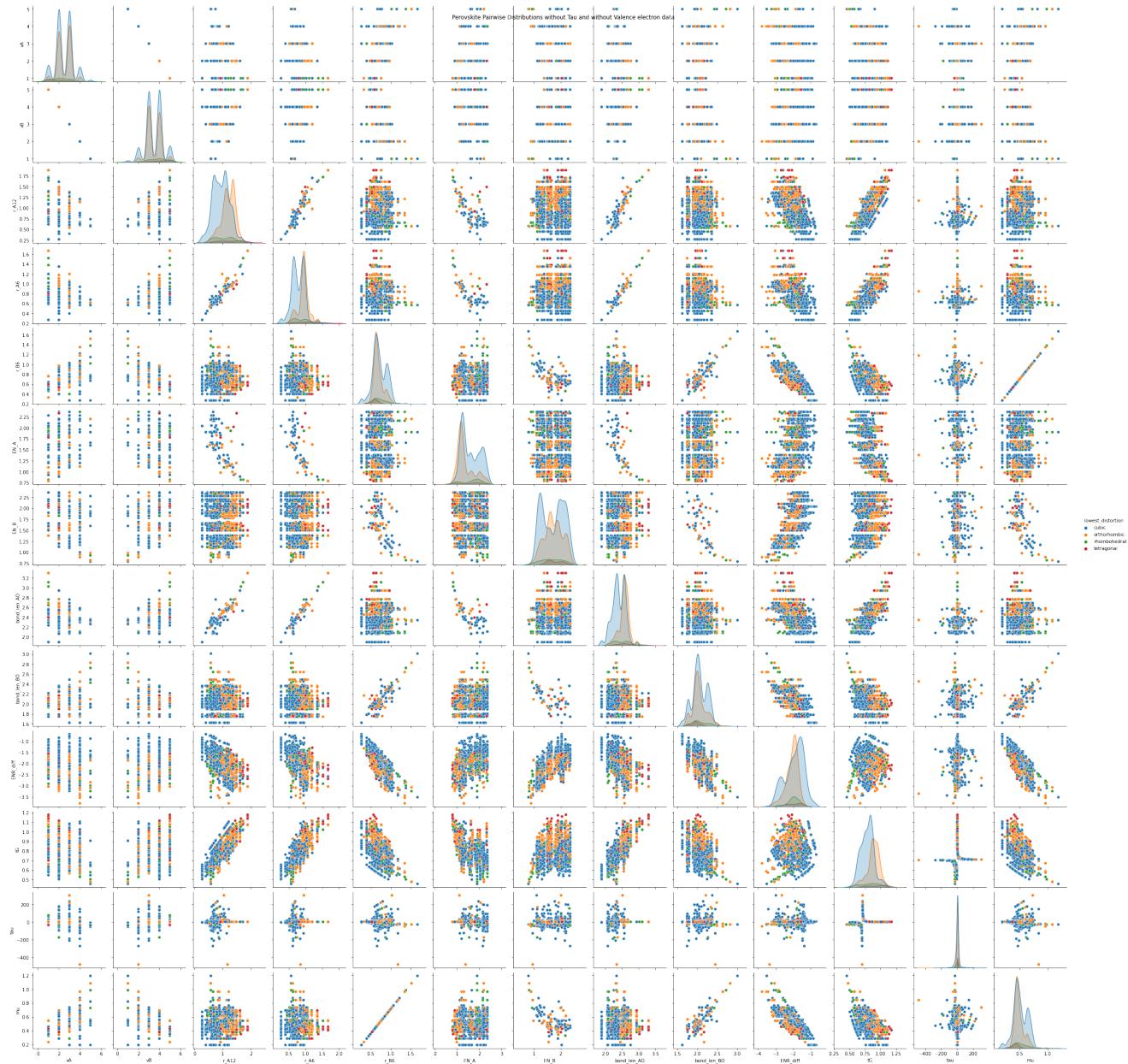


In [28]:

```
1 # Create a pair plot of the training dataset without tolerance factor (tau)
2 pair_plot = sns.pairplot(prv_train_tau, hue="lowest_distortion")
3
4 # Set the title for the pair plot
5 pair_plot.fig.suptitle("Perovskite Pairwise Distributions without Tau")
6
7 # Save the pair plot as an image
8 pair_plot.savefig("pairplot_perovskite_train_without_tau.jpg")
9
10 # Display the pair plot
11 plt.show()
12
```



```
In [29]: 1 # Create a pair plot of the training dataset without tolerance factor (tau)
2 pair_plot = sns.pairplot(prv_train_tau, hue="lowest_distortion")
3
4 # Set the title for the pair plot
5 pair_plot.fig.suptitle("Perovskite Pairwise Distributions without Tau and v")
6
7 # Save the pair plot as an image
8 pair_plot.savefig("pairplot_perovskite_train_without_tau_Ve.jpg")
9
10 # Display the pair plot
11 plt.show()
12
```



## Summary

The pair plots are useful in quickly visualizing any data that may be problematic due to a directly linear relationship that would confuse the model. (multivariate)

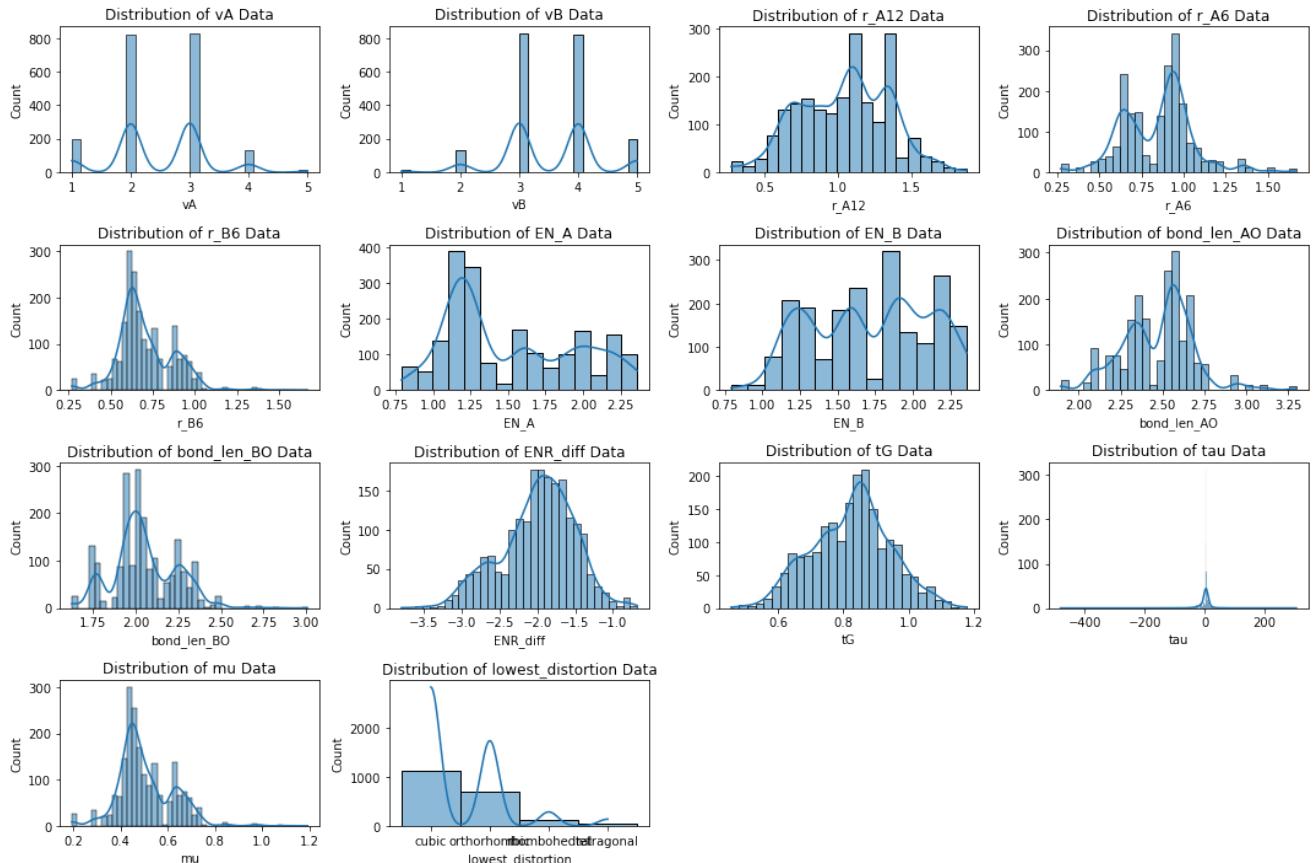
It is a good practice to plot distribution graphs and look for skewness of features. Kernel density estimate (kde) is a useful tool for plotting the shape of a distribution. (univariate data view, zooming in).

No new or interesting correlations are noted from the pairplot distributions.

## Histograms

In [30]:

```
1 # Set up the figure size for the plot
2 plt.figure(figsize=(15,10))
3
4 # Iterate over each column in the dataframe
5 for i, col in enumerate(prv_train_tau.columns, 1):
6     # Create subplots in a 4x4 grid
7     plt.subplot(4, 4, i)
8
9     # Set title for each subplot indicating the distribution of data for the
10    plt.title(f"Distribution of {col} Data")
11
12    # Plot the histogram with kernel density estimation using seaborn's his-
13    sns.histplot(prv_train_tau[col], kde=True)
14
15    # Adjust subplot layout
16    plt.tight_layout()
17
18    # Ensure the plot is shown
19    plt.plot()
20
```



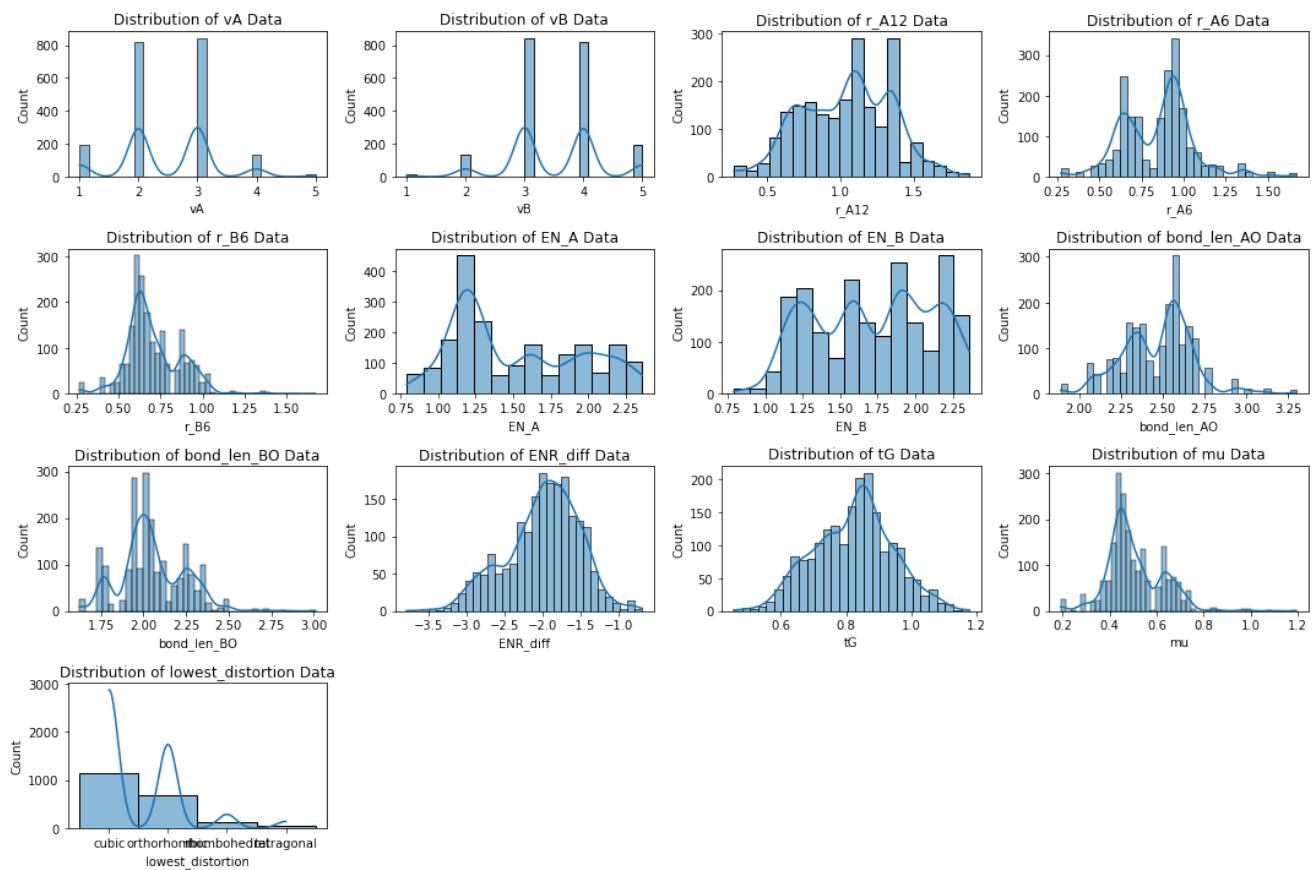
The valence electron data is showing a discrete pattern as expected and otherwise no notable outliers or abnormal distributions.

In [31]:

```

1 # Set up the figure size for the plot
2 plt.figure(figsize=(15,10))
3
4 # Iterate over each column in the dataframe
5 for i, col in enumerate(prv_train_no_tau.columns, 1):
6     # Create subplots in a 4x4 grid
7     plt.subplot(4, 4, i)
8
9     # Set title for each subplot indicating the distribution of data for the
10    plt.title(f"Distribution of {col} Data")
11
12    # Plot the histogram with kernel density estimation using seaborn's his-
13    sns.histplot(prv_train_no_tau[col], kde=True)
14
15    # Adjust subplot layout
16    plt.tight_layout()
17
18    # Ensure the plot is shown
19    plt.plot()
20

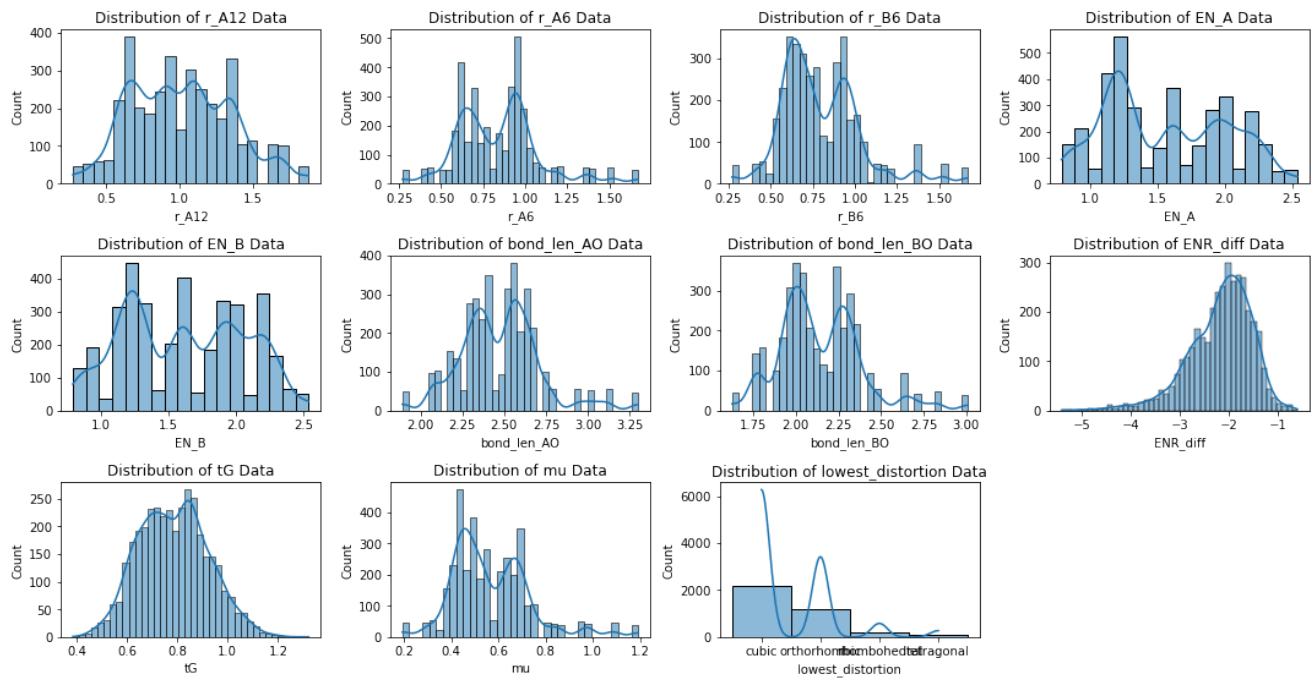
```



```

In [32]: 1 # Set up the figure size for the plot
2 plt.figure(figsize=(15,10))
3
4 # Iterate over each column in the dataframe
5 for i, col in enumerate(prv_train_no_tau_no_v.columns, 1):
6     # Create subplots in a 4x4 grid
7     plt.subplot(4, 4, i)
8
9     # Set title for each subplot indicating the distribution of data for the
10    plt.title(f"Distribution of {col} Data")
11
12    # Plot the histogram with kernel density estimation using seaborn's his
13    sns.histplot(prv_train_no_tau_no_v[col], kde=True)
14
15    # Adjust subplot layout
16    plt.tight_layout()
17
18    # Ensure the plot is shown
19    plt.plot()

```



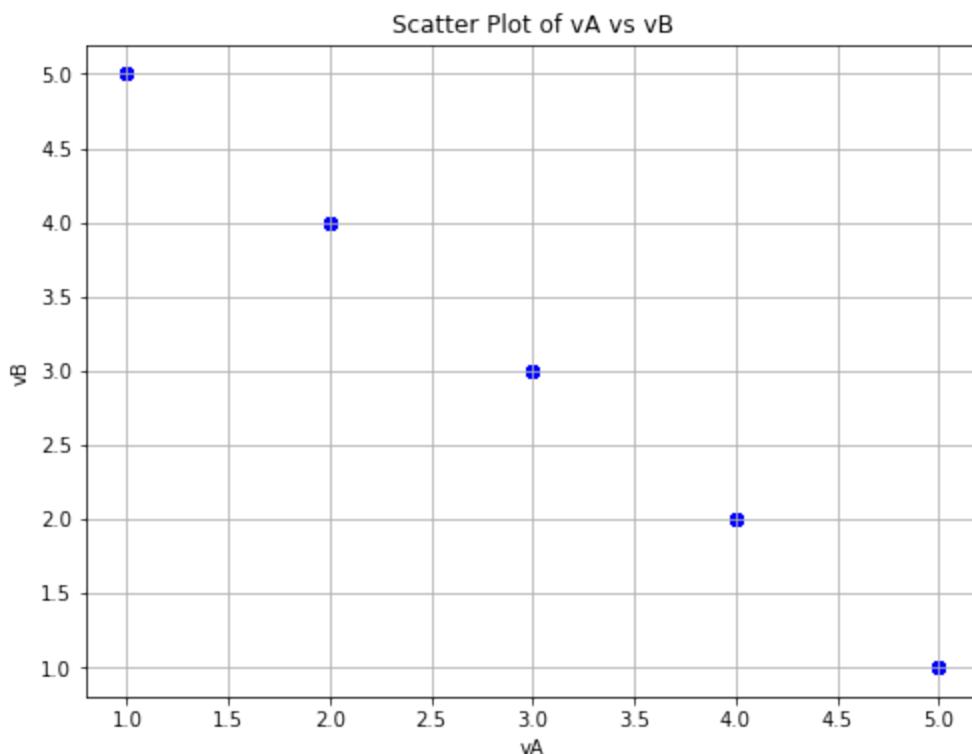
## Summary

At this point, there doesn't seem to be a need to standardize or normalize the data.

Since the data already follows a Gaussian (normal) distribution, normalization might not provide significant benefits in terms of scaling the data to a fixed range. With this normally distributed data, standardization is a better choice, but doesn't seem necessary. Standardization could be beneficial for maintaining the integrity of the data's distribution while making it more compatible with algorithms that expect standardized inputs.

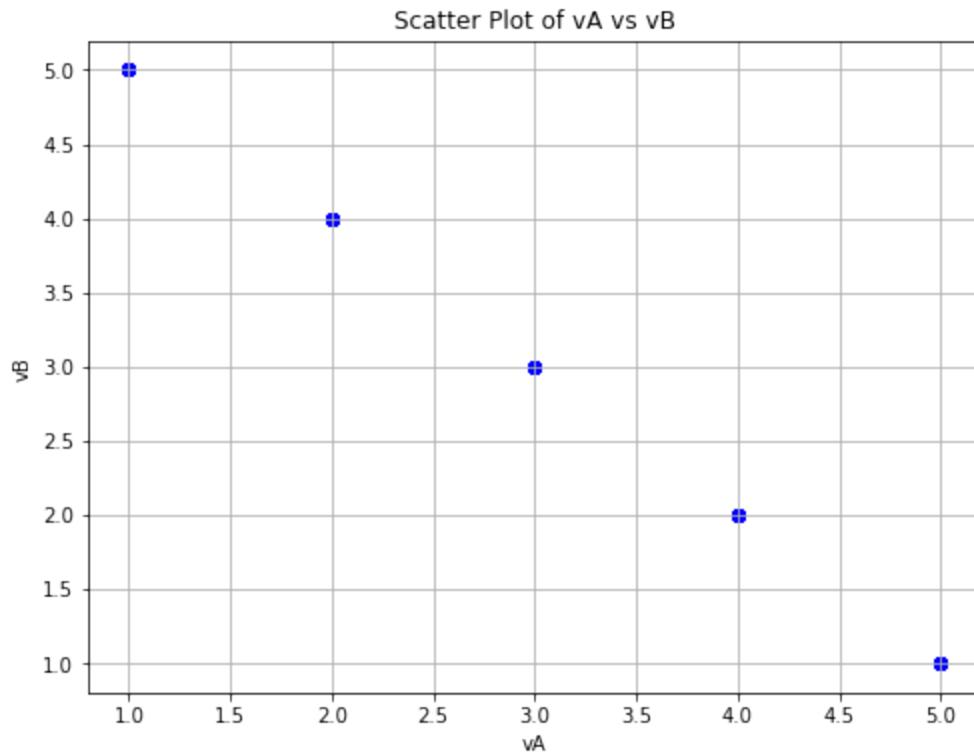
## Safety Check

```
In [33]: 1 # Extracting the 'vA' and 'vB' columns from the training dataset
2 vA = prv_train_tau['vA']
3 vB = prv_train_tau['vB']
4
5 # Creating a scatter plot of 'vA' vs 'vB'
6 plt.figure(figsize=(8, 6))
7 plt.scatter(vA, vB, color='blue', alpha=0.5)
8
9 # Setting the labels for x and y axes
10 plt.xlabel('vA')
11 plt.ylabel('vB')
12
13 # Setting the title for the scatter plot
14 plt.title('Scatter Plot of vA vs vB')
15
16 # Displaying the grid
17 plt.grid(True)
18
19 # Showing the scatter plot
20 plt.show()
21
```



In [34]:

```
1 # Extracting the 'vA' and 'vB' columns from the training dataset without tau
2 vA = prv_train_no_tau['vA']
3 vB = prv_train_no_tau['vB']
4
5 # Creating a scatter plot of 'vA' vs 'vB'
6 plt.figure(figsize=(8, 6))
7 plt.scatter(vA, vB, color='blue', alpha=0.5)
8
9 # Setting the labels for x and y axes
10 plt.xlabel('vA')
11 plt.ylabel('vB')
12
13 # Setting the title for the scatter plot
14 plt.title('Scatter Plot of vA vs vB')
15
16 # Displaying the grid
17 plt.grid(True)
18
19 # Showing the scatter plot
20 plt.show()
21
```



These plots show the linear relationship between the valence electrons. This explicitly shows that vA and vB add up to the constant 6.

## Final dataset choice:

It appears at this point that the dataset that is the most comprehensive would be best suited to answer our shareholders questions. The most comprehensive data set is the set that includes tau.

# Perform a Train-Test Split

```
In [35]: 1 # Extracting features (X_tau) and labels (y_tau) from the training dataset
2 X_tau = prv_train_tau.loc[:, 'vA':'mu']
3 y_tau = prv_train_tau.loc[:, 'lowest_distortion']
4
5 # Splitting the dataset into training and testing sets
6 # X contains features and y contains labels/targets
7 # X_train and y_train will contain the training data, while X_test and y_te
8 X_train, X_test, y_train, y_test = train_test_split(X_tau, y_tau, test_size
9
```

Here, `test_size=0.2` specifies that 20% of the data will be held out for testing. The `random_state` is set for reproducibility, ensuring the same split is obtained each time the code is run.

The test data must not be touched.

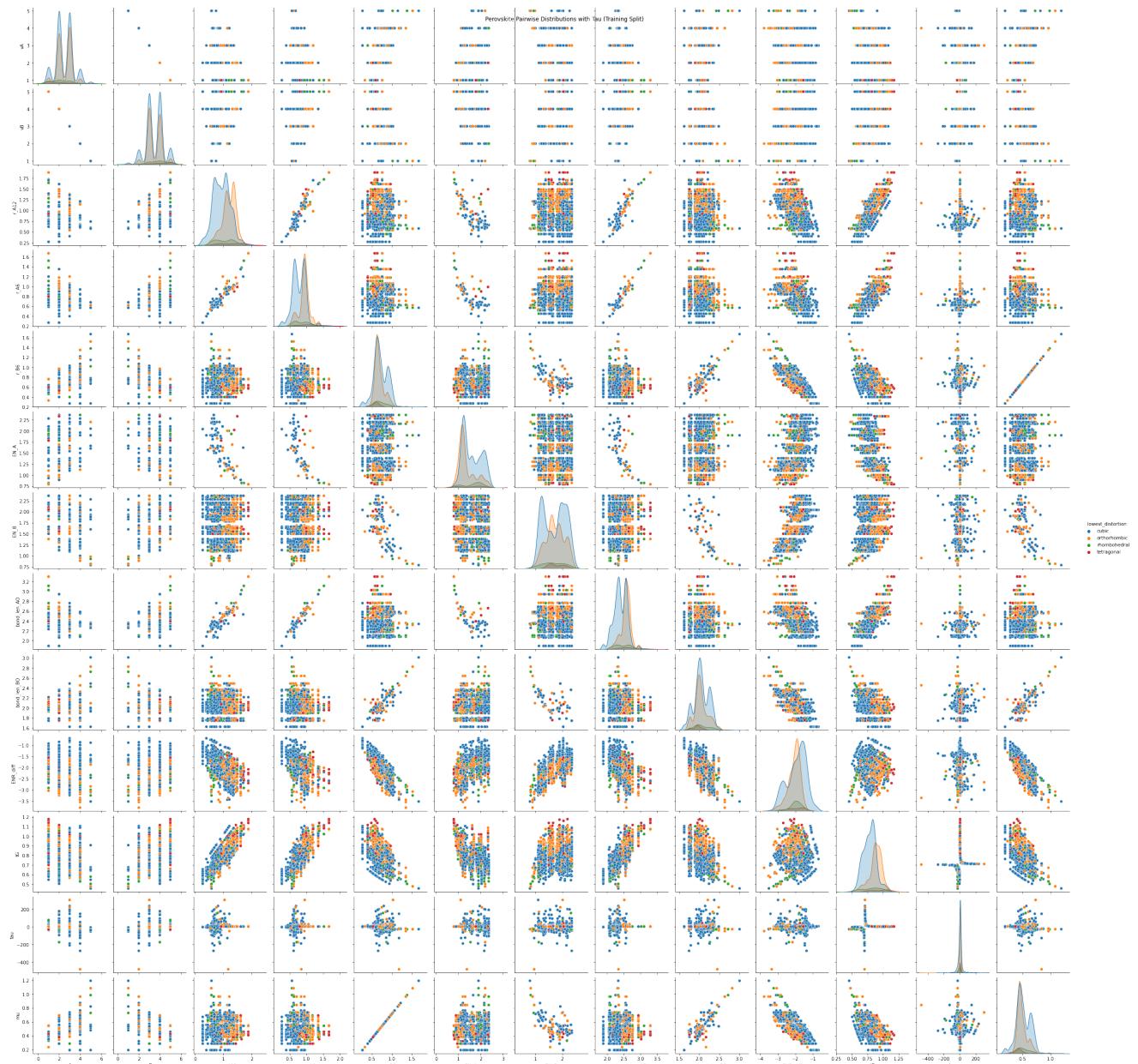
```
In [36]: 1 # Checking the lengths of the training and testing sets
2 len(X_train), len(X_test)
```

Out[36]: (1590, 398)

Verify that everything makes sense after the test-train split

In [37]: *in1and y\_train are training feature and target variables*

```
2
3     reate X_train and y_train along columns
4     data = pd.concat([X_train, y_train], axis=1)
5
6     ting a pair plot to visualize pairwise distributions of features in the trainin
7     lot = sns.pairplot(prv_train_tau, hue="lowest_distortion")
8
9     ng9a title to the pair plot
10    lot.fig.suptitle("Perovskite Pairwise Distributions with Tau (Training Split)")
11
12    ng12the pair plot as an image
13    lot.savefig("pairplot_perovskite_tau_training_split.jpg")
14
15    laying the pair plot
16
17
```



This is the primary indicator showing linear relationships that need to be monitored. Most linear relationships are not good for ML models because they often oversimplify the underlying patterns in the data and may not capture complex, non-linear relationships effectively (case by case basis, keep an eye on).

Asserting the dimensionality checks is not needed to verify in this case the size of the datasets (what they would asserted to be), because the lengths of the data confirmed that the test-train split did what it was told to do.

## **Decision Tree Model with Stratified K-Fold Cross-Validation**

A decision tree provides a good balance between interpretability, flexibility, and performance, making it a suitable choice for exploring the dataset and gaining insights before potentially exploring more complex models.

In [38]:

```
# Initialize lists to store scores for each fold
precision_scores = []
recall_scores = []
f1_scores = []
accuracy_scores = []
6
#Step 1, start training
# Initialize StratifiedKFold with 5 splits for cross-validation
kf = StratifiedKFold(n_splits=5)
10
# Iterate over each fold
11for fold, (trn_, val_) in enumerate(kf.split(X=X_train, y=y_train)):
13
14    # Split the data into training and validation sets for the current fold
15    X_train_Kfold = X_train.iloc[trn_]
16    y_train_Kfold = y_train.iloc[trn_]
17
18    X_valid = X_train.iloc[val_]
19    y_valid = y_train.iloc[val_]
20
21    # Initialize a Decision Tree classifier with specified hyperparameters
22    DTclf = DecisionTreeClassifier(criterion="entropy", max_depth = 10, random_
23
24    # Train the Decision Tree classifier on the training data for the current
25    DTclf.fit(X_train_Kfold, y_train_Kfold)
26
27    # Predict labels for the validation set
28    y_pred_Kfold = DTclf.predict(X_valid)
29
30    # Calculate scores for the current fold
31    report = classification_report(y_valid, y_pred_Kfold, output_dict=True)
32
33    # Append precision, recall, F1-score, and accuracy scores to respective l
34    precision_scores.append(report['weighted avg']['precision'])
35    recall_scores.append(report['weighted avg']['recall'])
36    f1_scores.append(report['weighted avg']['f1-score'])
37    accuracy_scores.append(report['accuracy'])
38    # Print classification report for the current fold
39    print(f"The fold is: {fold}:")
40    print(classification_report(y_valid, y_pred_Kfold))
41
42    # Calculate average scores across all folds
43    avg_precision = np.mean(precision_scores)
44    avg_recall = np.mean(recall_scores)
45    avg_f1 = np.mean(f1_scores)
46    avg_accuracy = np.mean(accuracy_scores)
47
48    # Print average evaluation metrics across all folds
49    print(f"Avg Precision: {avg_precision}")
50    print(f"Avg Recall: {avg_recall}")
51    print(f"Avg F1-score: {avg_f1}")
52    print(f"Avg Accuracy: {avg_accuracy}")
53
```

The fold is: 0:

	precision	recall	f1-score	support
cubic	0.85	0.83	0.84	178
orthorhombic	0.69	0.77	0.72	113
rhombohedral	0.40	0.21	0.28	19
tetragonal	0.33	0.38	0.35	8
accuracy			0.76	318
macro avg	0.57	0.55	0.55	318
weighted avg	0.75	0.76	0.75	318

The fold is: 1:

	precision	recall	f1-score	support
cubic	0.83	0.86	0.84	178
orthorhombic	0.76	0.71	0.73	112
rhombohedral	0.33	0.37	0.35	19
tetragonal	0.38	0.33	0.35	9
accuracy			0.76	318
macro avg	0.57	0.57	0.57	318
weighted avg	0.76	0.76	0.76	318

The fold is: 2:

	precision	recall	f1-score	support
cubic	0.85	0.89	0.87	178
orthorhombic	0.75	0.79	0.77	112
rhombohedral	0.33	0.21	0.26	19
tetragonal	1.00	0.44	0.62	9
accuracy			0.80	318
macro avg	0.73	0.58	0.63	318
weighted avg	0.79	0.80	0.79	318

The fold is: 3:

	precision	recall	f1-score	support
cubic	0.86	0.87	0.86	178
orthorhombic	0.72	0.75	0.74	112
rhombohedral	0.33	0.26	0.29	19
tetragonal	0.38	0.33	0.35	9
accuracy			0.77	318
macro avg	0.57	0.55	0.56	318
weighted avg	0.77	0.77	0.77	318

The fold is: 4:

	precision	recall	f1-score	support
cubic	0.84	0.93	0.88	179
orthorhombic	0.83	0.71	0.77	112
rhombohedral	0.62	0.44	0.52	18
tetragonal	0.20	0.22	0.21	9
accuracy			0.81	318
macro avg	0.62	0.58	0.59	318

weighted avg	0.81	0.81	0.80	318
--------------	------	------	------	-----

Average Precision: 0.7759573233355838

Average Recall: 0.779874213836478

Average F1-score: 0.7755647332614042

Average Accuracy: 0.779874213836478

In [39]:

```
1 print("Raw Counts")
2 print(prv_train_tau[\"lowest_distortion\"].value_counts())
3 print()
4 print("Percentages")
5 print(prv_train_tau[\"lowest_distortion\"].value_counts(normalize=True))
```

Raw Counts

```
cubic           1127
orthorhombic    692
rhombohedral     114
tetragonal       55
Name: lowest_distortion, dtype: Int64
```

Percentages

```
cubic          0.566901
orthorhombic   0.348089
rhombohedral   0.057344
tetragonal     0.027666
Name: lowest_distortion, dtype: float64
```

It is worth bringing up that the accuracy metric doesn't take into account an imbalanced dataset. If we had a model that always predicted a cubic crystal structure, the accuracy score would be only 56% (baseline, for uninformed decision, the worst it could be). (cell 39)

Using the Decision Tree Classifier (model), it is clear that the crystal structure accuracy score is performing better with the Stratified K Fold performance evaluation at 78%. (cell 38)

These are the example kfold scores (with default guess hyperparameterization) for each fold. At this point, the f1 score seems to be the most comprehensive. The closer the f1 score is to 1, the better the model will be at predicting the target. The closer to a score of 1 means that the model is doing well in both minimizing false positives and false negatives because both the precision and recall are high scores are well. The Decision Tree model was trained once for each of the 5 folds using a random seed of 42 so that the starting point for the randomness is chosen in a way that's consistent and predictable for the purposes of building this experiment.

Rhombohedral and tetragonal are training on little data so the scores are expected to be low compared to the cubic and orthorhombic which are training on a lot more data.

This presents a great opportunity to tune the hyperparameters and optimize the training before trying out different models.

# Feature Engineering

```
In [40]: 1 # Create a copy of the test/train DataFrame to avoid altering the original
2 prv_train_tau_bin = prv_train_tau.copy()
3
4 # Calculate the absolute electronegativity difference between elements A and B
5 prv_train_tau_bin['abs_EN_diff']=abs(prv_train_tau_bin['EN_A'] - prv_train_tau_bin['EN_B'])
6
```

## Does bond type influence the crystal structure shape?

The absolute value of the electronegativity difference between atoms A + B will be classified into different bond types from the describe method as shown here:

Electronegativity Difference	Bond Type Character	Label Number
0.0 - 0.4	Nonpolar Covalent	1
0.4 - 1.7	Polar Covalent	2
1.7 +	Ionic Bond	3

```
In [41]: 1 Pandas cut method to map the absolute value electronegativity between A+B into
2
3 cutoff for each bin
4 _tau_bin_edges = [0.0, 0.4, 1.7, 4]
5
6 ch6bin
7 _tau_bin_names = [1, 2, 3]
8
9 map for different ranges into which the values of the 'Electronegativity difference'
10 _tau_bin['abs_EN_diff_bin'] = pd.cut(prv_train_tau_bin.abs_EN_diff, prv_train_tau_bin.abs_EN_diff,
11 bins=_tau_bin_edges, labels=_tau_bin_names)
12
13 (prv_train_tau_bin)
14 prv_train_tau_bin.head()
15
```

1988

vA	vB	r_A12	r_A6	r_B6	EN_A	EN_B	bond_len_AO	bond_len_BO	ENR_diff	tG	tau		
2	2	4	0.92	0.67	0.53	1.83	1.88	2.290644	1.930311	-1.468464	0.849994	4.936558	0
3	1	5	1.64	1.38	0.62	0.82	2.36	3.025719	1.745600	-1.974429	1.064161	3.977376	0
6	1	5	0.92	0.80	0.62	1.78	2.36	2.392362	1.745600	-1.484143	0.812123	5.017992	0
10	3	3	1.36	1.03	0.90	1.10	1.23	2.689119	2.258656	-2.931286	0.848528	3.536265	0
11	2	4	1.19	0.99	0.61	1.22	1.54	2.550861	1.927849	-2.040571	0.911148	4.133678	0

```
In [42]: #1 Remove columns 'abs_EN_diff', 'EN_A', and 'EN_B' from the DataFrame to eliminate them
prv_train_tau_bin.drop(columns=['abs_EN_diff', 'EN_A', 'EN_B'], inplace=True)
3
#4 Reorder columns
new_order = ['vA', 'vB', 'r_A12', 'r_A6', 'r_B6', 'bond_len_A0', 'bond_len_B0']
6
#7 Reorder the columns
p8v_train_tau_bin = prv_train_tau_bin[new_order]
9
```

```
In [43]: #1 Splitting the dataset into features (X_tau_bin) and target labels (y_tau_bin)
X_tau_bin = prv_train_tau_bin.loc[:, 'vA':'abs_EN_diff_bin']
y3_tau_bin = prv_train_tau_bin.loc[:, 'lowest_distortion']
4
#5 Splitting the dataset into training and testing sets
#6 X_train_bin and y_train_bin will contain the training data, while X_test_bin,
X7_train_bin, X_test_bin, y_train_bin, y_test_bin = train_test_split(X_tau_bin,
8
```

```
In [44]: 1 # Print the lengths of training and testing data
2 len(X_train_bin), len(X_test_bin)
```

```
Out[44]: (1590, 398)
```



```
In [45]: 1from sklearn.metrics import classification_report
2
3# Initialize lists to store scores for each fold
4precision_scores_bin = []
5recall_scores_bin = []
6f1_scores_bin = []
7accuracy_scores_bin = []
8
9#Step 1, start training
10# Initialize StratifiedKFold with 5 splits for cross-validation
11kf = StratifiedKFold(n_splits=5)
12
13# Iterate over each fold
14for fold, (trn_, val_) in enumerate(kf.split(X=X_train_bin, y=y_train_bin)):
15
16    # Split the data into training and validation sets for the current fold
17    X_train_Kfold_bin = X_train_bin.iloc[trn_]
18    y_train_Kfold_bin = y_train_bin.iloc[trn_]
19
20    X_valid_bin = X_train_bin.iloc[val_]
21    y_valid_bin = y_train_bin.iloc[val_]
22
23    # Initialize a Decision Tree classifier with specified hyperparameters
24    DTclf_bin = DecisionTreeClassifier(criterion="entropy", max_depth=10, random_state=42)
25
26    # Train the Decision Tree classifier on the training data for the current fold
27    DTclf_bin.fit(X_train_Kfold_bin, y_train_Kfold_bin)
28
29    # Predict labels for the validation set
30    y_pred_Kfold_bin = DTclf_bin.predict(X_valid_bin)
31
32    # Calculate scores for the current fold
33    report_bin = classification_report(y_valid_bin, y_pred_Kfold_bin, output_dict=True)
34
35    # Append precision, recall, F1-score, and accuracy scores to respective lists
36    precision_scores_bin.append(report_bin['weighted avg']['precision'])
37    recall_scores_bin.append(report_bin['weighted avg']['recall'])
38    f1_scores_bin.append(report_bin['weighted avg']['f1-score'])
39    accuracy_scores_bin.append(report_bin['accuracy'])
40
41    # Print classification report for the current fold
42    print(f"The fold is: {fold}:")
43    print(classification_report(y_valid_bin, y_pred_Kfold_bin))
44
45# Calculate average scores across all folds
46avg_precision_bin = np.mean(precision_scores_bin)
47avg_recall_bin = np.mean(recall_scores_bin)
48avg_f1_bin = np.mean(f1_scores_bin)
49avg_accuracy_bin = np.mean(accuracy_scores_bin)
50
51# Print average evaluation metrics across all folds
52print(f"Average Precision Bin: {avg_precision_bin}")
53print(f"Average Recall Bin: {avg_recall_bin}")
54print(f"Average F1-score Bin: {avg_f1_bin}")
55print(f"Average Accuracy Bin: {avg_accuracy_bin}")
56
```



The fold is: 0:

	precision	recall	f1-score	support
cubic	0.84	0.81	0.83	178
orthorhombic	0.70	0.71	0.70	113
rhombohedral	0.23	0.26	0.24	19
tetragonal	0.25	0.25	0.25	8
accuracy			0.73	318
macro avg	0.50	0.51	0.51	318
weighted avg	0.74	0.73	0.73	318

The fold is: 1:

	precision	recall	f1-score	support
cubic	0.79	0.84	0.81	178
orthorhombic	0.75	0.66	0.70	112
rhombohedral	0.29	0.32	0.30	19
tetragonal	0.33	0.33	0.33	9
accuracy			0.73	318
macro avg	0.54	0.54	0.54	318
weighted avg	0.73	0.73	0.73	318

The fold is: 2:

	precision	recall	f1-score	support
cubic	0.82	0.84	0.83	178
orthorhombic	0.70	0.71	0.71	112
rhombohedral	0.56	0.47	0.51	19
tetragonal	0.67	0.44	0.53	9
accuracy			0.76	318
macro avg	0.69	0.62	0.65	318
weighted avg	0.76	0.76	0.76	318

The fold is: 3:

	precision	recall	f1-score	support
cubic	0.86	0.83	0.84	178
orthorhombic	0.72	0.79	0.75	112
rhombohedral	0.35	0.32	0.33	19
tetragonal	0.57	0.44	0.50	9
accuracy			0.77	318
macro avg	0.62	0.59	0.61	318
weighted avg	0.77	0.77	0.77	318

The fold is: 4:

	precision	recall	f1-score	support
cubic	0.82	0.89	0.86	179
orthorhombic	0.77	0.70	0.73	112
rhombohedral	0.43	0.33	0.38	18
tetragonal	0.25	0.22	0.24	9
accuracy			0.77	318
macro avg	0.57	0.54	0.55	318

weighted avg	0.77	0.77	0.77	318
--------------	------	------	------	-----

Average Precision Bin: 0.7521560849621787

Average Recall Bin: 0.7528301886792453

Average F1-score Bin: 0.7513915936097715

Average Accuracy Bin: 0.7528301886792453

This is a transformation and evaluation of whether or not the bins helped or not. The addition of the electronegativity difference column is feature engineering with the data.

The bins do not seem to improve the cross validation scores so we will move forward with the chosen dataset and not the binned dataset. Because most of the data is continuous and numeric (some of which derived) then it isn't expected that any other binning would have much of an impact.

If the pair plots had shown any bimodal distributions then perhaps binning would be more useful.

## Hyperparameter Tuning

In [46]:

```
1 # Start measuring the time taken for grid search
2 start_time = time.time()
3
4 # Set the random seed
5 N_SEED = 42
6
7 # Define the cross-validation strategy
8 cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=N_SEED)
9
10 # Define the Decision Tree classifier
11 DTclf = DecisionTreeClassifier(random_state=N_SEED)
12
13 # Define the hyperparameters grid
14 param_grid = {
15     'criterion': ['gini', 'entropy'],
16     'max_depth': [None, 5, 10, 15],
17     'min_samples_split': [2, 5, 10],
18     'min_samples_leaf': [1, 2, 4]
19 }
20
21 # Perform grid search
22 grid_search_cv = GridSearchCV(
23     estimator=DTclf,
24     param_grid=param_grid,
25     scoring='accuracy',
26     n_jobs=-1,
27     cv=cv)
28
29 grid_search_cv.fit(X_train, y_train)
30
31 # Measure the time taken for grid search
32 end_time = time.time()
33 elapsed_time = end_time - start_time
34 print("Grid search completed in {:.2f} seconds".format(elapsed_time))
35
36 # Print the best hyperparameters found
37 print(grid_search_cv.best_params_)
38
39
```

Grid search completed in 1.64 seconds

{'criterion': 'entropy', 'max\_depth': 15, 'min\_samples\_leaf': 1, 'min\_samples\_split': 2}

This hyperparameter tuning is interesting. It could be assumed that unlimited depth (layers/decisions on each branch of the tree) in a tree would be best at producing the highest accuracy, but in this case the hypertuning suggests that only 15 tree layers are needed to optimize the model. This might be because the data set is so small, so it seems like tuning hyperparameters may not make a huge difference.

## Train the Decision Tree Model

The next step will be to train all of the training data not just a piece (split).

```
In [47]: 1 # Define the Decision Tree classifier with tuned hyperparameters
2 all_DTclf = DecisionTreeClassifier(criterion="entropy", max_depth=15, random_state=42)
3
4 # Train the classifier on the entire training set
5 all_DTclf.fit(X_train, y_train) # Final step of training
6
```

```
Out[47]: DecisionTreeClassifier(criterion='entropy', max_depth=15, random_state=42)
```

## Performance Evaluation

Does the prediction x data match real-life y?

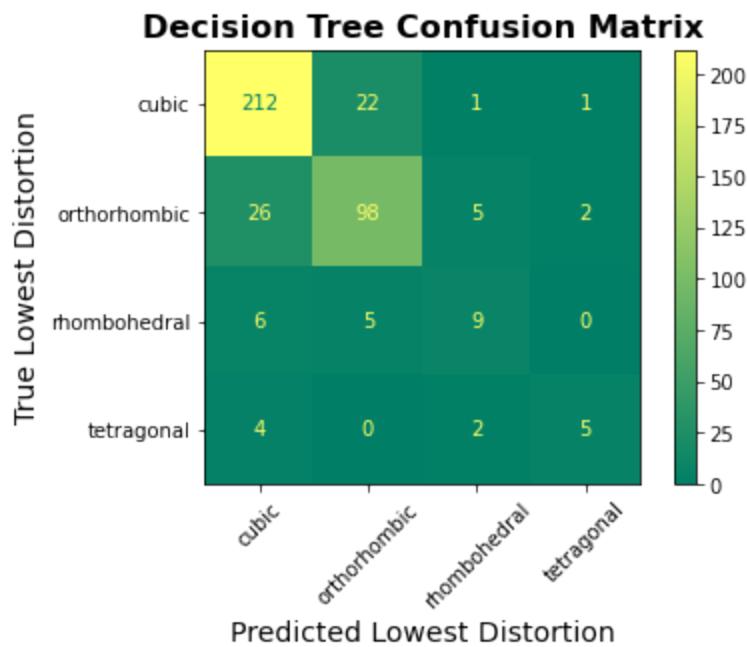
```
In [48]: 1 # Step 1: Apply the trained model to the holdout data to make predictions
2 y_pred_final_DT = all_DTclf.predict(X_test)
3
4 # Step 2: Assess the performance of the classifier
5
6 # Compute the confusion matrix to evaluate the classification performance
7 DTconf_matrix = confusion_matrix(y_test, y_pred_final_DT)
8 print("Confusion Matrix:\n", DTconf_matrix)
9
```

Confusion Matrix:

```
[[212  22   1   1]
 [ 26  98   5   2]
 [  6   5   9   0]
 [  4   0   2   5]]
```

```
In [49]: 1 # Create a ConfusionMatrixDisplay object with the confusion matrix and disp
2 disp = ConfusionMatrixDisplay(confusion_matrix=DTconf_matrix, display_labels=
3
4 # Set the size of the plot
5 plt.figure(figsize=(40, 35))
6
7 # Plot the confusion matrix
8 disp.plot(cmap='summer')
9
10 # Rotate x-axis labels
11 plt.xticks(rotation=45)
12
13 # Set the title of the plot
14 plt.title("Decision Tree Confusion Matrix", fontsize=16, fontweight='bold')
15
16 # Set the axis labels
17 plt.xlabel("Predicted Lowest Distortion", fontsize=14)
18 plt.ylabel("True Lowest Distortion", fontsize=14)
19
20 # Visualize confusion matrix
21 plt.show()
```

<Figure size 2880x2520 with 0 Axes>



It would appear that the classifier is only able to classify cubic shape with the most accuracy - it's optimized for this shape. This makes sense because the majority of the data points relate to the cubic shape.

```
In [50]: 1 # Calculate the accuracy of the classifier
2 DTaccuracy = accuracy_score(y_test, y_pred_final_DT)
3
4 # Print the accuracy score
5 print("Accuracy:", DTaccuracy)
6
7
```

Accuracy: 0.8140703517587939

This is great! Tuning the hyperparameters to maximize the accuracy increased the accuracy from 78% to 81%.

```
In [51]: 1 #Generate and print the classification report
2 DReport = classification_report(y_test, y_pred_final_DT)
3 print("Classification Report:\n", DReport)
4
```

	precision	recall	f1-score	support
cubic	0.85	0.90	0.88	236
orthorhombic	0.78	0.75	0.77	131
rhombohedral	0.53	0.45	0.49	20
tetragonal	0.62	0.45	0.53	11
accuracy			0.81	398
macro avg	0.70	0.64	0.66	398
weighted avg	0.81	0.81	0.81	398

This supports the conclusion above with the confusion matrix. This is reasonable given the size of the data set and the simplicity of the decision tree model.

Potentially, a feature could be dropped and this may improve the accuracy of the model.

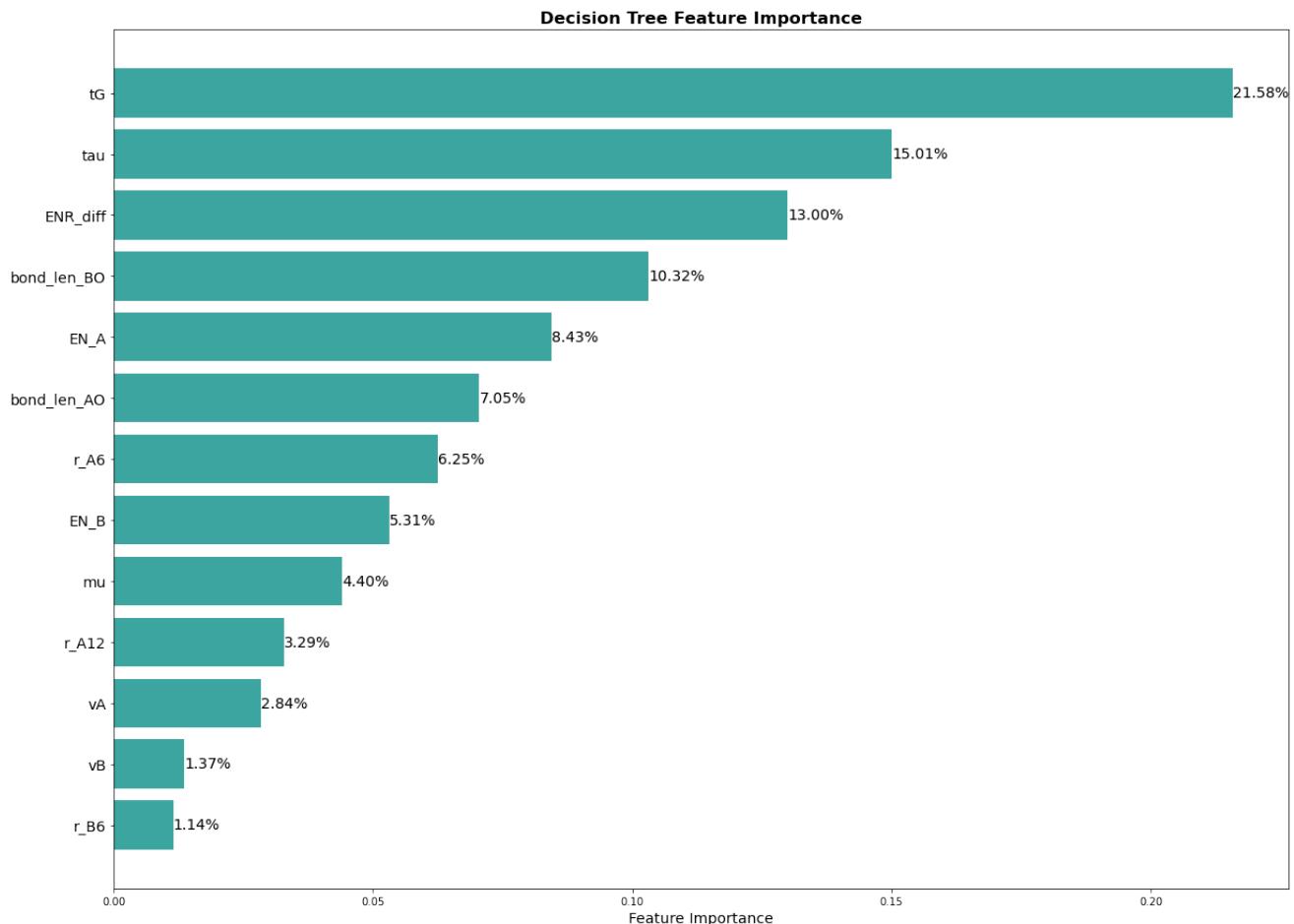
## Feature Importance

In [52]:

```
1 #Step 3, feature importance and printing a visualization
2
3 ## Checking Feature importance
4
5 # Create a figure with a specific size for the plot
6 plt.figure(figsize=(20, 15))
7
8 # Obtain feature importance from the trained decision tree classifier
9 DTimportance = all_DTclf.feature_importances_
10
11 # Sort the indices of features based on their importance
12 idxs = np.argsort(DTimportance)
13
14 # Set the title of the plot
15 plt.title("Decision Tree Feature Importance", fontsize=16, fontweight='bold')
16
17 # Create horizontal bar plot for feature importance
18 bars = plt.barh(range(len(idxs)), DTimportance[idxs], align="center", color="blue")
19
20 # Set y-axis labels to feature names based on their importance order
21 plt.yticks(range(len(idxs)), [X_train.columns[i] for i in idxs], fontsize=14)
22
23 # Set x-axis label
24 plt.xlabel("Feature Importance", fontsize=14)
25
26 # Add percentage labels to the bars
27 for i, bar in zip(idxs, bars):
28     yval = DTimportance[i]
29     plt.text(yval, bar.get_y() + bar.get_height()/2, "{:.2%}".format(yval),
30
31 # Print feature importance list
32 print("Feature Importance List:")
33 for i, feature_name in enumerate(X_train.columns[idxs]):
34     yval = DTimportance[idxs[i]]
35     print(f"{i + 1}. {feature_name}: {yval}")
36
37 #Display the plot
38 plt.show()
39
40
41
42
```

#### Feature Importance List:

1. r\_B6: 0.0114442161757059
2. vB: 0.013655687954807967
3. vA: 0.028397715225330956
4. r\_A12: 0.03292281355697572
5. mu: 0.0440063280003429
6. EN\_B: 0.05309493888597719
7. r\_A6: 0.06250300628790516
8. bond\_len\_A0: 0.07051139761566444
9. EN\_A: 0.08434850271968457
10. bond\_len\_BO: 0.10320230050295497
11. ENR\_diff: 0.13001005750909686
12. tau: 0.1500696911609554
13. tG: 0.21583334440459806



## Decision Tree Model Summary

This feature importance output shows gaps in the percentages for the top features. This may be highlighting some linear relationships and should be investigated in a future analysis.

Although tau doesn't show separation in the pairwise plot, it does seem to be a significant feature. Since tau is determined by factors of radii and bond lengths of A and B, the importance may also change and be dependent on the size of atom A or B.

The top 5 feature importance really highlights the relationship between the ENR\_diff and its relationship to EN\_A and bond\_len\_BO. The electronegativity radius difference most certainly is strongly related to the electronegativity of atom A because the stronger A is the shorter the bond length to A and the longer

the bond length is to B.

## **Random Forest Model with Stratified K-Fold Cross-Validation**

The random forest machine learning model is a similar algorithm however it's ensemble classifier which should provide an improvement from the decision tree classifier.

```
In [53]: 1 #start over! #Step 1, start training a different type of algorithm
2 # Define cross-validation strategy
3 kf = StratifiedKFold(n_splits=5)
4
5 # Iterate over each fold
6 for fold, (trn_, val_) in enumerate(kf.split(X=X_train, y=y_train)):
7
8     # Split data into training and validation sets
9     X_train_Kfold = X_train.iloc[trn_]
10    y_train_Kfold = y_train.iloc[trn_]
11
12    X_valid = X_train.iloc[val_]
13    y_valid = y_train.iloc[val_]
14
15    # Define Random Forest classifier
16    RFclf = RandomForestClassifier(n_estimators=200, criterion="entropy")
17    RFclf.fit(X_train_Kfold, y_train_Kfold) # Train the model
18    y_pred_Kfold = RFclf.predict(X_valid) # Make predictions
19
20    # Print classification report for the current fold
21    print(f"The fold is: {fold}:")
22    print(classification_report(y_valid, y_pred_Kfold))
23
```

The fold is: 0:

	precision	recall	f1-score	support
cubic	0.87	0.92	0.89	178
orthorhombic	0.78	0.81	0.79	113
rhombohedral	0.57	0.21	0.31	19
tetragonal	0.60	0.38	0.46	8
accuracy			0.82	318
macro avg	0.70	0.58	0.61	318
weighted avg	0.81	0.82	0.81	318

The fold is: 1:

	precision	recall	f1-score	support
cubic	0.86	0.92	0.89	178
orthorhombic	0.80	0.80	0.80	112
rhombohedral	0.67	0.42	0.52	19
tetragonal	1.00	0.33	0.50	9
accuracy			0.83	318
macro avg	0.83	0.62	0.68	318
weighted avg	0.83	0.83	0.82	318

The fold is: 2:

	precision	recall	f1-score	support
cubic	0.87	0.94	0.90	178
orthorhombic	0.82	0.79	0.81	112
rhombohedral	0.50	0.37	0.42	19
tetragonal	1.00	0.33	0.50	9
accuracy			0.84	318
macro avg	0.80	0.61	0.66	318
weighted avg	0.83	0.84	0.83	318

The fold is: 3:

	precision	recall	f1-score	support
cubic	0.85	0.96	0.90	178
orthorhombic	0.84	0.79	0.81	112
rhombohedral	0.82	0.47	0.60	19
tetragonal	1.00	0.33	0.50	9
accuracy			0.85	318
macro avg	0.88	0.64	0.70	318
weighted avg	0.85	0.85	0.84	318

The fold is: 4:

	precision	recall	f1-score	support
cubic	0.85	0.96	0.90	179
orthorhombic	0.87	0.79	0.83	112
rhombohedral	0.78	0.39	0.52	18
tetragonal	0.75	0.33	0.46	9
accuracy			0.85	318
macro avg	0.81	0.62	0.68	318

weighted avg	0.85	0.85	0.84	318
--------------	------	------	------	-----

## Hyperparameter Tuning

In [54]:

```
1 #next step, hyperparameter tuning
2 start_time = time.time()
3
4 # Define random seed and cross-validation strategy
5 N_SEED = 42
6 cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=N_SEED)
7
8 # Define Random Forest Classifier
9 RFclf = RandomForestClassifier(random_state=N_SEED)
10
11 # Define the hyperparameters grid
12 param_grid = {
13     'criterion': ['gini', 'entropy'],
14     'max_depth': [None, 5, 10, 15],
15     'min_samples_split': [2, 5, 10],
16     'min_samples_leaf': [1, 2, 4],
17     'n_estimators': [50, 150, 200, 500]
18 }
19
20 #Perform grid search
21 grid_search_cv = GridSearchCV(
22     estimator=RFclf,
23     param_grid=param_grid,
24     scoring='accuracy',
25     n_jobs=-1,
26     cv=cv)
27
28 grid_search_cv.fit(X_train, y_train)
29
30 # Calculate and print elapsed time for grid search
31 end_time = time.time()
32 elapsed_time = end_time - start_time
33 print("Grid search completed in {:.2f} seconds".format(elapsed_time))
34
35 # Print best hyperparameters found
36 print(grid_search_cv.best_params_)
37
38
```

Grid search completed in 68.76 seconds

{'criterion': 'entropy', 'max\_depth': None, 'min\_samples\_leaf': 1, 'min\_samples\_split': 5, 'n\_estimators': 500}

The impact of hyperparameter tuning might be less pronounced here compared to larger datasets. However, it's still valuable to perform hyperparameter tuning.

## Train the Random Forest Model

```
In [55]: 1 # Initialize a Random Forest classifier with tuned hyperparameters
2 all_RFclf = RandomForestClassifier(criterion="entropy", max_depth = None, ran
3
4 # Fit the classifier to the training data
5 all_RFclf.fit(X_train, y_train) #last step of training
6
```

```
Out[55]: RandomForestClassifier(criterion='entropy', min_samples_split=5,
                                  n_estimators=500, random_state=42)
```

## Performance Evaluation

Does the prediction x data match real-life y?

```
In [56]: 1 # Step 1: Apply the holdout data to the trained classifier
2 y_pred_final_RF = all_RFclf.predict(X_test)
3
4 #Step 2: assess the performance of the classifier
5
6 # Calculate the confusion matrix
7 RFconf_matrix = confusion_matrix(y_test, y_pred_final_RF)
8 print("Confusion Matrix:\n", RFconf_matrix)
9
```

Confusion Matrix:

```
[[220 15  0  1]
 [ 21 108  1  1]
 [  7  4  9  0]
 [  4  0  3  4]]
```

This matrix is better. The error in row 2 decreased from about 1/3 to 1/5.

```
In [57]: 1 #still step 2, measures the proportion of correctly classified instances ou
2
3 # Calculate and print the accuracy score
4 RFaccuracy = accuracy_score(y_test, y_pred_final_RF)
5 print("Accuracy:", RFaccuracy)
```

Accuracy: 0.8567839195979899

```
In [58]: #Still step 2, These metrics provide insights into the classifier's performance
2
#Calculate and print the classification report
RFreport = classification_report(y_test, y_pred_final_RF)
print("Classification Report:\n", RFreport)
6
```

Classification Report:

	precision	recall	f1-score	support
cubic	0.87	0.93	0.90	236
orthorhombic	0.85	0.82	0.84	131
rhombohedral	0.69	0.45	0.55	20
tetragonal	0.67	0.36	0.47	11
accuracy			0.86	398
macro avg	0.77	0.64	0.69	398
weighted avg	0.85	0.86	0.85	398

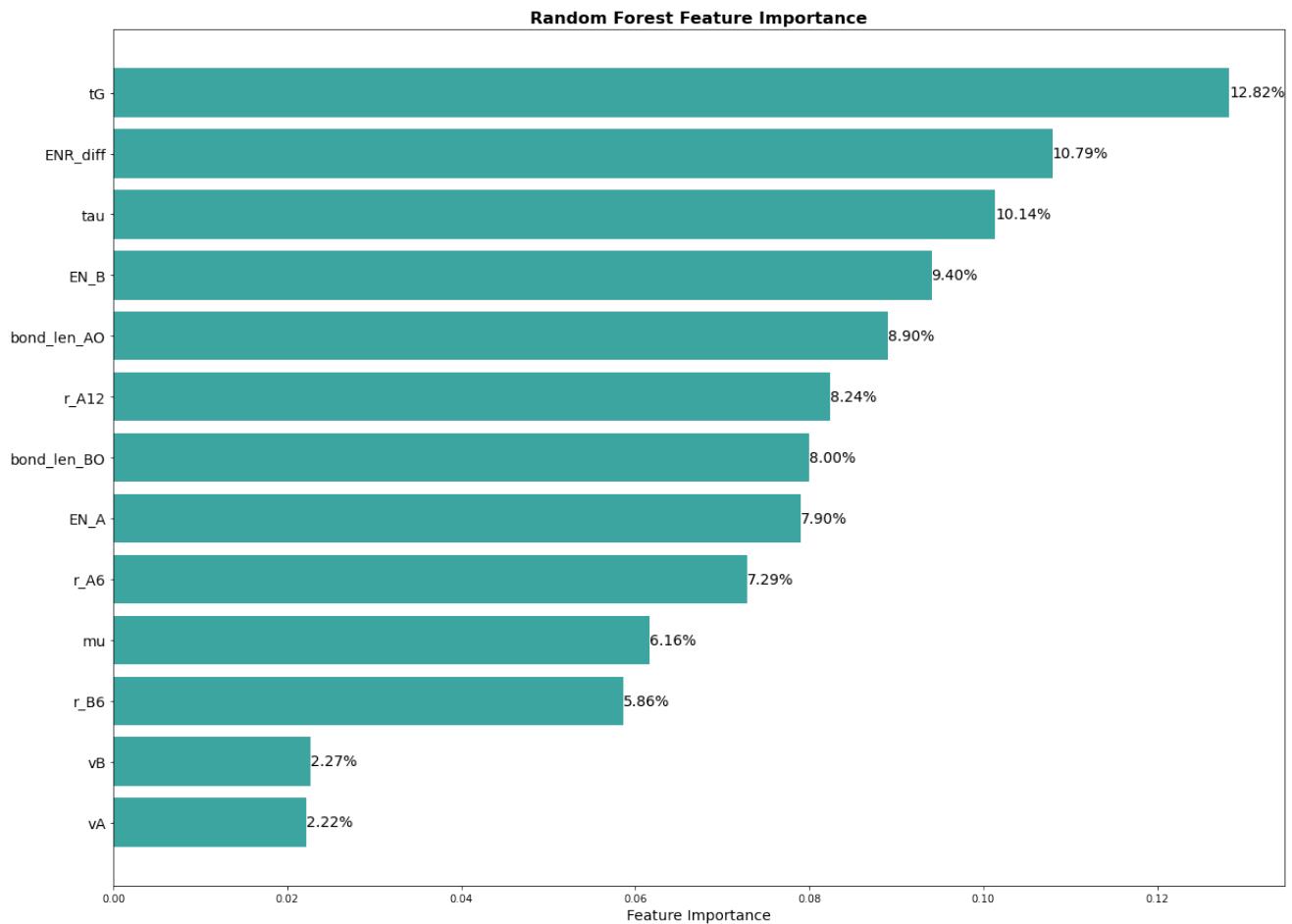
## Feature Importance

In [59]:

```
1 #Step 3, feature importance and printing a visualization
2
3 ## Checking Feature importance
4
5 # Set plot size
6 plt.figure(figsize=(20, 15))
7
8 # Get feature importance from the trained random forest classifier
9 RFimportance = all_RFclf.feature_importances_
10
11 # Sort indices based on feature importance
12 idxs = np.argsort(RFimportance)
13
14 # Set title
15 plt.title("Random Forest Feature Importance", fontsize=16, fontweight='bold')
16
17 # Create horizontal bar plot for feature importance
18 bars = plt.barh(range(len(idxs)), RFimportance[idxs], align="center", color='blue')
19
20 # Set y-ticks labels with feature names based on the sorted indices
21 plt.yticks(range(len(idxs)), [X_train.columns[i] for i in idxs], fontsize=14)
22
23 # Set label for x-axis
24 plt.xlabel("Feature Importance", fontsize=14)
25
26 # Add percentage labels to the bars
27 for i, bar in zip(idxs, bars):
28     feature_name = X_train.columns[i]
29     yval = RFimportance[i]
30     plt.text(yval, bar.get_y() + bar.get_height()/2, "{:.2%}".format(yval),
31
32 # Print feature importance list
33 print("Feature Importance List:")
34 for i, feature_name in enumerate(X_train.columns[idxs]):
35     yval = RFimportance[idxs[i]]
36     print(f"{i + 1}. {feature_name}: {yval}")
37
38 #Show plot
39 plt.show()
40
```

#### Feature Importance List:

1. vA: 0.022160397419173736
2. vB: 0.022682457628388688
3. r\_B6: 0.05859750987072454
4. mu: 0.06159343483801437
5. r\_A6: 0.07288902998776703
6. EN\_A: 0.078963615009264
7. bond\_len\_B0: 0.08001388535701298
8. r\_A12: 0.08244603789897259
9. bond\_len\_A0: 0.08904365790781456
10. EN\_B: 0.09404402383463224
11. tau: 0.1013717550249592
12. ENR\_diff: 0.10794889624140462
13. tG: 0.12824529898187145



## Random Forest Model Summary

This feature importance output shows much closer percentages for the top features than the decision tree model. This may be highlighting some linear relationships and should be investigated in a future analysis.

Although tau doesn't show separation in the pairwise plot, it does seem to be a significant feature. Since tau is determined by factors of radii and bond lengths of A and B, the importance may also change and be dependent on the size of atom A or B.

The top 5 feature importance really highlights the relationship between the ENR\_diff and its relationship to EN\_B and bond\_len\_AO. The electronegativity radius difference most certainly is strongly related to the electronegativity of atom B because the stronger B is the shorter the bond length to B and the longer the bond length is to A.

Finally, it is interesting that the Decision tree model prioritized the ENR\_diff, EN\_A, and bond\_len\_B.

## Support Vector Machine Model with Cross-Validation Score Method

Support Vector Machines (SVM) is a non tree classifier. It was chosen to compare the results of tree and nontree based models.

Support Vector Machines (SVM) are an effective solution for capturing complex nonlinear relationships in data. Because SVMs have margin maximization objective and regularization parameters, they preform well in high-dimensional or noisy datasets.

SVMs offer the ability to handle non-linear data through kernel functions, making them a valuable choice after decision trees and random forests in multiclassification problems.

In [60]:

```
1 #third model, Hyperparameter Tuning
2
3 # Define the list of kernels to try
4 kernels = ['linear', 'poly', 'rbf', 'sigmoid']
5
6 # Define hyperparameters grid for each kernel
7 param_grids = {
8     'linear': {'C': [0.1, 1, 10, 100]},
9     'poly': {'C': [1], 'degree': [3], 'gamma': ['scale', 'auto']},
10    'rbf': {'C': [0.1, 1, 10, 100], 'gamma': [0.01, 0.1, 1, 10, 'scale', 'auto']},
11    'sigmoid': {'C': [0.1, 1, 10, 100], 'gamma': [0.01, 0.1, 1, 10, 'scale', 'auto']}
12 }
13
14 # Iterate over each kernel and perform hyperparameter tuning
15 for kernel in kernels:
16     print(f"Tuning hyperparameters for SVM with {kernel} kernel")
17
18     # Start time for performance measurement
19     start_time = time.time()
20
21     # Define SVM classifier
22     SVMclf = SVC(kernel=kernel)
23
24     # Define hyperparameters grid
25     param_grid = param_grids[kernel]
26
27     # Perform grid search
28     grid_search_cv = GridSearchCV(
29         estimator=SVMclf,
30         param_grid=param_grid,
31         scoring='accuracy',
32         n_jobs=-1,
33         cv=cv
34     )
35
36     grid_search_cv.fit(X_train, y_train)
37
38     # End time and elapsed calculation for performance measurement
39     end_time = time.time()
40     elapsed_time = end_time - start_time
41
42     #Print results
43     print("Grid search for {} kernel completed in {:.2f} seconds".format(kernel))
44     print(f"Best parameters for {kernel} kernel: {grid_search_cv.best_params_}")
45
```

```

Tuning hyperparameters for SVM with linear kernel
Grid search for linear kernel completed in 806.65 seconds
Best parameters for linear kernel: {'C': 100}
Tuning hyperparameters for SVM with poly kernel
Grid search for poly kernel completed in 4643.68 seconds
Best parameters for poly kernel: {'C': 1, 'degree': 3, 'gamma': 'auto'}
Tuning hyperparameters for SVM with rbf kernel
Grid search for rbf kernel completed in 2.36 seconds
Best parameters for rbf kernel: {'C': 10, 'gamma': 10}
Tuning hyperparameters for SVM with sigmoid kernel
Grid search for sigmoid kernel completed in 0.62 seconds
Best parameters for sigmoid kernel: {'C': 10, 'gamma': 0.01}

```

## Hyperparameter Tuning

In [62]:

```

1 #assess the performance of each SVM classifier with different kernels using
2
3 # Start time for performance measurement
4 start_time = time.time()
5
6 # Define SVM classifiers with the best parameters for each kernel
7 svm_linear = SVC(kernel='linear', C=100)
8 svm_poly = SVC(kernel='poly', C=1, degree=3, gamma='auto')
9 svm_rbf = SVC(kernel='rbf', C=10, gamma=10)
10 svm_sigmoid = SVC(kernel='sigmoid', C=10, gamma=0.01)
11
12 # Perform cross-validation for each SVM classifier
13 svm_linear_scores = cross_val_score(svm_linear, X_train, y_train, cv=5)
14 svm_poly_scores = cross_val_score(svm_poly, X_train, y_train, cv=5)
15 svm_rbf_scores = cross_val_score(svm_rbf, X_train, y_train, cv=5)
16 svm_sigmoid_scores = cross_val_score(svm_sigmoid, X_train, y_train, cv=5)
17
18 # End time and elapsed calculation for performance measurement
19 end_time = time.time()
20 elapsed_time = end_time - start_time
21 print("Grid search for {} kernel completed in {:.2f} seconds".format(kernel))
22
23 # Print average cross-validation scores
24 print("Average cross-validation scores:")
25 print(f"Linear SVM: {svm_linear_scores.mean()}")
26 print(f"Polynomial SVM: {svm_poly_scores.mean()}")
27 print(f"RBF SVM: {svm_rbf_scores.mean()}")
28 print(f"Sigmoid SVM: {svm_sigmoid_scores.mean()}")
29

```

```

Grid search for sigmoid kernel completed in 8960.79 seconds
Average cross-validation scores:
Linear SVM: 0.7132075471698112
Polynomial SVM: 0.7459119496855345
RBF SVM: 0.7874213836477988
Sigmoid SVM: 0.6553459119496855

```

These scores provide insights into how well each SVM classifier generalizes to unseen data. The grid search results were used to tune the hyperparameters that are hard coded here.

The best performing SVM classifier is RBF SVM. The reason for this could be that polynomials may not effectively describe all boundaries, whereas the RBF kernel, being more robust, can accommodate a wider range of boundary shapes and variations.

Ranking the SVM classifiers in order of performance from best to worst would be

RBF (Gaussian). Polynomial. Linear. Sigmoid

## Train the SVM Model

In [63]:

```
1 #Model Training
2
3 # Start time for performance measurement
4 start_time = time.time()
5
6 # Train SVM classifiers with the best hyperparameters for each kernel
7 svm_linear_best = SVC(kernel='linear', C=100)
8 svm_rbf_best = SVC(kernel='rbf', C=10, gamma=10)
9 svm_sigmoid_best = SVC(kernel='sigmoid', C=10, gamma=0.01)
10 svm_poly_best = SVC(kernel='poly', C=100, degree=3)
11
12 svm_linear_best.fit(X_train, y_train)
13 svm_rbf_best.fit(X_train, y_train)
14 svm_sigmoid_best.fit(X_train, y_train)
15 svm_poly_best.fit(X_train, y_train)
16
17 # Model Evaluation
18
19 # Evaluate the trained SVM classifiers on the test set
20 svm_linear_accuracy = svm_linear_best.score(X_test, y_test)
21 svm_rbf_accuracy = svm_rbf_best.score(X_test, y_test)
22 svm_sigmoid_accuracy = svm_sigmoid_best.score(X_test, y_test)
23 svm_poly_accuracy = svm_poly_best.score(X_test, y_test)
24
25 print("Accuracy on the test set:")
26 print(f"Linear SVM: {svm_linear_accuracy}")
27 print(f"RBF SVM: {svm_rbf_accuracy}")
28 print(f"Sigmoid SVM: {svm_sigmoid_accuracy}")
29 print(f"Poly SVM: {svm_poly_accuracy}")
30
31 # End time and elapsed calculation for performance measurement
32 end_time = time.time()
33 elapsed_time = end_time - start_time
34
35 print("Grid search for {} kernel completed in {:.2f} seconds".format(kernel)
```

Accuracy on the test set:

Linear SVM: 0.6984924623115578

RBF SVM: 0.8291457286432161

Sigmoid SVM: 0.6155778894472361

Poly SVM: 0.592964824120603

Grid search for sigmoid kernel completed in 349.87 seconds

In the future, instead of training all models individually, I may explore advanced ensemble methods that combine predictions from multiple models, in the next steps.

Ensemble methods include: bagging, boosting, and stacking

In retrospect, I should have chosen advanced ensemble methods like gradient boosting, XGBoost, and LightGBM where the algorithms are specifically designed to handle imbalanced datasets.

## Performance Evaluation

In [64]:

```
1 # More Model Evaluation
2
3 # Start time for performance measurement
4 start_time = time.time()
5
6 # Confusion matrix and classification report for each SVM classifier
7 svm_classifiers = {
8     'Linear SVM': svm_linear_best,
9     'Polynomial SVM': svm_poly_best,
10    'RBF SVM': svm_rbf_best,
11    'Sigmoid SVM': svm_sigmoid_best
12 }
13
14 for name, clf in svm_classifiers.items():
15     print(f"\nEvaluation metrics for {name}:")
16     y_pred = clf.predict(X_test)
17     print("Confusion Matrix:")
18     print(confusion_matrix(y_test, y_pred))
19     print("\nClassification Report:")
20     print(classification_report(y_test, y_pred, zero_division=0.0))
21
22 # End time for performance measurement
23 end_time = time.time()
24 elapsed_time = end_time - start_time
25
26 print("Grid search for {} kernel completed in {:.2f} seconds".format(kernel)
27
```

Evaluation metrics for Linear SVM:

Confusion Matrix:

```
[[189  47  0  0]
 [ 42  89  0  0]
 [ 11   9  0  0]
 [  3   8  0  0]]
```

Classification Report:

	precision	recall	f1-score	support
cubic	0.77	0.80	0.79	236
orthorhombic	0.58	0.68	0.63	131
rhombohedral	0.00	0.00	0.00	20
tetragonal	0.00	0.00	0.00	11
accuracy			0.70	398
macro avg	0.34	0.37	0.35	398
weighted avg	0.65	0.70	0.67	398

Evaluation metrics for Polynomial SVM:

Confusion Matrix:

```
[[236  0  0  0]
 [131  0  0  0]
 [ 19   1  0  0]
 [ 11   0  0  0]]
```

Classification Report:

	precision	recall	f1-score	support
cubic	0.59	1.00	0.75	236
orthorhombic	0.00	0.00	0.00	131
rhombohedral	0.00	0.00	0.00	20
tetragonal	0.00	0.00	0.00	11
accuracy			0.59	398
macro avg	0.15	0.25	0.19	398
weighted avg	0.35	0.59	0.44	398

Evaluation metrics for RBF SVM:

Confusion Matrix:

```
[[221  11   1   3]
 [ 32  93   6   0]
 [  6   2  11   1]
 [  5   0   1  5]]
```

Classification Report:

	precision	recall	f1-score	support
cubic	0.84	0.94	0.88	236
orthorhombic	0.88	0.71	0.78	131
rhombohedral	0.58	0.55	0.56	20
tetragonal	0.56	0.45	0.50	11
accuracy			0.83	398
macro avg	0.71	0.66	0.68	398

weighted avg	0.83	0.83	0.82	398
--------------	------	------	------	-----

Evaluation metrics for Sigmoid SVM:

Confusion Matrix:

```
[[176  55   5   0]
 [ 40  69  22   0]
 [ 13   7   0   0]
 [  4   7   0   0]]
```

Classification Report:

	precision	recall	f1-score	support
cubic	0.76	0.75	0.75	236
orthorhombic	0.50	0.53	0.51	131
rhombohedral	0.00	0.00	0.00	20
tetragonal	0.00	0.00	0.00	11
accuracy			0.62	398
macro avg	0.31	0.32	0.32	398
weighted avg	0.61	0.62	0.61	398

Grid search for sigmoid kernel completed in 0.08 seconds

Based on the evaluation metrics for different SVM kernels, the RBF SVM achieves the highest accuracy of 83%, indicating good performance in distinguishing between different crystal structures.

All SVM kernels seem to struggle with correctly classifying samples from the 'rhombohedral' and 'tetragonal' classes, as indicated by low precision and recall scores. This might happen because the characteristics of these classes might be very similar or subtly different but challenging to distinguish. Further investigation into improving classification for these classes is needed.

## Feature Importance

For non-linear SVM models (e.g., polynomial, RBF, sigmoid), interpreting feature importance in terms of coefficients is not applicable because these models do not directly produce coefficients like linear SVMs.

Since the best kernel was RBF, permutation importance will be used.

Permutation importance is compatible with various models and is a technique that measures the increase in model error when a feature's values are randomly shuffled. By permuting feature values and observing the impact on model performance, the importance of each feature can be assessed. This technique is applicable to any machine learning model, including SVMs.

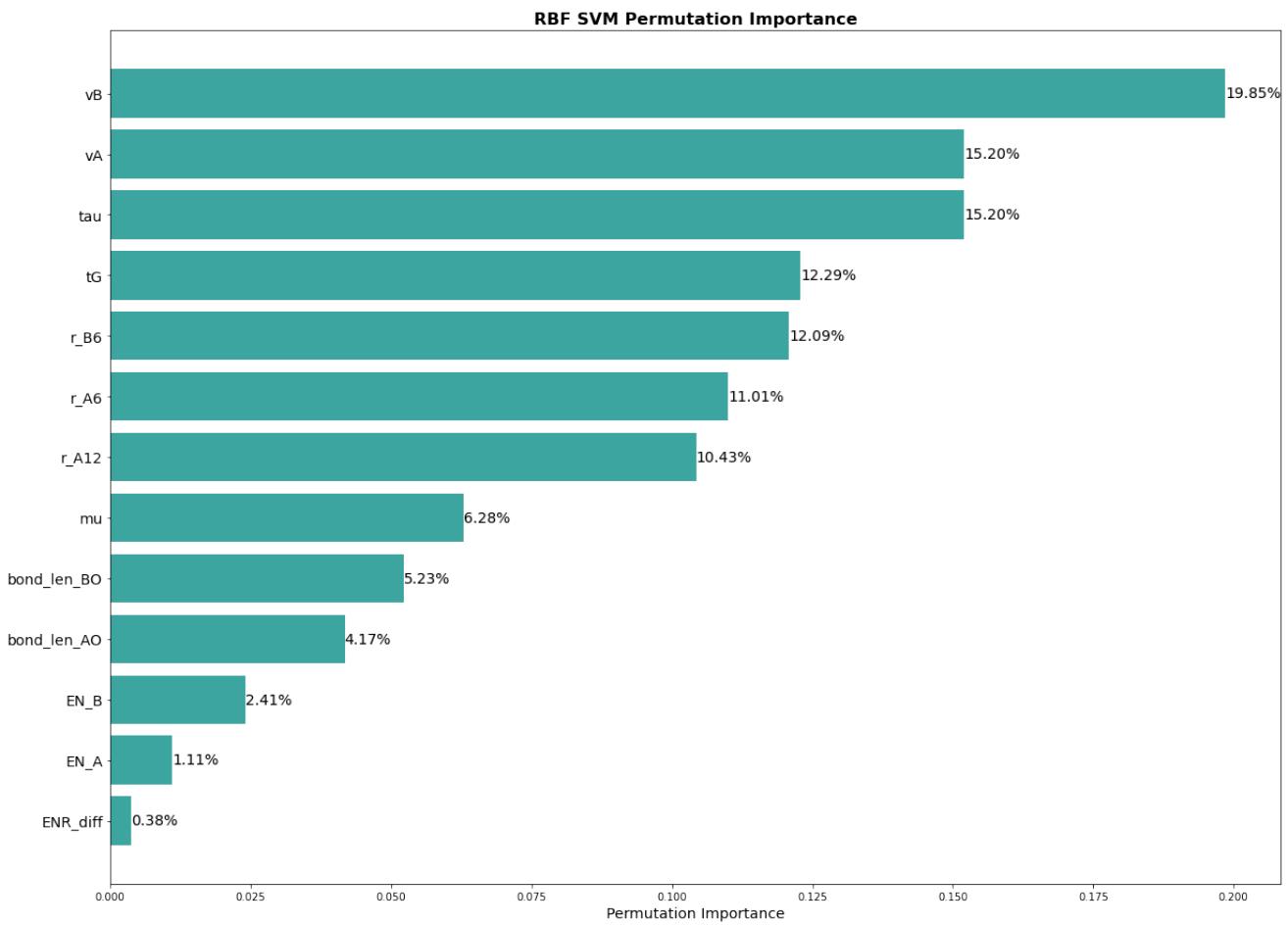
```
In [65]: 1 # Calculate permutation importance for the RBF SVM classifier
2 result = permutation_importance(svm_classifiers['RBF SVM'], X_test, y_test,
3 rbf_svm_importance = result.importances_mean
4
5 # Loop through each feature and its importance score, displaying them sequentially
6 for i, (feature, importance_score) in enumerate(zip(X_test.columns, rbf_svm_importance)):
7     print(f"{i + 1}. {feature}: {importance_score}")
8
9 # Print feature importance for the RBF SVM classifier
10 print("\nPermutation Importance for RBF SVM:")
11
```

```
1. vA: 0.1520100502512563
2. vB: 0.1520100502512563
3. r_A12: 0.10427135678391966
4. r_A6: 0.052261306532663344
5. r_B6: 0.02412060301507544
6. EN_A: 0.11005025125628146
7. EN_B: 0.12085427135678395
8. bond_len_A0: 0.06281407035175884
9. bond_len_B0: 0.04170854271356787
10. ENR_diff: 0.12286432160804021
11. tG: 0.011055276381909585
12. tau: 0.19849246231155784
13. mu: 0.0037688442211055717
```

Permutation Importance for RBF SVM:

In [75]:

```
1 # Display Permutation Importance
2
3 # Set plot size
4 plt.figure(figsize=(20, 15))
5
6 # Loop through each feature and its importance score, and sort them based on importance
7 rbf_svm_importance_sorted = [importance_score for _, importance_score in sorted(zip(X_test.columns, rbf_svm_importance))]
8
9 # Sort feature names accordingly
10 feature_names_sorted = [feature for feature, _ in sorted(zip(X_test.columns, rbf_svm_importance))]
11
12 # Set title
13 plt.title("RBF SVM Permutation Importance", fontsize=16, fontweight='bold')
14
15 # Create horizontal bar plot for feature importance
16 bars = plt.barh(range(len(feature_names_sorted)), rbf_svm_importance_sorted)
17
18 # Set y-ticks labels with feature names based on the sorted permutation importance
19 plt.yticks(range(len(feature_names_sorted)), feature_names_sorted, fontsize=14)
20
21 # Set label for x-axis
22 plt.xlabel("Permutation Importance", fontsize=14)
23
24 # Add percentage labels to the bars
25 for i, bar in enumerate(bars):
26     feature_name = feature_names_sorted[i]
27     importance_score = rbf_svm_importance_sorted[i]
28     plt.text(importance_score, bar.get_y() + bar.get_height()/2, f'{:.2%}'.format(importance_score), color='black' if importance_score > 0 else 'white')
29
30 # Show plot
31 plt.show()
32
```



## Support Vector Machine Model Summary

The most to least important permutation scores are:

1. tau: 0.19849246231155784
2. vA: 0.1520100502512563
3. vB: 0.1520100502512563
4. EN\_B: 0.12085427135678395
5. ENR\_diff: 0.12286432160804021
6. EN\_A: 0.11005025125628146
7. r\_A12: 0.10427135678391966
8. bond\_len\_AO: 0.06281407035175884
9. r\_A6: 0.052261306532663344
10. bond\_len\_BO: 0.04170854271356787
11. r\_B6: 0.02412060301507544
12. tG: 0.011055276381909585
13. mu: 0.0037688442211055717

Looking at the Permutation Importance scores, the larger the decrease in performance when a particular feature is shuffled, the more important that feature is considered to be. So in this case the RBF SVM is relying on tau, vA, vB, EN\_B, and ENR\_diff to make accurate predictions.

This has brought a new insight. When looking at the pairwise plot, the tau feature doesn't show any distinctive separation in the univariate plot, this agrees with the histogram data as well. However, looking at the tau/tG pairwise plot, there is a distinctive nonlinear separation (possibly a inverse relationship)

which means that SVM brought an advantage over the decision tree and random forest when highlighting this relationship.

Simply looking at the pairwise univariate plot the top expected features would be: tG, ENR\_diff, EN\_B, r\_A6 and r\_A12. While the pairwise plot is useful, the machine learning models are required to fully construct an understanding of the data.

It is surprising to see three discrete values ranked in the top 5 most important features according to the RBF SVM and that r\_A12 and tG ranked so low.

Discrete data is likely to be further from the boundary lines and so this result might suggest that RBF SVM handles the discrete values really well and is adept at capturing and utilizing the nonlinear relationships between these categorical features and the target variable. This capability allows the model to effectively discriminate between different 'lowest distortion' structures, leading to accurate predictions even in datasets with complex, nonlinear structures.

## Compare at all 3 models

Below is a table that compares the accuracy, recall, and weighted avg f1-score of each model.

### Machine Learning Model Comparison

Model	Precision	Recall	Accuracy	F1-Score
Decision Tree	81%	81%	81%	81%
Random Forest	85%	86%	86%	85%
RBF Support Vector Machine	83%	83%	83%	82%

These percentages show the best possible 'lowest distortion' structural prediction performance from the featured data.



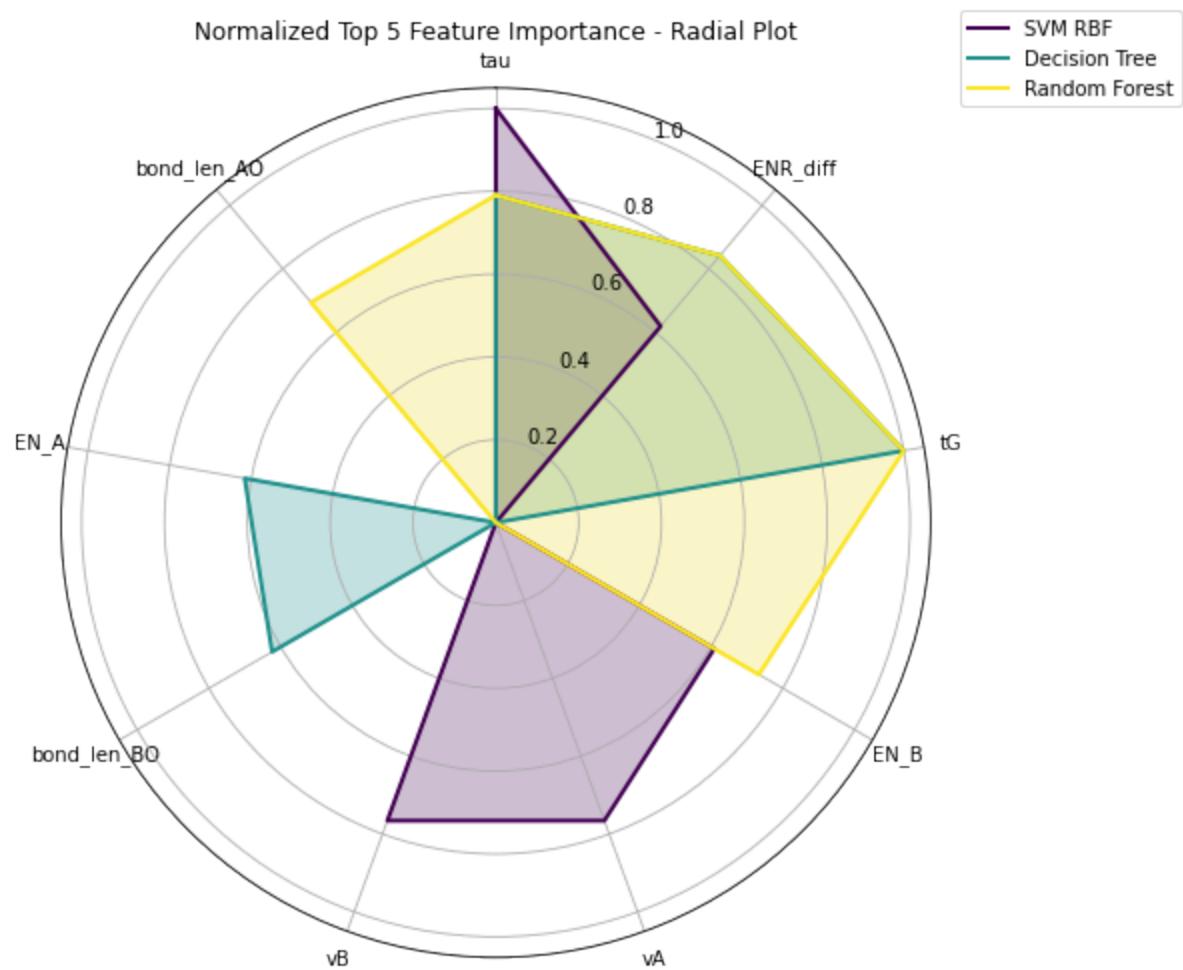
In [67]: #Final visualization

```
2
#Define all features
all_features = ['tau', 'ENR_diff', 'tG', 'EN_B', 'vA', 'vB', 'bond_len_B0',
5
#Define the top 5 ranked features and their importance scores for SVM RBF
svm_rbf_features = ['tau', 'vA', 'vB', 'EN_B', 'ENR_diff']
svm_rbf_scores = [0.19849246231155784, 0.1520100502512563, 0.1520100502512563,
9
#Define the top 5 ranked features and their importance scores for Decision Tree
decision_tree_features = ['tG', 'tau', 'ENR_diff', 'bond_len_B0', 'EN_A']
decision_tree_scores = [0.12824529898187145, 0.1013717550249592, 0.10794889624
13
#Define the top 5 ranked features and their importance scores for Random Forest
random_forest_features = ['tG', 'ENR_diff', 'tau', 'EN_B', 'bond_len_A0']
random_forest_scores = [0.12824529898187145, 0.10794889624140462, 0.1013717550
17
#Normalize scores for each model
max_score_svm_rbf = max(svm_rbf_scores)
normalized_svm_rbf_scores = [score / max_score_svm_rbf for score in svm_rbf_sc
21
max_score_decision_tree = max(decision_tree_scores)
normalized_decision_tree_scores = [score / max_score_decision_tree for score i
24
max_score_random_forest = max(random_forest_scores)
normalized_random_forest_scores = [score / max_score_random_forest for score i
27
#Initialize lists to store adjusted scores for each model
adjusted_svm_rbf_scores = []
adjusted_decision_tree_scores = []
adjusted_random_forest_scores = []
32
#Iterate over all features
for feature in all_features:
35    # Check if the feature is in the top 5 ranked features for SVM RBF
36    if feature in svm_rbf_features:
37        # If present, append its score
38        adjusted_svm_rbf_scores.append(normalized_svm_rbf_scores[svm_rbf_featu
39    else:
40        # If not present, append zero
41        adjusted_svm_rbf_scores.append(0)
42
43    # Repeat the same process for Decision Tree and Random Forest
44    if feature in decision_tree_features:
45        adjusted_decision_tree_scores.append(normalized_decision_tree_scores[d
46    else:
47        adjusted_decision_tree_scores.append(0)
48
49    if feature in random_forest_features:
50        adjusted_random_forest_scores.append(normalized_random_forest_scores[r
51    else:
52        adjusted_random_forest_scores.append(0)
53
#Convert to DataFrame
adjusted_df = pd.DataFrame({
55    'SVM RBF': adjusted_svm_rbf_scores,
56    'Decision Tree': adjusted_decision_tree_scores,
57    'Random Forest': adjusted_random_forest_scores
58})
```

```

59 index=all_features)
60
61 # Plotting Radial Chart
62 angles = np.linspace(0, 2 * np.pi, len(all_features), endpoint=False).tolist()
63 angles += angles[:1] # Close the plot
64 fig, ax = plt.subplots(figsize=(8, 8), subplot_kw=dict(polar=True))
65 colors = plt.cm.viridis(np.linspace(0, 1, len(adjusted_df.columns)))
66
67 for i, (model, scores) in enumerate(adjusted_df.iteritems()):
68     ax.plot(angles, scores.tolist() + scores[:1].tolist(), color=colors[i], la
69     ax.fill(angles, scores.tolist() + scores[:1].tolist(), color=colors[i], al
70     ax.set_theta_offset(np.pi / 2)
71     ax.set_theta_direction(-1)
72     plt.xticks(angles[:-1], all_features)
73     plt.legend(loc='upper right', bbox_to_anchor=(1.3, 1.1))
74     plt.title('Normalized Top 5 Feature Importance - Radial Plot')
75     plt.show()
76

```



## Conclusion

This analysis reveals that the Random Forest model outperforms the other models tested, making it the best choice for predicting molecular structures from this analysis. Random Forest, an ensemble learning method, harnesses the power of multiple decision trees, combining their outputs to produce more accurate predictions. This ensemble approach is advantageous compared to standalone models, as it reduces the risk of overfitting and improves generalization by averaging the predictions of individual trees.

### *Tau's Importance, Simplicity, and Overfitting:*

The consistent presence of 'tau' among the top 5 important features across different models highlights its significance in predicting the target variable. As a simple 1D descriptor, 'tau' offers efficiency and effectiveness in capturing patterns without introducing unnecessary complexity. Its simplicity makes it less prone to overfitting, contributing to model robustness and generalization.

### *Goldschmidt as a Descriptor:*

Goldschmidt's tolerance factor involves multiple dimensions, making it more complex. This complexity could potentially impact its performance in predictive modeling compared to simpler descriptors like 'tau'. While Goldschmidt may provide comprehensive insights into molecular structures, its complexity may pose challenges in model interpretation and generalization.

### *Final thoughts:*

Both Random Forest and RBF Support Vector Machine models offer unique features. Random Forest excels in handling complex datasets. For instance, in the dataset, with features such as bond lengths, bond angles, and electronegativity values, Random Forest effectively captures the complex relationships among these features to predict molecular properties.

RBF SVM demonstrates proficiency in capturing complex patterns within the data, including potential nonlinear relationships. While 'tau' and 'VA' are important features in the model, it's important to note that the relationship between these features and the target variable may exhibit complexity. For example, 'tau', despite being a one-dimensional descriptor, ranks highly in importance for RBF SVM, suggesting its relevance in capturing nuanced variations in molecular structures.

However, when considering predictive performance and generalization across diverse features like those present in molecular structures, Random Forest's ensemble approach may offer superior capabilities. This is evidenced by its ability to handle complex feature interactions and provide robust predictions across various structural parameters, ultimately enhancing the model's accuracy and reliability.

## **Summary**

The objective was to develop a predictive model for molecular structure determination, leveraging machine learning algorithms. The analysis involved three models: Decision Tree, Random Forest, and Support Vector Machine with Radial Basis Function (RBF) kernel.

### **Key Findings:**

#### **1. Decision Tree Model**

Top 5 feature importance

- tG
  - tau
  - ENR\_diff
  - bond\_len\_BO
  - EN\_A
- \*Notably, 'ENR\_diff' and 'EN\_A' exhibited strong relationships with 'bond\_len\_BO'.

Decision Tree Evaluation

- Precision: 81%
- Recall: 81%
- Accuracy: 81%
- F1-Score: 81%

## 2. Random Forest Model

Top 5 feature importance

- tG
  - ENR\_diff
  - tau
  - EN\_B
  - bond\_len\_AO
- \*suggesting correlations between 'ENR\_diff' and 'EN\_B' with 'bond\_len\_AO'.

Random Forest Evaluation

- Precision: 85%
- Recall: 86%
- Accuracy: 86%
- F1-Score: 85%

## 3. RBF Support Vector Machine

Top 5 feature importance

- tau
  - vA
  - vB
  - EN\_B
  - ENR\_diff
- \*indicating the model's capability to handle discrete values effectively.

RBF Support Vector Machine Evaluation

- Precision: 83%
- Recall: 83%
- Accuracy: 83%
- F1-Score: 82%

# Recommendations

- Utilize the Random Forest model.
- Enhance dataset with additional empirical data.
- Address missing values and discrepancies using insights from feature importance analysis to ensure data quality.
- Strategically target discrete values: Leverage RBF SVM's proficiency in capturing nonlinear relationships to target molecules with discrete values for improved predictive accuracy.

# Next Steps

There are several directions that could be explored in the next phase of this project.

#### 1. Gather Additional Empirical Data

- Focus on collecting empirical data, especially experimental data related to molecular structures. Bond lengths, bond angles, valences are critical empirical parameters.
- Consider expanding the dataset to include a wide range of molecular structures and variations to capture diverse structural characteristics.

#### 2. Improve Electronegativity and Ionic Radius Estimates:

- Investigate methods to improve the accuracy of electronegativity and ionic radius estimates using data-driven approaches. Machine learning models can be trained to refine and optimize these parameters based on observed structural characteristics.
- Assess the impact of adjusted electronegativity and ionic radius values on predictive model performance and molecular structure determination.

#### 3. Feature Segmentation

- Utilize unsupervised machine learning techniques like clustering to identify patterns and group similar molecular structures. This can help in segmenting the dataset into distinct clusters without relying on labeled data.
- Explore the use of clustering algorithms such as K-means or hierarchical clustering to identify structural similarities and differences.

#### 4. Utilize Structural Parameters for Prediction:

- Leverage structural parameters such as tG, tau, and mu to predict molecular properties and behaviors. These parameters can serve as valuable predictors in machine learning models.
- Explore existing relationships between structural parameters and molecular properties, and consider incorporating them into the feature set for predictive modeling.

#### 5. Deep Learning

- Investigate the application of deep learning models, such as neural networks, for molecular structure determination. Deep learning techniques can effectively capture complex relationships and patterns in the data.
- Explore the use of deep learning architectures like convolutional neural networks (CNNs) or recurrent neural networks (RNNs) for feature extraction and classification tasks.

#### 5. Analyze Halides for Big Data Processing:

- Explore the inclusion of halides in the dataset for big data processing and analysis. Investigate how variations in halide compositions contribute to structural diversity and affect predictive model outcomes.
- Consider incorporating halide-specific features or parameters into the dataset to enhance the predictive capabilities of machine learning models.

## References:

1. <https://www.kaggle.com/code/durgancegaur/a-guide-to-any-classification-problem#Exploratory-Data-Analysis> (<https://www.kaggle.com/code/durgancegaur/a-guide-to-any-classification-problem>)

- problem#Exploratory-Data-Analysis) (Kaggle guide)
2. <https://www.kaggle.com/datasets/meetnagadia/crystal-structure-dataset?resource=download> (<https://www.kaggle.com/datasets/meetnagadia/crystal-structure-dataset?resource=download>) (kaggle dataset)
  3. <https://www.sciencedirect.com/topics/engineering/perovskite-structure#:~:text=As%20shown%20in%20Figure%203,a%20total%20charge%20of%20%2B6> (<https://www.sciencedirect.com/topics/engineering/perovskite-structure#:~:text=As%20shown%20in%20Figure%203,a%20total%20charge%20of%20%2B6>). (reference literature)
  4. [https://en.wikipedia.org/wiki/Perovskite\\_\(structure\)](https://en.wikipedia.org/wiki/Perovskite_(structure)) ([https://en.wikipedia.org/wiki/Perovskite\\_\(structure\)](https://en.wikipedia.org/wiki/Perovskite_(structure))). (reference literature)
  5. <https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html> (<https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html>). (documentation)
  6. <https://www.nanowerk.com/what-are-perovskites.php> (<https://www.nanowerk.com/what-are-perovskites.php>). (reference literature)
  7. <https://github.com/scikit-learn/scikit-learn/issues/22230> (<https://github.com/scikit-learn/scikit-learn/issues/22230>). (documentation)
  8. <https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html> (<https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html>). (documentation)
  9. [https://scikit-learn.org/stable/modules/generated/sklearn.model\\_selection.cross\\_val\\_score.html](https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.cross_val_score.html) ([https://scikit-learn.org/stable/modules/generated/sklearn.model\\_selection.cross\\_val\\_score.html](https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.cross_val_score.html)). (documentation)
  10. <https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html> (<https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html>). (documentation)
  11. [https://scikit-learn.org/stable/modules/generated/sklearn.metrics.classification\\_report.html](https://scikit-learn.org/stable/modules/generated/sklearn.metrics.classification_report.html) ([https://scikit-learn.org/stable/modules/generated/sklearn.metrics.classification\\_report.html](https://scikit-learn.org/stable/modules/generated/sklearn.metrics.classification_report.html)). (documentation)
  12. <https://matplotlib.org/stable/users/explain/colors/colormaps.html> (<https://matplotlib.org/stable/users/explain/colors/colormaps.html>). (documentation)
  13. <https://www.color-hex.com/> (<https://www.color-hex.com/>). (hex codes)
  14. <https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html> (<https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html>). (documentation)
  15. <https://stackoverflow.com/questions/39759623/jupyter-notebook-server-password-invalid> (<https://stackoverflow.com/questions/39759623/jupyter-notebook-server-password-invalid>). (overcome randomly imposed password)
  16. <https://stackoverflow.com/questions/46551551/list-running-jupyter-notebooks-and-tokens> (<https://stackoverflow.com/questions/46551551/list-running-jupyter-notebooks-and-tokens>). (see which servers are running in python)
  17. [https://github.com/mark-barbour/ds\\_flex\\_resources](https://github.com/mark-barbour/ds_flex_resources) ([https://github.com/mark-barbour/ds\\_flex\\_resources](https://github.com/mark-barbour/ds_flex_resources)). (project set up guide)
  18. <https://www.youtube.com/watch?v=bLURfkSXnGw> (<https://www.youtube.com/watch?v=bLURfkSXnGw>). (colab intro)
  19. <https://medium.com/@maxtingle/10-jupyter-notebook-extensions-making-my-life-easier-f40139a334ce> (<https://medium.com/@maxtingle/10-jupyter-notebook-extensions-making-my-life-easier-f40139a334ce>). (jupyter notebook extensions list)
  20. [https://matplotlib.org/stable/gallery/color/colormap\\_reference.html](https://matplotlib.org/stable/gallery/color/colormap_reference.html) ([https://matplotlib.org/stable/gallery/color/colormap\\_reference.html](https://matplotlib.org/stable/gallery/color/colormap_reference.html)). (color map reference)

21. <https://www.markdownguide.org/extended-syntax/> (<https://www.markdownguide.org/extended-syntax/>) (markdown table guide)
22. <https://aiplanet.com/challenges/206/data-sprint-70-crystal-structure-classification-206/overview/about> (<https://aiplanet.com/challenges/206/data-sprint-70-crystal-structure-classification-206/overview/about>)
23. [https://www.tablesgenerator.com/markdown\\_tables](https://www.tablesgenerator.com/markdown_tables)  
([https://www.tablesgenerator.com/markdown\\_tables](https://www.tablesgenerator.com/markdown_tables)) (markdown table generator)
24. <https://stackoverflow.com/questions/46551551/list-running-jupyter-notebooks-and-tokens>  
(<https://stackoverflow.com/questions/46551551/list-running-jupyter-notebooks-and-tokens>) (finding tokens)
25. <https://www.youtube.com/watch?v=4jRBRDbJemM> (<https://www.youtube.com/watch?v=4jRBRDbJemM>) (AUC\_ROC explained)
26. [https://scikit-learn.org/stable/modules/generated/sklearn.metrics.classification\\_report.html](https://scikit-learn.org/stable/modules/generated/sklearn.metrics.classification_report.html)  
([https://scikit-learn.org/stable/modules/generated/sklearn.metrics.classification\\_report.html](https://scikit-learn.org/stable/modules/generated/sklearn.metrics.classification_report.html)) (documentation)
27. <https://www.science.org/doi/10.1126/sciadv.aav0693>  
(<https://www.science.org/doi/10.1126/sciadv.aav0693>)