

```
In [ ]: import sys
print(sys.version)
```

```
In [ ]: import io, contextlib, os
cwd = os.getcwd()
cwd
```

```
In [ ]: import pandas as pd
import numpy as np
import math
from scipy import stats
```

```
In [ ]: import matplotlib.pyplot as plt
import seaborn as sns
```

```
In [ ]: from pymatgen.core.composition import Composition
```

```
In [ ]: from sklearn.preprocessing import LabelEncoder, OneHotEncoder
from sklearn.model_selection import train_test_split, cross_val_score, cross_val_score
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.ensemble import RandomForestClassifier
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
```

```
In [ ]: import shap
shap.initjs()
```

Notebook User Controls

```
In [ ]: #product mapping: {'0': 0, 'Alcohols': 1, 'C1': 2, 'C2+': 3, 'C2H4': 4, 'C2H6': 5, 'C3H8': 6, 'C4H10': 7, 'C4H8': 8, 'C5H12': 9, 'C6H14': 10, 'C6H12': 11, 'C7H16': 12, 'C7H14': 13, 'C8H18': 14, 'C8H16': 15, 'C9H20': 16, 'C9H18': 17, 'C10H22': 18, 'C10H20': 19, 'C11H24': 20, 'C11H22': 21, 'C12H26': 22, 'C12H24': 23, 'C13H28': 24, 'C13H26': 25, 'C14H30': 26, 'C14H28': 27, 'C15H32': 28, 'C15H30': 29, 'C16H34': 30, 'C16H32': 31, 'C17H38': 32, 'C17H36': 33, 'C18H40': 34, 'C18H38': 35, 'C19H42': 36, 'C19H40': 37, 'C20H44': 38, 'C20H42': 39, 'C21H46': 40, 'C21H44': 41, 'C22H48': 42, 'C22H46': 43, 'C23H50': 44, 'C23H48': 45, 'C24H52': 46, 'C24H50': 47, 'C25H54': 48, 'C25H52': 49, 'C26H56': 50, 'C26H54': 51, 'C27H58': 52, 'C27H56': 53, 'C28H60': 54, 'C28H58': 55, 'C29H62': 56, 'C29H60': 57, 'C30H64': 58, 'C30H62': 59, 'C31H66': 60, 'C31H64': 61, 'C32H68': 62, 'C32H66': 63, 'C33H70': 64, 'C33H68': 65, 'C34H72': 66, 'C34H70': 67, 'C35H74': 68, 'C35H72': 69, 'C36H76': 70, 'C36H74': 71, 'C37H78': 72, 'C37H76': 73, 'C38H80': 74, 'C38H78': 75, 'C39H82': 76, 'C39H80': 77, 'C40H84': 78, 'C40H82': 79, 'C41H86': 80, 'C41H84': 81, 'C42H88': 82, 'C42H86': 83, 'C43H90': 84, 'C43H88': 85, 'C44H92': 86, 'C44H90': 87, 'C45H94': 88, 'C45H92': 89, 'C46H96': 90, 'C46H94': 91, 'C47H98': 92, 'C47H96': 93, 'C48H100': 94, 'C48H98': 95, 'C49H102': 96, 'C49H100': 97, 'C50H104': 98, 'C50H102': 99}
#IMPORTANT bug workaround: make sure product "A" is the one that appears first in the list

product_A = "CO" #carbon monoxide
product_B = "HCOOH" #formic acid

# product_A = "C2H4" #ethylene
# product_B = "CH3OH" #methanol

# product_A = "C2H4" #ethylene
# product_B = "CO" #carbon monoxide

# product_A = "C1" #single carbon
# product_B = "C2+" #multicarbon

# product_A = "C2H4" #ethylene
# product_B = "CH4" #methane

# product_A = "CH3OH" #methanol
```

```

# product_B = "CH4" #methane

# product_A = "CH4" #methane
# product_B = "CO" #carbon monoxide

choice_intermediates = ["COOH_111_ocp", "CO_211_ocp", "CO_110_ocp", "CO_111_
# choice_intermediates = ["CO_211_ocp", "CO_110_ocp", "CO_111_ocp", "C_111_c
# choice_intermediates = ["O_111_ocp"]

#_ch represents catalysis-hub energies

CO2_data_file = 'GPT_CO2_Modified.csv'
OCP_adsorbates_folder = "OCP_Relaxations"
hub_file = "cat_hub_search_results.csv"

add_name_tag = True

stratify = True           #ensures training and testing sets have equal label di
downsample = False       #reduces label with majority support to match minor
max_depth = 30           #the max depth that cross validation will search (was c

train_test_graph = True   #plot training and testing scores for each cro
save_train_test_graphs = True   #saves train test graphs if True; train_
save_df_csv_files = False

##Label Encode
structure_label_encode    = True
structure_onehot_encode   = False
electrolyte_label_encode  = True
electrolyte_onehot_encode = False

##One-Hot Encode
# structure_label_encode   = False
# structure_onehot_encode  = True
# electrolyte_label_encode = False
# electrolyte_onehot_encode = True

criterion_choice = 'log_loss'
#node split criterion, any of: 'gini', 'entropy', 'log_loss'
max_features_method_choice = 'log2'
#number of features considered when looking for a best split, any of: 'sqrt'

run_2026_test = False
run_confidence_intervals_analysis = False
run_hyperparams_test = False
run_mlp_code = False
#mlp controls are down below

```

```

In [ ]: def play_sound(sound):
        os.system(f"afplay /System/Library/Sounds/{sound}.aiff")

mac_sounds = [
    "Blow",
    "Bottle",
    "Frog",

```

```

    "Funk",
    "Glass",
    "Hero",
    "Morse",
    "Ping",
    "Pop",
    "Purr",
    "Sosumi",
    "Submarine",
    "Tink"
]

# for sound in mac_sounds:
#     play_sound(sound)

# play_sound("Funk")

```

```

In [ ]: if (structure_label_encode == True and structure_onehot_encode == True) or (
        raise ValueError("Invalid input provided")
        # makes sure *_label_encode and *_onehot_encode aren't both True for the
output_folder_name_tag = ""
if add_name_tag:
    play_sound("Funk")
    output_folder_name_tag = str(" " + input("Enter output file name tag: "))

choice_intermediates_files = []
for intermediate in choice_intermediates:
    filename = intermediate + ".txt"
    choice_intermediates_files.append(filename)

```

Training Data Preprocessing Steps

```

In [ ]: output_results_folder = str(product_A + " vs " + product_B + output_folder_r
if not os.path.exists(output_results_folder):
    os.makedirs(output_results_folder)

```

```

In [ ]: def preprocess_dataframe(df):

    mask_not_specified = ((df["voltage"] == "Not specified") | (df["units_of
df.loc[mask_not_specified, "voltage"] = 0
    mask_mV = (df["units_of_voltage"] == "mV") & (~mask_not_specified)
    df.loc[mask_mV, "voltage"] = pd.to_numeric(df.loc[mask_mV, "voltage"], er
    df = df.drop(columns=["file_name", "metal", "reference_electrode", "unit

    structure_encoding = {
        "Not specified": 0,
        "other": 0,
        "Nanoparticle": 10,
        "Nanoparticles": 10,
        "Nanoparticle/Nanorod": 10,
        "Core/Shell Nanoparticles": 10,
        "Nanoparticles supported by nanosheets": 10,
    }

```

```

"Thin film": 20,
"Foil, Film": 30,
"Foil": 30,
"Anodized Foil": 30,
"Porous": 40,
"Porous Thin Film": 40,
"Porous, Oxide-derived": 50,
"Porous, Oxide Derived": 50,
"Oxide derived": 50,
"Oxide derived, Polycrystalline": 50,
"Polycrystalline": 60,
"Cu-CI-ED (Polycrystalline)": 60,
"Amorphous": 70}

```

```

electrolyte_encoding = {
    "Not specified": 0,
    "Sodium Based": 10,
    "NaOH": 20,
    "NaClO4": 30,
    "Na2SO4": 40,
    "NaHCO3": 50,
    "Bicarbonate": 60,
    "KHCO3": 70,
    "K2CO3": 80,
    "KOH": 90,
    "KCl": 100,
    "K2SO4": 110,
    "Potassium Based": 120,
    "Cs based": 130,
    "Acid": 140,
    "phosphate buffer": 150,
    "other aqueous": 160,
    "Acetonitrile": 170,
    "TBAPF6/MeCN": 180,
    "TBAPF6": 190,
    "TBAPF6/DMF": 200,
    "DMF": 210,
    "other organic": 220,
    "Ionic Liquids": 230,
    "YSZ": 240,
    "other": 250}

```

```

#%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

```

structure_encoding_onehot = {
    "other": 0,
    "Nanoparticle": 10,
    "Thin film": 20,
    "Foil, Film": 30,
    "Porous": 40,
    "Porous, Oxide-derived": 50,
    "Oxide derived, Polycrystalline": 50,
    "Polycrystalline": 60,
    "Amorphous": 70}

```

```

if structure_label_encode:
    df["structure"] = df["structure"].map(structure_encoding).fillna(0).
elif structure_onehot_encode:
    df["structure"] = df["structure"].fillna("Not specified")
    df["structure"] = df["structure"].map(structure_encoding).fillna(0).

structure_encoding_onehot_flipped = {value: key for key, value in st
df["structure"] = df["structure"].map(structure_encoding_onehot_flip

df = pd.get_dummies(df, columns=["structure"], prefix="structure")
structure_cols = [col for col in df.columns if col.startswith("struc
df[structure_cols] = df[structure_cols].astype(int)

if electrolyte_label_encode:
    df["electrolyte"] = df["electrolyte"].map(electrolyte_encoding).fill
elif electrolyte_onehot_encode:
    df["electrolyte"] = df["electrolyte"].fillna("Not specified")

df = pd.get_dummies(df, columns=["electrolyte"], prefix="electrolyte
electrolyte_cols = [col for col in df.columns if col.startswith("ele
df[electrolyte_cols] = df[electrolyte_cols].astype(int)

# df["surface_area"] = df["surface_area"].replace("Not specified", 0)
# df["surface_area"] = LabelEncoder().fit_transform(df["surface_area"].a

df = df.drop('surface_area', axis=1)

df["product"] = df["product"].replace("Not specified", 0)
le_product = LabelEncoder()
df["product"] = le_product.fit_transform(df["product"].astype(str))
product_mapping = {label: int(code) for label, code in zip(le_product.cl

df["GDE"] = df["GDE"].map({"Yes": 1.0, "No": 0.0, "Not specified": 0.0})

for col in ["Molarity", "pH"]:
    df[col] = df[col].replace("Not specified", np.nan)
    df[col] = pd.to_numeric(df[col], errors="coerce")
    mode_val = df[col].mode()[0] if df[col].notna().sum() else 0
    df[col] = df[col].fillna(mode_val)

df["voltage"] = pd.to_numeric(df["voltage"], errors="coerce")
df["voltage"] = df["voltage"].fillna(df["voltage"].mean())

element_cols = ["Ag", "Al", "Au", "Bi", "Ca", "Cd", "Co", "Cr", "Cu", "Fe", "Ga", "
df = df[df[element_cols].sum(axis=1) != 0]

return df, product_mapping

```

```
In [ ]: df = pd.read_csv(CO2_data_file)
```

```
In [ ]: number_of_rows = len(df)
        number_of_columns = df.shape[1]
```

```
print(f"{number_of_rows} rows")
print(f"{number_of_columns} columns")
```

```
In [ ]: if product_A == "C1" and product_B == "C2+":
        singlecarbons = ["CH3OH", "CH4", "CO", "HCOOH", "syngas"]
        multicarbons = ["C2H4", "C2H5OH", "C2H6O"]
        for product in singlecarbons:
            df['product'] = df['product'].str.replace(product, "C1")
        for product in multicarbons:
            df['product'] = df['product'].str.replace(product, "C2+")
```

```
In [ ]: df, product_mapping = preprocess_dataframe(df)
df
```

```
In [ ]: print(product_mapping)
```

```
In [ ]: counts = df["product"].value_counts().sort_index()

int_to_name = {v: k for k, v in product_mapping.items()}

table = counts.reset_index()
table.columns = ["product_id", "count"]

table["product_name"] = table["product_id"].map(int_to_name).fillna("Unknown")

print(table[["product_id", "product_name", "count"]].to_string(index=False))
```

```
In [ ]: element_cols = ["Ag", "Al", "Au", "Bi", "Ca", "Cd", "Co", "Cr", "Cu", "Fe", "Ga", "Hf",
number_of_rows = len(df)
number_of_columns = df.shape[1]
print(f"{number_of_rows} rows")
print(f"{number_of_columns} columns")
```

```
In [ ]: def round_sig(x, sig=3):
        if pd.isna(x):
            return x
        return round(x, sig - int(np.floor(np.log10(abs(x)))) - 1)
```

```
In [ ]: df_hub_raw = pd.read_csv(hub_file)

df_hub = (df_hub_raw.dropna(subset=["bulk_db"])
          .pivot_table(index="bulk_db",
                        columns="adsorbate",
                        values="E_ads_eV",
                        aggfunc="first")
          .dropna(how="all"))
df_hub.columns = [f"{c}_hub" for c in df_hub.columns]
df_hub.reset_index(inplace=True)

for el in element_cols:
    df_hub[el] = 0
```

```
In [ ]: def add_flags(row):
        comp = Composition(row["bulk_db"])
```

```

for el in comp.get_el_amt_dict():
    if el in element_cols:
        row[el] = 1
return row

```

```
In [ ]: df_hub = df_hub.apply(add_flags, axis=1)
```

```

In [ ]: for filename in os.listdir(OCP_adsorbates_folder):
        if filename.endswith(".txt") and filename in choice_intermediates_files:
            file_path = os.path.join(OCP_adsorbates_folder, filename)
            relax_df = pd.read_csv(file_path, sep=r'\s+')
            relax_col = os.path.splitext(filename)[0]

            relax_df = relax_df[element_cols + ["relaxation_energy"]]\
                .rename(columns={"relaxation_energy": relax_col})

            relax_df[relax_col] = relax_df[relax_col].apply(round_sig)
            df = pd.merge(df, relax_df, on=element_cols, how="left")

hub_cols = [c for c in df_hub.columns if c.endswith("_hub")]
df = pd.merge(df, df_hub[element_cols + hub_cols], on=element_cols, how="left")

```

```
In [ ]: df
```

```
In [ ]: df.fillna(0, inplace=True)
df
```

```

In [ ]: counts = df["product"].value_counts().sort_index()

int_to_name = {v: k for k, v in product_mapping.items()}

table = counts.reset_index()
table.columns = ["product_id", "count"]

table["product_name"] = table["product_id"].map(int_to_name).fillna("Unknown")
print(table[["product_id", "product_name", "count"]].to_string(index=False))

```

```

In [ ]: selected_encoded = [product_mapping[product_A], product_mapping[product_B]]

df_binary = df[df["product"].isin(selected_encoded)].copy()

if downsample:
    counts = df_binary["product"].value_counts()
    maj_label = counts.idxmax()
    min_label = counts.idxmin()
    df_major = df_binary[df_binary["product"] == maj_label]
    df_minor = df_binary[df_binary["product"] == min_label]
    df_major_down = df_major.sample(n=len(df_minor), replace=False, random_state=None)
    source_df = (pd.concat([df_major_down, df_minor]).sample(frac=1, random_state=None)).copy()
    print("Resampling is enabled")
else:
    source_df = df_binary
    print("Resampling is disabled")

```

```
print("Class counts:")
print(source_df["product"].value_counts())

counts_binary = source_df["product"].value_counts().sort_index()
```

```
In [ ]: df_binary
```

```
In [ ]: counts_binary = df_binary["product"].value_counts().sort_index()
print(counts_binary)
```

```
In [ ]: print("Number of non-zero entries per column:\n")
num_rows = len(df)
for col in df.columns:
    count = np.count_nonzero(df[col])
    if count > 0:
        percent_nonzero = round_sig((count/num_rows)*100)
    else:
        percent_nonzero = 0.00
    print(f"{col}: \t {count} / {num_rows}, ({percent_nonzero}%")
```

Random Forest Training and Metric Calculation

```
In [ ]: def run_random_forest(df, dataset_name, product_mapping, show_feature_metric

    cwd_outputfolder_location = os.path.join(os.getcwd(), output_results_fol
    path_to_dataset_loc = os.path.join(cwd_outputfolder_location, dataset_na
    os.makedirs(path_to_dataset_loc, exist_ok=True)

    buf = io.StringIO() #suggestion from ChatGPT to help log print statement
    with contextlib.redirect_stdout(buf):
        print(f"Running cross validation for dataset {dataset_name}:\n")

        X, y = df.drop(columns=["product"]), df["product"]

        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
        X_train, X_val, y_train, y_val = train_test_split(X_train, y_train,

        values = [i for i in range(1, 30)]
        train_scores, test_scores = list(), list()
        for i in values:
            model_cv = RandomForestClassifier(max_depth=i, random_state = 77
            model_cv.fit(X_train, y_train)
            train_yhat = model_cv.predict(X_train)
            train_acc = accuracy_score(y_train, train_yhat)
            train_scores.append(train_acc)
            test_yhat = model_cv.predict(X_val)
            test_acc = accuracy_score(y_val, test_yhat)
            test_scores.append(test_acc)
            # print('>%d, train: %.3f, test: %.3f' % (i, train_acc, test_acc)
        plt.plot(values, train_scores, '-o', label='Train')
        plt.plot(values, test_scores, '-o', label='Test')
        plt.legend()
```



```

if save_train_test_graphs:
    savefolder_path = os.path.join(path_to_dataset_loc, f"{dataset_r
    plt.savefig(savefolder_path)
plt.show()

best_depth = int(input("Select the best depth: "))
print(f"Selected depth of {best_depth}")
print(f"\nBinary Classification: {product_A} vs {product_B}" + output
print(f"\nSubsets: {dataset_name}")

model = RandomForestClassifier(n_estimators = 100, max_depth = best_
model.fit(X_train,y_train)
y_pred = model.predict(X_test)
label_map = {selected_encoded[0]: product_A, selected_encoded[1]: pr
y_test_named = [label_map[v] for v in y_test]
y_pred_named = [label_map[v] for v in y_pred]
print(classification_report(y_test_named, y_pred_named))

scores_rf = cross_val_score(model, X, y, cv=10)
print("Cross-validation scores:", scores_rf)
formatted_cv_scores = scores_rf.mean()
print(f"Mean cross-validation accuracy (Random Forest): {formatted_c
print(f"split criterion: {criterion}, max_features method: {max_feat

product_A_value = product_mapping[product_A]
product_B_value = product_mapping[product_B]
f1_scoring = {"f1_pos": make_scorer(f1_score, pos_label=product_A_va
f1_cv_results = cross_validate(model, X, y, cv=10, scoring=f1_scorin
product_A_mean_f1 = f1_cv_results["test_f1_pos"].mean()
product_B_mean_f1 = f1_cv_results["test_f1_neg"].mean()
print(f"{product_A} F1 scores average: {product_A_mean_f1:.4f}")
print(f"{product_B} F1 scores average: {product_B_mean_f1:.4f}")

print(y_train.value_counts())

cm = confusion_matrix(y_test, y_pred) #used chatgpt suggestions for
title_cm = f"Random Forest: {dataset_name}"
plt.figure(figsize=(5, 4))
sns.heatmap(cm, annot=True, fmt="d", cmap="Greens",
            xticklabels=[product_A, product_B],
            yticklabels=[product_A, product_B])
plt.xlabel("Predicted"); plt.ylabel("Actual"); plt.title(title_cm)
plt.tight_layout(); plt.savefig(os.path.join(path_to_dataset_loc, f"
plt.show()

if show_feature_metrics:

    # importances = pd.Series(model.feature_importances_, index=X.co
    # importances.sort_values(ascending=False).plot(kind="barh", fig
    # plt.gca().invert_yaxis()
    # fi_title = f"Feature Importance - {dataset_name}"
    # plt.xlabel("Mean Decrease in Impurity"); plt.title(fi_title)
    # plt.tight_layout(); plt.savefig(os.path.join(path_to_dataset_l
    # plt.show()

importances = pd.Series(model.feature_importances_, index=X.colu

```

```

importances.sort_values(ascending=False).head(10).plot(kind="bar")
plt.gca().invert_yaxis()
fi_title = f"Feature Importance - {dataset_name}"
plt.xlabel("Mean Decrease in Impurity"); plt.title(fi_title)
plt.tight_layout(); plt.savefig(os.path.join(path_to_dataset_loc,
plt.show()

explainer = shap.Explainer(model, X_train, model_output="probability")
shap_values = explainer(X_test, check_additivity=False)[..., 1]

n_feat = X.shape[1]
shap.summary_plot(shap_values, X_test, max_display=n_feat,
                  plot_size=(10, max(6, n_feat * 0.3)), show=False)
shap_title = f"SHAP Summary - {dataset_name}"
plt.title(shap_title, fontsize=13)
plt.tight_layout(); plt.savefig(os.path.join(path_to_dataset_loc,
plt.show()

shap.summary_plot(shap_values, X_test, max_display=10, show=False)
shap_title_top10 = f"SHAP Top10 Summary - {dataset_name}"
plt.title(shap_title_top10, fontsize=13)
plt.tight_layout()
plt.savefig(os.path.join(path_to_dataset_loc, f"{shap_title_top10}
plt.show()

log = buf.getvalue()
print(log)
with open(os.path.join(path_to_dataset_loc, f"{dataset_name}.txt"), "w")
    f.write(log)

##COMMENT THIS OUT UNLESS DOING 2026 TEST:
#####
# return model
#####

```

Gradient Boost Classifier Training and Metrics Calculation

```

In [ ]: from sklearn.ensemble import GradientBoostingClassifier

def run_gradient_boost(df, dataset_name, product_mapping, show_feature_metrics):
    cwd_outputfolder_location = os.path.join(os.getcwd(), output_results_folder)
    path_to_dataset_loc = os.path.join(cwd_outputfolder_location, dataset_name)
    os.makedirs(path_to_dataset_loc, exist_ok=True)

    buf = io.StringIO()
    with contextlib.redirect_stdout(buf):
        print(f"Running cross validation for dataset {dataset_name}:\n")

        X, y = df.drop(columns=["product"]), df["product"]
        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
        X_train, X_val, y_train, y_val = train_test_split(X_train, y_train,

        values = [i for i in range(1, 30)]
        train_scores, test_scores = list(), list()

```

```

for i in values:
    model_cv = GradientBoostingClassifier(max_depth=i, random_state=
    model_cv.fit(X_train, y_train)

    train_yhat = model_cv.predict(X_train)
    train_acc = accuracy_score(y_train, train_yhat)
    train_scores.append(train_acc)

    test_yhat = model_cv.predict(X_val)
    test_acc = accuracy_score(y_val, test_yhat)
    test_scores.append(test_acc)

plt.plot(values, train_scores, '-o', label='Train')
plt.plot(values, test_scores, '-o', label='Test')
plt.legend()

if save_train_test_graphs:
    savefolder_path = os.path.join(path_to_dataset_loc, f"{dataset_r
    plt.savefig(savefolder_path)
plt.show()

best_depth = int(input("Select the best depth for Gradient Boost: "))
print(f"Selected depth of {best_depth}")

print(f"\nBinary Classification: {product_A} vs {product_B}" + output
print(f"\nSubsets: {dataset_name}")

model = GradientBoostingClassifier(n_estimators=100, max_depth=best_
model.fit(X_train, y_train)
y_pred = model.predict(X_test)

label_map = {selected_encoded[0]: product_A, selected_encoded[1]: pr
y_test_named = [label_map[v] for v in y_test]
y_pred_named = [label_map[v] for v in y_pred]

print(classification_report(y_test_named, y_pred_named))

scores_gb = cross_val_score(model, X, y, cv=10)
print("Cross-validation scores:", scores_gb)
formatted_cv_scores = scores_gb.mean()
print(f"Mean cross-validation accuracy (Gradient Boost): {formatted_
print(f"split criterion: {criterion}, max_features method: {max_feat

product_A_value = product_mapping[product_A]
product_B_value = product_mapping[product_B]
f1_scoring = {"f1_pos": make_scorer(f1_score, pos_label=product_A_va
               "f1_neg": make_scorer(f1_score, pos_label=product_B_va

f1_cv_results = cross_validate(model, X, y, cv=10, scoring=f1_scoring
product_A_mean_f1 = f1_cv_results["test_f1_pos"].mean()
product_B_mean_f1 = f1_cv_results["test_f1_neg"].mean()

print(f"{product_A} F1 scores average: {product_A_mean_f1:.4f}")
print(f"{product_B} F1 scores average: {product_B_mean_f1:.4f}")

cm = confusion_matrix(y_test, y_pred)

```

```

title_cm = f"Gradient Boost: {dataset_name}"
plt.figure(figsize=(5, 4))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=[proc
plt.xlabel("Predicted"); plt.ylabel("Actual"); plt.title(title_cm)
plt.tight_layout(); plt.savefig(os.path.join(path_to_dataset_loc, f"
plt.show()

if show_feature_metrics:
    importances = pd.Series(model.feature_importances_, index=X.colu
    importances.sort_values(ascending=False).head(10).plot(kind="bar
    plt.gca().invert_yaxis()
    fi_title = f"Feature Importance (GB) - {dataset_name}"
    plt.xlabel("Mean Decrease in Impurity"); plt.title(fi_title)
    plt.tight_layout(); plt.savefig(os.path.join(path_to_dataset_loc
    plt.show()

log = buf.getvalue()
print(log)

```

```

In [ ]: Exp_cols = ["voltage", "GDE", "Molarity", "pH"]

if "structure" in source_df.columns:
    Exp_cols.append("structure")
else:
    Exp_cols += [c for c in source_df.columns if c.startswith("structure_")]

if "electrolyte" in source_df.columns:
    Exp_cols.append("electrolyte")
else:
    Exp_cols += [c for c in source_df.columns if c.startswith("electrolyte_")

```

```

In [ ]: Comp_cols=element_cols
OCP_cols = [c for c in df.columns if c.endswith("_ocp")]
Hub_cols = [c for c in df.columns if c.endswith("_hub")]

```

```

In [ ]: df_exp = source_df[["product"] + Exp_cols].copy()
df_comp = source_df[["product"] + Comp_cols].copy()
df_ocp = source_df[["product"] + OCP_cols].copy()
df_hub = source_df[["product"] + Hub_cols].copy()

```

```

In [ ]: # df_exp = df_binary[["product"] + Exp_cols].copy()
df_exp

```

```

In [ ]: # df_comp = df_binary[["product"] + element_cols].copy()
df_comp

```

```

In [ ]: # df_ocp = df_binary[["product"] + OCP_cols].copy()
df_ocp

```

```

In [ ]: # df_hub = df_binary[["product"] + Hub_cols].copy()
df_hub

```

```

In [ ]: # df_relaxations_binary.to_csv('relaxations_binary.csv', index=False)

```

```
In [ ]: if save_df_csv_files:
        df_exp.to_csv('df_exp.csv', index=False)
        df_comp.to_csv('df_comp.csv', index=False)
        df_hub.to_csv('df_hub.csv', index=False)
        df_ocp.to_csv('df_ocp.csv', index=False)
```

Subset Partiioning

```
In [ ]: df_exp_comp_ocp_hub = source_df

df_comp_exp      = source_df[["product"] + Comp_cols + Exp_cols].copy()
df_hub_exp       = source_df[["product"] + Hub_cols + Exp_cols].copy()
df_ocp_exp       = source_df[["product"] + OCP_cols + Exp_cols].copy()
df_ocp_comp_exp  = source_df[["product"] + OCP_cols + Comp_cols + Exp_cols].c
df_hub_comp_exp  = source_df[["product"] + Hub_cols + Comp_cols + Exp_cols].c
df_comp_ocp      = source_df[["product"] + Comp_cols + OCP_cols].copy()
df_comp_hub      = source_df[["product"] + Comp_cols + Hub_cols].copy()
df_comp_hub_ocp  = source_df[["product"] + Comp_cols + Hub_cols + OCP_cols].c
df_hub_ocp       = source_df[["product"] + Hub_cols + OCP_cols].copy()
```

Run Statements for Random Forest Shallow Learning

```
In [ ]: run_random_forest(df_exp, "Exp", product_mapping, show_feature_metrics=True,
```

```
In [ ]: run_random_forest(df_comp, "Comp", product_mapping, show_feature_metrics=Tru
```

```
In [ ]: run_random_forest(df_ocp, "OCP", product_mapping, show_feature_metrics=True,
```

```
In [ ]: run_random_forest(df_hub, "Hub", product_mapping, show_feature_metrics=True,
```

```
In [ ]: run_random_forest(df_comp_exp, "Exp + Comp", product_mapping, show_feature_m
```

```
In [ ]: run_random_forest(df_hub_exp, "Exp + Hub", product_mapping, show_feature_met
```

```
In [ ]: run_random_forest(df_ocp_exp, "Exp + OCP", product_mapping, show_feature_met
```

```
In [ ]: run_random_forest(df_ocp_comp_exp, "Exp + Comp + OCP", product_mapping, show
```

```
In [ ]: run_random_forest(df_hub_comp_exp, "Exp + Comp + Hub", product_mapping, show
```

```
In [ ]: run_random_forest(df_exp_comp_ocp_hub, "Exp + Comp + OCP + Hub", product_map
```

```
In [ ]: run_random_forest(df_comp_ocp, "Comp + OCP", product_mapping, show_feature_m
```

```
In [ ]: run_random_forest(df_comp_hub, "Comp + Hub", product_mapping, show_feature_m
```

```
In [ ]: run_random_forest(df_comp_hub_ocp, "Comp + OCP + Hub", product_mapping, show
```

```
In [ ]: run_random_forest(df_hub_ocp, "OCP + Hub", product_mapping, show_feature_met
```

Run Statements for Gradient Boost Shallow Learning

```
In [ ]: # run_gradient_boost(df_exp, "Exp", product_mapping, show_feature_metrics=Tr
# run_gradient_boost(df_comp, "Comp", product_mapping, show_feature_metrics=
# run_gradient_boost(df_ocp, "OCP", product_mapping, show_feature_metrics=Tr
# run_gradient_boost(df_hub, "Hub", product_mapping, show_feature_metrics=Tr
# run_gradient_boost(df_comp_exp, "Exp + Comp", product_mapping, show_featur
# run_gradient_boost(df_hub_exp, "Exp + Hub", product_mapping, show_feature_
# run_gradient_boost(df_ocp_exp, "Exp + OCP", product_mapping, show_feature_
# run_gradient_boost(df_ocp_comp_exp, "Exp + Comp + OCP", product_mapping, s
# run_gradient_boost(df_hub_comp_exp, "Exp + Comp + Hub", product_mapping, s
# run_gradient_boost(df_exp_comp_ocp_hub, "Exp + Comp + OCP + Hub", product_
# run_gradient_boost(df_comp_ocp, "Comp + OCP", product_mapping, show_featur
# run_gradient_boost(df_comp_hub, "Comp + Hub", product_mapping, show_featur
# run_gradient_boost(df_comp_hub_ocp, "Comp + OCP + Hub", product_mapping, s
# run_gradient_boost(df_hub_ocp, "OCP + Hub", product_mapping, show_feature_
```

```
In [ ]: def run_rf_with_ci(df, dataset_name, product_mapping):
    print(f"Running Random Forest with 95% Confidence Intervals: {dataset_na

    X = df.drop(columns=["product"])
    y = df["product"]

    X_train_full, X_test, y_train_full, y_test = train_test_split(X, y, test
    X_train, X_val, y_train, y_val = train_test_split(X_train_full, y_train

    depths = range(1, 31)
    train_scores, val_scores = [], []

    for d in depths:
        model_cv = RandomForestClassifier(n_estimators=100, max_depth=d, ran
        model_cv.fit(X_train, y_train)
        train_scores.append(accuracy_score(y_train, model_cv.predict(X_train
        val_scores.append(accuracy_score(y_val, model_cv.predict(X_val)))

    plt.figure(figsize=(8, 4))
    plt.plot(depths, train_scores, '-o', label='Train Accuracy')
    plt.plot(depths, val_scores, '-o', label='Validation Accuracy')
    plt.title(f"RF Depth Search: {dataset_name}")
    plt.xlabel("Max Depth"); plt.ylabel("Accuracy"); plt.legend(); plt.show(

    best_depth = int(input(f"Select the best depth for {dataset_name} (1-30)

    model = RandomForestClassifier(n_estimators=100, max_depth=best_depth, r
    model.fit(X_train_full, y_train_full)

    prod_A_val = product_mapping[product_A]
    prod_B_val = product_mapping[product_B]

    scoring = {
        'acc': 'accuracy',
        'f1_A': make_scorer(f1_score, pos_label=prod_A_val),
```

```

        'f1_B': make_scorer(f1_score, pos_label=prod_B_val)
    }

    cv_res = cross_validate(model, X, y, cv=10, scoring=scoring)

    #extracting probabilities for product_B from every single tree
    pos_label_idx = list(model.classes_).index(prod_B_val)
    all_tree_probs = np.array([tree.predict_proba(X_test)[:, pos_label_idx]

    #Calculate Mean, Standard Deviation, and 95% CI per sample
    # 95% CI is defined as Mean +/- (1.96 * Std Dev)
    mean_probs = np.mean(all_tree_probs, axis=0)
    std_probs = np.std(all_tree_probs, axis=0)
    ci_bounds = 1.96 * std_probs

    #define "CI Width" as the range [Mean - 1.96*SD, Mean + 1.96*SD]
    #smaller width means the trees are in higher agreement (consistent)
    avg_ci_width = np.mean(ci_bounds * 2)

    print("\n" + "="*50)
    print(f"UNCERTAINTY & CONSISTENCY ANALYSIS: {dataset_name}")
    print(f"Mean CV Accuracy: {cv_res['test_acc'].mean():.4f} (± {cv_res['te
    print(f"{product_A} Mean F1: {cv_res['test_f1_A'].mean():.4f} (± {cv_res
    print(f"{product_B} Mean F1: {cv_res['test_f1_B'].mean():.4f} (± {cv_res
    print("-" * 50)
    print(f"Average 95% Confidence Interval Width: {avg_ci_width:.4f}")
    print(f"Mean Predictive Variance (Std Dev): {np.mean(std_probs):.4f}")
    print("="*50 + "\n")

    return model

```

Run Statement for All 14 Subsets Including the Confidence Interval Calculation

```

In [ ]: if run_confidence_intervals_analysis == True:
    run_rf_with_ci(df_exp, "Exp", product_mapping)
    run_rf_with_ci(df_comp, "Comp", product_mapping)
    run_rf_with_ci(df_ocp, "OCP", product_mapping)
    run_rf_with_ci(df_hub, "Hub", product_mapping)
    run_rf_with_ci(df_comp_exp, "Exp + Comp", product_mapping)
    run_rf_with_ci(df_hub_exp, "Exp + Hub", product_mapping)
    run_rf_with_ci(df_ocp_exp, "Exp + OCP", product_mapping)
    run_rf_with_ci(df_ocp_comp_exp, "Exp + Comp + OCP", product_mapping)
    run_rf_with_ci(df_hub_comp_exp, "Exp + Comp + Hub", product_mapping)
    run_rf_with_ci(df_exp_comp_ocp_hub, "Exp + Comp + OCP + Hub", product_ma
    run_rf_with_ci(df_comp_ocp, "Comp + OCP", product_mapping)
    run_rf_with_ci(df_comp_hub, "Comp + Hub", product_mapping)
    run_rf_with_ci(df_comp_hub_ocp, "Comp + OCP + Hub", product_mapping)
    run_rf_with_ci(df_hub_ocp, "OCP + Hub", product_mapping)
else:
    pass

```

Testing Historically-Trained Model on Recent Literature

```
In [ ]: run_random_forest(df_exp_comp_ocp_hub, "Exp + Comp + OCP + Hub", product_map
```

```
In [ ]: Exp_list = ['voltage', 'GDE', 'Molarity', 'pH', 'structure', 'electrolyte']

Comp_list = ["Ag", "Al", "Au", "Bi", "Ca", "Cd", "Co", "Cr", "Cu", "Fe", "Ga", "Hf", "Ir", "K", "Mg", "Mn", "Mo", "Nb", "Ni", "Pb", "Pd", "Pt", "Rh", "Ru", "Sb", "Sc", "Se", "Sn", "Te", "Ti", "U", "V", "W", "Zn", "Zr"]

OCP_list = ['COOH_111_ocp', 'CH_111_ocp', 'CO_211_ocp', 'C_111_ocp', 'CO_110_ocp', 'CO_110_211_ocp', 'CO_110_211_211_ocp', 'CO_110_211_211_211_ocp']

Hub_list = ['C_hub', 'CH_hub', 'CH2_hub', 'CH3_hub', 'CH4_hub', 'CO_hub', 'CO2_hub', 'CO2_211_hub', 'CO2_211_211_hub', 'CO2_211_211_211_hub']

product = ['product']
```

```
In [ ]: new_2026_Exp = {'voltage': -0.8437, 'GDE': 0.0, 'Molarity': 0.1, 'pH': 6.8,
                        }

new_2026_Comp = {'Ag': 0, 'Al': 0, 'Au': 0, 'Bi': 1, 'Ca': 0, 'Cd': 0, 'Co': 0, 'Cr': 0, 'Cu': 0, 'Fe': 0, 'Ga': 0, 'Hf': 0, 'In': 0, 'Ir': 0, 'K': 0, 'Mg': 0, 'Mn': 0, 'Mo': 0, 'Nb': 0, 'Ni': 0, 'Pb': 0, 'Pd': 0, 'Pt': 0, 'Rh': 0, 'Ru': 0, 'Sb': 0, 'Sc': 0, 'Se': 0, 'Sn': 0, 'Te': 0, 'Ti': 0, 'U': 0, 'V': 0, 'W': 0, 'Zn': 0, 'Zr': 0,
                    }
```

```
In [ ]: mask = (df_exp_comp_ocp_hub[Comp_list] == pd.Series(new_2026_Comp)).all(axis=1)
match = df_exp_comp_ocp_hub[mask]

ocp_hub_values = match[OCP_list + Hub_list].iloc[0].to_dict()
ocp_hub_values
```

```
In [ ]: combined_data = {**new_2026_Exp, **new_2026_Comp, **ocp_hub_values}
df_2026_test = pd.DataFrame([combined_data])
df_2026_test
```

Run Statement for Test on Recent Literature Data

```
In [ ]: if run_2026_test == True:
    trained_rf = run_random_forest(df_exp_comp_ocp_hub, "Exp + Comp + OCP + Hub", product_mapping, show_feature_metrics=True,
                                   criterion=criterion_choice,
                                   max_features=max_features_method_choice)

    features = df_exp_comp_ocp_hub.drop(columns=["product"]).columns
    df_2026_test_aligned = df_2026_test[features]

    pred_code = trained_rf.predict(df_2026_test_aligned)[0]
    pred_probs = trained_rf.predict_proba(df_2026_test_aligned)[0]

    prediction_name = [name for name, code in product_mapping.items() if code == pred_code]
```



```

    print(f"Prediction for 2026 Data: {prediction_name}")
    print(f"Confidence: {max(pred_probs)*100:.2f}%")
else:
    pass

```

Save a dataframe as .csv:

```

In [ ]: choice_df = df_exp_comp_ocp_hub
choice_df.to_csv('df.csv', index=False)

```

Random Forest hyperparams test

```

In [ ]: if run_hyperparams_test:
    criterion_list = ['gini', 'entropy', 'log_loss']
    max_features_tuple = ['sqrt', 'log2', None, 0.25, 0.5, 0.75]

    for criterion in criterion_list:
        for max_features in max_features_tuple:
            print("Criterion: ", criterion)
            print("Max_features mode: ", max_features)
            run_random_forest(df_comp_exp, "Exp + Comp", product_mapping, sh
# run_random_forest(df_exp_comp_ocp_hub, "Exp + Comp + OCP + Hub

```

```

In [ ]: play_sound("Purr")

```

MLP

```

In [ ]: if not run_mlp_code:
    raise KeyboardInterrupt("run_mlp_code set to False, ending process")

```

```

In [ ]: from tensorflow import keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Input, Dense
# from tensorflow.keras.optimizers.legacy import Adam
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.utils import to_categorical

```

```

In [ ]: import keras_tuner.tuners
from kerastuner.tuners import RandomSearch
from kerastuner.engine.hyperparameters import HyperParameters

```

```

In [ ]: import time
LOG_DIR = f"{int(time.time())}"

```

User Controls

```

In [ ]: # mlp_user_controls = {
#       "max_search_trials": 4,

```

```
# "search_executions_per_trial": 5,
# "search_verbose_mode": 2,
# "search_num_epochs": 5,
# "search_batch_size": 8,
# "best_fit_num_epochs": 100,
# "best_fit_batch_size": 32,
# "best_fit_verbose_mode": 1,
# "min_num_layers": 1,
# "max_num_layers": 5,
# "min_num_nodes": 5,
# "max_num_nodes": 10,
# "node_step_size": 1,
# "learning_rate": 0.001
# }
```

```
mlp_user_controls = {
    "max_search_trials": 100,
    "search_executions_per_trial": 5,
    "search_verbose_mode": 2,
    "search_num_epochs": 5,
    "search_batch_size": 8,
    "best_fit_num_epochs": 100,
    "best_fit_batch_size": 32,
    "best_fit_verbose_mode": 1,
}
```

Handler for Data and MLP Training

```
In [ ]: def run_mlp(df, dataset_name, mlp_user_controls):

    max_search_trials = mlp_user_controls.get("max_search_trials")
    search_executions_per_trial = mlp_user_controls.get("search_executions_per_trial")
    search_verbose_mode = mlp_user_controls.get("search_verbose_mode")
    search_num_epochs = mlp_user_controls.get("search_num_epochs")
    search_batch_size = mlp_user_controls.get("search_batch_size")
    best_fit_num_epochs = mlp_user_controls.get("best_fit_num_epochs")
    best_fit_batch_size = mlp_user_controls.get("best_fit_batch_size")
    best_fit_verbose_mode = mlp_user_controls.get("best_fit_verbose_mode")

    y = df.pop('product')
    X = df
    num_features = len(X.columns)
    print(f"Number of Features: {num_features}")

    product_A_code = product_mapping[product_A]
    product_B_code = product_mapping[product_B]

    y_transformed = y.replace(to_replace={product_A_code: 0, product_B_code: 1})
    y_onehot = to_categorical(y_transformed, num_classes=2)

    scaler = StandardScaler()
    X = scaler.fit_transform(X)
    X_train, X_test, y_train, y_test = train_test_split(X, y_transformed, test_size=0.2, random_state=42)
```

```

y_train_hot = to_categorical(y_train)
y_test_hot = to_categorical(y_test)

tuner = RandomSearch(
    build_model,
    objective=['accuracy'],
    max_trials = max_search_trials,
    executions_per_trial = search_executions_per_trial,
    directory=LOG_DIR)
print(f"Tuner:\n{tuner}")

tuner.search(x=X_train,
             y=y_train_hot,
             verbose=search_verbose_mode,
             epochs=search_num_epochs,
             batch_size = search_batch_size,
             validation_data=(X_test, y_test_hot))

print(f"Tuner search space summary:\n")
tuner.search_space_summary()

print(f"Tuner results summary:\n")
tuner.results_summary()

best_model = tuner.get_best_models()[0]
best_model.fit(X_train, y_train_hot, epochs=best_fit_num_epochs, batch_s

y_pred = best_model.predict(X_test)
y_pred_numbers = np.argmax(y_pred, axis = 1)
report = classification_report(y_test, y_pred_numbers)
print(f"Classification Report:\n{report}")

cm = confusion_matrix(y_test, y_pred_numbers)
title_cm = f"MLP Confusion Matrix - {dataset_name}"
plt.figure(figsize=(5, 4))
sns.heatmap(cm, annot=True, fmt="d", cmap="plasma",
            xticklabels=[product_A, product_B],
            yticklabels=[product_A, product_B])
plt.xlabel("Predicted"); plt.ylabel("Actual"); plt.title(title_cm)
plt.show()

```

Definition to Construct MLP Architecture

```

In [ ]: def build_model(hp):

    min_layers = 1
    max_layers = 15
    min_nodes = 2
    max_nodes = 20
    node_step = 1
    learning_rate = 0.001

    model = keras.Sequential()
    model.add(keras.layers.Flatten())

```

```

for i in range(hp.Int("num_layers", min_layers, max_layers)):
    model.add(
        keras.layers.Dense(
            units=hp.Int(f"units_{i}", min_value=min_nodes, max_value=ma
            activation=hp.Choice("activation", ["relu", "tanh"]),
        )
    model.add(keras.layers.Dense(2 ,activation='softmax'))
    opt = Adam(learning_rate=learning_rate)
    # model.compile(loss='categorical_crossentropy')
    model.compile(loss='categorical_crossentropy', optimizer=opt, metrics=['

return model

```

Run MLP Statement

```
In [ ]: run_mlp(df_exp_comp_ocp_hub, "Exp + Comp + OCP + Hub", mlp_user_controls)
```

```
In [ ]:
```

```
In [ ]:
```

```
In [ ]:
```

```
In [ ]:
```