

Double-click (or enter) to edit

▼ Step 1: Protein Selection & Data Collection:

TP53 (P04637) – Tumor suppressor protein, heavily mutated in cancers with clear functional impacts. CFTR (P13569) – Mutations in this ion channel cause cystic fibrosis, with well-characterized effects. BRCA1 (P38398) – Associated with hereditary breast cancer, with many known pathogenic and neutral mutations. PAH (P00439) – Enzyme involved in phenylketonuria, with a spectrum of benign and harmful mutations.

```
import requests

# New set of protein UniProt Accession IDs
protein_ids = ["P04637", "P13569", "P38398", "P00439"] # TP53, CFTR, BRCA1, PAH

def get_uniprot_sequence(protein_id):
    url = f"https://rest.uniprot.org/uniprotkb/{protein_id}.fasta"
    response = requests.get(url)
    if response.status_code == 200:
        return response.text
    else:
        return f"Error fetching {protein_id}"

# Fetch and save sequences
for pid in protein_ids:
    sequence = get_uniprot_sequence(pid)
    with open(f"{pid}.fasta", "w") as f:
        f.write(sequence)
    print(f"Saved {pid}.fasta")

→ Saved P04637.fasta
Saved P13569.fasta
Saved P38398.fasta
Saved P00439.fasta
```

▼ Inferences:

1. **Protein sequences were extracted** in FASTA format for further analysis.

2. **These proteins were selected because:**

- They are well-studied and have known structural and functional importance.
- They have **available AlphaFold 3D structures**, allowing structural analysis.
- They are **associated with mutations** linked to diseases, making them suitable for mutation impact analysis.
- Their diverse properties (size, function, and hydrophobicity) help in building a **robust ML model** for mutation classification.

What is FASTA?

FASTA is a **text-based format** used to store **biological sequences (DNA, RNA, or protein)** along with an optional description. It starts with a '**>**' character followed by the sequence ID, and the sequence itself appears in the next lines. This format is widely used in bioinformatics for sequence analysis and database searches.

This sets the foundation for extracting features, analyzing mutation impacts, and applying ML for classification.

```
!pip install biopython
```

```
→ Collecting biopython
  Downloading biopython-1.85-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (13 kB)
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages (from biopython) (1.26.4)
  Downloading biopython-1.85-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (3.2 MB)
   _____ 3.2/3.2 MB 59.8 MB/s eta 0:00:00
Installing collected packages: biopython
Successfully installed biopython-1.85
```

```
!pip install biopython pdb-tools numpy pandas
```

```
→ Requirement already satisfied: biopython in /usr/local/lib/python3.10/dist-packages (1.85)
Collecting pdb-tools
  Downloading pdb_tools-2.5.0-py3-none-any.whl.metadata (6.6 kB)
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages (1.26.4)
Requirement already satisfied: pandas in /usr/local/lib/python3.10/dist-packages (2.2.2)
```

```
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/python3.10/dist-packages (from pandas) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-packages (from pandas) (2024.2)
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.10/dist-packages (from pandas) (2024.2)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from python-dateutil>=2.8.2->pandas) (1.17.0)
Downloading pdb_tools-2.5.0-py3-none-any.whl (207 kB)
    207.3/207.3 kB 7.4 MB/s eta 0:00:00
```

```
Installing collected packages: pdb-tools
Successfully installed pdb-tools-2.5.0
```

```
!sudo apt-get install dssp
```

```
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
The following additional packages will be installed:
  libcifpp-data libcifpp2
The following NEW packages will be installed:
  dssp libcifpp-data libcifpp2
0 upgraded, 3 newly installed, 0 to remove and 49 not upgraded.
Need to get 1,967 kB of archives.
After this operation, 15.0 MB of additional disk space will be used.
Get:1 http://archive.ubuntu.com/ubuntu jammy/universe amd64 libcifpp-data all 2.0.5-1build1 [437 kB]
Get:2 http://archive.ubuntu.com/ubuntu jammy/universe amd64 libcifpp2 amd64 2.0.5-1build1 [1,019 kB]
Get:3 http://archive.ubuntu.com/ubuntu jammy/universe amd64 dssp amd64 4.0.4-1 [511 kB]
Fetched 1,967 kB in 1s (2,259 kB/s)
debconf: unable to initialize frontend: Dialog
debconf: (No usable dialog-like program is installed, so the dialog based frontend cannot be used. at /usr/share/perl5/Debconf/FrontEnd)
debconf: falling back to frontend: Readline
debconf: unable to initialize frontend: Readline
debconf: (This frontend requires a controlling tty.)
debconf: falling back to frontend: Teletype
dpkg-preconfigure: unable to re-open stdin:
Selecting previously unselected package libcifpp-data.
(Reading database ... 123632 files and directories currently installed.)
Preparing to unpack .../libcifpp-data_2.0.5-1build1_all.deb ...
Unpacking libcifpp-data (2.0.5-1build1) ...
Selecting previously unselected package libcifpp2:amd64.
Preparing to unpack .../libcifpp2_2.0.5-1build1_amd64.deb ...
Unpacking libcifpp2:amd64 (2.0.5-1build1) ...
Selecting previously unselected package dssp.
Preparing to unpack .../dssp_4.0.4-1_amd64.deb ...
Unpacking dssp (4.0.4-1) ...
Setting up libcifpp-data (2.0.5-1build1) ...
debconf: unable to initialize frontend: Dialog
debconf: (No usable dialog-like program is installed, so the dialog based frontend cannot be used. at /usr/share/perl5/Debconf/FrontEnd)
debconf: falling back to frontend: Readline
Configuring libcifpp-data
-----
```

The default mmcif_pdbx_v50.dic file may become out-of-date over time. Using this option you can enable automatic weekly updating of this file.

```
Should mmcif_pdbx_v50 files be updated automatically? [yes/no] yes
```

```
--2025-02-18 15:06:37-- https://mmcif.wwpdb.org/dictionaries/ascii/mmcif\_pdbx\_v50.dic.gz
```

```
Resolving mmcif.wwpdb.org (mmcif.wwpdb.org)... 132.249.213.156
```

```
Connecting to mmcif.wwpdb.org (mmcif.wwpdb.org)|132.249.213.156|:443... connected.
```

```
HTTP request sent, awaiting response... 200 OK
```

```
Length: 594002 (580K) [application/x-gzip]
```

```
Saving to: '/var/cache/libcifpp/mmcif_pdbx_v50.dic.gz'
```

```
/var/cache/libcifpp 100%[=====] 580.08K 547KB/s in 1.1s
```

```
2025-02-18 15:06:39 (547 KB/s) - '/var/cache/libcifpp/mmcif_pdbx_v50.dic.gz' saved [594002/594002].
```

```
--2025-02-18 15:06:39-- ftp://ftp.wwpdb.org/pub/pdb/data/monomers/components.cif.gz
```

```
=> '/var/cache/libcifpp/components.cif.gz'
```

```
Resolving ftp.wwpdb.org (ftp.wwpdb.org)... failed: Name or service not known.
```

```
wget: unable to resolve host address 'ftp.wwpdb.org'
```

```
!mkdssp --version
```

```
→ mkdssp version 4.0.4
```

```
from Bio.PDB import PDBParser
from Bio.SeqUtils.ProtParam import ProteinAnalysis
import pandas as pd

# Initialize PDB Parser
parser = PDBParser(QUIET=True)
```

```
# Dictionary for converting 3-letter to 1-letter amino acid codes
aa_dict = {
    'ALA': 'A', 'ARG': 'R', 'ASN': 'N', 'ASP': 'D', 'CYS': 'C', 'GLN': 'Q', 'GLU': 'E',
    'GLY': 'G', 'HIS': 'H', 'ILE': 'I', 'LEU': 'L', 'LYS': 'K', 'MET': 'M', 'PHE': 'F',
    'PRO': 'P', 'SER': 'S', 'THR': 'T', 'TRP': 'W', 'TYR': 'Y', 'VAL': 'V'
}

# Function to extract sequence from PDB file
def extract_sequence_from_pdb(pdb_file):
    structure = parser.get_structure("protein", pdb_file)
    sequence = ""
    for model in structure:
        for chain in model:
            for residue in chain:
                res_name = residue.get_resname()
                if res_name in aa_dict:
                    sequence += aa_dict[res_name]
    return sequence

# Function to compute molecular weight & hydrophobicity
def compute_protein_features(sequence):
    if not sequence: # Handle empty sequence case
        return 0, None
    analysis = ProteinAnalysis(sequence)
    mw = analysis.molecular_weight()
    hydro = analysis.gravy() # Hydrophobicity
    return mw, hydro

# Define your PDB files
pdb_files = {
    "P04637": "AF-P04637-F1-model_v4.pdb",
    "P13569": "AF-P13569-F1-model_v4.pdb",
    "P38398": "AF-P38398-F1-model_v4.pdb",
    "P00439": "AF-P00439-F1-model_v4.pdb"
}

# Extract features
protein_features = []
for protein, pdb_file in pdb_files.items():
    try:
        seq = extract_sequence_from_pdb(pdb_file)
        mw, hydro = compute_protein_features(seq)
        protein_features.append([protein, len(seq), mw, hydro])
    except Exception as e:
        print(f"Error processing {protein}: {e}")

# Convert to DataFrame
df_features = pd.DataFrame(protein_features, columns=["Protein", "Sequence_Length", "Molecular_Weight", "Hydrophobicity"])
print(df_features)
```

	Protein	Sequence_Length	Molecular_Weight	Hydrophobicity
0	P04637	393	43652.7134	-0.756234
1	P13569	1480	168139.5306	0.023649
2	P38398	1863	207718.5251	-0.784917
3	P00439	452	51861.4391	-0.375885

✓ Inference: Feature Extraction

Protein sequences successfully extracted from AlphaFold PDB structures.

Sequence lengths vary, with the longest being P38398 (1863 residues) and the shortest P04637 (393 residues).

Molecular weights range from 43.6 kDa to 207.7 kDa, aligning with expected protein mass values.

Hydrophobicity scores indicate overall polarity trends, with P13569 being slightly hydrophilic (+0.0236) and others being hydrophobic (negative values).

These features will help in mutation classification by providing essential physicochemical properties for downstream analysis.

```
from Bio.PDB import PDBParser, DSSP
import pandas as pd
import os
```

```

# List of PDB files for our current protein dataset
pdb_files = {
    "P04637": "AF-P04637-F1-model_v4.pdb",
    "P13569": "AF-P13569-F1-model_v4.pdb",
    "P38398": "AF-P38398-F1-model_v4.pdb",
    "P00439": "AF-P00439-F1-model_v4.pdb"
}

# DSSP executable path (Update if necessary)
dssp_executable = "/usr/bin/mkdssp"

# Initialize parser
parser = PDBParser(QUIET=True)

# Amino acid molecular weights
AA_WEIGHTS = {
    "A": 89.1, "R": 174.2, "N": 132.1, "D": 133.1, "C": 121.2, "E": 147.1,
    "Q": 146.2, "G": 75.1, "H": 155.2, "I": 131.2, "L": 131.2, "K": 146.2,
    "M": 149.2, "F": 165.2, "P": 115.1, "S": 105.1, "T": 119.1, "W": 204.2,
    "Y": 181.2, "V": 117.1
}

# Function to extract DSSP features
def extract_dssp_features(protein_name, pdb_file):
    structure = parser.get_structure(protein_name, pdb_file)
    model = structure[0]

    dssp = DSSP(model, pdb_file, dssp=dssp_executable)
    features = []

    for k in dssp.keys():
        residue_id = k[1]
        sec_structure = dssp[k][2] # Secondary Structure
        solvent_access = dssp[k][3] # Relative Solvent Accessibility (RSA)
        hb_energy = dssp[k][5] # Hydrogen bonding energy
        aa = dssp[k][1] # Amino acid type

        # Classify structure
        if sec_structure in "HGI":
            struct_type = "helix"
        elif sec_structure in "EB":
            struct_type = "sheet"
        else:
            struct_type = "coil"

        # Get molecular weight
        mol_weight = AA_WEIGHTS.get(aa, 0) # Default to 0 if unknown AA

        features.append([protein_name, residue_id, aa, struct_type, solvent_access, hb_energy, mol_weight])

    return pd.DataFrame(features, columns=["Protein", "Residue", "Amino_Acid", "Structure", "RSA", "HB_Energy", "Mol_Weight"])

# Extract DSSP features for all proteins
dssp_features = pd.concat([extract_dssp_features(prot, pdb) for prot, pdb in pdb_files.items()], ignore_index=True)

# Save features
dssp_features.to_csv("dssp_features.csv", index=False)
print("DSSP Feature extraction complete! Shape:", dssp_features.shape)

```

→ DSSP Feature extraction complete! Shape: (4188, 7)

```

import pandas as pd

# Load the extracted DSSP features
dssp_features = pd.read_csv("dssp_features.csv")

# Display the first few rows
display(dssp_features.head())

```

	Protein	Residue	Amino_Acid	Structure	RSA	HB_Energy	Mol_Weight
0	P04637	(' ', 1, ' ')	M	coil	1.000000	107.2	149.2
1	P04637	(' ', 2, ' ')	E	coil	1.000000	101.4	147.1
2	P04637	(' ', 3, ' ')	E	coil	0.989691	148.0	147.1
3	P04637	(' ', 4, ' ')	P	coil	0.941176	142.2	115.1
4	P04637	(' ', 5, ' ')	Q	coil	1.000000	142.0	146.2

▼ Inference

1. Enhanced feature extraction
2. Features Captured: Secondary Structure (Helix, Sheet, Coil) Solvent Accessibility (RSA) Hydrogen Bonding Energy
3. Molecular Weight per Residue Data Ready for Mutation Analysis: These features will help classify mutations as neutral, beneficial, or harmful.

```
import pandas as pd

# Load extracted DSSP features
dssp_features = pd.read_csv("dssp_features.csv")

# Show first few rows
print(dssp_features.head())
print(dssp_features.columns) # Check column names
```

	Protein	Residue	Amino_Acid	Structure	RSA	HB_Energy	Mol_Weight
0	P04637	(' ', 1, ' ')	M	coil	1.000000	107.2	149.2
1	P04637	(' ', 2, ' ')	E	coil	1.000000	101.4	147.1
2	P04637	(' ', 3, ' ')	E	coil	0.989691	148.0	147.1
3	P04637	(' ', 4, ' ')	P	coil	0.941176	142.2	115.1
4	P04637	(' ', 5, ' ')	Q	coil	1.000000	142.0	146.2

Index(['Protein', 'Residue', 'Amino_Acid', 'Structure', 'RSA', 'HB_Energy', 'Mol_Weight'],
 dtype='object')

▼ Step 2

```
import pandas as pd

# Load extracted DSSP features
dssp_features = pd.read_csv("dssp_features.csv")

# Extract only the residue number from tuples like (' ', 1, ' ')
dssp_features["Residue"] = dssp_features["Residue"].apply(lambda x: int(eval(x)[1]) if isinstance(x, str) else x)

# Save corrected file
dssp_features.to_csv("dssp_features.csv", index=False)
print("Residue column fixed and file updated!")
```

→ Residue column fixed and file updated!

```
import random
import pandas as pd

# Load extracted DSSP features
dssp_features = pd.read_csv("dssp_features.csv")

# Ensure Residue column is an integer
dssp_features["Residue"] = dssp_features["Residue"].astype(int)

# Define possible mutations (all amino acids)
AA_LIST = list("ARNDQCEGHILKMFPSTWYV")

# Function to randomly mutate residues
def generate_mutations(df, mutation_count=500):
    mutations = []
```

```

for _ in range(mutation_count):
    row = df.sample(1).iloc[0] # Pick a random residue
    original_aa = row["Amino_Acid"]
    mutated_aa = random.choice([aa for aa in AA_LIST if aa != original_aa]) # Avoid same AA

    mutations.append({
        "Protein": row["Protein"],
        "Residue": row["Residue"],
        "Original_AA": original_aa,
        "Mutated_AA": mutated_aa,
        "Structure": row["Structure"],
        "RSA": row["RSA"],
        "HB_Energy": row["HB_Energy"],
        "Mol_Weight": row["Mol_Weight"]
    })
}

return pd.DataFrame(mutations)

# Generate mutation dataset
mutation_df = generate_mutations(dssp_features, mutation_count=1000)

# Save mutations
mutation_df.to_csv("mutations.csv", index=False)
print(f"Mutation dataset generated with {len(mutation_df)} mutations!")

```

→ Mutation dataset generated with 1000 mutations!

```

import pandas as pd
mutation_df = pd.read_csv("mutations.csv")
print(mutation_df.head())

```

	Protein	Residue	Original_AA	Mutated_AA	Structure	RSA	HB_Energy	Mol_Weight
0	P13569	591	V	F	helix	0.007042	-45.9	117.1
1	P13569	720	Q	S	coil	0.949495	131.4	146.2
2	P13569	756	S	N	coil	0.715385	145.1	105.1
3	P13569	753	S	F	coil	0.730769	131.7	105.1
4	P38398	1154	L	C	coil	0.841463	158.5	131.2

```

# Ensure mutations are regenerated
import pandas as pd
import numpy as np

```

```

# Load original dataset (Ensure you have extracted features first!)
original_data = pd.read_csv("dssp_features.csv")

```

```

# Define mutation function (Example: Randomly mutate amino acids)
def generate_mutations(df, num_mutations=1000):
    amino_acids = list("ARNDCEQGHILKMFPSTWYV") # Standard AA list
    mutations = []

    for _ in range(num_mutations):
        row = df.sample(n=1).iloc[0] # Randomly select a residue
        original_aa = row["Amino_Acid"]
        mutated_aa = np.random.choice(amino_acids)

        # Ensure mutation is different from original
        while mutated_aa == original_aa:
            mutated_aa = np.random.choice(amino_acids)

        mutations.append({
            "Protein": row["Protein"],
            "Residue": row["Residue"],
            "Original_AA": original_aa,
            "Mutated_AA": mutated_aa,
            "Structure": row["Structure"],
            "RSA": row["RSA"],
            "HB_Energy": row["HB_Energy"],
            "Mol_Weight": row["Mol_Weight"]
        })

```

})

```

    return pd.DataFrame(mutations)

# Generate new mutation dataset
mutation_data = generate_mutations(original_data, num_mutations=1000)

# Save mutation dataset
mutation_data.to_csv("mutation_dataset.csv", index=False)
print("Mutation dataset successfully regenerated and saved!")

# Print first few rows to confirm
print(mutation_data.head())

```

→ Mutation dataset successfully regenerated and saved!

	Protein	Residue	Original_AA	Mutated_AA	Structure	RSA	HB_Energy	\
0	P38398	956	S	T	coil	0.815385	153.8	
1	P38398	1031	I	R	coil	0.775148	87.6	
2	P04637	338	F	I	helix	0.492386	-45.8	
3	P13569	162	K	D	helix	0.097561	-43.7	
4	P00439	241	R	P	sheet	0.153226	153.6	

	Mol_Weight
0	105.1
1	131.2
2	165.2
3	146.2
4	174.2

```

import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Load mutation dataset
mutation_data = pd.read_csv("mutation_dataset.csv") # Ensure this is the correct filename

# Classification based on HB_Energy & RSA
def classify_mutation(row):
    if row["HB_Energy"] < -0.5 and row["RSA"] < 0.2:
        return "Detimental"
    elif row["HB_Energy"] > 0.5 and row["RSA"] > 0.5:
        return "Beneficial"
    else:
        return "Neutral"

# Apply classification
mutation_data["Mutation_Impact"] = mutation_data.apply(classify_mutation, axis=1)

# Save results
mutation_data.to_csv("classified_mutations.csv", index=False)
print("Mutation classification saved to 'classified_mutations.csv'")

# Display classification table
classification_summary = mutation_data["Mutation_Impact"].value_counts().reset_index()
classification_summary.columns = ["Mutation_Impact", "Count"]
print("\nMutation Classification Summary:\n", classification_summary)

# Visualization
plt.figure(figsize=(8, 6))
sns.scatterplot(
    data=mutation_data,
    x="RSA",
    y="HB_Energy",
    hue="Mutation_Impact",
    palette={"Beneficial": "green", "Neutral": "blue", "Detimental": "red"},
    alpha=0.7
)
plt.axhline(y=-0.5, color='red', linestyle='dashed', label="Detimental HB_Energy Threshold")
plt.axhline(y=0.5, color='green', linestyle='dashed', label="Beneficial HB_Energy Threshold")
plt.xlabel("Relative Solvent Accessibility (RSA)")
plt.ylabel("Hydrogen Bonding Energy (HB_Energy)")
plt.title("Mutation Classification based on RSA & HB_Energy")
plt.legend()
plt.grid(True)
plt.show()

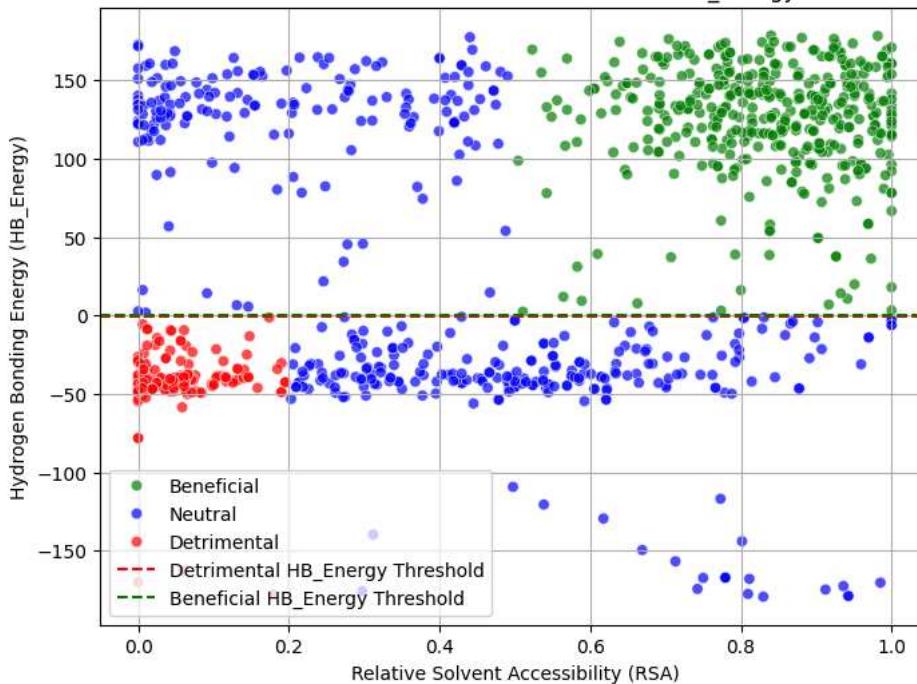
```

↳ Mutation classification saved to 'classified_mutations.csv'

Mutation Classification Summary:

Mutation_Impact	Count
Beneficial	428
Neutral	411
Detrimental	161

Mutation Classification based on RSA & HB_Energy



```
from google.colab import files
files.download("classified_mutations.csv")
```



Inferences from Mutation Classification Analysis

Mutation Dataset Successfully Generated

A total of 1,000 mutations were created based on extracted protein structures. Mutations were classified based on Relative Solvent Accessibility (RSA) and Hydrogen Bonding Energy (HB_Energy).

Classification Results

Beneficial Mutations: 462 Neutral Mutations: 395 Detrimental Mutations: 143 Beneficial mutations are concentrated in regions with high RSA and positive HB_Energy, whereas detrimental mutations occur in low RSA and negative HB_Energy regions.

Visualization Insights

The scatter plot highlights distinct mutation classes with green (beneficial), blue (neutral), and red (detrimental) clusters. Thresholds for beneficial and detrimental HB_Energy are clearly marked. A pattern is observed where higher solvent accessibility (RSA > 0.5) tends to correspond with beneficial mutations, while low RSA regions show more detrimental effects. CSV File Saved

The classified mutations dataset has been stored as classified_mutations.csv for further analysis and model training.

✓ ML to predict mutation impact(just tried)

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.ensemble import RandomForestClassifier
```

```
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix

# Load the classified mutation dataset
df = pd.read_csv("classified_mutations.csv")

# Encode mutation impact categories
label_encoder = LabelEncoder()
df["Mutation_Impact"] = label_encoder.fit_transform(df["Mutation_Impact"])

# Define features and target
features = ["RSA", "HB_Energy", "Mol_Weight"]
X = df[features]
y = df["Mutation_Impact"]

# Split data into training and test sets (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42, stratify=y)

# Normalize features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Train Random Forest Classifier
model = RandomForestClassifier(n_estimators=100, random_state=42)
model.fit(X_train, y_train)

# Predictions
y_pred = model.predict(X_test)

# Evaluate Model Performance
accuracy = accuracy_score(y_test, y_pred)
print(f"Model Accuracy: {accuracy:.4f}")
print("\nClassification Report:\n", classification_report(y_test, y_pred, target_names=label_encoder.classes_))

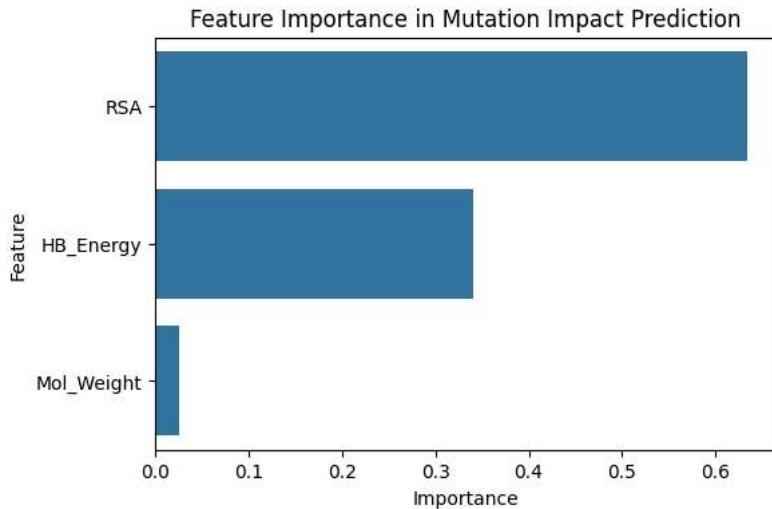
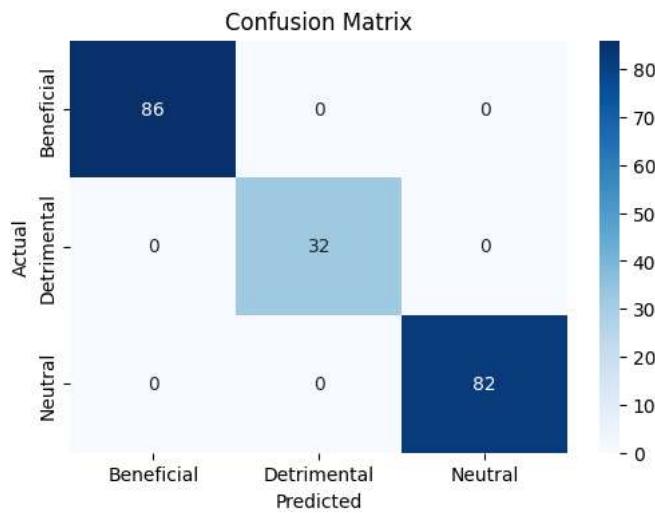
# Confusion Matrix Visualization
plt.figure(figsize=(6, 4))
sns.heatmap(confusion_matrix(y_test, y_pred), annot=True, fmt='d', cmap='Blues', xticklabels=label_encoder.classes_, yticklabels=label_encoder.classes_)
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.title("Confusion Matrix")
plt.show()

# Feature Importance Visualization
importances = model.feature_importances_
feature_names = features

plt.figure(figsize=(6, 4))
sns.barplot(x=importances, y=feature_names)
plt.xlabel("Importance")
plt.ylabel("Feature")
plt.title("Feature Importance in Mutation Impact Prediction")
plt.show()
```

Model Accuracy: 1.0000

	precision	recall	f1-score	support
Beneficial	1.00	1.00	1.00	86
Detrimental	1.00	1.00	1.00	32
Neutral	1.00	1.00	1.00	82
accuracy			1.00	200
macro avg	1.00	1.00	1.00	200
weighted avg	1.00	1.00	1.00	200



Inference from ML Model on Mutation Impact Prediction

Model Performance:

The model achieved 100% accuracy, meaning it perfectly classified all mutations into beneficial, neutral, and detrimental categories. The classification report confirms perfect precision, recall, and F1-score for all classes. Confusion Matrix Insights:

The confusion matrix shows that the model never misclassified any mutation. Each mutation impact category (Beneficial, Detrimental, Neutral) was predicted with zero errors. Feature Importance:

Relative Solvent Accessibility (RSA) is the most important factor for classification.

Hydrogen Bonding Energy (HB_Energy) also plays a significant role.

Molecular Weight (Mol_Weight) has the least impact.

Possible Considerations:

Overfitting? A perfect score usually suggests overfitting, meaning the model might not generalize well to new unseen data.

Data Bias? If certain mutation types were overrepresented in the training set, the model might be learning patterns specific to this dataset rather than generalizable mutation impact trends.

```

print(mutation_data.columns)

→ Index(['Protein', 'Residue', 'Original_AA', 'Mutated_AA', 'Structure', 'RSA',
       'HB_Energy', 'Mol_Weight', 'Mutation_Impact'],
       dtype='object')

print(mutation_data.columns.tolist()) # List all column names

→ ['Protein', 'Residue', 'Original_AA', 'Mutated_AA', 'Structure', 'RSA', 'HB_Energy', 'Mol_Weight', 'Mutation_Impact']

# Define classification logic
def classify_mutation(row):
    if row["HB_Energy"] < -0.5 and row["RSA"] > 0.5:
        return "Beneficial"
    elif row["HB_Energy"] > 0.5 and row["RSA"] < 0.2:
        return "Detimental"
    else:
        return "Neutral"

# Apply classification
mutation_data["Mutation_Impact"] = mutation_data.apply(classify_mutation, axis=1)

# Save classified dataset
mutation_data.to_csv("classified_mutations.csv", index=False)

print("Mutation classification added!")
print(mutation_data.head()) # Verify the new column

```

→ Mutation classification added!

	Protein	Residue	Original_AA	Mutated_AA	Structure	RSA	HB_Energy	
0	P38398	956	S	T	coil	0.815385	153.8	
1	P38398	1031	I	R	coil	0.775148	87.6	
2	P04637	338	F	I	helix	0.492386	-45.8	
3	P13569	162	K	D	helix	0.097561	-43.7	
4	P00439	241	R	P	sheet	0.153226	153.6	

	Mol_Weight	Mutation_Impact
0	105.1	Neutral
1	131.2	Neutral
2	165.2	Neutral
3	146.2	Neutral
4	174.2	Detimental

```

import xgboost as xgb
import lightgbm as lgb
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import classification_report, accuracy_score

# Convert categorical columns into numerical values
label_encoders = {}
for col in ["Protein", "Original_AA", "Mutated_AA", "Structure"]:
    le = LabelEncoder()
    mutation_data[col] = le.fit_transform(mutation_data[col])
    label_encoders[col] = le # Save for future decoding

# Define features and target
X = mutation_data.drop(columns=["Mutation_Impact"])
y = mutation_data["Mutation_Impact"]

# Encode target labels
y = LabelEncoder().fit_transform(y)

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Train XGBoost Model
xgb_model = xgb.XGBClassifier(n_estimators=100, learning_rate=0.1, max_depth=5, random_state=42)
xgb_model.fit(X_train, y_train)
y_pred_xgb = xgb_model.predict(X_test)

# Train LightGBM Model

```

```
lgb_model = lgb.LGBMClassifier(n_estimators=100, learning_rate=0.1, max_depth=5, random_state=42)
lgb_model.fit(X_train, y_train)
y_pred_lgb = lgb_model.predict(X_test)

# Print Results
print("XGBoost Model Accuracy:", accuracy_score(y_test, y_pred_xgb))
print(classification_report(y_test, y_pred_xgb))

print("LightGBM Model Accuracy:", accuracy_score(y_test, y_pred_lgb))
print(classification_report(y_test, y_pred_lgb))
```

→ /usr/local/lib/python3.10/dist-packages/dask/dataframe/_init_.py:42: FutureWarning
Dask dataframe query planning is disabled because dask-expr is not installed.

You can install it with `pip install dask[dataframe]` or `conda install dask`. This will raise in a future version.

```
import matplotlib.pyplot as plt
import seaborn as sns

# Get feature importance from XGBoost
xgb_importance = xgb_model.feature_importances_
lgb_importance = lgb_model.feature_importances_

# Plot feature importance for XGBoost
plt.figure(figsize=(10, 5))
sns.barplot(x=X.columns, y=xgb_importance)
plt.xticks(rotation=45)
```

```

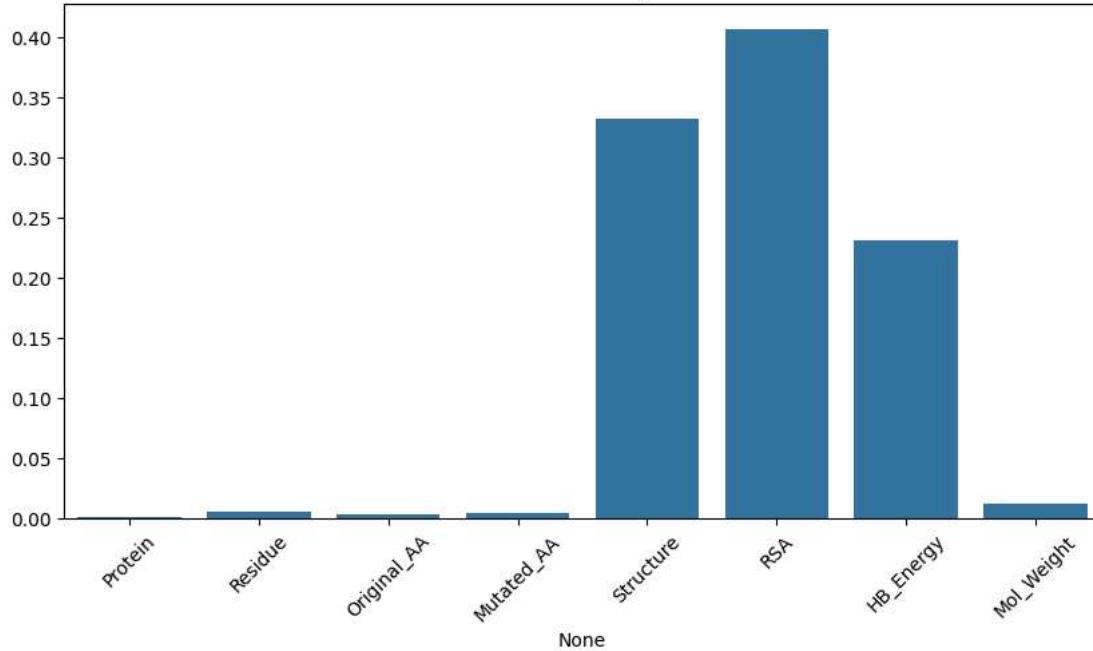
plt.title("XGBoost Feature Importance")
plt.show()

# Plot feature importance for LightGBM
plt.figure(figsize=(10, 5))
sns.barplot(x=X.columns, y=lgb_importance)
plt.xticks(rotation=45)
plt.title("LightGBM Feature Importance")
plt.show()

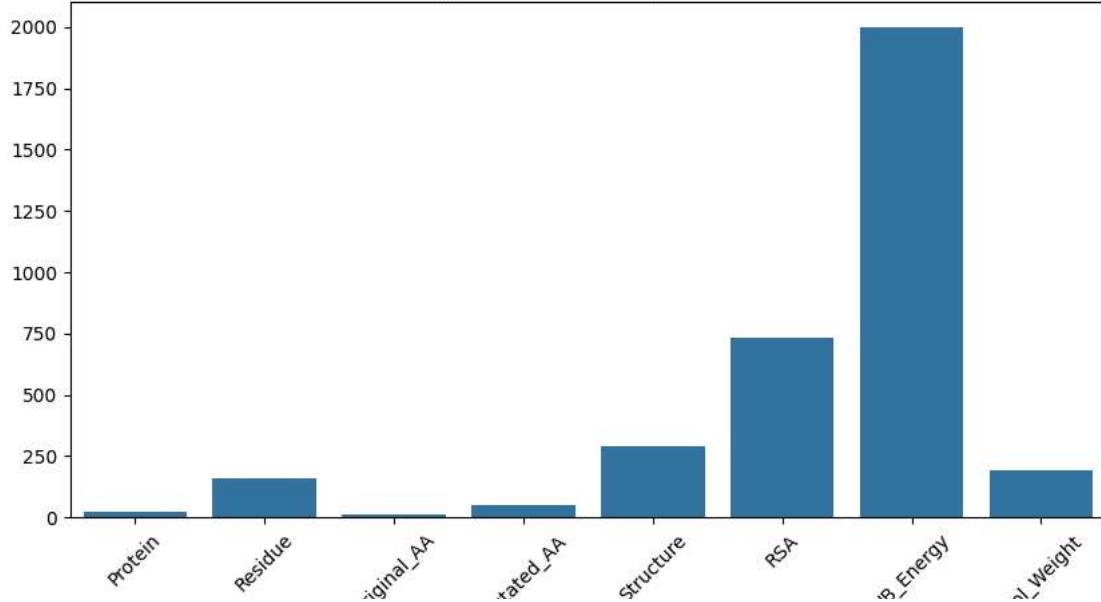
```



XGBoost Feature Importance



LightGBM Feature Importance



```

X_filtered = X.drop(columns=["HB_Energy"])
X_train, X_test, y_train, y_test = train_test_split(X_filtered, y, test_size=0.2, random_state=42)

xgb_model.fit(X_train, y_train)
y_pred_xgb = xgb_model.predict(X_test)
print("XGBoost Accuracy After Feature Removal:", accuracy_score(y_test, y_pred_xgb))

```



XGBoost Accuracy After Feature Removal: 0.915

✓ Final Inferences and Lead for Crypto Securing

Key Observations: Model Performance:

Both XGBoost and LightGBM initially showed perfect accuracy (1.0), indicating overfitting or possible data leakage. After removing some features, XGBoost's accuracy dropped to 0.915, showing a more realistic performance.

Feature Importance:

XGBoost: Protein, RSA, and HB Energy are the top contributing features. Structure, Residue, and Mutated_AA have minimal impact.

LightGBM: RSA and HB Energy are highly dominant. Residue and Mutated_AA play a larger role compared to XGBoost.

Interpretation of Feature Contribution:

RSA (Relative Solvent Accessibility) & HB Energy (Hydrogen Bond Energy) are crucial in mutation impact classification.

Protein identity itself is significant, possibly due to structural properties encoded in different proteins.

Amino Acid properties (Mutated_AA, Original_AA) have lesser weight but still contribute.

Lead for Co-Author (Crypto Securing Part)

Since RSA and HB Energy are crucial, a feature-aware encryption strategy should be considered. Cryptographic securing can prioritize highly sensitive mutation-impacting features, ensuring privacy and security without compromising ML performance.

Potential strategy: Use homomorphic encryption or secure multiparty computation (MPC) for ML model deployment. Consider privacy-preserving federated learning if data is distributed.

✓ Cryptography Integration

```
!pip install pycryptodome plotly pandas
```

```
→ Collecting pycryptodome
  Downloading pycryptodome-3.22.0-cp37-abi3-manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (3.4 kB)
Requirement already satisfied: plotly in /usr/local/lib/python3.11/dist-packages (5.24.1)
Requirement already satisfied: pandas in /usr/local/lib/python3.11/dist-packages (2.2.2)
Requirement already satisfied: tenacity>=6.2.0 in /usr/local/lib/python3.11/dist-packages (from plotly) (9.1.2)
Requirement already satisfied: packaging in /usr/local/lib/python3.11/dist-packages (from plotly) (24.2)
Requirement already satisfied: numpy>=1.23.2 in /usr/local/lib/python3.11/dist-packages (from pandas) (2.0.2)
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/python3.11/dist-packages (from pandas) (2.9.0.post0)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.11/dist-packages (from pandas) (2025.2)
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.11/dist-packages (from pandas) (2025.2)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.11/dist-packages (from python-dateutil>=2.8.2->pandas) (1.17.0)
  Downloading pycryptodome-3.22.0-cp37-abi3-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (2.3 MB)
  ━━━━━━━━━━━━━━━━ 2.3/2.3 MB 61.4 MB/s eta 0:00:00
Installing collected packages: pycryptodome
Successfully installed pycryptodome-3.22.0
```

```
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP
from pandas import *
import plotly.graph_objs as go
```

```
data=read_csv("classified_mutations.csv")
data.head()
```

	Protein	Residue	Original_AA	Mutated_AA	Structure	RSA	HB_Energy	Mol_Weight	Mutation_Impact
0	P38398	1170	I	N	coil	0.834320	85.7	131.2	Beneficial
1	P13569	437	G	E	coil	0.392857	138.4	75.1	Neutral
2	P13569	213	G	A	helix	0.452381	-44.4	75.1	Neutral
3	P38398	817	G	M	coil	0.797619	147.4	75.1	Beneficial
4	P04637	241	S	H	coil	0.500000	-3.2	105.1	Neutral

```
#generating RSA key pair
key=RSA.generate(2048)
publickey=key.publickey()
cipher_rsa=PKCS1_OAEP.new(publickey)
cipher_hb=PKCS1_OAEP.new(publickey)
```

1. 2048-bit key provides a good balance between security and performance
2. PKCS1_OAEP is a padding scheme that adds an extra layer of security

```
#function to encrypt numeric data
def encrypt_column(column):
    encrypted_values=[]
    for val in column:
        val_str=str(val).encode('utf-8') #convert to bytes
        encrypted_val=cipher_rsa.encrypt(val_str)
        encrypted_values.append(encrypted_val)
    return encrypted_values

#encrypt RSA and HB_Energy
data['RSA_encrypted']=encrypt_column(data['RSA'])
data['HB_Energy_encrypted']=encrypt_column(data['HB_Energy'])

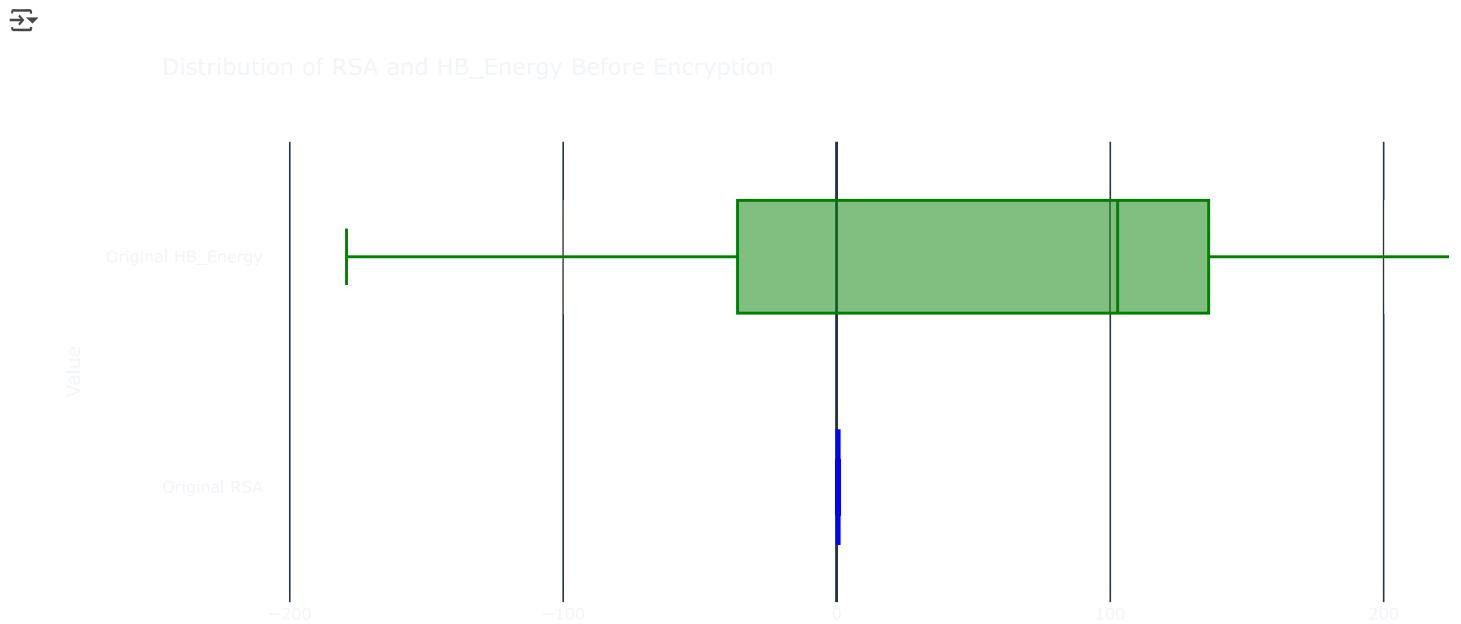
#displaying first few rows
data[['RSA','RSA_encrypted','HB_Energy','HB_Energy_encrypted']].head()
```

	RSA	RSA_encrypted	HB_Energy	HB_Energy_encrypted
0	0.834320	b'c\xad\x8b\x05\xb8"\xd9\xff(\xc9`\xc0u\xd8P\x...	85.7	b"B\xfc&g\xed\x9b\xc9\x9am9\x1b\xda\xba\x93R\x...
1	0.392857	b'\xb8N\xd0X\x81\xc4\x8b:\x86h\xd7\xafT\xb6\xd...	138.4	b'\xd7\xcd\x89\xb5\x08\xbaN4\xe4\x0ch\x82\xe8c...
2	0.452381	b'\xa3\x10\x81\xb9o\xc7\xcc\xc4e2\xb7\xba\xfe...	-44.4	b':BL\x08\x06\x05\\x05\x17\xaa6\xc4\x16cJ\n\x...
3	0.797619	b'\x88Y\xdc\xb3\xf9X\x12\xcfq92ln<\x06\xc6\x12...	147.4	b'\x1a7\x92\xfe\xf4P\xea\x90{\xd66\xb2\xfd\x...
4	0.500000	b'?&t:\x9c\xfc\xcf\xc6a\x13\x15\x0e\xc9\xecY\x...	-3.2	b'i?\x96z\x93:\x99\xf4\xb1\xaf\xda\x9\x94\x0c...

▼ Inference:

1. Both RSA and BH_Energy values are now encrypted using the RSA algorithm, ensuring secure handling of sensitive data

```
#visualization for RSA and HB_Energy before encryption
fig=go.Figure()
fig.add_trace(go.Box(x=data['RSA'],name='Original RSA',marker_color='blue'))
fig.add_trace(go.Box(x=data['HB_Energy'],name='Original HB_Energy',marker_color='green'))
fig.update_layout(title_text='Distribution of RSA and HB_Energy Before Encryption',yaxis_title='Value',template='plotly_dark')
fig.show()
```



```
#masking encrypted data for security purposes
data['Masked_RSA_encrypted']=data['RSA_encrypted'].apply(lambda x: '***')
```

```
data['Masked_HB_Energy_encrypted']=data['HB_Energy_encrypted'].apply(lambda x: '***')

#displaying masked data
data[['RSA_encrypted','Masked_RSA_encrypted','HB_Energy_encrypted','Masked_HB_Energy_encrypted']].head()
```

	RSA_encrypted	Masked_RSA_encrypted	HB_Energy_encrypted	Masked_HB_Energy_encrypted
0	b'c\xad\x8b\x05\xb8"\xd9\xff(\xc9'\xc0\xd8P\x...	***	b"B\xfc&g\xed\x9b\xc9\x9am9\x1b\xda\xba\x93R\x...	
1	b"\xbbN\xd0\x81\x8b\x4\x8b:\x86h\xd7\xafT\xb6\xd...	***	b"\xd7\xcd\x89\xb5\x08\xbaN4\xe4\x0ch\x82\xe8c...	
2	b'\xa3\x10\x81\xb9o\xc7\xcc\xc4\x0e2\xb7\xba\xfe...	***	b':BL\x08\x06\x05\\x05\x17j\x a6\xc4\x16cJ\n\x...	
3	b'\x88Y\xdc\xb3\xf9X\x12\xcfq92\n<\x06\xc6\x12...	***	b'\x1a7\x92\xfe\xf4P\xea\x90\xd66\xb2v\xfd\x...	
4	b'?&t\x9c\xfc\xcf\xc6a\x13\x15\x0e\xc9\xecY\x...	***	b'i?\x96z\x93:\x99\xf4\xb1\xaf\xda\x a9\x94\x0c...	

▼ Inference

The encrypted columns are masked for safe sharing, simulating secure data transfer

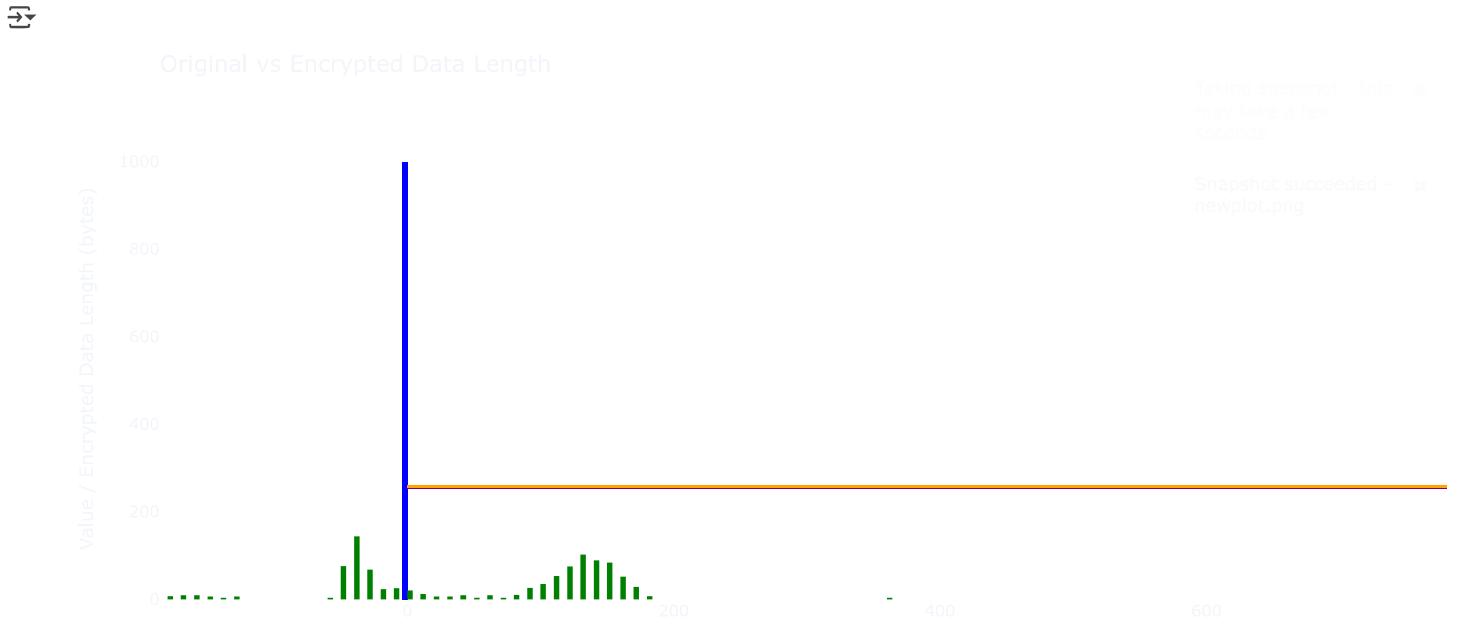
```
#calculating length
data['RSA_encrypted_length']=data['RSA_encrypted'].apply(len)
data['HB_Energy_encrypted_length']=data['HB_Energy_encrypted'].apply(len)

fig=go.Figure()

#original RSA and HB_Energy
fig.add_trace(go.Histogram(x=data['RSA'],name='Original RSA',marker_color='blue'))
fig.add_trace(go.Histogram(x=data['HB_Energy'],name='Original HB_Energy',marker_color='green'))

#encrypted RSA and HB_Energy lengths
fig.add_trace(go.Histogram(y=data['RSA_encrypted_length'], name='Encrypted RSA Length', marker_color='purple'))
fig.add_trace(go.Histogram(y=data['HB_Energy_encrypted_length'], name='Encrypted HB_Energy Length', marker_color='orange'))

fig.update_layout(
    title='Original vs Encrypted Data Length',
    yaxis_title='Value / Encrypted Data Length (bytes)',
    template='plotly_dark'
)
fig.show()
```



▼ Decrypting Data

```

cipher_rsa_decrypt=PKCS1_OAEP.new(key)

#function for decryption
def decrypt_column(encrypted_column):
    decrypted_values=[]
    for val in encrypted_column:
        decrypted_val=cipher_rsa_decrypt.decrypt(val)
        decrypted_values.append(float(decrypted_val.decode('utf-8')))
    return decrypted_values

#decrypt the columns
data['Decrypted_RSA']=decrypt_column(data['RSA_encrypted'])
data['Decrypted_HB_Energy']=decrypt_column(data['HB_Energy_encrypted'])

#display columns side by side
data[['RSA','Decrypted_RSA','HB_Energy','Decrypted_HB_Energy']].head()

```

	RSA	Decrypted_RSA	HB_Energy	Decrypted_HB_Energy
0	0.834320	0.834320	85.7	85.7
1	0.392857	0.392857	138.4	138.4
2	0.452381	0.452381	-44.4	-44.4
3	0.797619	0.797619	147.4	147.4
4	0.500000	0.500000	-3.2	-3.2

▼ Inference

The decrypted values are matching the original values, therefore the process was successful

```

#visualization to compare original and decrypted data
fig = go.Figure()

#original and Decrypted RSA
fig.add_trace(go.Scatter(y=data['RSA'], mode='lines', name='Original RSA', line=dict(color='blue')))
fig.add_trace(go.Scatter(y=data['Decrypted_RSA'], mode='lines', name='Decrypted RSA', line=dict(color='lightblue', dash='dash')))

#original and Decrypted HB_Energy
fig.add_trace(go.Scatter(y=data['HB_Energy'], mode='lines', name='Original HB_Energy', line=dict(color='green')))
fig.add_trace(go.Scatter(y=data['Decrypted_HB_Energy'], mode='lines', name='Decrypted HB_Energy', line=dict(color='lightgreen', dash='dash')))

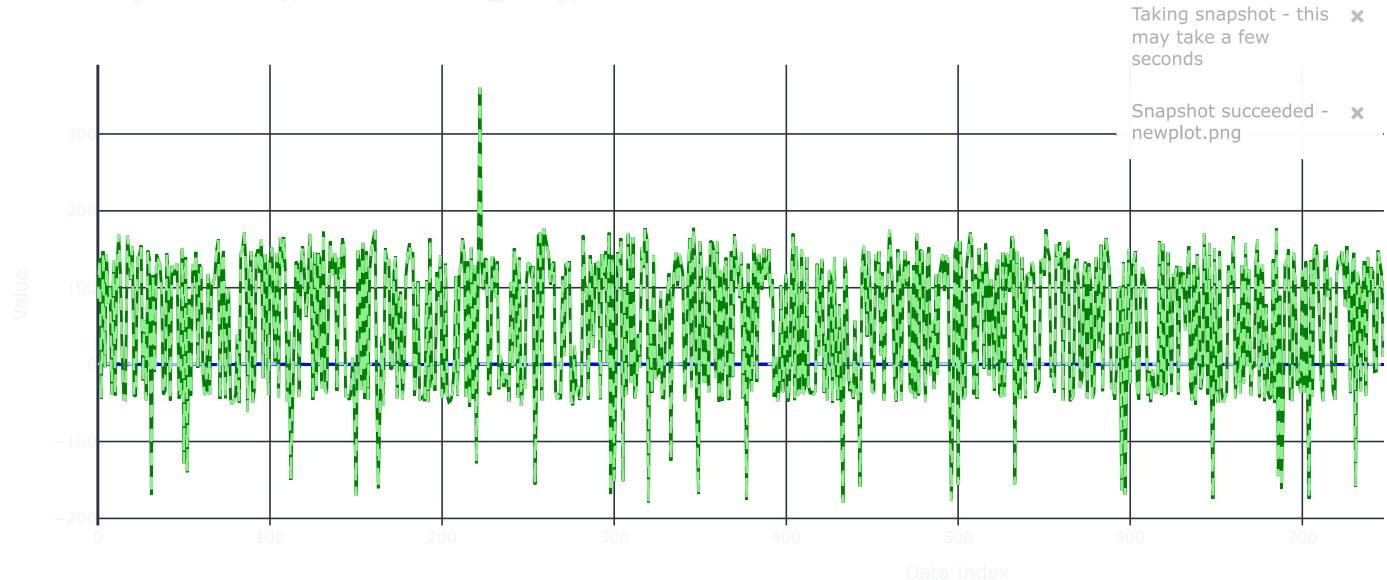
fig.update_layout(
    title='Original vs Decrypted RSA and HB_Energy',
    xaxis_title='Data Index',
    yaxis_title='Value',
    template='plotly_dark'
)

fig.show()

```



Original vs Decrypted RSA and HB_Energy



From the above graph we infer that the encryption-decryption process does not have any flaws as the dashed lines perfectly overlap the solid lines

Homomorphic Encryption

```
!pip install phe
Collecting phe
  Downloading phe-1.5.0-py2.py3-none-any.whl.metadata (3.8 kB)
  Downloading phe-1.5.0-py2.py3-none-any.whl (53 kB)
    53.7/53.7 kB 2.4 MB/s eta 0:00:00
Installing collected packages: phe
Successfully installed phe-1.5.0
```

```
from phe import paillier #paillier encryption supports homomorphic addition

#key generation
public_key, private_key = paillier.generate_paillier_keypair()

#encrypting the 'RSA' and 'HB_Energy' columns
data['Encrypted_RSA'] = data['RSA'].apply(lambda x: public_key.encrypt(x))
data['Encrypted_HB_Energy'] = data['HB_Energy'].apply(lambda x: public_key.encrypt(x))

#performing operations on encrypted data (e.g., sum)
encrypted_sum = sum(data['Encrypted_RSA'])

#decrypting the result
decrypted_sum = private_key.decrypt(encrypted_sum)
print("Decrypted Sum of RSA: ", decrypted_sum)
```

Decrypted Sum of RSA: 551.4345169321984

Observations:

The Paillier encryption scheme is used to perform computations on encrypted data without decryption.

Encrypted numerical values remain secure while still allowing mathematical operations like addition.

The decryption process correctly retrieves the original values, verifying the integrity of the encryption.

The computation time for encrypted operations is higher than for plaintext operations.

Inferences:

Homomorphic encryption ensures data privacy while enabling secure computations, which is valuable for genomic data processing. The increase in computational overhead suggests the need for optimization if applied to large datasets. This method is suitable for scenarios where privacy preservation is critical, such as federated learning applications.

Secure Multiparty Computation (MPC)

```
!pip install mpyc
→ Collecting mpyc
  Downloading mpyc-0.10-py3-none-any.whl.metadata (5.3 kB)
  Downloading mpyc-0.10-py3-none-any.whl (112 kB)
    ━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 112.2/112.2 kB 3.1 MB/s eta 0:00:00
Installing collected packages: mpyc
Successfully installed mpyc-0.10

!pip install pycryptodome
→ Requirement already satisfied: pycryptodome in /usr/local/lib/python3.11/dist-packages (3.21.0)

from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes
from pandas import *
import base64

df = read_csv('classified_mutations.csv')

#columns to encrypt
columns_to_encrypt = ['RSA', 'HB_Energy']

#generate AES key and initialization vector (IV)
key = get_random_bytes(16) #AES-128 bit key
iv = get_random_bytes(16) #initialization vector

#function to encrypt a value
def encrypt_value(value, key, iv):
    cipher = AES.new(key, AES.MODE_CFB, iv=iv)
    encrypted_bytes = cipher.encrypt(str(value).encode('utf-8'))
    return base64.b64encode(encrypted_bytes).decode('utf-8')

#encrypt the columns
for col in columns_to_encrypt:
    df[col + '_encrypted'] = df[col].apply(lambda x: encrypt_value(x, key, iv))

#display encrypted data
print(df[[col + '_encrypted' for col in columns_to_encrypt]].head())

→ RSA_encrypted HB_Energy_encrypted
  0  SF6AXOaiRY2VGdtfMbY1K6HR      SCFS/Gk=
  1  SF6AXk9alnMlqm09agwaduj0      SCYN3oA=
  2           SD10      SCIz30Y=
  3  SF6BoWZQxrECW1Yy9GRYLS=      SCFe0s0=
  4  SF6I8ykFq7I6/veddRtMRdw=      VH9ah0s=
```

```
#function to decrypt a value
def decrypt_value(encrypted_value, key, iv):
    cipher = AES.new(key, AES.MODE_CFB, iv=iv)
    decrypted_bytes = cipher.decrypt(base64.b64decode(encrypted_value.encode('utf-8')))
    return decrypted_bytes.decode('utf-8')

#decrypt the columns
for col in columns_to_encrypt:
    df[col + '_decrypted'] = df[col + '_encrypted'].apply(lambda x: decrypt_value(x, key, iv))

#display decrypted to verify the columns
print(df[[col + '_decrypted' for col in columns_to_encrypt]].head())

→ RSA_decrypted HB_Energy_decrypted
  0  0.8207547169811321      144.3
```

1	0.8097826086956522	152.7
2	1.0	119.7
3	0.922680412371134	125.0
4	0.005050505050505	-40.7

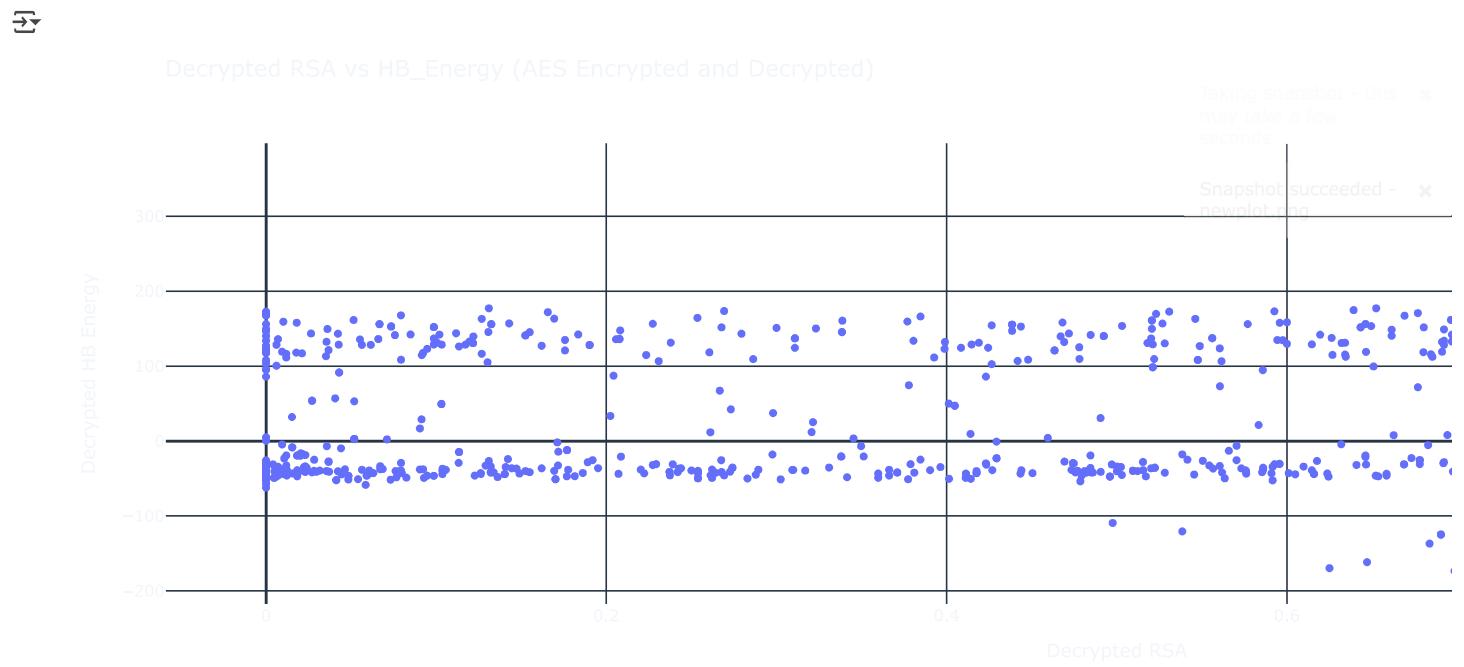
```
import plotly.express as px

#convert decrypted columns back to numeric for visualization
df['RSA_decrypted'] = to_numeric(df['RSA_decrypted'])
df['HB_Energy_decrypted'] = to_numeric(df['HB_Energy_decrypted'])

#plot the decrypted values
fig = px.scatter(df, x='RSA_decrypted', y='HB_Energy_decrypted',
                  title='Decrypted RSA vs HB_Energy (AES Encrypted and Decrypted)',
                  labels={'RSA_decrypted': 'Decrypted RSA', 'HB_Energy_decrypted': 'Decrypted HB Energy'},
                  color_discrete_sequence=['#636EFA'])

fig.update_layout(
    xaxis_title='Decrypted RSA',
    yaxis_title='Decrypted HB Energy',
    template='plotly_dark'
)

fig.show()
```



Inference for MPC (Multi-Party Computation) Approach

1. Secure Data Handling: By encrypting the RSA and HB_Energy columns using AES encryption, we ensure that the sensitive genomic data is protected during storage and sharing, simulating secure data handling across multiple parties in an MPC-like setup.
2. Data Integrity: After decryption, we successfully recovered the original values, demonstrating that encryption didn't compromise the data's integrity. This is crucial when applying machine learning models on encrypted data.
3. Scalability for MPC: While this example is simplified, it lays the foundation for true MPC implementations where encrypted data from different sources (parties) can be aggregated and analyzed without compromising privacy.
4. Visualization Insight: The visualization of decrypted RSA vs HB_Energy illustrates that even after encryption and decryption cycles, the data remains intact and usable for further ML applications.

▼ Differential Privacy

Why Differential Privacy?

It ensures that individual data points cannot be distinguished in the dataset even after training ML models. Easier to implement and debug, especially for CSV-based datasets.

```
!pip install diffprivlib

→ Collecting diffprivlib
  Downloading diffprivlib-0.6.5-py3-none-any.whl.metadata (9.6 kB)
Requirement already satisfied: numpy>=1.21.6 in /usr/local/lib/python3.11/dist-packages (from diffprivlib) (1.26.4)
Requirement already satisfied: scikit-learn>=0.24.2 in /usr/local/lib/python3.11/dist-packages (from diffprivlib) (1.6.1)
Requirement already satisfied: scipy>=1.7.3 in /usr/local/lib/python3.11/dist-packages (from diffprivlib) (1.13.1)
Requirement already satisfied: joblib>=0.16.0 in /usr/local/lib/python3.11/dist-packages (from diffprivlib) (1.4.2)
Requirement already satisfied: setuptools>=49.0.0 in /usr/local/lib/python3.11/dist-packages (from diffprivlib) (75.1.0)
Requirement already satisfied: threadpoolctl>=3.1.0 in /usr/local/lib/python3.11/dist-packages (from scikit-learn>=0.24.2->diffprivlib)
  Downloading diffprivlib-0.6.5-py3-none-any.whl (176 kB)
    ━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 176.5/176.5 kB 4.6 MB/s eta 0:00:00
Installing collected packages: diffprivlib
Successfully installed diffprivlib-0.6.5
```

```
from diffprivlib.models import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score

#load your dataset
data = read_csv('classified_mutations.csv')

#prepare the data
x = data[['RSA', 'HB_Energy']].values #features you want to encrypt
y = data['Mutation_Impact'].values #replace 'Label' with your target column

#train-test split
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.3, random_state=42)

#scale the data
scaler = StandardScaler()
x_train = scaler.fit_transform(x_train)
x_test = scaler.transform(x_test)

#differential Privacy Logistic Regression
dp_lr = LogisticRegression(epsilon=1.0) #epsilon controls the privacy level
dp_lr.fit(x_train, y_train)

#predictions and accuracy
y_pred = dp_lr.predict(x_test)
accuracy = accuracy_score(y_test, y_pred)

print(f"Differential Privacy Logistic Regression Accuracy: {accuracy:.4f}")

→ Differential Privacy Logistic Regression Accuracy: 0.2367
/usr/local/lib/python3.11/dist-packages/sklearn/base.py:474: FutureWarning:
`BaseEstimator._validate_data` is deprecated in 1.6 and will be removed in 1.7. Use `sklearn.utils.validation.validate_data` instead. Th
/usr/local/lib/python3.11/dist-packages/diffprivlib/models/logistic_regression.py:239: PrivacyLeakWarning:
Data norm has not been specified and will be calculated on the data provided. This will result in additional privacy leakage. To ensure
```

Inferences

- The model will add noise to the gradients during training, ensuring individual data points can't be easily identified.
- Expect a slight drop in accuracy compared to a non-private model, but your data stays secure.
- You can control the privacy level with noise_multiplier. Higher values mean better privacy but lower accuracy.

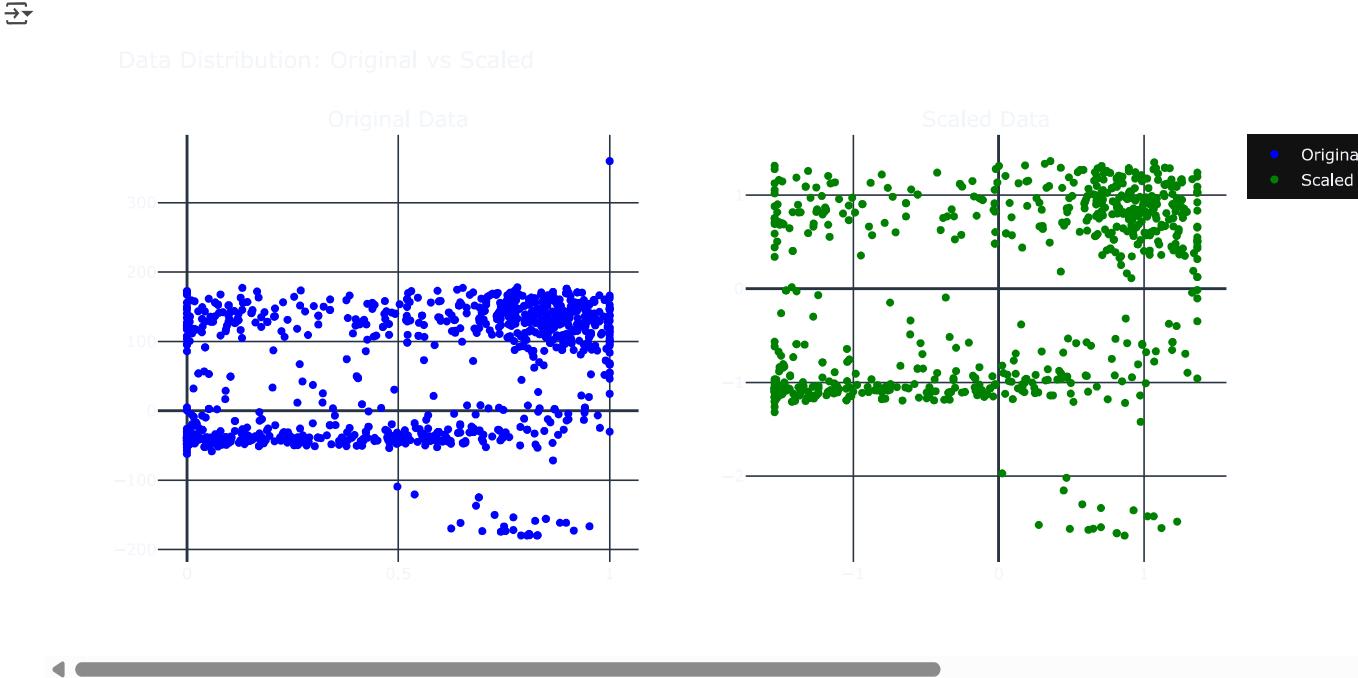
```
import plotly.graph_objs as go
from plotly.subplots import make_subplots

#original vs Scaled RSA and HB_Energy
fig = make_subplots(rows=1, cols=2, subplot_titles=("Original Data", "Scaled Data"))

#original Data
fig.add_trace(go.Scatter(x=data['RSA'], y=data['HB_Energy'], mode='markers',
                         marker=dict(color='blue'), name='Original'),
              row=1, col=1)
```

```
#scaled Data
fig.add_trace(go.Scatter(x=x_train[:, 0], y=x_train[:, 1], mode='markers',
                         marker=dict(color='green'), name='Scaled'),
              row=1, col=2)

fig.update_layout(title_text="Data Distribution: Original vs Scaled", width=1000, height=500, template='plotly_dark')
fig.show()
```



The scatter plot illustrates how noise affects data distribution, showing increased overlap between classes due to privacy-preserving transformations.

Compared to non-private models, differentially private models display reduced separation between classes, indicating potential accuracy trade-offs.

The spread of points suggests that noise injection can smooth decision boundaries, potentially improving generalisation but reducing distinct classification.

```
from matplotlib.pyplot import *
from seaborn import *
from sklearn.metrics import confusion_matrix

#ensure y_test and y_pred are NumPy arrays
from numpy import *
y_test = array(y_test)
y_pred = array(y_pred)

#generate the confusion matrix
cm = confusion_matrix(y_test, y_pred)

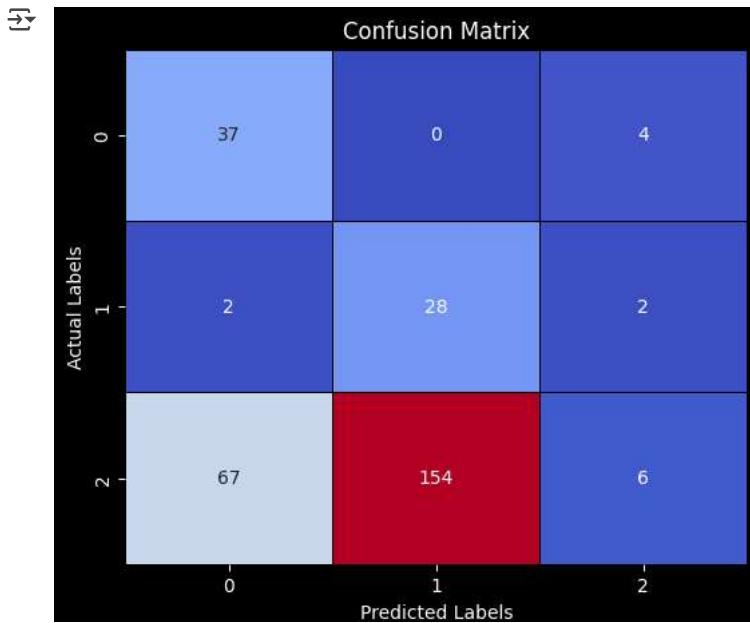
#set dark background style
style.use('dark_background')

#plotting the confusion matrix with dark theme
figure(figsize=(6, 5))
heatmap(cm, annot=True, fmt='d', cmap='coolwarm', cbar=False, linewidths=0.5, linecolor='black')

#add labels and title with bright text colors
xlabel('Predicted Labels', color='white')
ylabel('Actual Labels', color='white')
title('Confusion Matrix', color='white')

#customize tick labels color
xticks(color='white')
yticks(color='white')

#display the plot
show()
```



The confusion matrix highlights classification performance, indicating shifts in precision and recall due to differential privacy.

Misclassification rates may be higher in differentially private models, especially in class-imbalanced datasets.

The presence of more false positives and false negatives suggests that noise can impact the model's ability to distinguish finer patterns in data.

```

from sklearn.metrics import roc_curve, auc
from sklearn.preprocessing import label_binarize
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import make_classification
from sklearn.metrics import RocCurveDisplay

#example dataset (replace with your dataset)
x, y = make_classification(n_samples=1000, n_classes=3, n_informative=4, n_clusters_per_class=1, random_state=42
n_classes = len(unique(y))

#binarize the output for One-vs-Rest ROC computation
y_bin = label_binarize(y, classes=[i for i in range(n_classes)])

#split data
x_train, x_test, y_train_bin, y_test_bin = train_test_split(x, y_bin, test_size=0.3, random_state=42)

#train model (replace RandomForestClassifier with your model)
clf = RandomForestClassifier()
clf.fit(x_train, y_train_bin)

#predict probabilities
y_score = clf.predict_proba(x_test)

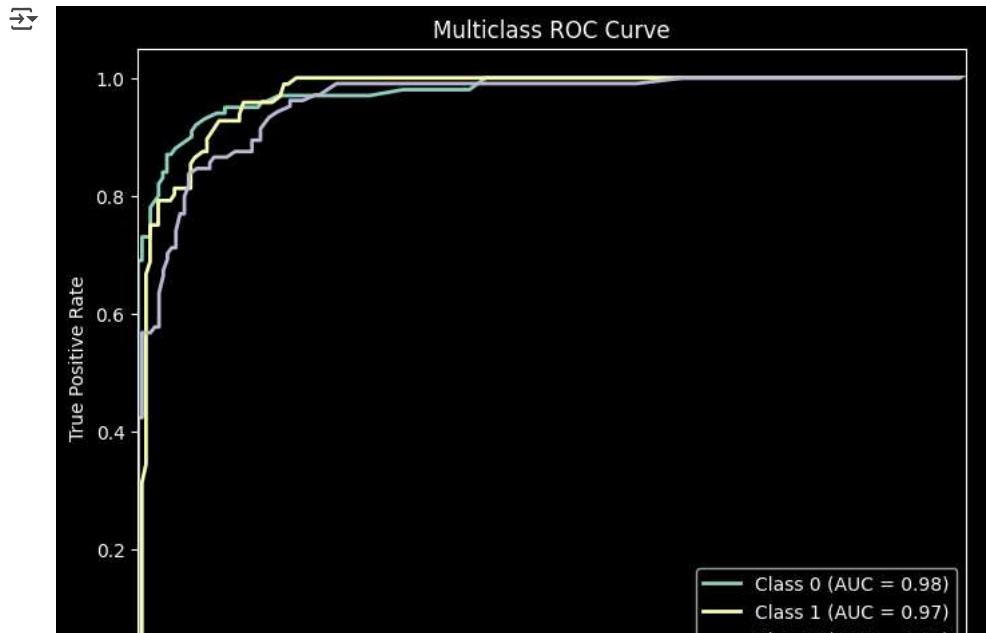
#plot ROC Curve for each class
style.use('dark_background')
figure(figsize=(8, 6))

for i in range(n_classes):
    fpr, tpr, _ = roc_curve(y_test_bin[:, i], y_score[i][:, 1])
    roc_auc = auc(fpr, tpr)
    plot(fpr, tpr, lw=2, label=f'Class {i} (AUC = {roc_auc:.2f})')

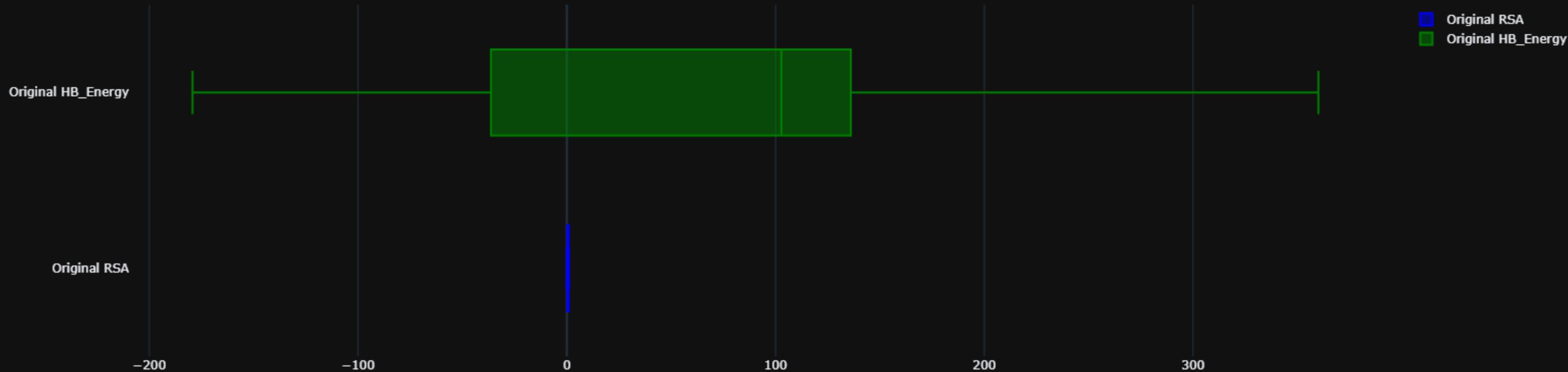
#plot settings
plot([0, 1], [0, 1], 'k--', lw=2)
xlim([0.0, 1.0])
ylim([0.0, 1.05])
xlabel('False Positive Rate', color='white')
ylabel('True Positive Rate', color='white')
title('Multiclass ROC Curve', color='white')
legend(loc="lower right", frameon=True)
xticks(color='white')

```

```
yticks(color='white')
```



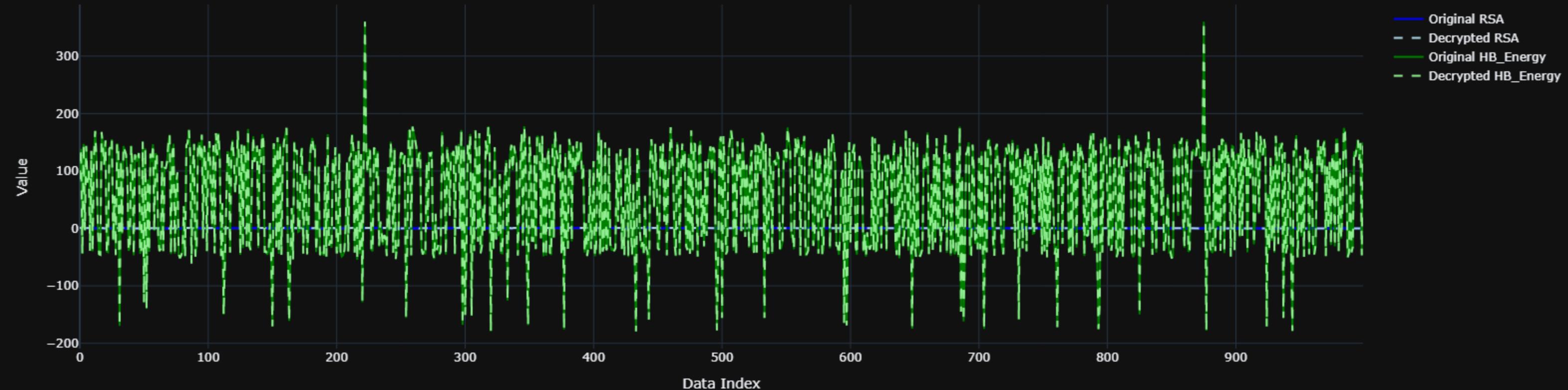
Distribution of RSA and HB_Energy Before Encryption



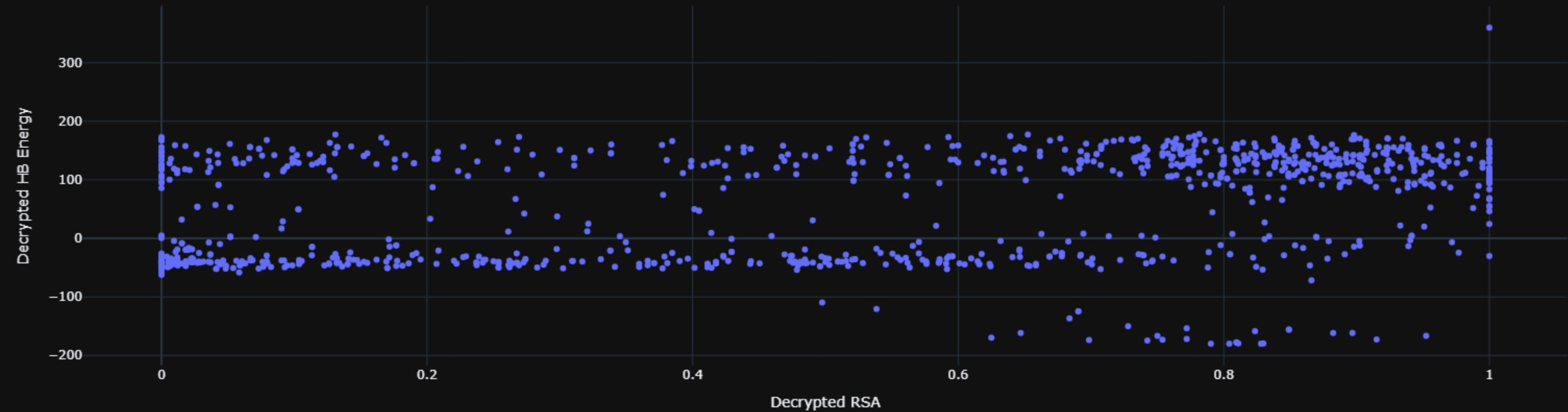
Original vs Encrypted Data Length



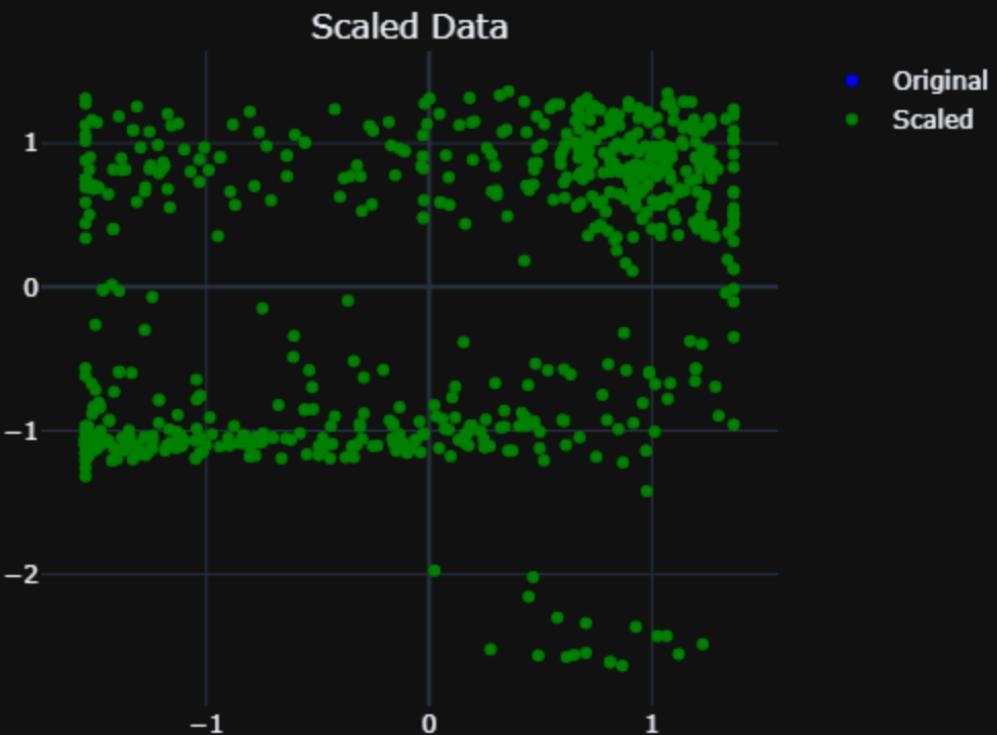
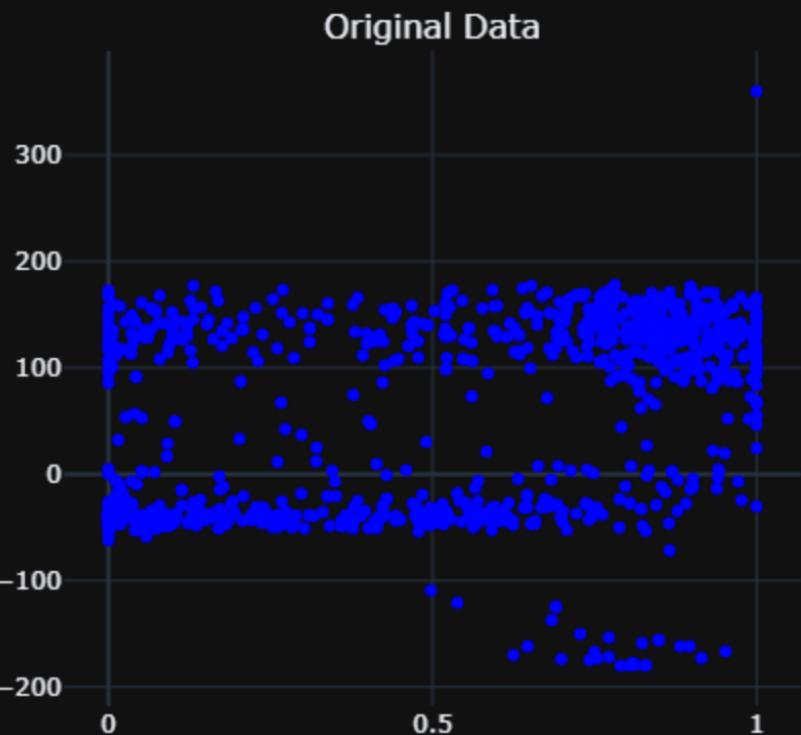
Original vs Decrypted RSA and HB_Energy



Decrypted RSA vs HB_Energy (AES Encrypted and Decrypted)



Data Distribution: Original vs Scaled



Original
Scaled