In [304…
```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.decomposition import PCA
import numpy as np
from sklearn.metrics import precision_score, recall_score, f1_score
from RevGEN_MLP import RevGEN_MLP
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from scipy.stats import gaussian_kde
from matplotlib.colors import ListedColormap
from sklearn.ensemble import IsolationForest,RandomForestClassifier
from sklearn.metrics import accuracy_score
from sklearn.model_selection import StratifiedKFold
from sklearn.model_selection import cross_val_score
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.metrics import accuracy_score, classification_report
```

RevGEN-MLP

Loading and Preparing Data for Training

In [305…
```
df = pd.read_csv("wine+quality\\winequality-white.csv", sep=";") # Insert path to dataset here
df['good'] = (df['quality'] >= 7).astype(int)
X = df.drop(['quality', 'good'], axis=1)
y = df['good']

column_names = X.columns

X = X.values
y = y.values
```

In [306…
```
# Split into train/test
X_train, X_test, y_train, y_test= train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)
# One-hot encode labels
encoder = OneHotEncoder(sparse_output=False)
y_train_encoded = encoder.fit_transform(y_train.reshape(-1, 1))
y_test_encoded =encoder.transform(y_test.reshape(-1, 1))

# Scale features
ANN_scaler = StandardScaler()
X_train_scaled = ANN_scaler.fit_transform(X_train)
X_test_scaled = ANN_scaler.transform(X_test)
```

Training RevGEN-MLP

In [307…
```
num_layer = 3
num_epochs = 101
```

In [308…
```
# Set seed for reproducibility
np.random.seed(42)

# Initialize model
model = RevGEN_MLP(
    n_layers=num_layer,
    x=X_train_scaled[0].reshape(-1, 1),
    y_actual=y_train_encoded[0].reshape(-1, 1),
    epochs=num_epochs,
    loss_function="cross_entropy"
)

# Training loop
for epoch in range(num_epochs):
    loss_epoch = 0
    correct_train = 0

    # Shuffle training indices
    indices = np.random.permutation(X_train_scaled.shape[0])

    for i in indices:
        x_sample = X_train_scaled[i].reshape(-1, 1)
        y_sample = y_train_encoded[i].reshape(-1, 1)

        # Train and accumulate loss
```

```
        model.train(input=x_sample, target=y_sample)
        loss_epoch += model.loss_fn(input=x_sample, target=y_sample)

        # Predict and count correct predictions
        pred = model.forward(x_sample)
        if np.argmax(pred[:2]) == np.argmax(y_sample):
            correct_train += 1

    # Compute average training metrics
    avg_train_loss = loss_epoch / X_train_scaled.shape[0]
    train_accuracy = correct_train / X_train_scaled.shape[0]

    # Evaluate on test set every 10 epochs
    if epoch % 10 == 0 or epoch == num_epochs - 1:
        correct_test = 0
        test_loss_epoch = 0

        for i in range(X_test_scaled.shape[0]):
            x_sample = X_test_scaled[i].reshape(-1, 1)
            y_sample = y_test_encoded[i].reshape(-1, 1)

            pred = model.forward(x_sample)
            if np.argmax(pred[:2]) == np.argmax(y_sample):
                correct_test += 1

            test_loss_epoch += model.loss_fn(input=x_sample, target=y_sample)

        avg_test_loss = test_loss_epoch / X_test_scaled.shape[0]
        test_accuracy = correct_test / X_test_scaled.shape[0]

        print(f"Epoch {epoch:3d} | "
              f"Train Loss: {avg_train_loss:.4f} | Train Acc: {train_accuracy * 100:.2f}% | "
              f"Test Loss: {avg_test_loss:.4f} | Test Acc: {test_accuracy * 100:.2f}%")
```

```
Epoch   0 | Train Loss: 0.5016 | Train Acc: 77.44% | Test Loss: 0.4701 | Test Acc: 78.16%
Epoch  10 | Train Loss: 0.3573 | Train Acc: 81.60% | Test Loss: 0.4194 | Test Acc: 78.16%
Epoch  20 | Train Loss: 0.3402 | Train Acc: 83.51% | Test Loss: 0.4025 | Test Acc: 80.82%
Epoch  30 | Train Loss: 0.3296 | Train Acc: 83.97% | Test Loss: 0.3938 | Test Acc: 82.24%
Epoch  40 | Train Loss: 0.3293 | Train Acc: 83.92% | Test Loss: 0.3862 | Test Acc: 81.94%
Epoch  50 | Train Loss: 0.3202 | Train Acc: 84.64% | Test Loss: 0.3898 | Test Acc: 81.63%
Epoch  60 | Train Loss: 0.3135 | Train Acc: 85.17% | Test Loss: 0.3814 | Test Acc: 82.35%
Epoch  70 | Train Loss: 0.3057 | Train Acc: 85.40% | Test Loss: 0.3785 | Test Acc: 82.35%
Epoch  80 | Train Loss: 0.2987 | Train Acc: 85.71% | Test Loss: 0.3795 | Test Acc: 82.55%
Epoch  90 | Train Loss: 0.2936 | Train Acc: 86.29% | Test Loss: 0.3792 | Test Acc: 82.55%
Epoch 100 | Train Loss: 0.2876 | Train Acc: 86.65% | Test Loss: 0.3653 | Test Acc: 83.37%
```

In [309…

```
correct = 0
y_preds = []
y_true = []

for i in range(X_test_scaled.shape[0]):
    x_sample = X_test_scaled[i].reshape(-1, 1)
    y_sample = y_test_encoded[i].reshape(-1, 1)

    pred = model.forward(x_sample)
    pred_class = np.argmax(pred[0:2])
    true_class = np.argmax(y_sample)

    y_preds.append(pred_class)
    y_true.append(true_class)

    if pred_class == true_class:
        correct += 1

x_sample = X_test_scaled[0].reshape(-1, 1)
pred = model.forward(x_sample)


test_accuracy = correct / X_test_scaled.shape[0]
precision = precision_score(y_true, y_preds, average='macro', zero_division=0)
recall = recall_score(y_true, y_preds, average='macro', zero_division=0)
f1 = f1_score(y_true, y_preds, average='macro', zero_division=0)

print(f"Test Accuracy: {test_accuracy * 100:.2f}%")
print(f"Precision:     {precision * 100:.2f}%")
print(f"Recall:        {recall * 100:.2f}%")
```

```
Test Accuracy: 83.37%
Precision:     75.82%
Recall:        72.14%
```

Invertibility Check

Small reconstruction errors close to zero are expected due to floating-point precision limits

In [310…

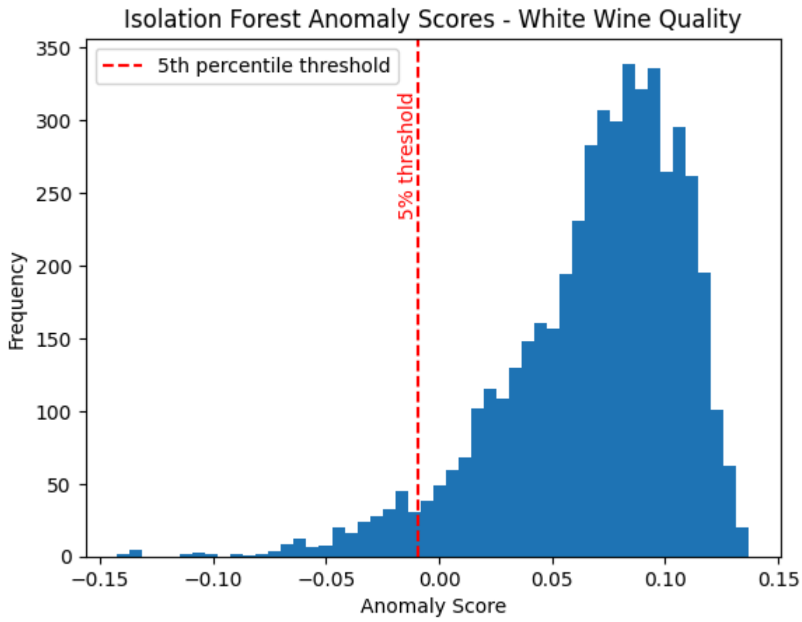
```
x_sample = X_test_scaled[0].reshape(-1, 1)

# Unscale
x_sample_unscaled = ANN_scaler.inverse_transform(x_sample.reshape(1, -1))
print("Original Sample:", x_sample_unscaled)

# Forward pass
pred = model.forward(x_sample)
print("\nOutput (Classes):", pred[0:2].ravel())
print("Output (Latent Variable):", pred[2:].ravel())
# Reconstruct
reconstructed_sample = model.reverse(pred)

# Unscale reconstruction
reconstructed_sample_unscaled = ANN_scaler.inverse_transform(
    reconstructed_sample.reshape(1, -1)
)
print("\nReconstructed Sample:", reconstructed_sample_unscaled.ravel())

mse_scaled = np.mean((x_sample - reconstructed_sample)**2)
mse_unscaled = np.mean((x_sample_unscaled - reconstructed_sample_unscaled)**2)
print("\nMSE Error (Scaled Data): ", mse_scaled)
print("MSE Error (Unscaled Data): ", mse_unscaled)
```

```
Original Sample: [[6.0000e+00 1.7000e-01 3.6000e-01 1.7000e+00 4.2000e-02 1.4000e+01
  6.1000e+01 9.9144e-01 3.2200e+00 5.4000e-01 1.0800e+01]]

Output (Classes): [0.98050259 0.01949741]
Output (Latent Variable): [ 3.92619796e+00 -4.75037097e-02 -3.29723407e-02 -1.94034664e-02
 -3.49284453e-02 -1.56047654e-03 -4.75244660e-02  9.84343548e-01
 -6.25475829e-02]

Reconstructed Sample: [6.00000000e+00 1.70000000e-01 3.60000000e-01 1.70000002e+00
 4.20000001e-02 1.39999999e+01 6.10000002e+01 9.91440000e-01
 3.22000000e+00 5.39999999e-01 1.08000000e+01]

MSE Error (Scaled Data):  2.2025489842542087e-17
MSE Error (Unscaled Data):  3.1172022567704e-15
```

Generation

In [311…

```
detector = IsolationForest(random_state=42).fit(X)


scores = detector.decision_function(X)

# Plot histogram
plt.hist(scores, bins=50)
threshold = np.percentile(scores, 5)
plt.axvline(threshold, color='red', linestyle='--', label='5th percentile threshold')

# Add label
plt.text(threshold, plt.ylim()[1]*0.9, '5% threshold', color='red', rotation=90, va='top', ha='right')

plt.title("Isolation Forest Anomaly Scores - White Wine Quality")
plt.xlabel("Anomaly Score")
plt.ylabel("Frequency")
plt.legend()
plt.show()
```



In [312…

```
def generate_data_with_epsilon(model,epsilon):
    pred_classes = []
    inputs = []
    original_inputs = []
    original_classes = []
    original_prob = []
    pred_probability = []

    for test,truth in zip(X_test_scaled,y_test_encoded):
        x_sample = test.reshape(-1, 1)
        pred = model.forward(x_sample)

        class_probs = pred[0:2].ravel()
        max_prob = np.max(class_probs)
        if np.argmax(pred[:2]) == np.argmax(truth):
            if max_prob >= 0.8:
                original_prob.append(max_prob)
                pred_class = np.argmax(class_probs)
                original_inputs.append(ANN_scaler.inverse_transform(x_sample.reshape(1, -1))[0])
                original_classes.append(pred_class)
                vector = pred.copy()
                vector[2:] = vector[2:] + epsilon

                new_input = model.reverse(input=vector)
                new_pred = model.forward(new_input)
                max_prob = np.max(new_pred[0:2])
                pred_probability.append(max_prob)
                new_pred_class = np.argmax(new_pred[0:2])

                new_input_unscaled = ANN_scaler.inverse_transform(new_input.reshape(1, -1))[0]

                pred_classes.append(new_pred_class)
                inputs.append(new_input_unscaled)

    # Build DataFrame
    data = {col: [] for col in column_names}
    data["Class"] = []
    data["Probability"] = []

    for sample in inputs:
        for i, col in enumerate(column_names):
            data[col].append(sample[i])

    for cls in pred_classes:
        data["Class"].append(cls)

    for prob in pred_probability:
        data["Probability"].append(prob * 100)

    generated_df = pd.DataFrame(data=data)
    return generated_df
```

Anomaly Score Threshold

The following code decides the anomaly score threshold used to guide generation. The first threshold represents moderately anomalous data whereas the second threshold represents extremely anomalous data.

In [313…

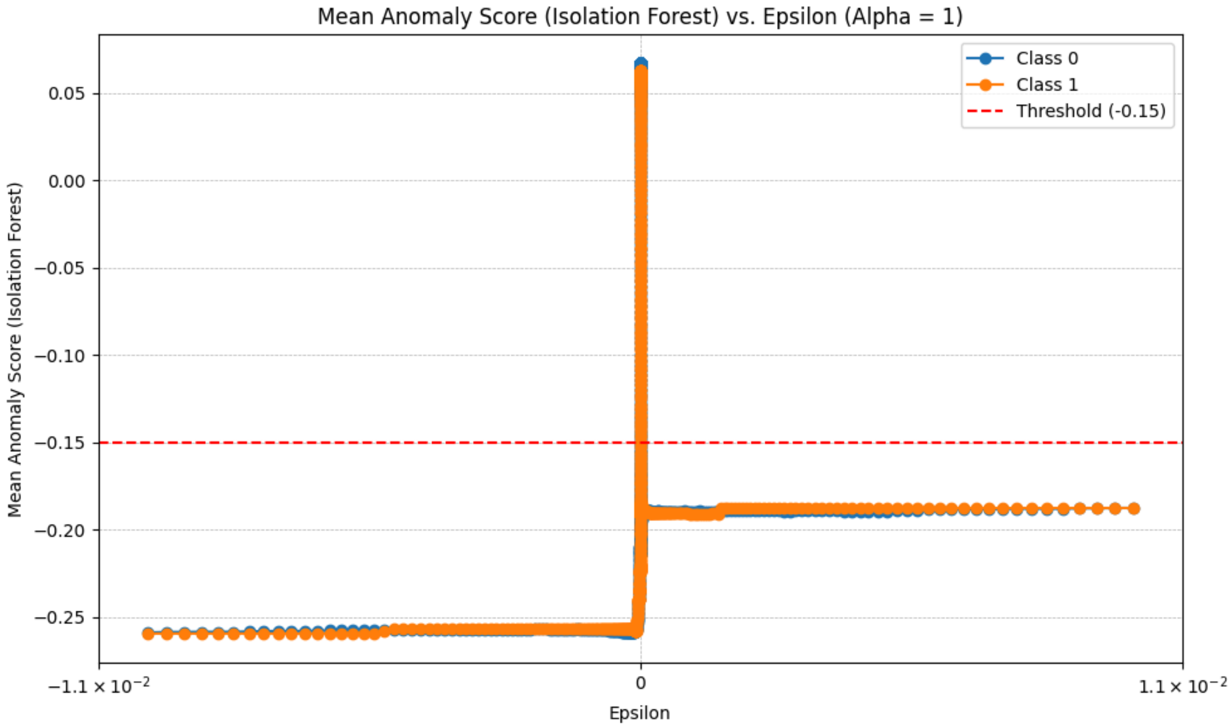
```
threshold_1 = np.percentile(scores,5)
threshold_2 = -0.15
```

In [314…

```
exponents = np.linspace(-12, -2 , 600)
positive_epsilons = 10 ** exponents
negative_epsilons = -positive_epsilons
epsilons = np.sort(np.concatenate([negative_epsilons, positive_epsilons]))

threshold = threshold_2 # Change threshold as needed
results = []

for epsilon in epsilons:
    generated_df = generate_data_with_epsilon(model, epsilon)
    anomaly_score_results = []

    for cls in generated_df["Class"].unique():
        subset = generated_df[generated_df["Class"] == cls]
        features = subset.drop(columns=["Class", "Probability"]).to_numpy()
        score = detector.decision_function(features)

        anomaly_score_results.append({
            "Class": cls,
            "Score": score.mean()
        })

    scored_df = pd.DataFrame(anomaly_score_results)
    mean_score_by_class = scored_df.set_index("Class")["Score"]
    below_threshold = mean_score_by_class[mean_score_by_class < threshold].to_dict()

    results.append({
        "epsilon": epsilon,
        "mean_score": mean_score_by_class.to_dict(),
    })
```

In [315…

```
class_names = []
for r in results:
    for cls in r["mean_score"]:
        if cls not in class_names:
            class_names.append(cls)
class_names.sort()

# Plot
plt.figure(figsize=(10, 6))
for cls in class_names:
    eps = []
    overlaps = []
    for r in results:
        eps.append(r["epsilon"])
        overlaps.append(r["mean_score"].get(cls))
    plt.plot(eps, overlaps, marker='o', label=f'Class {cls}')


# Threshold line
plt.axhline(y=threshold, color='red', linestyle='--', label=f'Threshold ({threshold})')

# Log scale on x-axis

plt.xlabel("Epsilon")
plt.ylabel("Mean Anomaly Score (Isolation Forest)")

plt.title("Mean Anomaly Score (Isolation Forest) vs. Epsilon (Alpha = 1)")
plt.legend()
plt.xscale("symlog")
plt.grid(True, which="both", ls="--", linewidth=0.5)
plt.tight_layout()
plt.show()
```



In [316…

```
best_epsilons = {}
best_eps_overall = None
max_abs_eps = 0  # Track largest absolute epsilon

for cls in class_names:
    best_pos_eps = None
    best_neg_eps = None
    best_pos_diff = float('inf')
    best_neg_diff = float('inf')

    for r in results:
        score = r["mean_score"].get(cls)
        eps = r["epsilon"]

        if score is not None and score < threshold:
            diff = threshold - score

            if eps > 0 and diff < best_pos_diff:
                best_pos_diff = diff
                best_pos_eps = eps

            elif eps < 0 and diff < best_neg_diff:
                best_neg_diff = diff
                best_neg_eps = eps

    best_epsilons[cls] = {
        "best_positive": best_pos_eps,
        "best_negative": best_neg_eps
    }

    for eps in [best_pos_eps, best_neg_eps]:
        if eps is not None and abs(eps) > max_abs_eps:
            max_abs_eps = abs(eps)
            best_eps_overall = eps

for cls, eps_dict in best_epsilons.items():
    print(f"Class {cls}:")
    print(f"  Best positive epsilon: {eps_dict['best_positive']}")
    print(f"  Best negative epsilon: {eps_dict['best_negative']}")

print(f"\nBest overall epsilon across all classes: {best_eps_overall}")
```

```
Class 0:
  Best positive epsilon: 2.467860087458031e-07
  Best negative epsilon: -2.285238607695462e-07
Class 1:
  Best positive epsilon: 2.769516817469001e-07
  Best negative epsilon: -1.885641372950549e-07

Best overall epsilon across all classes: 2.769516817469001e-07
```

Generated Confidently Classified Anomalies

In [317…

```
epsilon = max_abs_eps


pred_classes = []
inputs = []
original_inputs = []
original_classes = []
original_prob = []
pred_probability = []

for test,truth in zip(X_test_scaled,y_test_encoded):
    x_sample = test.reshape(-1, 1)
    pred = model.forward(x_sample)

    class_probs = pred[0:2].ravel()
    max_prob = np.max(class_probs)
    if np.argmax(pred[:2]) == np.argmax(truth):
        if max_prob >= 0.8:
            original_prob.append(max_prob)
            pred_class = np.argmax(class_probs)
            original_inputs.append(ANN_scaler.inverse_transform(x_sample.reshape(1, -1))[0])
            original_classes.append(pred_class)

            vector_neg_epsilon = pred.copy()
            vector_neg_epsilon[2:] = vector_neg_epsilon[2:] - epsilon

            # Revese pass for when epsilon is substracted to latent variables
            new_input_neg = model.reverse(input=vector_neg_epsilon)
            new_pred = model.forward(new_input_neg)
            max_prob = np.max(new_pred[0:2])
            pred_probability.append(max_prob)
            new_pred_class_neg = np.argmax(new_pred[0:2])

            vector_pos_epsilon = pred.copy()
            vector_pos_epsilon[2:] = vector_pos_epsilon[2:] + epsilon

            # Revese pass for when epsilon is added to latent variables
            new_input_pos = model.reverse(input=vector_pos_epsilon)
            new_pred = model.forward(new_input_pos)
            max_prob_pos = np.max(new_pred[0:2])
            pred_probability.append(max_prob_pos)
            new_pred_class_pos = np.argmax(new_pred[0:2])

            new_input_unscaled_pos = ANN_scaler.inverse_transform(new_input_pos.reshape(1, -1))[0]
            new_input_unscaled_neg = ANN_scaler.inverse_transform(new_input_neg.reshape(1, -1))[0]

            pred_classes.append(new_pred_class_pos)
            inputs.append(new_input_unscaled_pos)

            pred_classes.append(new_pred_class_neg)
            inputs.append(new_input_unscaled_neg)



data = {col: [] for col in column_names}
data["Class"] = []
data["Probability"] = []

for sample in inputs:
    for i, col in enumerate(column_names):
        data[col].append(sample[i])

for cls in pred_classes:
    data["Class"].append(cls)

for prob in pred_probability:
    data["Probability"].append(prob * 100)

generated_df = pd.DataFrame(data=data)
```

In [318…

```
generated_df = generated_df.drop_duplicates()
```

In [319…

```
prediction_list = detector.predict(inputs).tolist()


anomalies = prediction_list.count(-1) / len(inputs)
print(f"Anomaly rate: {anomalies:.2%}")


inputs = np.array(inputs)
predictions = np.array(prediction_list)

anomalous_inputs = inputs[predictions == -1]
```

```
generated_anomalies = pd.DataFrame(anomalous_inputs, columns=column_names)
```

```
Anomaly rate: 96.82%
```

In [320…

```
generated_anomalies=generated_anomalies.drop_duplicates().reset_index(drop=True)
```

In [321…

```
prediction_list = detector.predict(X).tolist()

anomalies = prediction_list.count(-1)/len(X)

print(anomalies)
```

```
0.06288280930992242
```

In [322…

```
generated_anomalies
```

Out[322…

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol
0	6.638056	-0.254595	2.005973	61.009207	0.313283	-149.417803	586.931828	1.037689	3.159388	-1.704244	41.136716
1	6.138499	0.818355	1.167119	18.833559	0.155920	49.093370	281.641924	0.989733	2.872475	0.744614	12.146138
2	10.558523	0.332413	0.510640	-25.920354	0.146480	-37.915215	-8.224125	0.996807	3.258488	-0.076455	16.101660
3	15.325635	-0.137105	0.625696	11.516162	0.052455	-101.817652	80.939176	0.977256	3.445755	1.205317	5.829430
4	2.527764	-1.581751	-1.426902	0.437439	0.263298	-258.334009	441.853298	1.035607	4.941588	-4.760838	44.067869
...	...	...	...	...	...	...	...	...	...	...	...
1058	16.851784	2.002280	4.835400	119.823058	0.365007	289.560921	1098.417528	1.009379	0.857623	2.422075	12.885632
1059	9.067925	0.198275	2.041501	130.569115	0.306971	-39.819218	936.524656	1.063838	2.648050	-1.731709	38.123577
1060	5.242954	-1.683543	-3.220992	-88.917411	-0.154664	-434.786442	-548.940859	0.943230	6.022245	0.054602	4.820626
1061	3.677648	-0.695023	-0.684513	-11.623281	-0.046505	-113.578102	99.666006	1.020792	4.443418	-1.535666	25.587627
1062	5.469491	-0.928761	-1.710991	-52.786363	-0.107396	-245.385770	-253.856741	0.962619	4.831910	0.354565	7.606839

1063 rows × 11 columns

Transferability Of Confidently Clasified Anomalies

Testing Settings

In [323…

```
confidence_threshold = 0.8
```

Randomn Forest

In [324…

```
RF_model = RandomForestClassifier(n_estimators=200,random_state=42)
RF_model.fit(X_train,y_train)


y_pred = RF_model.predict(X_test)



test_acc = accuracy_score(y_test, y_pred)
test_precision = precision_score(y_test, y_pred,average="macro")
test_recall = recall_score(y_test, y_pred,average="macro")
print("Test Accuracy:", test_acc)
print("Test Precision:", test_precision)
print("Test Recall:", test_recall)
```

```
Test Accuracy: 0.8908163265306123
Test Precision: 0.8663732677590605
Test Recall: 0.7937426297169812
```

In [325…

```
probs_all = RF_model.predict_proba(generated_anomalies.values)

max_probs = np.max(probs_all, axis=1)
pred_classes = np.argmax(probs_all, axis=1)


mask = max_probs >= confidence_threshold
RF_anomalies_list = generated_anomalies.values[mask]
max_prob_rf = pred_classes[mask]
high_confidence_count = np.sum(mask)
```

In [326…
```
robustness = high_confidence_count/len(generated_anomalies.values)
print(robustness)
```

```
0.09407337723424271
```

In [327…
```
RF_df = pd.DataFrame(RF_anomalies_list, columns = column_names)
```

In [328…
```
len(RF_anomalies_list)
```

Out[328…
```
100
```

Testing On Neural Networks

In [329…
```
np.random.seed(42)
torch.manual_seed(42)
torch.cuda.manual_seed_all(42)
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False

# Model architecture
class SimpleNet(nn.Module):
    def __init__(self, input_dim):
        super(SimpleNet, self).__init__()
        self.net = nn.Sequential(
            nn.Linear(input_dim, 64),
            nn.ReLU(),
            nn.Linear(64, 32),
            nn.ReLU(),
            nn.Linear(32, 16),
            nn.ReLU(),
            nn.Linear(16, 8),
            nn.ReLU(),
            nn.Linear(8, 2)
        )

    def forward(self, x):
        return self.net(x)


X_train_tensor = torch.tensor(X_train_scaled, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train, dtype=torch.long)
X_test_tensor = torch.tensor(X_test_scaled, dtype=torch.float32)
y_test_tensor = torch.tensor(y_test, dtype=torch.long)

# Initialize model, loss, optimizer
test_model = SimpleNet(input_dim=X_train_tensor.shape[1])
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(test_model.parameters(), lr=0.001)

# Training loop
num_epochs = 440
for epoch in range(num_epochs):
    test_model.train()
    optimizer.zero_grad()
    outputs = test_model(X_train_tensor)
    loss = criterion(outputs, y_train_tensor)
    loss.backward()
    optimizer.step()

    if epoch % 10 == 0 or epoch == num_epochs - 1:
        test_model.eval()
        with torch.no_grad():
            test_outputs = test_model(X_test_tensor)
            preds = torch.argmax(test_outputs, dim=1)
            acc = accuracy_score(y_test_tensor, preds)
            print(f"Epoch {epoch:3d} | Loss: {loss.item():.4f} | Test Accuracy: {acc * 100:.2f}%")
```

```
Epoch   0 | Loss: 0.6743 | Test Accuracy: 78.37%
Epoch  10 | Loss: 0.6495 | Test Accuracy: 78.37%
Epoch  20 | Loss: 0.6229 | Test Accuracy: 78.37%
Epoch  30 | Loss: 0.5871 | Test Accuracy: 78.37%
Epoch  40 | Loss: 0.5369 | Test Accuracy: 78.37%
Epoch  50 | Loss: 0.4868 | Test Accuracy: 78.37%
Epoch  60 | Loss: 0.4642 | Test Accuracy: 78.37%
Epoch  70 | Loss: 0.4469 | Test Accuracy: 78.37%
Epoch  80 | Loss: 0.4328 | Test Accuracy: 78.37%
Epoch  90 | Loss: 0.4198 | Test Accuracy: 78.37%
Epoch 100 | Loss: 0.4093 | Test Accuracy: 78.37%
Epoch 110 | Loss: 0.4013 | Test Accuracy: 78.98%
Epoch 120 | Loss: 0.3944 | Test Accuracy: 81.12%
Epoch 130 | Loss: 0.3881 | Test Accuracy: 82.35%
Epoch 140 | Loss: 0.3824 | Test Accuracy: 81.12%
Epoch 150 | Loss: 0.3769 | Test Accuracy: 80.82%
Epoch 160 | Loss: 0.3718 | Test Accuracy: 80.82%
Epoch 170 | Loss: 0.3668 | Test Accuracy: 80.31%
Epoch 180 | Loss: 0.3615 | Test Accuracy: 80.82%
Epoch 190 | Loss: 0.3560 | Test Accuracy: 81.12%
Epoch 200 | Loss: 0.3502 | Test Accuracy: 80.61%
Epoch 210 | Loss: 0.3440 | Test Accuracy: 80.51%
Epoch 220 | Loss: 0.3369 | Test Accuracy: 80.82%
Epoch 230 | Loss: 0.3293 | Test Accuracy: 81.33%
Epoch 240 | Loss: 0.3213 | Test Accuracy: 82.04%
Epoch 250 | Loss: 0.3126 | Test Accuracy: 82.04%
Epoch 260 | Loss: 0.3029 | Test Accuracy: 82.65%
Epoch 270 | Loss: 0.2918 | Test Accuracy: 82.86%
Epoch 280 | Loss: 0.2804 | Test Accuracy: 83.37%
Epoch 290 | Loss: 0.2694 | Test Accuracy: 82.76%
Epoch 300 | Loss: 0.2586 | Test Accuracy: 83.16%
Epoch 310 | Loss: 0.2473 | Test Accuracy: 83.27%
Epoch 320 | Loss: 0.2370 | Test Accuracy: 83.98%
Epoch 330 | Loss: 0.2280 | Test Accuracy: 83.67%
Epoch 340 | Loss: 0.2193 | Test Accuracy: 83.67%
Epoch 350 | Loss: 0.2121 | Test Accuracy: 84.18%
Epoch 360 | Loss: 0.2049 | Test Accuracy: 84.49%
Epoch 370 | Loss: 0.1982 | Test Accuracy: 84.08%
Epoch 380 | Loss: 0.1916 | Test Accuracy: 84.39%
Epoch 390 | Loss: 0.1856 | Test Accuracy: 84.08%
Epoch 400 | Loss: 0.1800 | Test Accuracy: 84.80%
Epoch 410 | Loss: 0.1743 | Test Accuracy: 84.90%
Epoch 420 | Loss: 0.1691 | Test Accuracy: 84.29%
Epoch 430 | Loss: 0.1651 | Test Accuracy: 85.20%
Epoch 439 | Loss: 0.1630 | Test Accuracy: 85.41%
```

In [330…

```python
test_model.eval()
correct = 0
y_preds = []
y_true = []

for i in range(X_test_scaled.shape[0]):
    x_sample = torch.tensor(X_test_scaled[i].reshape(1, -1), dtype=torch.float32)
    y_sample = y_test_encoded[i].reshape(-1)  # Assuming one-hot encoded

    with torch.no_grad():
        logits = test_model(x_sample)
        probs = torch.softmax(logits, dim=1).numpy().flatten()
        pred_class = np.argmax(probs)
        true_class = np.argmax(y_sample)

    y_preds.append(pred_class)
    y_true.append(true_class)

    if pred_class == true_class:
        correct += 1


x_sample = torch.tensor(X_test_scaled[0].reshape(1, -1), dtype=torch.float32)
with torch.no_grad():
    pred = test_model(x_sample)


test_accuracy = correct / len(X_test_scaled)
precision = precision_score(y_true, y_preds, average='macro', zero_division=0)
recall = recall_score(y_true, y_preds, average='macro', zero_division=0)
f1 = f1_score(y_true, y_preds, average='macro', zero_division=0)

print(f"Test Accuracy: {test_accuracy * 100:.2f}%")
print(f"Precision:     {precision * 100:.2f}%")
print(f"Recall:        {recall * 100:.2f}%")
print(f"F1 Score:      {f1 * 100:.2f}%")
```

```
Test Accuracy: 85.41%
Precision:     78.71%
Recall:        77.20%
F1 Score:      77.91%
```

In [331…

```
MLP_anomalies_list = []
high_confidence_count = 0
test = []
scaled_anomalous_inputs = ANN_scaler.transform(generated_anomalies.values)


test_model.eval()

for x in scaled_anomalous_inputs:
    x_tensor = torch.tensor(x.reshape(1, -1), dtype=torch.float32)  # shape: [1, input_dim]
    with torch.no_grad():
        logits = test_model(x_tensor)  # shape: [1, 2]
        probs = torch.softmax(logits, dim=1).numpy().flatten()  # convert to numpy array
        max_prob = np.max(probs)

    if max_prob >= confidence_threshold:
        high_confidence_count += 1
        MLP_anomalies_list.append(x)
        if np.argmax(probs) ==1:
            test.append(x)


MLP_anomalies_list = ANN_scaler.inverse_transform(MLP_anomalies_list)


robustness = high_confidence_count / len(generated_anomalies.values)
print(f"Robustness: {robustness}")
```

```
Robustness: 0.9877704609595485
```

In [332…

```
len(MLP_anomalies_list)
```

Out[332… 1050

In [333…

```
MLP_df = pd.DataFrame(MLP_anomalies_list, columns = column_names)
```

KNN Classifier

In [334…

```
from sklearn.neighbors import KNeighborsClassifier

n = 5
neigh = KNeighborsClassifier(n_neighbors=n)
neigh.fit(X_train_scaled, y_train)

y_pred = neigh.predict(X_test_scaled)

test_acc = accuracy_score(y_test, y_pred)
test_precision = precision_score(y_test, y_pred,average="macro")
test_recall = recall_score(y_test, y_pred,average="macro")
print("Test Accuracy:", test_acc)
print("Test Precision:", test_precision)
print("Test Recall:", test_recall)
```

```
Test Accuracy: 0.8418367346938775
Test Precision: 0.7709322843652139
Test Recall: 0.7385883451257862
```

In [335…

```
KNN_anomalies_list = []
high_confidence_count = 0

scaled_anomalous_inputs = ANN_scaler.transform(generated_anomalies.values)

for x in scaled_anomalous_inputs:
    probs = neigh.predict_proba(x.reshape(1, -1))
    max_prob = np.max(probs)

    if max_prob >= confidence_threshold:
        high_confidence_count += 1
        KNN_anomalies_list.append(x)

robustness = high_confidence_count / len(scaled_anomalous_inputs)
KNN_anomalies_list = ANN_scaler.inverse_transform(KNN_anomalies_list)
print(robustness)
```

```
0.8447789275634995
```

In [336…

```
len(KNN_anomalies_list)
```

Out[336… 898

In [337…
```
knn_df = pd.DataFrame(KNN_anomalies_list, columns = column_names)
```

Checking Shared Vulnerabilities

In [338…
```
feature_names = column_names.tolist()
```

In [339…
```
# Drop duplicates in each DataFrame
df_rf = RF_df.drop_duplicates()
df_knn = knn_df.drop_duplicates()
df_nn = MLP_df.drop_duplicates()

# Merge on all columns to find common rows
common_rows = df_rf.merge(df_knn, how='inner').merge(df_nn, how='inner')

print("Number of common rows:", len(common_rows))
```

```
Number of common rows: 89
```

In [340…
```
# Convert to NumPy array if needed
inputs = common_rows.values

# Predict with both models
rf_preds = RF_model.predict(inputs)
knn_preds = neigh.predict(inputs)

# Find indices where class == 1
rf_class1_indices = np.where(rf_preds == 1)[0]
knn_class1_indices = np.where(knn_preds == 1)[0]

# Extract corresponding samples
rf_class1_samples = inputs[rf_class1_indices]
knn_class1_samples = inputs[knn_class1_indices]
```

In [341…
```
common_rows
```

Out[341…

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol
0	6.619644	0.725416	1.169992	17.344795	0.133857	79.184379	271.801423	0.995876	2.889040	0.734420	9.590628
1	7.131828	0.488843	0.638656	17.297463	0.075126	82.282676	161.853428	0.994377	2.921193	0.709142	9.564306
2	6.438273	0.843791	0.941368	30.852560	0.143286	70.454402	320.134621	0.995497	2.918892	0.616214	10.750064
3	7.770855	0.478410	0.632777	23.874041	0.067491	89.246862	190.458986	0.996373	2.904713	0.743790	8.957409
4	6.955438	0.386481	1.001164	12.443815	0.071601	66.952040	247.900174	0.995215	3.042596	0.581948	9.182088
...	...	...	...	...	...	...	...	...	...	...	...
84	6.112677	0.334243	0.286982	8.328684	0.055494	88.856758	50.597346	0.996917	2.948322	0.611817	7.639219
85	10.797688	0.983184	2.376787	57.817333	0.155730	147.198132	481.909374	1.004500	1.972519	1.271915	9.915220
86	8.328823	0.595884	0.839332	33.656592	0.095419	95.008638	263.783051	0.998548	2.790368	0.723396	9.256311
87	8.349452	0.439609	0.551893	18.302610	0.065321	52.645978	118.127321	0.993491	3.058263	0.628026	9.043823
88	7.816388	0.422726	0.554827	16.202754	0.058204	54.011559	109.120821	0.994899	3.075205	0.765567	7.694477

89 rows × 11 columns

In [342…
```
# Filter rows where any feature column has a negative value
negative_rows = common_rows[feature_names][(common_rows[feature_names] < 0).any(axis=1)]
```

In [343…
```
negative_rows
```

Out[343…

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol
19	5.982089	0.199720	0.008589	-3.455079	0.007816	37.162714	99.332774	0.994164	3.467651	0.260337	8.736993
47	4.839145	0.146253	-0.304224	4.579052	0.042293	66.979595	25.338325	0.996992	3.256260	0.522848	5.238934
58	9.446042	0.240549	0.122134	-6.646533	0.092361	6.782285	113.525672	0.998674	3.301367	0.043708	10.669548

In [344…
```
negative_rows.reset_index(drop=True)
```

Out[344...

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol
0	5.982089	0.199720	0.008589	-3.455079	0.007816	37.162714	99.332774	0.994164	3.467651	0.260337	8.736993
1	4.839145	0.146253	-0.304224	4.579052	0.042293	66.979595	25.338325	0.996992	3.256260	0.522848	5.238934
2	9.446042	0.240549	0.122134	-6.646533	0.092361	6.782285	113.525672	0.998674	3.301367	0.043708	10.669548

In [345...

```
common_rows[feature_names].iloc[14]
```

Out[345...

```
fixed acidity            7.400440
volatile acidity         0.575708
citric acid              1.144452
residual sugar          23.253191
chlorides                0.101103
free sulfur dioxide     92.268847
total sulfur dioxide   336.973133
density                  0.996635
pH                       2.789534
sulphates                0.857810
alcohol                 10.159199
Name: 14, dtype: float64
```

Example of Anomalous Sample

In [346...

```
sample_row = negative_rows.iloc[1]
print("Input row (original scale):")
print(sample_row)
```

```
Input row (original scale):
fixed acidity            4.839145
volatile acidity         0.146253
citric acid             -0.304224
residual sugar           4.579052
chlorides                0.042293
free sulfur dioxide     66.979595
total sulfur dioxide    25.338325
density                  0.996992
pH                       3.256260
sulphates                0.522848
alcohol                  5.238934
Name: 47, dtype: float64
```

In [347...

```
sample_row = common_rows.iloc[14]  # You can change the index if needed

scaled_sample = ANN_scaler.transform(sample_row.values.reshape(1, -1))

x_tensor = torch.tensor(scaled_sample, dtype=torch.float32)
test_model.eval()

with torch.no_grad():
    logits = test_model(x_tensor)
    probs = torch.softmax(logits, dim=1).numpy().flatten()
    max_prob = np.max(probs)
    predicted_class = np.argmax(probs)
print("Neural Network Model")
print("Probability of Low Quality:", probs[0]*100,"%")
print("Probability of Hih Quality Quality:", probs[1]*100,"%")
```

```
Neural Network Model
Probability of Low Quality: 100.0 %
Probability of Hih Quality Quality: 1.2265634e-20 %
```

In [348...

```
probs = neigh.predict_proba(sample_row.values.reshape(1,-1))
print("K-NN Model (K = 5)")
print("Probability of Low Quality:", probs[0][0]*100,"%")
print("Probability of Hih Quality Quality:", probs[0][1]*100,"%")
```

```
K-NN Model (K = 5)
Probability of Low Quality: 100.0 %
Probability of Hih Quality Quality: 0.0 %
```

In [349...

```
probs = RF_model.predict_proba(sample_row.values.reshape(1,-1))
print("Random Forest Model")
print("Probability of Low Quality:", probs[0][0]*100,"%")
print("Probability of Hih Quality Quality:", probs[0][1]*100,"%")
```

```
Random Forest Model
Probability of Low Quality: 80.5 %
Probability of Hih Quality Quality: 19.5 %

Probability of Low Quality: 80.5 %
Probability of Hih Quality Quality: 19.5 %
```