In [258…

```
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 [259…

```
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 [260…

```
# 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 [261…

```
num_layer = 3
num_epochs = 101
```

In [262…

```
# 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 [263…

```
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 [264…

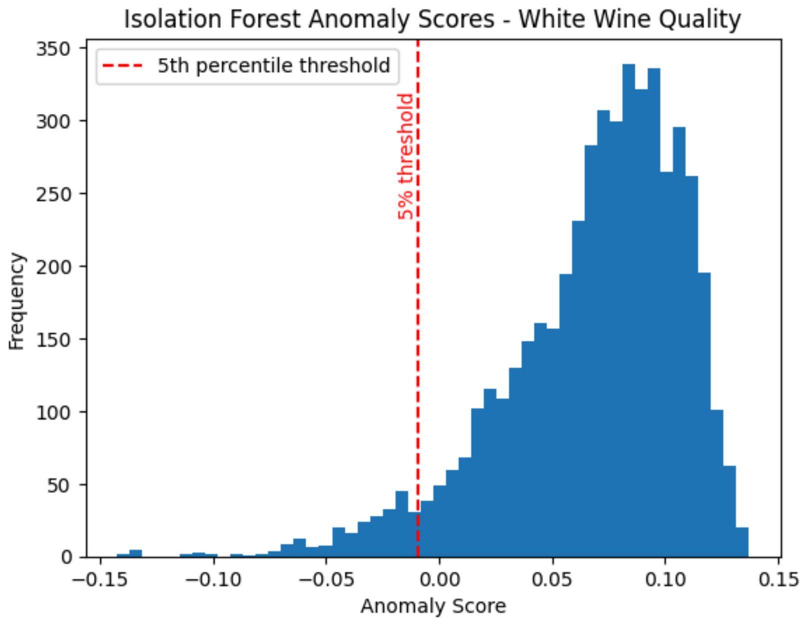
```
x_sample = X_test_scaled[0].reshape(-1, 1)

# 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 [265…

```
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 [266…

```
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
```

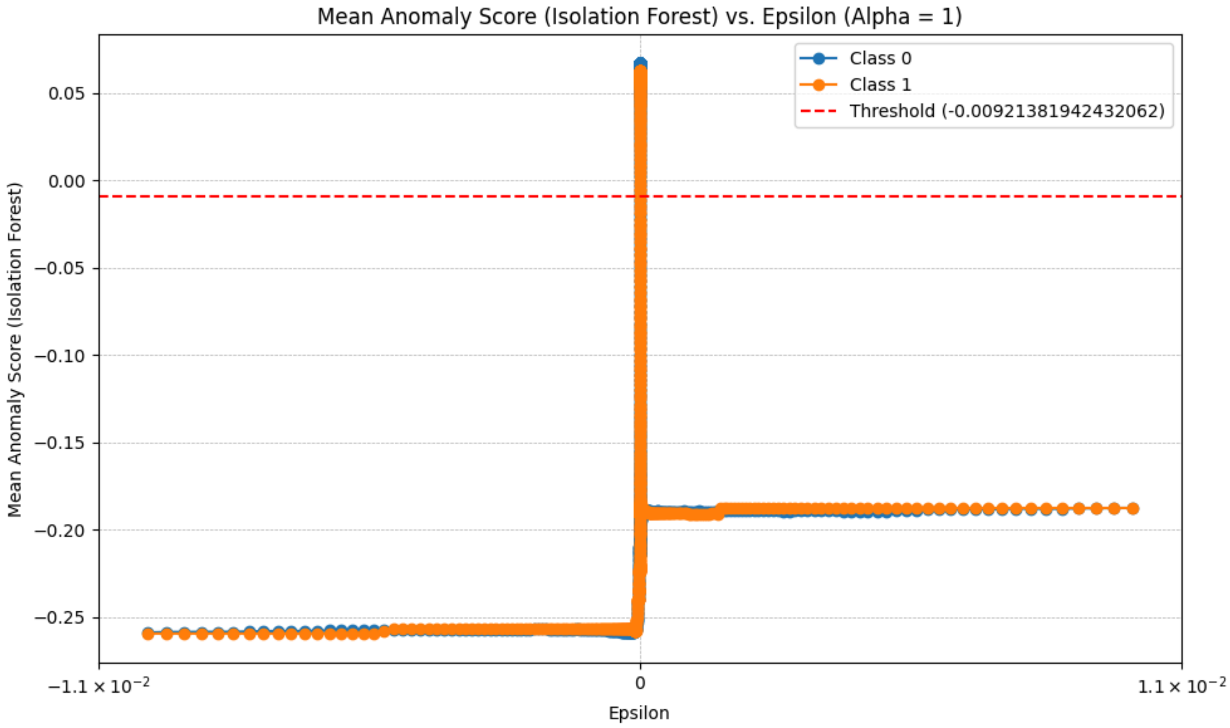
Anomaly Score Threshold

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 [267…
```
threshold_1 = np.percentile(scores,5)
threshold_2 = -0.15
```

In [268…
```
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_1 # 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 [269…
```
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 [270…

```
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: 3.7522389225313985e-08
  Best negative epsilon: -3.474573495735988e-08
Class 1:
  Best positive epsilon: 2.8670089481484573e-08
  Best negative epsilon: -2.554733341531624e-08

Best overall epsilon across all classes: 3.7522389225313985e-08
```

Generated Confidently Classified Anomalies

In [271…

```
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 [272…

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

In [273…

```
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: 53.18%
```

In [274…

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

In [275…

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

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

print(anomalies)
```

```
0.06288280930992242
```

In [276…

```
generated_anomalies
```

Out[276…

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol
0	6.086446	0.112474	0.583002	9.735420	0.078754	-8.140419	132.255094	0.997706	3.211788	0.235942	14.910125
1	9.597688	0.307134	0.515876	2.180455	0.049811	11.198634	15.950341	0.995507	2.859222	0.440415	10.810118
2	9.729108	0.101217	0.347579	3.575136	0.004413	-16.999349	2.872941	0.990959	2.941063	0.582996	9.033697
3	5.555354	0.016548	0.083258	1.278796	0.136015	-6.542715	145.541693	0.998059	3.432095	-0.316792	14.705821
4	6.106977	0.398598	0.658312	6.125418	0.092710	40.878054	192.868882	0.996068	3.264201	0.365540	9.252220
...	...	...	...	...	...	...	...	...	...	...	...
575	7.254596	0.160508	-0.071140	-6.603866	0.060905	118.752057	244.464510	0.993339	3.474930	0.406945	10.869855
576	7.745404	0.379492	0.691140	18.203867	0.053095	143.247943	381.535492	0.995861	2.885070	0.773055	10.130145
577	6.570582	0.345718	0.578047	35.604521	0.082153	33.619794	269.186573	1.008135	2.927249	0.262278	13.835876
578	5.932483	-0.012909	-0.368617	-3.227965	0.002782	-43.001946	0.231800	0.986019	3.603846	0.540398	7.963634
579	6.667309	0.126884	0.114089	-4.393753	0.012747	-15.603043	60.836853	0.986529	3.447220	0.747697	10.223431

580 rows × 11 columns

Transferability Of Confidently Clasified Anomalies

Testing Settings

In [277…

```
confidence_threshold = 0.8
```

Randomn Forest

In [278…

```
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 [279…

```
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 [280…
```
robustness = high_confidence_count/len(generated_anomalies.values)
print(robustness)
```

```
0.3120689655172414
```

In [281…
```
RF_df = pd.DataFrame(RF_anomalies_list, columns = column_names)
```

In [282…
```
len(RF_anomalies_list)
```

Out[282…
```
181
```

Testing On Neural Networks

In [283…
```
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  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 [284…

```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 [285…

```
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: 0.9620689655172414
```

In [286…

```
len(MLP_anomalies_list)
```

Out[286… 558

In [287…

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

KNN Classifier

In [288…

```
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 [289…

```
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)
```

```
0.8689655172413793
```

In [290…

```
len(KNN_anomalies_list)
```

Out[290… 504

In [291…
```
knn_df = pd.DataFrame(KNN_anomalies_list, columns = column_names)
```

Checking Shared Vulnerabilities

In [292…
```
feature_names = column_names.tolist()
```

In [ ]:
```
# 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: 175
```

In [294…
```
# 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 [295…
```
common_rows
```

Out[295…

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol
0	9.597688	0.307134	0.515876	2.180455	0.049811	11.198634	15.950341	0.995507	2.859222	0.440415	10.810118
1	6.106977	0.398598	0.658312	6.125418	0.092710	40.878054	192.868882	0.996068	3.264201	0.365540	9.252220
2	6.293023	0.501402	0.801688	8.274583	0.105290	53.121951	211.131122	0.995572	3.155799	0.494460	9.147780
3	6.539539	0.566692	0.706849	19.732222	0.060097	84.150819	196.144227	0.995985	2.988620	0.885661	8.632607
4	7.026955	0.476785	0.615207	17.000645	0.073933	82.178961	160.166482	0.994863	2.931235	0.671063	9.772395
...	...	...	...	...	...	...	...	...	...	...	...
170	6.969716	0.447190	0.659189	23.975092	0.068123	85.590345	246.353353	0.998559	2.874866	0.746867	8.715935
171	8.171122	0.415752	0.536599	15.638762	0.067464	47.425637	105.926753	0.993054	3.074616	0.582376	9.450628
172	7.634846	0.513103	0.251729	12.723552	0.077459	36.546933	155.910549	1.001677	3.232583	0.356781	10.144241
173	7.610545	0.249777	0.352794	13.843082	0.077915	10.505264	140.725924	1.000278	3.273470	0.134926	10.917273
174	7.745404	0.379492	0.691140	18.203867	0.053095	143.247943	381.535492	0.995861	2.885070	0.773055	10.130145

175 rows × 11 columns

In [296…
```
# 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 [297…
```
negative_rows
```

Out[297…

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol
11	5.535680	0.121270	0.018666	-3.184112	0.014720	46.376041	150.461355	0.993912	3.288320	0.295626	9.143875
14	6.290587	0.427309	-0.260097	-6.052626	0.021421	-10.483904	61.032187	0.986866	3.355864	0.445388	8.996359
26	6.206194	0.321373	0.044749	-2.184957	0.020074	5.662817	65.865415	0.989398	3.250668	0.476161	8.963407
30	8.118618	0.469542	0.691650	-0.108228	0.215253	25.046513	177.544802	0.996380	3.158287	0.385128	10.530506
46	9.593996	1.025138	0.384150	-1.597036	0.042891	44.602843	165.630049	0.998769	3.043500	0.408887	10.511838
49	7.069956	0.393858	0.167114	1.684714	0.062314	-20.075436	68.344077	0.985221	3.362987	0.466079	9.898336
58	6.244912	0.108512	-0.043215	1.632855	0.140079	27.763642	109.620568	0.992356	3.055638	0.514244	7.385408
74	6.109634	0.162299	0.042065	-0.850620	0.023369	-7.223909	55.094690	0.989732	3.390418	0.491652	8.932659
77	6.251114	0.214155	0.063757	-3.328095	0.020184	4.624540	99.552407	0.993348	3.213874	0.216625	9.268605
78	5.853652	0.028557	0.020012	-7.146336	0.027531	31.015026	107.122188	0.992802	3.219658	0.245165	9.294121
90	6.052540	0.498555	-0.060431	0.373474	0.037512	11.448171	154.141503	0.992103	3.325719	0.509858	8.227442
94	7.491870	0.373341	0.033311	-0.821683	0.017792	-15.884827	66.500531	0.988002	3.177260	0.333239	9.706036
100	6.853973	0.136171	0.008831	-0.048106	0.023302	-18.018820	56.979251	0.987176	3.337411	0.478306	9.724123
101	5.768085	0.030900	0.395587	-0.658790	0.015249	8.006535	110.431788	0.995521	3.346019	0.140844	9.353341
104	6.768259	0.216597	-0.048608	-0.111871	0.021329	7.097973	96.136291	0.989506	3.369727	0.454715	8.828130
107	7.430412	0.093461	-0.012158	-3.292199	0.028621	-4.684588	27.376304	0.985707	3.170362	0.534332	9.528912
117	6.205999	0.203580	0.152772	-0.179144	0.030020	-7.401202	78.042455	0.989682	3.175592	0.631116	8.508034
118	6.140264	0.044401	-0.111963	-8.194881	0.006293	26.813371	81.549597	0.992393	3.304863	0.096834	10.005137
134	5.985239	0.334834	0.297655	-0.254301	0.107361	5.326056	36.137575	0.989774	3.255648	0.520277	8.079908
144	6.218968	0.309468	0.029699	-0.162545	0.118048	54.520779	181.148742	0.996321	3.302286	0.254045	10.811922
159	6.696087	0.407913	-0.048879	-2.098421	0.056640	17.236542	67.682094	0.996294	3.469830	0.196641	10.246661
160	6.855161	0.230980	0.092447	-0.532175	0.186738	21.496664	42.991580	0.992230	3.306240	0.539875	7.094077

In [298…

```
negative_rows.reset_index(drop=True)
```

Out[298…

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol
0	5.535680	0.121270	0.018666	-3.184112	0.014720	46.376041	150.461355	0.993912	3.288320	0.295626	9.143875
1	6.290587	0.427309	-0.260097	-6.052626	0.021421	-10.483904	61.032187	0.986866	3.355864	0.445388	8.996359
2	6.206194	0.321373	0.044749	-2.184957	0.020074	5.662817	65.865415	0.989398	3.250668	0.476161	8.963407
3	8.118618	0.469542	0.691650	-0.108228	0.215253	25.046513	177.544802	0.996380	3.158287	0.385128	10.530506
4	9.593996	1.025138	0.384150	-1.597036	0.042891	44.602843	165.630049	0.998769	3.043500	0.408887	10.511838
5	7.069956	0.393858	0.167114	1.684714	0.062314	-20.075436	68.344077	0.985221	3.362987	0.466079	9.898336
6	6.244912	0.108512	-0.043215	1.632855	0.140079	27.763642	109.620568	0.992356	3.055638	0.514244	7.385408
7	6.109634	0.162299	0.042065	-0.850620	0.023369	-7.223909	55.094690	0.989732	3.390418	0.491652	8.932659
8	6.251114	0.214155	0.063757	-3.328095	0.020184	4.624540	99.552407	0.993348	3.213874	0.216625	9.268605
9	5.853652	0.028557	0.020012	-7.146336	0.027531	31.015026	107.122188	0.992802	3.219658	0.245165	9.294121
10	6.052540	0.498555	-0.060431	0.373474	0.037512	11.448171	154.141503	0.992103	3.325719	0.509858	8.227442
11	7.491870	0.373341	0.033311	-0.821683	0.017792	-15.884827	66.500531	0.988002	3.177260	0.333239	9.706036
12	6.853973	0.136171	0.008831	-0.048106	0.023302	-18.018820	56.979251	0.987176	3.337411	0.478306	9.724123
13	5.768085	0.030900	0.395587	-0.658790	0.015249	8.006535	110.431788	0.995521	3.346019	0.140844	9.353341
14	6.768259	0.216597	-0.048608	-0.111871	0.021329	7.097973	96.136291	0.989506	3.369727	0.454715	8.828130
15	7.430412	0.093461	-0.012158	-3.292199	0.028621	-4.684588	27.376304	0.985707	3.170362	0.534332	9.528912
16	6.205999	0.203580	0.152772	-0.179144	0.030020	-7.401202	78.042455	0.989682	3.175592	0.631116	8.508034
17	6.140264	0.044401	-0.111963	-8.194881	0.006293	26.813371	81.549597	0.992393	3.304863	0.096834	10.005137
18	5.985239	0.334834	0.297655	-0.254301	0.107361	5.326056	36.137575	0.989774	3.255648	0.520277	8.079908
19	6.218968	0.309468	0.029699	-0.162545	0.118048	54.520779	181.148742	0.996321	3.302286	0.254045	10.811922
20	6.696087	0.407913	-0.048879	-2.098421	0.056640	17.236542	67.682094	0.996294	3.469830	0.196641	10.246661
21	6.855161	0.230980	0.092447	-0.532175	0.186738	21.496664	42.991580	0.992230	3.306240	0.539875	7.094077

In [299…

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

Out[299…

```
fixed acidity            6.290587
volatile acidity         0.427309
citric acid             -0.260097
residual sugar          -6.052626
chlorides                0.021421
free sulfur dioxide    -10.483904
total sulfur dioxide    61.032187
density                  0.986866
pH                       3.355864
sulphates                0.445388
alcohol                  8.996359
Name: 14, dtype: float64
```

Example of Anomalous Sample

In [300…

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

```
Input row (original scale):
fixed acidity            6.290587
volatile acidity         0.427309
citric acid             -0.260097
residual sugar          -6.052626
chlorides                0.021421
free sulfur dioxide    -10.483904
total sulfur dioxide    61.032187
density                  0.986866
pH                       3.355864
sulphates                0.445388
alcohol                  8.996359
Name: 14, dtype: float64
```

In [301…

```
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: 99.99154 %
Probability of Hih Quality Quality: 0.008464073 %
```

In [302…

```
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 [303…

```
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: 83.0 %
Probability of Hih Quality Quality: 17.0 %
```