In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import accuracy_score, f1_score, recall_score
```

1. SVM with Submanifold Minimization

In [2]:
```
class SVM:
    def __init__(self, learning_rate=0.001, lambda_param=0.01, n_iters=1000):
        self.lr = learning_rate
        self.lambda_param = lambda_param
        self.n_iters = n_iters
        self.w = None
        self.b = None

    def fit(self, X, y):
        n_samples, n_features = X.shape

        # Map labels to +1 and -1
        y_ = np.where(y <= 0, -1, 1)

        # Initialize weights and bias
        self.w = np.zeros(n_features)
        self.b = 0

        # Submanifold Minimization Algorithm
        for _ in range(self.n_iters):
            for idx, x_i in enumerate(X):
                condition = y_[idx] * (np.dot(x_i, self.w) - self.b) >= 1
                if condition:
                    # Update weights for correctly classified points
                    self.w -= self.lr * (2 * self.lambda_param * self.w)
                else:
                    # Update weights and bias for misclassified points
                    self.w -= self.lr * (
                        2 * self.lambda_param * self.w - np.dot(x_i, y_[idx])
                    )
                    self.b -= self.lr * y_[idx]

    def predict(self, X):
        approx = np.dot(X, self.w) - self.b
        return np.sign(approx)
```

In [3]:
```
X_pos = np.array([[2.0, 2.2], [2.7, 2.5], [2.3, 2.0], [3.1, 2.3], [2.5, 2.4], [2.8, 2.7]])
y_pos = np.ones(len(X_pos))

X_neg = np.array([[1.6, 1.5], [2.0, 1.9], [2.1, 1.8], [1.7, 1.6], [1.8, 1.7], [2.0, 1.6]])
y_neg = -np.ones(len(X_neg))

# Combining the  positive and negative classes
X = np.vstack((X_pos, X_neg))
y = np.hstack((y_pos, y_neg))

clf = SVM(learning_rate=0.001, lambda_param=0.01, n_iters=1000)
clf.fit(X, y)

# Model parameters
print("Weights:", clf.w)
print("Bias:", clf.b)

# Predictions
predictions = clf.predict(X)
print("Predictions:", predictions)
```

```
Weights: [0.35074838 0.51297922]
Bias: 1.313999999999966
Predictions: [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
```

In [ ]:

In [4]:
```
accuracy = accuracy_score(y, predictions)
f1 = f1_score(y, predictions)
recall = recall_score(y, predictions)

print("Accuracy:", accuracy)
print("F1 Score:", f1)
print("Recall:", recall)
```

```
Accuracy: 0.5
F1 Score: 0.6666666666666666
Recall: 1.0
```

In [ ]:

In [ ]:

2. Imbalanced Classification with Sampling Techniques and MLP

In [5]:
```python
data = pd.read_csv('creditcard.csv')
data.head()
from sklearn.model_selection import train_test_split
X = data.drop('Class', axis=1)
y = data['Class']
```

In [6]:
```python
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
X_train.shape
```

Out[6]:
```
(227845, 30)
```

a.) Applying Smote Oversampling and Rnadom Undersampling

In [7]:
```python
from imblearn.over_sampling import SMOTE
from imblearn.under_sampling import RandomUnderSampler

# SMOTE Oversampling
smote = SMOTE(random_state=42)
X_train_smote, y_train_smote = smote.fit_resample(X_train, y_train)

#Random Undersampling
undersampler = RandomUnderSampler(random_state=42)
X_train_under, y_train_under = undersampler.fit_resample(X_train, y_train)

X_train_smote.shape, X_train_under.shape
```

Out[7]:
```
((454902, 30), (788, 30))
```

In [ ]:

b.)

In [8]:
```python
import torch
from torch import nn, optim
from torch.utils.data import DataLoader, TensorDataset
import torch.nn.functional as F
from tqdm import trange
from sklearn.metrics import accuracy_score, recall_score, f1_score

# Custom MLP Class
class CustomMLP(nn.Module):
    def __init__(self, input_size, hidden_layers):
        super(CustomMLP, self).__init__()
        self.layers = nn.ModuleList()
        self.activation_fn = nn.ReLU()

        # Hidden layers
        prev_size = input_size
        for hidden_size in hidden_layers:
            self.layers.append(nn.Linear(prev_size, hidden_size))
            prev_size = hidden_size

        # Output layer
        self.output = nn.Linear(prev_size, 1)

    def forward(self, x):
        for layer in self.layers:
            x = self.activation_fn(layer(x))
        return torch.sigmoid(self.output(x))
```

In [9]:
```python
# Dataset Preparation
def prepare_dataloader(X, y, batch_size=64):
    dataset = TensorDataset(torch.tensor(X.values, dtype=torch.float32),
                            torch.tensor(y.values, dtype=torch.float32).unsqueeze(1))
    return DataLoader(dataset, batch_size=batch_size, shuffle=True)



# Model Initialization
```

```python
input_size = X_train.shape[1]
hidden_layers = [64, 32, 16]
model = CustomMLP(input_size, hidden_layers)
device = "cpu"
model.to(device)

# Optimizer and Loss Function
optimizer = optim.Adam(model.parameters(), lr=0.001)
criterion = nn.BCELoss()
```

In [ ]:

```python

```

In [12]:

```python
# Training and test Functions
def train_epoch(model, dataloader, criterion, optimizer, device):
    model.train()
    train_loss = 0
    for X_batch, y_batch in dataloader:
        X_batch, y_batch = X_batch.to(device), y_batch.to(device)

        # Forward pass
        predictions = model(X_batch)
        loss = criterion(predictions, y_batch)

        # Backward pass
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        train_loss += loss.item() * X_batch.size(0)

    return train_loss / len(dataloader.dataset)

def test_epoch(model, dataloader, criterion, device):
    model.eval()
    test_loss = 0
    with torch.no_grad():
        for X_batch, y_batch in dataloader:
            X_batch, y_batch = X_batch.to(device), y_batch.to(device)
            predictions = model(X_batch)
            loss = criterion(predictions, y_batch)
            test_loss += loss.item() * X_batch.size(0)

    return test_loss / len(dataloader.dataset)


n_epochs = 50
batch_size = 64

datasets = {
    "Original": (X_train, y_train),
    "SMOTE": (X_train_smote, y_train_smote),
    "Undersampled": (X_train_under, y_train_under)
}

results = {}

for name, (X_data, y_data) in datasets.items():
    print(f"\nTraining on {name} Dataset...")
    train_loader = prepare_dataloader(X_data, y_data, batch_size)
    test_loader = prepare_dataloader(X_test, y_test, batch_size)

    model = CustomMLP(input_size, hidden_layers).to(device)
    optimizer = optim.Adam(model.parameters(), lr=0.001)
    criterion = nn.BCELoss()

    train_losses = []
    test_losses = []

    for epoch in trange(n_epochs, desc="Epochs"):
        train_loss = train_epoch(model, train_loader, criterion, optimizer, device)
        train_losses.append(train_loss)

        test_loss = test_epoch(model, test_loader, criterion, device)
        test_losses.append(test_loss)

        if epoch % 10 == 0:
            print(f"Epoch {epoch+1}/{n_epochs} - Train Loss: {train_loss:.4f} - Test Loss: {test_loss:.4f}")

    results[name] = {"train_losses": train_losses, "test_losses": test_losses}
```

```
Training on Original Dataset...
Epochs:   2%|▏         | 1/50 [00:30<24:55, 30.52s/it]
Epoch 1/50 - Train Loss: 0.1985 - Test Loss: 0.1720
```

```
Epochs:  22%|██        | 11/50 [04:18<14:36, 22.48s/it]
Epoch 11/50 - Train Loss: 0.1729 - Test Loss: 0.1720
Epochs:  42%|████      | 21/50 [2:30:00<59:04, 122.23s/it]
Epoch 21/50 - Train Loss: 0.1729 - Test Loss: 0.1720
Epochs:  62%|██████    | 31/50 [2:33:02<06:37, 20.92s/it]
Epoch 31/50 - Train Loss: 0.1729 - Test Loss: 0.1720
Epochs:  82%|████████  | 41/50 [2:36:10<03:02, 20.28s/it]
Epoch 41/50 - Train Loss: 0.1729 - Test Loss: 0.1720
Epochs: 100%|██████████| 50/50 [2:39:32<00:00, 191.44s/it]
Training on SMOTE Dataset...
Epochs:   2%|▏         | 1/50 [00:31<25:38, 31.39s/it]
Epoch 1/50 - Train Loss: 49.6609 - Test Loss: 99.7009
Epochs:  22%|██        | 11/50 [09:49<50:57, 78.41s/it]
Epoch 11/50 - Train Loss: 49.8580 - Test Loss: 99.6393
Epochs:  42%|████      | 21/50 [20:27<28:35, 59.14s/it]
Epoch 21/50 - Train Loss: 49.8167 - Test Loss: 0.1756
Epochs:  62%|██████    | 31/50 [36:28<45:31, 143.76s/it]
Epoch 31/50 - Train Loss: 49.7608 - Test Loss: 0.1756
Epochs:  82%|████████  | 41/50 [45:40<08:54, 59.43s/it]
Epoch 41/50 - Train Loss: 49.2281 - Test Loss: 98.6588
Epochs: 100%|██████████| 50/50 [1:00:09<00:00, 72.18s/it]
Training on Undersampled Dataset...
Epochs:   2%|▏         | 1/50 [00:02<02:00,  2.47s/it]
Epoch 1/50 - Train Loss: 47.1822 - Test Loss: 99.1081
Epochs:  22%|██        | 11/50 [00:32<01:53,  2.91s/it]
Epoch 11/50 - Train Loss: 49.7696 - Test Loss: 99.1953
Epochs:  42%|████      | 21/50 [01:14<02:16,  4.70s/it]
Epoch 21/50 - Train Loss: 49.5472 - Test Loss: 98.9971
Epochs:  62%|██████    | 31/50 [01:45<00:56,  2.95s/it]
Epoch 31/50 - Train Loss: 49.8874 - Test Loss: 99.5621
Epochs:  82%|████████  | 41/50 [04:24<00:52,  5.86s/it]
Epoch 41/50 - Train Loss: 49.8861 - Test Loss: 99.5444
Epochs: 100%|██████████| 50/50 [04:48<00:00,  5.77s/it]
```

In [ ]:

In [15]:

```python
# Function to evaluate a model
def evaluate_model(model, X_test, y_test, device):
    model.eval()
    with torch.no_grad():
        X_test_tensor = torch.tensor(X_test.values, dtype=torch.float32).to(device)
        y_test_tensor = torch.tensor(y_test.values, dtype=torch.float32).unsqueeze(1).to(device)

        predictions = model(X_test_tensor).cpu().numpy()
        predictions = (predictions >= 0.5).astype(int)

        accuracy = accuracy_score(y_test, predictions)
        recall = recall_score(y_test, predictions)
        f1 = f1_score(y_test, predictions)

        return accuracy, recall, f1

# Evaluating the models for each dataset
metrics = {}
for name, (X_data, y_data) in datasets.items():
    print(f"Evaluating model trained on {name} dataset...")
    accuracy, recall, f1 = evaluate_model(model, X_test, y_test, device)
    metrics[name] = {"Accuracy": accuracy, "Recall": recall, "F1-Score": f1}

# Display metrics
for dataset, values in metrics.items():
    print(f"\nMetrics for {dataset} Dataset:")
    for metric, value in values.items():
        print(f"{metric}: {value:.4f}")
```

```
Evaluating model trained on Original dataset...
Evaluating model trained on SMOTE dataset...
Evaluating model trained on Undersampled dataset...

Metrics for Original Dataset:
Accuracy: 0.0018
Recall: 1.0000
F1-Score: 0.0034

Metrics for SMOTE Dataset:
Accuracy: 0.0018
Recall: 1.0000
F1-Score: 0.0034

Metrics for Undersampled Dataset:
Accuracy: 0.0018
Recall: 1.0000
F1-Score: 0.0034
```

Comparing Results

- Original Dataset: Has less loss than the sampled datasets
- SMOTE Oversamplinga: low accuracy
- Random Undersampling: low accuracy

TRADEOFFS

- Accuracy decreasees with SMOTE and undersampling due to changes in class distributions.

In [ ]: