Oversampling and Undersampling In Machine Learning:

Oversampling increases the number of minority class samples by duplicating existing samples or creating new new synthetic ones.

Undersampling reduces the number of examples in the majority class by randomly removing some instances. It can be used when you have large dataset and want to reduce it’s size for faster processing.

Techniques:

Random Undersampling: Randomly remove samples from majority which posseses the risk of losing useful data.

Random Oversampling: Simply duplicates minority samples. Overfitting because duplicates don’t don’t add new information.

SMOTE: Creates synthetic samples between existing minority points using interpolation.

(if two points are close then a new synthetic point is created in between.) Very popular and reduces overfitting. And not ideal for categorical data directly and can create noisy overlapping samples if data has outliers. So you have to remove outliers before using SMOTE.

Types of SMOTE:

ADSYN: focuses more on minority samples that are harder to learn.

SMOTENC: Works for categorical + numerical data.

Combines Approach for heavy imbalance:

SMOTE + Tomek Links

SMOTE + ENN

Algorithm Level Approaches – class\_weight='balanced' (in model)

from imblearn.combine import SMOTEENN

from imblearn.over\_sampling import SMOTE

from imblearn.under\_sampling import TomekLinks

from sklearn.model\_selection import train\_test\_split

# Split data

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, stratify=y, test\_size=0.2)

# Apply SMOTE

smote = SMOTE(sampling\_strategy='auto', random\_state=42)

X\_res, y\_res = smote.fit\_resample(X\_train, y\_train)

# Or Hybrid

smote\_enn = SMOTEENN(random\_state=42)

X\_res, y\_res = smote\_enn.fit\_resample(X\_train, y\_train)

|  |  |
| --- | --- |
| Majority data is *very large* (e.g., millions), minority is *tiny* | **SMOTE + Tomek Links** or **SMOTE + ENN** |

|  |  |
| --- | --- |
| Minority class has <100 samples | **ADASYN** or **Augmentation** (e.g., GANs, image transformations) |

|  |  |
| --- | --- |
| Dataset is mostly **categorical** | **SMOTENC** |

|  |  |
| --- | --- |
| You have **continuous + categorical** mix | **KMeans-SMOTE** or **SMOTENC** |

|  |  |
| --- | --- |
| You want quick & simple fix | class\_weight='balanced' (in model) |

|  |  |
| --- | --- |
| You want maximum performance | Use **hybrid**: SMOTE + undersampling + class weighting |

Practical tips:

Always split your data first, then apply oversampling only on the training set, not the test set.

Check class distribution before and after sampling.

Use cross-validation with stratification (StratifiedKFold).

Evaluate using Recall, F1, ROC-AUC, not Accuracy (accuracy is misleading in imbalanced datasets).

AutoEncoder:

You can use it to:

Learn structure of minority data.

Generate new synthetic samples from that learned representation.

from tensorflow.keras.models import Model

from tensorflow.keras.layers import Input, Dense

# Autoencoder for minority class

input\_dim = X\_minority.shape[1]

encoding\_dim = 8

input\_layer = Input(shape=(input\_dim,))

encoder = Dense(encoding\_dim, activation='relu')(input\_layer)

decoder = Dense(input\_dim, activation='sigmoid')(encoder)

autoencoder = Model(inputs=input\_layer, outputs=decoder)

autoencoder.compile(optimizer='adam', loss='mse')

autoencoder.fit(X\_minority, X\_minority, epochs=100, batch\_size=16)

# Generate synthetic samples

encoded = encoder.predict(X\_minority)

noise = np.random.normal(0, 0.1, encoded.shape)

synthetic\_encoded = encoded + noise

synthetic\_samples = decoder.predict(synthetic\_encoded)

**Advantages:**

* Learns the real feature distribution.
* Doesn’t just interpolate like SMOTE.
* Great for **numerical**, **image**, and **tabular** data.

❌ **Disadvantages:**

* Requires enough minority samples to learn structure (at least ~30–50).
* Harder to tune than SMOTE.

**2. GAN-based Sampling (Generative Adversarial Networks)**

This is the **most powerful modern technique** for extremely imbalanced datasets.

**🔹 What It Is:**

A **GAN (Generative Adversarial Network)** consists of two neural networks:

1. **Generator (G)** → Creates fake (synthetic) data
2. **Discriminator (D)** → Tries to distinguish real from fake

They train together in a **zero-sum game**, and the generator learns to create **realistic synthetic samples**.

**How GANs Are Used for Imbalanced Data**

Train a GAN **only on minority class data**.  
Once trained, use the generator to create **realistic synthetic samples** of the minority class, which you add to your dataset.

| **GAN Type** | **Description** | **Use Case** |
| --- | --- | --- |
| **Vanilla GAN** | Basic GAN, works on numeric data | Tabular imbalance |
| **Conditional GAN (cGAN)** | Adds class labels to control generation | Multi-class data |
| **Wasserstein GAN (WGAN)** | Improves training stability | Complex data distributions |
| **Tabular GAN (CTGAN, TVAE)** | Specialized for tabular data | Imbalanced structured data |

from sdv.tabular import CTGAN

ctgan = CTGAN(epochs=300)

ctgan.fit(X\_minority)

synthetic\_data = ctgan.sample(1000) # generate 1000 new minority samples

**Advantages:**

* Generates diverse, realistic samples.
* Works even for extremely rare minority cases (<50 samples).
* Great for **high-dimensional, non-linear, and mixed-type data**.

❌ **Disadvantages:**

* Computationally expensive.
* Requires careful tuning (GANs can collapse or overfit).

**3. Variational Autoencoders (VAE)**

VAEs are similar to autoencoders but **probabilistic** — they learn a *distribution* in latent space instead of fixed points.

You can sample from that distribution to create **new synthetic data** that reflects the variability of the minority class

# Conceptually:

latent\_vector = np.random.normal(0, 1, (num\_samples, latent\_dim))

synthetic\_samples = decoder.predict(latent\_vector)

✅ **Pros:**

* More diverse and stable than plain autoencoders
* Generates smooth variations of minority samples

❌ **Cons:**

* Harder to train
* Needs enough minority examples to learn distribution

**5. Best Practice Workflow for Severe Imbalance**

If your minority data is *very small* and imbalance is *extreme*, use this hybrid strategy:

1. **Train Autoencoder or VAE** on minority samples → generate synthetic data.
2. Combine those with the **original data**.
3. Apply **SMOTE + Tomek Links** for extra balance refinement.
4. Use **class weights** in your final model.

This pipeline ensures:

* Synthetic diversity (GAN/Autoencoder)
* Boundary cleaning (Tomek Links)
* Balanced learning (Class weights)

Process:

Step 1: Collect data

Step 2: Separate minority and majority classes

Step 3: Train GAN or Autoencoder on minority class

Step 4: Generate synthetic minority data

Step 5: Merge with original dataset

Step 6: Apply SMOTE + Tomek Links for fine balancing

Step 7: Train classifier with class weights

For very few minority class and categorical data:

# Example

from sklearn.preprocessing import OneHotEncoder, MinMaxScaler

from tensorflow.keras.models import Model

from tensorflow.keras.layers import Input, Dense

# Encode categorical + scale numeric

# X\_processed = np.hstack([encoded\_categorical, scaled\_numeric])

input\_dim = X\_processed.shape[1]

encoding\_dim = 8

input\_layer = Input(shape=(input\_dim,))

encoder = Dense(encoding\_dim, activation='relu')(input\_layer)

decoder = Dense(input\_dim, activation='sigmoid')(encoder)

autoencoder = Model(inputs=input\_layer, outputs=decoder)

autoencoder.compile(optimizer='adam', loss='mse')

autoencoder.fit(X\_processed, X\_processed, epochs=100, batch\_size=32)

Regularization:

**Choosing the Right Regularization**

| **Situation** | **Recommended** |
| --- | --- |
| Too many irrelevant features | **L1 (Lasso)** |
| Multicollinearity (correlated features) | **L2 (Ridge)** |
| Both sparsity + correlation | **Elastic Net** |
| Neural Network | **Dropout / L2 Weight Decay** |

**1. L1 Regularization (Lasso Regression)**

* Adds **absolute values of weights** as penalty:
* Encourages **sparsity** — it can **reduce some coefficients to zero**, effectively doing **feature selection**.
* Useful when you have **many features**, and you believe **only a few are important**.

🟢 **Key point:** L1 = feature selection + regularization.

**L2 Regularization (Ridge Regression)**

* Adds **square of weights** as penalty:
* It **shrinks coefficients** towards zero but never makes them exactly zero.
* Helps prevent large weights and reduces variance in the model.

🟢 **Key point:** L2 = weight shrinkage (no feature removal).

**3. Elastic Net Regularization**

* Combines **L1 and L2 penalties**:

It **balances** between sparsity (L1) and smoothness (L2).

* Very useful when you have **many correlated features**.

🟢 **Key point:** Elastic Net = mix of Ridge + Lasso.