#1)

Your code is setting up a machine learning pipeline for **credit card fraud detection** using **classification models** while addressing **class imbalance**. Below is a breakdown of the code along with key points.

**Code Explanation**

**1. Importing Libraries**

The script imports necessary libraries for:

* **Numerical & Data Handling:** numpy, pandas
* **Machine Learning Models:** sklearn (Logistic Regression, SVM, Decision Tree, etc.)
* **Imbalanced Data Handling:** imblearn (SMOTE, NearMiss)
* **Visualization:** matplotlib, seaborn
* **Dimensionality Reduction:** PCA, t-SNE, Truncated SVD
* **Performance Metrics:** accuracy\_score, precision\_score, recall\_score, etc.

**2. Loading the Dataset**

df = pd.read\_csv('/content/creditcard.csv')

df.head()

* Reads the **credit card fraud detection dataset**.
* Displays the first few rows.

**Key Points**

**1. Data Understanding & Preprocessing**

* The dataset consists of **credit card transactions** with a **binary label** (Class column):
  + 0 → Legitimate Transaction
  + 1 → Fraudulent Transaction
* The dataset is highly **imbalanced** (fraud cases are much fewer than normal cases).
* Feature engineering may be needed for better model performance.

**2. Handling Class Imbalance**

* **SMOTE (Synthetic Minority Over-sampling Technique)**: Increases fraud samples artificially.
* **NearMiss (Under-sampling)**: Reduces non-fraud samples to balance the dataset.

**3. Model Selection**

The code imports various classification models:

* **Logistic Regression** (Good baseline)
* **Support Vector Machine (SVM)** (Handles high-dimensional data well)
* **K-Nearest Neighbors (KNN)** (Sensitive to class imbalance)
* **Decision Trees** (Can overfit but useful)
* **Random Forest** (Good performance, handles imbalanced data well)

**4. Performance Metrics**

* **Accuracy**: Not useful in imbalanced datasets.
* **Precision & Recall**: More important (we want fewer false negatives).
* **F1-Score**: Balances Precision & Recall.
* **ROC-AUC Score**: Measures model discrimination ability.

**Next Steps**

* **Check dataset info** (df.info())
* **Visualize class distribution** (sns.countplot(x='Class', data=df))
* **Split data into training/testing** (train\_test\_split)
* **Train models & evaluate using metrics**
* **Optimize with hyperparameter tuning** (GridSearchCV)

Would you like a more detailed guide on data preprocessing or model training? 🚀

#2)

Your code is focused on **splitting the dataset** while maintaining the original class distribution using **Stratified K-Fold cross-validation**. Below is an explanation and key points.

**Code Explanation**

**1. Checking Class Distribution**

print('No Frauds', round(df['Class'].value\_counts()[0]/len(df) \* 100,2), '% of the dataset')

print('Frauds', round(df['Class'].value\_counts()[1]/len(df) \* 100,2), '% of the dataset')

* Prints the percentage of fraudulent and non-fraudulent transactions.
* Since fraud cases are rare, handling class imbalance is necessary.

**2. Splitting Features & Labels**

X = df.drop('Class', axis=1) # Features

y = df['Class'] # Target label (fraud or not)

* X contains transaction details (features).
* y contains fraud labels (0 for normal, 1 for fraud).

**3. Using Stratified K-Fold for Balanced Splitting**

sss = StratifiedKFold(n\_splits=5, random\_state=None, shuffle=False)

* Ensures each fold maintains the same proportion of fraud and non-fraud cases.
* n\_splits=5: The dataset will be divided into **5 folds**.

for train\_index, test\_index in sss.split(X, y):

print("Train:", train\_index, "Test:", test\_index)

original\_Xtrain, original\_Xtest = X.iloc[train\_index], X.iloc[test\_index]

original\_ytrain, original\_ytest = y.iloc[train\_index], y.iloc[test\_index]

* Splits the data into **train and test sets** while preserving the class ratio.
* **Prints train & test indices** to verify how the data is being split.

**4. Converting Data to NumPy Arrays**

original\_Xtrain = original\_Xtrain.values

original\_Xtest = original\_Xtest.values

original\_ytrain = original\_ytrain.values

original\_ytest = original\_ytest.values

* Converts Pandas DataFrame to **NumPy arrays** for efficient processing.

**5. Checking Label Distribution in Train/Test Sets**

train\_unique\_label, train\_counts\_label = np.unique(original\_ytrain, return\_counts=True)

test\_unique\_label, test\_counts\_label = np.unique(original\_ytest, return\_counts=True)

* Extracts **unique class labels** and their respective counts.

print('-' \* 100)

print('Label Distributions: \n')

print(train\_counts\_label/ len(original\_ytrain)) # Distribution in train set

print(test\_counts\_label/ len(original\_ytest)) # Distribution in test set

* **Prints class distributions** in both train and test sets to ensure they match the original dataset.

**Key Points**

✅ **Stratified Sampling**: Ensures both train and test sets maintain the same fraud-to-non-fraud ratio.  
✅ **Prevents Bias**: Avoids models being trained on a dataset where fraud cases are **over/under-represented**.  
✅ **Converts Data to NumPy**: Makes processing faster and compatible with ML models.  
✅ **Validates Class Distribution**: Helps verify that the stratified split worked correctly.

**Next Steps**

* **Feature Scaling**: Apply StandardScaler or MinMaxScaler for normalization.
* **Class Imbalance Handling**: Use **SMOTE (oversampling)** or **NearMiss (undersampling)**.
* **Train & Evaluate Models**: Use classifiers like **Random Forest, Logistic Regression, SVM, etc.**.

Would you like help with **feature scaling or model training** next? 🚀

#3)

**. Feature Scaling**

Feature scaling ensures that numerical values are within a similar range, improving model performance.

**🔹 Key Points**

1. **Why?** Many ML algorithms (SVM, Logistic Regression, KNN) perform better with scaled features.
2. **Standardization (Z-score scaling):** (X - mean) / std deviation – centers around zero.
3. **Normalization (Min-Max Scaling):** (X - min) / (max - min) – scales between 0 and 1.
4. **Which to use?**
   * **Standardization** → Recommended for normal distribution.
   * **Min-Max Scaling** → Useful for bounded data (e.g., image processing).
5. **Implementation:**
6. from sklearn.preprocessing import StandardScaler
7. scaler = StandardScaler()
8. X\_train\_scaled = scaler.fit\_transform(original\_Xtrain)
9. X\_test\_scaled = scaler.transform(original\_Xtest)
10. **Avoid Data Leakage:** Always fit on **training data only** and apply to test data.
11. **Check Distribution:** Use histograms before & after scaling (sns.histplot()).
12. **Scaling with Pipelines:** Can be integrated into ML pipelines (make\_pipeline()).
13. **Effect on Distance-Based Models:** Essential for **KNN, SVM, PCA, t-SNE**.
14. **Inverse Transform Possible:** You can revert scaling using .inverse\_transform().

**#4**

**Handling Class Imbalance**

Fraud detection datasets have very few fraud cases, making imbalance handling necessary.

**🔹 Key Points**

1. **Why?** Imbalanced data leads to biased models that favor the majority class.
2. **Oversampling (SMOTE):** Creates synthetic minority class samples.
3. from imblearn.over\_sampling import SMOTE
4. smote = SMOTE(sampling\_strategy=0.5) # Adjust ratio
5. X\_train\_sm, y\_train\_sm = smote.fit\_resample(X\_train\_scaled, original\_ytrain)
6. **Undersampling (NearMiss):** Removes majority class samples to balance the dataset.
7. from imblearn.under\_sampling import NearMiss
8. nm = NearMiss()
9. X\_train\_nm, y\_train\_nm = nm.fit\_resample(X\_train\_scaled, original\_ytrain)
10. **Hybrid Approach:** Combines **SMOTE (oversampling)** & **NearMiss (undersampling)**.
11. **Choose Based on Dataset Size:** If data is **small**, use **oversampling**; if **large**, try **undersampling**.
12. **Stratified Sampling:** Always check fraud-to-non-fraud ratio remains realistic.
13. **Check Performance Metrics:** Compare models with & without rebalancing.
14. **Avoid Overfitting:** Oversampling may lead to **overfitting** if synthetic samples are too similar.
15. **Ensemble Methods Help:** Models like **Random Forest & XGBoost** handle imbalance well.
16. **Evaluate on Original Distribution:** Ensure model is tested on **real-world** class distribution.

**#5**

**Model Training**

Different classifiers will be trained and compared for fraud detection.

**🔹 Key Points**

1. **Baseline Model First:** Start with **Logistic Regression** for quick benchmarking.
2. **Tree-Based Models Perform Well:** **Random Forest, XGBoost** handle imbalance naturally.
3. **Distance-Based Models Need Scaling:** **KNN, SVM** require standardized data.
4. **Train on Balanced Data:** Use SMOTE or NearMiss datasets for training.
5. **Cross-Validation is Essential:** Use **StratifiedKFold** to ensure balanced splits.
6. from sklearn.model\_selection import StratifiedKFold
7. skf = StratifiedKFold(n\_splits=5)
8. **Hyperparameter Tuning:** Use GridSearchCV to optimize models.
9. **Feature Selection:** Use **PCA or Feature Importance** to reduce dimensionality.
10. **Try Different Models:**
    * Logistic Regression
    * SVM
    * Random Forest
    * XGBoost
    * Neural Networks (TensorFlow/PyTorch)
11. **Use Pipelines:** Combine **scaling, resampling, and modeling** into one workflow.
12. **Save Models for Deployment:** Use joblib to save trained models.

**#6**

**Model Evaluation**

Choosing the right metric is crucial for fraud detection.

**🔹 Key Points**

1. **Accuracy is Misleading:** High accuracy doesn’t mean good fraud detection.
2. **Use Precision & Recall:**
   * **Precision:** TP / (TP + FP) → How many predicted frauds were correct?
   * **Recall:** TP / (TP + FN) → How many actual frauds were detected?
3. **F1-Score:** Balances precision & recall.
4. from sklearn.metrics import f1\_score
5. print(f1\_score(y\_test, y\_pred))
6. **ROC-AUC Score:** Measures classifier’s ability to distinguish between classes.
7. **Confusion Matrix:** Visualizes **TP, FP, TN, FN** to understand errors.
8. **Precision-Recall Curve:** Useful when positive class (fraud) is rare.
9. **Threshold Tuning:** Adjust classification threshold for better fraud detection.
10. **Check False Negatives:** Missing a fraud is worse than predicting false frauds.
11. **Cross-Validation Metrics:** Ensure consistency across different train-test splits.
12. **Compare Models Fairly:** Always evaluate **on the same test set**.

**#7**

**Model Optimization**

Fine-tuning models for better fraud detection performance.

**🔹 Key Points**

1. **Hyperparameter Tuning:** Use GridSearchCV or RandomizedSearchCV.
2. **Feature Engineering:** Create new features from transaction time, amount, etc.
3. **Feature Selection:** Remove redundant features using **PCA, Recursive Feature Elimination (RFE)**.
4. **Regularization Helps:** L1 (Lasso) & L2 (Ridge) prevent overfitting.
5. **Try Different Sampling Strategies:** Experiment with different SMOTE/undersampling ratios.
6. **Use Ensemble Learning:** Combine multiple models for better predictions.
7. **Fine-Tune Class Weights:** Set class\_weight='balanced' in models like SVM, Logistic Regression.
8. **Deep Learning for Feature Extraction:** Autoencoders can learn useful patterns.
9. **Handle Anomalies Separately:** Anomaly detection techniques (e.g., Isolation Forest) might help.
10. **Real-World Testing:** Validate on an unseen, **real-world distribution dataset**.

**Conclusion**

By following these steps, you can **build a robust fraud detection model**. 🚀  
Would you like a **hands-on implementation** of any of these steps? 😊