Feature Selection Techniques for Fraud Detection

In this video, we will walk through a comprehensive process of applying the following feature engineering techniques:

1. Filter Methods
2. Wrapper Methods
3. Embedded Methods

Import necessary libraries

In [2]:
```
import pandas as pd
import numpy as np
from sklearn.feature_selection import VarianceThreshold
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import chi2
from sklearn.feature_selection import mutual_info_classif
from sklearn.model_selection import train_test_split
```

Import the dataset

In [3]:
```
# Load data into pandas DataFrame
df = pd.read_csv('C:/Users/Amarkou/Documents/Ecourse/creditcard.csv')
# Select the first 30,000 rows of the DataFrame
df = df.head(30000)
```

Split data into training and testing sets

In [4]:
```
# Separate target from features
X = df.drop("Class", axis=1)
y = df["Class"]

# Train test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

1. Filter Methods

1.1 Pearson Correlation

In [5]:
```
# Calculate the Pearson correlation coefficients
cor_list = []
for i in X_train.columns.tolist():
    cor = np.corrcoef(X_train[i], y_train)[0, 1]
    cor_list.append(cor)

# Replace NaN with 0
cor_list = [0 if np.isnan(i) else i for i in cor_list]

# Feature name
cor_feature = X_train.iloc[:,np.argsort(np.abs(cor_list))[-10:]].columns.tolist()

# Feature selection
X_train_filtered = X_train[cor_feature]
X_test_filtered = X_test[cor_feature]
```

1.2 Variance Threshold

Variance Threshold is a simple baseline approach to feature selection. It removes all features which variance doesn't meet some threshold. By default, it removes all zero-variance features, i.e., features that have the same value in all samples.

In [6]:
```
# Implementing Variance Threshold
selector = VarianceThreshold(threshold=0.5)
selector.fit_transform(X_train)

# Get columns to keep and create new dataframe with those only
cols = selector.get_support(indices=True)
X_train_low_variance = X_train.iloc[:,cols]
```

1.3 Chi-Squared Test

The Chi-Square statistic is commonly used for testing relationships between categorical variables. In feature selection, we aim to select the features which are highly dependent on the response.

In [8]:
```
# Apply Chi-Squared Test
chi_selector = SelectKBest(chi2, k=10)
chi_selector.fit_transform(abs(X_train), y_train)

# Get columns to keep and create new dataframe with those only
cols = chi_selector.get_support(indices=True)
X_train_chi2 = X_train.iloc[:,cols]
```

1.4 Mutual Information

Mutual information measures the information that X and Y share: It measures how much knowing one of these variables reduces uncertainty about the other. For example, if X and Y are independent, then knowing X does not give any information about Y and vice versa, so their mutual information is zero.

In [9]:
```
# Apply Mutual Information
mi_selector = SelectKBest(mutual_info_classif, k=10)
mi_selector.fit_transform(X_train, y_train)

# Get columns to keep and create new dataframe with those only
cols = mi_selector.get_support(indices=True)
X_train_mi = X_train.iloc[:,cols]
```

2. Wrapper Methods

2.1 Recursive Feature Elimination (RFE)

Recursive feature elimination (RFE) is a feature selection method that fits a model and removes the weakest feature (or features) until the specified number of features is reached.

We're starting by initializing the RFE model using logistic regression as the estimator. Then, we fit the model to our training data and transform our data to only include the selected features.

In [10]:
```
from sklearn.feature_selection import RFE
from sklearn.linear_model import LogisticRegression

# Initialize an RFE model using the logistic regression estimator
model = LogisticRegression(max_iter=1000)
rfe = RFE(estimator=model, n_features_to_select=10, step=1)

# Fit the model
rfe.fit(X_train, y_train)

# Transform the data
X_train_rfe = rfe.transform(X_train)
X_test_rfe = rfe.transform(X_test)
```

```
C:\Users\AMarkou\Anaconda3\lib\site-packages\sklearn\linear_model\_logistic.py:458: ConvergenceWarning: lbfg
s failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html (https://scikit-learn.org/stable/modules/prep
rocessing.html)
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression (https://scikit-learn.org/
stable/modules/linear_model.html#logistic-regression)
  n_iter_i = _check_optimize_result(
C:\Users\AMarkou\Anaconda3\lib\site-packages\sklearn\linear_model\_logistic.py:458: ConvergenceWarning: lbfg
s failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html (https://scikit-learn.org/stable/modules/prep
rocessing.html)
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression (https://scikit-learn.org/
stable/modules/linear_model.html#logistic-regression)
  n_iter_i = _check_optimize_result(
```

2.2 Sequential Feature Selection (SFS)

Sequential Feature Selection (SFS) is a type of greedy search algorithm that is used to reduce an initial d-dimensional feature space to a k-dimensional feature subspace where k < d.

We're starting by initializing the SFS model using KNN as the estimator. Then, we fit the model to our training data and transform our data to only include the selected features.

In [11]:
```
from sklearn.feature_selection import SequentialFeatureSelector
from sklearn.neighbors import KNeighborsClassifier

# Initialize an SFS model using the KNN estimator
knn = KNeighborsClassifier(n_neighbors=3)
sfs = SequentialFeatureSelector(knn, n_features_to_select=10)

# Fit the model
sfs.fit(X_train, y_train)

# Transform the data
X_train_sfs = sfs.transform(X_train)
X_test_sfs = sfs.transform(X_test)
```

2.3 Genetic Algorithms

Genetic Algorithms are search based algorithms based on the concepts of natural selection and genetics.

Unfortunately, there's no direct implementation for GA in scikit-learn, but there are several packages, such as DEAP, that can be used for this.

Here, we define a custom evaluation function that uses a basic neural network classifier. Then we define the various genetic algorithm functions and parameters, including creating the initial population, defining the evaluation, mating, mutation, and selection methods, and then run the algorithm for a defined number of generations.

In [12]:
```
# Install DEAP if not already installed
# !pip install deap

from deap import creator, base, tools, algorithms
from sklearn import neural_network

# Define the evaluation function
def evaluate(individual):
    mask = list(map(bool, individual))
    return (neural_network.MLPClassifier().fit(X_train.iloc[:, mask], y_train).score(X_test.iloc[:, mask], y_t

# Define the genetic algorithm functions and parameters
creator.create("FitnessMax", base.Fitness, weights=(1.0,))
creator.create("Individual", list, fitness=creator.FitnessMax)

toolbox = base.Toolbox()
toolbox.register("attr_bool", np.random.choice, 2, p=[0.1, 0.9])
toolbox.register("individual", tools.initRepeat, creator.Individual, toolbox.attr_bool, len(X_train.columns))
toolbox.register("population", tools.initRepeat, list, toolbox.individual)
toolbox.register("evaluate", evaluate)
toolbox.register("mate", tools.cxTwoPoint)
toolbox.register("mutate", tools.mutFlipBit, indpb=0.05)
toolbox.register("select", tools.selTournament, tournsize=3)

# Run the genetic algorithm
pop = toolbox.population(n=50)
hof = tools.HallOfFame(1)
stats = tools.Statistics(lambda ind: ind.fitness.values)
stats.register("avg", np.mean)
stats.register("std", np.std)
stats.register("min", np.min)
stats.register("max", np.max)

pop, log = algorithms.eaSimple(pop, toolbox, cxpb=0.5, mutpb=0.2, ngen=40,
                               stats=stats, halloffame=hof, verbose=True)
```

gen	nevals	avg	std	min	max
0	50	0.996833	0.00125698	0.993333	0.999167
1	36	0.997483	0.00185719	0.987833	0.999167
2	27	0.998393	0.000803299	0.996167	0.999167
3	24	0.998783	0.000551009	0.996333	0.999333
4	27	0.998863	0.000430362	0.997667	0.9995
5	26	0.998733	0.00106771	0.993167	0.9995
6	32	0.998983	0.000316667	0.998	0.9995
7	27	0.998993	0.00041628	0.997333	0.9995
8	24	0.99891	0.000850627	0.9955	0.9995
9	25	0.99911	0.000414474	0.998	0.9995
10	26	0.999147	0.000336056	0.998167	0.9995
11	35	0.998967	0.000422953	0.998167	0.9995
12	27	0.99904	0.000541151	0.996	0.9995
13	24	0.998993	0.000645463	0.996	0.9995
14	25	0.999067	0.000521749	0.996833	0.9995
15	33	0.998977	0.00045948	0.997333	0.9995
16	34	0.998613	0.00179178	0.986667	0.9995
17	37	0.998957	0.000337984	0.998167	0.9995
18	34	0.99883	0.000661219	0.995167	0.9995
19	25	0.99897	0.00049068	0.997167	0.9995
20	35	0.99894	0.000366303	0.998	0.999667
21	24	0.999007	0.000452106	0.997	0.999667
22	31	0.999047	0.000344867	0.998333	0.999667
23	33	0.998997	0.000374893	0.998	0.999667
24	19	0.999173	0.000374106	0.998167	0.999667
25	29	0.999143	0.000393008	0.998	0.999667
26	32	0.9991	0.00034641	0.998167	0.999667
27	35	0.999037	0.000383391	0.998167	0.999667
28	30	0.999103	0.000358841	0.998167	0.999667
29	23	0.999127	0.000347307	0.998167	0.999667
30	25	0.99906	0.00049862	0.997	0.999667
31	34	0.99888	0.00136953	0.989667	0.999667
32	25	0.999127	0.000376475	0.998167	0.999667
33	25	0.99912	0.000452204	0.997833	0.999667
34	37	0.998923	0.00060938	0.996167	0.999667
35	33	0.998973	0.000498174	0.997	0.999667
36	25	0.99902	0.000383898	0.998167	0.999667
37	28	0.999033	0.000377124	0.998167	0.999667
38	26	0.99906	0.0005393	0.996	0.999667
39	33	0.99903	0.000676338	0.994833	0.999667
40	31	0.999087	0.000377477	0.998	0.999667

We're selecting the top features according to the optimal solution (individual) found by the genetic algorithm.

In [14]:
```
# Select top features
top_features = [X_train.columns[i] for i in range(len(hof[0])) if hof[0][i] == 1]
X_train_ga = X_train[top_features]
X_test_ga = X_test[top_features]
```

3. Embedded Methods

3.1 Lasso Regression

Lasso (Least Absolute Shrinkage and Selection Operator) adds "absolute value of magnitude" of coefficient as penalty term to the loss function. This can lead to the reduction of some coefficients to zero, effectively performing feature selection.

In [15]:
```
from sklearn.linear_model import LassoCV

# Initialize the LassoCV model
lasso = LassoCV(cv=5)

# Fit the model
lasso.fit(X_train, y_train)

# Select features
lasso_mask = lasso.coef_ != 0
X_train_lasso = X_train.loc[:, lasso_mask]
X_test_lasso = X_test.loc[:, lasso_mask]
```

3.2 Elastic Net

Elastic Net is a middle ground between Lasso Regression and Ridge Regression. It includes the penalties of both models, effectively shrinking some coefficients and setting some to zero for feature selection.

In [16]:
```
from sklearn.linear_model import ElasticNetCV

# Initialize the ElasticNetCV model
elastic = ElasticNetCV(cv=5)

# Fit the model
elastic.fit(X_train, y_train)

# Select features
elastic_mask = elastic.coef_ != 0
X_train_elastic = X_train.loc[:, elastic_mask]
X_test_elastic = X_test.loc[:, elastic_mask]
```

3.3 Decision Trees

Decision Trees are able to rank features based on their importance by the amount that each feature decrease the weighted impurity.

In [17]:
```
from sklearn.tree import DecisionTreeClassifier

# Initialize the DecisionTreeClassifier model
tree = DecisionTreeClassifier()

# Fit the model
tree.fit(X_train, y_train)

# Select features based on feature importances
tree_mask = tree.feature_importances_ > 0.01
X_train_tree = X_train.loc[:, tree_mask]
X_test_tree = X_test.loc[:, tree_mask]
```

3.4 Random Forests

Similarly to Decision Trees, Random Forests are also able to rank features based on their importance.

In [18]:
```
from sklearn.ensemble import RandomForestClassifier

# Initialize the RandomForestClassifier model
forest = RandomForestClassifier()

# Fit the model
forest.fit(X_train, y_train)

# Select features based on feature importances
forest_mask = forest.feature_importances_ > 0.01
X_train_forest = X_train.loc[:, forest_mask]
X_test_forest = X_test.loc[:, forest_mask]
```

3.5 Gradient Boosting

Just like Decision Trees and Random Forests, Gradient Boosting models are also able to rank features based on their importance.

In [19]:
```
from sklearn.ensemble import GradientBoostingClassifier

# Initialize the GradientBoostingClassifier model
gbc = GradientBoostingClassifier()

# Fit the model
gbc.fit(X_train, y_train)

# Select features based on feature importances
gbc_mask = gbc.feature_importances_ > 0.01
X_train_gbc = X_train.loc[:, gbc_mask]
X_test_gbc = X_test.loc[:, gbc_mask]
```