7. Applications of Data Mining for Fraud Detection - Part 4: Insurance

In this video, we will walk through a comprehensive process of applying machine learning techniques using real-life data. We will train test and evaluate from the following family of algorithms:

1. Supervised
2. Ensemble
3. Unsupervised

Import necessary libraries

In [2]:
```
# Import necessary libraries
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.impute import SimpleImputer
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from sklearn.naive_bayes import GaussianNB
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import silhouette_score
from sklearn.cluster import KMeans
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, roc_auc_score, roc_curve
from sklearn.model_selection import GridSearchCV
import matplotlib.pyplot as plt

import warnings


import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.metrics import confusion_matrix, accuracy_score, f1_score, roc_auc_score, classification_report
from sklearn.ensemble import BaggingClassifier, RandomForestClassifier, AdaBoostClassifier, GradientBoostingCl
from xgboost import XGBClassifier

from sklearn.ensemble import StackingClassifier, VotingClassifier

# Ignore all warnings
warnings.filterwarnings("ignore")
```

Import the dataset

In [3]:
```
# Load the data
df = pd.read_csv('insurance_claims.csv')

# Convert categorical 'fraud_reported' to numerical
df['fraud_reported'] = df['fraud_reported'].apply(lambda x: 1 if x == 'Y' else 0)

# Select numerical columns only
num_cols = df.select_dtypes(include=['float64', 'int64']).columns
df = df[num_cols]

# Fill NaNs
df.fillna(df.mean(), inplace=True)
df
```

Out[3]:

	months_as_customer	age	policy_number	policy_deductable	policy_annual_premium	umbrella_limit	insured_zip	capital-gains	capital-loss
0	328	48	521585	1000	1406.91	0	466132	53300	0
1	228	42	342868	2000	1197.22	5000000	468176	0	0
2	134	29	687698	2000	1413.14	5000000	430632	35100	0
3	256	41	227811	2000	1415.74	6000000	608117	48900	-62400
4	228	44	367455	1000	1583.91	6000000	610706	66000	-46000
...	...	...	...	...	...	...	...	...	...
995	3	38	941851	1000	1310.80	0	431289	0	0
996	285	41	186934	1000	1436.79	0	608177	70900	0
997	130	34	918516	500	1383.49	3000000	442797	35100	0
998	458	62	533940	2000	1356.92	5000000	441714	0	0
999	456	60	556080	1000	766.19	0	612260	0	0

1000 rows × 20 columns

Split data into training and testing sets

In [36]:
```
# Split the data into training and test sets
X = df.drop('fraud_reported', axis=1)
y = df['fraud_reported']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Fill NaNs
imp = SimpleImputer(strategy='mean') # you can use 'median' or 'most_frequent' depending on your data
X_train = imp.fit_transform(X_train)
X_test = imp.transform(X_test)

# Save column names
columns = X.columns

# Normalize the data
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
#y_train = scaler.fit_transform(y_train)
#y_test = scaler.fit_transform(y_test)
```

Supervised Learning Modelling Process

Define hyperparameters to tune for each algorithm

In [12]:
```
lr_params = {'C': [0.1, 1, 10], 'penalty': ['l1', 'l2'], 'solver': ['liblinear']}
dt_params = {'criterion': ['gini', 'entropy'], 'max_depth': [5, 10, 20]}
nb_params = {}
svm_params = {'C': [0.1, 1, 10], 'kernel': ['linear', 'poly', 'rbf', 'sigmoid']}
nn_params = {'hidden_layer_sizes': [(50,), (100,), (50, 50)], 'activation': ['relu', 'logistic'], 'solver': [
```

Train and Test the models

In [16]:
```
# Logistic regression
lr = LogisticRegression(max_iter=1000)
lr_gs = GridSearchCV(lr, lr_params, scoring='roc_auc', cv=5)
lr_gs.fit(X_train, y_train)
lr_preds = lr_gs.predict(X_test)

# Decision trees
dt = DecisionTreeClassifier()
dt_gs = GridSearchCV(dt, dt_params, scoring='roc_auc', cv=5)
dt_gs.fit(X_train, y_train)
dt_preds = dt_gs.predict(X_test)

# Naive Bayes
nb = GaussianNB()
nb_gs = GridSearchCV(nb, nb_params, scoring='roc_auc', cv=5)
nb_gs.fit(X_train, y_train)
nb_preds = nb_gs.predict(X_test)

# SVM
svm = SVC(probability=True,max_iter=1000)
svm_gs = GridSearchCV(svm, svm_params, scoring='roc_auc', cv=5)
svm_gs.fit(X_train, y_train)
svm_preds = svm_gs.predict(X_test)

# Neural Networks
nn = MLPClassifier(max_iter=1000)
nn_gs = GridSearchCV(nn, nn_params, scoring='roc_auc', cv=5)
nn_gs.fit(X_train, y_train)
nn_preds = nn_gs.predict(X_test)
```

Evaluation of algorithms

In [17]:
```
print("Logistic Regression Metrics:")
print("Accuracy:", accuracy_score(y_test, lr_preds))
print("Precision:", precision_score(y_test, lr_preds))
print("Recall:", recall_score(y_test, lr_preds))
print("F1 Score:", f1_score(y_test, lr_preds))
print("AUC Score:", roc_auc_score(y_test, lr_preds))
print("\n")

print("Decision Trees Metrics:")
print("Accuracy:", accuracy_score(y_test, dt_preds))
print("Precision:", precision_score(y_test, dt_preds))
print("Recall:", recall_score(y_test, dt_preds))
print("F1 Score:", f1_score(y_test, dt_preds))
print("AUC Score:", roc_auc_score(y_test, dt_preds))
print("\n")

print("Naive Bayes Metrics:")
print("Accuracy:", accuracy_score(y_test, nb_preds))
print("Precision:", precision_score(y_test, nb_preds))
print("Recall:", recall_score(y_test, nb_preds))
print("F1 Score:", f1_score(y_test, nb_preds))
print("AUC Score:", roc_auc_score(y_test, nb_preds))
print("\n")

print("SVM Metrics:")
print("Accuracy:", accuracy_score(y_test, svm_preds))
print("Precision:", precision_score(y_test, svm_preds))
print("Recall:", recall_score(y_test, svm_preds))
print("F1 Score:", f1_score(y_test, svm_preds))
print("AUC Score:", roc_auc_score(y_test, svm_preds))
print("\n")

print("Neural Networks Metrics:")
print("Accuracy:", accuracy_score(y_test, nn_preds))
print("Precision:", precision_score(y_test, nn_preds))
print("Recall:", recall_score(y_test, nn_preds))
print("F1 Score:", f1_score(y_test, nn_preds))
print("AUC Score:", roc_auc_score(y_test, nn_preds))
print("\n")
```

```
Logistic Regression Metrics:
Accuracy: 0.725
Precision: 0.0
Recall: 0.0
F1 Score: 0.0
AUC Score: 0.5


Decision Trees Metrics:
Accuracy: 0.685
Precision: 0.25
Recall: 0.07272727272727272
F1 Score: 0.11267605633802816
AUC Score: 0.4949843260188088


Naive Bayes Metrics:
Accuracy: 0.645
Precision: 0.23333333333333334
Recall: 0.12727272727272726
F1 Score: 0.16470588235294117
AUC Score: 0.4843260188087774


SVM Metrics:
Accuracy: 0.655
Precision: 0.25
Recall: 0.12727272727272726
F1 Score: 0.1686746987951807
AUC Score: 0.4912225705329153


Neural Networks Metrics:
Accuracy: 0.73
Precision: 1.0
Recall: 0.01818181818181818
F1 Score: 0.03571428571428572
AUC Score: 0.509090909090909
```

Ensemble Learning Modelling Process

Define hyperparameters to tune for each algorithm

In [18]:
```
bag_params = {
    'n_estimators': [10, 50, 100, 200],
    'max_samples': [0.5, 1.0],
    'max_features': [0.5, 1.0],
    'bootstrap': [True, False],
    'bootstrap_features': [True, False]
}


rf_params = {
    'n_estimators': [100, 200, 500],
    'max_depth': [None, 10, 20, 30],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'bootstrap': [True, False]
}


ada_params = {
    'n_estimators': [50, 100, 200],
    'learning_rate': [0.001, 0.01, 0.1, 1.0]
}


gb_params = {
    'n_estimators': [100, 200, 500],
    'learning_rate': [0.001, 0.01, 0.1, 1.0],
    'subsample': [0.5, 1.0],
    'max_depth': [3, 5, 10]
}


xgb_params = {
    'n_estimators': [100, 200, 500],
    'learning_rate': [0.001, 0.01, 0.1, 1.0],
    'subsample': [0.5, 1.0],
    'max_depth': [3, 5, 10],
    'colsample_bytree': [0.5, 1.0],
    'gamma': [0, 0.1, 0.2]
```

```
}
```

Train and Test the models

In [19]:
```
# Bagging
bag = BaggingClassifier()
bag_gs = GridSearchCV(bag, bag_params, scoring='roc_auc', cv=5)
bag_gs.fit(X_train, y_train)
bag_preds = bag_gs.predict(X_test)

# Random Forest
rf = RandomForestClassifier()
rf_gs = GridSearchCV(rf, rf_params, scoring='roc_auc', cv=5)
rf_gs.fit(X_train, y_train)
rf_preds = rf_gs.predict(X_test)

# AdaBoost
ada = AdaBoostClassifier()
ada_gs = GridSearchCV(ada, ada_params, scoring='roc_auc', cv=5)
ada_gs.fit(X_train, y_train)
ada_preds = ada_gs.predict(X_test)

# Gradient Boosting
gb = GradientBoostingClassifier()
gb_gs = GridSearchCV(gb, gb_params, scoring='roc_auc', cv=5)
gb_gs.fit(X_train, y_train)
gb_preds = gb_gs.predict(X_test)

# XGBoost
xgb = XGBClassifier()
xgb_gs = GridSearchCV(xgb, xgb_params, scoring='roc_auc', cv=5)
xgb_gs.fit(X_train, y_train)
xgb_preds = xgb_gs.predict(X_test)
```

Evaluation of algorithms

In [20]:
```
print("Bagging Metrics:")
print("Accuracy:", accuracy_score(y_test, bag_preds))
print("Precision:", precision_score(y_test, bag_preds))
print("Recall:", recall_score(y_test, bag_preds))
print("F1 Score:", f1_score(y_test, bag_preds))
print("AUC Score:", roc_auc_score(y_test, bag_preds))
print("\n")

print("Random Forest Metrics:")
print("Accuracy:", accuracy_score(y_test, rf_preds))
print("Precision:", precision_score(y_test, rf_preds))
print("Recall:", recall_score(y_test, rf_preds))
print("F1 Score:", f1_score(y_test, rf_preds))
print("AUC Score:", roc_auc_score(y_test, rf_preds))
print("\n")

print("AdaBoost Metrics:")
print("Accuracy:", accuracy_score(y_test, ada_preds))
print("Precision:", precision_score(y_test, ada_preds))
print("Recall:", recall_score(y_test, ada_preds))
print("F1 Score:", f1_score(y_test, ada_preds))
print("AUC Score:", roc_auc_score(y_test, ada_preds))
print("\n")

print("Gradient Boosting Metrics:")
print("Accuracy:", accuracy_score(y_test, gb_preds))
print("Precision:", precision_score(y_test, gb_preds))
print("Recall:", recall_score(y_test, gb_preds))
print("F1 Score:", f1_score(y_test, gb_preds))
print("AUC Score:", roc_auc_score(y_test, gb_preds))
print("\n")

print("XGBoost Metrics:")
print("Accuracy:", accuracy_score(y_test, xgb_preds))
print("Precision:", precision_score(y_test, xgb_preds))
print("Recall:", recall_score(y_test, xgb_preds))
print("F1 Score:", f1_score(y_test, xgb_preds))
print("AUC Score:", roc_auc_score(y_test, xgb_preds))
print("\n")
```

```
Bagging Metrics:
Accuracy: 0.695
Precision: 0.25
Recall: 0.05454545454545454
F1 Score: 0.08955223880597014
AUC Score: 0.49623824451410653


Random Forest Metrics:
Accuracy: 0.715
Precision: 0.0
Recall: 0.0
F1 Score: 0.0
AUC Score: 0.49310344827586206


AdaBoost Metrics:
Accuracy: 0.725
Precision: 0.0
Recall: 0.0
F1 Score: 0.0
AUC Score: 0.5


Gradient Boosting Metrics:
Accuracy: 0.725
Precision: 0.0
Recall: 0.0
F1 Score: 0.0
AUC Score: 0.5


XGBoost Metrics:
Accuracy: 0.68
Precision: 0.23529411764705882
Recall: 0.07272727272727272
F1 Score: 0.11111111111111113
AUC Score: 0.49153605015673985
```

Unsupervised Learning Process

In [38]:
```
# Apply K-Means clustering
kmeans = KMeans(n_clusters=2, random_state=42)
kmeans.fit(X_train)

# Predict the clusters for test data
clusters = kmeans.predict(X_test)

# Append clusters and actual fraud labels to the test dataset
test_df = pd.DataFrame(X_test)
test_df['cluster'] = clusters
test_df['isFraud'] = y_test.values

# Calculate fraud rates for each cluster
cluster_fraud_rates = test_df.groupby('cluster')['isFraud'].mean()

print("Fraud Rate for each cluster:\n")
print(cluster_fraud_rates)

print("Silhouette Score: ", silhouette_score(X_test, clusters))
```

```
Fraud Rate for each cluster:

cluster
0    0.333333
1    0.100000
Name: isFraud, dtype: float64
Silhouette Score:  0.15695094582860666
```