**Anomaly Detection Model Building and Evaluation**

# anomaly\_detection\_models.py

Import pandas as pd

Import numpy as np

Import matplotlib.pyplot as plt

Import seaborn as sns

From sklearn.ensemble import IsolationForest

From sklearn.svm import OneClassSVM

From sklearn.metrics import (classification\_report, confusion\_matrix,

Roc\_auc\_score, precision\_recall\_curve, average\_precision\_score)

From tensorflow.keras.models import Model

From tensorflow.keras.layers import Input, Dense, Dropout

From tensorflow.keras.optimizers import Adam

Import warnings

Warnings.filterwarnings(‘ignore’)

**Class AnomalyDetectionModels:**

Def \_\_init\_\_(self, preprocessed\_data):

Self.data = preprocessed\_data

Self.models = {}

Self.results = {}

Def train\_isolation\_forest(self, contamination=0.01, random\_state=42):

“””Train Isolation Forest model”””

Print(“Training Isolation Forest…”)

# **For Isolation Forest, we’ll use only normal transactions for training**

Normal\_mask = self.data[‘y\_train’] == 0

X\_train\_normal = self.data[‘X\_train\_scaled’][normal\_mask]

Iso\_forest = IsolationForest(

Contamination=contamination,

Random\_state=random\_state,

N\_estimators=100,

Max\_samples=’auto’

)

Iso\_forest.fit(X\_train\_normal)

Self.models[‘isolation\_forest’] = iso\_forest

# **Make predictions**

Train\_predictions = iso\_forest.predict(X\_train\_normal)

Test\_predictions = iso\_forest.predict(self.data[‘X\_test\_scaled’])

# **Convert predictions (1 for normal, -1 for anomaly) to binary (0 for normal, 1 for anomaly)**

Y\_pred\_train = (train\_predictions == -1).astype(int)

Y\_pred\_test = (test\_predictions == -1).astype(int)

# **For test set, we have true labels**

Y\_true\_test = self.data[‘y\_test’]

# **Calculate anomaly scores**

Anomaly\_scores\_train = iso\_forest.decision\_function(X\_train\_normal)

Anomaly\_scores\_test = iso\_forest.decision\_function(self.data[‘X\_test\_scaled’])

Self.results[‘isolation\_forest’] = {

‘y\_pred\_train’: y\_pred\_train,

‘y\_pred\_test’: y\_pred\_test,

‘y\_true\_test’: y\_true\_test,

‘anomaly\_scores\_test’: anomaly\_scores\_test

}

Return iso\_forest

Def train\_one\_class\_svm(self, nu=0.01, kernel=’rbf’, gamma=’scale’):

“””Train One-Class SVM model”””

Print(“Training One-Class SVM…”)

# **Use only normal transactions for trainin**g

Normal\_mask = self.data[‘y\_train’] == 0

X\_train\_normal = self.data[‘X\_train\_scaled’][normal\_mask]

Oc\_svm = OneClassSVM(

Nu=nu,

Kernel=kernel,

Gamma=gamma

)

Oc\_svm.fit(X\_train\_normal)

Self.models[‘one\_class\_svm’] = oc\_svm

# **Make predictions**

Train\_predictions = oc\_svm.predict(X\_train\_normal)

Test\_predictions = oc\_svm.predict(self.data[‘X\_test\_scaled’])

# **Convert predictions (1 for normal, -1 for anomaly) to binary (0 for normal, 1 for anomaly)**

Y\_pred\_train = (train\_predictions == -1).astype(int)

Y\_pred\_test = (test\_predictions == -1).astype(int)

Y\_true\_test = self.data[‘y\_test’]

# **Calculate decision function (anomaly scores)**

Anomaly\_scores\_test = oc\_svm.decision\_function(self.data[‘X\_test\_scaled’])

Self.results[‘one\_class\_svm’] = {

‘y\_pred\_train’: y\_pred\_train,

‘y\_pred\_test’: y\_pred\_test,

‘y\_true\_test’: y\_true\_test,

‘anomaly\_scores\_test’: anomaly\_scores\_test

}

Return oc\_svm

Def build\_autoencoder(self, input\_dim, encoding\_dim=16):

“””Build autoencoder model”””

# **Encoder**

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

Encoded = Dense(64, activation=’relu’)(input\_layer)

Encoded = Dropout(0.1)(encoded)

Encoded = Dense(32, activation=’relu’)(encoded)

Encoded = Dropout(0.1)(encoded)

Encoded = Dense(encoding\_dim, activation=’relu’)(encoded)

# **Decoder**

Decoded = Dense(32, activation=’relu’)(encoded)

Decoded = Dropout(0.1)(decoded)

Decoded = Dense(64, activation=’relu’)(decoded)

Decoded = Dropout(0.1)(decoded)

Decoded = Dense(input\_dim, activation=’sigmoid’)(decoded)

Autoencoder = Model(input\_layer, decoded)

Autoencoder.compile(optimizer=Adam(learning\_rate=0.001), loss=’mse’)

Return autoencoder

Def train\_autoencoder(self, epochs=50, batch\_size=32, validation\_split=0.1):

“””Train autoencoder model”””

Print(“Training Autoencoder…”)

# **Use only normal transactions for training**

Normal\_mask = self.data[‘y\_train’] == 0

X\_train\_normal = self.data[‘X\_train\_scaled’][normal\_mask]

Input\_dim = X\_train\_normal.shape[1]

Autoencoder = self.build\_autoencoder(input\_dim)

# **Train the autoencoder**

History = autoencoder.fit(

X\_train\_normal, X\_train\_normal,

Epochs=epochs,

Batch\_size=batch\_size,

Validation\_split=validation\_split,

Shuffle=True,

Verbose=1

)

Self.models[‘autoencoder’] = autoencoder

# **Calculate reconstruction error**

X\_test\_reconstructed = autoencoder.predict(self.data[‘X\_test\_scaled’])

Reconstruction\_error = np.mean(np.power(self.data[‘X\_test\_scaled’] – X\_test\_reconstructed, 2), axis=1)

# **Set threshold based on training reconstruction errors**

X\_train\_normal\_reconstructed = autoencoder.predict(X\_train\_normal)

Train\_reconstruction\_error = np.mean(np.power(X\_train\_normal – X\_train\_normal\_reconstructed, 2), axis=1)

Threshold = np.percentile(train\_reconstruction\_error, 95)

# **Make predictions based on threshold**

Y\_pred\_test = (reconstruction\_error > threshold).astype(int)

Y\_true\_test = self.data[‘y\_test’]

Self.results[‘autoencoder’] = {

‘y\_pred\_test’: y\_pred\_test,

‘y\_true\_test’: y\_true\_test,

‘anomaly\_scores\_test’: reconstruction\_error,

‘threshold’: threshold,

‘history’: history

}

Return autoencoder

Def evaluate\_model(self, model\_name):

“””Evaluate model performance”””

If model\_name not in self.results:

Raise ValueError(f”No results found for {model\_name}”)

Result = self.results[model\_name]

Y\_true = result[‘y\_true\_test’]

Y\_pred = result[‘y\_pred\_test’]

Print(f”\n=== {model\_name.upper()} EVALUATION ===”)

Print(classification\_report(y\_true, y\_pred, target\_names=[‘Legitimate’, ‘Fraud’]))

# **Calculate additional metrics**

Cm = confusion\_matrix(y\_true, y\_pred)

Auc\_score = roc\_auc\_score(y\_true, result[‘anomaly\_scores\_test’])

Ap\_score = average\_precision\_score(y\_true, result[‘anomaly\_scores\_test’])

Print(f”AUC Score: {auc\_score:.4f}”)

Print(f”Average Precision: {ap\_score:.4f}”)

# **Plot confusion matrix**

Plt.figure(figsize=(8, 6))

Sns.heatmap(cm, annot=True, fmt=’d’, cmap=’Blues’,

Xticklabels=[‘Predicted Legit’, ‘Predicted Fraud’],

Yticklabels=[‘Actual Legit’, ‘Actual Fraud’])

Plt.title(f’Confusion Matrix – {model\_name}’)

Plt.show()

# **Plot precision-recall curve**

Precision, recall, \_ = precision\_recall\_curve(y\_true, result[‘anomaly\_scores\_test’])

Plt.figure(figsize=(8, 6))

Plt.plot(recall, precision, marker=’.’)

Plt.xlabel(‘Recall’)

Plt.ylabel(‘Precision’)

Plt.title(f’Precision-Recall Curve – {model\_name}’)

Plt.grid(True)

Plt.show()

Return {

‘classification\_report’: classification\_report(y\_true, y\_pred, output\_dict=True),

‘auc\_score’: auc\_score,

‘ap\_score’: ap\_score,

‘confusion\_matrix’: cm

}

Def compare\_models(self):

“””Compare performance of all models”””

Comparison\_results = {}

For model\_name in self.models.keys():

If model\_name in self.results:

Eval\_results = self.evaluate\_model(model\_name)

Comparison\_results[model\_name] = {

‘precision’: eval\_results[‘classification\_report’][‘1’][‘precision’],

‘recall’: eval\_results[‘classification\_report’][‘1’][‘recall’],

‘f1\_score’: eval\_results[‘classification\_report’][‘1’][‘f1-score’],

‘auc\_score’: eval\_results[‘auc\_score’],

‘ap\_score’: eval\_results[‘ap\_score’]

}

# **Create comparison dataframe**

Comparison\_df = pd.DataFrame(comparison\_results).T

Comparison\_df = comparison\_df.round(4)

Print(“\n=== MODEL COMPARISON ===”)

Print(comparison\_df)

# **Plot comparison**

Metrics = [‘precision’, ‘recall’, ‘f1\_score’, ‘auc\_score’, ‘ap\_score’]

Comparison\_df[metrics].plot(kind=’bar’, figsize=(12, 8))

Plt.title(‘Model Performance Comparison’)

Plt.ylabel(‘Score’)

Plt.xticks(rotation=45)

Plt.legend(bbox\_to\_anchor=(1.05, 1), loc=’upper left’)

Plt.tight\_layout()

Plt.show()

Return comparison\_df

# **Main execution**

If \_\_name\_\_ == “\_\_main\_\_”:

# Assuming preprocessed\_data is available from EDA

# This would typically be imported from the EDA module

Pass