```
import yfinance as yf
import numpy as np
import pandas as pd
from sklearn.preprocessing import KBinsDiscretizer
import torch
import torch.nn as nn
from torch.utils.data import DataLoader, Dataset
import nltk
from nltk import Nonterminal, PCFG, ProbabilisticProduction
from nltk.parse import ViterbiParser
from tqdm import tqdm
import random
import warnings

warnings.filterwarnings("ignore")

nltk.download('punkt', quiet=True)


def get_stock_data(tickers, start, end):
    data_list = []
    for ticker in tickers:
        stock = yf.download(ticker, start=start, end=end)
        if stock.empty:
            print(f"Warning: No data fetched for ticker {ticker}.")
            continue
        stock['Return'] = stock['Adj Close'].pct_change() * 100
        stock = stock.dropna()
        stock = stock[['Return']]
        stock['Ticker'] = ticker
        data_list.append(stock)
    if not data_list:
        raise ValueError("No stock data fetched. Please check the ticker symbols and date range.")
    combined_data = pd.concat(data_list).reset_index()
    return combined_data


def bin_returns(stock_data, bins=[-20, -10, -5, 0, 5, 10, 20], labels=["D", "C-", "C", "B", "A", "A+"]):
    stock_data['State'] = pd.cut(stock_data['Return'], bins=bins, labels=labels, right=False)
    stock_data['State'] = stock_data['State'].fillna(labels[0])  # Assign lowest state for outliers
    return stock_data


class CFGParser:
    def __init__(self, initial_grammar_str):
        self.grammar = PCFG.fromstring(initial_grammar_str)
        self.parser = ViterbiParser(self.grammar)
        self.new_productions = []
        self.nonterminal_productions = {}
        self.next_nonterminal_index = 1
        self.rule_to_idx = {}

        for prod in self.grammar.productions():
            lhs = prod.lhs()
            if lhs not in self.nonterminal_productions:
                self.nonterminal_productions[lhs] = []
            self.nonterminal_productions[lhs].append(prod)

            rule_str = str(prod)
            if rule_str not in self.rule_to_idx:
                self.rule_to_idx[rule_str] = len(self.rule_to_idx)

    def parse(self, sentence):
        tokens = sentence.split()
        try:
            parsed_trees = list(self.parser.parse(tokens))
            print(f"Successfully parsed sentence: {sentence}")
            rules = [prod for tree in parsed_trees for prod in tree.productions()]
            rule_indices = []

            for rule in rules:
                rule_str = str(rule)
                if rule_str not in self.rule_to_idx:
                    self.rule_to_idx[rule_str] = len(self.rule_to_idx)
                rule_idx = self.rule_to_idx[rule_str]
                rule_indices.append(rule_idx)
```

```
        except ValueError:
            parsed_trees = []
            rule_indices = []

        # If parsing fails, learn new grammar
        if not parsed_trees:
            print(f"Parsing failed for sentence: {sentence}")
            self.learn_new_grammar(tokens)

        return parsed_trees, rule_indices

    def learn_new_grammar(self, tokens):
        print(f"Learning new grammar rules from tokens: {tokens}")
        # Enforce Chomsky Normal Form
        if len(tokens) > 2:
            for i in range(0, len(tokens) - 1, 2):
                new_nonterminal = Nonterminal(f"X{self.next_nonterminal_index}")
                self.next_nonterminal_index += 1
                rule_rhs = tokens[i:i+2] if i + 1 < len(tokens) else [tokens[i]]
                new_production = ProbabilisticProduction(new_nonterminal, rule_rhs, prob=1.0)
                self.new_productions.append(new_production)
                print(f"Created new production: {new_production}")
        else:
            new_nonterminal = Nonterminal(f"X{self.next_nonterminal_index}")
            self.next_nonterminal_index += 1
            new_production = ProbabilisticProduction(new_nonterminal, tokens, prob=1.0)
            self.new_productions.append(new_production)
            print(f"Created new production: {new_production}")

        self.genetic_algorithm()

        # Add new productions to nonterminal_productions and rule_to_idx
        for prod in self.new_productions:
            lhs = prod.lhs()
            if lhs not in self.nonterminal_productions:
                self.nonterminal_productions[lhs] = []
            self.nonterminal_productions[lhs].append(prod)

            rule_str = str(prod)
            if rule_str not in self.rule_to_idx:
                self.rule_to_idx[rule_str] = len(self.rule_to_idx)

        # Recalculate probabilities
        self._recalculate_probabilities()

        # Update the grammar with new rules
        all_productions = [prod for prods in self.nonterminal_productions.values() for prod in prods]
        self.grammar = PCFG(self.grammar.start(), all_productions)
        self.parser = ViterbiParser(self.grammar)

        self.new_productions = []

    def _recalculate_probabilities(self):
        for lhs, productions in self.nonterminal_productions.items():
            total_productions = len(productions)
            for i in range(total_productions):
                productions[i] = ProbabilisticProduction(lhs, productions[i].rhs(), prob=1.0 / total_productions)
        print("Recalculated probabilities for non-terminals.")

    def genetic_algorithm(self):
        if len(self.new_productions) < 2:
            print("Not enough productions for genetic algorithm.")
            return

        # Crossover: Pick two random productions and swap their right-hand sides
        parent1, parent2 = random.sample(self.new_productions, 2)
        if len(parent1.rhs()) == 2 and len(parent2.rhs()) == 2:
            print(f"Applying crossover between {parent1} and {parent2}")
            child1_rhs = [parent1.rhs()[0], parent2.rhs()[1]]
            child2_rhs = [parent2.rhs()[0], parent1.rhs()[1]]

            child1 = ProbabilisticProduction(parent1.lhs(), child1_rhs, prob=1.0)
            child2 = ProbabilisticProduction(parent2.lhs(), child2_rhs, prob=1.0)

            print(f"Created child productions: {child1}, {child2}")

            # Mutation: Randomly change one of the non-terminal symbols
```

```
            if random.random() < 0.2:
                mutated_rhs = [Nonterminal(f"X{random.randint(1, self.next_nonterminal_index)}"), parent1.rhs()[1]]
                mutated = ProbabilisticProduction(parent1.lhs(), mutated_rhs, prob=1.0)
                self.new_productions.append(mutated)
                print(f"Applied mutation to create: {mutated}")

            self.new_productions.extend([child1, child2])
        else:
            print("Productions not suitable for crossover.")

    def print_learned_grammar(self):
        """
        Prints all the learned grammar rules.
        """
        all_productions = [prod for prods in self.nonterminal_productions.values() for prod in prods]
        print("\nLearned CFG Grammar Rules:")
        for production in all_productions:
            print(production)


class StockSequenceDataset(Dataset):
    def __init__(self, data_sequences, rule_vector_size):
        """
        Initializes the dataset with input sequences, rule vectors, and labels.

        Args:
            data_sequences (list): List of tuples (input_seq, rule_vector, label).
            rule_vector_size (int): Size of the grammar rule vector.
        """
        self.data_sequences = data_sequences
        self.rule_vector_size = rule_vector_size

    def __len__(self):
        return len(self.data_sequences)

    def __getitem__(self, idx):
        input_seq, rule_vector, label = self.data_sequences[idx]
        input_seq = torch.tensor(input_seq, dtype=torch.long)
        rule_vector = torch.tensor(rule_vector, dtype=torch.float32)
        label = torch.tensor(label, dtype=torch.long)
        return input_seq, rule_vector, label


# Transformer Model Definition
class TransformerModel(nn.Module):
    def __init__(self, vocab_size, rule_vector_size, d_model=128, nhead=8, num_layers=3, num_classes=6):
        super(TransformerModel, self).__init__()
        self.embedding = nn.Embedding(vocab_size, d_model)
        self.transformer_layer = nn.TransformerEncoderLayer(d_model=d_model, nhead=nhead)
        self.transformer = nn.TransformerEncoder(self.transformer_layer, num_layers=num_layers)
        self.fc = nn.Linear(d_model + rule_vector_size, num_classes)
    def forward(self, src, rule_vector):
        src = self.embedding(src)  # (batch_size, seq_len, d_model)
        src = src.permute(1, 0, 2)  # (seq_len, batch_size, d_model)
        src = self.transformer(src)  # (seq_len, batch_size, d_model)
        src = src.mean(dim=0)  # (batch_size, d_model)
        combined = torch.cat((src, rule_vector), dim=1)  # (batch_size, d_model + rule_vector_size)
        out = self.fc(combined)  # (batch_size, num_classes)
        return out


def train_transformer(train_data, val_data, vocab_size, rule_vector_size, num_classes, epochs=10, lr=0.001):
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model = TransformerModel(vocab_size, rule_vector_size, num_classes=num_classes).to(device)
    criterion = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=lr)
    train_dataloader = DataLoader(train_data, batch_size=32, shuffle=True)
    val_dataloader = DataLoader(val_data, batch_size=32, shuffle=False)

    for epoch in range(epochs):
        running_loss = 0.0
        model.train()
        for i, (inputs, rule_vectors, labels) in enumerate(tqdm(train_dataloader, desc=f"Epoch {epoch+1}/{epochs}")):
            inputs, rule_vectors, labels = inputs.to(device), rule_vectors.to(device), labels.to(device)
            optimizer.zero_grad()
            outputs = model(inputs, rule_vectors)
            loss = criterion(outputs, labels)
            loss.backward()
```

```
            optimizer.step()
            running_loss += loss.item()

            if (i + 1) % 100 == 0:
                avg_loss = running_loss / 100
                print(f"[Epoch {epoch+1}, Batch {i+1}] Loss: {avg_loss:.4f}")
                running_loss = 0.0

        val_accuracy = evaluate_model(model, val_data)
        print(f"Epoch {epoch+1}, Validation Accuracy: {val_accuracy:.2f}%")

    return model

def evaluate_model(model, data):
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    dataloader = DataLoader(data, batch_size=32, shuffle=False)
    model.eval()
    correct = 0
    total = 0

    with torch.no_grad():
        for inputs, rule_vectors, labels in dataloader:
            inputs, rule_vectors, labels = inputs.to(device), rule_vectors.to(device), labels.to(device)
            outputs = model(inputs, rule_vectors)
            _, predicted = torch.max(outputs, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    accuracy = (correct / total) * 100
    return accuracy


def main():
    # Initial grammar
    initial_grammar_str = """
    S -> A B [1.0]
    A -> 'A+' [1.0]
    B -> C D [1.0]
    C -> 'C-' [1.0]
    D -> 'C' [1.0]
    """

    parser = CFGParser(initial_grammar_str)

    tickers = ["AAPL"]  # ["AAPL", "MSFT", "NVDA", "AMZN"]
    start_date = "2021-01-01"
    end_date = "2024-01-01"

    print("Fetching stock data...")
    stock_data = get_stock_data(tickers, start=start_date, end=end_date)
    binned_data = bin_returns(stock_data)

    state_to_idx = {"D": 0, "C-": 1, "C": 2, "B": 3, "A": 4, "A+": 5}
    idx_to_state = {v: k for k, v in state_to_idx.items()}

    seq_len = 10
    data_sequences = []

    print("Generating input sequences...")
    for ticker in tickers:
        ticker_data = binned_data[binned_data['Ticker'] == ticker]
        state_sequence = [state_to_idx[state] for state in ticker_data['State']]

        for i in range(len(state_sequence) - seq_len):
            input_seq = state_sequence[i:i+seq_len]
            label = state_sequence[i+seq_len]
            data_sequences.append((input_seq, label))

    print("Shuffling data sequences...")
    random.shuffle(data_sequences)

    # Parse sequences and collect rule indices
    print("Starting unsupervised learning with CFGParser...")
    parsed_data_sequences = []
    for (input_seq, label) in tqdm(data_sequences, desc="Parsing sequences"):
        state_chunk = ' '.join([idx_to_state[state] for state in input_seq])
        parsed_trees, rule_indices = parser.parse(state_chunk)
```

```
            parsed_data_sequences.append((input_seq, rule_indices, label))

    # Determine the size of the rule vector
    rule_vector_size = len(parser.rule_to_idx)
    print(f"\nTotal unique grammar rules: {rule_vector_size}")

    # Convert rule indices to frequency vectors
    print("Generating grammar rule frequency vectors...")
    final_data_sequences = []
    for (input_seq, rule_indices, label) in parsed_data_sequences:
        rule_vector = np.zeros(rule_vector_size, dtype=np.float32)
        for idx in rule_indices:
            rule_vector[idx] += 1.0
        final_data_sequences.append((input_seq, rule_vector, label))

    print("Splitting data into train, validation, and test sets...")
    train_size = int(0.7 * len(final_data_sequences))
    val_size = int(0.15 * len(final_data_sequences))
    test_size = len(final_data_sequences) - train_size - val_size

    train_data_sequences = final_data_sequences[:train_size]
    val_data_sequences = final_data_sequences[train_size:train_size + val_size]
    test_data_sequences = final_data_sequences[train_size + val_size:]

    print("Creating datasets...")
    train_dataset = StockSequenceDataset(train_data_sequences, rule_vector_size)
    val_dataset = StockSequenceDataset(val_data_sequences, rule_vector_size)
    test_dataset = StockSequenceDataset(test_data_sequences, rule_vector_size)

    vocab_size = len(state_to_idx)
    num_classes = len(state_to_idx)
    print("\nStarting training of Transformer model...")
    model = train_transformer(train_dataset, val_dataset, vocab_size, rule_vector_size, num_classes, epochs=10)

    print("\nEvaluating model on training data...")
    train_accuracy = evaluate_model(model, train_dataset)
    print(f"Training Accuracy: {train_accuracy:.2f}%")

    print("\nEvaluating model on validation data...")
    val_accuracy = evaluate_model(model, val_dataset)
    print(f"Validation Accuracy: {val_accuracy:.2f}%")

    print("\nEvaluating model on test data...")
    test_accuracy = evaluate_model(model, test_dataset)
    print(f"Test Accuracy: {test_accuracy:.2f}%")

    # parser.print_learned_grammar()

if __name__ == "__main__":
    main()
```

```
[*********************100%***********************]  1 of 1 completed
Fetching stock data...
Generating input sequences...
Shuffling data sequences...
Starting unsupervised learning with CFGParser...
Parsing sequences:   1%|          | 7/742 [00:00<00:10, 67.84it/s]Parsing failed for sentence: B C C C C B C B B C
Learning new grammar rules from tokens: ['B', 'C', 'C', 'C', 'C', 'B', 'C', 'B', 'B', 'C']
Created new production: X1 -> 'B' 'C' [1.0]
Created new production: X2 -> 'C' 'C' [1.0]
Created new production: X3 -> 'C' 'B' [1.0]
Created new production: X4 -> 'C' 'B' [1.0]
Created new production: X5 -> 'B' 'C' [1.0]
Applying crossover between X1 -> 'B' 'C' [1.0] and X4 -> 'C' 'B' [1.0]
Created child productions: X1 -> 'B' 'B' [1.0], X4 -> 'C' 'C' [1.0]
Recalculated probabilities for non-terminals.
Successfully parsed sentence: B B C C C B B C B B
Parsing failed for sentence: B B C C C B B C B B
Learning new grammar rules from tokens: ['B', 'B', 'C', 'C', 'C', 'B', 'B', 'C', 'B', 'B']
Created new production: X6 -> 'B' 'B' [1.0]
Created new production: X7 -> 'C' 'C' [1.0]
Created new production: X8 -> 'C' 'B' [1.0]
Created new production: X9 -> 'B' 'C' [1.0]
Created new production: X10 -> 'B' 'B' [1.0]
Applying crossover between X10 -> 'B' 'B' [1.0] and X7 -> 'C' 'C' [1.0]
Created child productions: X10 -> 'B' 'C' [1.0], X7 -> 'C' 'B' [1.0]
Applied mutation to create: X10 -> X4 'B' [1.0]
Recalculated probabilities for non-terminals.
Successfully parsed sentence: C B C B B B C B B B
Parsing failed for sentence: C B C B B B C B B B
```

```
Learning new grammar rules from tokens: ['C', 'B', 'C', 'B', 'B', 'B', 'C', 'B', 'B', 'B']
Created new production: X11 -> 'C' 'B' [1.0]
Created new production: X12 -> 'C' 'B' [1.0]
Created new production: X13 -> 'B' 'B' [1.0]
Created new production: X14 -> 'C' 'B' [1.0]
Created new production: X15 -> 'B' 'B' [1.0]
Applying crossover between X13 -> 'B' 'B' [1.0] and X14 -> 'C' 'B' [1.0]
Created child productions: X13 -> 'B' 'B' [1.0], X14 -> 'C' 'B' [1.0]
Recalculated probabilities for non-terminals.
Successfully parsed sentence: C B B C B C B C C B
Parsing failed for sentence: C B B C B C B C C B
Learning new grammar rules from tokens: ['C', 'B', 'B', 'C', 'B', 'C', 'B', 'C', 'C', 'B']
Created new production: X16 -> 'C' 'B' [1.0]
Created new production: X17 -> 'B' 'C' [1.0]
Created new production: X18 -> 'B' 'C' [1.0]
Created new production: X19 -> 'B' 'C' [1.0]
Created new production: X20 -> 'C' 'B' [1.0]
Applying crossover between X17 -> 'B' 'C' [1.0] and X16 -> 'C' 'B' [1.0]
Created child productions: X17 -> 'B' 'B' [1.0], X16 -> 'C' 'C' [1.0]
Recalculated probabilities for non-terminals.
Successfully parsed sentence: B B B C C C B B C B
Parsing failed for sentence: B B B C C C B B C B
Learning new grammar rules from tokens: ['B', 'B', 'B', 'C', 'C', 'C', 'B', 'B', 'C', 'B']
Created new production: X21 -> 'B' 'B' [1.0]
Created new production: X22 -> 'B' 'C' [1.0]
Created new production: X23 -> 'C' 'C' [1.0]
Created new production: X24 -> 'B' 'B' [1.0]
Created new production: X25 -> 'C' 'B' [1.0]
Applying crossover between X25 -> 'C' 'B' [1.0] and X24 -> 'B' 'B' [1.0]
```