Common Task 1: Dataset preprocessing

Description: Use **Sympy or Mathematica** to generate datasets of functions with their Taylor expansions up the fourth order. Tokenize the dataset.

```
import sympy as sp

x = sp.symbols('x')

functions = [sp.sin(x), sp.exp(x), sp.ln(1 + x), sp.cos(x), sp.tan(x)]

taylor_data = {}

for func in functions:
    taylor_series = sp.series(func, x, 0, 5).removeO()
    taylor_data[str(func)] = str(taylor_series)

for func, taylor in taylor_data.items():
    print(f"Function: {func}\nTaylor Expansion: {taylor}\n")
```

```
Function: sin(x)
Taylor Expansion: -x**3/6 + x

Function: exp(x)
Taylor Expansion: x**4/24 + x**3/6 + x**2/2 + x + 1

Function: log(x + 1)
Taylor Expansion: -x**4/4 + x**3/3 - x**2/2 + x

Function: cos(x)
Taylor Expansion: x**4/24 - x**2/2 + 1

Function: tan(x)
Taylor Expansion: x**3/3 + x
```

Common Task 2: Use LSTM model

Please train an **LSTM model** to learn the Taylor expansion of each function. You can use a deep learning algorithm of your choice (in Keras/TF or Pytorch).

```
import re
from tensorflow.keras.preprocessing.text import Tokenizer

def tokenize(expression):
    tokens = re.findall(r'\d+|\w+|[+\-*/^()]', expression)
    return tokens

tokenized_data = {func: tokenize(taylor) for func, taylor in taylor_data.items()}

text_sequences = [' '.join(tokens) for tokens in tokenized_data.values()]

tokenizer = Tokenizer(filters='', lower=False)
tokenizer.fit_on_texts(text_sequences)

sequences = tokenizer.texts_to_sequences(text_sequences)
```

```
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
import numpy as np

max_seq_length = max(len(seq) for seq in sequences)
padded_sequences = pad_sequences(sequences, maxlen=max_seq_length, padding='post')

# X = np.array([seq[:-1] for seq in padded_sequences.values()])
# y = np.array([seq[1:] for seq in padded_sequences.values()])
X = np.array([seq[:-1] for seq in padded_sequences])
y = np.array([seq[1:] for seq in padded_sequences])
```

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Embedding, Bidirectional

vocab_size = len(tokenizer.word_index) + 1
```

```
model = Sequential([
    Embedding(input_dim=vocab_size, output_dim=50, input_length=max_seq_length - 1),
    Bidirectional(LSTM(100, return_sequences=True)),
    Bidirectional(LSTM(100, return_sequences=True)),
    Dense(vocab_size, activation='softmax')
])

model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
model.fit(X, y, epochs=250, verbose=1)
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 141ms/step - accuracy: 0.9913 - loss: 0.1378
Epoch 223/250
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 117ms/step - accuracy: 1.0000 - loss: 0.1391
Epoch 224/250
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 104ms/step - accuracy: 1.0000 - loss: 0.1485
Epoch 225/250
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 117ms/step - accuracy: 0.9913 - loss: 0.1377
Epoch 226/250
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 138ms/step - accuracy: 1.0000 - loss: 0.1264
Epoch 227/250
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 138ms/step - accuracy: 1.0000 - loss: 0.1212
Epoch 228/250
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 71ms/step - accuracy: 1.0000 - loss: 0.1167
Epoch 229/250
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 64ms/step - accuracy: 1.0000 - loss: 0.1145
Epoch 230/250
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 66ms/step - accuracy: 1.0000 - loss: 0.1130
Epoch 231/250
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 142ms/step - accuracy: 1.0000 - loss: 0.1089
Epoch 232/250
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 146ms/step - accuracy: 1.0000 - loss: 0.1052
Epoch 233/250
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 63ms/step - accuracy: 1.0000 - loss: 0.1028
Epoch 234/250
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 64ms/step - accuracy: 1.0000 - loss: 0.1003
Epoch 235/250
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 142ms/step - accuracy: 1.0000 - loss: 0.0991
Epoch 236/250
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 67ms/step - accuracy: 1.0000 - loss: 0.0964
Epoch 237/250
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 68ms/step - accuracy: 1.0000 - loss: 0.0934
Epoch 238/250
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 67ms/step - accuracy: 1.0000 - loss: 0.0922
Epoch 239/250
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 139ms/step - accuracy: 1.0000 - loss: 0.0913
Epoch 240/250
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 69ms/step - accuracy: 1.0000 - loss: 0.0886
Epoch 241/250
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 66ms/step - accuracy: 1.0000 - loss: 0.0855
Epoch 242/250
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 65ms/step - accuracy: 1.0000 - loss: 0.0843
Epoch 243/250
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 65ms/step - accuracy: 1.0000 - loss: 0.0837
Epoch 244/250
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 82ms/step - accuracy: 1.0000 - loss: 0.0826
Epoch 245/250
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 66ms/step - accuracy: 1.0000 - loss: 0.0812
Epoch 246/250
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 70ms/step - accuracy: 1.0000 - loss: 0.0797
Epoch 247/250
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 139ms/step - accuracy: 1.0000 - loss: 0.0782
Epoch 248/250
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 62ms/step - accuracy: 1.0000 - loss: 0.0770
Epoch 249/250
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 140ms/step - accuracy: 1.0000 - loss: 0.0759
Epoch 250/250
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 61ms/step - accuracy: 1.0000 - loss: 0.0747
<keras.src.callbacks.history.History at 0x7b64f836ead0>
```

```
def evaluate_sequence_accuracy(model, X, y, tokenizer):
    total_tokens = 0
    correct_tokens = 0

    preds = model.predict(X, verbose=0)
    pred_tokens = np.argmax(preds, axis=-1)

    for true_seq, pred_seq in zip(y, pred_tokens):
        for true_token, pred_token in zip(true_seq, pred_seq):
            if true_token != 0:  # skip padding
                total_tokens += 1
                if true_token == pred_token:
                    correct_tokens += 1

    accuracy = (correct_tokens / total_tokens) * 100 if total_tokens > 0 else 0
    return accuracy
```

```
accuracy = evaluate_sequence_accuracy(model, X, y, tokenizer)
print(f"Sequence Prediction Accuracy: {accuracy:.2f}%")
```

```
Sequence Prediction Accuracy: 100.00%
```

```
def print_predicted_sequences(model, X, tokenizer):
    preds = model.predict(X, verbose=0)
    pred_token_ids = np.argmax(preds, axis=-1)

    for i, pred_ids in enumerate(pred_token_ids):
        pred_tokens = [tokenizer.index_word.get(id, '') for id in pred_ids]
        true_tokens = [tokenizer.index_word.get(id, '') for id in y[i]]

        print(f"Example {i + 1}")
        print(f"Predicted: {' '.join(pred_tokens)}")
        print(f"Actual   : {' '.join(true_tokens)}")
        print("=" * 50)

print_predicted_sequences(model, X, tokenizer)
```

```
Example 1
Predicted: x * * 3 / 6 + x
Actual   : x * * 3 / 6 + x
==================================================
Example 2
Predicted: * * 4 / 24 + x * * 3 / 6 + x * * 2 / 2 + x + 1
Actual   : * * 4 / 24 + x * * 3 / 6 + x * * 2 / 2 + x + 1
==================================================
Example 3
Predicted: x * * 4 / 4 + x * * 3 / 3 - x * * 2 / 2 + x
Actual   : x * * 4 / 4 + x * * 3 / 3 - x * * 2 / 2 + x
==================================================
Example 4
Predicted: * * 4 / 24 - x * * 2 / 2 + 1
Actual   : * * 4 / 24 - x * * 2 / 2 + 1
==================================================
Example 5
Predicted: * * 3 / 3 + x
Actual   : * * 3 / 3 + x
==================================================
```

Specific Task 3: Use Transformer model

Please train a **Transformer** model to learn the Taylor expansion of each function.

```
import torch
from torch.utils.data import DataLoader, TensorDataset
input_sequences = [torch.tensor(seq[:-1]) for seq in padded_sequences]
target_sequences = [torch.tensor(seq[1:]) for seq in padded_sequences]

max_seq_length = max(len(seq) for seq in input_sequences)

input_tensors = []
target_tensors = []
attention_masks = []

for inp_seq, tgt_seq in zip(input_sequences, target_sequences):
    inp_padded = torch.cat([inp_seq, torch.zeros(max_seq_length - len(inp_seq), dtype=torch.long)])
    tgt_padded = torch.cat([tgt_seq, torch.zeros(max_seq_length - len(tgt_seq), dtype=torch.long)])

    attention_mask = (inp_padded != 0).long()

    input_tensors.append(inp_padded)
    target_tensors.append(tgt_padded)
    attention_masks.append(attention_mask)

input_tensors = torch.stack(input_tensors)
target_tensors = torch.stack(target_tensors)
attention_masks = torch.stack(attention_masks)

dataset = TensorDataset(input_tensors, target_tensors, attention_masks)
dataloader = DataLoader(dataset, batch_size=16, shuffle=True)
```

```
import torch.nn as nn
import torch.optim as optim
```

```
class TransformerModel(nn.Module):
    def __init__(self, vocab_size, d_model=64, nhead=8, num_layers=6, dim_feedforward=512, max_seq_length=50):
        super(TransformerModel, self).__init__()

        self.embedding = nn.Embedding(vocab_size, d_model)
        self.positional_encoding = nn.Parameter(torch.randn(max_seq_length, d_model))

        self.transformer = nn.Transformer(
            d_model=d_model,
            nhead=nhead,
            num_encoder_layers=num_layers,
            num_decoder_layers=num_layers,
            dim_feedforward=dim_feedforward,
            batch_first=True
        )

        self.fc_out = nn.Linear(d_model, vocab_size)

    def forward(self, src, tgt, src_mask=None, tgt_mask=None, src_key_padding_mask=None, tgt_key_padding_mask=None):
        src = self.embedding(src) + self.positional_encoding[:src.size(1), :]
        tgt = self.embedding(tgt) + self.positional_encoding[:tgt.size(1), :]

        output = self.transformer(
            src, tgt,
            src_mask=src_mask, tgt_mask=tgt_mask,
            src_key_padding_mask=src_key_padding_mask,
            tgt_key_padding_mask=tgt_key_padding_mask
        )

        return self.fc_out(output)

vocab_size = len(tokenizer.word_index) + 1
model = TransformerModel(vocab_size)

criterion = nn.CrossEntropyLoss(ignore_index=0)  # Ignore padding tokens
optimizer = optim.Adam(model.parameters(), lr=0.001)



device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)

def train(model, dataloader, criterion, optimizer, num_epochs=150):
    model.train()

    for epoch in range(num_epochs):
        total_loss = 0

        for src, tgt, attn_mask in dataloader:
            src, tgt, attn_mask = src.to(device), tgt.to(device), attn_mask.to(device)

            optimizer.zero_grad()
            output = model(src, tgt[:, :-1])
            loss = criterion(output.view(-1, vocab_size), tgt[:, 1:].reshape(-1))

            loss.backward()
            optimizer.step()

            total_loss += loss.item()

        print(f"Epoch {epoch+1}, Loss: {total_loss / len(dataloader)}")

train(model, dataloader, criterion, optimizer)
```

```
Epoch 113, Loss: 0.005122999195009⁄47
Epoch 114, Loss: 0.004965719301253557
Epoch 115, Loss: 0.005089770536869764
Epoch 116, Loss: 0.005083614960312843
Epoch 117, Loss: 0.004948973190039396
Epoch 118, Loss: 0.005054616369307041
Epoch 119, Loss: 0.004841612186282873
Epoch 120, Loss: 0.004669151734560728
Epoch 121, Loss: 0.004905519541352987
Epoch 122, Loss: 0.004921883810311556
Epoch 123, Loss: 0.004759583622217178
Epoch 124, Loss: 0.004804203752428293
Epoch 125, Loss: 0.004638882353901863
Epoch 126, Loss: 0.004638850223273039
Epoch 127, Loss: 0.004574175458401442
Epoch 128, Loss: 0.004691298119723797
Epoch 129, Loss: 0.004608858376741409
Epoch 130, Loss: 0.004472014959901571
Epoch 131, Loss: 0.004635822027921677
Epoch 132, Loss: 0.004636678844690323
Epoch 133, Loss: 0.004519424866884947
Epoch 134, Loss: 0.004539265763014555
Epoch 135, Loss: 0.004607728682458401
Epoch 136, Loss: 0.0043059466406702995
Epoch 137, Loss: 0.004562489688396454
Epoch 138, Loss: 0.004520062357187271
Epoch 139, Loss: 0.004501312971115112
Epoch 140, Loss: 0.0045350356958806515
Epoch 141, Loss: 0.0044078766368329525
Epoch 142, Loss: 0.004653407726436853
Epoch 143, Loss: 0.004484421573579311
Epoch 144, Loss: 0.004209138453006744
Epoch 145, Loss: 0.0044922237284481525
Epoch 146, Loss: 0.004319129977375269
Epoch 147, Loss: 0.004250429105013609
Epoch 148, Loss: 0.004228577483445406
Epoch 149, Loss: 0.004135802388191223
Epoch 150, Loss: 0.004082707688212395
```

```
def evaluate_sequence_accuracy(model, dataloader, tokenizer):
    model.eval()
    total_tokens = 0
    correct_tokens = 0

    with torch.no_grad():
        for src, tgt, attn_mask in dataloader:
            src, tgt = src.to(device), tgt.to(device)
            output = model(src, tgt[:, :-1])
            predictions = output.argmax(dim=-1)

            true_seq = tgt[:, 1:]
            mask = true_seq != 0
            correct = (predictions == true_seq) & mask
            correct_tokens += correct.sum().item()
            total_tokens += mask.sum().item()

    accuracy = (correct_tokens / total_tokens) * 100 if total_tokens > 0 else 0
    return accuracy

accuracy = evaluate_sequence_accuracy(model, dataloader, tokenizer)
print(f"\n Sequence Prediction Accuracy: {accuracy:.2f}%")
```

```
Sequence Prediction Accuracy: 100.00%
```

Start coding or generate with AI.

Start coding or generate with AI.