Milestone4: Model Training, Evaluation and Testing

Introduction

This report outlines the progress made on Milestone 4:

Goal: model training, evaluation and testing

Minimum expectation	Tool suggestion
Integration of the training and evaluation code in the MLOps platform	ZenML, TFX trainer
Development of behavioural model tests	Pytest
Optional: Energy efficiency measurement	Carbon code

Training Step

The training step involves integrating the Huggingface transformers library to import the model and retrain it based on the available data using PyTorch. The process includes initializing a TweetsDataset class for data handling, loading the model and tokenizer, training the model with AdamW optimizer, and saving the trained model and tokenizer.

TweetsDataset class:

```python
from transformers import AutoModelForSequenceClassification, AutoTokenizer
import torch
from torch.utils.data import Dataset, DataLoader
from torch.optim import AdamW
from tqdm import tqdm

class TweetsDataset(Dataset):
    def __init__(self, texts, labels, tokenizer, max_length=512):
        self.tokenizer = tokenizer
        self.texts = texts
        self.labels = labels
        self.max_length = max_length

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

    def __getitem__(self, idx):
        text = self.texts[idx]
        labels = self.labels[idx]
        encoding = self.tokenizer(text, padding='max_length', max_length=self.max_length, truncation=True, return_attention_mask=True)
        return {
            'input_ids': torch.tensor(encoding['input_ids']),
            'attention_mask': torch.tensor(encoding['attention_mask']),
            'labels': torch.tensor(labels)
        }
```

Training step:

```python
def train_model_step(texts: list, labels: list) -> str:
    device = torch.device("cpu")
    task = 'sentiment'
    MODEL = f"cardiffnlp/twitter-roberta-base-{task}"
    tokenizer = AutoTokenizer.from_pretrained(MODEL)
    model = AutoModelForSequenceClassification.from_pretrained(MODEL, num_labels=len(label_mapping)).to(device)

    dataset = TweetsDataset(texts, labels, tokenizer)
    train_loader = DataLoader(dataset, batch_size=16, shuffle=True)

    optimizer = AdamW(model.parameters(), lr=5e-5)

    model.train()
    for epoch in range(2):  # loop over the dataset multiple times
        total_loss = 0.0
        for batch in tqdm(train_loader, desc=f"Training Epoch {epoch + 1}"):
            batch = {k: v.to(device) for k, v in batch.items()}
            outputs = model(**batch)

            loss = outputs.loss
            total_loss += loss.item()

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

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

    # Save the trained model
    model_path = "trained_sentiment_model"
```

Evaluation Step

The evaluation step assesses the model's performance using accuracy, precision, recall, and F1 score metrics. Predictions are mapped to their corresponding labels before calculating these metrics, since the predictions are strings (positive, negative, neutral) to int (0,1,2). The results are logged using MLflow, demonstrating the model's effectiveness in sentiment analysis along with prior experiments.

Evaluation code:

```python
@step
def evaluate_predictions(predictions: list, true_labels: list) -> dict:
    predictions_mapped = [label_mapping[pred] for pred in predictions]

    accuracy = accuracy_score(true_labels, predictions_mapped)
    precision = precision_score(true_labels, predictions_mapped, average='weighted', zero_division=0)
    recall = recall_score(true_labels, predictions_mapped, average='weighted', zero_division=0)
    f1 = f1_score(true_labels, predictions_mapped, average='weighted', zero_division=0)

    mlflow.log_metric("accuracy", accuracy)
    mlflow.log_metric("precision", precision)
    mlflow.log_metric("recall", recall)
    mlflow.log_metric("f1_score", f1)

    return {"accuracy": accuracy, "precision": precision, "recall": recall, "f1": f1}
```

Behavioral model testing using Pytest:

Behavioral model testing is conducted using Pytest to ensure the reliability of the data pipeline. Tests include checking the functionality of reading labels and tweets from files, evaluating predictions, inserting preprocessed tweets into a database, and the preprocess and model inference steps. These tests validate the integrity and accuracy of the pipeline processes.

Read labels and tweets tests: Data Tests.

This function checks a series of assertions: if the file is a list of labels, if every label is an integer, then checks if the labels are not empty.

For the second function, it asserts that the tweets are in a list format and that each tweet is of string type.

```python
#Data tests: Input feature code is tested.
def test_read_labels_from_file():
    labels = read_labels_from_file('../val_labels.txt')
    assert isinstance(labels, list)
    assert all(isinstance(label, int) for label in labels)
    assert len(labels) > 0

def test_read_tweets_from_file():
    tweets = read_tweets_from_file('../val_text.txt')
    assert isinstance(tweets, list)
    assert all(isinstance(tweet, str) for tweet in tweets)
```

Integration testing with external feature stores: ML Infrastructure Tests

- **@patch('pipeline.Cluster.connect'):** to avoid making real connections to a Cassandra database during testing.
- Uses only mock variables.
- Asserts that the Cassandra execution was called as many times as there are rows to make sure that all data has been logged successfully.

```python
#ML Infrastructure tests: Testing integration with external feature stores.
@patch('pipeline.Cluster.connect')
def test_insert_preprocessed_tweets_into_cassandra(mock_connect):
    mock_session = MagicMock()
    mock_connect.return_value = mock_session
    processed_texts = ["Test tweet 1 processed", "Test tweet 2 processed"]
    insert_preprocessed_tweets_into_cassandra(processed_texts)
    assert mock_session.execute.call_count == len(processed_texts)
```

Preprocessing, evaluation, and model inference unit testing: Unit Tests

- **For the preprocessing function,** uses mock tweets with their desired expected output which is given manually as a variable then asserts that the actual output given by the function is identical to the expected output given manually.
- **For the model inference step,** it asserts that the predictions are a list and that they all are one of 3 words (positive, negative, neutral) and that there are as many predictions as there are tweets.
- **For the model evaluation step:**
    - **@patch('pipeline.mlflow.log_metric'):** replaces the MLFlow functions with mocks functions to not require external dependencies or the MLFlow server to be running in the testing phase to further focus on the functionality.
    - Then asserts that the output of the function is in key value format for the metrics.
    - It also asserts that the keys are accuracy, precision, recall and f1-score.
    - Finally, it asserts that all the values of the metrics are floats.

```python
#Unit tests:
@patch('pipeline.mlflow.log_metric')
def test_evaluate_predictions(mock_log):
    predictions = ['positive', 'negative', 'neutral']
    true_labels = [2, 0, 1]
    results = evaluate_predictions(predictions, true_labels)
    assert isinstance(results, dict)
    assert set(results.keys()) == {"accuracy", "precision", "recall", "f1"}
    assert all(isinstance(value, float) for value in results.values())

def test_preprocess_step():
    test_tweets = [
        "@user1 this is a test! http://testurl.com",
        "Normal text without users or urls"
    ]
    expected_output = [
        "@user this is a test! http",
        "Normal text without users or urls"
    ]
    preprocessed = preprocess_step(test_tweets)
    assert preprocessed == expected_output

def test_model_inference_step():
    test_tweets = [
        "This is a positive tweet",
        "This is a negative tweet",
        "This is a neutral tweet"
    ]
    predictions = model_inference_step(test_tweets)
    assert isinstance(predictions, list)
    assert all(prediction in ["positive", "negative", "neutral"] for prediction in predictions)
    assert len(predictions) == len(test_tweets)
```

Checking predictions quality: Monitoring Tests.

Simply compares the metrics of the present model which are logged into MLFlow with the given thresholds, if the metrics are lower outputs an alert.

```python
#Monitoring tests: Prediction quality has not regressed.
THRESHOLDS = {
    "accuracy": 0.8,
    "precision": 0.75,
    "recall": 0.8,
    "f1": 0.75
}

def fetch_latest_metrics(experiment_name):
    client = MlflowClient()
    experiment = client.get_experiment_by_name(experiment_name)
    if not experiment:
        return "Experiment not found."

    runs = client.search_runs(experiment_ids=[experiment.experiment_id], order_by=["start_time DESC"], max_results=1)
    if not runs:
        return "No runs found."

    return runs[0].data.metrics

def check_metrics_against_thresholds(metrics):
    alerts = []
    for metric, threshold in THRESHOLDS.items():
        if metric in metrics and metrics[metric] < threshold:
            alerts.append(f"Alert: {metric} dropped below threshold. Value: {metrics[metric]}, Threshold: {threshold}")
    return alerts

experiment_name = "Default"
metrics = fetch_latest_metrics(experiment_name)
if isinstance(metrics, str):
    print(metrics)
    alerts = check_metrics_against_thresholds(metrics)
    for alert in alerts:
        print(alert)
```

```
Metric accuracy is within the threshold
Metric precision is within the threshold
Metric recall is within the threshold
Metric f1 is within the threshold
```