Arabic Name Segmentation with T5

A fine-tuned T5 model for automatically segmenting Arabic names written in Latin script (without spaces) into properly formatted names with spaces.

📋 Overview

This project trains a T5-small model to segment concatenated Arabic names into individual name components.

Example:

```
Input:  mohamedmustafamahmoud
Output: Mohamed Mustafa Mahmoud
```

🎯 Use Cases

- Processing forms where names were entered without spaces
- Cleaning messy name databases
- Preprocessing for NER (Named Entity Recognition) systems
- Data normalization pipelines

📊 Requirements

```bash
pip install transformers torch pandas scikit-learn datasets
```

Minimum Requirements:

- Python 3.8+
- 4GB RAM (CPU inference)
- 8GB RAM (training)
- 2GB disk space for model files

🚀 Quick Start

1. Prepare Your Data

Create a CSV file (`names.csv`) with two columns:

```csv
input,target
mohamedmustafamahmoud,Mohamed Mustafa Mahmoud
ahmedhassanali,Ahmed Hassan Ali
abdulrahmanyoussef,Abdul Rahman Youssef
```

Or use Python dictionary:

```python
data = {
    'input': ['mohamedali', 'hassanomar', ...],
    'target': ['Mohamed Ali', 'Hassan Omar', ...]
}
```

2. Train the Model

Run the training script:

```python
python train_segmenter.py
```

Training Time:

- 1000 samples: ~20-30 minutes (CPU) / ~5-8 minutes (GPU)
- Disk space needed: ~1.2GB

Expected Performance:

- Training accuracy: 90-95%
- Validation accuracy: 85-92%
- Works best with 500+ training examples

3. Model Output

After training completes, you'll find:

- `./t5-name-segmenter-final/` - Final trained model
- Training logs and checkpoints in `./t5-name-segmenter/`

📖 Usage Guide

Loading the Trained Model

python

```python
from transformers import T5Tokenizer, T5ForConditionalGeneration

# Load model
tokenizer = T5Tokenizer.from_pretrained('./t5-name-segmenter-final')
model = T5ForConditionalGeneration.from_pretrained('./t5-name-segmenter-final')

print("✓ Model loaded successfully!")
```

Basic Inference

python

```python
def segment_name(name):
    """Segment a concatenated Arabic name"""
    # Prepare input
    input_text = f"segment arabic name: {name}"
    inputs = tokenizer(input_text, return_tensors="pt", max_length=128, truncation=True)

    # Generate segmentation
    outputs = model.generate(
        **inputs,
        max_length=128,
        num_beams=4,
        early_stopping=True
    )

    # Decode result
    result = tokenizer.decode(outputs[0], skip_special_tokens=True)
    return result

# Test it
print(segment_name('mohamedaliahmed'))
# Output: Mohamed Ali Ahmed
```

Batch Inference

python

```python
def segment_names_batch(names, batch_size=16):
    """Process multiple names efficiently"""
    results = []

    for i in range(0, len(names), batch_size):
        batch = names[i:i+batch_size]
        inputs_text = [f"segment arabic name: {name}" for name in batch]

        inputs = tokenizer(
            inputs_text,
            return_tensors="pt",
            padding=True,
            truncation=True,
            max_length=128
        )

        outputs = model.generate(
            **inputs,
            max_length=128,
            num_beams=4,
            early_stopping=True
        )

        batch_results = tokenizer.batch_decode(outputs, skip_special_tokens=True)
        results.extend(batch_results)

    return results

# Process multiple names
names = ['mohamedali', 'hassanomar', 'khaledahmed']
segmented = segment_names_batch(names)
for original, result in zip(names, segmented):
    print(f"{original} → {result}")
```

⚡ Performance & Inference Time

CPU Inference (Normal laptop/desktop):

- Single name: ~0.3-0.8 seconds
- Batch of 10: ~2-4 seconds
- Batch of 100: ~15-25 seconds

GPU Inference (T4/V100):

- Single name: ~0.05-0.1 seconds
- Batch of 10: ~0.2-0.4 seconds
- Batch of 100: ~2-4 seconds

Tips for Faster Inference:

- Use batch processing for multiple names
- Set `num_beams=1` for greedy decoding (faster but less accurate)
- Use GPU when available
- Enable FP16 on GPU: `model.half()`

Optimized Fast Inference

```python
```

```python
import torch

# Enable optimizations
model.eval()  # Set to evaluation mode
if torch.cuda.is_available():
    model = model.cuda()
    model = model.half()  # FP16 for faster inference

def segment_name_fast(name):
    """Faster inference with fewer beams"""
    input_text = f"segment arabic name: {name}"
    inputs = tokenizer(input_text, return_tensors="pt")

    if torch.cuda.is_available():
        inputs = {k: v.cuda() for k, v in inputs.items()}

    with torch.no_grad():  # Disable gradient calculation
        outputs = model.generate(
            **inputs,
            max_length=128,
            num_beams=1,  # Greedy decoding (faster)
            do_sample=False
        )

    return tokenizer.decode(outputs[0], skip_special_tokens=True)
```

🔄 Retraining on New Data

Update Training Data

```python
```

```python
# Load your new dataset
import pandas as pd

df = pd.read_csv('new_names.csv')
new_data = {
    'input': df['input'].tolist(),
    'target': df['target'].tolist()
}

# In the training script, replace the data dictionary with new_data
```

Fine-tune Existing Model

```python
# Load your previously trained model
model = T5ForConditionalGeneration.from_pretrained('./t5-name-segmenter-final')
tokenizer = T5Tokenizer.from_pretrained('./t5-name-segmenter-final')

# Continue training with new data
# Use lower learning rate for fine-tuning
training_args = Seq2SeqTrainingArguments(
    output_dir="./t5-name-segmenter-v2",
    learning_rate=1e-5,  # Lower learning rate
    num_train_epochs=10,  # Fewer epochs
    # ... other args
)
```

Incremental Learning

```python
# Combine old and new data
combined_data = {
    'input': old_inputs + new_inputs,
    'target': old_targets + new_targets
}

# Train on combined dataset
# This prevents catastrophic forgetting
```

🔗 Integration into Larger Pipelines

Flask API Example

python

```python
from flask import Flask, request, jsonify
from transformers import T5Tokenizer, T5ForConditionalGeneration

app = Flask(__name__)

# Load model once at startup
tokenizer = T5Tokenizer.from_pretrained('./t5-name-segmenter-final')
model = T5ForConditionalGeneration.from_pretrained('./t5-name-segmenter-final')
model.eval()

@app.route('/segment', methods=['POST'])
def segment_endpoint():
    data = request.json
    name = data.get('name', '')

    if not name:
        return jsonify({'error': 'No name provided'}), 400

    # Segment name
    input_text = f"segment arabic name: {name}"
    inputs = tokenizer(input_text, return_tensors="pt")
    outputs = model.generate(**inputs, max_length=128, num_beams=4)
    result = tokenizer.decode(outputs[0], skip_special_tokens=True)

    return jsonify({'original': name, 'segmented': result})

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

Test the API:

bash

```bash
curl -X POST http://localhost:5000/segment \
  -H "Content-Type: application/json" \
  -d '{"name": "mohamedaliahmed"}'
```

Pandas DataFrame Processing

python

```python
import pandas as pd

# Load data
df = pd.read_csv('unsegmented_names.csv')

# Apply segmentation
df['segmented_name'] = df['unsegmented_name'].apply(
    lambda x: segment_name(x) if pd.notna(x) else None
)

# Save results
df.to_csv('segmented_names.csv', index=False)
```

Apache Spark Integration (Big Data)

python

```python
from pyspark.sql import SparkSession
from pyspark.sql.functions import udf
from pyspark.sql.types import StringType

spark = SparkSession.builder.appName("NameSegmentation").getOrCreate()

# Broadcast model to all workers
broadcast_model = spark.sparkContext.broadcast(model)
broadcast_tokenizer = spark.sparkContext.broadcast(tokenizer)

# Create UDF
@udf(returnType=StringType())
def segment_name_udf(name):
    if not name:
        return None
    model = broadcast_model.value
    tokenizer = broadcast_tokenizer.value
    # ... segmentation logic
    return result

# Process large dataset
df = spark.read.csv('large_names.csv', header=True)
df = df.withColumn('segmented', segment_name_udf('unsegmented_name'))
df.write.csv('output_segmented.csv')
```

Streaming Pipeline Example

python

```python
import time
from queue import Queue
from threading import Thread

# Create processing queue
input_queue = Queue()
output_queue = Queue()

def process_worker():
    """Background worker for continuous processing"""
    while True:
        name = input_queue.get()
        if name is None:  # Poison pill to stop
            break

        result = segment_name(name)
        output_queue.put((name, result))
        input_queue.task_done()

# Start worker
worker = Thread(target=process_worker, daemon=True)
worker.start()

# Add names to queue
for name in ['mohamedali', 'hassanomar', 'khaledahmed']:
    input_queue.put(name)

# Get results
while not output_queue.empty():
    original, segmented = output_queue.get()
    print(f"{original} → {segmented}")
```

🐛 Troubleshooting

Common Issues

1. Out of Memory during training

```
python
```

```
# Reduce batch size
per_device_train_batch_size=4  # Instead of 8

# Enable gradient checkpointing
model.gradient_checkpointing_enable()
```

2. Slow inference

python

```
# Use greedy decoding instead of beam search
outputs = model.generate(**inputs, num_beams=1)

# Or use batch processing
results = segment_names_batch(names, batch_size=32)
```

3. Poor accuracy

- Ensure training data quality (correct segmentations)
- Add more training examples (aim for 1000+)
- Increase training epochs
- Check for consistent name formatting

📈 Model Evaluation

Calculate Accuracy

python

```
def evaluate_model(test_inputs, test_targets):
    """Evaluate model on test set"""
    correct = 0
    total = len(test_inputs)

    for inp, target in zip(test_inputs, test_targets):
        predicted = segment_name(inp)
        if predicted.strip().lower() == target.strip().lower():
            correct += 1

    accuracy = correct / total
    print(f"Accuracy: {accuracy*100:.2f}% ({correct}/{total})")
    return accuracy

# Test on validation set
accuracy = evaluate_model(val_inputs, val_targets)
```

📝 License

MIT License - Feel free to use and modify for your projects.

🤝 Contributing

Contributions welcome! Please ensure your training data respects privacy and data protection regulations.

⚠️ Important Notes

- **Privacy**: Ensure you have rights to use the name data
- **Bias**: Model performance depends on training data diversity
- **Language**: Optimized for Arabic names in Latin script
- **Adaptation**: Can be retrained for other languages/scripts

---

📞 Support

For issues or questions:

1. Check the troubleshooting section
2. Review the training logs for errors

3. Ensure all dependencies are installed correctly

Model Version: 1.0

Last Updated: October 2025