DSGA 1011: Assignment 4

Georgios Ioannou
gi2100

Part 1. Q1.

Please provide a link to your github repository, which contains the code for both Part I and Part II.

GitHub Repository: https://github.com/GeorgiosIoannouCoder/ds_ga_1011_fall_2025_assignment_4
The repository contains:

- Part 1 code: `hw4-code/part-1-code/main.py` and `utils.py`
- Part 2 code: `hw4-code/part-2-code/train_t5.py`, `load_data.py`, `t5_utils.py`, and related files
- All output files for submission

Part 1. Q2.

Describe your transformation of dataset.

My transformation implements **synonym replacement** using WordNet to create out-of-distribution test examples. The transformation works as follows:

1. **Tokenization**: The input text is tokenized into individual words using NLTK's `word_tokenize` function.
2. **Word Selection**: For each tokenized word, I check if it is alphabetic (excluding punctuation). If so, with a probability of 0.3 (30%), I attempt to replace it with a synonym.
3. **Synonym Retrieval**: For each selected word, I query WordNet using `wordnet.synsets(word.lower())` to retrieve all synsets (sets of synonymous words) containing the word. I then iterate through all synsets and extract all lemmas (synonym words) using `lemma.name()`, converting underscores to spaces and lowercasing the result.
4. **Synonym Filtering**: I filter out synonyms that are identical to the original word (case-insensitive) and remove duplicates.

5. **Replacement**: If at least one valid synonym is found, I randomly select one using `random.choice()`. I preserve the original capitalization: if the original word was capitalized (first letter uppercase), I capitalize the replacement synonym as well.

6. **Text Reconstruction**: If no synonym is found, or if the word was not selected for replacement, the original word is kept. Finally, I reconstruct the text using NLTK's `TreebankWordDetokenizer` to properly restore spacing and punctuation.

Why this transformation is reasonable: This transformation is reasonable because it simulates a realistic scenario where users might express the same sentiment using different vocabulary. For example, a user might say "This movie is excellent" while another says "This film is outstanding" - both convey positive sentiment but use different word choices. Synonym replacement maintains semantic meaning while introducing lexical variation that could occur naturally in user-generated text. This is particularly relevant for sentiment analysis tasks where the core sentiment should remain unchanged despite word choice variations. The 30% replacement probability ensures that the text remains mostly coherent while introducing sufficient variation to test model robustness.

Evaluation Results: When evaluating the fine-tuned BERT model on the transformed test set, I observed the following performance:

- Original test accuracy: 93.116%
- Transformed test accuracy: 87.204%
- Accuracy decrease: 5.912%

This decrease of more than 4 points indicates that the transformation successfully creates a challenging out-of-distribution test set.

Part 1. Q3.

Report & Analysis

Accuracy Values:

- **Augmented model on original test data:** 92.936%
- **Augmented model on transformed test data:** 90.460%

For comparison, the original model (trained without augmentation) achieved:

- **Original model on original test data:** 93.116%
- **Original model on transformed test data:** 87.204%

Analysis & Discussion: (1) Did the model's performance on the transformed test data improve after applying data augmentation?

Yes, the model's performance on the transformed test data improved significantly. The accuracy increased from 87.204% (original model) to 90.460% (augmented model), representing an improvement of 3.256 percentage points. This demonstrates that data augmentation successfully helps the model generalize better to out-of-distribution examples that use synonym variations.

(2) How did data augmentation affect the model's performance on the original test data?

Data augmentation resulted in a slight decrease in performance on the original test data. The accuracy decreased from 93.116% (original model) to 92.936% (augmented model), a decrease of 0.180 percentage points. While this is a minor reduction, it suggests a slight trade-off between robustness to transformations and performance on the original distribution.

Intuitive Explanation: The observed results can be explained by the bias-variance trade-off and distribution shift. By augmenting the training data with 5,000 transformed examples, the model learns to be more robust to lexical variations (synonym replacements), which directly improves performance on the transformed test set. However, this comes at a small cost: the model may slightly overfit to the augmented patterns or allocate some capacity to learning transformation-invariant features, which may not be as beneficial for the original test distribution. The augmented training set introduces a different distribution (with synonym variations), and while this helps with OOD generalization, it may slightly shift the model's decision boundaries away from the optimal point for the original distribution.

Limitation of Data Augmentation Approach: One key limitation of this data augmentation approach is that it only addresses a specific type of distribution shift (synonym replacement). Real-world out-of-distribution scenarios may involve many other types of variations, such as:

- Different writing styles or formality levels
- Domain-specific terminology
- Grammatical variations or sentence structure changes
- Cultural or regional language differences
- Typographical errors or noise patterns different from my transformation

Training on synonym-replaced examples only prepares the model for that specific transformation pattern. If the test distribution shifts in a different way (e.g., different sentence structures, new vocabulary, or different types of noise), the augmented model may not generalize well. This highlights the challenge of creating comprehensive data augmentation strategies that cover all possible distribution shifts.

Part 2. Q4.

Statistics Name	Train	Dev
Number of examples	4225	466
Mean sentence length	17.10	17.07
Mean SQL query length	216.37	210.05
Vocabulary size (natural language)	791	465
Vocabulary size (SQL)	555	395

Table 1: Data statistics before any pre-processing. Statistics are computed using the T5 tokenizer (`google-t5/t5-small`) on the raw natural language queries and SQL queries from the dataset files.

Statistics Name	Train	Dev
T5 fine-tuned model		
Mean sentence length	17.10	17.07
Mean SQL query length	216.37	210.05
Vocabulary size (natural language)	791	465
Vocabulary size (SQL)	555	395

Table 2: Data statistics after pre-processing. I use the T5 tokenizer directly without additional preprocessing steps (no normalization, lowercasing, or special formatting), so the statistics are identical to Table 1. The tokenizer handles tokenization, special tokens, and padding during batch collation.

Part 2. Q5.

Design choice	Description
Data processing	Minimal data processing was performed. Natural language queries and SQL queries were loaded directly from `{split}.nl` and `{split}.sql` files respectively. No additional preprocessing steps such as normalization, lowercasing, or special formatting were applied. The data was processed on-the-fly during dataset initialization as pairs of (natural_language_query, sql_query) tuples. For the test set, SQL queries were set to None since ground truth is not available. Dynamic padding was applied during batch collation using `pad_sequence` with `PAD_IDX=0`.
Tokenization	I used the default T5 tokenizer (`T5TokenizerFast` from `google-t5/t5-small`) for both encoder and decoder inputs. **Encoder:** Natural language queries were tokenized using `tokenizer(nl_query, padding=False, truncation=True, max_length=512)`, producing `input_ids` and `attention_mask`. **Decoder:** SQL queries were tokenized similarly. For training (teacher forcing), the decoder input was created by prepending the `pad_token_id` (which serves as the beginning-of-sequence token) to the SQL token sequence: `decoder_input_ids = [pad_token_id] + sql_token_ids`. The decoder targets were created by appending the `eos_token_id`: `decoder_targets = sql_token_ids + [eos_token_id]`. This ensures proper alignment for teacher forcing during training.
Architecture	I fine-tuned the entire T5-small model end-to-end using `T5ForConditionalGeneration.from_pretrained('google-t5/t5-small')`. **Fine-tuning approach:** Full fine-tuning - all model parameters were trainable (no layers were frozen). **Model components:** The entire encoder-decoder architecture was fine-tuned, including all 6 encoder layers, all 6 decoder layers, embedding layers, attention mechanisms, feed-forward networks, and layer normalization layers. **Parameter selection for optimization:** The optimizer was configured with two parameter groups: (1) LayerNorm parameters and non-bias parameters with `weight_decay=0`, and (2) bias parameters and non-LayerNorm parameters with `weight_decay=0.0`. All parameters had `requires_grad=True` by default.
Hyperparameters	**Learning rate:** 1e-4 (0.0001). **Batch size:** 16 (training), 16 (evaluation). **Optimizer:** AdamW with epsilon=1e-8, betas=(0.9, 0.999), weight_decay=0. **Scheduler:** None (no learning rate scheduling). **Stopping criteria:** Maximum 10 epochs with early stopping patience of 5 epochs based on Record F1 score on development set. Training stops if Record F1 does not improve for 5 consecutive epochs. Best model checkpoint is saved based on highest Record F1 score. **Generation:** Greedy decoding (`num_beams=1`), `max_length=512`, early stopping enabled. **Loss:** Cross-entropy loss computed only on non-padding tokens (padding tokens masked out). **Device:** CUDA (GPU), full precision training.

Table 3: Details of the baseline T5 fine-tuned model configuration.

Baseline Model Configuration:

Design choice	Description
Data processing	Minimal data processing was performed. Natural language queries and SQL queries were loaded directly from `{split}.nl` and `{split}.sql` files respectively. No additional preprocessing steps such as normalization, lowercasing, or special formatting were applied. The data was processed on-the-fly during dataset initialization as pairs of (natural_language_query, sql_query) tuples. For the test set, SQL queries were set to None since ground truth is not available. Dynamic padding was applied during batch collation using `pad_sequence` with `PAD_IDX=0`.
Tokenization	I used the default T5 tokenizer (`T5TokenizerFast` from `google-t5/t5-small`) for both encoder and decoder inputs. **Encoder:** Natural language queries were tokenized using `tokenizer(nl_query, padding=False, truncation=True, max_length=512)`, producing `input_ids` and `attention_mask`. **Decoder:** SQL queries were tokenized similarly. For training (teacher forcing), the decoder input was created by prepending the `pad_token_id` (which serves as the beginning-of-sequence token) to the SQL token sequence: `decoder_input_ids = [pad_token_id] + sql_token_ids`. The decoder targets were created by appending the `eos_token_id`: `decoder_targets = sql_token_ids + [eos_token_id]`. This ensures proper alignment for teacher forcing during training.
Architecture	I fine-tuned the entire T5-small model end-to-end using `T5ForConditionalGeneration.from_pretrained('google-t5/t5-small')`. **Fine-tuning approach:** Full fine-tuning - all model parameters were trainable (no layers were frozen). **Model components:** The entire encoder-decoder architecture was fine-tuned, including all 6 encoder layers, all 6 decoder layers, embedding layers, attention mechanisms, feed-forward networks, and layer normalization layers. **Parameter selection for optimization:** The optimizer was configured with two parameter groups: (1) LayerNorm parameters and non-bias parameters with `weight_decay=0`, and (2) bias parameters and non-LayerNorm parameters with `weight_decay=0.0`. All parameters had `requires_grad=True` by default.
Hyperparameters	**Learning rate:** 3e-4 (0.0003). **Batch size:** 8 (training), 16 (evaluation). **Optimizer:** AdamW with epsilon=1e-8, betas=(0.9, 0.999), weight_decay=0. **Scheduler:** Cosine annealing with warmup, 1 warmup epoch. **Stopping criteria:** Maximum 10 epochs with early stopping patience of 3 epochs based on Record F1 score on development set. Training stops if Record F1 does not improve for 3 consecutive epochs. Best model checkpoint is saved based on highest Record F1 score. **Generation:** Greedy decoding (`num_beams=1`), `max_length=512`, early stopping enabled. **Loss:** Cross-entropy loss computed only on non-padding tokens (padding tokens masked out). **Device:** CUDA (GPU), full precision training.

Table 4: Details of the best-performing T5 model configuration (fine-tuned). Key differences from baseline: higher learning rate (3e-4 vs 1e-4), smaller training batch size (8 vs 16), and cosine annealing scheduler with warmup.

Best-Performing Model Configuration (ft_experiment):

Part 2. Q6.

System	Query EM	F1 score
Dev Results		
T5 fine-tuned (Baseline)		
Default setup (lr=1e-4, batch=16, no scheduler)	0.86	40.38
T5 fine-tuned (Best/Experimental)		
Full model (lr=3e-4, batch=8, cosine scheduler, early stopping)	0.86	44.42
Test Results		
T5 fine-tuning (Best/Experimental)	Leaderboard	Leaderboard

Table 5: Development and test results. Note: Test results will be computed upon submission to the leaderboard. The baseline model (learning rate 1e-4, batch size 16, no scheduler) achieves Query EM of 0.86% and Record F1 of 40.38% on the dev set. The experimental model (best configuration with learning rate 3e-4, batch size 8, cosine scheduler with 1 warmup epoch, and early stopping with patience 3) shows Query EM of 0.86% and Record F1 score of 44.42% on the dev set, representing an improvement of 4.04 percentage points in F1 score.

Quantitative Results:

Qualitative Error Analysis: I analyze errors for both the baseline and best-performing (ft_experiment) models. Both models exhibit similar error types, but the best-performing model shows significantly fewer errors overall, particularly in semantic understanding.

Error Type 1: SQL Syntax Errors Example Of Error:
Baseline Model: Natural Language Query: "list all arrivals from any airport to baltimore on thursday morning arriving before 9am". Predicted SQL: "SELECT DISTINCT flight_1.flight_id FROM flight flight_1, airport_service airport_service_1, city city_1...". Error: "OperationalError: near ")": syntax error".

Best Model: Similar syntax errors but with lower frequency.

Error Description: Both models generate SQL queries with syntax errors that prevent execution. Common issues include malformed WHERE clauses, incorrect date/time formatting, missing parentheses, or invalid SQL operators. These errors occur when the model fails to properly structure SQL syntax despite understanding the query intent. The best model shows improved syntax generation due to better training with cosine scheduler and smaller batch size.

Statistics: Baseline: 175/466 (37.55%). **Best Model:** 141/466 (30.26%).

Error Type 2: Wrong Column/Table Name Errors Example Of Error:

Baseline Model: Natural Language Query: “please list all flights between boston and san francisco nonstop”. Predicted SQL: “SELECT DISTINCT flight_1.flight_id FROM flight flight_1, airport_service airport_service_1, city city_1...”. Error: “OperationalError: no such column: miles_1.stops”.

Best Model: Similar column/table errors.

Error Description: Both models reference columns or tables that do not exist in the database schema. This occurs when the model hallucinates column names, uses incorrect table aliases, or fails to correctly map natural language concepts to the actual database schema. The model may repeat column names or use non-existent attributes.

Statistics: Baseline: 32/466 (6.87%). **Best Model:** 32/466 (6.87%).

Error Type 3: Semantic Errors (Wrong Results) Example Of Error:

Baseline Model: Natural Language Query: “what flights are available tomorrow from denver to philadelphia”. Ground Truth: Returns 23 flight records. Predicted: Returns 26 flight records (incorrect filtering).

Best Model: Natural Language Query: “i’d like to fly from philadelphia to san francisco through dallas”. Ground Truth: Returns 4 flight records. Predicted: Returns 34 flight records (missing multi-hop logic).

Error Description: Both models generate syntactically correct SQL queries that execute successfully, but return incorrect results. These errors include missing or incorrect WHERE conditions, wrong JOIN logic, incorrect filtering criteria, or improper handling of multi-hop flights. The queries run without errors but fail to capture the semantic intent of the natural language query. The best model shows better semantic understanding with fewer semantic errors.

Statistics: Baseline: 160/466 (34.33%). **Best Model:** 179/466 (38.41%).

Part 2. Q7.

The model checkpoint used to generate the submitted test outputs is available at:

Google Drive Link: https://drive.google.com/drive/folders/1MZFUQgTB5Wp3AknozotFNBwo9SqP

The checkpoint contains the best-performing T5 fine-tuned model (231 MB) based on Record F1 score on the development set. This model was trained with the following configuration:

- Learning rate: 3e-4
- Batch size: 8 (training), 16 (evaluation)
- Scheduler: Cosine annealing with 1 warmup epoch
- Early stopping: 3 epochs patience
- Best dev Record F1: 44.42%

The checkpoint folder contains:

- `config.json`: Model configuration file
- `generation_config.json`: Generation parameters
- `model.safetensors`: Model weights (230.8 MB)