Lab: Training models

Hakyung Sung
University of Hawaii at Manoa

November 18, 2026

Assignment 4

- Example tagger (just “follow” the code) - 20%
- Your tagger - 60%
- Generative tagger (zero-shot prompting) - 20%

Section 1. Train an example tagger

Steps

- Prepare dataset
- Install packages
- Ready for training
- Training
- Evaluation

These steps also go for building your tagger

1. Prepare dataset

Token classification: predict a label for each token.

```
And|O we|O packed|INTRAN_RES up|O .|O

After|O all|O this|O problems|O
I|O became|INTRAN_RES
dissappointed|O .|O
```

The second sentence is wrapped here for readability. In the file, each sentence occupies one line.

How is the data organized?

- Each entry follows `word|label`.
- Annotated verbs receive an ASC label.
- Other tokens receive `O`.

1. Prepare dataset: Download the example data

Download and extract `asc-simple.zip`.

```
asc-simple/
    train.txt
    dev.txt
    test.txt
```

Each file

- `train.txt`: train the model.
- `dev.txt`: evaluate during development and select the model.
- `test.txt`: evaluate the final model on held-out data.

1. Prepare dataset: Your own

- Choose 2–4 linguistic labels for words.
  - Me: ASC types (9 labels: TRAN_S, INTRAN_S)
  - Discourse markers: `CONTRAST`, `CAUSE`, `ADDITION`
  - Should be intuitive!
- Write a definition for each label and apply it consistently (where to tag).
- Use capital letters and underscores: `LABEL_NAME`.
- Label the relevant **words**, not whole sentences. Unmarked words automatically receive `O`.

1. Prepare dataset: Your own

- Aim for 100 sentences (minimum: 50). Write your own or use sources you have permission to use.
- Write into a .txt file.
- Include at least 15–20 instances of each label across varied contexts.
- Keep sentences short to medium length; avoid near-identical examples.
- Exclude private information about real people.
- The script splits the data into training, development, and test sets in an 8:1:1 ratio. (100 sentences $\rightarrow$ 80 training, 10 development, 10 test)

1. Prepare dataset: Your own

```
Maria [moved|PAST] to Seoul .
We [can|MODAL] meet tomorrow .
They [went|PAST] home .
She [has to|MODAL] finish .
The weather is nice today .
```

USE THIS FORMAT

- Format: `[word|LABEL]`.
- Unmarked words become `O`.
- In `[has to|MODAL]`, both words receive `MODAL`.
- Unlabeled sentences are allowed, but most should contain a target.

Save as `sentences.txt`: plain text, UTF-8, one sentence per line.

1. Prepare dataset: Your own

- The code in *Step 4. Split the data with a script* will help you to have all three files (`train.txt, dev.txt, test.txt`) for training.

- Check `train.txt` file after you convert your annotated file (`sentences.txt`).

Tip: Before you run the code - CHECK!

- Check that your labels match your definitions and use consistent spelling.
- Keep one sentence per line, with no blank lines in the middle.
- Use `[`, `]`, and `|` only for annotation.
- Separate punctuation with spaces (e.g., `today .`), or let the script handle it.
- Confirm that the file is saved as `sentences.txt` with UTF-8 encoding.

Section 2. Install packages

2. Install the packages

In your terminal:

```
pip install torch transformers datasets accelerate scikit-learn
```

In Google Colab:

```
!pip install torch transformers datasets accelerate scikit-learn
```

Packages for...

- `transformers`: load and train pretrained models
- `datasets`: organize data
- `torch`: run the model
- `accelerate`: support training
- `scikit-learn`: evaluate performance

Section 3. Ready for training

3. Prepare the training files

Set up the dataset and Python script.

```
Assignment4 (working directory)/
  train.txt
  dev.txt
  test.txt
  train_tagger.py
```

In short, create `train_tagger.py` in the same folder (or working directory) and add the Python code in the following slides.

3. Read the datasets

Convert each sentence into tokens and labels.

```
from pathlib import Path

def read_dataset(file_path):
    rows = []

    for line_number, line in enumerate(
        Path(file_path).read_text(
            encoding="utf-8-sig"
        ).splitlines(),
        start=1,
    ):
        line = line.strip()
        if not line:
            continue

        tokens = []
        tags = []
...
```

What does this function do?

- Reads one sentence per line.
- Splits each `word|label` entry.
- Stores words and labels in separate lists.
- Checks for invalid annotations.

3. Load and inspect the datasets

Read the three files and check their contents.

```
train_rows = read_dataset("train.txt")
dev_rows = read_dataset("dev.txt")
test_rows = read_dataset("test.txt")
```

Check dataset sizes:

```
print("Training sentences:", len(train_rows))
print("Development sentences:", len(dev_rows))
print("Test sentences:", len(test_rows))
```

Inspect the first example:

```
print(train_rows[0])
```

What should we check?

- Are all three files loaded?
- How many sentences are in each split?
- Does each token have a corresponding label?

3. Create the label mappings

Convert linguistic labels into numeric IDs.

```
label_list = sorted({
    tag
    for row in train_rows
    for tag in row["tags"]
})

print(label_list)

label2id = {
    label: index
    for index, label in enumerate(label_list)
}
id2label = {
    index: label
    for label, index in label2id.items()
}
print(label2id)
```

Why do we need this?

- Models predict numbers rather than label names.
- `label2id`: label $\rightarrow$ ID
- `id2label`: ID $\rightarrow$ label

3. Validate and organize the datasets

Check label consistency and preserve the data splits.

```
for split_name, split_rows in [
    ("dev", dev_rows),
    ("test", test_rows),
]:
    unknown_labels = {
        tag
        for row in split_rows
        for tag in row["tags"]
        if tag not in label2id
    }

    if unknown_labels:
        raise ValueError(
            f"{split_name} contains labels "
            f"absent from training: "
            f"{sorted(unknown_labels)}"
        )
```

Why these steps?

- Check that the development and test sets use labels included in the training label inventory.

*If an unknown label is found, check annotation consistency. Do not move dev/test examples into training (this is cheating)!

3. Validate and organize the datasets

Check label consistency and preserve the data splits.

```
from datasets import Dataset, DatasetDict

dataset = DatasetDict({
    "train": Dataset.from_list(train_rows),
    "dev": Dataset.from_list(dev_rows),
    "test": Dataset.from_list(test_rows),
})

print(dataset)
```

Why these steps?

- Organize data using Hugging Face `DatasetDict`.
- Keep the original train, dev, and test splits.

Section 4. Training

4. Load the tokenizer and model

Fine-tune DistilBERT for ASC token classification.

```
from transformers import (
    AutoTokenizer,
    AutoModelForTokenClassification,
    DataCollatorForTokenClassification,
    Trainer,
    TrainingArguments,
    set_seed,
)

set_seed(42)

model_name = "distilbert-base-cased"

tokenizer = AutoTokenizer.from_pretrained(
    model_name,
    use_fast=True,
)
...
```

What are we loading?

- **Tokenizer**: converts text into model inputs.
- **DistilBERT**: a smaller, faster version of BERT.
- **Classification layer**: predicts one of our eight labels.
- As we learned in the class, the model will learn these weights during training.

4. Tokenization and label alignment

One original word may become multiple subtokens.

Original words/labels:

```
tokens = ["I", "became", "disappointed"]

tags = ["O", "INTRAN_RES", "O"]
```

Illustrative example:

```
Token           Label
----------------------------
[CLS]           -100
I                  O
became       INTRAN_RES
disappoint         O
##ed              -100
[SEP]           -100
```

How are labels aligned?

- Assign the word's label to its first subtoken.
- Assign `-100` to additional subtokens.
- Also assign `-100` to special tokens.

4. Tokenize and align labels

Align word-level labels with subtokenized inputs.

```
def tokenize_and_align_labels(examples):
    tokenized_inputs = tokenizer(
        examples["tokens"],
        is_split_into_words=True,
        truncation=True,
        max_length=512,
    )

...
```

What does this function do?

- Tokenizes the input words.
- Finds which word each subtoken belongs to.
- Assigns labels to first subtokens.
- Ignores other positions using `-100`.

4. Configure training

Specify how the model will be fine-tuned.

```
training_args = TrainingArguments(
    output_dir="tagger_checkpoints",
    learning_rate=3e-5,
    per_device_train_batch_size=8,
    per_device_eval_batch_size=8,
    num_train_epochs=3,
    weight_decay=0.01,
    eval_strategy="epoch",
    save_strategy="epoch",
    load_best_model_at_end=True,
    metric_for_best_model="eval_loss",
    greater_is_better=False,
    save_total_limit=2,
    report_to="none",
    dataloader_num_workers=0,
    seed=42,
)
```

Key training settings

- **Learning rate**: `3e-5`
- **Batch size**: 8
- **Epochs**: 3
- **Evaluation**: after each epoch
- **Best checkpoint**: lowest dev loss

*Set `num_train_epochs=1` for a quick test run. One epoch means one complete pass through the training data.

4. Create the trainer

Create the data collator:

```
data_collator = DataCollatorForTokenClassification(
    tokenizer=tokenizer,
)
```

Create the trainer:

```
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=tokenized_dataset["train"],
    eval_dataset=tokenized_dataset["dev"],
    processing_class=tokenizer,
    data_collator=data_collator,
)
```

What do these tools do?

- **Data collator**: pads inputs and labels within each batch.
- **Trainer**: manages training, evaluation, and checkpoint saving.
- **Dev set**: used to select the best checkpoint.

4. Train and save the model

Train and save the trained tagger.

Add to the end of `train_tagger.py`:

```
if __name__ == "__main__":
    trainer.train()

    trainer.save_model("asc_tagger")
    tokenizer.save_pretrained("asc_tagger")
```

Run in your terminal:

```
python train_tagger.py
```

Output folders:

```
tagger_checkpoints/
asc_tagger/
```

What happens during training?

- Evaluate on the dev set each epoch.
- Restore the checkpoint with the lowest dev loss.
- Save the selected model and tokenizer.

Section 5. Evaluation

5. Prepare for evaluation

Check your working folder/directory:

```
Assignment4/
  eval_tagger.py
  test.txt
  asc_tagger/
  train.txt
  dev.txt
  train_tagger.py
```

Create `eval_tagger.py` and add the Python code from the following slides in order.

5. Load the trained model

```
from pathlib import Path

import numpy as np
from datasets import Dataset
from sklearn.metrics import accuracy_score, classification_repor
from transformers import (
    AutoTokenizer,
    AutoModelForTokenClassification,
    DataCollatorForTokenClassification,
    Trainer,
    TrainingArguments,
)

tokenizer = AutoTokenizer.from_pretrained("asc_tagger")
model = AutoModelForTokenClassification.from_pretrained("asc_tag
label2id = model.config.label2id
id2label = model.config.id2label
```

Why load these files?

- `asc_tagger`: the trained model
- **Tokenizer**: processes text in the same way as training.
- **Label mappings**: preserve the meaning of numeric label IDs.

5. Read/tokenize the test dataset

```
def read_dataset(file_path):
    rows = []
    text = Path(file_path).read_text(
        encoding="utf-8-sig"
    )
...

test_rows = read_dataset("test.txt")
test_dataset = Dataset.from_list(test_rows)
print("Test sentences:", len(test_rows))

def tokenize_and_align_labels(examples):
    encoded = tokenizer(
        examples["tokens"],
        is_split_into_words=True,
        truncation=False,
    )
    ...
```

What does this function do?

- Reads the `word|label` format.
- Separates tokens and labels.
- Checks labels against the saved inventory.
- Stores each sentence as a dictionary.

5. Generate predictions

Use the trained model to predict test labels.

```
tokenized_test = test_dataset.map(
    tokenize_and_align_labels,
    batched=True,
    remove_columns=["tokens", "tags"],
)

data_collator = DataCollatorForTokenClassification(
    tokenizer=tokenizer,
)

...

trainer = Trainer(
    model=model,
    args=evaluation_args,
    processing_class=tokenizer,
    data_collator=data_collator,
)

prediction_output = trainer.predict(tokenized_test)
```

What happens here?

- Convert test data into model inputs.
- Pad examples within each batch.
- Generate prediction scores for each token.
- Do not train the model again.

5. Extract predicted labels

Select the highest-scoring label:

```
predicted_ids = np.argmax(
    prediction_output.predictions,
    axis=-1,
)

correct_ids = prediction_output.label_ids
```

The two arrays now contain matching correct and predicted labels for the original words.

Why do we filter?

- `argmax`: ?

5. Calculate evaluation scores

```
accuracy = accuracy_score(
    correct_labels,
    predicted_labels,
)

...

report = classification_report(
    correct_labels,
    predicted_labels,
    labels=asc_label_ids,
    target_names=asc_label_names,
    digits=4,
    zero_division=0,
)

print(report)
```

How do we evaluate?

- **Accuracy**: ?
- **Precision**: ?
- **Recall**: ?
- **F1**: ?

5. Save the results

```
results_text = (
    f"Token accuracy (including O): {accuracy:.4f}\n\n"
    f"ASC classification report:\n{report}"
)

output_dir = Path("evaluation_results")
output_dir.mkdir(parents=True, exist_ok=True)

report_path = output_dir / "test_report.txt"
report_path.write_text(
    results_text, encoding="utf-8"
)

print(f"Results saved to {report_path}")
```

Run in your terminal:

```
python eval_tagger.py
```

Output: `evaluation_results/test_report.txt`

6. Coming back to your tagger

Use your own annotated data to fine-tune a model.

Follow the previous tutorials:

- Prepare the data
- Train the model
- Evaluate the model

Change the training settings:

```
num_train_epochs=10,
per_device_train_batch_size=4,
```

The label inventory is created automatically from your `train.txt`.

What is different?

- Use your own `train.txt`, `dev.txt`, and `test.txt`.
- Train for 10 epochs instead of 3.
- Use batch size 4 instead of 8.

6. Customize your model

Save your trained model:

```
trainer.save_model("tense_tagger")
tokenizer.save_pretrained("tense_tagger")
```

Load the same model for evaluation:

```
tokenizer = AutoTokenizer.from_pretrained(
    "tense_tagger"
)

model = AutoModelForTokenClassification.from_pretrained(
    "tense_tagger"
)
```

For languages other than English:

```
model_name = "distilbert-base-multilingual-cased"
```

Remember

- Use a descriptive name (e.g., `tense_tagger`).
- Use the same folder name when saving and loading.
- English: `distilbert-base-cased`.
- Other languages: multilingual DistilBERT.

6. Understand your results

1. Which label has the highest F1? Which has the lowest?
2. Check the `support` column. How many examples are available?
3. Find a word in `test.txt` that the model might have labeled incorrectly. Why is it difficult?
4. Were your annotations consistent? Would another annotator agree?
5. How could you improve your model?

6. Try your tagger on new sentence

```
from transformers import pipeline

tagger = pipeline(
    "token-classification",
    model="your_tagger_name",
    aggregation_strategy="simple",
)

for item in tagger("Write your own new sentence here."):
    print(
        item["word"],
        item["entity_group"],
        round(float(item["score"]), 2)
    )
```

Replace `your_tagger_name` with the folder containing your saved model.

What does this code do?

- Loads your fine-tuned tagger.
- Predicts labels for new text.
- Prints predicted labels and confidence scores.
- Typically omits tokens labeled `O`.

Assignment 4

Please check the submission_note and the description/rubrics for the Assignment 4 (on Lamakū).

Checklist

1. Example tagger

`test_report.txt` — Evaluation results from the example tagger

2. Your tagger

`submission_note.txt` — see here.

`sentences.txt` — annotated sentences

`train.txt`, `dev.txt`, `test.txt` — Split and labeled datasets

`test_report.txt` — Evaluation results

3. Generative tagger

`zero_shot_prompting` — Evidence of zero-shot prompting (screenshot or script) showing the prompt used to identify your target feature

`test_report.txt` — Evaluation results