Tutorial: Statistical Language Processing (CL III)
Summer 2024

Assignment 2: Due Monday, June 3, 2024 - 17:00

Submission

See the general rules which apply to all assignments.

Train your model using the CPU.

Submit the following files to Moodle (1 submission per group):

- preprocessor.py
- trainer.py
- grid_search.py
- evaluator.py
- Models/ (best-model.pt, best-model-info.json)
- Discussion as pdf

Classification with a FeedForward Network

In this assignment you will train and evaluate a feedforward neural network (FNN) to classify German news articles into 9 classes (Etat, Inland, International, Kultur, Panorama, Sport, Web, Wirtschaft, Wissenschaft). You will learn a technique to prevent overfitting the model during training (by evaluating the model at each epoch on the dev data, and returning the model state with the best macro-averaged F1 score). You will also learn how to save and load both training data and models, including model metadata.

An overview of the steps:

- Preprocess: preprocess articles and create files containing X and y tensors for train, dev, and test data
- Train a baseline FNN
- Use grid search for hyperparameter tuning
- Evaluate your best model (as generated by the grid search) on the held-out test data

Patrick Loeber's tutorial on Feed Forward Networks should be helpful to you in completing this assignment. You will not use all of the components that are used in the lesson, but it shows how to create and train the network.

Dataset

We will use a modified version of the 10kGNAD data, which contains roughly 10,000 German news articles classified by topic (Etat, Inland, International, Kultur, Panorama, Sport, Web, Wirtschaft, Wissenschaft). The original data uses a train/test split (90/10). The modified data is split into train/dev/test (80/10/10). You will find the train, dev, and test sets in the Data directory of the starter code, as well as the label codes you should use.

Note: Even though the data files use the extension .csv, they are not well-formed csv files. You will need to handle that.

Starter Code

Directory Structure

- Models:
  - contains the distributed baseline model and its metadata
  - save your trained best model here
- Data:
  - train,dev,test splits as .csv
  - label_map.json to use as class codes
  - save your data tensors here
  - also put the fasttext-de-mini embeddings here (used in Lab2)
- UnitTestData: files used for unit tests

Code Files

- constants.py: defines keys for the dictionaries you will save and load

- preprocessor.py:

  Motivation: Preprocessing with spaCy takes a long time (about 8 minutes for all of the data). To prevent having to preprocess every time you train or evaluate a model, preprocess once and save the preprocessed data in such a way that it can be loaded very quickly to use for training and evaluation.

  The Preprocessor class is used to perform the following tasks:

  - Read the 10kGNAD data into class variables
  - Preprocess each text and calculate its mean embedding, generating a list of mean embedding tensors. The resulting tensors are then *stacked* to create one tensor of shape (n_samples, embedding_dim).
  - Convert the gold class labels (Etat, Inland, International, ...) to an encoded tensor, using the mapping in Data/label_map.json.
  - Save the generated X and y tensors (along with the label map) to file.

  Preprocess the train, dev, and test data. Normally the held-out test data would not be available until after training. Use the train and dev data for training. Use the test data for final evaluation.

  Use the de_core_news_sm spaCy model for annotating the articles, and fasttext-de-mini embeddings. You can disable parsing and NER annotations (which take a long time and are not needed) by using **disable=['parser', 'ner']** as an argument to spacy.load().

- trainer.py:

  The Trainer class is used to instantiate and train a feedforward network as implemented in class FeedForwardNet, and performs the following tasks:

  - Load the train and dev tensors generated by Preprocessor (don't use test tensors for training!).
  - Instantiate a FeedForwardNet with the appropriate parameters and given hyperparameters.
  - Train the model as instructed in the code. See also the additional instructions below.
  - Save a trained model.

  Additional Instructions: We know that if the final loss of a trained model is low, it has learned the training data well. However, the loss is not always a good indication of how the model will perform on unseen data. If the model overfits to the training data, it may not be able to generalize to unseen data well. One way to prevent overfitting is to evaluate the model at each epoch (using the dev data), then use the model that performed best. On each epoch in _training_loop(), evaluate your model using the _macro_f1() method. If the model evaluation is better at this epoch than any of the previous epochs, save the model state, F1 score, and current epoch. Start with a vanilla training loop, such as the one in Patrick Loeber, Lesson 6, and keep track of the best model.

  See the pytorch docs for an example of how to save and load torch models. Use the dictionary keys (defined in constants.py) as instructed in the code. Note that we don't save the loss function or optimizer (just their names), since those are not necessary if the model is intended to be used for inference only.

  You should get a macro-averaged F1-score ≈ 0.6168 on a baseline model. You can find a baseline model and its metadata in the Models directory. As you can see from the metadata, the baseline model was initialized with CrossEntropyLoss as the cost function, AdamW as the optimizer, hidden size = 8, n_epochs = 200, learning rate = .01.

  Class FeedForwardNet is fully implemented and does not require any changes.

  [Training should be fast - about 5-10 seconds.]

- grid_search.py:

  Grid search is a simple way to see how different combinations of hyperparameters affect the performance of a model. Implement a grid search according to the instructions in the code.

- evaluator.py:

  The Evaluator class is used to evaluate a model on the test data (the held-out data that was not used for training). Two performance reports are generated, one as a string (for printing), and another as a dictionary (for programmatically extracting report values).

  - Load a trained model previously saved by Trainer.
  - Load test tensors generated by Preprocessor.
  - Generate performance reports.
  - Evaluate models

Discussion

Report the results (in two tables) of your grid search and the F1-macro scores of the models evaluated by Evaluator. Include a brief discussion of your results.

FeedForwardNet

The network is already implemented in class FeedForwardNet in trainer.py. You don't need to change it, but here is a short description of the network. It is called feed-forward because the output of each layer is fed as input to the next layer, and the layers are traversed sequentially.

The model has the following parameters (these are the variables passed in when instantiating a FeedForwardNet):

- n_dims: number of dimensions in the training data, in this case the dimension of the embeddings.
- hidden_size: number of neurons in the hidden layer (not the number of hidden layers!).
- n_classes: number of classes, in this case 9

Note that hidden_size is a hyperparameter, it can be any integer of your choice (although it is recommended to choose a value between n_classes and n_dims). n_classes and n_dims are NOT hyperparameters - n_classes is determined by the training data, and n_dims is determined by the embeddings.

The model has 3 layers, with Linear() functions in between layers to calculate the relevant linear transformations (xw + b). In the model initialization, the shapes of the weight matrices are specified in the parameters to the two Linear() functions. We have 2 weight matrices, one for the transformation from the input layer to the hidden layer, and one from the hidden layer to the output layer.

The architecture of the model:

- Input Layer:

  The first layer of a network is called the input layer. The input layer contains the inputs to the network, in this case the article embeddings. No activation function is applied to the input layer.

- Linear Transformation from input layer to hidden layer:

  Performs a linear transformation (xw + b), where x is the training input, and w and b are the weights and bias used by the linear1 object. The input size is the number of dimensions in the training data (in this case the dimension of the embeddings). The output size is the number of neurons in the hidden layer, which is an integer passed in when creating the model. The weight matrix is of shape (input_size, output_size)

- Hidden Layer:

  All layers between the input and output layers are called hidden layers. Hidden layers apply non-linear activation functions. This network has only 1 hidden layer which applies ReLU, the recommended activation function for hidden layers. Recall that activation functions are used to determine if a node is activated or not (or the level of activation).

- Linear Transformation from hidden layer to output layer:

  Performs a linear transformation (xw + b), where x is the output of the hidden layer, and w and b are the weights and bias used by the linear2 object. The input size is the size of the hidden layer. The output size of last layer of a multiclass classifier must be the number of classes (in this case 9) The weight matrix is of shape (input_size, output_size)

  When the model is used for inference (making predictions), this is the output of the model, of shape (n_samples, n_classes). The output for each sample is interpreted as a ranking of the classes, where the index of the highest value is the label code of the predicted class. For example, if there are 3 classes with label mapping 'Sport':0, 'Web':1, 'Kultur':2, and the output of the last layer for one sample is [-0.4, 0.25, 0.15], then the prediction is 'Web' for that sample.

- Output Layer:

  The last layer of a network is called the output layer. We use softmax as the activation function. However, **we don't explicitly add the softmax layer**. PyTorch includes softmax in CrossEntropyLoss, because it applies some tricks that make it necessary to perform both operations together. CrossEntropyLoss is used as the loss function because this is a multiclass classification task.

Hyperparameters are parameters of the model that you can change and experiment with to find the best model, such as hidden_size, n_epochs, learning_rate, and optimizer.

Unlike the training we have done so far, we don't update the weights manually in our training loop when using PyTorch. Instead, we choose an **optimizer** to update the weights. The learning rate specified when constructing the optimizer is used as an initial learning rate, but the optimizer may change it during the course of training. There are many optimization algorithms, and new ones or improvements to older ones are still being discovered. Although the choice of optimizer is a hyperparameter, for this assignment we will use AdamW, which is currently considered a good choice.

Point Distribution

Total Points: 50

- preprocessor (Total: 12 pts)
  - _load_csv_data() (2 pts)
  - _load_label_map() (2 pts)
  - load_data() (1 pt)
  - _preprocess_text() (1 pt)
  - _calc_mean_embedding() (1 pts)
  - _generate_X_tensor() (1 pt)
  - _generate_y_tensor() (1 pt)
  - generate_tensors() (1 pt)
  - save_tensors() (1 pt)
  - main (1 pt)
- trainer.Trainer (Total: 18 pts)
  - _load_train_tensors() (1 pt)
  - _load_dev_tensors() (1 pt)
  - load_data() (1 pt)
  - _macro_f1() (2 pts)
  - _training_loop() (5 pts)
  - train() (4 pts)
  - _save_best_model() (4 pts)
- grid_search.py (6 pts)
- evaluator (Total: 10 pts)
  - load_model() (4 pts)
  - load_data() (1 pt)
  - evaluate_model() (3 pts)
  - main (2 pts)
- Discussion (4 pts)