**dataset.py**

This file defines the PhishingDataset class, which is a custom PyTorch Dataset. Its main purpose is

to prepare your URL data for model training.

\* `clean\_text(text)`: A utility function that preprocesses a URL string by removing protocols (like

http://), www., and then splitting the URL into tokens based on common delimiters (like /, ., ?).

\* `PhishingDataset` class:

\* Takes a list of URLs, their corresponding labels, a vocabulary (stoi), and a maximum sequence

length (max\_len).

\* The \_\_len\_\_ method returns the total number of samples in the dataset.

\* The \_\_getitem\_\_ method is crucial: it takes an index, retrieves a URL and its label, cleans

the URL using clean\_text, converts the URL tokens into numerical IDs using the vocabulary,

and pads/truncates the sequence to max\_len. It then returns the numericalized URL and its

label as PyTorch tensors.

**eval.py**

This script is responsible for evaluating the performance of a trained model on a given dataset.

\* It loads a pre-trained model checkpoint (best.pt), including the model's state dictionary and

hyperparameters (like max\_len, emb\_dim, hid).

\* It loads the vocabulary (itos.txt) to ensure consistent tokenization.

\* It uses the PhishingDataset to load the evaluation data.

\* It performs inference on the data, collecting the true labels and predicted probabilities.

\* Finally, it calculates and prints standard classification metrics such as precision, recall,

F1-score, accuracy, confusion matrix, PR-AUC, and ROC-AUC.

**export.py**

This script packages your trained PyTorch model into a .mar (Model Archive) file, which is the

standard format for deploying models with TorchServe.

\* It loads the best model checkpoint (best.pt).

\* It saves the model's state dictionary (model.pt) and the vocabulary (itos.txt) into a temporary

directory.

\* It copies the custom handler.py into this temporary directory.

\* It then uses the torch-model-archiver command-line tool to create the .mar file, bundling all

necessary components (model, handler, extra files like vocabulary) into a single archive.

\* Finally, it cleans up the temporary directory.

**handler.py**

This is a custom TorchServe handler that defines how your model processes incoming requests and

returns predictions. It's the bridge between TorchServe and your PyTorch model.

\* `clean\_text(text)`: A local copy of the URL cleaning function, ensuring the handler is

self-contained.

\* `PhishHandler` class: Inherits from BaseHandler provided by TorchServe.

\* `initialize(self, ctx)`: Called once when the model is loaded by TorchServe. It sets up the

device (CPU/GPU), loads the vocabulary (itos.txt), and loads the PhishingClassifier model

using the saved state dictionary and hyperparameters from the best.pt checkpoint. It also

initializes PAD and UNK token IDs.

\* `load\_model(self, model\_dir)`: A helper method within the handler to load the actual PyTorch

model and its state dictionary.

\* `\_encode(self, text)`: Converts a raw URL string into a numerical PyTorch tensor, similar to

how PhishingDataset does it, using the loaded vocabulary and max\_len.

\* `preprocess(self, data)`: Takes raw input data (e.g., from a curl request), extracts the URL

strings, and converts them into a batch of numerical tensors ready for the model.

\* `inference(self, model\_input)`: Passes the preprocessed input through the loaded

PhishingClassifier model to get raw logits, then applies a softmax to get probabilities and

determines the predicted class and confidence.

\* `postprocess(self, inference\_output)`: Formats the model's raw predictions (class IDs and

confidences) into a human-readable JSON response, mapping class IDs back to "benign" or

"phish" labels.

**model.py**

This file defines the neural network architecture for your phishing classifier.

\* `PhishingClassifier` class: Inherits from torch.nn.Module.

\* `\_\_init\_\_`: Initializes the layers of the model:

\* nn.Embedding: Converts numerical token IDs into dense vector representations.

\* nn.GRU: A Gated Recurrent Unit layer that processes the sequence of embedded tokens

bidirectionally to capture context from both directions.

\* nn.Dropout: A regularization layer to prevent overfitting.

\* nn.Linear: A fully connected layer that maps the GRU's output to the final class

probabilities.

\* `forward(self, x)`: Defines the forward pass of the model. It takes the input tensor x

(numericalized URLs), passes it through the embedding layer, then the GRU, takes the last

hidden state, applies dropout, and finally passes it through the linear layer to get the

output logits.

**train.py**

This is the main script for training your PhishingClassifier model.

\* It sets up the training environment, including random seeds for reproducibility and creating an

output directory.

\* It loads the dataset splits (training, validation, test) and builds the vocabulary using

functions from utils.py.

\* It calculates class weights to handle potential class imbalance in the dataset.

\* It initializes the PhishingClassifier model, defines the loss function (CrossEntropyLoss),

optimizer (AdamW), and a learning rate scheduler.

\* It implements the training loop, iterating over epochs and batches. For each batch, it performs a

forward pass, calculates the loss, backpropagates, and updates model weights.

\* It includes a validation loop to monitor performance on unseen data and saves the model with the

best validation loss.

\* It uses argparse to allow all training parameters to be configured via command-line arguments.

**tune.py**

This script automates the process of hyperparameter tuning to find the best combination of

parameters for your model.

\* It defines a search space for various hyperparameters (epochs, batch size, learning rate, etc.).

\* It runs multiple "trials," where in each trial:

\* It randomly samples a set of hyperparameters from the defined search space.

\* It calls the train.train() function (from train.py) with these sampled hyperparameters.

\* It records the validation loss returned by the training process.

\* It keeps track of the best validation loss found across all trials and the corresponding set of

hyperparameters.

\* Finally, it prints the best hyperparameters, which you can then use to train your final model.

**utils.py**

This file contains various utility functions that support the main scripts.

\* `seed\_everything(seed)`: Ensures reproducibility by setting random seeds for Python, NumPy, and

PyTorch.

\* `device()`: Detects and returns the appropriate PyTorch device (CUDA if available, otherwise

CPU).

\* `build\_vocab(texts, min\_freq, specials)`: Builds a vocabulary from a list of text (URL tokens).

It counts token frequencies and includes tokens that appear at least min\_freq times, along with

special tokens like <pad> and <unk>.

\* `load\_splits(csv\_path, test\_size, val\_size, min\_freq, max\_len)`: Loads your raw CSV data, cleans

it, performs stratified train-validation-test splits, builds the vocabulary based on the training

data, and then creates PhishingDataset objects for each split.

\* `save\_vocab(itos, path)`: Saves the itos (integer-to-string) vocabulary list to a text file,

where each line is a token.

\* `load\_vocab(path)`: Loads the itos vocabulary list from a text file.

This comprehensive breakdown should give you a clear understanding of each component of your

PhishServe project.