SRC Usage Guide

This document describes the purpose of each major file and folder inside the `src/` package and where those files are used (app, scripts, tests, or other modules). Use this guide for presentations, onboarding, or for quickly locating functionality to modify.

How to read this file

- Each section corresponds to a `src/` subfolder or important root module.
- For each file: **Purpose**, **Key functions/classes**, and **Where used / imported by**.
- Example import snippets are included where helpful.

`src/app/app.py`: Main Streamlit application

- Purpose: Runs the interactive web UI that ties together preprocessing, model inference, explainability, LLM explanations, and safety checks.
- Key features: single-text analysis tab, batch CSV processing, model comparison tab, model info tab, token-level highlighting, LLM rationale display, crisis detection and resource suggestions.
- Where used: Run directly with `streamlit run src/app/app.py`; imports modules from `src.data`, `src.explainability`, `src.models`, `src.safety`, `src.core`, and `src.prompts`.
- Example import (inside app):
  - `from src.data.loaders import MentalHealthDataset`
  - `from src.explainability.token_attribution import TokenAttribution`

`src/data/`

- Purpose: All dataset-loading and preprocessing utilities.

Files:

- `loaders.py`
  - Purpose: `MentalHealthDataset` unified loader and light validation/preprocessing helpers.
  - Key classes/functions: `MentalHealthDataset`, loader helpers for Dreaddit/CLPsych/eRisk (helper functions inside file).
  - Where used: Training scripts, `src/app/app.py`, `scripts/*` that load CSVs for batch inference or model training.
  - Example:
    - `from src.data.loaders import MentalHealthDataset`
- `preprocessing.py` (and `filters.py`)
  - Purpose: Text cleaning, token normalization, filtering short/invalid posts, and upstream `is_valid_text` helper used by `loaders`.
  - Where used: `loaders.py`, training pipeline, and the Streamlit app before inference.

`src/evaluation/`

- Purpose: Evaluation and comparison metrics for model performance and explanation quality.

Files:

- `metrics.py`
  - Purpose: `Evaluator` utilities to compute accuracy, precision/recall/F1, AUC, confusion matrices, and explanation quality stats.
  - Where used: Model training/evaluation scripts (`train_*`, `compare_models.py`), test suite, and `src/app` model comparison tab.
  - Example import: `from src.evaluation.metrics import Evaluator`
- `model_comparison.py`, `faithfulness_metrics.py`, `explainability_metrics.py`, `clinical_validity.py`
  - Purpose: Specialized routines for comparing models, measuring faithfulness of explanations, and clinical-alignment checks.
  - Where used: `scripts/benchmark.py`, `scripts/compare_models.py`, `tests/`, and the web `Model Comparison` tab.

`src/explainability/`

- Purpose: Collection of explainers and helper tools used to produce token-level or example-level explanations.

Files (high level):

- `token_attribution.py`
  - Purpose: Extract token-level importances (attention/grad-based) and produce token→weight maps used by the UI heatmap.
  - Where used: `src/app` (highlighting), `predict_depression.py`, `scripts/quick_start.py`.
- `attention.py`, `attention_supervision.py`
  - Purpose: Attention extraction & optional supervision heuristics (e.g., attention normalization, top-k highlighting).
  - Where used: Attention-based explainers, tests, and the app's token visualization pipeline.
- `integrated_gradients.py`
  - Purpose: IG-based attributions for gradient-capable models.
  - Where used: Explainability tests and optional explainability mode in app.
- `lime_explainer.py`, `shap_explainer.py`
  - Purpose: LIME and SHAP wrappers (optional dependencies). Provide perturbation-based explanations.

    - Where used: `scripts/benchmark.py`, `tests` (skipped if dependencies missing), and `src/app` explainability options.

- `llm_explainer.py`
    - Purpose: Build LLM prompts and convert attention/weights → clinician-friendly rationales; helper to call `src.models.llm_adapter`.
    - Where used: `src/app` LLM explanation panel, `scripts/demo.py` for example outputs.

- `rule_explainer.py`, `dsm_phq.py`
    - Purpose: Rule-based mapping from keywords/phrases → DSM/PHQ symptom labels; deterministic fallback when LLMs are unavailable.
    - Where used: `src/app` (DSM mapping display), `scripts/quick_start.py`, tests and evaluation for clinical-validity scores.

- `developer_tools.py`
    - Purpose: Small helpers used by devs to visualize, export, and debug explanation outputs.
    - Where used: Developer utilities in `scripts/` and during debugging sessions.

`src/core/`

- Purpose: Project-wide constants, config management, and small core utilities used everywhere.

Files:

- `config.py`
    - Purpose: `Config` class for YAML-based configuration loading and typed accessors.
    - Where used: Any module that requires config (training scripts, `src/app`, evaluation/benchmarking code).
    - Example: `from src.core.config import Config; cfg = Config.from_default()`

- `constants.py`
    - Purpose: Global constants such as label maps, seed defaults, and environment flags.
    - Where used: Training code, evaluation, and the UI.

`src/models/`

- Purpose: Model loading, adapters for classical ML and LLM interfaces, and prediction wrappers.

Files:

- `llm_adapter.py`
    - Purpose: Centralized LLM prompt building and provider adapters (OpenAI, Groq, HuggingFace, local). Contains `build_instruction_prompt` and provider-specific wrappers.

- Where used: `src/explainability/llm_explainer.py`, `src/app` LLM panel, `scripts/demo.py`.
- Example: `from src.models.llm_adapter import build_instruction_prompt`

- `classical.py`, `calibration.py`
  - Purpose: Traditional classifier wrappers, probability calibration utilities, and ensembling helpers.
  - Where used: `train_depression_classifier.py`, `scripts/compare_models.py`, `src/app` inference engine.

`src/prompts/`

- Purpose: Prompt templates and prompt manager for building consistent LLM requests.

Files:

- `manager.py`
  - Purpose: `PromptManager` class to load `.txt` templates (zero-shot, few-shot, CoT) and build final prompts.
  - Where used: `src/models/llm_adapter.py`, `src/explainability/llm_explainer.py`, tests that validate prompts.
  - Example: `from src.prompts.manager import PromptManager; pm = PromptManager(); pm.build_prompt('zero_shot', text)`

- Template files (e.g., `zero_shot.txt`, `few_shot.txt`, `cot.txt`)
  - Purpose: Canonical text templates used to ensure consistent LLM prompting across runs.
  - Where used: All LLM calls.

`src/safety/`

- Purpose: Safety-first logic: crisis detection, ethical constraints, disclaimers, and routing to resources.

Files:

- `ethical_guard.py`
  - Purpose: `detect_crisis_risk` and related patterns (keyword lists, regex matchers) that trigger emergency flow.
  - Where used: `src/app` (immediate safety check before showing explanations), `scripts/demo.py` and inference code to avoid unsafe outputs.
  - Example: `from src.safety.ethical_guard import detect_crisis_risk`

- `__init__.py` and config helpers
  - Purpose: Expose safety thresholds and constant resource text.

Cross-file usage patterns & guidance

- The central orchestrator is `src/app/app.py`. When the app runs it will typically:
  - Load config via `src.core.config.Config`.
  - Preprocess text with `src.data.preprocessing`.
  - Run `src.safety.ethical_guard.detect_crisis_risk` before any model inference.
  - Run prediction using classifier wrappers from `src.models`.
  - Produce token-level attributions from `src.explainability.token_attribution` and map them to DSM symptoms via `src.explainability.dsm_phq`.
  - Optionally call `src.models.llm_adapter` + `src.prompts.manager` (via `src.explainability.llm_explainer`) for natural-language rationales.
- Scripts (in `/scripts`) typically import the same `src.*` modules, for example:
  - `scripts/quick_start.py` uses `src.explainability.rule_explainer` + `src.data.loaders`.
  - `scripts/demo.py` uses `src.models.llm_adapter`, `src.explainability.llm_explainer`, and `src.safety.ethical_guard`.
  - `scripts/benchmark.py` uses `src.evaluation` and `src.explainability` modules.

Quick import examples

- Prediction + explanation in a script:

```
from src.models.classical import load_classifier, predict_text
from src.explainability.token_attribution import TokenAttribution
from src.safety.ethical_guard import detect_crisis_risk

text = "I've been feeling hopeless and tired"
if detect_crisis_risk(text)['crisis']:
    print('Crisis detected — escalate to resource flow')
else:
    model = load_classifier('models/trained/bert-best')
    pred, probs = predict_text(model, text)
    attributor = TokenAttribution(model)
    token_weights = attributor.attribute(text)
    print(pred, probs, token_weights)
```

Training scripts

- Purpose: Scripts that fine-tune or evaluate models on datasets. These scripts may live at repository root (e.g., `train_depression_classifier.py`) or in `scripts/`.
- Where used: CLI training, experimentation, and producing checkpoints used by the app and inference scripts.

File: `train_depression_classifier.py`

- Purpose: Fine-tunes transformer models (BERT/RoBERTa/DistilBERT) on project datasets and saves checkpoints and evaluation reports.
- Key behaviour: Loads a CSV, splits train/test, tokenizes, trains with HuggingFace `Trainer`, saves model and tokenizer, writes a `training_report.json`.
- Current integration: This file is a self-contained trainer (uses `transformers` and `datasets`) and does not currently import `src.data.loaders` or `src.models.classical`.
- Recommendation: For consistency and maintainability, refactor `train_depression_classifier.py` to reuse `src/` utilities:
  - Use `from src.data.loaders import MentalHealthDataset` or adapt `load_data` to call `src.data.loaders`.
  - Use model wrappers from `src.models` (e.g., `src.models.classical`) to load and wrap models consistently with inference code.
  - Use `src.core.config.Config` for paths and hyperparameter defaults.

Example refactor snippet (suggested):

```
# inside train_depression_classifier.py
from src.data.loaders import MentalHealthDataset, load_custom_csv  #
hypothetical helper
from src.core.config import Config
from src.models.classical import build_transformer_for_classification

cfg = Config.from_default()
df = load_custom_csv(cfg.get('data.train_csv', 'data/dreaddit-train.csv'))
model = build_transformer_for_classification(args.model, num_labels=2)
```

Why refactor: ensures consistent preprocessing, shared constants (label mappings), and easier testing. The app and inference paths will then load models and datasets consistently.

Testing & developer notes

- Tests import most of these modules; if you add a new file to `src/` also add a lightweight unit test under `tests/` that verifies imports and basic behavior (no heavy LLM calls).
- Optional libraries: `lime` and `shap` are optional. Code paths that use them guard imports and tests skip those modules when they are not installed.
- Configuration: Use `src/core/config.py` to centralize filepaths and external API keys (do not hard-code keys in repo).

Next steps / recommended edits

- If you'd like, I can:
  - Add inline cross-links from this markdown to the `README.md` and `SCRIPTS_USAGE_GUIDE.pdf`.
  - Expand any file section into a code-snippet walkthrough (e.g., a full example of running a batch CSV through `src/app`'s backend functions).
  - Run the Streamlit app and capture runtime import traces to auto-generate a usage map.

File generated: `SRC_USAGE_GUIDE.md` in repository root.