```
"""
hallucination_shield.py
----------------------
**HallucinationShield** – audit-grade, self-contained safety filter for
language-model outputs. It assigns a composite confidence score and flags
potential hallucinations based on *factuality, semantic-role consistency,
discourse coherence,* and *tone*. No external NLP libraries are required.
The class is designed for drop-in use as a post-generation guardrail in
multi-agent or real-time pipelines.
"""
from __future__ import annotations
import re
from collections import defaultdict
from typing import Dict, List, Optional, Tuple
class HallucinationShield:
"""
Parameters
----------
weights : Optional[Dict[str, float]]
Custom weights for the composite confidence score. keys ``factuality``, ``consistency``,
``discourse`` and ``tone``.
Must contain the
"""
# ---------------------------- configuration ---------------------------- #
_DEFAULT_WEIGHTS: Dict[str, float] = {
"factuality": 0.30,
"consistency": 0.30,
"discourse": 0.25,
"tone": 0.15,
}
# broad verb lexicon – auxiliaries + common lexical verbs
_VERB_LEXICON = (
"am is are was were be been being has have had do does did will would "
"shall should can could may might must become becomes became "
"say says said make makes made take takes took know knows knew think "
"thought get gets got give gives gave see sees saw believe believes "
"want wants wanted need needs needed feel feels felt leave leaves left "
"call calls called find finds found put puts put tell tells told"
).split()
_CERTAINTY_WORDS = {
"definitely", "certainly", "undoubtedly", "always", "never",
"clearly", "obviously", "without doubt", "no doubt",
}
_DISCOURSE_MARKERS = {
"however", "but", "nevertheless", "nonetheless",
"alternatively", "on the other hand",}
_CAUSAL_MARKERS = {"therefore", "thus", "hence", "so", "as a result"}
_SOFTEN_MAP = {
"definitely": "possibly",
"certainly": "likely",
"undoubtedly": "probably",
"always": "often",
"never": "rarely",
```

```
"clearly": "apparently",
"obviously": "apparently",
"without doubt": "with some doubt",
"no doubt": "perhaps",
}
# ------------------------------ lifecycle ------------------------------ #
def __init__(self, *, weights: Optional[Dict[str, float]] = None) -> None:
self._weights = weights or self._DEFAULT_WEIGHTS.copy()
# ----------------------------------------------------------------------- #
# PUBLIC API #
# ----------------------------------------------------------------------- #
def evaluate(self, text: str) -> Dict:
"""
Analyse *text* and return a structured dictionary suitable for logging.
Returns
-------
dict
confidence_score : float in [0, 1]
confidence_level : "High Confidence" | "Moderate Confidence"
| "Low Confidence"
issues_detected : list[str]
rationale : str
suggested_rewrite : Optional[str]
"""
text = text.strip()
factual = self._score_factuality(text)
consistency = self._score_consistency(text) # SRL-aware
discourse = self._score_discourse(text)
tone = self._score_tone(text)
# composite score
component_scores = {
"factuality": factual,
"consistency": consistency,
"discourse": discourse,
"tone": tone,
}
conf = sum(component_scores[k] * self._weights[k]
for k in component_scores)
if conf >= 0.80:level = "High Confidence"
elif conf >= 0.60:
level = "Moderate Confidence"
else:
level = "Low Confidence"
issues, rationale_bits = [], []
if factual < 0.60:
issues.append("factual_issue")
rationale_bits.append("low factuality")
if consistency < 0.60:
issues.append("srl_conflict")
rationale_bits.append("subject–verb polarity conflict")
if discourse < 0.60:
issues.append("discourse_shift")
rationale_bits.append("weak discourse coherence")
if tone < 0.60:
```

```
issues.append("overconfident_tone")
rationale_bits.append("strong certainty language")
suggestion = None
if level != "High Confidence":
suggestion = self._revise(text, tone, consistency, discourse)
return {
"confidence_score": round(conf, 3),
"confidence_level": level,
"issues_detected": issues,
"rationale": "; ".join(rationale_bits) or "no significant issues",
"suggested_rewrite": suggestion,
}
# ---------------------------------------------------------------------- #
# KNOWLEDGE-CHECK PLACEHOLDER #
# ---------------------------------------------------------------------- #
def verify_with_knowledge_base(self, statement: str) -> float:
"""
Stub for integration with an external knowledge base.
Returns 0.5 by default (unknown factuality).
"""
return 0.5
# ---------------------------------------------------------------------- #
# WEIGHT OPTIMISER #
# ---------------------------------------------------------------------- #
def optimize_weights(self, new_weights: Dict[str, float]) -> None:
"""
Replace the current weight configuration (keys must match default).
"""
required = set(self._DEFAULT_WEIGHTS)
if set(new_weights) != required:
raise ValueError(f"weights must contain keys {required}")
if sum(new_weights.values()) <= 0:raise ValueError("weight sum must be positive")
self._weights = new_weights.copy()
# ---------------------------------------------------------------------- #
# INTERNAL HELPER FUNCTIONS #
# ---------------------------------------------------------------------- #
_ENTITY_RGX = re.compile(r"\b[A-Z][a-zA-Z]+\b")
def _extract_entities(self, text: str) -> List[str]:
"""Very naive named-entity extractor: consecutive capitalised tokens."""
tokens = self._ENTITY_RGX.findall(text)
ents, i = [], 0
while i < len(tokens):
ent = tokens[i]
j = i + 1
while j < len(tokens) and tokens[j][0].isupper():
ent += " " + tokens[j]
j += 1
ents.append(ent)
i = j
return ents
# ..... #
# ---------- SEMANTIC ROLE LABEL ---------- #
# ..... #
_NEGATIONS = {
```

```
"not", "no", "never", "none", "n't", "cannot", "can't",
"won't", "shouldn't", "couldn't", "didn't", "doesn't",
}
def _parse_clause(self, clause: str) -> Dict:
"""
Attempt SRL on a *single simple clause* (subject–verb–object).
Handles
-------
* Passive voice – “was chosen by Alice” → subj=alice, obj=implicit
* Compound verbs – “smiled and nodded” → verb="smiled & nodded"
* Negation – any neg-word within ±2 tokens of main verb
"""
tokens = clause.strip().split()
if not tokens:
return {"subject": None, "verb": None, "object": None,
"negated": False, "parse_ok": False,
"error": "empty clause"}
lowered = [t.lower() for t in tokens]
# --- passive voice ------------------------------------------------- #
# pattern: be-aux + past-participle [+ by agent]
try:
be_idx = next(i for i, t in enumerate(lowered)
if t in {"is", "are", "was", "were", "been", "being", "be"})if be_idx + 1 < len(tokens):
pp = tokens[be_idx + 1]
if pp.endswith("ed") or pp.lower() in {"given", "seen", "known", "made"}:
# Is there a "by <agent>"?
if "by" in lowered[be_idx + 2:]:
by_idx = lowered.index("by", be_idx + 2)
agent = " ".join(tokens[by_idx + 1:]).strip()
patient = " ".join(tokens[:be_idx]).strip()
neg = any(tok in self._NEGATIONS
for tok in lowered[max(0, be_idx - 2):be_idx + 3])
return {
"subject": agent.lower() if agent else None,
"verb": pp.lower(),
"object": patient.lower() if patient else None,
"negated": neg,
"parse_ok": True,
"error": None,
}
except StopIteration:
pass # no be-aux – fall back to active parsing
# --- active voice -------------------------------------------------- #
verb_idx = None
for i, t in enumerate(lowered):
if t in self._VERB_LEXICON:
verb_idx = i
break
if verb_idx is None:
return {"subject": None, "verb": None, "object": None,
"negated": False, "parse_ok": False,
"error": "verb not found"}
# compound verbs joined by "and"/"or"
```

```
verb_chain = [tokens[verb_idx].lower()]
i = verb_idx + 1
while i + 1 < len(tokens) and lowered[i] in {"and", "or"} \
and lowered[i + 1] in self._VERB_LEXICON:
verb_chain.append(tokens[i + 1].lower())
i += 2
subj = " ".join(tokens[:verb_idx]).strip() or None
obj = " ".join(tokens[i:]).strip() or None
neg_window = lowered[max(0, verb_idx - 2):verb_idx + 3]
neg = any(tok in self._NEGATIONS for tok in neg_window)
parse_ok = subj is not None
return {
"subject": subj.lower() if subj else None,
"verb": " & ".join(verb_chain),
"object": obj.lower() if obj else None,
"negated": neg,
"parse_ok": parse_ok,
"error": None if parse_ok else "subject missing",
}# ..... #
def _extract_srl(self, sentence: str) -> Dict:
"""
Robust SRL extractor for arbitrary sentences.
* Splits sentence into simple clauses on `;`, `,`, ` and `, ` but `.
* Parses each clause via `_parse_clause`.
* If **exactly one** clause yields `parse_ok=True`, that SRL is chosen.
Otherwise returns `parse_ok=False` with an explanatory `error`.
"""
sent = sentence.strip()
if not sent:
return {"subject": None, "verb": None, "object": None,
"negated": False, "parse_ok": False,
"error": "empty sentence"}
clauses = re.split(r"\s*(?:;|,|\sand\s|\sbut\s)\s*", sent)
ok_parses = [self._parse_clause(c) for c in clauses if c.strip()]
ok_parses = [p for p in ok_parses if p["parse_ok"]]
if len(ok_parses) == 1:
return ok_parses[0]
if len(ok_parses) == 0:
return {"subject": None, "verb": None, "object": None,
"negated": False, "parse_ok": False,
"error": "unparsable"}
return {"subject": None, "verb": None, "object": None,
"negated": False, "parse_ok": False,
"error": "ambiguous syntax"}
# ---------------------------------------------------------------------- #
# SCORING #
# ---------------------------------------------------------------------- #
def _score_factuality(self, text: str) -> float:
ents = self._extract_entities(text)
hedge_words = {"may", "might", "could", "possibly", "reportedly",
"allegedly"}
hedge_bonus = 0.03 * sum(text.lower().count(h) for h in hedge_words)
base = 0.75 - 0.05 * len(ents) + hedge_bonus
kb_score = self.verify_with_knowledge_base(text)
```

```
return (max(0.0, min(1.0, base)) + kb_score) / 2
# ..... #
def _score_consistency(self, text: str) -> float:
claims: Dict[Tuple[str, str, Optional[str]], set[bool]] = defaultdict(set)
for sent in re.split(r"[.!?]\s*", text):
if not sent:
continue
srl = self._extract_srl(sent)
if not srl["parse_ok"]:
continue
key = (srl["subject"] or "", srl["verb"], srl["object"])
claims[key].add(srl["negated"])if not claims:
return 0.90
contradictions = sum(1 for v in claims.values() if len(v) > 1)
return max(0.0, 1.0 - contradictions / len(claims))
# ..... #
def _score_discourse(self, text: str) -> float:
sents = [s.strip() for s in re.split(r"[.!?]", text) if s.strip()]
if len(sents) <= 1:
return 0.9
entity_seq, gender_map, inconsist = [], {}, 0
pron_map = {"he": "M", "she": "F", "they": "PL"}
for idx, s in enumerate(sents):
srl = self._extract_srl(s)
subj = srl["subject"]
entity_seq.append(subj)
if idx and subj and subj != entity_seq[idx - 1]:
prev_tail = sents[idx - 1].lower().split()[-3:]
if not any(m in prev_tail for m in self._DISCOURSE_MARKERS):
inconsist += 1
# gender consistency
for p in re.findall(r"\b(he|she|they)\b", s, re.I):
g = pron_map[p.lower()]
if subj:
if subj not in gender_map:
gender_map[subj] = g
elif gender_map[subj] != g:
inconsist += 1
# causal marker must have premise
if any(s.lower().startswith(c + " ") for c in self._CAUSAL_MARKERS):
prev = sents[idx - 1].lower()
if not re.search(r"\b(because|since|due to|after|as)\b", prev):
inconsist += 1
return max(0.0, 1.0 - inconsist / (len(sents) - 1))
# ..... #
def _score_tone(self, text: str) -> float:
hits = sum(len(re.findall(rf"\b{re.escape(w)}\b", text, re.I))
for w in self._CERTAINTY_WORDS)
return 1.0 if hits == 0 else max(0.5, 1.0 - 0.1 * hits)
# ---------------------------------------------------------------------- #
# REWRITE #
# ---------------------------------------------------------------------- #
def _revise(self, text: str, tone: float,
consistency: float, discourse: float) -> str:
```

```
"""
Return a softened / annotated rewrite when confidence is low."""
revised = text
if tone < 0.80:
for strong, soft in self._SOFTEN_MAP.items():
revised = re.sub(rf"\b{re.escape(strong)}\b",
soft, revised, flags=re.I)
notes = []
if consistency < 0.80:
notes.append("possible contradictory claims")
if discourse < 0.80:
notes.append("weak discourse flow")
if notes:
revised += "\n\nNote: " + "; ".join(notes) + ". Please verify."
return revised
```

Supplemental Artifacts and License:

Two small JSON files containing representative clean and adversarial test cases were used during development of this reference module. These artifacts are provided for transparency and reproducibility only and are not required for understanding or use of the reference implementation. The artifacts are hosted alongside this document for local inspection and offline analysis. Reference implementation released under the MIT License for unrestricted use.