ДОДАТОК А

Програмна реалізація Python

Лістинг А.1 – файл calibration.py

```
from __future__ import annotations

from dataclasses import dataclass
from itertools import combinations, product
from pathlib import Path
from typing import Iterable, Sequence

import numpy as np
from scipy.stats import chi2_contingency
from sklearn.linear_model import LogisticRegression
from sklearn.metrics.pairwise import cosine_similarity

from aa.model_classic import ClassicVectorizer
from aa.features import (
    FUNCTION_WORDS_SET,
    build_delta_corpus,
    default_delta_tokenizer,
    delta_features,
)

CALIBRATION_VERSION = "1.1"
CALIBRATION_TEMPERATURE = 0.7

MANUAL_FEATURE_NAMES = (
    "punctuation_frequency",
    "emoji_frequency",
    "capitalization_frequency",
    "avg_word_length",
    "avg_sentence_length",
    "function_word_frequency",
```

```
)
OPTIONAL_EMBED_FEATURE = "cosine_embed"
BASE_CALIBRATION_FEATURES = (
    "cosine_tfidf",
    "delta_score",
    "delta_probability",
    "writeprints_cosine",
    "chi2_function_words",
    "jsd_function_words",
)


def calibration_feature_names(vectorizer) -> tuple[str, ...]:
    feature_names = ["cosine_tfidf"]
    if _has_embedding_channel(vectorizer):
        feature_names.append(OPTIONAL_EMBED_FEATURE)
    feature_names.extend(
        [
            "delta_score",
            "delta_probability",
            "writeprints_cosine",
            "chi2_function_words",
            "jsd_function_words",
        ]
    )
    feature_names.extend(f"manual_diff_{name}"  for  name  in
MANUAL_FEATURE_NAMES)
    return tuple(feature_names)


DEFAULT_DATASET_DIR  =  Path(__file__).resolve().parents[1]  /
"tests" / "data_samples"


def _load_texts(dataset_dir: Path) -> dict[str, list[str]]:
    authors: dict[str, list[str]] = {}
```

```
    for author_dir in sorted(dataset_dir.glob("*")):
        if not author_dir.is_dir():
            continue
        texts: list[str] = []
        for text_file in sorted(author_dir.glob("*.txt")):
            texts.append(text_file.read_text(encoding="utf-
8"))
        if texts:
            authors[author_dir.name] = texts
    if not authors:
        raise FileNotFoundError(
            f"Calibration dataset not found or empty at
{dataset_dir}."
        )
    return authors


def _load_calibration_corpus(dataset_dir: Path | None = None) -
> tuple[list[str], list[str]]:
    dataset_path = dataset_dir or DEFAULT_DATASET_DIR
    authors_map = _load_texts(dataset_path)

    texts: list[str] = []
    authors: list[str] = []
    for author, author_texts in authors_map.items():
        texts.extend(author_texts)
        authors.extend([author] * len(author_texts))

    return texts, authors


def _same_author_pairs(texts: Sequence[str]) ->
Iterable[tuple[str, str]]:
    for a, b in combinations(texts, 2):
        yield a, b
        yield b, a
```

```
def  _different_author_pairs(texts_a:  Sequence[str],  texts_b:
Sequence[str]) -> Iterable[tuple[str, str]]:
    for a, b in product(texts_a, texts_b):
        yield a, b
        yield b, a


def build_calibration_pairs(dataset_dir: Path | None = None) ->
tuple[list[tuple[str, str]], list[int]]:
    dataset_path = dataset_dir or DEFAULT_DATASET_DIR
    authors = _load_texts(dataset_path)

    pairs: list[tuple[str, str]] = []
    labels: list[int] = []

    for author_texts in authors.values():
        for text_a, text_b in _same_author_pairs(author_texts):
            pairs.append((text_a, text_b))
            labels.append(1)

    author_names = sorted(authors)
    for idx, author_a in enumerate(author_names):
        for author_b in author_names[idx + 1 :]:
            for           text_a,           text_b           in
_different_author_pairs(authors[author_a], authors[author_b]):
                pairs.append((text_a, text_b))
                labels.append(0)

    return pairs, labels


def _word_counts(text: str) -> tuple[int, int]:
    words = [token for token in text.split() if token]
    total = len(words)
```

```
    function_count = sum(1 for token in words if token.lower()
in FUNCTION_WORDS_SET)
    return function_count, total


def _chi2_from_counts(count_a: int, total_a: int, count_b: int,
total_b: int) -> float:
    if total_a == 0 or total_b == 0:
        return 0.0

    rest_a = max(total_a - count_a, 0)
    rest_b = max(total_b - count_b, 0)

    contingency = np.array(
        [
            [count_a, rest_a],
            [count_b, rest_b],
        ],
        dtype=float,
    )

    row_sums = contingency.sum(axis=1)
    col_sums = contingency.sum(axis=0)
    if np.any(row_sums == 0.0) or np.any(col_sums == 0.0):
        return 0.0

    with np.errstate(divide="ignore", invalid="ignore"):
        try:
            stat,  _,  _,  _  =  chi2_contingency(contingency,
correction=False)
        except ValueError:
            return 0.0

    if not np.isfinite(stat):
        return 0.0
```

```
    return float(stat)


def _js_divergence_from_counts(count_a: int, total_a: int,
count_b: int, total_b: int) -> float:
    if total_a == 0 or total_b == 0:
        return 0.0

    rest_a = max(total_a - count_a, 0)
    rest_b = max(total_b - count_b, 0)

    dist_a = np.array([count_a, rest_a], dtype=float)
    dist_b = np.array([count_b, rest_b], dtype=float)

    total_dist_a = dist_a.sum()
    total_dist_b = dist_b.sum()
    if total_dist_a == 0.0 or total_dist_b == 0.0:
        return 0.0

    dist_a /= total_dist_a
    dist_b /= total_dist_b

    mean_dist = 0.5 * (dist_a + dist_b)

    def _kl_divergence(p: np.ndarray, q: np.ndarray) -> float:
        mask = (p > 0) & (q > 0)
        if not np.any(mask):
            return 0.0
        return float(np.sum(p[mask] * np.log(p[mask] /
q[mask])))

    js_value = 0.5 * (_kl_divergence(dist_a, mean_dist) +
_kl_divergence(dist_b, mean_dist))
    if not np.isfinite(js_value):
        return 0.0
    return float(js_value)
```

```
def compute_manual_feature_vector(vectorizer, text: str) -> np.ndarray:
    values = [float(func(text)) for func in getattr(vectorizer, "manual_features", [])]
    if not values:
        return np.zeros(0, dtype=float)
    return np.asarray(values, dtype=float)


def compute_writeprints_vector(vectorizer, text: str, *, manual_vector: np.ndarray | None = None) -> np.ndarray:
    writeprints_functions = getattr(vectorizer, "writeprints_features", None)
    if writeprints_functions:
        values = [float(func(text)) for func in writeprints_functions]
        return np.asarray(values, dtype=float)
    if manual_vector is not None:
        return manual_vector
    return compute_manual_feature_vector(vectorizer, text)


def _cosine_dense(vec_a: np.ndarray, vec_b: np.ndarray) -> float:
    if vec_a.size == 0 or vec_b.size == 0:
        return 0.0
    norm_a = float(np.linalg.norm(vec_a))
    norm_b = float(np.linalg.norm(vec_b))
    if norm_a == 0.0 or norm_b == 0.0:
        return 0.0
    cosine_value = float(np.dot(vec_a, vec_b) / (norm_a * norm_b))
    if not np.isfinite(cosine_value):
        return 0.0
```

```
    return cosine_value


def _delta_features_for_pair(vectorizer, text_a: str, text_b:
str) -> tuple[float, float]:
    if not getattr(vectorizer, "use_delta", False):
        return 0.0, 0.0

    tokenizer = getattr(vectorizer, "delta_tokenizer", None) or
default_delta_tokenizer
    segment_length            =            getattr(vectorizer,
"delta_segment_length", 1000)
    probability_model          =          getattr(vectorizer,
"delta_probability_model", None)

    try:
        corpus = build_delta_corpus(
            [text_a],
            ["author_a"],
            tokenizer=tokenizer,
            segment_length=segment_length,
        )
        delta_matrix = delta_features(
            corpus,
            [text_b],
            tokenizer=tokenizer,
            probability_model=probability_model,
            segment_length=segment_length,
        )
    except (RuntimeError, ValueError):
        return 0.0, 0.0

    delta_score = float(delta_matrix[0, 0])
    delta_probability = float(delta_matrix[0, 1])
    if not np.isfinite(delta_score):
        delta_score = 0.0
```

```
    if not np.isfinite(delta_probability):
        delta_probability = 0.0
    return delta_score, delta_probability


def _has_embedding_channel(vectorizer) -> bool:
    method = getattr(vectorizer, "has_embedding_channel", None)
    if callable(method):
        try:
            return bool(method())
        except Exception:
            return False
    return False


def _embedding_vector(vectorizer, text: str):
    method = getattr(vectorizer, "embedding_vector", None)
    if not callable(method):
        return None
    try:
        return method(text)
    except Exception:
        return None


def compute_pair_features(
    vectorizer,
    text_a: str,
    text_b: str,
    *,
    vector_a=None,
    vector_b=None,
    manual_a: np.ndarray | None = None,
    manual_b: np.ndarray | None = None,
    cosine_value: float | None = None,
    embedding_a=None,
```

```
    embedding_b=None,
) -> np.ndarray:
    tfidf_a = vector_a if vector_a is not None else
vectorizer.tfidf.transform([text_a])
    tfidf_b = vector_b if vector_b is not None else
vectorizer.tfidf.transform([text_b])

    if cosine_value is None:
        cosine = float(cosine_similarity(tfidf_a,
tfidf_b)[0][0])
    else:
        cosine = float(cosine_value)

    manual_a_vec = manual_a if manual_a is not None else
compute_manual_feature_vector(vectorizer, text_a)
    manual_b_vec = manual_b if manual_b is not None else
compute_manual_feature_vector(vectorizer, text_b)

    writeprints_a_vec = compute_writeprints_vector(vectorizer,
text_a, manual_vector=manual_a_vec)
    writeprints_b_vec = compute_writeprints_vector(vectorizer,
text_b, manual_vector=manual_b_vec)
    writeprints_cosine = _cosine_dense(writeprints_a_vec,
writeprints_b_vec)

    manual_diff = np.abs(manual_a_vec - manual_b_vec) if
manual_a_vec.size or manual_b_vec.size else np.zeros(0,
dtype=float)

    fw_a, total_a = _word_counts(text_a)
    fw_b, total_b = _word_counts(text_b)
    chi2_function = _chi2_from_counts(fw_a, total_a, fw_b,
total_b)
    jsd_function = _js_divergence_from_counts(fw_a, total_a,
fw_b, total_b)
```

```
    delta_score,                delta_probability                =
_delta_features_for_pair(vectorizer, text_a, text_b)

    use_embed = _has_embedding_channel(vectorizer)
    if use_embed:
        emb_a_vec = embedding_a if embedding_a is not None else
_embedding_vector(vectorizer, text_a)
        emb_b_vec = embedding_b if embedding_b is not None else
_embedding_vector(vectorizer, text_b)
        if emb_a_vec is not None and emb_b_vec is not None:
            emb_a_arr = np.asarray(emb_a_vec, dtype=float)
            emb_b_arr = np.asarray(emb_b_vec, dtype=float)
            embed_cosine = _cosine_dense(emb_a_arr, emb_b_arr)
        else:
            embed_cosine = 0.0
    else:
        embed_cosine = None

    feature_values: list[float] = [float(cosine)]
    if use_embed:
        feature_values.append(float(embed_cosine))
    feature_values.extend(
        [
            float(delta_score),
            float(delta_probability),
            float(writeprints_cosine),
            float(chi2_function),
            float(jsd_function),
        ]
    )

    if manual_diff.size:
        feature_values.extend([float(value)   for   value   in
manual_diff])

    return np.asarray(feature_values, dtype=float)
```

```
@dataclass
class CosineCalibrator:
    model: LogisticRegression
    version: str = CALIBRATION_VERSION
    feature_names: tuple[str, ...] = BASE_CALIBRATION_FEATURES
    vectorizer: ClassicVectorizer | None = None
    tfidf_vocabulary: dict[str, int] | None = None

    def predict_proba(self, features: np.ndarray) -> float:
        probability                                          =
self.model.predict_proba(features.reshape(1, -1))[0, 1]
        return float(probability)


def train_calibrator(
    vectorizer,
    dataset_dir: Path | None = None,
    *,
    random_state: int = 42,
    calibration_vectorizer: ClassicVectorizer | None = None,
) -> CosineCalibrator:
    pairs, labels = build_calibration_pairs(dataset_dir)
    if not pairs:
        raise ValueError("Calibration dataset must contain at
least one pair.")

    if calibration_vectorizer is None:
        calibration_vectorizer = ClassicVectorizer(
            use_delta=getattr(vectorizer, "use_delta", False),
            delta_segment_length=getattr(vectorizer,
"delta_segment_length", 1000),
            use_embeddings=getattr(vectorizer,
"_use_embeddings", True),
        )
```

```
        calibration_vectorizer.tfidf.min_df                    =
getattr(vectorizer.tfidf, "min_df", 1)
        texts, authors = _load_calibration_corpus(dataset_dir)
        calibration_vectorizer.fit(
            texts,
            authors=authors if calibration_vectorizer.use_delta
else None,
        )

    feature_matrix = np.vstack(
        [
            compute_pair_features(calibration_vectorizer,
text_a, text_b)
            for text_a, text_b in pairs
        ]
    )
    feature_names                                              =
calibration_feature_names(calibration_vectorizer)
    model = LogisticRegression(
        random_state=random_state,
        max_iter=1000,
        class_weight="balanced",
    )
    model.fit(feature_matrix, labels)

    vocab_snapshot    =    getattr(calibration_vectorizer.tfidf,
"vocabulary_", {}) or {}
    return CosineCalibrator(
        model=model,
        version=CALIBRATION_VERSION,
        feature_names=feature_names,
        vectorizer=calibration_vectorizer,
        tfidf_vocabulary=dict(vocab_snapshot),
    )
```

```
def _sharpen_probability(p: float, temperature: float) -> float:
    if not np.isfinite(p):
        return 0.5

    p = float(np.clip(p, 1e-6, 1.0 - 1e-6))
    if temperature <= 0.0 or temperature == 1.0:
        return p

    logit = np.log(p / (1.0 - p))
    logit /= temperature
    p_sharp = 1.0 / (1.0 + np.exp(-logit))
    return float(np.clip(p_sharp, 0.0, 1.0))


def calibration_probability(
    calibrator: CosineCalibrator | None,
    features: np.ndarray,
    fallback_probability: float,
) -> float:
    if calibrator is None:
        return fallback_probability

    try:
        probability = calibrator.predict_proba(features)
    except ValueError:
        return fallback_probability

    if not np.isfinite(probability):
        return fallback_probability

    p = float(np.clip(probability, 0.0, 1.0))

    if CALIBRATION_TEMPERATURE is not None:
        p = _sharpen_probability(p, CALIBRATION_TEMPERATURE)

    return p
```

Лістинг А.2 – файл model_classic.py

```
from __future__ import annotations

from collections import Counter
from typing import Sequence

import numpy as np
from scipy.sparse import csr_matrix, hstack
from sklearn.calibration import CalibratedClassifierCV
from sklearn.svm import LinearSVC

from aa.features import (
    avg_sentence_length,
    avg_word_length,
    build_delta_corpus,
    calibrate_delta_corpus,
    capitalization_frequency,
    char_tfidf_vectorizer,
    default_delta_tokenizer,
    delta_features,
    emoji_frequency,
    function_word_frequency,
    punctuation_frequency,
    stack_features,
)


class ClassicVectorizer:
    def __init__(
        self,
        use_delta: bool = False,
```

```
        delta_segment_length: int = 1000,
        use_embeddings: bool = True,
        delta_calibrate: bool = True,
    ):
        self.use_delta = use_delta
        self.tfidf = char_tfidf_vectorizer()
        self.manual_features = [
            punctuation_frequency,
            emoji_frequency,
            capitalization_frequency,
            avg_word_length,
            avg_sentence_length,
            function_word_frequency,
        ]
        self.delta_corpus = None
        self.delta_probability_model = None
        self.delta_tokenizer = default_delta_tokenizer if
use_delta else None
        self.delta_segment_length = delta_segment_length
        self.delta_calibration_warning: str | None = None
        self.delta_calibrate = delta_calibrate
        self._use_embeddings = use_embeddings
        self._embedder = None
        self._embedding_attempted = False
        self._embedding_feature_active = False
        self.embedding_warning: str | None = None


    def fit(self, texts: Sequence[str], authors: Sequence[str]
| None = None):
        n_docs = len(texts)
        if n_docs == 0:
            raise ValueError("ClassicVectorizer.fit requires
at least one text.")
```

```
        min_df = self.tfidf.min_df
        if isinstance(min_df, int) and min_df > n_docs:
            adjusted_min_df = max(1, n_docs)
            self.tfidf.set_params(min_df=adjusted_min_df)

        self.tfidf.fit(texts)

        if self.use_delta:
            if authors is None:
                raise ValueError("Authors are required when
use_delta=True.")

            corpus = build_delta_corpus(
                texts,
                authors,
                tokenizer=self.delta_tokenizer,
                segment_length=self.delta_segment_length,
            )

            probability_model = None
            self.delta_calibration_warning = None

            if self.delta_calibrate:
                try:
                    calibrate_delta_corpus(corpus)
                    probability_model =
corpus.probability_model
                except Exception as exc:
                    probability_model = None
                    self.delta_calibration_warning = str(exc)
            else:
                probability_model = None
```

```
                self.delta_calibration_warning = (
                    "Delta calibration skipped
(delta_calibrate=False); "
                    "Delta probability feature will be neutral
(0.5)."
                )

            self.delta_corpus = corpus
            self.delta_probability_model = probability_model
        else:
            self.delta_corpus = None
            self.delta_probability_model = None
            self.delta_calibration_warning = None

        return self

    def transform(self, texts: Sequence[str]):
        X_tfidf = self.tfidf.transform(texts)

        X_manual = np.array([[f(text) for f in
self.manual_features] for text in texts])
        X_combined = stack_features(X_tfidf, X_manual)

        if getattr(self, "_use_embeddings", False) and
self.has_embedding_channel():
            embed_vectors: list[np.ndarray | None] = []
            first_dim: int | None = None
            any_non_null = False

            for text in texts:
                vec = self.embedding_vector(text)
                if vec is None:
                    embed_vectors.append(None)
```

```
                    continue

                v = np.asarray(vec, dtype=np.float32).ravel()
                if first_dim is None:
                    first_dim = int(v.shape[0])
                any_non_null = True
                embed_vectors.append(v)

            if any_non_null and first_dim is not None and
first_dim > 0:
                embed_matrix = np.zeros((len(texts),
first_dim), dtype=np.float32)
                for i, v in enumerate(embed_vectors):
                    if v is None:
                        continue
                    v = np.asarray(v,
dtype=np.float32).ravel()
                    if v.shape[0] == first_dim:
                        embed_matrix[i, :] = v
                    elif v.shape[0] > first_dim:
                        embed_matrix[i, :] = v[:first_dim]
                    else:
                        embed_matrix[i, : v.shape[0]] = v
                X_combined = stack_features(X_combined,
embed_matrix)

        if self.use_delta:
            if self.delta_corpus is None:
                raise RuntimeError("ClassicVectorizer not
fitted with Delta features.")
            delta_matrix = delta_features(
                self.delta_corpus,
                list(texts),
                tokenizer=self.delta_tokenizer,
```

```
probability_model=self.delta_probability_model,
                segment_length=self.delta_segment_length,
            )
            X_combined = hstack([X_combined,
csr_matrix(delta_matrix)])

        return X_combined

    def fit_transform(self, texts: Sequence[str], authors:
Sequence[str] | None = None):
        return self.fit(texts,
authors=authors).transform(texts)

    def _ensure_embedder(self):
        if not self._use_embeddings:
            return None
        if self._embedder is not None:
            self._embedding_feature_active = True
            return self._embedder
        if self._embedding_attempted:
            return None
        self._embedding_attempted = True
        try:
            from aa.model_embed import MiniLMEmbedder
        except ImportError as exc:  # pragma: no cover -
optional dependency
            self.embedding_warning = f"sentence-transformers
unavailable: {exc}".strip()
            self._use_embeddings = False
            return None
        try:
            self._embedder = MiniLMEmbedder()
            self._embedding_feature_active = True
```

```
            except RuntimeError as exc:
                self.embedding_warning = str(exc)
                self._use_embeddings = False
                self._embedder = None
            return self._embedder

        def has_embedding_channel(self) -> bool:
            if self._embedding_feature_active:
                return True
            if not self._use_embeddings:
                return False
            embedder = self._ensure_embedder()
            return embedder is not None

        def embedding_vector(self, text: str):
            if not self._use_embeddings:
                return None
            embedder = self._ensure_embedder()
            if embedder is None:
                return None
            try:
                return embedder.embed_text(text)
            except RuntimeError as exc:
                self.embedding_warning = str(exc)
                self._use_embeddings = False
                return None


    class ClassicClassifier:
        def __init__(self):
            self.clf: CalibratedClassifierCV | None = None
            self.calibration_warning: str | None = None
```

```
    def fit(self, X, y):
        label_counts = Counter(y)
        min_class_count = min(label_counts.values()) if
label_counts else 0
        self.calibration_warning = None

        if len(label_counts) < 2:
            raise ValueError(
                "ClassicClassifier.fit requires samples from
at least 2 distinct classes."
            )

        if min_class_count < 2:
            base_clf = LinearSVC(
                random_state=42,
                dual=True,
                max_iter=20000,
            )
            base_clf.fit(X, y)

            self.clf = CalibratedClassifierCV(
                base_clf,
                method="isotonic",
                cv="prefit",
            )
            self.clf.fit(X, y)

            self.calibration_warning = (
                "Calibrator was trained with cv='prefit'
because at least one "
                "class had fewer than 2 samples. Probability
estimates may be "
```

```
                "less reliable; consider adding more training
texts per author."
            )
            return self

        cv_folds = min(3, min_class_count)

        base_clf = LinearSVC(
            random_state=42,
            dual=True,
            max_iter=20000,
        )
        self.clf = CalibratedClassifierCV(
            base_clf,
            method="isotonic",
            cv=cv_folds,
        )
        self.clf.fit(X, y)
        return self

    def predict(self, X):
        if self.clf is None:
            raise RuntimeError("Classifier not fitted.")
        return self.clf.predict(X)

    def predict_proba(self, X):
        if self.clf is None:
            raise RuntimeError("Classifier not fitted.")
        return self.clf.predict_proba(X)
```

ДОДАТОК Б

Відомість кваліфікаційної роботи

Позначення	Найменування	Дод. відомості
	Текстові документи	
1.	Пояснювальна записка	131 с.
	Інші документи	
2.	Презентаційні матеріали	28 плакатів

.	.	Прізвище та ініціали.	Підп.	Дата			
Розробив		Капін С.Д.			Система авторської атрибуції із застосуванням ансамблю класичних і нейронних моделей	Шифр групи	Код напр./спец.
Перевірив		Волощук О.Б.				ДСм-24-1	122
Н.контр.		Малєєва І.А.				ХНУРЕ кафедра ШІ	
Затв.		Чала Л.Е.					