ДОДАТОК А

Програмна реалізація експерименту

```
def normalize(X, method="none"):
    if method == "none":
        return X
    elif method == "minmax":
        return MinMaxScaler().fit_transform(X)
    elif method == "robust":
        return RobustScaler().fit_transform(X)
    elif method == "standard":
        return StandardScaler().fit_transform(X)
    elif method == "instancenorm":
            return (X - X.mean(axis=1, keepdims=True)) /
(X.std(axis=1, keepdims=True) + 1e-8)
    elif method == "adanorm":
        mu = X.mean(axis=0)
        sigma = X.std(axis=0) + 1e-8
        alpha = 0.5
        normed = (1 - alpha) * ((X - mu) / sigma) + alpha * (
            (X - X.mean(axis=1, keepdims=True)) / (X.std(axis=1,
keepdims=True) + 1e-8)
        )
        return normed
    else:
            raise ValueError(f"Unknown normalization method:
{method}")

def select_features(X, y, method="none", n_components=50):
    if method == "none":
        return X
    elif method == "pca":
        return PCA(n_components=n_components).fit_transform(X)
    elif method == "autoencoder":
        from tensorflow.keras import layers, models
```

```
        input_dim = X.shape[1]
        encoder = models.Sequential([
            layers.Input(shape=(input_dim,)),
            layers.Dense(128, activation='relu'),
            layers.Dense(64, activation='relu'),
            layers.Dense(n_components, activation='linear'),
        ])
        decoder = models.Sequential([
            layers.Input(shape=(n_components,)),
            layers.Dense(64, activation='relu'),
            layers.Dense(128, activation='relu'),
            layers.Dense(input_dim, activation='linear')
        ])
        autoencoder = models.Sequential([encoder, decoder])
        autoencoder.compile(optimizer='adam', loss='mse')
            autoencoder.fit(X,  X,  epochs=10,  batch_size=128,
verbose=0)
        return encoder.predict(X)
    else:
         raise ValueError(f"Unknown feature selection method:
{method}")

def augment(X, y, method="none"):
    if method == "none":
        return X, y
    elif method == "gaussian":
        noise = np.random.normal(0, 0.01, X.shape)
        return X + noise, y
    elif method == "mixup":
        alpha = 0.2
        lam = np.random.beta(alpha, alpha, size=len(X))
        idx = np.random.permutation(len(X))
        lam_x = lam[:, None]
        X_mix = lam_x * X + (1 - lam_x) * X[idx]
        choose_first = np.random.rand(len(X)) < lam
```

```
        y_mix = np.where(choose_first, y, y[idx]).astype(int)
        return X_mix, y_mix
    elif method == "TimeGAN":
        try:
            from ydata_synthetic.synthesizers.timeseries import
TimeSeriesSynthesizer
                from  ydata_synthetic.synthesizers.base  import
ModelParameters, TrainParameters
        except ImportError as exc:
            print(f"TimeGAN not available ({exc}); falling back
to gaussian noise augmentation.")
            noise = np.random.normal(0, 0.01, X.shape)
            return X + noise, y
        seq_len = 1
        num_cols = [f"feature_{i}" for i in range(X.shape[1])]
        df = pd.DataFrame(X, columns=num_cols)
        model_parameters = ModelParameters(
            batch_size=min(64, len(df)),
            lr=1e-4,
            latent_dim=min(24, X.shape[1]),
            gamma=1,
            seq_len=seq_len,
            n_cols=X.shape[1],
            n_features=X.shape[1]
        )
        train_arguments = TrainParameters(
            epochs=5,
            sequence_length=seq_len,
            number_sequences=X.shape[1]
        )
        try:
                                                synthesizer      =
TimeSeriesSynthesizer(modelname="timegan",
model_parameters=model_parameters)
```

```
            synthesizer.fit(df, train_arguments=train_arguments,
num_cols=num_cols)
             synthetic_samples = synthesizer.sample(len(df))
            synth_data = np.array([sample.values.reshape(-1) for
sample in synthetic_samples])
        except Exception as exc:
              print(f"TimeGAN training failed ({exc}); falling
back to gaussian noise augmentation.")
            noise = np.random.normal(0, 0.01, X.shape)
            return X + noise, y
        synth_data = synth_data[:, :X.shape[1]]
        y_synth = np.random.choice(y, len(synth_data))
            return  np.vstack([X,  synth_data]),  np.hstack([y,
y_synth])
    else:
             raise  ValueError(f"Unknown  augmentation  method:
{method}")
def get_svm():
return SVC(kernel="rbf", C=1.0, gamma="scale")
def get_rf():
return                 RandomForestClassifier(n_estimators=100,
random_state=42)
def get_mlp(input_dim, n_classes):
model = models.Sequential([
layers.Input(shape=(input_dim,)),
layers.Dense(128, activation="relu"),
layers.Dense(64, activation="relu"),
layers.Dense(n_classes, activation="softmax")
])
model.compile(optimizer="adam",
loss="sparse_categorical_crossentropy", metrics=["accuracy"])
return model
def get_cnn(input_dim, n_classes):
model = models.Sequential([
layers.Input(shape=(input_dim, 1)),
```

```
layers.Conv1D(32, 3, activation="relu"),
layers.MaxPooling1D(2),
layers.Flatten(),
layers.Dense(64, activation="relu"),
layers.Dense(n_classes, activation="softmax")
])
model.compile(optimizer="adam",
loss="sparse_categorical_crossentropy", metrics=["accuracy"])
return model
def run_experiment(
normalizer,
selector,
augmenter,
classifier,
*,
dataset_variant="engineered",
verbose=True,
return_model=False,
):
config = {
"normalizer": normalizer,
"selector": selector,
"augmenter": augmenter,
"classifier": classifier,
"dataset_variant": dataset_variant,
}
X, y = load_dataset(dataset_variant, path=DATASET_PATH)
X = normalize(X, normalizer)
X = select_features(X, y, selector)
X, y = augment(X, y, augmenter)
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)
label_ids = np.unique(np.concatenate([y_train,
y_test])).astype(int)
```

```
target_names = [LABEL_MAP.get(int(lbl), str(int(lbl))) for lbl
in label_ids]
history = None
process = psutil.Process(os.getpid())
mem_before = process.memory_info().rss
tracemalloc.start()
start_time = time.perf_counter()
if classifier == "svm":
model = get_svm()
model.fit(X_train, y_train)
y_pred = model.predict(X_test)
elif classifier == "rf":
model = get_rf()
model.fit(X_train, y_train)
y_pred = model.predict(X_test)
elif classifier == "mlp":
model = get_mlp(X_train.shape[1], len(np.unique(y)))
history = model.fit(X_train, y_train, epochs=10, batch_size=64,
verbose=0)
y_pred = np.argmax(model.predict(X_test), axis=1)
elif classifier == "cnn":
X_train = np.expand_dims(X_train, -1)
X_test = np.expand_dims(X_test, -1)
model = get_cnn(X_train.shape[1], len(np.unique(y)))
history = model.fit(X_train, y_train, epochs=10, batch_size=64,
verbose=0)
y_pred = np.argmax(model.predict(X_test), axis=1)
else:
raise ValueError(f"Unknown classifier: {classifier}")
train_time_sec = time.perf_counter() - start_time
current_mem, peak_mem = tracemalloc.get_traced_memory()
tracemalloc.stop()
mem_after = process.memory_info().rss
memory_delta_mb = (mem_after - mem_before) / (1024 ** 2)
peak_memory_mb = peak_mem / (1024 ** 2)
```

```
acc = accuracy_score(y_test, y_pred)
report_dict = classification_report(
y_test,
y_pred,
labels=label_ids,
target_names=target_names,
output_dict=True,
zero_division=0,
)
if verbose:
print(f" Accuracy: {acc:.4f}")
print(
classification_report(
y_test,
y_pred,
labels=label_ids,
target_names=target_names,
digits=4,
zero_division=0,
)
)
print(f" Train + inference time: {train_time_sec:.2f} s")
print(f" Peak traced memory: {peak_memory_mb:.2f} MB | ΔRSS:
{memory_delta_mb:.2f} MB")
results = {
"config": config,
"dataset_variant": dataset_variant,
"accuracy": acc,
"report": report_dict,
"labels": label_ids.tolist(),
"label_names": target_names,
"y_test": y_test,
"y_pred": y_pred,
"history": history.history if history is not None else None,
"perf": {
```

```
"train_time_sec": train_time_sec,
"peak_memory_mb": peak_memory_mb,
"rss_delta_mb": memory_delta_mb,
},
}
if return_model:
results["model"] = model
return results
CONFIG = {
    "normalizer": "standard",
    "selector": "pca",
    "augmenter": "smote",
    "classifier": "cnn"
}
print("Running experiment with configuration:")
for k, v in CONFIG.items():
    print(f"  {k}: {v}")
results = run_experiment(**CONFIG)
plot_experiment_results(results)
def plot_experiment_results(results, label_names=None):
    y_test = results["y_test"]
    y_pred = results["y_pred"]
    labels = results.get("labels")
    if labels is None:
        labels = np.unique(y_test)
    else:
        labels = np.array(labels)
    if label_names is None:
        label_names = results.get("label_names")
    if label_names is None:
         label_names = [LABEL_MAP.get(int(lbl), str(int(lbl)))
for lbl in labels]
    cm = confusion_matrix(y_test, y_pred, labels=labels)
    row_sums = cm.sum(axis=1, keepdims=True)
```

```
    cm_percent = np.divide(cm, row_sums, where=row_sums != 0) *
100
    fig, axes = plt.subplots(1, 2, figsize=(18, 7))
    sns.heatmap(
        cm,
        annot=True,
        fmt="d",
        cmap="Blues",
        xticklabels=label_names,
        yticklabels=label_names,
        cbar_kws={"label": "Count"},
        ax=axes[0],
        annot_kws={"size": 14},
    )
    axes[0].set_xlabel("Predicted label", fontsize=16)
    axes[0].set_ylabel("True label", fontsize=16)
    axes[0].set_title("Confusion Matrix (Counts)", fontsize=18,
pad=15)
          axes[0].set_xticklabels(label_names,    rotation=45,
ha="right", fontsize=14)
           axes[0].set_yticklabels(label_names,    rotation=0,
fontsize=14)
    sns.heatmap(
        cm_percent,
        annot=True,
        fmt=".1f",
        cmap="Greens",
        xticklabels=label_names,
        yticklabels=label_names,
        cbar_kws={"label": "Percentage"},
        ax=axes[1],
        annot_kws={"size": 14},
    )
    axes[1].set_xlabel("Predicted label", fontsize=16)
    axes[1].set_ylabel("True label", fontsize=16)
```

```
    axes[1].set_title("Confusion Matrix (Row %)", fontsize=18,
pad=15)
          axes[1].set_xticklabels(label_names,    rotation=45,
ha="right", fontsize=14)
           axes[1].set_yticklabels(label_names,    rotation=0,
fontsize=14)
    plt.tight_layout(pad=3.0)
    plt.show()
    report_df = pd.DataFrame(results["report"]).T
    if "support" in report_df.columns:
                                    report_df["support"]       =
report_df["support"].fillna(0).astype(int)
    display(report_df.round(3))
    perf = results.get("perf")
    if perf:
        print(
                            f"   Train   +   inference   time:
{perf['train_time_sec']:.2f} s | "
            f" Peak traced memory: {perf['peak_memory_mb']:.2f}
MB | ΔRSS: {perf['rss_delta_mb']:.2f} MB"
        )
    history = results.get("history")
    if history:
        history_df = pd.DataFrame(history)
        plt.figure(figsize=(7, 4))
        epochs = history_df.index + 1
        if "loss" in history_df:
              plt.plot(epochs, history_df["loss"], marker="o",
label="loss")
        for metric in ["accuracy", "val_accuracy", "val_loss"]:
            if metric in history_df:
               plt.plot(epochs, history_df[metric], marker="o",
label=metric)
        plt.xlabel("Epoch")
        plt.ylabel("Metric value")
```

```
plt.title("Training History")
plt.xticks(epochs)
plt.grid(alpha=0.3)
plt.legend()
plt.tight_layout()
plt.show()
```

ДОДАТОК Б

Відомість кваліфікаційної роботи

Позначення	Найменування	Дод. відомості
	Текстові документи	
1.	Пояснювальна записка	75 с.
	Інші документи	
2.	Презентаційні матеріали	25 плакатів

.	.	Прізвище та ініціали.	Підп.	Дата
Розробив		Умнов К.Г.		08.12
Перевірив		Кулішова Н.Є.		09.12
Н.контр.		Малєєва І.А.		
Затв.		Чала Л.Е.		

Дослідження методів попередньої обробки в завданні розпізнавання людської активності	Шифр групи: ДСм-24-1; Код напр./спец.: 122; ХНУРЕ кафедра ШІ