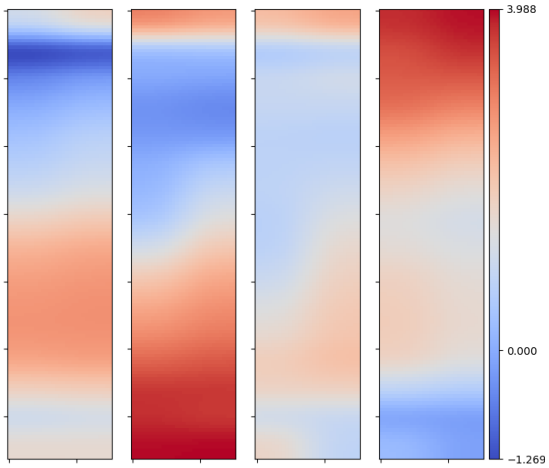
Victoria Droplet Learning Project



Script Overview & User Guide

Description

The Victoria Droplet Learning Project (VDLP) aims to apply interpretable ML methods on problems concerning liquid droplet evaporations. These evaporations are treated as a multivariate time series problem, however we also decompose them into a non time-series and image format and apply several ML and DL approaches to them. This document presents an overview of the preprocessing and ML scripts developed in support of this research, highlighting pertinent information and potential problems for any future user or developer.

Contents

Contributors

Asher Stout authored this document with assistance from Prof. Andrew Lensen, Prof. Gideon Gouws and Harith Al-Sahaf.

1 Introduction

G'day. Now I have your attention, it's important to describe the research at hand and how it translates into a Machine Learning (ML) problem. VDLP's aim is to obtain information about the dynamics of droplet evaporation and feed it to ML models. From an initial sequence of images, a preprocessing script extracts important features of a droplet and exports it in CSV format. Each sample corresponds to a single CSV file with columns representing observed features and rows the timestep they were observed at in the sequence, and thus can represent multivariate time-series data as much as it can image data.

As of writing this data is exclusively applied to classification problems, though extending it to regression is a fairly straightforward next step. We classify samples through 4 ML approaches; simple non time-series models, pure time-series models, fully-convolutional CNNs, and Vision Transformers (ViTs). In addition to this, a PSO-driven feature selection experiment and a clustering-based outlier identification method have also been implemented. All our methods are implemented through a single, dynamic script that is designed for ease-of-use and modularity.

This document covers the details and use of both the preprocessing and experiment scripts, along with some of the motivations and implementation details of our chosen ML methods. As this project is steeped in technical and internal jargon, important terms will be emphasized in **bold**.

2 Data

As of writing VDLP has been primarily concerned with analyzing the evaporations of milk droplets. These come in four classes; dark blue (**DBM**), green trim (**GTM**), light blue (**LBM**) and protein plus (**LBP+**). What differentiates milk from other liquids like water is its proteins and fats. These prevent the droplet from fully evaporating and lead to recognizable differences in the dynamics of evaporation depending on the fat and protein contents.

Samples are obtained in a lab environment in the form of a sequence of profile shots of a droplet taken over a short period. An example evaporation for a GTM sample is presented in Fig 1. The original samples vary in both the number of images and their resolutions, making any direct attempt using machine learning difficult. We impose a standard on this data through an independent preprocessing step.

2.1 Preprocessing

The preprocessing script is separate from our main ML script and was developed for use in lab environments. The aim of the script is to condense a single sample into 2 csv files with height values observed at certain proportional segments of an image. The script is implemented in the `preprocessing/` folder. In practice this script is designed for use by Gideon and you will likely only need to review it if there are newly-reported bugs when samples are being processed.

A **droplet** is a sample at a given time, either in its original (.tiff) format or preprocessed (.csv) format. In its image format, a droplet and the substrate it rests on are black pixels while the background is white. The blurriness of the droplet (i.e. the size of the gradient from black to white along its edge) is due to the focus of the camera and can also vary between samples.

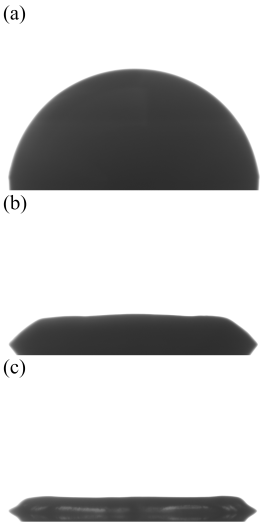


Figure 1. Droplet evaporation, after (a) 20 seconds, (b) 100 seconds, and (c) 200 seconds.

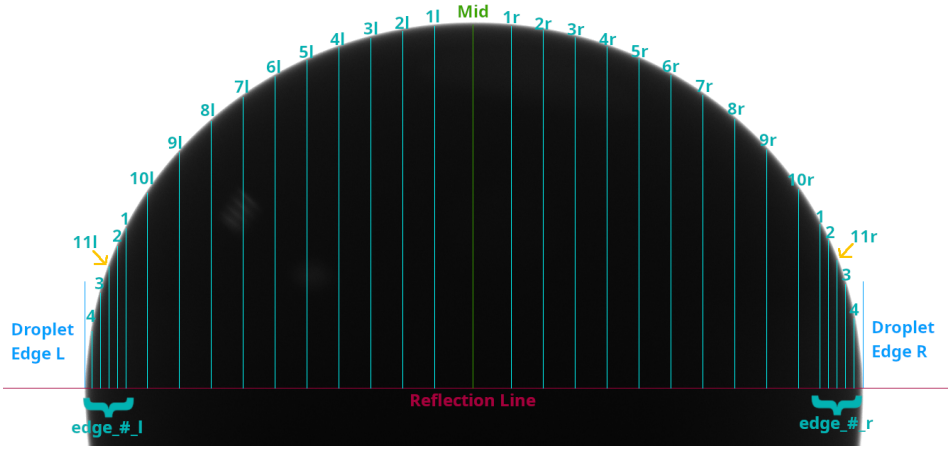


Figure 2. Anatomy of droplet heights extracted from an image.

A droplet's **midpoint** is an anchor for our script, defined as the dead centre of the droplet found halfway along its base width. To find its base width, the **reflection point** or **reflection line** must first be computed. The reflection line is the location of the substrate, with any black pixels below the line being part of the droplet's reflection. Our script identifies this point by looking for parallelism at the sides of the droplet; when we are around the reflection line, the shape of the droplet above/below it should be relatively identical. This is the most common point of failure for our script as parallelism is tricky to identify the smoother a droplet is. To alleviate this shortcoming, the script always computes the reflection line from the final droplet of a sample where the parallelism is more pronounced. We set the reflection line and width to be identical for all droplets within an image, even though in practice some slight variations do occur.

Having computed our core values, the preprocessing script extracts 15 **droplet heights** on either side of the droplet. 11 of these are extracted from evenly-space intervals that are proportional to the width of the droplet, and can therefore be observed as "percentiles" of the sample. The remaining four are concentrated at the edge of the droplet, which observes some of the most significant variance in terms of height. Droplet heights are recorded in pixels.

There are several command-line arguments the script will accept, though only a few are of critical importance. Two options are implemented for the `--mode` the script operates in, `single` and `multi`. Under `single` the sample with the folder name passed through `--dataset` is processed, while under `multi` every sample in the data folder is processed[1].

The arguments `--annotate` and `--crop` are vital to the performance of the script. `--annotate` will re-export each image in a sample with visual references indicating where the script is looking for key aspects of the drop, similar to our display of the positioning of extracted droplet features in

[1] `multi` has not seen as much use as the `single` mode and remains partially deprecated; please verify its performance prior to any important preprocessing.

Figure 2. This can be used to verify the locations of our midpoint, reflection line, etc. `--crop` vertically crops a droplet below the reflection line to reduce the number of black pixels in each image. `--crop` also automatically saves & loads cropped samples from a `cropped/` folder, though it does not overwrite previously-cropped files and it is recommended to routinely clean out the directory.

There is a special third mode of the script used for debugging purposes called `--width_only`. It does not impose a constant width across all droplets and only exports the width and midpoint height to a .csv file.

With each sample processed with the script, two .csv files are produced. The **processed** file averages out the height observations on each side of the droplet for a total of *16* height features. The **raw** file maintains the variances on either side for *31* height features (note that the midpoint is the only height not in a pair).

3 Experiment Script

Our experiments are all run through `ml/main.py` which provides a standardized setup and delegates to the correct experiment function. Some core command-line arguments are presented in Table 1.

argument	description
`--name`	Name of the experiment (used when exporting files)
`--seed`	Ensures reproducibility between runs
`--verbose`	When included, important logging will also be printed to the console. Some visualizations are only exported under verbosity
`--save`	Saves any trained models and other run-specific information
`--load`	Loads any trained DL models with the same experiment name
`--overwrite`	When paired with save, discards existing results in the experiment directory
`--experiment`	Allows the model to delegate to the correct experiment
`--model`	For our baselines/clustering, this allows for the training of a single model opposed to all baselines
`--num_states`	Number of independent runs to average results across

Table 1. General arguments for experiments

It is strongly recommended to run the script while cd'd into the `ml/` directory. In a single script run an experiment directory will be created in `output/experiments` where all files produced during the run including logs, figures, models, arguments etc. are saved. Renaming or replacing files within an experiment folder is generally discouraged, as this may lead to issues when loading or overwriting it. If an experiment is outdated remove it from the experiments directory.

The data we use for experiments are 65 samples that have been pushed through our preprocessing script. Through our clustering experiments we discovered the presence of outliers in our dataset, and therefore provide the following subdivisions of our data:

1. **processsed** Refined set of 61 samples. Still contains several potential outliers. *This set should be used in most circumstances.*
2. **outliers** Collection of 4 samples, 1 from each class, which achieved 100% misclassification rate.
3. **processed_full** Our original data, containing both outliers and verified data.

Before the run is delegated to the selected experiment, a number of checks and setup steps are performed. Logging is initialized, directories created or cleared, and most importantly the data is read and formatted. The format of the data used in experiments depends on several command-line arguments and the experiment chosen. As an example, our time-series baselines and DL models require the data to be formatted as a matrix, while our clustering and baselines requires a flattened feature vector. The argument `--type` must be set to `processed` or `raw` to use either the 16- or 31-feature data. Please refer to the code for use cases of other data processing arguments.

4 Machine Learning

The implementation of our ML models use sk-learn, sktime, and PySwarms. These experiments are less computationally expensive than our DL models and are a good starting point for observing differences in performance between data selection schemes.

4.1 Baselines

We define a set of 6 classification baselines in `ml/models.py`. The models are each defined in a function within the `Baseline` class, which also provides logging, aggregation, and seed support. To add a new model to the baselines, create it within a new function and assign it to a new entry in the `models` function. The hyperparameters for existing models can also be modified in this way, however we have tuned the current implementations to what we observed as optimal.

Model & result saving as well as verbosity-enabled figures can be enabled via command-line in-line with all our experiments. While most figures are generated in `ml/experiments.py`, some information (such as feature importances and decision tree splits) are computed and exported directly in the model functions.

There are two command-line arguments exclusive to our ML baselines (and which are not compatible with the time-series baselines); `--only_acc` disables the export of figures/results which

aren't concerned with model performance, while `--importance` enables the exporting of computed feature importances from the Logistic Regression and Decision Tree baselines. Neither of these are typically needed for a script run.

4.2 Time-series Baselines

We define a set of 6 time-series classification models in `ml/models.py` that are structured identically to our original baselines. The models are implemented in the class `TSBaselines` and run through a separate experiment to the non time-series baselines, as the multivariate time-series data must retain its 2-dimensional shape. Our time-series baselines are, on average, more computationally expensive than our ML baselines; a particular bottleneck is the HIVE-COTE classifier, which could be manually removed if this proves problematic.

4.3 Clustering

We define an agglomerative hierarchical clustering method in `ml/models.py`. The purpose of this experiment is to generate informative visualizations about our data, as opposed to measuring the performance of the clustering method itself. New clustering methods can be added similarly to our ML baselines; our `Clustering` class contains a function to automatically produce a dendrogram for any hierarchical models which requires an assumed structure to the results; please see the implementation for further details.

4.4 PSO

Our PSO experiment was designed to identify which timesteps of a sample's evaporation are most informative. In effect the experiment is an auxiliary feature selection step that is performed independently of any other ML or DL experiment run. We train this method through a hybrid PCA and Clustering-driven loss. An additional output of this experiment is a binary array stored in a NumPy file containing the features of the final particle.

The PSO experiment has its own set of command-line arguments unique to it which enable setting the number of particles, number of iterations, initialization scheme and other attributes of the experiment. These can be reviewed in `ml/main.py`.

5 Deep Learning

We take two approaches for deep learning, a CNN and a Vision Transformer (ViT). Both our models are implemented using PyTorch and running them is mostly identical to our previous experiments with a few additional command-line arguments. These DL models are capable of producing informative visualizations of model performance and are therefore generally preferred over our baselines for interpretability. There are a suite of PyTorch-exclusive command-line arguments used by both our CNN and ViT.

5.1 CNNs

Our CNN models are implemented in `ml/nn.py`. We provide two implementations, a LeNet-based architecture and a fully convolutional model. The fully convolutional model is capable of producing Class Activation Maps (CAMs) which provide an overview of image regions influencing a sample's predicted class.

We provide several command-line arguments which augment PyTorch experiment parameters such as the number of training epochs and the dataset split ratio among others; these are prefixed with `pyt_`. To prevent overloading the user with required parameters for a single run some experiment paramters like the learning rate scheduler must be implemented or altered in the code directly. Like the aforementioned experiments, the CNN models we provide are capable of having their results aggregated. We supplement this with several visualizations of performance and learned features which we will discuss in greater depth in the Ad-hoc visualizations section.

Training and validation are performed in two functions in `ml/experiments.py` which return a log of model performances at the epoch which are returned to the CNN experiment function.

5.2 Vision Transformer

We implement our transformer model in `ml/vit.py` based on the Vision Transformer (ViT) encoder-only architecture with an auxiliary classification network. Our ViT model utilizes the same training and validation functions as our CNN models and follows the same logic with respect to logging and producing figures. Training can be changed via the same command-line arguments as in our other DL experiment, with some exclusive to the ViT which are prefixed with `vit_`. Please refer to the documentation for further information.

During our experiments with ViT models, we observed a balance between model complexity and performance. Even under what we determined to be ideal settings, around 1/3 of all models converged to a poor optima; we call these **fails**. Even minor changes to the architecture and training of the ViT can significantly influence the fail rate. Caution is advised when drawing conclusions from the model as a full experiment run using over >30 seeds will be needed to verify performance.

6 Visualizations

Visualizations are a crucial part of VDLP. The images and GIFs that are produced give insights into the performance of models, importance of features, and trends in the embedding space. The code for visualizations is also the most error-prone and verbose of any in the project, which can make interpretation difficult. We aim to provide a brief overview of some important features of our scripts, and where they may go awry.

6.1 Ad-hoc

We define *ad-hoc* visualization functions in `ml/plots.py`. These are functions which our clustering, PCA and DL experiments use and are of paramount importance to interpreting the results of our models. Some of the figures which can be produced include:

- Attention visualizations for our ViT.
- Plotting sample-wise misclassification rates.
- Confusion matrices.
- Visualizations of droplets in a 2d 'image' space.
- Class Activation Maps.
- Various indicators of model performance, e.g. performance by epoch, overviews over multiple seeds.

The models and format required for each function are unique, and we aim to guide the user by providing assert statements which reject any invalid input for our functions.

6.2 Post-hoc for Baselines

Our *post-hoc* visualizations functions in `visuals/visuals.py` rely on aggregated performance in the format produced by our Baselines models. These functions are the most deprecated part of the project; they are exclusive to the Baseline models and primarily visualize differences in performance at different parameter settings. There are also several functions which produce figures highlighting important features from the DT and LogReg baselines. The file contains a `main` script which will automatically export the visualizations; no command-line arguments are provided and any updates to directories/figures mnust be made directly in the code.