# Final Project for CS 372

*Timothy Ford*

## Algorithm, Application, Language Choice

- *Support Vector Machine (SVM)*
- *Audio Classification*
- *Python 3.9*

## Where It Is Used

A Support Vector Machine (SVM) is a classifier that finds a hyperplane to separate data into their respective classes. It performs well, even with small amounts of data, for nearly any machine learning application that needs to classify two groups of information.

### Other applications

- *Sentiment analysis*
- *Document Classification*

### Alternative algorithms

- *Perceptron*
- *Logistic Regression*

### Reason for choice

*At my startup HomeMetrics, we are very interested in determining whether your water fixtures in your house have a leak. If we play our cards right and build our training data appropriately, we can use a linear classifier to take audio signals recorded near a sink and determine whether the sink is off or has a drip. Given that I am making my own data set, SVM's being effective at small sample sizes sounds exactly like what we need!*

## How Your Project Works

The project comes in 4 parts:

### Data recording:

The first part of the project is a utility script to record the audio. This python script simply records audio form the default microphone of the system it's being run on. There are multiple convenience functions to store this data as .wav files as well as store the raw floating point data in a Pandas Dataframe for later manipulation. The generate_sample configuration in Pycharm will run a script that is a hybrid of the feature engineering portion and the data recording portions of the project and will generate a csv that contains necessary computed features as well as the class. The csv will look as follows:

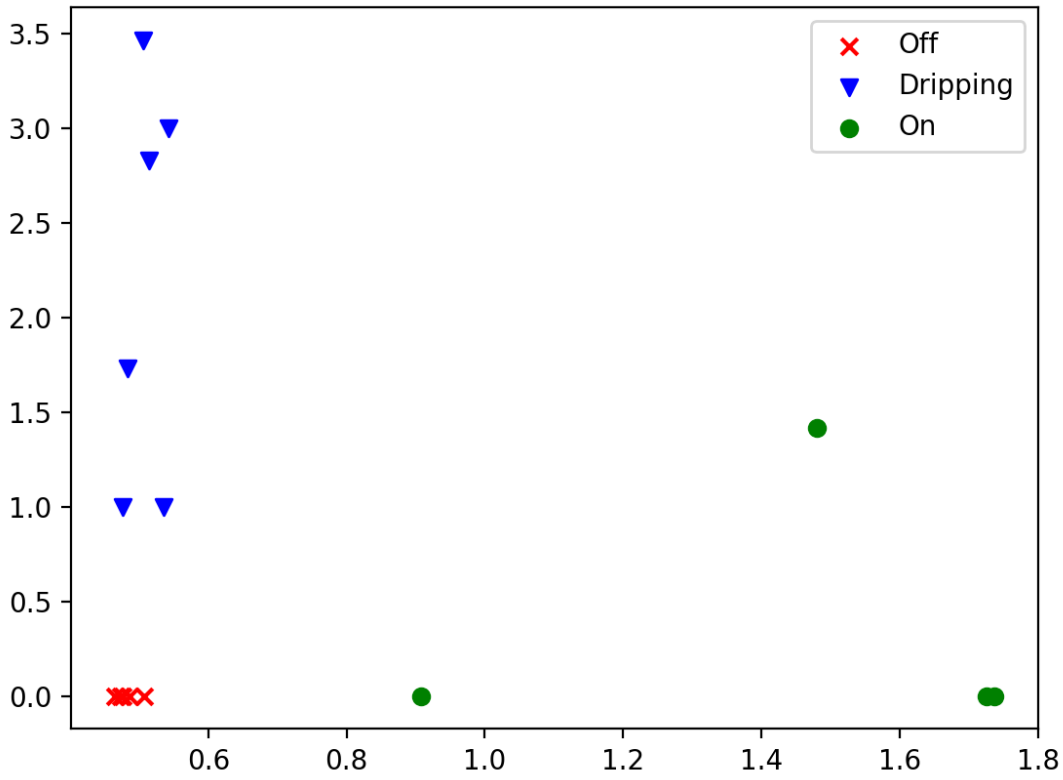
train_water

| | amplitude | peak_count | classification |
|---|---|---|---|
| **0** | 0.21625074744224500 | 0 | OFF |
| **1** | 0.226943239569664 | 0 | OFF |
| **2** | 0.2369009554386140 | 0 | OFF |
| **3** | 0.22363173961639400 | 0 | OFF |
| **4** | 0.2566768527030950 | 0 | OFF |
| **5** | 0.29368454217910800 | 9 | DRIP |
| **6** | 0.26410049200058000 | 8 | DRIP |
| **7** | 0.2561451494693760 | 12 | DRIP |
| **8** | 0.2332666665315630 | 3 | DRIP |
| **9** | 0.22621366381645200 | 1 | DRIP |
| **10** | 0.28727492690086400 | 1 | DRIP |
| **11** | 0.8248866200447080 | 0 | ON |
| **12** | 2.975916862487790 | 0 | ON |
| **13** | 2.1904098987579300 | 2 | ON |
| **14** | 3.015955686569210 | 0 | ON |

This allows us to plot our data on a scatter plot to see how it separates and whether a linear classifier will be suitable to separate the data.

## Feature Engineering

For feature engineering, after poking around Librosa (the Python audio processing library I used), I found the spectrogram to give helpful insight into how I might extract helpful features for my data. It seemed like average overall amplitude was a pretty tell-tale sign of the sink being on and a detection of tempo seemed to indicate a drip. I used a Short-Time Fourier Transform (STFT) and then broke it down into magnitude/phase to get the amplitude and used the tempo gram to determine the drips. In order to make the tempogram successful, my measure ended up being the number of peaks that a peak-counting algorithm could count on the tempogram. I then transformed the data by a square root so that it looked like it separated slightly more naturally (it was oddly spread out before). The separation ends up looking like this:




## Classifying with the SVM

Next we need to classify our data using an SVM. The SVM has a couple parts that make it work. In the end, it's really just a linear combination of weights, one per feature, and an intercept with a penalty function that penalizes it when misclassifies a data point. You run it against a training data set to determine 'appropriate' weights and then, you can enter in values and predict what class they might be in.

```
def __init__(self, epochs, rate=0.0001):
    self.epochs = epochs
    self.rate = rate
    self.current_epoch = 1
    self.w1 = np.float64(0)
    self.w2 = np.float64(0)
    self.b = np.float64(0)
```

It runs for a certain number of times, known as epochs, and attempts to generate an optimal hyperplane (one that maximizes the margins between the support vectors, or those points closest to the 'edge' of the classification), by generating a predicted set of values from a linear combination of it's weights and intercept and using a loss function to update the weights/intercept based on the error of that predicted set.

```
def fit(self, data, classes):
    for epoch in range(self.current_epoch, self.epochs):
        self.current_epoch = epoch
        y_pred = (self.w1 * data[:, 0] + self.w2 * data[:,
1] + self.b) * classes
```

The loss function I used was the Hinge Loss function, which is intended to be used as a 'maximum-margin' loss function. When the value predicted is correctly classified, the weights are only updated with the regularization parameter scaled by the learning rate. However, if the value is misclassified, the loss function uses gradient descent to optimize the classification and update the weights using the class value, data values, as well as the learning rate and regularization parameter.

```
for index, value in enumerate(y_pred):
    if value < 1:
        m1_deriv += data[index, 0] * classes[index]
        m2_deriv += data[index, 1] * classes[index]
        b_deriv += classes[index]
self.w1 += self.rate * (m1_deriv - 2 *
self.regularization() * self.w1)
self.w2 += self.rate * (m2_deriv - 2 *
self.regularization() * self.w2)
self.b += self.rate * (b_deriv - 2 * self.regularization()
* self.b)
```

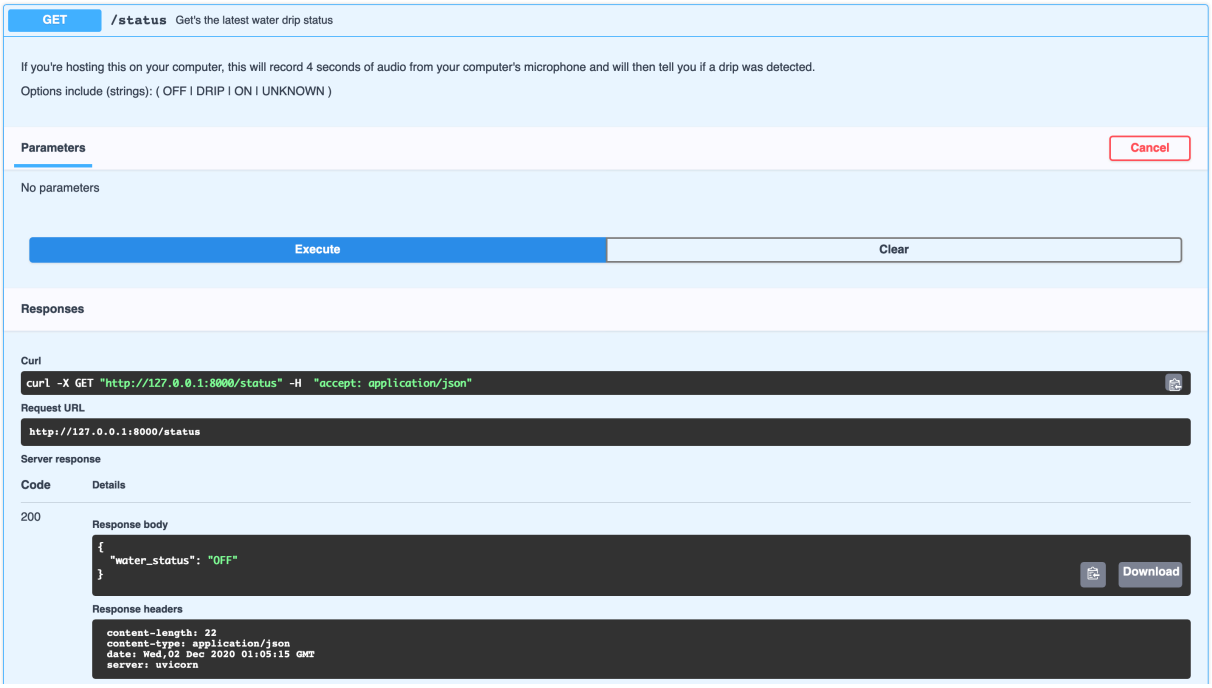
The regularization parameter prevents the model from overfitting by slowing the magnitude of change allowed the longer the model has been training.

```
def regularization(self) -> float:
    return 1 / self.current_epoch
```

## Running the Web Server

The final portion of the project is the webserver. This is just a quick and dirty FastAPI server that exposes 2 endpoints, a '/root' endpoint which doesn't do anything particularly interesting, and a '/status' endpoint, that uses the system default mic and the trained SVM drip/on models from the training datasets I've already generated to give a classification for current conditions.

When you hit the '/status' endpoint, it's a GET request, it will record audio data from the mic for a predetermined amount of time (4 seconds at the moment), and then will run the two models to detect if the just recorded data has classified to be dripping or on. It will then return a simple JSON blob with the current water state.




It is self-documented using OpenAPI docs. If you want to mess around with it, use the "run_fastapi" Pycharm configuration and navigate to http://127.0.0.1:8000/docs#/default/get_water_status_status_get.

# Run time

This paper discusses SVM run time and comes up with a general O(n^2) (https://www.csie.ntu.edu.tw/~cjlin/papers/libsvm.pdf). However, I believe my implementation runs in O(epochs * len(dataset)) since my SVM model has a for loop that iterates for epoch number of times and, for each epoch, adjusts the weights for each value in the provided training dataset.

# Program usage or README

Code is hosted and documented here:

https://github.com/timwford/svm_audio_classification

# References


https://monkeylearn.com/blog/introduction-to-support-vector-machines-svm/

https://en.wikipedia.org/wiki/Linear_classifier

https://sites.google.com/site/machinelearningnotebook2/classification/binary-classification/linear-classifiers

https://scikit-learn.org/stable/modules/svm.html

https://en.wikipedia.org/wiki/Loss_function

https://en.wikipedia.org/wiki/Hinge_loss

https://towardsdatascience.com/support-vector-machine-introduction-to-machine-learning-algorithms-934a444fca47

https://web.mit.edu/6.034/wwwbob/svm.pdf

https://en.wikipedia.org/wiki/Support_vector_machine

https://www.csie.ntu.edu.tw/~cjlin/papers/libsvm.pdf