Lab 8: Support Vector Machines

Due Thursday by 11:59pm **Points** 5 **Submitting** an external tool
Available Oct 19 at 12:01am - Nov 19 at 11:59pm about 1 month

Lab 8: Support Vector Machines

To complete Lab 8, download: lab8.zip.

Contents

The More, the Merrier

So far in 6.034, we've discussed a few different supervised machine learning algorithms:

- **k-nearest neighbors (kNN)**, which classify points based on which training points are nearby
- **identification trees (ID trees)**, which classify points using a tree-based exploration
- **neural networks (NN)**, which classify points by iteratively applying small, primitive mathematical operations on features

The Support Vector Machine (SVM) is yet another supervised machine learning algorithm. An SVM classifies a point by, conceptually, comparing it against the most "important" training points, which are called the *support vectors*. The support vectors of classification *C* which are most similar to **x** win the vote, and **x** is consequently classified as *C*. In this way, an SVM can be likened to doing nearest neighbors comparison.

Importantly, SVMs can be very robust, as the principle paradigm of the SVM algorithm is that *the classifier always maximizes the margin between the differently-classified data points*. Visually, if you imagine drawing a decision boundary in a plane to separate two different classifications of data, an SVM will always position the decision boundary in such a way as to maximize the distance to the closest training points.

You might find the following diagram (inspired by Robert McIntyre's notes) helpful as a visual reference throughout this lab

SvmDiagram.jpg

The Support Vector Machine

An SVM is a numeric classifier. That means that all of the features of the data must be *numeric*, not symbolic. Furthermore, in this class, we'll assume that the SVM is a *binary* classifier: that is, it classifies points as one of two classifications. We'll typically call the classifications "+" and "-".

A trained SVM is defined by two values:

- A normal vector **w** (also called the weight vector), which solely determines the shape and direction of the decision boundary.
- A scalar offset **b**, which solely determines the position of the decision boundary with respect to the origin.

A trained SVM can then classify a point **x** by computing $w \cdot x + b$. If this value is positive, **x** is classified as +; otherwise, **x** is classified as -.

The decision boundary is coerced by support vectors, so called because these vectors (data points) *support* the boundary: if any of these points are moved or eliminated, the decision boundary changes! All support vectors lie on a *gutter*, which can be thought of as a line running parallel to the decision boundary. There are two gutters: one gutter hosts positive support vectors, and the other, negative support vectors.

Note that, though a support vector is always on a gutter, it's **not** necessarily true that every data point on a gutter is a support vector.

Equation Reference

Below are the five principle SVM equations, as taught in lecture and recitation. Equations 1-3 define the decision boundary and the margin width, while Equations 4 and 5 can be used to calculate the alpha (supportiveness) values for the training points.

$$[\text{Eq } 1] \quad \vec{w} \cdot \vec{x} + b = 0$$

$$[\text{Eq } 2] \quad \text{margin width} = \frac{2}{||\vec{w}||}$$

$$[\text{Eq } 3] \quad \vec{w} \cdot \vec{x_i} + b = y_i \text{ where } y_i \text{ is the classification (+1 or -1)}$$

$$[\text{Eq } 4] \quad \sum_{\substack{\forall\, training \\ points\ i}} y_i\ \alpha_i = 0$$

$$[\text{Eq } 5] \quad \vec{w} = \sum_{\substack{\forall\, training \\ points\ i}} y_i\ \alpha_i\, \vec{x}_i$$

For more information about how to apply these equations, see:

- Dylan's guide to solving SVM quiz problems (http://web.mit.edu/dxh/www/svm.html)
- Robert McIntyre's SVM notes (http://web.mit.edu/dxh/www/rlm-svm-notes.pdf)

- SVM notes (https://ai6034.mit.edu/wiki/images/SVM_and_Boosting.pdf) , from the **Reference material and playlist**

Part 1: Vector Math

We'll start with some basic functions for manipulating vectors. For now, we will represent an *n*-dimensional vector as a list or tuple of *n* coordinates. `dot_product` should compute the dot product of two vectors, while `norm` computes the length of a vector. Implement both functions.

(If these look familiar, it's probably because you saw them on Lab 5. Feel free to copy your implementations from your `lab8.py`.)

```
def dot_product(u, v):
```

```
def norm(v):
```

Later, when you need to actually manipulate vectors, note that we have also provided methods for adding two vectors (`vector_add`) and for multiplying a scalar by a vector (`scalar_mult`). (See the **API**, below.)

Part 2: Using the SVM Boundary Equations

Equations 1 and 3 above describe how certain data points must interact with the SVM decision boundary and gutters.

We will start by leveraging Equation 1 to classify a point with a given SVM (ignoring the point's actual classification, if any). Per Equation 1,

- a point's classification is +1 when `w·x + b > 0`
- a point's classification is -1 when `w·x + b < 0`

If `w·x + b = 0`, the point lies on the decision boundary, so its classification is ambiguous.

First, implement `positiveness(svm, point)` which evaluates the expression `w·x + b` to calculate how positive a point is. The first argument, `svm`, is a **`SupportVectorMachine` object**, and `point` is a **`Point` object**.

```
def positiveness(svm, point):
```

Next, classify a point as +1 or -1. If the point lies on the boundary, return 0.

```
def classify(svm, point):
```

Next, use the SVM's current decision boundary to calculate its margin width (Equation 2):

```
def margin_width(svm):
```

Finally, we will check that the gutter constraint is satisfied with the SVM's current support vectors and boundary. The gutter constraint imposes two restrictions simultaneously:

1. The positiveness of a positive support vector must be +1, and the positiveness of a negative support vector must be -1.
2. No training point may lie strictly between the gutters. That is, each training point should have a positiveness value indicating that it either lies on a gutter or outside the margin.

Note that the gutter constraint does not check whether points are classified *correctly*: it just checks that the gutter constraint holds for the current assigned classification of a point.

Implement `check_gutter_constraint`, which should return a set (not a list) of the training points that violate one or both conditions:

```
def check_gutter_constraint(svm):
```

Part 3: Supportiveness

To train a support vector machine, every training point is assigned a supportiveness value (also known as an alpha value, or a Lagrange multiplier), representing how important the point is in determining (or "supporting") the decision boundary. The supportiveness values must satisfy a number of conditions, which we will explore below.

Supportiveness is never negative, and is 0 for non-support-vectors

First, implement `check_alpha_signs` to ensure that each point's supportiveness value satisfies two conditions:

1. Each training point should have a non-negative supportiveness.
2. Each support vector should have positive supportiveness, while each non-support vector should have a supportiveness of 0.

Note: each point's supportiveness value can be accessed a property of the point. See the **API** for more information.

This function, like `check_gutter_constraint` above, should return a set of the training points that violate either of the supportiveness conditions.

```
def check_alpha_signs(svm):
```

Supportiveness values must be internally consistent

Implement `check_alpha_equations` to check that the SVM's supportiveness values are:

1. consistent with its boundary equation, according to Equation 4, and
2. consistent with the classifications of its training points, according to Equation 5.

This function should return `True` if both Equations 4 and 5 are satisfied, otherwise `False`. Remember the `vector_add` and `scalar_mult` helper functions are given to you in the **API** if you'd like to use them in your implementation.

```
def check_alpha_equations(svm):
```

Part 4: Evaluating Accuracy

Once a support vector machine has been trained -- or even while it is being trained -- we want to know how well it has classified the training data. Write a function that checks whether the training points were classified correctly and returns a set containing the training points that were misclassified, if any. Hint: You might find it helpful to use a function that you perviously defined.

```
def misclassified_training_points(svm):
```

Part 5: Training a support vector machine

So far, we have seen how to calculate the final parameters of an SVM (given the decision boundary), and we've used the equations to assess how well an SVM has been trained, but we haven't actually attempted to train an SVM.

In practice, training an SVM is a hill-climbing problem in alpha-space using the Lagrangian. There's a bit of math involved, and the equations are typically solved using Sequential Minimal Optimization (SMO), a type of quadratic programming (which is similar to linear programming, but more complicated).

...but if that sounds scary, don't worry -- we've provided code to do most of it for you!

What `train_svm` does

In train_svm.py, we've provided some SVM-solving code, originally written by past 6.034 student Crystal Pan. Here is a very high-level pseudocode overview of what the function `train_svm` (in train_svm.py) does:

```
while (alphas are still changing) and (iteration < max_iter):
    for i in training_points:
        for j in training_points:
            Update i.alpha and j.alpha using SMO to minimize ||w|| (i.e. maximize the margin w
idth)

        Update the SVM's w, b, and support_vectors (using a function you'll write)
        Update the displayed graph using display_svm.py

Print the final decision boundary and number of misclassified points
Return the trained SVM
```

We've also provided visualization code in `display_svm.py`, originally written by past 6.034 student Kelly Shen. This code is automatically called by `train_svm`, although you can adjust the parameters (or disable graphing) by calling `train_svm` with **additional arguments**.

Note that the visualization requires the Python module Matplotlib. If you don't currently have Matplotlib, **refer to lab 6 for instructions on how to get it installed (https://ai6034.mit.edu/wiki/index.php?title=Lab_6#What_if_I_don.27t_have_Matplotlib_and_NumPy.3F)** .

If you still can't get Matplotlib to work, you can also disable visualization by commenting out the line

```
from display_svm import create_svm_graph</tt>
```

in `train_svm.py` and running `train_svm` with the argument `show_graph=False`. (If you do this, however, you may need to add print statements in order to answer some of the multiple-choice questions below: specifically, Questions 2 and 5-8. Also, you won't get to watch the SVM being trained!)

Your task: Update SVM from alpha values

Your task is to take the alpha values determined by SMO and use them to determine the support vectors, the normal vector **w**, and the offset **b**:

- Any training point with alpha > 0 is a support vector.
- `w` can be calculated using Equation 5.
- If training is complete, `b` can be calculated using the gutter constraint (Equation 3). However, during training, the gutter constraint will produce different values of `b` depending on which support vector is used! To resolve this ambiguity, we will instead take the average of two values: the *minimum* value of `b` produced by a *negative* support vector, and the *maximum* value of `b` produced by a *positive* support vector. (Can you figure out why?)

Implement the function `update_svm_from_alphas`, which takes in a `SupportVectorMachine`, then uses the SVM's training points and alpha values to update `w`, `b`, and `support_vectors`. This function should return the updated SVM. (If the input SVM already has `w`, `b`, and/or `support_vectors` defined, ignore them and overwrite them. For this function, you may assume that the SVM is 2-dimensional and that it has at least one training point with alpha > 0.)

Important: Do NOT use `svm.copy()` in this function or instantiate a new SVM using the `SupportVectorMachine` constructor, as training (and the test cases) will likely fail.

```
def update_svm_from_alphas(svm):
```

`train_svm` will call `update_svm_from_alphas`, so once you've implemented it, you should be able to train an SVM on a dataset and visualize the results!

Arguments for `train_svm`

The function `train_svm` has only one **required** argument:

- **`training_points`**: A list of training points as `Point` objects.

`train_svm` also has many optional arguments (with defaults given):

- **`kernel_fn`**`=dot_product`: The kernel function (a `function` object) to be used in the SVM. Currently, the visualization only supports linear kernels (i.e. it can only draw straight lines).
- **`max_iter`**`=500`: The maximum number of iterations to perform (an `int`). Each iteration consists of considering every pair of training points. (So if there are *n* training points, one iteration considers up to n^2 pairs of training points. Note that the visualization can update up to *n* times *per* iteration, not just once per iteration.)
- **`show_graph`**`=True`: Boolean value indicating whether to display a graph of the SVM.
- **`animate`**`=True`: Boolean value indicating whether to display updates on the SVM graph during training (only applies if `show_graph` is `True`).
- **`animation_delay`**`=0.5`: Number of seconds to delay between graph updates (only applies if both `show_graph` and `animate` are `True`).
- **`manual_animation`**`=False`: Boolean value indicating whether to pause execution after each animation update (only applies if `show_graph` is `True`). If `True`, you will need to manually close the graph window after each update. This option may be used as a work-around if normal animation isn't working on your machine.

We have also provided five sample datasets in `svm_data.py`:

- `sample_data_1` and `sample_data_2`, drawn with ASCII art near the top of `train_svm.py`
- `recit_data`, the dataset from recitation and from **Robert McIntyre's notes (http://web.mit.edu/dxh/www/rlm-svm-notes.pdf)** (although Robert labels the points A, B, C, D, whereas here they are called A, B, D, E)
- `harvard_mit_data`, the Harvard/MIT data from **2014 Quiz 3, Problem 2, Part B (http://courses.csail.mit.edu/6.034f/Examinations/2014q3.pdf)**

- `unseparable_data`, the Harvard/MIT data with an additional MIT point at (4,4) that makes the data not linearly separable

How to run `train_svm`

Note that there is currently no command-line interface for `train_svm`, so if you use a command line, you can either change the parameters in `train_svm.py` and re-run it multiple times, or you can load it into an interactive Python shell by running the command `python3` in a command line and then (in the Python shell) `from train_svm import *`.

If you just run `train_svm.py`, it will automatically run whatever functions are called at the bottom of the file (by default, training on `sample_data_1`). You can comment/uncomment the functions, change the arguments, and add additional functions.

In an interactive Python shell, you can call the function `train_svm(my_data, ...)` on the various datasets, using the **arguments described above**.

Part 6: Multiple Choice Questions about SVM Training

Questions 1-4: Running `train_svm`

For Questions 1-4, fill in the appropriate `ANSWER_n` with an integer.

> Note: Adjusting the animation delay or other arguments above may be useful for answering some of these questions.

Question 1

Try training an SVM on the Harvard/MIT data from **2014 Quiz 3 (http://courses.csail.mit.edu/6.034f/Examinations/2014q3.pdf)** (`harvard_mit_data`). How many iterations does it take?

Question 2

During training for the Harvard/MIT data, what is the maximum number of support vectors shown on the graph at once? (Assume that all training points with alpha > 0, displayed as circled points, are support vectors.)

Note: If you are using `show_graph=False`, one way to keep track of the number of support vectors is to record `svm.support_vectors` at every (graphical) iteration of training. See line 144 of `train_svm.py` for an example of where you can record the current set of support vectors in each iteration.

Question 3

After training for the Harvard/MIT data, how many support vectors are there?

Question 4

Try training an SVM on the recitation dataset (`recit_data`). How many iterations does it take?

Questions 5-10: Identifying points

For Questions 5-10, consider what happens as the SVM trains on the recitation data, which consists of four points: A(1,3), B(1,1), D(2,2), and E(3,2). (Note that this is different from the labeling in Robert McIntyre's notes -- he uses the same four points, but labeled A, B, C, D.)

If you are using `show_graph=False`, you may want to put a print statement near the SVM graph update line in `train_svm`. See line 144 of `train_svm.py` for an example of where you can put such a print statement.

For each question, fill in the appropriate `ANSWER_n` with a list of point names, selected from A, B, D, E (e.g. `['A', 'B', 'E']`).

Question 5

When a boundary **first appears** on the graph, which points **are support vectors**?

Question 6

When a boundary **first appears** on the graph, which training points **appear to lie on the gutters**?

Question 7

When a boundary **changes** on the graph (i.e. the second boundary that appears), which points **are support vectors**?

Question 8

When a boundary **changes** on the graph (i.e. the second boundary that appears), which points **appear to lie on the gutters**?

Question 9

When **training is complete**, which points **are support vectors**?

Question 10

When **training is complete**, which points **appear to lie on the gutters**?

Questions 11-16: True/False

Answer the following questions about SVMs *in general*, assuming that the data is linearly separable. (You may want to consider the provided datasets as examples.)

For each question, fill in the appropriate `ANSWER_n` with `True` or `False`.

Question 11
During training, all support vectors lie on the gutters.

Question 12
After training, all support vectors lie on the gutters.

Question 13
During training, all points on the gutters are support vectors.

Question 14
After training, all points on the gutters are support vectors.

Question 15
During training, no points can lie between the gutters (i.e. in the margin).

Question 16
After training, no points can lie between the gutters (i.e. in the margin).

Questions 17-19: General Multiple Choice

Answer the following questions about SVMs *in general*, assuming that the data is linearly separable. (You may want to copy and modify one of the provided datasets to experimentally determine the answers.)

For each question, fill in the appropriate `ANSWER_n` with a list of all answers that apply (e.g. `[1, 3]`), from the following choices:

1. The decision boundary may change.
2. The decision boundary may stay the same.
3. The margin width may decrease.
4. The margin width may increase.
5. The margin width may stay the same.
6. The number of support vectors may decrease.
7. The number of support vectors may increase.
8. The number of support vectors may stay the same.

Question 17
If you start with a trained SVM and move one of the support vectors *directly toward* the decision boundary (i.e. in a direction perpendicular to the decision boundary and moving closer to the boundary), then retrain, what could happen?

Question 18

If you start with a trained SVM and move one of the support vectors *directly away from* the decision boundary (i.e. in a direction perpendicular to the decision boundary and moving away from the boundary), then retrain, what could happen?

Question 19

If you start with a trained SVM and move one of the support vectors *along its gutter* (parallel to the decision boundary), then retrain, what could happen?

Question 20: And if the data is NOT linearly separable...

Question 20

What does our SVM trainer do if the data is *not* linearly separable? (If you're not sure, try training on `unseparable_data`.)

Fill in `ANSWER_20` with the **one** best answer as an `int` (e.g. `3`):

1. It identifies outlier points and systematically eliminates them from the data to avoid overfitting.
2. It gradually relaxes constraints until a solution is found.
3. It determines that the data is not separable and raises an exception or returns None.
4. It determines that the data is not separable, so it instead returns the best available solution.
5. It continues attempting to find a solution until training terminates because the alpha values are no longer changing, due to convergence.
6. It continues attempting to find a solution until it times out by reaching the maximum number of iterations.

If you want to do more...

If you want to learn more about training SVMs, the following resources (mostly beyond the scope of 6.034) may be helpful:

- **SVM lecture notes (http://cs.nyu.edu/~dsontag/courses/ml12/slides/lecture5.pdf)** from a machine-learning class at NYU
- **Paper on Sequential Minimal Optimization (http://www-ai.cs.uni-dortmund.de/LEHRE/SEMINARE/SS09/AKTARBEITENDESDM/LITERATUR/PlattSMO.pdf)**
- **SVM notes (https://ai6034.mit.edu/wiki/images/SVM_and_Boosting.pdf)**, from the **Reference material and playlist**
- **SVM slides (http://courses.csail.mit.edu/6.034f/ai3/SVM.pdf)**, from the **Reference material and playlist**
- Wikipedia articles on **SVM math (https://en.wikipedia.org/wiki/Support_vector_machine#Linear_SVM)** and **Sequential Minimal Optimization (https://en.wikipedia.org/wiki/Sequential_minimal_optimization)**

Possible extensions of this lab include:

- Training on other datasets, such as ones from past quiz problems or from the internet
- Defining some non-linear kernel functions and extend train_svm to handle them (primarily by adding a `kernel_fn` argument to your `update_svm_from_alphas`), then training with the alternate kernels
- Extending the visualization code to display non-linear kernels
- Extending `train_svm` to handle 1-D, 3-D, or higher-dimensional SVMs

If you do something cool, we'd love to see it! Feel free to send your code and/or results to 6.034-2018-staff@mit.edu (ideally with some sort of documentation). Your code could even end up in a future version of this lab! (With your permission, of course.)

API

The file `svm_api.py` defines the `Point` and `SupportVectorMachine` classes, as well as some helper functions for vector math, all described below.

`Point`

A `Point` object has the following attributes:

`name`
The name of the point (a string).

`coords`
The coordinates of the point, represented as a vector (a tuple or list of numbers).

`classification`
The true classification of the point, if known, as a number, typically +1 or -1. Initialized by default to None.

`alpha`
The supportiveness (alpha) value of the point, if assigned.

The `Point` class supports iteration, to iterate through the coordinates.

`SupportVectorMachine`

A `SupportVectorMachine` is a classifier that uses a decision boundary to classify points. The decision boundary is defined by a normal vector `w` (a vector, represented as a list or tuple of coordinates), and an offset `b` (a number). The SVM also has a list of training points and optionally a list of support vectors. You can access its parameters using these attributes:

`w`
The normal vector **w**.

`b`
The offset **b**.

`training_points`
A list of `Point` objects with known classifications.

`support_vectors`
A list of `Point` objects that serve as support vectors for the SVM. Every support vector is also a training point.

To set an SVM's decision boundary, you can modify the parameters `w` and `b` directly, or use...

`set_boundary(new_w, new_b)`
Updates `w` and `b`, returning the modified SVM.

To instantiate a new SVM, use the constructor:

```
my_svm = SupportVectorMachine(w, b, training_points, support_vectors)
```

The `SupportVectorMachine` class also has a `.copy()` method for making a deep copy of an SVM.

Helper functions for vector math

`vector_add(vec1, vec2)`
Given two vectors represented as iterable vectors (lists or tuples of coordinates) or `Points`, returns their vector sum as a list of coordinates.

`scalar_mult(c, vec)`
Given a constant scalar and an iterable vector (as a tuple or list of coordinates) or a `Point`, returns a scaled list of coordinates.

Survey

Please answer these questions at the bottom of your lab file:

- `NAME`: What is your name? (string)
- `COLLABORATORS`: Other than 6.034 staff, whom did you work with on this lab? (string, or empty string if you worked alone)

- `HOW_MANY_HOURS_THIS_LAB_TOOK`: Approximately how many hours did you spend on this lab? (number or string)
- `WHAT_I_FOUND_INTERESTING`: Which parts of this lab, if any, did you find interesting? (string)
- `WHAT_I_FOUND_BORING`: Which parts of this lab, if any, did you find boring or tedious? (string)
- (optional) `SUGGESTIONS`: What specific changes would you recommend, if any, to improve this lab for future years? (string)

(We'd ask which parts you find confusing, but if you're confused you should really ask a TA.)

Submit Programming Assignment

Upload all files for your submission

SUBMISSION METHOD

[✓ Upload] GitHub Bitbucket

DRAG & DROP

Any file(s) including .zip. Click to browse.