COMP0088
Introduction to Machine Learning
Lab Assignment 4

Matthew Caldwell

October 20, 2021

Task Summary

Week 4 Notes

In contrast to labs 2 & 3, this week's exercises are more concerned with how the learning algorithms behave than with the nuts and bolts of implementation. If you are especially enthusiastic there are **Further exploration** tasks on implementing support vector machines, but otherwise we will just use a standard implementation from `scikit-learn`. Instead, you are invited to generate data that will probe the model behaviour.

The assignment script is `week_4.py`, generating output file `week_4.pdf`. There are some potentially-useful command line options, which you can list using:

```
$ python week_4.py -h
```

These include options for setting the `gamma` and `C` hyperparameters for SVM fitting.

Example output from a completed assignment is shown in Fig. 1.

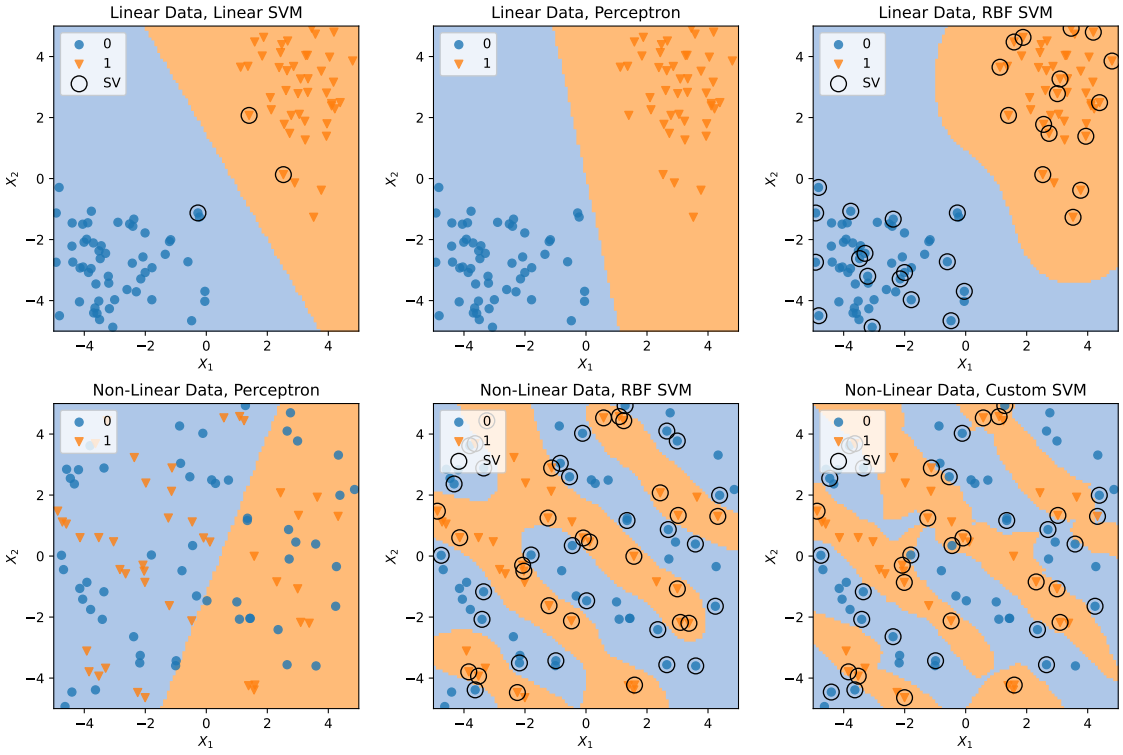

Figure 1: Example output from the `week_4.py` script

1 Linear SVM

As discussed in the lectures, a linear SVM fits a decision boundary to maximise the **geometric margin** between that boundary and the worst case (i.e., nearest) training samples in each class. For a **hard margin** classifier, the classes must be **linearly separable**, whereas a **soft margin** SVM can allow misclassifications to an extent governed by a **cost** hyperparameter, C.

The geometric margin for a sample $(\mathbf{x}, y)$ is defined as:

$$M_g = \frac{y(\mathbf{x} \cdot \mathbf{w} + b)}{\|\mathbf{w}\|} \tag{1}$$

where $\mathbf{w}$ and b are the weights and bias defining the boundary, and y uses the $\{-1, +1\}$ labelling convention. The geometric margin for a whole dataset is the minimum of the geometric margins for all samples in the set.

1.1 Generate more realistic binary classification data

In the exercises for week 1 you were asked to generate linearly separable data, but the generating model was quite unrealistic—most data is unlikely to exhibit such a tight and consistent boundary, with neither errors nor any appreciable margin between the classes. Such data doesn't provide much intuition for SVM behaviour.

Implement the body of the following function in `week_4.py` to provide a more realistic source of linearly separable data for which there actually is some separation between the classes.

```
def generate_margined_binary_data ( num_samples, count, limits, rng ):
    """
    Draw random samples from a linearly-separable binary model
    with some non-negligible margin between classes. (The exact
    form of the model is up to you.)

    # Arguments
        num_samples: number of samples to generate
            (ie, the number of rows in the returned X
            and the length of the returned y)
        count: the number of feature dimensions
        limits: a tuple (low, high) specifying the value
            range of all the features x_i
        rng: an instance of numpy.random.Generator
            from which to draw random numbers

    # Returns
        X: a matrix of sample vectors, where
            the samples are the rows and the
            features are the columns
            ie, its size should be:
              num_samples x count
        y: a vector of num_samples binary labels
    """
    # TODO: implement this
    return None, None
```

You are free to devise any generating model you like as long as it produces some non-negligible margin between the classes. If you have the time and inclination, it might be worth trying out more than one formulation. If you are short on inspiration, feel free to ask the TAs for suggestions. The data you generate here will be used for the top row of plots in the output file.

Note that the returned label vector should continue to use the same $\{0, 1\}$ labelling convention as in previous weeks.

1.2 Calculate the geometric margin of a linear decision boundary

Implement the body of the following function in `week_4.py`:

```
def geometric_margin ( X, y, weights, bias ):
    """
    Calculate the geometric margin for a given
    dataset and linear decision boundary. May be
    negative if any of the samples are
    misclassified.

    # Arguments
        X: an array of sample data, where rows are samples
           and columns are features.
        y: vector of ground truth labels for the samples,
           must be same length as number of rows in X
        weights: a vector of weights defining the direction
           of the decision boundary, must be the same
           length as the number of features
        bias: scalar intercept value specifying the position
           of the boundary

    # Returns:
        g: the geometric margin -- ie, the minimum distance
           of any of the samples on the correct side of the
           boundary (or the negative greatest distance on the
           wrong side)
    """
    assert(X.shape[0] == len(y))
    assert(X.shape[1] == len(weights))

    # TODO: implement this
    return None
```

Note that the supplied label vector y will use the $\{0, 1\}$ labelling convention, not $\{-1, +1\}$. You will need to account for this in your calculations.

2 Perceptron

The perceptron is a simple single-layer linear model for binary classification. Its decision function is exactly the linear model boundary decision function described in Task 2 of Lab 1.1:

$$\hat{y} = \begin{cases} 1 & \text{if } \mathbf{w} \cdot \mathbf{x} \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

The model is trained iteratively, one training sample at a time, updating the weights at each step as:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha(y - \hat{y})\mathbf{x} \tag{3}$$

where α is the learning rate. Training continues until there are no errors (and hence no updates) for the whole training set. Noting that $\nabla_{\mathbf{w}}(\mathbf{w} \cdot \mathbf{x}) = \mathbf{x}$ and $(y - \hat{y})$ is the direction of any error, we can see that this update rule is effectively stochastic gradient descent on the **margin** of misclassified samples.

The perceptron is guaranteed to converge for linearly separable data, though time to converge depends on the margin. The boundary may be non-optimal and depends on the order in which samples are presented. It is guaranteed *never* to converge when the data are not linearly separable—there will always be an error somewhere.

2.1 Learn a decision boundary with the perceptron algorithm

Implement the body of the following function in `week_4.py`:

```
def perceptron_train ( X, y, alpha=1, max_epochs=50, include_bias=True ):
    """
    Learn a linear decision boundary using the
    perceptron algorithm.

    # Arguments
        X: an array of sample data, where rows are samples
            and columns are features.
        y: vector of ground truth labels for the samples,
            must be same length as number of rows in X
        alpha: learning rate, ie how much to adjust the
            boundary for each misclassified sample
        max_epochs: maximum number of passes over the
            training set before admitting defeat
        include_bias: whether to automatically add a
            a constant bias feature x0

    # Returns:
        weights: vector of feature weights defining the
            decision boundary, either same length as number of
            columns in X or 1 greater if include_bias is True.
            (note that a weights vector will be returned even if
            the algorithm fails to converge)
    """
    assert(X.shape[0] == len(y))
```

```
    # TODO: implement this
    return None
```

As mentioned in the docstring, you should return the final weights vector even if the algorithm fails to converge.

Input labels will be $\{0, 1\}$.

2.2 Predict binary labels from the learned boundary

Implement the body of the following function in `week_4.py`:

```
def perceptron_predict ( test_X, weights ):
    """
    Predict binary labels for a dataset using a specified
    decision boundary. (This is intended for us with a boundary
    learned with the perceptron algorithm, but any suitable
    weights vector can be used.)

    # Arguments
        test_X: an array of sample data, where rows are samples
            and columns are features.
        weights: vector of feature weights defining the
            decision boundary, either same length as number of
            columns in X or 1 greater -- in the latter case it
            is assumed to contain a bias term, and test_X will
            have a constant term x0=1 prepended

    # Returns
        pred_y: a vector of predicted binary labels
            corresponding to the samples in test_X

    """
    assert(test_X.shape[1] in (len(weights),len(weights)-1))

    # TODO: implement this
    return None
```

The return predictions should be in $\{0, 1\}$.

3 Non-Linear Data

The behaviour of linear classifiers on linearly-separable data is relatively uncontentious. It's in the face of data that is not linearly separable that things become interesting.

Notoriously, the perceptron algorithm will fail outright in such cases. A linear SVM will find the best boundary it can, but can still only draw a linear boundary so must concede errors somewhere. But by implicitly expanding into a much higher dimensional basis, a kernel SVM can fit nearly any kind of data.

3.1 Generate non-linear binary data

Implement the body of the following function in `week_4.py`:

```
def generate_binary_nonlinear_2d ( num_samples, limits, rng ):
    """
    Draw random samples from a binary model that is *not*
    linearly separable in its 2D feature space. (The exact
    form of the model is up to you.)

    # Arguments
        num_samples: number of samples to generate
            (ie, the number of rows in the returned X
            and the length of the returned y)
        limits: a tuple (low, high) specifying the value
            range of all the features x_i
        rng: an instance of numpy.random.Generator
            from which to draw random numbers

    # Returns
        X: a matrix of sample vectors, where
            the samples are the rows and the
            features are the columns
            ie, its size should be:
              num_samples x count
        y: a vector of num_samples binary labels
    """
    # TODO: implement this
    return None, None
```

You are free to devise any generating model you like, and it is definitely worth trying out more than one candidate. Once again, feel free to ask the TAs for suggestions if you're short on inspiration. The data you generate here will be used for the bottom row of plots in the output file.

As in the previous tasks, the returned label vector should continue to use the $\{0, 1\}$ labelling convention.

4 Kernel Functions

The SVM implementation in `scikit-learn` (specifically `sklearn.svm.SVC`) has in-built (and optimised) support for the commonly-used **radial basis** and **polynomial** kernels, but also allows for arbitrary kernels to be used by supplying a function that computes the Gram matrix.

NB: generating and using a custom kernel matrix can be very slow for large datasets, so I'd recommend keeping `num_samples` small for this task, especially while hacking around with it.

4.1 Implement a custom kernel function

Implement the body of the following function in `week_4.py`:

```
def custom_kernel ( X1, X2, gamma=0.5 ):
    """
    Custom kernel function for use with a support
    vector classifier.

    # Arguments
        X1: first array of sample data for comparison,
            with samples as rows and features as
            columns (size N1 x M)
        X2: second array of sample data for comparison,
            with samples as rows and features as
            columns (size N2 x M; may be the same
            as X1)
        gamma: a scaling hyperparameter for the similarity
            function

    # Returns
        K: the Gram matrix for the kernel, giving the
           kernel space inner products for each pairing of
           a vector in X1 with one in X2 (size N1 x N2)
    """
    assert(X1.shape[1] == X2.shape[1])

    # TODO: implement this
    return None
```

You can define any kernel function you like here, just remember that the resulting Gram matrix must be symmetric and positive semi-definite. Can you come up with a kernel that interacts interestingly with the data model you've implemented in Task 3.1?

(The `gamma` parameter here is passed in from the `gamma` command line option, just as it is for the RBF kernel. You are not required to use it, but the option is there in case you want to experiment with it.)

5 Further exploration

If you have exhausted the previous exercises, or just have a hankering to implement the SVM algorithm rather than simply using it, then try out one or more of the following optional tasks.

These tasks are not included in the `week_3.py` script skeleton. You can add them in, but it will probably be easier to start from scratch in a new script of your own.

5.1 Implement a linear SVM using hinge loss optimisation

As discussed in lecture 4.4, (linear) SVM fitting can be implemented via gradient descent optimisation of a regularised **hinge loss**:

$$\mathbf{w}^\star, b^\star = \underset{\mathbf{w},b}{\operatorname{argmin}} \frac{1}{k}\sum_i^k \max(0, 1 - y_i(\mathbf{x}_i \cdot \mathbf{w} + b)) + \frac{\lambda}{2}\|\mathbf{w}\|^2 \tag{4}$$

where we've chosen to average the hinge loss over the mini-batch size k. The hinge loss is not differentiable, but we can define a subgradient. If L is the hinge loss for an individual sample:

$$L(\mathbf{x}, y) = \max(0, 1 - y(\mathbf{x} \cdot \mathbf{w} + b)) \tag{5}$$

then

$$\nabla_{\mathbf{w}} L = -y\mathbf{x} \cdot \mathbf{1}(y(\mathbf{x} \cdot \mathbf{w} + b) < 1) \tag{6}$$

$$\frac{\partial L}{\partial b} = -y \cdot \mathbf{1}(y(\mathbf{x} \cdot \mathbf{w} + b) < 1) \tag{7}$$

and the mini-batch parameter updates are:

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha\left[\lambda\mathbf{w} - \frac{1}{k}\sum_i y_i\mathbf{x}_i \cdot \mathbf{1}(y(\mathbf{x} \cdot \mathbf{w} + b) < 1)\right] \tag{8}$$

$$b \leftarrow b - \alpha\left[-\frac{1}{k}\sum_i y_i \cdot \mathbf{1}(y_i(\mathbf{x}_i \cdot \mathbf{w} + b) < 1)\right] \tag{9}$$

where α is the learning rate.

Try implementing this and seeing how it behaves. If you consider the stochastic gradient descent case (i.e., $k = 1$), how does this relate to the perceptron algorithm?

5.2 Implement a kernel-capable SVM using a QP solver

The standard NumPy/SciPy stack does not include a quadratic programming solver, so you'll need to install other packages to perform this task. I'd suggest using cvxpy, though you could also try cvxopt.

QP solvers require the problem to posed in very specific forms, which vary according to the implementation. Getting to grips with this can be a little involved, which is one reason why this task is not part of the main lab exercises. But sample code is easy to find—e.g., see this example for cvxpy, which implements a slightly different (L_1 regularised) SVM in QP form.

5.3 Use an SVM to classify handwritten digits

One of the early SVM successes, which helped build the reputation of the method, was in recognition of handwritten digits. Such recognition remains a very common computer vision testbed task.

The standard dataset for digit recognition tasks is MNIST, in which the individual digit samples are 28×28 pixel greyscale images. MNIST data can be downloaded from the OpenML repository using `scikit-learn`'s dataset functions:

```
from sklearn.datasets import fetch_openml
X, y = fetch_openml('mnist_784', version=1, return_X_y=True, as_frame=False)
```

`scikit-learn` also provides a much smaller dataset of 8×8 pixel handwritten digit images which you may prefer to use for more tractable computation:

```
from sklearn.datasets import load_digits
X, y = load_digits(return_X_y=True, as_frame=False)
```

For either dataset, you will need to split the data into disjoint **train, test and validation sets** and determine appropriate hyperparameters to use for training your SVM.

Note also that this is a multiclass problem, while SVMs are inherently binary classifiers, so an appropriate aggregation scheme is needed—however, the `sklearn.svm.SVC` class will take care of this for you, employing a **one-vs-one** decision strategy.