10
Support Vector Machines

In this chapter we focus on *classification*, the problem of using *features* to predict which *class* an observation belongs to. We are particularly interested in distinguishing among two classes, a problem known as binary classification. We will introduce the Support Vector Machine, or SVM, a framework for solving classification problems using optimization and linear algebra.

"Machine" refers to an algorithm. We'll explain what a support vector is later in the chapter.

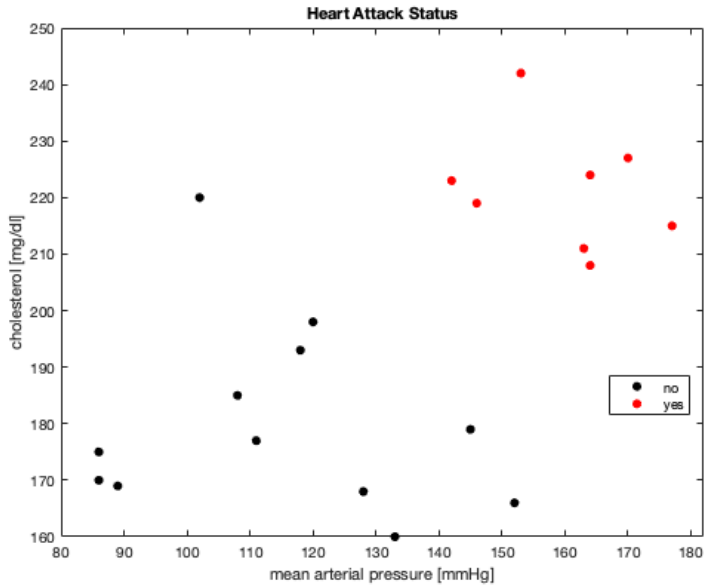
As an example, consider the following dataset that reports the blood pressure and cholesterol levels of 20 patients. Twelve of the patients have not experienced a heart attack, but the remaining eight have. Let's load and plot the data.

```
load HeartAttack.mat
hatk
```

	BloodPressure	Cholesterol	HeartAttack
1	133	160	'no'
2	152	166	'no'
3	128	168	'no'
4	89	169	'no'
5	86	170	'no'
6	86	175	'no'
7	111	177	'no'
8	145	179	'no'
9	108	185	'no'
10	118	193	'no'

```
gscatter(hatk.BloodPressure,hatk.Cholesterol, ...
```

```
        hatk.HeartAttack,'kr');
hold on
xlabel('mean arterial pressure [mmHg]');
ylabel('cholesterol [mg/dl]');
title('Heart Attack Status')
hold off
```



It's clear that we can separate the patients who experienced a heart attack from the ones who did not. However, the separation requires knowledge of both blood pressure and cholesterol levels. There is no cholesterol level alone that separates the two classes, and the same is true for blood pressure. We want to identify a hyperplane that separates the classes so we can predict the heart attack risk for other patients.

For small datasets like this, it is possible to simply draw a line that separates the classes. For problems with only two features, classification is often trivial. However, classifying with thousands of features cannot be done *ad hoc*. Fortunately, everything we will learn in lower dimensions generalizes easily to higher dimensions.

10.1 Separating Hyperplanes

First we *code* the points by setting one group equal to +1 and the other group to −1. For our heart attack data, patients who experienced a heart attack are +1 and the rest are −1.

The term *code* in this context is unrelated to computer programming.

```
hatk.HeartAttack = [-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
                     1  1  1  1  1  1  1  1]'
```

	BloodPressure	Cholesterol	HeartAttack
1	133	160	-1
2	152	166	-1
3	128	168	-1
4	89	169	-1
5	86	170	-1
6	86	175	-1
7	111	177	-1
8	145	179	-1
9	108	185	-1
10	118	193	-1

The +1 and −1 designations are arbitrary — it doesn't matter which group is which. Switching the +1 and −1 codings will give the same classifer. The resulting hyperplane will have the normal vector pointing the opposite way, but this does not affect performance of the classifier.

Our goal is to find a hyperplane that separates the +1 and −1 points. Recall that any hyperplane can be represented in the Hesse form as

$$\mathbf{a} \cdot \mathbf{x} = b$$

This is the Hesse form, not the Hesse normal form since **a** has not been normalized.

where **a** is a vector of coefficients and b is a scalar. Normally our goal is to find values for the vector **x**. For classification problems we know that values of **x** (the features) for each point. Our goal is to find the coefficients **a** and the scalar b that define the separating hyperplane.

We want to choose **a** and b such that all of the +1 points are on one side of the hyperplane and all of the −1 points lie on the other size. By convention, we will put the +1 points above the plane, which we enforce with the constraint

$$\mathbf{a} \cdot \mathbf{x} \geq b$$

for any values **x** that are coded +1. Similarly, we require the −1 points be below the hyperplane with the constraint

$$\mathbf{a} \cdot \mathbf{x} \leq b$$

for any values **x** that are coded −1. Note that there are usually infinitely many hyperplanes that can separate the +1 and −1 points. We

want to find the hyperplane that maximizes the gap, or *margin*, between the +1 and −1 points. The hyperplane that maximizes this gap is called the *maximal margin hyperplane*.

To find the maximal margin hyperplane, we start with two parallel hyperplanes. We require all the +1 points be above the top plane and all −1 points be below the bottom plane. We push the two planes apart until the top plane hits the nearest +1 point and the bottom plane hits the nearest −1 point. When the gap between the two planes is maximized, we know that the maximal margin hyperplane will sit halfway in between the two planes.

Let's formalize this procedure. We define the top plane to be $\mathbf{a} \cdot \mathbf{x} = b + 1$ and the bottom plane to be $\mathbf{a} \cdot \mathbf{x} = b - 1$. Since both planes have the same normal vector $\mathbf{a}$ we know they are parallel. The ±1 terms are arbitrary since we aren't requiring the vector $\mathbf{a}$ to be a unit vector. How far apart are these planes? Let's convert the planes to the Hess normal form. Then the top plane is at a distance of $(b+1)/\|\mathbf{a}\|$ from the origin and the bottom plane is at distance $(b-1)/\|\mathbf{a}\|$. The distance between the planes is therefore

$$\frac{b+1}{\|\mathbf{a}\|} - \frac{b-1}{\|\mathbf{a}\|} = \frac{2}{\|\mathbf{a}\|}$$

The gap between the planes is inversely proportional to the magnitude of $\mathbf{a}$. Maximizing the separation between the planes is equivalent to minimizing the magnitude of $\mathbf{a}$. All together, the maximal margin hyperplane is the solution to the following constrained optimization problem:

$$\begin{aligned} \underset{\mathbf{a},b}{\text{minimize}} \quad & \|\mathbf{a}\| \\ \text{subject to} \quad & \mathbf{a} \cdot \mathbf{x} \geq b + 1 \quad \text{for the } +1 \text{ points} \\ & \mathbf{a} \cdot \mathbf{x} \leq b - 1 \quad \text{for the } -1 \text{ points} \end{aligned}$$

This might seem like a difficult optimization, but there is an important simplification. Remember the definition of the magnitude

$$\|\mathbf{a}\| = \sqrt{a_1^2 + a_2^2 + \cdots + a_n^2}$$

The square root function is *monotonically increasing*, meaning it always increases as its argument increases. Because of monotonicity, minimizing the square root of an input is equivalent to minimizing the input itself. Rather than minimize $\|\mathbf{a}\|$ we can simply minimize the expression $a_1^2 + a_2^2 + \cdots + a_n^2$. The classification problem becomes

Functions like $\cos(x)$ are not monotonic as they both increase and decrease as x increases.

$$\begin{aligned} \underset{\mathbf{a},b}{\text{minimize}} \quad & a_1^2 + a_2^2 + \cdots + a_n^2 \\ \text{subject to} \quad & \mathbf{a} \cdot \mathbf{x} \geq b + 1 \quad \text{for the } +1 \text{ points} \\ & \mathbf{a} \cdot \mathbf{x} \leq b - 1 \quad \text{for the } -1 \text{ points} \end{aligned}$$

Since the objective is purely quadratic with positive coefficients, we know it is convex. We also know that the constraints are linear and therefore also convex. We are minimizing a convex function over a convex set, which is easily solved.

10.2 Setting up the SVM Quadratic Program

The SVM problem outlined above is a *quadratic program (QP)*, a term in optimization that means a problem with a quadratic objective function and a set of linear constraints. Let's set up a QP for a subset of the heart attack data:

BloodPressure	Cholesterol	HeartAttack
133	160	-1
89	169	-1
164	224	+1
153	242	+1

1. Define the dimensions of the problem. We have two predictor variables: blood pressure and cholesterol level. Let's set x_1 = blood pressure and x_2 = cholesterol. We then know that $\mathbf{a}$ has two dimensions (a_1 and a_2).

2. Write out the objective function. The objective is to minimize the magnitude of $\mathbf{a}$, or
$$\underset{a_1,a_2,b}{\text{minimize}}\ a_1^2 + a_2^2$$

3. Write out constraints for each point by substituting values for $\mathbf{x}$. We have four points so we will have four constraints. The constraints for the -1 points are
$$\begin{aligned} 133a_1 + 160a_2 &\leq b - 1 \\ 89a_1 + 169a_2 &\leq b - 1 \end{aligned}$$
For the +1 points we have
$$\begin{aligned} 164a_1 + 224a_2 &\geq b + 1 \\ 153a_1 + 242a_2 &\geq b + 1 \end{aligned}$$

All together, the quadratic program for finding the SVM classifier for these data is
$$\begin{aligned} \underset{a_1,a_2,b}{\text{minimize}} \quad & a_1^2 + a_2^2 \\ \text{subject to} \quad & 133a_1 + 160a_2 \leq b - 1 \\ & 89a_1 + 169a_2 \leq b - 1 \\ & 164a_1 + 224a_2 \geq b + 1 \\ & 153a_1 + 242a_2 \geq b + 1 \end{aligned}$$

10.3 SVM in MATLAB

Setting up an SVM problem by hand is informative but unwieldy for large datasets. There are several software libraries for efficiently formulating and solving SVM problems. We will use the `fitcsvm` function to fit an SVM classifier. The function takes two arguments: a matrix of features and a vector with class codings. Let's begin by putting our two features into a matrix.

The name `fitcsvm` stands for "fit classifier SVM".

```
features = [hatk.BloodPressure hatk.Cholesterol]
```

```
features = 20x2
   133   160
   152   166
   128   168
    89   169
    86   170
    86   175
   111   177
   145   179
   108   185
   118   193
```

Now we call `fitcsvm` and store the output in a variable that we'll call `model`.

```
model = fitcsvm(features, hatk.HeartAttack)
```

```
model =
  ClassificationSVM
             ResponseName: 'Y'
    CategoricalPredictors: []
               ClassNames: [-1 1]
           ScoreTransform: 'none'
          NumObservations: 20
                    Alpha: [3x1 double]
```

```
                    Bias: -16.4864
         KernelParameters: [1x1 struct]
           BoxConstraints: [20x1 double]
          ConvergenceInfo: [1x1 struct]
          IsSupportVector: [20x1 logical]
                   Solver: 'SMO'
```

The model object contains lots of information. Some important pieces are the values for **a** (`model.Beta`) and value of the scalar b (`model.Bias`)

```
model.Beta
```

```
ans = 2x1
    0.0465
    0.0488
```

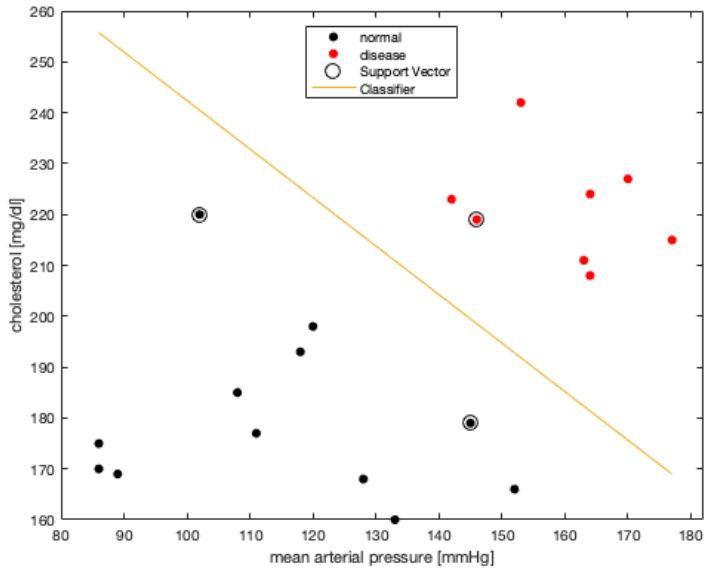
```
model.Bias
```

```
ans = -16.4864
```

We can use these values to plot the maximal margin hyperplane.

```
bp = hatk.BloodPressure; ch = hatk.Cholesterol;
gscatter(bp,ch,hatk.HeartAttack,'kr');
hold on
xlabel('mean arterial pressure [mmHg]');
ylabel('cholesterol [mg/dl]');
plot(bp(model.IsSupportVector), ...
     ch(model.IsSupportVector), ...
     'ko', 'MarkerSize',10);
plot(bp, ...
```

```
    -model.Beta(1)/model.Beta(2)*bp ...
        - (model.Bias)/model.Beta(2))
legend('normal','disease','Support Vector','Classifier');
hold off
```



We've also identified the *support vectors* on the above plot. Remember that to find the maximal margin hyperplane we push two parallel planes outward until they hit the nearest +1 and −1 points. These nearest points are called the support vectors since they "support" the planes. Support vectors are what determine the location of the maximal margin hyperplane; their importance gives rise to the name Support Vector Machine.

So far we've trained an SVM model. We can also make predictions about new patients using the model object and the Matlab function `predict`. The `predict` function accepts a model object and a matrix of features for the unknown observations. It returns predictions (+1 or −1) for each observation. Let's make predictions for two new patients with the following data:

```
BloodPressure   Cholesterol
-------------   -----------
     153            230
      99            132
```

```
predict(model, [153 230; 99 132])
```

```
ans = 2x1
     1
    -1
```

Our model predicts the first patient would have a history of heart attack while the second patient would not.

10.4 Cross Validation

We need a method to assess the accuracy of our classification. There is one primary rule when validating a classifier — **never train and validate a model with the same data**. Models are inherently biased toward the data they were trained on. If the same data are fed back into a model, you can easily overestimate the performance of your classifier.

This leaves us with a problem. Data are expensive and difficult to acquire. Most engineers don't want to set aside some of their hard-earned data just to validate their model. After all, the best model usually results from training with the most data. The solutions is a technique called *cross validation*. The most popular method of cross validation is k-fold cross validation. In k-fold cross validation, the data are split randomly into k equal sized groups, or *folds*. One first fold is set aside for validation and a model is trained on the remaining $k - 1$ folds. The process is repeated by leaving out the second fold, and so on, for a total of k separate validation experiments. The overall accuracy of the model is the average of the k validation experiments.

For example, imagine we had 200 data points and wanted to perform a 10-fold cross validation (i.e. $k = 10$). We first randomly split the data into 10 folds, each fold having 20 data points. We set aside the first fold for validation and train the model with the remaining nine folds (180 data points). Then we test the classifier on each of the 20 points in the first fold. This entire process is repeated for each of the 10 folds, and we report the average accuracy of the 10 validation experiments.

The limiting case for cross validation is when k equals the number of data points, e.g. a 200-fold cross validation in the previous example. Such a procedure is called a *leave-one-out* (L1O) cross validation since each fold contains only a single point.

What is the best value of k for a k-fold cross validation? It depends. There is a tradeoff between the accuracy of the validation and the computation time. Let's return to our example with 200 data points. The best model we can fit (the one we would use for prediction) would train with all 200 data points. Thus a leave-one-out cross validation that trains models with 199 points is closest to the fully trained model with 200 points. Unfortunately, a leave-one-out cross validation requires us to retrain the model 200 times, which can be very computationally demanding for large data sets. Additionally, the accuracy of each of the leave-one-out folds does not differ by much since only a single point was changed in the training dataset. As a result, leave-one-out cross validations are not great at assessing the robustness of the model. They are, however, great for measuring the accuracy of a model trained on the entire dataset.

For k-fold cross validation with smaller values of k (4 – 10 are popular choices), we always underestimate the performance of our model. With 200 data points and $k = 4$, the cross validation measures the performance of models trained with 150 data points; this should be worse than the complete model trained on all 200 data points. On the bright side, cross validation with small values for k are computationally efficient. These experiments also give a hint as to the model's robustness. If the accuracy for each fold varies wildly, we suspect that the performance of the classifier depends strongly on size or particulars of a set of training data.

Finally, remember that data points are assigned to fold randomly, so the results of a k-fold cross validation are stochastic. If the cross validation is repeated and different points are assigned to different folds, the reported accuracy can change. This is not a problem with leave-one-out cross validation because each point is removed individually.

10.5 k-fold Cross Validation in Matlab

Performing a k-fold cross validation requires 1.) randomizing the folds, 2.) retraining the model, and 3.) classifying each fold. Fortunately, there is a Matlab function to perform k-fold cross validation. We can perform a 5-fold cross validation on our heart attack SVM model as follows

```
xval = crossval(model,'Kfold',5)
```

```
xval =
  classreg.learning.partition.ClassificationPartitionedModel
    CrossValidatedModel: 'SVM'
         PredictorNames: {'x1'  'x2'}
           ResponseName: 'Y'
        NumObservations: 20
                  KFold: 5
              Partition: [1x1 cvpartition]
             ClassNames: [-1 1]
         ScoreTransform: 'none'
```

The object returned by `crossval` contains information about how the folds were created and tested. The accuracy of the classifier is measured by the *loss*, with lower loss meaning a better model. We can find the loss by calling the `kfoldLoss` function on our `crossval` return object.

Note the capital "L" in `kfoldLoss`.

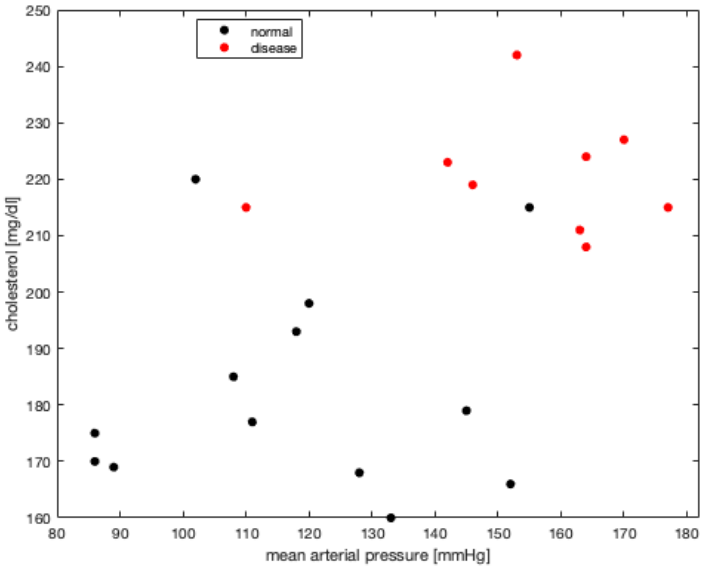
```
kfoldLoss(xval)
```

```
ans = 0
```

10.6 Soft Classifiers

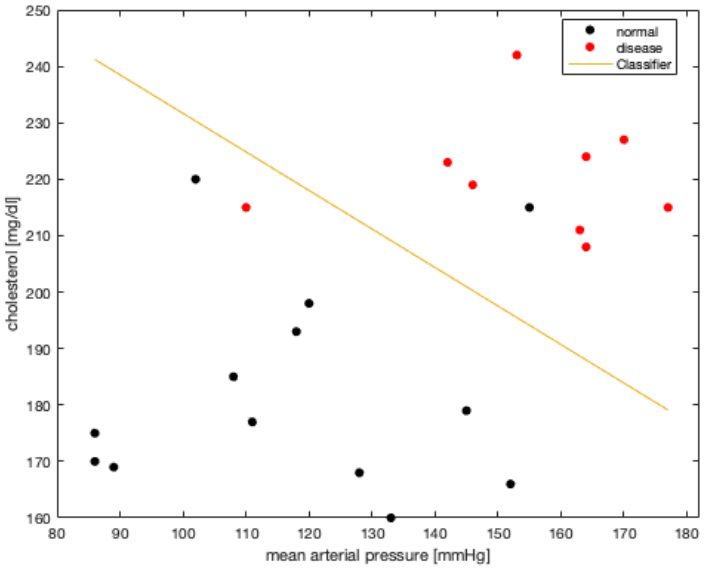
Our heart attack data was perfectly classifiable since we could cleanly separate the +1 and −1 classes. This is not always the case, especially for biological datasets. Let's add two points to our dataset and replot the data.

```
hatk(end+1,:) = {155,215,-1};
hatk(end+1,:) = {110,215, 1};
gscatter(hatk.BloodPressure,hatk.Cholesterol, ...
         hatk.HeartAttack,'kr');
hold on
xlabel('mean arterial pressure [mmHg]');
ylabel('cholesterol [mg/dl]');
legend('normal','disease')
hold off
```



With the new data, it doesn't appear that we can perfectly separate the heart attacks from the rest. Let's try to refit our model.

```
mdl = fitcsvm([hatk.BloodPressure hatk.Cholesterol], ...
              hatk.HeartAttack);
gscatter(hatk.BloodPressure,hatk.Cholesterol, ...
         hatk.HeartAttack,'kr');
hold on
xlabel('mean arterial pressure [mmHg]');
ylabel('cholesterol [mg/dl]');
plot(hatk.BloodPressure, ...
     -mdl.Beta(1)/mdl.Beta(2)*hatk.BloodPressure ...
         - (mdl.Bias)/mdl.Beta(2))
legend('normal','disease','Classifier');
hold off
```



Now we have some points that sit on the wrong side of the classifier. Our accuracy should have decreased (i.e. out loss during cross validation should increase).

```
xval = crossval(mdl,'Kfold',5); kfoldLoss(xval)
```

```
ans = 0.0909
```

The SVM formulation we described above is called a *hard classifier* since it requires that all points be on the correct side of the classifier. In practice, SVM software packages use a *soft classifier* where points can appear on the wrong side. When solving SVM problems with soft classifiers, the goal is to both maximize the separation and minimize the number of incorrectly classified points. We will not discuss the mathematical formulation of soft classifiers in this class.