Understanding Support Vector Machines

For today's class, we're going to play with a very fast and robust solver to the SVM learning problem we've been talking about in class. This will give you some experience with a state of the are SVM implementation (and, in general, classification algorithm) and you're also going to get a bit more comfortable running things from the command-line.

1 SVM^{light}

The SVM solver we're going to play with today is SVM^{light} which is a freely available solver for the SVM learning problem available at:

`http://svmlight.joachims.org/`

It has the standard binary classification version, but it also has many other variants like a multi-class version. SVM^{light} is my personal favorite SVM implementation (and, probably classifier) since it is *extremely* fast and does very well with high dimensional data sets.

2 Lab starter

To get started, ssh into `little.cs.pomona.edu` and then copy over everything in the starter for this lab into your home directory:

`/home/dkauchak/PUBLIC/cs158/svm_lab_starter/`

In there you will find four directories:

- `svmlight:` The basic, binary classification version of SVM^{light}
- `svmlight_multiclass:` The multiclass version of SVM^{light}
- `bin:` Some helper scripts to help us out
- `data:` Versions of our two data sets that have been formatted to be read by SVM^{light}

Throughout the rest of this handout I'll assume you're running all of the commands from the based of this starter directory.

3 Binary classification

To start with, let's see how well the SVM^{light} does on the titanic problem. In the data directory there is a file called `titanic.svm`. Open up this file with your favorite text editor and take a look at the file format. Each line represents an example. The line starts with the label and then the features are written as "<`featureNumber`>`:`<`value`>". Given this, do you think SVM^{light} can handle a sparse feature representation?

Before we can train our classifier, we first need to split our data into training and testing. First, create a working directory to play in:

```
mkdir working
```

Now, let's figure out how man lines/examples are in the titanic file. Type:

```
wc -l data/titanic.svm
```

You should have gotten back that there were 714 lines in the file.

Now, let's create an 80/20 split (i.e. 571/143 examples). To do this, we'll just take the first examples as training and the last examples as testing. To do this, we're going to use the `head` and `tail` commands.

```
head -n 571 data/titanic.svm > working/titanic.train.svm
tail -n 143 data/titanic.svm > working/titanic.test.svm
```

(If you're not familiar with it, the `>` send the output from a program to a particular file.)

Now, run the SVM^{light} classifier without any arguments to see what parameters it takes:

```
svmlight/svm_learn
```

At the top you should see the usage which says something like:

```
usage: svm_learn [options] example_file model_file
```

The `example_file` is the input file (i.e. training file) and the `model_file` is the output file that contains the model. Notice also there are a whole bunch of other parameters to play with! For now, though, just run it with the default parameters:

```
svmlight/svm_learn working/titanic.train.svm working/titanic.model
```

As I mentioned before, SVM^{light} uses a very efficient optimization algorithm and it should finish almost immediately. Take a look at the output that the SVM training gives you. See if you can interpret the output (you should be able to understand at least half of it).

Before we try and classify the data, let's take a peak at the model file to see what SVM^{light} actually learned. Using a text editor, open the file `working/titanic.model`. The first few lines display the options that were used to train the model and some other meta-data about training. After that, what is displayed are all of the support vectors! Because SVM^{light} allows for other types of models besides just linear, this turns out to be a better representation.

However, one of the nice things about a linear model is that they're easy to understand. For example, we can look at which weights have the largest magnitude and that should be indicative of the more important features (why?). I've provided a script in the `bin` directory that goes from the support vector representation and reproduces the actual weights:

```
python bin/svm2weight.py working/titanic.model
```

Do the weights make sense? Recall that the SVM learning problem can be thought of as minimizing hinge loss with L2 regularization. Does the results reflect this regularization (*Hint: the results are one reason why people prefer L1 regularization over L2*).

Ok, now let's classify our test data and see how we did. Run the classify method without any parameters to see what parameters it takes:

```
svmlight/svm_classify
```

The first parameter is the test example, the second the learned model and the third it the output file where the predictions will go. So, to classify run:

```
svmlight/svm_classify working/titanic.test.svm working/titanic.model
  working/titanic.output
```

(all one line)

Besides classifying all the examples, the classify program also outputs some evaluation statistics for us. How did we do? Remember this is just one split of the data, so we shouldn't make any major claims.

4 Binary wine prediction

To play with a more interesting binary classification problem (well, at least more features) I picked two of the wine categories and made a binary classification problem in the file `data/wine.binary.svm` which is a file that has Cabernet-Sauvignon as the positive class and Pinot-Noir as the negative class.

1. split the data into an 80/20 train/test split using the same technique as above
2. train the svm classifier
3. test the svm classifier

How do we do?

To try and understand why we do so well, let's take a look at the feature weights again:

```
python bin/svm2weight.py working/wine.model
```

This time, we see way, way too many to really make any sense of the model. What we're really interested in are the features that have the largest magnitude. So, let's sort the file based on that:

```
python bin/svm2weight.py working/wine.model | sort -n -k 3 >
   working/wine.weights
```

(all one line)

This sends the weights to the built in sorting method (sorting them by the third column and treating them as numbers) and then sends the output to `wine.weights`.

If you open up this file at the beginning will be the most negative words/features and at the end the most positive words/features (minus the fact that sorting doesn't handle scientific notation correctly).

Right now, these features are just numbers. To understand what features they represent, open up the file `data/wine.features` which has the corresponding words for each feature. Look at the top few most negative and positive features. Do they make sense?

5 Multiclass classification

SVM^{light} also has a multiclass variant that has a similar interface to the binary classifier. The one difference is that it does NOT automatically set C for you. Recall that for the SVM, this trades off the margin and the slack penalties with larger C biasing towards weighting the slack penalties more. If we interpret this as a gradient descent method, though, the C constant trades off the loss versus the regularization with smaller C biasing towards more regularization.

1. split the wine data `wine.svm` into an 80/20 train/test split using the same technique as above
2. train the svm classifier

   You'll need to set the C parameter, to do this, use the `-c` flag:

   ```
   svmlight_multiclass/svm_multiclass_learn -c <some_number>
      working/wine.train.svm working/wine.model
   ```

   (all one line)
3. test the svm classifier

   ```
   svmlight_multiclass/svm_multiclass_classify working/wine.test.svm
      working/wine.model working/wine.out
   ```

   (all one line)

How did you do? Most likely, not that well :(The choice of C is critical to the performance of the SVM classifier. For the binary classifier, there are reasonable heuristics for setting it. For the multiclass classifier, we're going to have to do it the old-fashioned way, brute force search.

The following commands will check all the C values between 1 and 10 with increments of 1:

```
for i in `seq 1 1 10`;
do
svmlight_multiclass/svm_multiclass_learn -c $i wine.train.svm wine.model;
svmlight_multiclass/svm_multiclass_classify wine.test.svm wine.model wine.out;
done;
```

(Either multiple lines or on a single line)

This will print out all of the testing results for the 10 experiments. If you just want to see the Zero/one loss (i.e. the error) you can add a `grep` statement at the end:

```
for i in `seq 1 1 10`;
do
svmlight_multiclass/svm_multiclass_learn -c $i wine.train.svm wine.model;
svmlight_multiclass/svm_multiclass_classify wine.test.svm wine.model wine.out;
done | grep "Zero/one-error on test set"
```

(Either multiple lines or on a single line)

6 Competition

Time permitting, let's see who can develop the best SVM classifier for our wine task. I will put together a random train/test split of our data. When you're ready, you can see how well you do on the data.

Here are a few tips for experimenting:

- Look at the documentation for the train method (i.e. run it without any parameters) and see what other options are supported. Try some of these out.
- So far, we've only done one train/test split. If you want, you can write a short script (python, perl?) to generate more random splits. Another way is to randomly shuffle the data and then use our same `head`/`tail` approach. To create a random shuffle of the data we can use the `sort` method:

  ```
  sort -R data/wine.svm > working/wine.random.svm
  ```

  After doing this, `wine.random.svm` will have a random permutation of the lines in `wine.svm`

Have fun!