Kobee Raveendran
CAP 5610
Fall 2019

Assignment 2: Support Vector Machines (SVMs)

1 Kernel Selection and Hyperparameter Tuning

1.1 Kernel selection

For the first step of the assignment, we were tasked with testing four kernels for decision boundary generation: the radial basis function (RBF), linear, polynomial, and sigmoid kernels. These are explored below, where x and x' are feature vectors in an input space.

- RBF: $K(x, x') = \exp(\|x - x'\|/(2\sigma^2))$
- Linear: $K(x, x') = x^T x'$
- Polynomial (of power p): $K(x, x') = (1 + x^T x')^p$
- Sigmoid: $K(x, x') = tanh(\beta_0 x^T x' + \beta_1)$

The RBF kernel tends to do well in higher-dimensional data since it creates nonlinear combinations of features in such a way that they can then be separated linearly. For example, in the commonly cited toy dataset consisting of classes in rings around each other, "pushing" each class's feature vectors upward from the middle allows us to then separate them using a hyperplane.

On the contrary, the linear kernel does well if the data is already linearly separable, and typically poorly otherwise, since it becomes difficult, if not impossible, to separate the data using a single line as the dimensionality of the dataset increases.

The polynomial kernel serves as a middle ground, in that, based on the degree of the function that is used to derive the polynomial function, it can draw better decision boundaries for high dimensional data than a linear kernel could. The higher the degree, the easier it will be for the polynomial function to represent high dimensional data. Of course, this also means that using high degree polynomial functions on a low dimensional dataset (such as one that is linearly separable) is not only unnecessary but can also hurt test accuracy due to overfitting (fitting to the training data too closely, preventing it from being able to generalize properly to unseen data in the test set).

The sigmoid kernel comes from its use in logistic regression and neural network models, where it is used as an activation function. In some cases, such as in the concentric circles

dataset, sigmoid can perform well enough when drawing its decision boundary. However, from experience with a few other datasets (as well as the one in this assignment), sigmoid performs poorly and RBF seems to be a safer choice, possibly due to the hyperbolic tangent function's tendency to pull decision boundaries to corners (which is noticeable in plots of decision boundaries where sigmoid performs poorly; some decision boundaries appear parabolic in shape, leading it to poorly fit the data).

1.2 Hyperparameter tuning

My selection of optimal hyperparameters relies on the use of grid search, which is implemented within `scikit-learn`. Additionally, prior to applying grid search, I applied standard K-fold cross validation with `k = 5`. Since I originally included code that essentially mimicked grid search at first (though later swapped out for `sklearn`'s implementation), I will include an explanation of it below with the rest of my process for completing step 1 (from the beginning) and choosing the optimal hyperparameters.

1. Read in the dataset line by line, removing the first index (column) since it is simply the index in the dataset, and is thus useless for training and predicting as it is not actually part of the input feature vector.

2. Shuffle the dataset to prevent entire folds from being monopolized by a single class and raise the chances of having all classes present in each fold. Occasionally, due purely to luck, some class may be extremely underrepresented, and running the code again solves this problem. Once shuffled, I partition the dataset into k-relatively-equally-sized portions.

3. Once I have the 5 folds generated, I begin looping through the 5 folds.

    a. In each iteration, the current fold is kept as the temporary test set and the other $k-1$ folds serve as the temporary training set.

    b. From here, I additionally separate the training and test sets into their feature vector and label parts, respectively labeled as `X_train`, `Y_train` and `X_test`, `Y_test`.

    c. I then prepare a parameter dictionary to supply to grid search. The parameters I use are the four kernels (discussed above), a range of gamma values (mostly negative and positive powers of 10), a range of C values (also negative and positive powers of 10, though one order of magnitude higher),

and a range of degrees for the polynomial function (only the integers between 2 and 4, inclusive).

d. Once I supply an SVM, the parameter dictionary, and number of CV folds to grid search, it does the following (mirroring my originally manual implementation):

    i. Of the elements of the training set, perform k-fold cross validation with `k = 5`. This should create a new training subset of the larger training set for the grid search to use (4/5 the original training set), and a test set (1/5 the original training set) for evaluating the different parameter combinations on.

    ii. Loop through each parameter in a nested fashion, training (fitting) an SVM for each combination of parameters, and evaluating each based on accuracy performance on the test set.

        1. For example, for each kernel, for each value of gamma, for each value of C, etc. train an SVM with those parameters set. This means that N_{gamma} * N_C * $N_{kernels}$ * $N_{degrees}$ SVMs are created, each trained using a specific parameter combination.

        2. Each parameter combination's SVM is evaluated on the test set, and the SVMs that performed best have their parameters extracted and stored for the current fold.

    iii. Finally, I plot the results of grid search, using a combination of the *gamma* and *C* parameters as the dependent variables, with the `mean_test_score` (the average grid search test set accuracy) as the dependent variable. I've included versions using either *gamma* or *C* as the x-axis. These plots can be found in the *figures* folder (generated from my own runs) and can be recreated by running `svm.py` again.

In grid search, I had to supply a dictionary of parameters. I chose those parameters based on some trial runs and intuition from other machine learning hyperparameters such as learning rate in neural networks. Gamma is a parameter used for the RBF, sigmoid, and polynomial kernels, and it affects the ease of placing a hyperplane once the data is raised into a higher dimensional space. Looking at the equation, gamma seems to represent inverse variance, so it is fair to say that a small value of gamma yields high variance (and

low bias), while large values of gamma give low variance and high bias. The other parameter, C, controls the error penalty for misclassified examples, and ties in to the slack variables mentioned in lecture. The use of slack variables meant that the model did not have to correctly classify every single example and instead 'correct' the classification by moving that example by some degree determined by the slack variable for a correct classification. The parameter C prevents us from relying too heavily on the slack variables in order to still get a model that performs well while avoiding using very high values of the slack variable to 'cheat' our way into correct classifications all the time. Low values of C do not punish the use of slack variables, while higher values of C penalize it increasingly.

Specifically, for *gamma* and *c*, I chose powers of 10 and close deviations from them, as these are often also common learning rates and other hyperparameters. More specifically, for *gamma*, I chose 0.0001, 0.0003, 0.001, 0.003, 0.01, 0.03, and 0.1. I initially had smaller values (some orders of magnitude smaller) but through many runs these values were rarely or never picked as optimal values, so I removed them and shifted up one order of magnitude. Also, for values of *gamma* larger than this, grid search times increased noticeably. I had a similar rationale for *C*, though I saw through experimentation that *C* values as small as the lower end of the *gamma* range performed poorly and also weren't chosen, so I moved up several orders of magnitude to get the following range: 0.01, 0.1, 1, 10, and 100.

I also had to supply values for *degree*, though this was more straightforward and easier to do. The *degree* parameter influences the degree (or maximum exponent) of the polynomial function. High degrees mean potentially better ease of fitting to high dimensional data, but having a degree that is too high can lead to overfitting (i.e. polynomial of degree 10 fitting to a linearly-separable dataset will try to tightly match the training set instances). My choices for degrees were the integers between 2 and 4, inclusive. I excluded a degree of 1 because the polynomial kernel of degree 1 is essentially devolves into the linear kernel, which I'm already testing. I also excluded degrees above 4 simply because they took too long to evaluate with grid search, since training time for polynomial SVMs rises with their degree. For convenience and quicker testing, I limited my degree to 4 (though 5 could eventually finish, it still took many minutes, and above 5, grid search times exceeded an hour).

Note: as mentioned above, plots of the results of grid search are included in the `figures` folder, and can later be generated anew by simply running `svm.py`.

1.3 Results

Below are my results of the best parameters and their accuracies on each fold:

Fold	Optimal Parameters	Validation Accuracy (%)
1	Kernel: RBF gamma: 0.03 C: 100 degree: 2	67.84
2	Kernel: RBF gamma: 0.03 C: 100 degree: 2	71.93
3	Kernel: RBF gamma: 0.03 C: 100 degree: 2	71.93
4	Kernel: RBF gamma: 0.1 C: 10 degree: 2	71.93
5	Kernel: RBF gamma: 0.03 C: 100 degree: 2	71.51

2 One-vs.-One (binary) and One-vs.-Rest (multiclass) classifiers

For SVMs, `scikit-learn` implicitly uses a one-vs.-one (OvO) classifier, even for multiclass problems, since SVMs were initially designed for binary classification. In OvO classification, a total of $K\ (K-1)\ /\ 2$ binary classifiers are trained, where K is the number of classes. Each of these binary classifiers are given only subsets of the training set that contain elements belonging to the two classes they are supposed to classify. For example, one such classifier, say classifier AB, will be given only data belonging to classes A and B. All of these classifiers are trained on different datasets comprised of these pairs of classes. At prediction time, an unknown sample is presented to each classifier, and each classifier will classify it into one of their two classes, even if its true class does not belong to that specific classifier's class (for example, classifier AB could receive a test example belonging to class C). Once all classifiers have made predictions, a single class is chosen by majority vote among all classifiers. This class becomes the final prediction of the SVM.

In contrast, in a One-vs.-Rest (OvR) classifier, only K classifiers are trained. In OvR classification for SVMs, each of the K classifiers is still a binary classifier, but the class distribution of the data it is given is different. Rather than belonging to two of the K classes, the ith classifier is given data belonging to the ith class, and all of the rest of the data in the dataset. The data for this ith classifier are then classified as either 1) belonging to the ith class, or 2) not belonging to the ith class. At prediction time, each of the K classifiers are given the unknown test example, and classify it as belonging to their respective class or not by assigning it a probability. At the end, the classifier that gave the example the highest probability 'wins', and the class it represents becomes the SVM's final prediction.

Below is the general procedure I followed for this part, using the built-in `sklearn` functions `OneVsOneClassifier` and `OneVsRestClassifier`.

1. Recreate an SVM using the optimal parameters found in part 1 (above), and enforcing OvO by wrapping it in the `OneVsOneClassifier.`

2. Fit this OvO classifier on the training data (from the remaining $k-1$ folds), and make predictions on the test set (current fold). Record prediction accuracy by summing the number of test elements predicted correctly and dividing by the total number of test elements. Record execution time (in seconds) for *only* the fitting process.

3. Repeat step 1 on a new SVM, this time wrapping it in the `OneVsRestClassifier`.

4. Fit the OvR classifier on the same training set, and make predictions on the same test set. Also record prediction accuracy and execution time in a similar manner to step 2.

3 Accuracy and training time comparisons between OvO and OvR Classifiers

In practice, after running both versions of the SVMs, I noticed several characteristics of each. Regarding the OvO classifiers, they were (in general) slower to train and fit to the data by an entire order of magnitude or more on some folds. This is to be expected, however. Revisiting the description of OvO classifiers, they train a significantly larger number of individual binary classifiers: $K\,(K-1)\,/\,2$ as opposed to OvR's K. Thus, it makes sense that training OvO classifiers should take longer on average.

A characteristic of the OvR classifiers was that they sometimes traded training time for classification performance (with respect to accuracy), though they were still somewhat keeping up with the OvO classifiers. I believe they may sometimes perform worse due to the problems each of the K sub-SVMs has with their datasets: each dataset is hugely imbalanced, and this is problem is exacerbated when a dataset has a high number of classes.

3.1 Results

	Average Accuracy (%)	Training time (s)
One-vs-One	72.41	0.012
One-vs-Rest	70.99	0.0098

4 Weighted Classes

As specified in the writeup, the frequencies of each class appearing in the Glass dataset is not uniform across all classes. It is a less severe version of the dataset imbalance I discussed in part 3, since there are more than just 2 classes, but there is an imbalance nonetheless. We circumvent this by giving each class's set of training examples a weight based on its frequency in the dataset. I used the `sklearn` SVM's `class_weight` parameter to accomplish this. Furthermore, instead of manually calculating weights, simply supplying `'balanced'` lets `sklearn` automatically calculate the weights for each class using the list of class labels (`Y_train`). For each weight, it calculates a simple inversely-proportional value to assign as its weight:

```
class_weight = n_samples / (n_classes * np.bincount(y))
```

After calculating class weights, we can train using the weights to give each class a more 'fair' representation in the data, thus improving our classification accuracy (typically).

Below are my results for weighted classes on both the OvO SVM and OvR SVM. I tried testing it on OvR out of curiosity, because I believed that, if the classes were balanced by weights, maybe OvR could more frequently match or exceed the performance of OvO.

	Average Accuracy (%)	Average Training Time (s)
Balanced OvO	64.90	0.0022
Balanced OvR	67.29	0.0102

As can be seen, the balanced OvR generally outperformed the OvO in terms of accuracy, though it suffered in training time surprisingly. I believe this time penalty can be attributed to any overhead caused by balancing the weights; for OvO, there will be much less weight balancing since the classes of each SVM are much closer in frequency throughout their pairs of datasets. However, for OvR, much more balancing must be done since each classifier gets only one class and all of the other classes combined.