Deep Learning - Neural Networks

Aristotle Universtity Thessaloniki - School of Informatics

Assignment 2: Support Vector Machines

Antoniou, Antonios - 9482

aantonii@ece.auth.gr

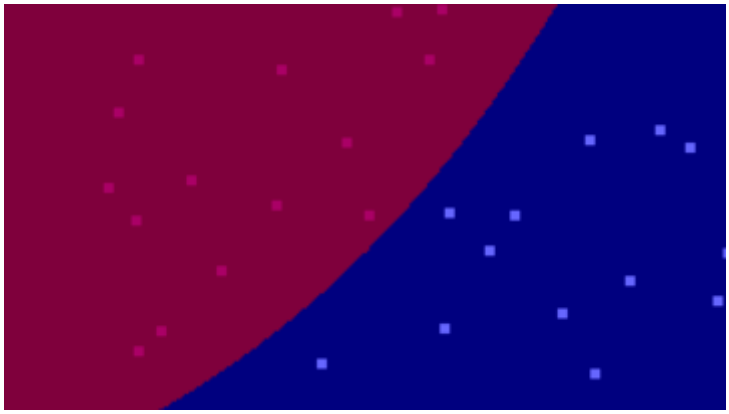
GitHub repository can be found here

Introduction

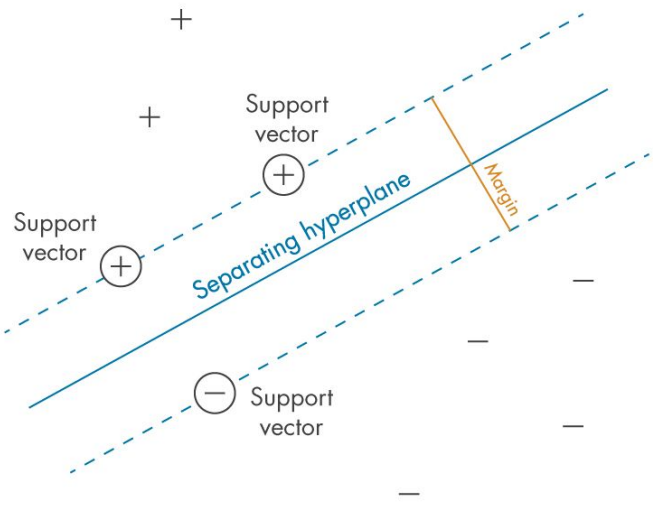
The purpose of the second assignment of the Deep Learning course is to build a Support Vector Machine **(SVM)**, and experiment with the different types of hyperparamters and pre-processing of the dataset the model will be trained upon. Due to the diversity of hyperparameters and how they impact the result of a predictor's training, accuracy and ability to classify a newly introduced data point, we first need to clear up the fundamentals of SVM's.

How does an SVM work?

An SVM's function is based on the training datapoints that reside on the **separating hyperplane** between two classes, called the ***support vectors***. For example, if we use the libsvm official site to build a basic two-class dataset, we can see how that notion affects the forming of the hyperplane that's used to differentiate between them.



The support vectors here are the datapoints lying closest to the hyperplane, and are the only points actively participating in the training procedure and refining the classifier. To make that statement clearer:



Credit: Matlab. https://www.mathworks.com/discovery/support-vector-machine.html

Despite the multitude of both + and - instances, the only ones that matter to the SVM are the highlighted ones, that lie on either side of the separating area.

What is the role of support vectors?

During both training and predicting, a datapoint (or *vector*, as has already been addressed as) is essentially compared to every support vector. That comparison, is done, not in the **dimension** (and space in general) that is "native" to the data. SVM's apply **transformations** to the data, in an attempt to augment the probability of the dataset being **linearly discriminant**. This is done through a ***Kernel function***, and this is where the variety of options in order to solve the same problem starts to set in. A few examples, for two vectors $\boldsymbol{x_0}$ and $\boldsymbol{x_1}$:

1. Linear: $\boldsymbol{K(x_0, x_1) = x_0^T \cdot x_1}$
2. Polynomial: $\boldsymbol{K(x_0, x_1) = (\alpha x_0^T \cdot x_1 + r)^d}$
3. Gaussian:
    - a. $\boldsymbol{K(x_0, x_1) = exp(-\frac{||x_0 - x_1||^2}{2\sigma^2}), or}$
    - b. $\boldsymbol{K(x_0, x_1) = exp(-\gamma||x_0 - x_1||^2)}$

Needless to say, some libraries make a few more assumptions about the values of some coefficients. For example, *libsvm* supposes that $\boldsymbol{2\sigma^2 = \#features}$, $\gamma = \frac{1}{\#features}$. So, depending on the framework one uses to train a model, there is more or less flexibility depending on the type of SVM and kernel they want to use.

Outline of other hyperparameters

The training of an SVM is translated into the solution of a **quadratic programming problem**:

4. $Q(a) = \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j d_i d_j x_i^T x_j$

The problem above is derived, or is the **dual problem**, of a **primal** one. Briefly explained, the dual problem is a transformation of an optimization problem, that yields the same results, by optimizing a different set of variables:

5. $J(\underline{w}, b, \alpha) = \frac{1}{2} \underline{w}^T \underline{w} - \sum_{i=1}^{N} \alpha_i [d_i (\underline{w}^T x_i + b) - 1]$

The α coefficients in (4) are non-zero for the indices that correspond to the support vectors of either class.
However, the formula above makes the assumption that, even on a higher dimension, all vectors are **distinguishable**. So, we introduce the C quantity, and its w_j counterparts. These parameters are a way to tell the SVM **how much we care about it making a couple of errors** (the falsely classified elements could be noisy after all, this is not the model's fault, but a pre-processing matter). So, they are used to ensure that the margin of the resulting hyperplane (see image above), is still optimally large, even though the two classes aren't perfectly divided.
So the primal problem in (5) will be transformed into finding the minimum of:

6. $\Phi(\underline{w}, \xi) = \frac{1}{2}\underline{w}^T\underline{w} + C\sum_{i=1}^{N}\xi_i$

This is where C comes into play. Regarding the w_j parameters, they are weights that are added to the ξ_i ones, for class j.
After that overview, we now have an idea of all the variation we can provide to an SVM, in order to help the model reach a solution we deem trustworthy.

Building an SVM with libsvm

The solution to the optimization problem that we researched above means **massive computation and memory workload**. Many methods have been suggested, implemented and used, such as **decomposition** of the solution matrix. Consequently, we will use **libsvm** for creating a problem to be optimized and training the respective SVM.

Suppose n is the total number of datapoints in the training set, and f is the number of useful features we will use after pre-processing. The library expects an array X of size (n, f) and Y of size $(n, 1)$. With these two arrays we form an `svm_problem`. The second component of training with libsvm is the `svm_parameter` string, that dictates the values of the parameters in the libsvm manual.

Regarding those parameters, an argument parser was built in such a way that both the parameter string would be conveniently given to the library and a selected set of values would be logged after every run, depending on the type of kernel utilized. This means that a call to the script will look like this:

```
.\src\svm.py -s 0 -t 1 -d 3 -r 0 -c 5000 -w0 1 -w1 2
```

And a logging entry of the run above would be:

```
s,t,d,g,r,c,h,time,acc
0,1,3,0.04,0.0,5000,1,236.507,69.240
```

At this point, it's useful to point out that the parser (based on the argparse python library) is set up with the same default values libsvm is, except for the `-h` shrinking parameter, that was set to 0. **The reason for that was that, in the particular examples that the predictor was tested on, it resulted in both smaller training times and the exact same rates in accuracy.**

Note: You can see the results of experimenting with shrinking in this log of results:

s	t	g	r	c	h	time	acc
0	1	0.04	0.1	10	0	161.168	69.428
0	1	0.04	0.1	10	1	186.201	69.432
0	1	0.04	0.1	100	0	153.604	71.113
0	1	0.04	0.1	100	1	176.692	71.109

Training tests conducted

The larger part of the tests was done on the "Body Signal of Smoking" dataset. It was selected because of its natively binary nature (are there signs of smoking or not?), large number of samples (55692) and adequate number of features (26). The number of features was deliberately kept somewhat low, in order to keep the training time *relatively* low.

Preprocessing

Upon examining the dataset, we can observe three preprocessing tasks that need to take place:

- Some columns of data are in string form, and they need to be encoded. Those are `gender` and `tartar`,
- Some columns contain the same value on both classes. That happens to be `oral`.
- The features that remained now need to have their values normalized, to make sure we don't artificially make one feature more important than the others.

To serve that purpose, we introduce the **`split_features_and_classes`** function:

```
def split_features_and_classes(
    df,
    class_col: str,
    encode: List[str] = None,
    drop: List[str] = None
) -> tuple[pandas.DataFrame, numpy.ndarray, numpy.ndarray]
```

It takes the column that corresponds to the y targets, the list of columns that need to be encoded into numeric values (with `sklearn.preprocessing.LabelEncoder`), and the values that will be dropped from the DataFrame. Here, apart from `oral`, we also drop `ID`.

Last, we need a function to get the whole X and Y arrays, and divide them into the training and testing set. Because we're dealing with support vectors, whose placements in the dataset is unknown, we only keep a little segment of the original data (more often 10%), so as not to disrupt class balance and minimize the probability of depriving the dataset of its support vectors. It is of great significance to note that there is a reason behind keeping some datapoints -even a few- for testing, with the risk of imbalance. We need to make sure that, after training, the model **wasn't**

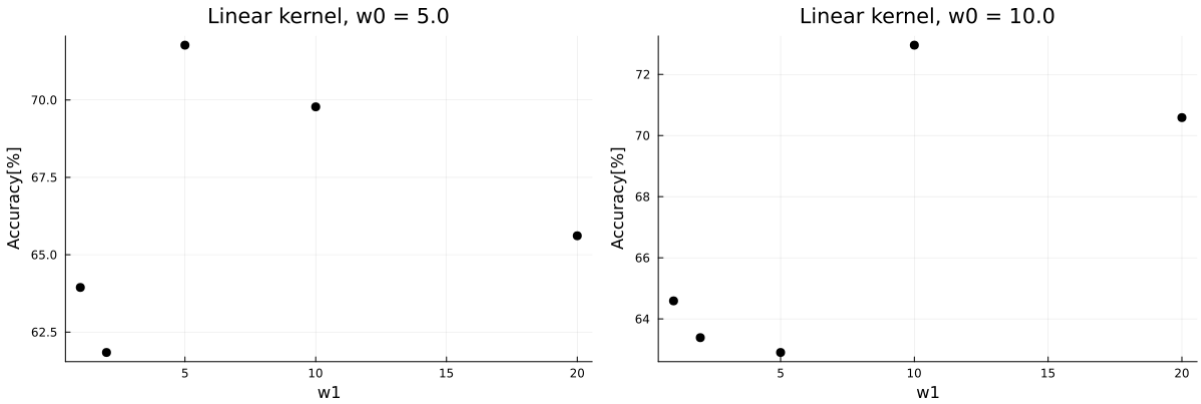
overfitted. The safest way to do that is by using vectors that had never been implanted into the dataset, up until the moment of testing. Think of the procedure as a very loosely put-together cross-validation.

Now that the dataset is normalized, sanitized, and split into training and testing samples, we can move on to the experimenting phase. We will see the results of using different kernels, and different hyperparameters given to them, on a **C-SVC**. For the mathematical notation of each of the kernels that will be examined, you can take another look at (1), (2) and (3).

Below are only *some* of the results visualized, enough to back the statements to be made regarding training the dataset with the selected kernels. All the graphs can be found in this folder of the repository.

Linear kernel

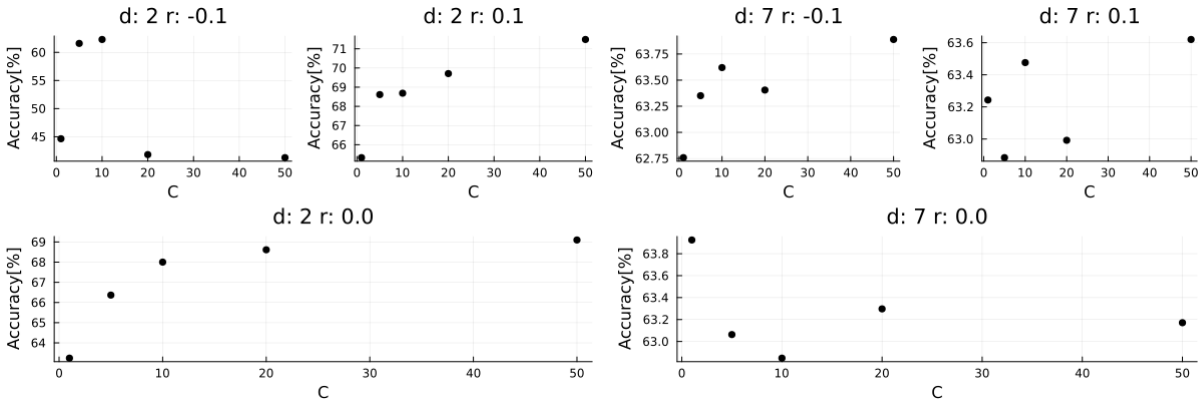
Here, we have room to experiment with the error weight parameters, C, w_0 and w_1. The majority of the samples correspond to non-smokers, so an initial thought would suggest that different w_0 and w_1 values could affect the resulting accuracy for the better.



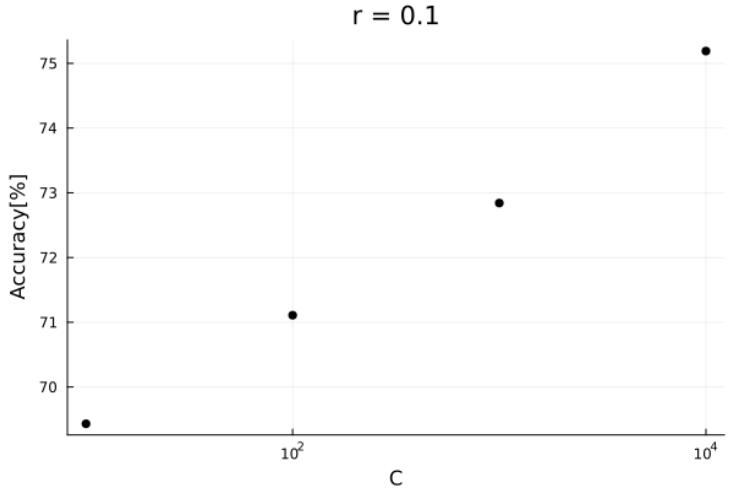
It's pretty clear to see that the accuracy of the model is about 10% greater for the same parameters but $w_1 \geq w_0$. This means that we penalize errors on class-0 (non-smokers) less in relation to the significance of making an error on class-1, whose population is half as dense.

Polynomial kernel

For the polynomial tests, we test different offsets $r = \{-0.1, 0.0, 0.1\}$, for degrees $d = [2, 7]$. The tests show that increasing the degree -for the specific dataset- only worsens accuracy:



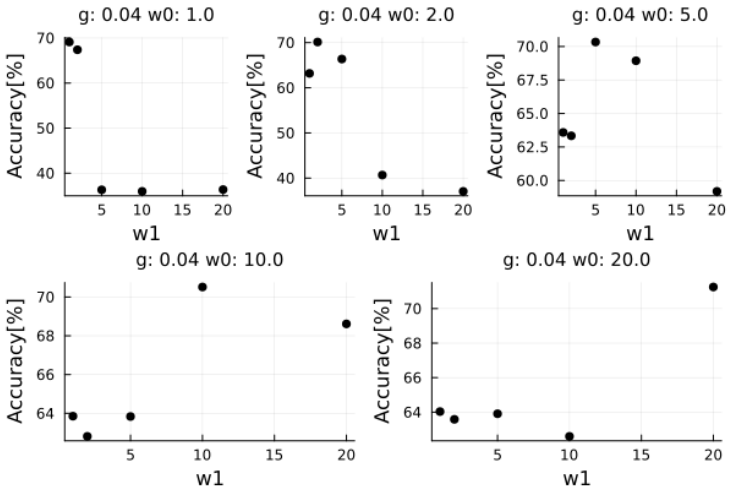
Since for $d = 2$ and $r = 0.1$ we can observe an upwards tendency in accuracy, we test for greater values of C:



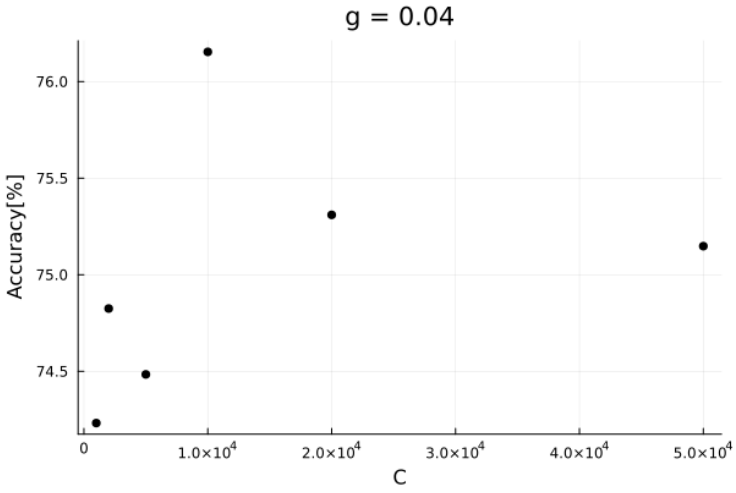
Now, we are happy with these results, but take them with a pinch of salt. Values of C, that are as great as the ones above conceal a really high risk of overfitting, by forming a tight space that classifies one of the classes, and essentially considers anythin outside that tight space to be the other class. However, in any way, augmenting C stops having such an impact. It looks like the plateau is 75% accuracy for a polynomial kernel of degree $d = 2$

RBF kernel

Values of γ in $[0.01, 0.05]$ with a step of 0.01 were used, together with pairs of w_0, w_1 in a grid of values $w = \{1.0, 2.0, 5.0, 10.0, 20.0\}$. All values of γ yielded the same results. Let's take $\gamma = 0.04$ for example, since it is the default value for the dataset:



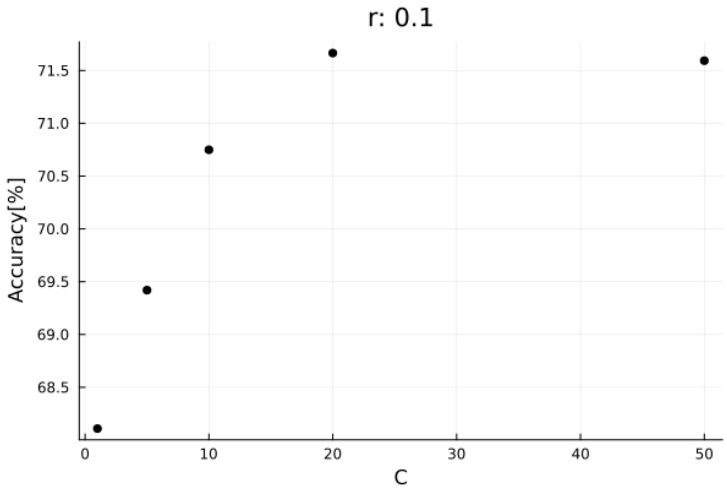
This time, the relation between the weight parameters is different. Accuracy takes its maximum value, when $w_0 = w1$. With this information, we conduct tests for $\gamma = 0.04$, default values $w_0 = w_1 = 1.0$, and large values of C:



Now, $C = 10000$ gives an accuracy of 76.154%, which is the best we have managed to achieve.

Sigmoid kernel

For experiments on the sigmoid kernel, we use the same w_0, w_1 pairs as the ones for the RBF kernel. We test them for $r = [-0.2, 0.2]$, with step equal to 0.1. Once again, the results are comparable, if not identical. For example:



Nearest Centroid and KNN Classifiers

The code for the classifiers can be found in the clustering directory of the source code.

Using the same pre-processing techniques, and the same fraction of data for validating the results of the training (10%), we train 3 classifiers:

Classifier	Training time[s.]	Accuracy[%]
NearestCentroid	0.015	65.505
1-NearestNeighbor	0.01	77.375
3-NearestNeighbrors	0.0049	69.977

The KNN classifier takes a minuscule fraction of the time that we need for an SVM to train on the dataset, with the resulting accuracy being comparable, or even better than all tests above.