**Support Vector Machines**

SVMs revolve around the notion of a “margin”—either side of a hyperplane that separates two data classes. Maximizing the margin and thereby creating the largest possible distance between the separating hyperplane and the instances on either side of it has been proven to reduce an upper bound on the expected generalization error.

In the case of linearly separable data, once the optimum separating hyperplane is found, data points that lie on its margin are known as support vector points and the solution is represented as a linear combination of only these points. Other data points are ignored. Therefore, the model complexity of an SVM is unaffected by the number of features encountered in the training data. The number of support vectors selected by the SVM learning algorithm is usually small. For this reason, SVMs are well suited to deal with learning tasks where the number of features is large with respect to the number of training instances.

Even though the maximum margin allows the SVM to select among multiple candidate hyperplanes, for many datasets, the SVM may not be able to find any separating hyperplane at all because the data contains misclassified instances. The problem can be addressed by using a *soft margin* that accepts some misclassifications of the training instances.

Nevertheless, most real-world problems involve non-separable data for which no hyperplane exists that successfully separates the positive from negative instances in the training set. One solution to the inseparability problem is to map the data onto a higher-dimensional space and define a separating hyperplane there. This higher-dimensional space is called the *feature space*, as opposed to the *input space* occupied by the training instances.

With an appropriately chosen feature space of sufficient dimensionality, any consistent training set can be made separable. A linear separation in feature space corresponds to a non-linear separation in the original input space. Mapping the data to some other (possibly infinite dimensional) Hilbert space H as : *Rd* → *H*. Then the training algorithm would only depend on the data through dot products in H, i.e., on functions of the form (*xi* )· (*x j* ). If there were a “kernel function” *K* such that *K*(*xi*,*xj*) = (*xi*) · (*xj*), we would only need to use *K* in the training algorithm, and would never need to explicitly determine . Thus, kernels are a special class of function that allow inner products to be calculated directly in feature space, without performing the mapping described above. Once a hyperplane has been created, the kernel function is used to map new points into the feature space for classification.

The selection of an appropriate kernel function is important, since the kernel function defines the feature space in which the training set instances will be classified. It is common practice to estimate a range of potential settings and use cross-validation over the training set to find the best one. For this reason, a limitation of SVMs is the low speed of the training. Selecting kernel settings can be regarded in a similar way to choosing the number of hidden nodes in a neural network. As long as the kernel function is legitimate, a SVM will operate correctly even if the designer does not know exactly what features of the training data are being used in the kernel-induced feature space.

The SVM methods are binary, thus in the case of multi- class problem one must reduce the problem to a set of multiple binary classification problems. Discrete data presents another problem, although with suitable rescaling good results can be obtained.

//Sample code in R – dataset-iris

install.packages("e1071")

library(e1071)

plot(iris)

plot(iris$Sepal.Length, iris$Sepal.width, col=iris$Species)

plot(iris$Petal.Length, iris$Petal.width, col=iris$Species)

head(iris,5)

attach(iris)

x <- subset(iris, select=-Species)

y <- Species

col <-c("Petal.Length","Petal.width","Species")

svmfit <- svm(Species ~ ., data=iris, kernel ="linear", cost=.1,scale=FALSE)

print(svmfit)

summary(svmfit)

svm\_model1 <- svm(x,y)

summary(svm\_model1)

pred <- predict(svm\_model1,x)

system.time(pred <- predict(svm\_model1,x))

table(pred,y)

svm\_tune <- tune(svm, train.x=x, train.y=y,

              kernel="linear", ranges=list(cost=10^(-1:2), gamma=c(.5,1,2)))

print(svm\_tune)

svm\_model\_after\_tune <- svm(Species ~ ., data=iris, kernel="linear", cost=1, gamma=0.5)

summary(svm\_model\_after\_tune)

pred <- predict(svm\_model\_after\_tune,x)

system.time(predict(svm\_model\_after\_tune,x))

table(pred,y)

1. <http://rischanlab.github.io/SVM.html>
2. <http://www.kdnuggets.com/2016/09/support-vector-machines-concise-technical-overview.html>
3. <https://www.youtube.com/watch?v=ueKqDlMxueE>
4. <http://www.kdnuggets.com/2016/07/support-vector-machines-simple-explanation.html>
5. <https://www.analyticsvidhya.com/blog/2015/10/understaing-support-vector-machine-example-code/>
6. <http://docs.opencv.org/2.4/doc/tutorials/ml/introduction_to_svm/introduction_to_svm.html>