CEEL 82B, Data Science, 2022

Lab 8: Support Vector Machine

**Objective**: Understanding Support Vector Machine algorithm through building SVM algorithm in Python

**Introduction**

An SVM is a numeric classifier. That means that all of the features of the data must be *numeric*, not symbolic. Furthermore, in this class, we'll assume that the SVM is a *binary* classifier: that is, it classifies points as one of two classifications. We'll typically call the classifications "+" and " -".

A trained SVM is defined by two values:

* A normal vector **w** (also called the weight vector), which solely determines the shape and direction of the decision boundary.
* A scalar offset **b**, which solely determines the position of the decision boundary with respect to the origin.

A trained SVM can then classify a point **x** by computing *w · x + b*. If this value is positive, **x** is classified as +; otherwise, **x** is classified as -.

The decision boundary is coerced by support vectors, so called because these vectors (data points) *support* the boundary: if any of these points are moved or eliminated, the decision boundary changes! All support vectors lie on a *gutter*, which can be thought of as a line running parallel to the decision boundary. There are two gutters: one gutter hosts positive support vectors, and the other, negative support vectors.

Note that, though a support vector is always on a gutter, it's **not** necessarily true that every data point on a gutter is a support vector.

Below are the five principle SVM equations, as taught in lecture and recitation. Equations 1-3 define the decision boundary and the margin width, while Equations 4 and 5 can be used to calculate the alpha (supportiveness) values for the training points.

1. **Equation Reference**

[Text

Description automatically generated with medium confidence](https://ai6034.mit.edu/wiki/index.php?title=Image:Lab6_Eqns.png)

**Python Code**

In [1]: **import pandas as pd import numpy as np import matplotlib as mpl**

**import matplotlib.pyplot as plt**

**from sklearn.metrics import** confusion\_matrix

%**matplotlib** inline

*# We'll define a function to draw a nice plot of an SVM*

**def** plot\_svc(svc, X, y, h=0.02, pad=0.25):

x\_min, x\_max = X[:, 0].min()-pad, X[:, 0].max()+pad

y\_min, y\_max = X[:, 1].min()-pad, X[:, 1].max()+pad

xx, yy = np.meshgrid(np.arange(x\_min, x\_max, h), np.arange(y\_min, y\_max Z = svc.predict(np.c\_[xx.ravel(), yy.ravel()])

Z = Z.reshape(xx.shape)

plt.contourf(xx, yy, Z, cmap=plt.cm.Paired, alpha=0.2)

plt.scatter(X[:,0], X[:,1], s=70, c=y, cmap=mpl.cm.Paired)

*# Support vectors indicated in plot by vertical lines*

sv = svc.support\_vectors\_

plt.scatter(sv[:,0], sv[:,1], c='k', marker='x', s=100, linewidths='1') plt.xlim(x\_min, x\_max)

plt.ylim(y\_min, y\_max) plt.xlabel('X1')

plt.ylabel('X2') plt.show()

print('Number of support vectors: ', svc.support\_.size)

# Support Vector Machines

In this lab, we’ll use the SVC module from the sklearn*.*svm package to demonstrate the support vector classifier and the SVM:

In [2]: **from sklearn.svm import** SVC

1. **Support vector Classifier**

The SVC() function can be used to fit a support vector classifier when the argument kernel = ”linear” is used. This function uses a slightly different formulation of the equations we saw in lecture to build the support vector classifier. The c argument allows us to specify the cost of a violation to the margin. When the c argument is **small**, then the margins will be wide and many support vectors will be on the margin or will violate the margin. When the c argument is large, then the margins will be narrow and there will be few support vectors on the margin or violating the margin.

We can use the SVC() function to fit the support vector classifier for a given value of the cost parameter. Here we demonstrate the use of this function on a two-dimensional example so that we can plot the resulting decision boundary. Let’s start by generating a set of observations, which belong to two classes:

In [3]: *# Generating random data: 20 observations of 2 features and divide into tw*

np.random.seed(5)

X = np.random.randn(20,2) y = np.repeat([1,-1], 10)

X[y == -1] = X[y == -1] +1

Let’s plot the data to see whether the classes are linearly separable:

In [4]: plt.scatter(X[:,0], X[:,1], s=70, c=y, cmap=mpl.cm.Paired) plt.xlabel('X1')

plt.ylabel('X2')

Linear or Non Linear?

**Non Linear**

Next, we fit the support vector classifier:

In [5]: svc = SVC(C=1, kernel='linear') svc.fit(X, y)

We can now plot the support vector classifier by calling the plot\_svc() function on the output of the call to SVC(), as well as the data used in the call to SVC():

In [6]: plot\_svc(svc, X, y)

Number of Support vectors? **13**

The region of feature space that will be assigned to the 1 class is shown in light blue, and the region that will be assigned to the +1 class is shown in brown. The decision boundary between the two classes is linear (because we used the argument kernel = ”linear”).

*−*

The support vectors are plotted with crosses and the remaining observations are plotted as circles; we see here that there are 13 support vectors. We can determine their identities as follows:

In [7]: svc.support\_

What if we instead used a smaller value of the cost parameter?

In [8]: svc2 = SVC(C=0.1, kernel='linear') svc2.fit(X, y)

plot\_svc(svc2, X, y)

Number of support vectors? **16**

Now that a smaller value of the c parameter is being used, we obtain a larger number of support vectors, because the margin is now **wider**.

The sklearn*.*grid\_search module includes a a function GridSearchCV() to perform cross- validation. In order to use this function, we pass in relevant information about the set of models that are under consideration. The following command indicates that we want perform 10-fold cross-validation to compare SVMs with a linear kernel, using a range of values of the cost param- eter:

In [9]: **from sklearn.grid\_search import** GridSearchCV

*# Select the optimal C parameter by cross-validation*

tuned\_parameters = [{'C': [0.001, 0.01, 0.1, 1, 5, 10, 100]}]

clf = GridSearchCV(SVC(kernel='linear'), tuned\_parameters, cv=10, scoring= clf.fit(X, y)

/Users/jcrouser/anaconda3/lib/python3.5/site-packages/sklearn/cross\_validation.py: "This module will be removed in 0.20.", DeprecationWarning)

/Users/jcrouser/anaconda3/lib/python3.5/site-packages/sklearn/grid\_search.py:43: DeprecationWarning)

We can easily access the cross-validation errors for each of these models:

In [10]: clf.grid\_scores\_

The GridSearchCV() function stores the best parameters obtained, which can be accessed as follows:

In [11]: clf.best\_params\_

c=0.001 is best according to GridSearchCV.

As usual, the predict() function can be used to predict the class label on a set of test observa- tions, at any given value of the cost parameter. Let’s generate a test data set:

In [12]: np.random.seed(1)

X\_test = np.random.randn(20,2) y\_test = np.random.choice([-1,1], 20)

X\_test[y\_test == 1] = X\_test[y\_test == 1] -1

Now we predict the class labels of these test observations. Here we use the best model obtained through cross-validation in order to make predictions:

In [13]: svc2 = SVC(C=0.001, kernel='linear') svc2.fit(X, y)

y\_pred = svc2.predict(X\_test)

pd.DataFrame(confusion\_matrix(y\_test, y\_pred), index=svc2.classes\_, colum

With this value of c, 14 of the test observations are correctly classified.

Now consider a situation in which the two classes are linearly separable. Then we can find a separating hyperplane using the svm() function. First we’ll give our simulated data a little nudge so that they are linearly separable:

In [14]: X\_test[y\_test == 1] = X\_test[y\_test == 1] -1

plt.scatter(X\_test[:,0], X\_test[:,1], s=70, c=y\_test, cmap=mpl.cm.Paired) plt.xlabel('X1')

plt.ylabel('X2')

Now the observations are **just barely linearly** separable. We fit the support vector classifier and plot the resulting hyperplane, using a very large value of cost so that no observations are misclassified.

In [15]: svc3 = SVC(C=1e5, kernel='linear') svc3.fit(X\_test, y\_test) plot\_svc(svc3, X\_test, y\_test)

Number of support vectors? **3**

No training errors were made and only three support vectors were used. However, we can see from the figure that the margin is very narrow (because the observations that are **not** support vectors, indicated as circles, are very close to the decision boundary). It seems likely that this model will perform poorly on test data. Let’s try a smaller value of cost:

In [16]: svc4 = SVC(C=1, kernel='linear') svc4.fit(X\_test, y\_test) plot\_svc(svc4, X\_test, y\_test)

Number of support vectors? **5**

Using cost = 1, we misclassify a training observation, but we also obtain a much wider margin and make use of five support vectors. It seems likely that this model will perform better on test data than the model with cost = 1e5.

# Support Vector Machine

Support Vectore Machine: Kernel Method

In order to fit an SVM using a **non-linear kernel**, we once again use the SVC() function. However, now we use a different value of the parameter kernel. To fit an SVM with a polynomial kernel we use kernel = ”poly”, and to fit an SVM with a radial kernel we use kernel = ”rbf”. In the former case we also use the degree argument to specify a degree for the polynomial kernel, and in the latter case we use gamma to specify a value of *γ* for the radial basis kernel.

Let’s generate some data with a non-linear class boundary:

In [20]: **from sklearn.model\_selection import** train\_test\_split np.random.seed(8)

X = np.random.randn(200,2)

X[:100] = X[:100] +2

X[101:150] = X[101:150] -2

y = np.concatenate([np.repeat(-1, 150), np.repeat(1,50)])

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, train\_size=0.5,

plt.scatter(X[:,0], X[:,1], s=70, c=y, cmap=mpl.cm.Paired) plt.xlabel('X1')

plt.ylabel('X2')

Attempt to see how one class is kind of stuck in the middle of another class? This suggests that we might want to use a **radial kernel** in our SVM. Now let’s fit the training data using the SVC() function with a radial kernel and *γ* = 1:

In [21]: svm = SVC(C=1.0, kernel='rbf', gamma=1) svm.fit(X\_train, y\_train)

plot\_svc(svm, X\_test, y\_test)

Number of support vectors? **54**

The plot shows that the resulting SVM has a decidedly non-linear boundary. We can see from the figure that there are a fair number of training errors in this SVM fit. If we increase the value of cost, we can reduce the number of training errors:

In [22]: *# Increasing C parameter, allowing more flexibility* svm2 = SVC(C=100, kernel='rbf', gamma=1.0) svm2.fit(X\_train, y\_train)

plot\_svc(svm2, X\_test, y\_test)

Number of support vectors: 36

However, this comes at the price of a more irregular decision boundary that seems to be at risk of overfitting the data. We can perform cross-validation using GridSearchCV() to select the best choice of *γ* and cost for an SVM with a radial kernel:

In [23]: tuned\_parameters = [{'C': [0.01, 0.1, 1, 10, 100],

'gamma': [0.5, 1,2,3,4]}]

clf = GridSearchCV(SVC(kernel='rbf'), tuned\_parameters, cv=10, scoring='a clf.fit(X\_train, y\_train)

clf.best\_params\_

Plot the resulting fit using the plot\_svc() function, and view the test set predictions for this model by applying the predict() function to the test data:

In [24]: plot\_svc(clf.best\_estimator\_, X\_test, y\_test) print(confusion\_matrix(y\_test, clf.best\_estimator\_.predict(X\_test))) print(clf.best\_estimator\_.score(X\_test, y\_test))

Number of support vectors? **44**

Test observations are correctly classified by this SVM?

[[72 2]

[12 14]]

Draw ROC curve and Analyse the result.

**0.86**

**Deliverable:**

You need to write document showing code that fulfills the objective requirement. You must write resulting inference for all intermediate steps, Must create github account and store your experiment write up there prior to next experiment.