SUPPORT VECTOR MACHINE

Report Homework 2

Enrico Cecchetti

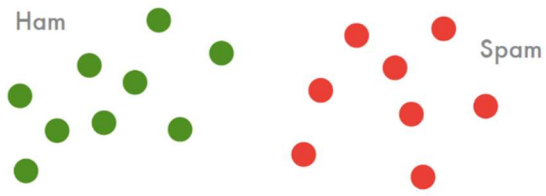
253823

Introduction

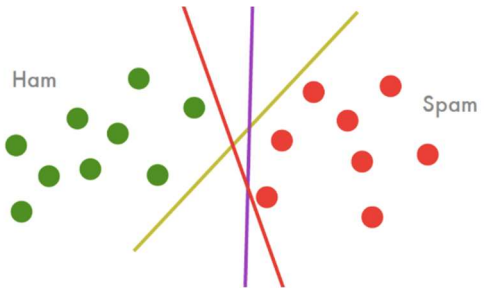
"Support Vector Machine" (SVM) is a supervised machine learning algorithm which can be used for both classification or regression challenges. In this algorithm, we plot each data item as a point in n-dimensional space (where n is number of features you have) with the value of each feature being the value of a particular coordinate. Then, we perform classification by finding the hyper-plane that differentiate the two classes very well (look at the below snapshot).
Support Vector Machine is a frontier which best segregates the two classes (hyper-plane/ line).

Procedure

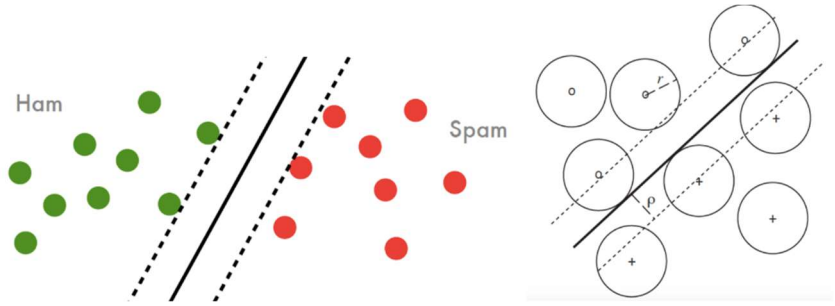
Now the burning question is "How can we identify the right hyper-plane?



You need to remember a thumb rule to identify the right hyper-plane: "Select the hyper-plane which segregates the two classes better".



Here, maximizing the distances between nearest data point (either class) and hyper-plane will help us to decide the right hyper-plane. This distance is called as **Margin**.



Another reason for selecting the hyper-plane with higher margin is robustness. If we select a hyper-plane having low margin, then there is high chance of miss-classification.

Other are:

- Symmetry breaking
- Easy to find for easy problems
- Indipendent on correctly classified instances

SVM IN PYTHON (linear kernel)

The sequential steps that you must follow to realize a SVM are :

- Load data and split them in Test and Train and Validation sets
- Optimize C with the training set
  For each C into (10^-3 % 10^3):
  - Train the SVM in the training set
  - Evaluate the method
- Use the best value of C and evaluate the model

Load data and split them into Test, Train and Validation sets

This extract explain how to do that, we decided to split into 5:2:3

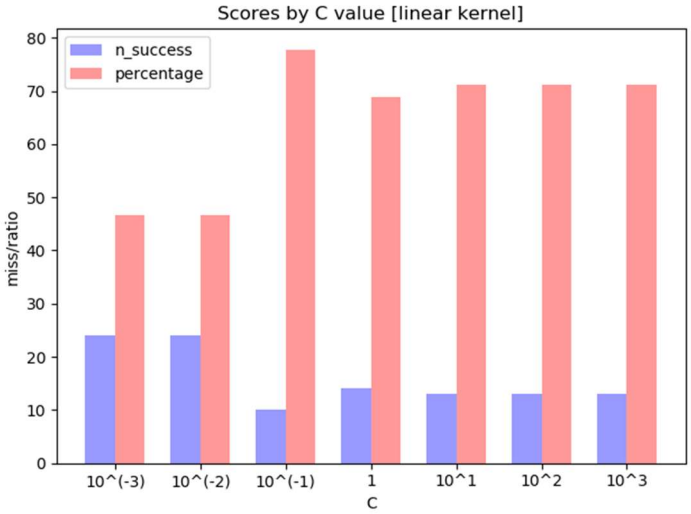
```
# import some data to play with
iris = datasets.load_iris()
# we only take the first two features.
X = iris.data[:, :2]
y = iris.target
X_train, X_temp, y_train, y_temp = train_test_split(X,
                                                    y,
                                                    test_size=0.5,
                                                    random_state=42)
X_validation, X_test, y_validation, y_test = train_test_split(X_temp,
                                                              y_temp,
                                                              test_size=0.4,
                                                              random_state=42)
```

I've separated the 2 passages into devidingf at first into 5:5 and then taking one of the block and dividing it twice into 2:3.

Optimize C with the training set

In the linear SVM the parameter to optimize is C.

But, why? How do the boundaries change?



The C parameter tells the SVM optimization how much you want to avoid misclassifying each training example. For large values of C, the optimization will choose a smaller-margin hyperplane if that hyperplane does a better job of getting all the training points classified correctly. Conversely, a very small value of C will cause the optimizer to look for a larger-margin separating hyperplane, even if that hyperplane misclassifies more points. For very tiny values of C, you should get misclassified examples, often even if your training data is linearly separable.

An exctract of the code about this step is this one

```
for i in C:
    print("scale type C = %.3f" % (i,))
    clf = svm.SVC(kernel='linear', C=i)
    clf.fit(X_train, y_train)

    # create a mesh to plot in
    x_min, x_max = X_train[:, 0].min() - 1, X_train[:, 0].max() + 1
    y_min, y_max = X_train[:, 1].min() - 1, X_train[:, 1].max() + 1
    h = (x_max / x_min) / 100
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))

    plt.subplot(3, 3, k)
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    plt.contourf(xx, yy, Z, cmap=plt.cm.Paired, alpha=0.8)

    plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap=plt.cm.Paired)
    plt.xlabel('Sepal length')
    plt.ylabel('Sepal width')
    plt.xlim(xx.min(), xx.max())
    plt.title('SVC with linear kernel')

    y_pred = clf.predict(X_validation)
    # print("Classifier say is %s when it's %s" % (y_pred, y_test[to_test]))

    mislabled[k-1] = (y_validation != y_pred).sum()
    mislabled_ratio[k-1] = (len(y_validation)-mislabled[k-1])*100/(len(y_validation))
```

```
    # take count of the best storage for show at the end
    if mislabled[k-1] < min_mislabled[1]:
        min_mislabled[0] = i
        min_mislabled[1] = mislabled[k-1]

    print("Number of mislabeled points out of a total %d is : %d"
          % (len(y_validation), mislabled[k-1]))
    print("SVM accuracy is : %.2f %s"
          % ((len(y_validation) - mislabled[k-1]) * 100 / (len(y_validation)), "%"))
    print("-----------------------------------------------------")

    k += 1
plt.show()
```

The only thing I've added is the plot of the boundaries.

The other code is the selection of the Kernel, linear here, and the C value(different for each step) and the validation through:

mislabled[k-1] = (y_validation != y_pred).sum()

It will count every value predicted that has been misslabled, and from that it will print the ratio.

The best value(minor) will be saved in *min_mislabeled* variable for keep the trace of the best C.

Use the best value of C and evaluate the model

```
print("scale type best C = %.3f" % (min_mislabled[0],))
clf = svm.SVC(kernel='linear', C=min_mislabled[0])
clf.fit(X_train, y_train)
[…]
y_pred = clf.predict(X_test)
mislabled_num = (y_test != y_pred).sum()
# print("Classifier say is %s when it's %s" % (y_pred, y_test[to_test]))

print("Number of mislabeled points out of a total %d is : %d"
      % (len(y_test), mislabled_num))
print("SVM accuracy is : %.2f %s"
      % ((len(y_test) - mislabled_num) * 100 / (len(y_test)), "%"))
print("-----------------------------------------------------")
```

Last step is to get the C obtained from the previous step and use it within the test group to evaluate the final model. There is not so much to speak about that, the result obtained are:

---

scale type best C = 0.100

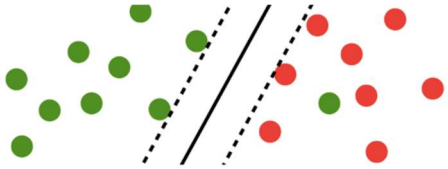
Number of mislabeled points out of a total 30 is : 5

SVM accuracy is : 83.33 %

Outliner

Not always is possible to classify correctly all the points without a misclassification.

For example in the figure we are not able to find an hyperplane that split the 2 classes with a straight line:

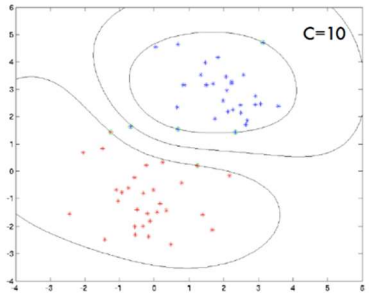


Those points are in the other side territory, as Outliners.

SVM has a feature to ignore outliers and find the hyper-plane that has maximum margin. Hence, we can say, SVM is robust to outliers.

Kernel

In addition to performing linear classification, SVMs can efficiently perform a non-linear classification using what is called the kernel trick, implicitly mapping their inputs into high-dimensional feature spaces.



In its simplest form, the kernel trick means transforming data into another dimension that has a clear dividing margin between classes of data.

SVM IN PYTHON (rbf kernel)

This step is the same of the previous one, but executed with a Non-Linear kernel. The only difference in the code is given from this:

```
clf = svm.SVC(kernel='rbf', C=i, gamma=j)
```

Another value will be important to mesure for the model evaluation, and its gamma.

The gamma parameter defines how far the influence of a single training example reaches, with low values meaning ‘far’ and high values meaning ‘close’. Intuitively, a small gamma value define a Gaussian function with a large variance. In this case, two points can be considered similar even if are far from each other. In the other hand, a large gamma value means define a Gaussian function with a small variance and in this case, two points are considered similar just if they are close to each other.

In conclusion, when gamma is very small, the model is too constrained and cannot capture the complexity or “shape” of the data.

The differences between Linear and RBF could be summarized as follows:

Linear SVM is a parametric model, an RBF kernel SVM isn't, and the complexity of the latter grows with the size of the training set. Not only is it more expensive to train an RBF kernel SVM, but you also have to keep the kernel matrix around, and the projection into this "infinite" higher dimensional space where the data becomes linearly separable is more expensive as well during prediction. Furthermore, you have more hyperparameters to tune, so model selection is more expensive as well! And finally, it's much easier to overfit a complex model.

In those cases, nonlinear kernels are not necessarily significantly more accurate than the linear one.

K-Fold

Cross-validation is a statistical method used to estimate the skill of machine learning models.
Cross-validation is a procedure used to evaluate models on a limited data.
This approach has a single parameter called K that points to the number of groups that a given data sample is to be split into.
Divide training data into **k** equal parts and do **k** rounds of validation. Each fold is then used once as a validation while the k - 1 remaining folds form the training set.

Procedure

The general procedure is as follows:

1. *Shuffle the dataset randomly.*
2. *Split the dataset into k groups*
3. *For each unique group:*
    1. *Take the group as a hold out or test data set*
    2. *Take the remaining groups as a training data set*
    3. *Fit a model on the training set and evaluate it on the test set*
    4. *Retain the evaluation score and discard the model*
4. *Summarize the skill of the model using the sample of model evaluation scores*

This procedure permits to average the accuracy.

K-FOLD IN PYTHON

An extract of the code is the follows :

```
min_mislabled = [10**(-3), 10**(-9), len(y_test)]
for j in gamma:
    k = 1
    for i in C:
        k_fold = KFold(n_splits=5)
        scores = list()
        for train_indices, test_indices in k_fold.split(X_train):
            print("scale type C = %.3f and C = %f" % (i, j))
            clf = svm.SVC(kernel='rbf', C=i, gamma=j)
            clf.fit(X_train, y_train)

            y_pred = clf.predict(X_test)
```

```
            mislabled_num = (y_test != y_pred).sum()
            if mislabled_num < min_mislabled[2]:
                min_mislabled[0] = i
                min_mislabled[1] = j
                min_mislabled[2] = mislabled_num

            print("Number of mislabeled points out of a total %d is : %d"
                  % (len(y_test), mislabled_num))
            print("SVM accuracy is : %.2f %s"
                  % ((len(y_test) - mislabled_num) * 100 / (len(y_test)), "%"))
            print("-------------------------------------------------------")
            k += 1
    plt.show()
```

In this sketch theres no need to devide the dataset onto 5:2:3 anymore thanks to the k-fold approach used. The whole procedure explained previously in theory is all done by the scikit learn function.

k_fold = KFold(n_splits=5)

Bibliography

[1] Barbara Caputo – Class Slides 2018/2019

[2] Analytics Vidhya - Understanding Support Vector Machine algorithm from examples - Sunil Ray, September 13, 2017 - *https://www.analyticsvidhya.com/blog/2017/09/understaing-support-vector-machine-example-code/*

[3] Jason Brownlee, PhD - A Gentle Introduction to k-fold Cross-Validation - May 23, 2018 in Statistical Methods - *https://machinelearningmastery.com/k-fold-cross-validation/*

[4] Scikit learn - sklearn.model_selection.KFold - *https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.KFold.html*