SVM from scratch: step by step in Python

Ford Combs Dec 17, 2019 · 8 min read

How to build a support vector machine using the Pegasos algorithm for stochastic gradient descent.

All of the code can be found here:
https://github.com/frodoCombs/SVM_tutorials

1 What are SVMs?

SVMs date back to the early 1960s when Vladimir Vapnik introduced the *Generalized Portrait Algorithm* (GPA) while working on pattern recognition [1,2]. Over the ensuing years kernels, large margin hyperplanes, and slack variables were developed and some site 1979 as the birth of SVMs with Vapnik's paper on statistical learning [3]. But what exactly is an SVM?

Formally, an SVM consists of a maximally separating hyperplane that can be used to classify data. While SVMs can exist in any number of dimensions, a simple two-dimensional model will be used as an example here because it is easy to visualize. In this data set, the samples have two features so each can be plotted in the XY plane as shown below.The sample classes are represented by the color of the point.

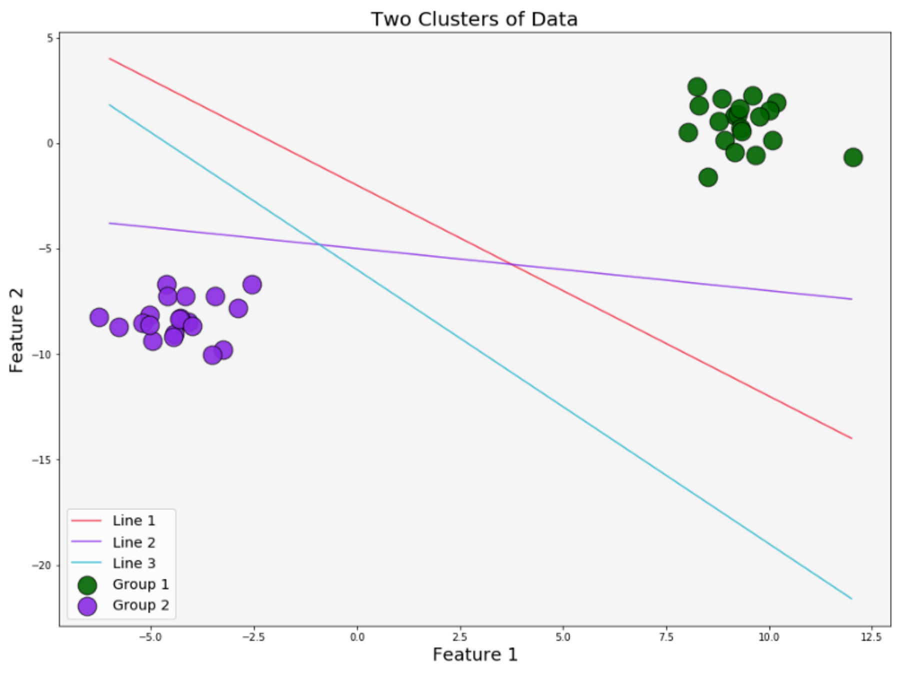


Figure 1. A 40-sample, 2-feature, and 2-class data set plotted in the plane. Feature 1 is shown on the X axis and Feature 2 on the Y axis. The class is shown by the color (green for Group 1 and purple for Group 2). Potential hyperplanes (lines in two dimensions) are shown in red, blue and purple.

> ***The main idea of the SVM is to find the maximally separating hyperplane.***

Figure 1 shows the 40-sample data set with two features (used as X and Y coordinates) and two classes (represented by color). The three lines represent hypothetical SVMs, where a new data point would be classified based on whether it resides on the side of the line containing purple samples or the side of the line containing green samples.

In **Figure 1**, each line would classify new data points differently. Line 3 gives a larger area to the green class so a new point, say (0,-2), would be classified as green even though it was closer to the purple cluster. The main idea of the SVM is to find the **maximally separating hyperplane**. In our example, that line might be Line 1, which seems to lie equally far away from each of the cluster. But how do we find the best hyperplane? **Figure 2** shows the output we are aiming for with our SVM. The classifier has created regions for each of the two classes and will classify new points based on which side of the line it occupies.

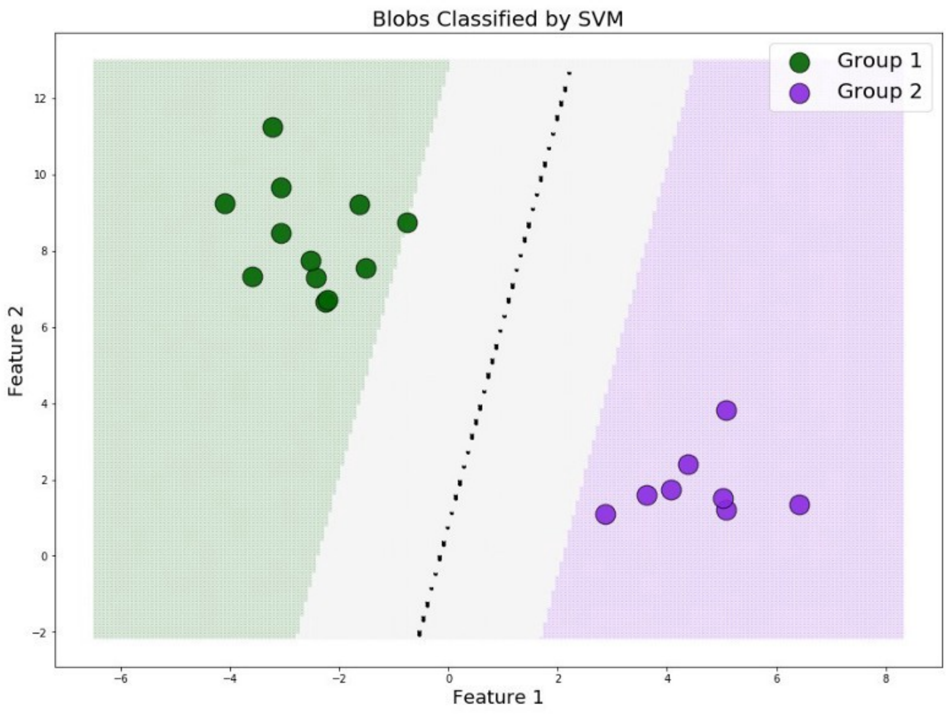


Figure 2. This figure shows the training data points and the regions as classified by the SVM.

2 SVM Objective

A training set, *S*, for an SVM is comprised of *m* samples. The features, ***x***, consist of real numbers and the classifications, *y*, must be -1 or 1.

$$S = \{(\mathbf{x}_i, y_i)\}_{i=1}^{m}$$

The SVM hyperlane is defined by the weight vector, *w*, and the bias, *b*, and is defined as:

$$\vec{w} \cdot \vec{x} - b = 0$$

The incorporation of the bias is further explained in section 4.

In the 2-feature example, this can be rewritten as:

$$((w_1 \times x_1)+(w_2 \times x_2)) - b = 0$$

One **important** aspect of SVMs can be seen in **Figure 2.** The black line represents the line described by the equation above, where the weights dotted with the sample equal 0. This should be compared with the colored regions begin, where the weights dotted with the samples are equal or greater than 1 or -1.

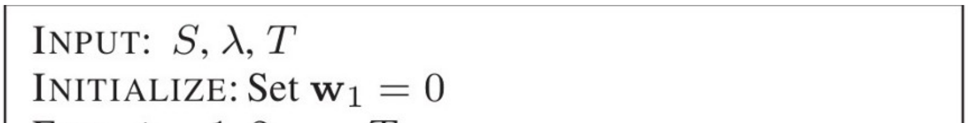
This leads us to the test of correct classification used in training. A correct classification occurs when the weights dotted with the samples multiplied by the classification is greater than 1. Remember that classes must be 1 or -1, so if the SVM has the correct weights then:

1. In the case of a sample with the true class of 1, the dot product of the weights and the sample should be positive, and a positive times 1 is positive.
2. In the case of a sample with the true class of -1, the dot product of the weights and the sample should be negative, and a negative times -1 is positive.

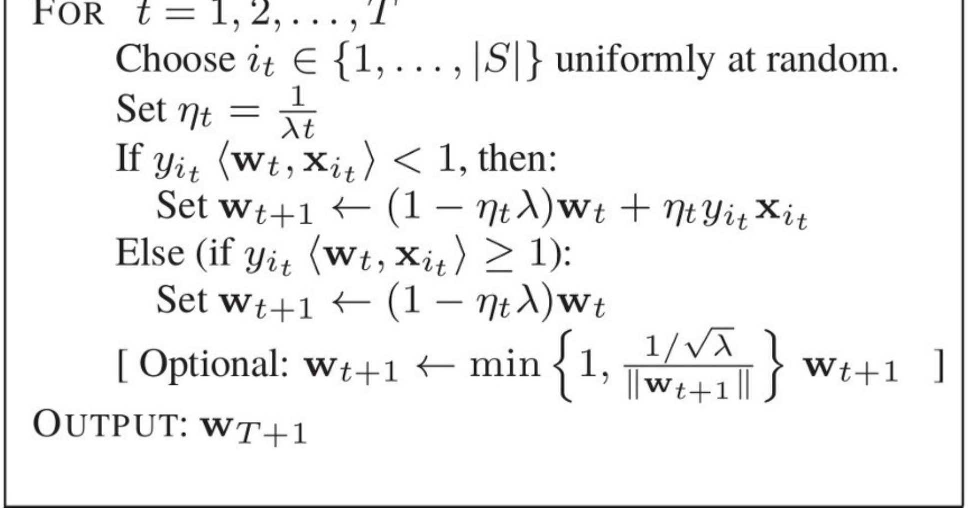
So, if the true class multiplied by the weights dotted with the sample is positive, the SVM has correctly classified the sample.

3 Pegasos Algorithm

There are many methods to find the optimal weight vector and one particularly common one is Sequential Minimal Optimization (SMO) [4]. The SMO algorithm breaks the quadratic programming optimization problem into smaller problems and is very effective at solving SVMs. But, SMO is rather complicated and this example strives for simplicity. The Pegasos algorithm [5] is much simpler and uses stochastic gradient descent (SGD) with a variable step size. SGD is not described here, but there are many good tutorials readily available on the internet. The Pegasos algorithm is shown below in **Figure 3.**



```
INPUT: $S, \lambda, T$
INITIALIZE: Set $\mathbf{w}_1 = 0$
```



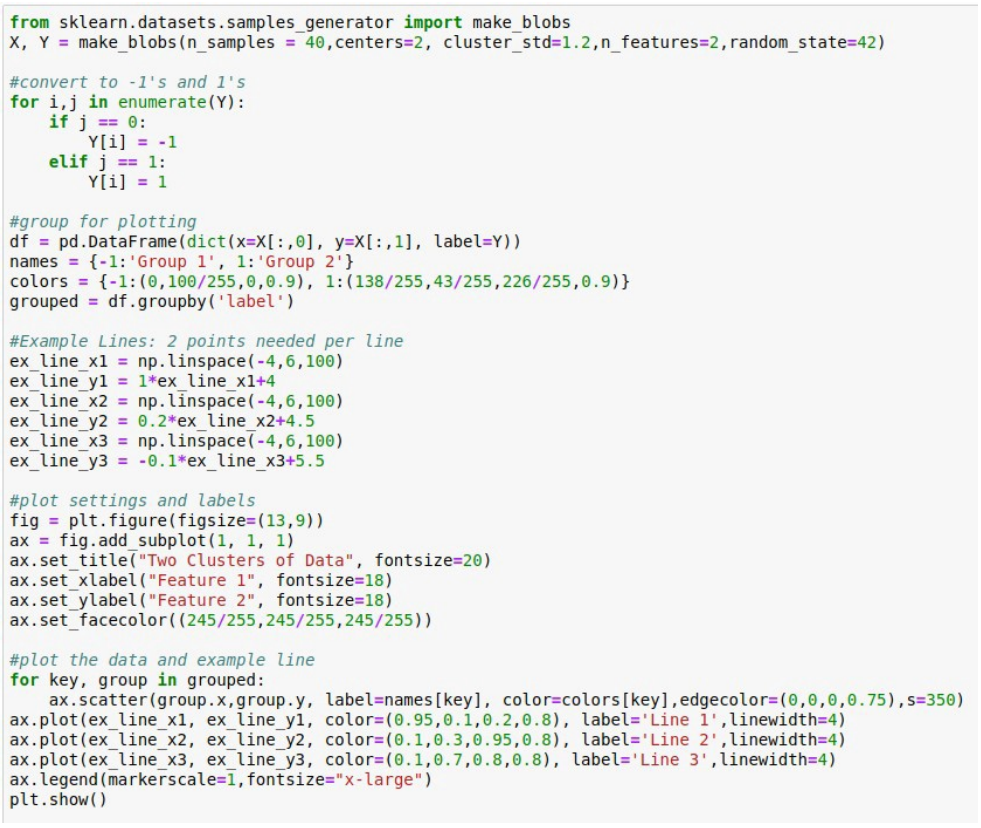
```
FOR $t = 1, 2, \ldots, T$
        Choose $i_t \in \{1, \ldots, |S|\}$ uniformly at random.
        Set $\eta_t = \frac{1}{\lambda t}$
        If $y_{i_t} \langle \mathbf{w}_t, \mathbf{x}_{i_t} \rangle < 1$, then:
            Set $\mathbf{w}_{t+1} \leftarrow (1 - \eta_t \lambda)\mathbf{w}_t + \eta_t y_{i_t} \mathbf{x}_{i_t}$
        Else (if $y_{i_t} \langle \mathbf{w}_t, \mathbf{x}_{i_t} \rangle \geq 1$):
            Set $\mathbf{w}_{t+1} \leftarrow (1 - \eta_t \lambda)\mathbf{w}_t$
        [ Optional: $\mathbf{w}_{t+1} \leftarrow \min\left\{1, \frac{1/\sqrt{\lambda}}{\|\mathbf{w}_{t+1}\|}\right\} \mathbf{w}_{t+1}$ ]
OUTPUT: $\mathbf{w}_{T+1}$
```

Figure 3. The Pegasos algorithm as described in pseudocode [5].

4 Step by Step in Python

All of the code can be found here:
https://github.com/frodoCombs/SVM_tutorials

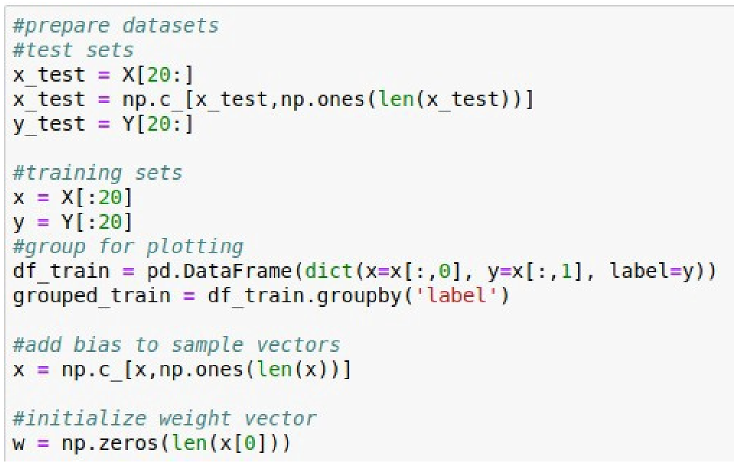
For this example, the data set was created using the make_blobs function from sklearn as shown below. It creates an X and Y, where the X contains the features and the Y contains the classifications. The function outputs classifications of 0 and 1. These classes need to be converted to -1's and 1's. The rest of the code block below is basically used for plotting and drawing **Figure 1**.



```
from sklearn.datasets.samples_generator import make_blobs
X, Y = make_blobs(n_samples = 40,centers=2, cluster_std=1.2,n_features=2,random_state=42)

#convert to -1's and 1's
for i,j in enumerate(Y):
    if j == 0:
        Y[i] = -1
    elif j == 1:
        Y[i] = 1

#group for plotting
df = pd.DataFrame(dict(x=X[:,0], y=X[:,1], label=Y))
names = {-1:'Group 1', 1:'Group 2'}
colors = {-1:(0,100/255,0,0.9), 1:(138/255,43/255,226/255,0.9)}
grouped = df.groupby('label')

#Example Lines: 2 points needed per line
ex_line_x1 = np.linspace(-4,6,100)
ex_line_y1 = 1*ex_line_x1+4
ex_line_x2 = np.linspace(-4,6,100)
ex_line_y2 = 0.2*ex_line_x2+4.5
ex_line_x3 = np.linspace(-4,6,100)
ex_line_y3 = -0.1*ex_line_x3+5.5

#plot settings and labels
fig = plt.figure(figsize=(13,9))
ax = fig.add_subplot(1, 1, 1)
ax.set_title("Two Clusters of Data", fontsize=20)
ax.set_xlabel("Feature 1", fontsize=18)
ax.set_ylabel("Feature 2", fontsize=18)
ax.set_facecolor((245/255,245/255,245/255))

#plot the data and example line
for key, group in grouped:
    ax.scatter(group.x,group.y, label=names[key], color=colors[key],edgecolor=(0,0,0,0.75),s=350)
ax.plot(ex_line_x1, ex_line_y1, color=(0.95,0.1,0.2,0.8), label='Line 1',linewidth=4)
ax.plot(ex_line_x2, ex_line_y2, color=(0.1,0.3,0.95,0.8), label='Line 2',linewidth=4)
ax.plot(ex_line_x3, ex_line_y3, color=(0.1,0.7,0.8,0.8), label='Line 3',linewidth=4)
ax.legend(markerscale=1,fontsize="x-large")
plt.show()
```

To incorporate the bias a new feature is added; its value is 1 for every sample.

The code below splits the data set into testing and training sets and initializes the weight vector. Each sample in the data set initially has two features and a class. But, a feature of value 1 is added to every sample regardless of class. Since the data set now contains 3 features, the weight vector also needs three values. When the weight vector is dotted with the feature vector, since the third feature value is always 1, the third weight will become the intercept, often called the *bias*. If this is not done, the hyperplane will always go through the origin, and this is unlikely to be the maximally separating hyperplane.



```
#prepare datasets
#test sets
x_test = X[20:]
x_test = np.c_[x_test,np.ones(len(x_test))]
y_test = Y[20:]

#training sets
x = X[:20]
y = Y[:20]
#group for plotting
df_train = pd.DataFrame(dict(x=x[:,0], y=x[:,1], label=y))
grouped_train = df_train.groupby('label')

#add bias to sample vectors
x = np.c_[x,np.ones(len(x))]

#initialize weight vector
w = np.zeros(len(x[0]))
```

The justification for adding a feature vector of 1's to the data set is explained again below. Since the third feature is always 1, the third weight value becomes the bias also known as the intercept.

$$((w_1 \times x_1) + (w_2 \times x_2) + (w_3 \times x_3)) = 0$$

$$since\ x_3 = 1, ((w_1 \times x_1)+(w_2 \times x_2)+w_3) = 0$$

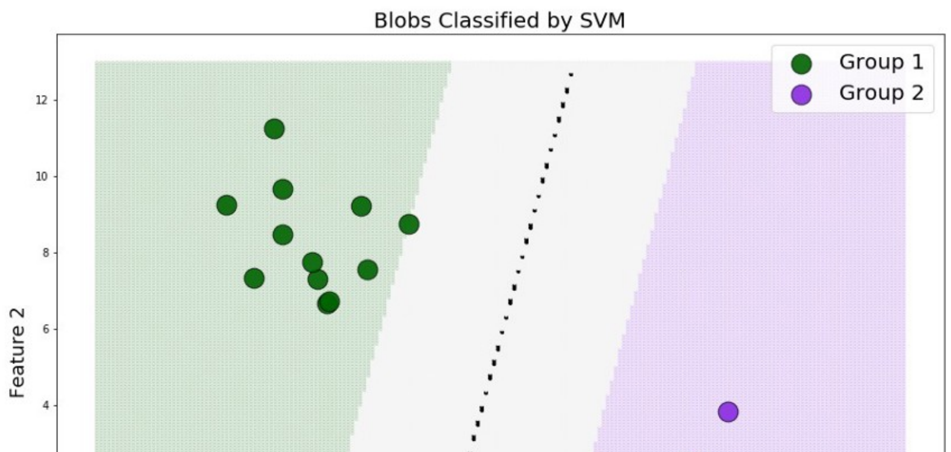
> ***If this is not done, the hyperplane will always go through the origin, and this is unlikely to be the maximally separating hyperplane.***

Below, is the code for the Pegasos algorithm [5]. The learning rate is 0.001 and held in the variable *lam*. The margin_current and margin_previous keeps track of the size of the margin (*remember SVMs want to maximize the margin*). The pos_support_vectors and neg_support_vectors variables will keep track of the number of support vectors found. In 2D, there will be at least two support vectors. (*Support vectors are the samples the lie directly on the margin, so a maximally wide margin will touch at least one of each class*). The variable *t* will hold the time step and the other variables are self explanatory.

At each iteration of the loop, the margin and time step are updated and the number of support vectors at that iteration are set to zero. Recall the pseudocode in **Figure 3**, next the update value is calculated. Eta or η in **Figure 3** is equal to one over lamda times t. Then *fac* is computed; it holds the full update, (1-ηλ)**w**, as seen in **Figure 3**. It is computed outside of the inner loop to save computational time. Then before entering the inner loop, the data set is shuffled, this isn't really necessary in this case, but in the *online* Pegasos algorithm only one sample is used for the update at each time step and should be chosen randomly. Here all the samples are looped over. In the inner loop, each sample is checked to see if it is a support vector, then checked to see if it is correctly classified by the current weight vector. The weight vector update is different for misclassifications. Finally, there is a little part of code that I inserted to check for *convergence*. In this case, I just check if an arbitrary amount of time steps have pass, if the margin has changed only by a small amount, and if there are at least one of each support vectors. If so, then the algorithm is done and the weight vector has been computed.

```
#learning rate
lam = 0.001
#array of number for shuffling
order = np.arange(0,len(x),1)

margin_current = 0
margin_previous = -10

pos_support_vectors = 0
neg_support_vectors = 0

not_converged = True
t =0
start_time = time.time()

while(not_converged):
    margin_previous = margin_current
    t += 1
    pos_support_vectors = 0
    neg_support_vectors = 0

    eta = 1/(lam*t)
    fac = (1-(eta*lam))*w
    random.shuffle(order)
    for i in order:
        prediction = np.dot(x[i],w)

        #check for support vectors
        if (round((prediction),1) == 1):
            pos_support_vectors += 1
            #pos support vec found
        if (round((prediction),1) == -1):
            neg_support_vectors += 1
            #neg support vec found

        #misclassification
        if (y[i]*prediction) < 1 :
            w = fac + eta*y[i]*x[i]
        #correct classification
        else:
            w = fac

    if(t>10000):
        margin_current = np.linalg.norm(w)
        if((pos_support_vectors > 0)and(neg_support_vectors > 0)and((margin_current - margin_previous) < 0.01)):
            not_converged = False

#print running time
print("--- %s seconds ---" % (time.time() - start_time))
```

For printing out figures that display the SVM, I like to create a grid and color each point based on its classification. All of this can be seen in the code that is linked above. In **Figure 4**, it can be seen that the SVM correctly classified every sample, and the margins can be seen as green and purple regions. In **Figure 5**, it can be seen that every sample in the test set was also correctly classified.



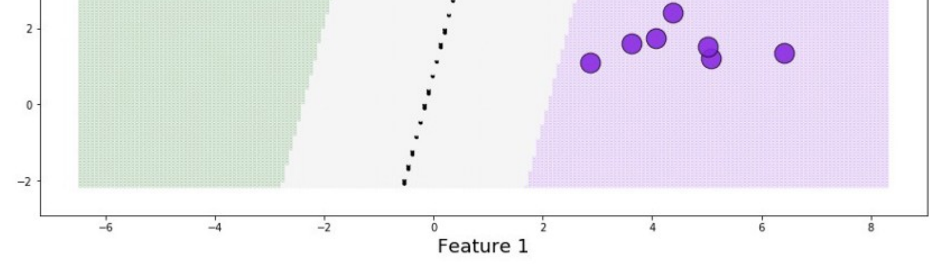


Figure 4. Training data set classification by SVM.

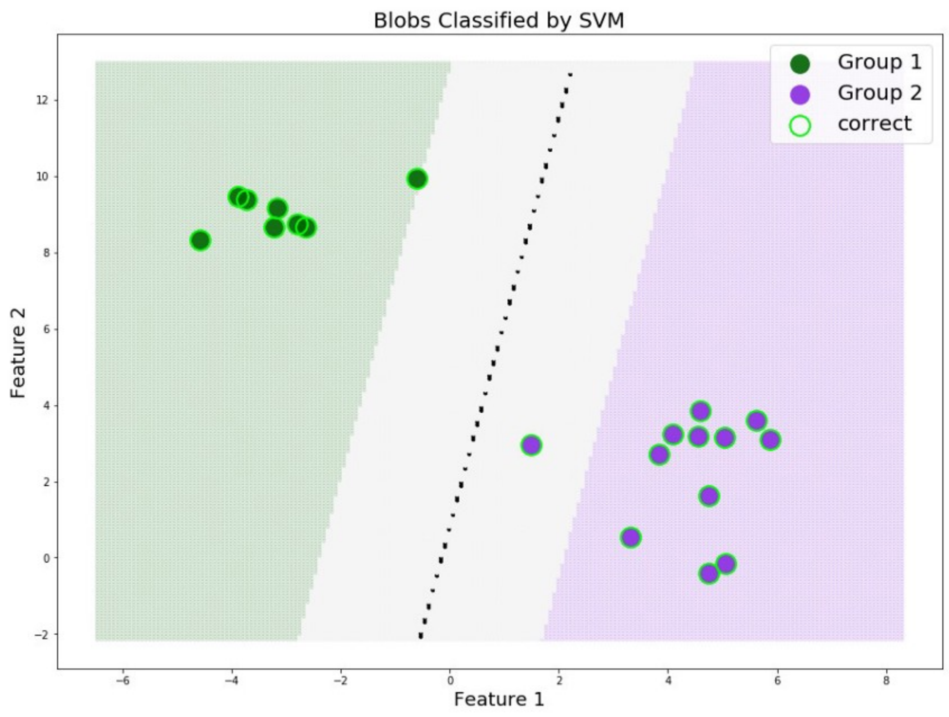


Figure 5. Test data set classification by SVM.

References

1. VAPNIK, V., and A. LERNER, 1963. Pattern recognition using generalized portrait method. *Automation and Remote Control*, **24**, 774–780.
2. VAPNIK, V., and A. CHERVONENKIS, 1964. A note on one class of perceptrons. *Automation and Remote Control*, **25**.
3. VAPNIK, V., 1979. *Estimation of Dependences Based on Empirical Data* [in Russian]. Moscow: Nauka.
4. Platt, J.,1998. *Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machines.* Microsoft Research Technical Report.
5. Shalev-Shwartz S, Singer Y, Srebro N, Cotter A (2011) Pegasos: Primal estimated sub-gradient solver for SVM. Math Program. https://doi.org/10.1007/s10107-010-0420-4

Machine Learning