# SVMs for feature selection

Robert Timpe
rtimpe@cs.ucsd.edu

## ABSTRACT

Feature selection is an important problem in machine learning. Data sets often have redundant or unnecessary features/dimensions that only serve to make classification more difficult. Feature selection aims to identify and remove these features. By using SVMs to do feature selection, we can remove bad features while simultaneously learning a good classification rule. Traditional SVMs use the $l_2$ norm as a form of regularization. This gives rise to a nice convex formulation with an interesting dual problem. To get better feature selection performance, we can replace the $l_2$ norm with the $l_0$ norm, which counts the number of non-zero entries in a vector. Minimizing this quantity while simultaneously minimizing classification error will clearly lead to good feature selection performance. Unfortunately, this makes the problem non-convex, and minimizing this problem directly is NP-HARD [1]. Instead, various approximations have been suggested, as in [3] and [4]. In this project, I investigate these two methods for feature selection and compare their performance to traditional SVMs.

## 1. INTRODUCTION

In a traditional SVM, we are interested in solving the problem

$$\begin{aligned} \min_{w,b,\xi} \quad & ||w||_2^2 + \lambda \sum_{i=1}^{m} \xi_i \\ \text{such that} \quad & y_i(w^T x_i + b) \geq 1 - \xi_i, \;\; i = 1, ..., m \\ & \xi \succeq 0 \end{aligned} \tag{1}$$

Here, we have input data $(x_1, ..., x_m) \in \mathbb{R}^n$ with labels $y_i \in \{0, 1\}$. $\lambda$ is a hyperparameter that controls the importance of regularization vs classification accuracy. Note that this is a quadratic program. We then classify a new point $x$ according to the rule $\text{sign}(w^T x + b)$. Minimizing the $l_2$ norm will tend to result in a vector $w$ with no very large components, but will not necessarily set many of its elements to 0. Replacing the $l_2$ norm with an $l_0$ norm will favor solutions with many zeros in the elements of $w$. Of course, the $l_0$ norm is not a true norm - it does not satisfy the triangle inequality $||x + y||_0 \leq ||x||_0 + ||y||_0$. For example, consider the Unfortunately, solving the resulting problem directly is NP-HARD [1]. Instead, we consider various approximations, discussed in the next section.

## 2. APPROXIMATING THE $L_0$ NORM

### 2.1 Feature Selection via Concave Minimization

In [3], the Feature Selection via Concave Minimization (FSV) algorithm is introduced. This uses the approximation

$$||x||_0 \approx \sum_{i=1}^{n} 1 - e^{-\alpha |x_i|} \tag{2}$$

where $\alpha \in \mathbb{R}_+$ is an approximation parameter. Note that, for larger values of $\alpha$ the approximation becomes more accurate. In fact, for large enough values of $\alpha$, this approximation has a solution equal to the non-smooth optimal solution [2] [3]. Unfortunately, using larger values of $\alpha$ makes the problem more difficult to solve and increases the computation time needed [3].

Note that, while the approximation (2) gives a nice smooth approximation to the $l_0$ norm, it is a concave function. We can show this using the composition rules for convex functions. Letting

$$\begin{aligned} h(x) &= e^{-x} \\ g(x) &= |x| \\ f(x) &= h(g(x)) \end{aligned}$$

we see that $g(x)$ is convex (it is a norm) and $h(x)$ is concave (its second derivative is always non-negative). Therefore, $f(x) = e^{-|x|}$ is a convex function and $-f(x)$ is a concave function. This means that if we replace the $l_2$ norm with this approximation, then problem (1) will no longer be a convex optimization problem. This is the FSV problem from [3], stated below.

$$\begin{aligned} \min_{w,b,\xi,v} \quad & \frac{1-\lambda}{m} \sum_{i=1}^{m} \xi_i + \lambda \sum_{i=1}^{n} (1 - e^{-\alpha v}) \\ \text{such that} \quad & y_i(w^T x_i + b) \geq 1 - \xi_i, \;\; i = 1, ..., m \\ & \xi \succeq 0 \\ & -v \leq w \leq v \end{aligned} \tag{3}$$

Note that we have removed the absolute value by introduc-

ing the new optimization variable $v$. Also we require that $\lambda \in [0,1]$. We can then find an approximate solution to (3) by repeatedly solving

$$\begin{aligned} \min_{w,b,\xi,v} \quad & \frac{1-\lambda}{m}\sum_{i=1}^{m}\xi_i + \lambda\alpha(e^{-\alpha v^i})^T(v-v^i) \\ \text{such that} \quad & y_i(w^T x_i + b) \geq 1-\xi_i, \;\; i=1,...,m \\ & \xi \succeq 0 \\ & -v \leq w \leq v \end{aligned} \tag{4}$$

where $v^i$ is the optimal solution to $v$ from the previous iteration. Note that this is an LP - $e^{-\alpha v^i}$ is a constant (vector), so all of the optimization variables appear linearly. The full algorithm is given below [3]

**Algorithm 1** Successive Linearization Algorithm for FSV

1: **procedure** SLA
2: Initialize $w^0, b^0, \xi^0, v^0$ randomly
3: **for** $i = 1, 2, ...$ **do**
4: Solve (4)
5: Let $w^i, b^i, \xi^i, v^i$ be the optimal values found
6: **if** $\frac{1-\lambda}{m}(\sum_{j=1}^{m}(\xi_j^i - \xi_j^{i-1})) + \lambda\alpha(e^{-\alpha v^i})^T(v^i - v^{i-1}) = 0$ **then**
7: End

## 2.2 Direct $l_0$ norm optimization

In [4], a method of optimizing the $l_0$ norm directly is introduced. This method is based on an EM algorithm, and so will converge to a local (though not necessarily global) minimum. They show that, given a good starting point, the algorithm asymptotically converges to the true $l_0$ norm. The algorithm involves iteratively solving the following problem

$$\begin{aligned} \min_{w,b,\xi} \quad & \lambda\sum_{i=1}^{m}\xi_i + w^T\Lambda^{t-1}w \\ \text{such that} \quad & y_i(w^T x_i + b) \geq 1-\xi_i, \;\; i=1,...,m \\ & \xi \succeq 0 \end{aligned} \tag{5}$$

and we define

$$\Lambda^{t-1} = diag(\frac{1}{(w_1^{(t-1)})^2}, ..., \frac{1}{(w_n^{(t-1)})^2})$$

This problem is solved iteratively until a stable $w$ and $b$ are found. Note that this is a quadratic program. $\Lambda^{t-1}$ is a constant with respect to the current parameters, and the other optimization variables appear linearly. Additionally, $\Lambda^{t-1}$ is a diagonal matrix with positive entries, so it is positive semidefinite. Therefore, the function $w^T\Lambda^{t-1}w$ is convex.

### 2.2.1 Dual problem

In addition to the primal problem, the authors in [4] also give a method for solving the dual problem. As with traditional SVMs, solving the dual problem allows us to use the kernel trick to efficiently represent inputs $x$ in some higher dimensional space with the transformation $\Phi(x)$. We then make decisions according to the rule $sign(\sum_{i=1}^{m}\alpha_i\Phi(x_i))$. We can accomplish this by iteratively solving the following problem

$$\begin{aligned} \min_{\alpha,b,\xi} \quad & \lambda\sum_{i=1}^{m}\xi_i + \alpha^T\Lambda^{(t-1)}\alpha \\ \text{such that} \quad & y_i(w^T\Phi(x_i) + b) \geq 1-\xi_i, \;\; i=1,...,m \\ & \xi \succeq 0 \end{aligned}$$

where

$$\Lambda^{(t-1)} = diag(\frac{1}{(\alpha_1^{(t-1)})^2}, ..., \frac{1}{(\alpha_n^{(t-1)})^2})$$

Again, we can easily see that this is a QP since $\Lambda^{(t-1)}$ is a constant, psd matrix. To make the use of the kernel trick more explicit, we can re-write this problem as

$$\begin{aligned} \min_{\alpha,b,\xi} \quad & \lambda\sum_{i=1}^{m}\xi_i + \alpha^T\Lambda^{(t-1)}\alpha \\ \text{such that} \quad & y_i(K_i\alpha + b) \geq 1-\xi_i, \;\; i=1,...,m \\ & \xi \succeq 0 \end{aligned}$$

where $K_i$ is the $i^{th}$ row of the kernel matrix $K$ with $K_{ij} = \Phi(x_i)\cdot\Phi(x_j)$.

While the primal problem attempts to maximize the number of zero elements in $w$, the dual problem tries to maximize the number of $\alpha_i$ that are zero (i.e. minimize the number of support vectors). This can be useful in cases where there are more features than training points. It also allows us to use a kernel function, which may improve classification accuracy by using a non-linear decision boundary (in the original input space). But it is not a form of feature selection, so we do not consider it as carefully here.

# 3. RESULTS

## 3.1 Toy Data

We first explore the performance of various formulations on a toy dataset. This dataset has 2 classes and 3 features. The first two features are sampled from a two-dimensional gaussian distribution centered at either $(0,1)$ or $(1,0)$, each with covariance matrix

$$\Sigma = \begin{bmatrix} .5 & 0 \\ 0 & .5 \end{bmatrix}$$

The third dimension is a number chosen uniformly at random from between 1 and 500. The idea is to have 2 features that are useful for classification (i.e. the two gaussian features) and one that is just noise.

| $\lambda$ | Accuracy | Value of $w_3$ |
|---|---|---|
| .1 | 83.5 | 2.36e-3 |
| .2 | 83.5 | 2.16e-3 |
| .3 | 83.5 | 2.16e-3 |
| .4 | 83.5 | 1.94e-3 |
| .5 | 83.5 | 1.75e-3 |
| .7 | 75.5 | 7.04e-4 |
| .8 | 74.0 | 1.58e-13 |
| .9 | 50.0 | 1.74e-14 |

**Table 1: FSV results with different values of $\lambda$**

| $\alpha$ | Accuracy | Value of $w_3$ |
|---|---|---|
| .01 | 83.5 | 2.40e-3 |
| .1 | 84.0 | 1.34e-3 |
| 1 | 77.5 | -1.32e-4 |
| 2 | 74.5 | 5.78e-4 |
| 4 | 85.0 | 1.04e-3 |
| 10 | 83.5 | 1.94e-3 |
| 20 | 83.5 | 1.04e-4 |
| 30 | 83.0 | -1.16e-13 |
| 50 | 83.0 | -1.04e-14 |
| 100 | 83.5 | 2.36e-3 |

**Table 2: FSV results with different values of $\alpha$**

Table 1 shows the results of the FSV algorithm with various values of $\lambda$ (and $\alpha = 5$). Lower values of $\lambda$ give higher accuracy, while higher values result in values of $w$ with more near-zero entries. At $\lambda = 9$, $w$ is essentially all 0's. Figure 1 shows the decision boundary found by FSV on this data. This figure only shows the first two dimensions of the data.

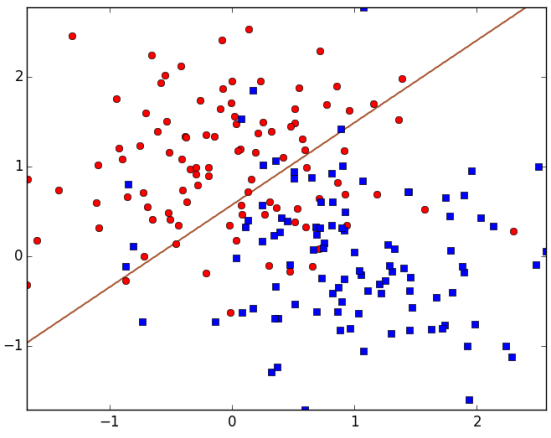


**Figure 1: Toy data with FSV decision boundary for $\lambda$=.1 and $\alpha$=5**

Table 2 shows the results of FSV with different values of $\alpha$ (with $\lambda = .5$). The results are somewhat inconsistent. For a certain range of values (around 30-50), FSV gives different values of $w_3$. These changes are likely due to the non-convexity of the original problem. Different values of $\alpha$ give different approximations, which have minima near different local minima of the original problem.

| $\gamma$ | Accuracy | Value of $w_3$ |
|---|---|---|
| 1.5 | 74.0 | -7.02e-14 |
| 2 | 83.0 | -2.50e-13 |
| 3 | 83.0 | -3.26e-14 |
| 4 | 83.0 | 4.12e-13 |
| 6 | 83.0 | 4.09e-37 |

**Table 3: FSV results with different values of $\gamma$**

| $\lambda$ | Accuracy | Value of $w_3$ | Value of $w_1$ |
|---|---|---|---|
| .01 | 50.5 | 1e-5 | 1e-5 |
| .1 | 77.0 | 1e-5 | 1e-5 |
| 1 | 77.0 | 1e-5 | 1e-5 |
| 10 | 77.0 | 1e-5 | 1e-5 |
| 100 | 77.0 | 1e-5 | 1e-5 |
| 1000 | 77.0 | 1e-5 | 1e-5 |

**Table 4: Direct $l_0$ norm optimization results for different values of $\lambda$**

In addition to using a fixed alpha, the authors in [3] suggest increasing $\alpha$ with each iteration (although they do not give results of this modified algorithm). Table 3 shows the results of changing $\alpha$ with each iteration (and $\lambda = .5$). We update $\alpha$ as $\alpha = \gamma\alpha$ after each loop iteration. For larger values of $\gamma$, this gives very good results. The accuracy is high and $w_3$ becomes extremely small. Unlike the results from changing $\lambda$, we are decreasing the importance of the unhelpful feature while still maintaining high accuracy.

We also test the direct $l_0$ optimization method on this dataset. Unfortunately, the algorithm as stated runs into numerical issues computing $\Lambda^{t-1}$. As the $w^{t-1}$ terms approach zero, taking $\frac{1}{w^{t-1}}$ becomes problematic. To fix this issue, I clamp the values of $w^{t-1}$ at 1e-5. This still allows the elements of $w$ to get very close to 0, but avoids numerical issues.

Table 4 shows the results of direct $l_0$ optimization for different values of $\lambda$. Interestingly, the value of $\lambda$ has almost no effect, other than at extremely small values. As $\lambda$ becomes larger, the model gets all of its classification accuracy from the value of $w_2$. This is somewhat surprising, as larger values of $\lambda$ put more emphasis on classification accuracy, instead of minimizing the $l_0$ norm.

Finally, Table 5 shows the results of training a traditional SVM on this dataset. Unsurprisingly, it gets similar accuracy to the $l_0$ norm methods, but does not shrink $w_3$ quite as much, even for very small values of $\lambda$. This makes sense because, when using the $l_2$ norm, there is less benefit to shrinking an already small component.

### *3.1.1 Dual problem*

Table 6 shows the results of training the dual problem with a linear kernel on the first two dimensions of this dataset. Even for larger values of $\lambda$, there are only 3 or 4 support vectors. The dual approach does not work well when the third (noise) dimension is included. This is because it is difficult to choose a small number of support vectors that both do a good job of classifying the first two dimensions and the third dimension. So instead, it just ends up setting all of the $\alpha$'s to 0.

| $\lambda$ | Accuracy | Value of $w_3$ |
|---|---|---|
| .001 | 58.0 | 2.77e-3 |
| .1 | 84.0 | 1.41e-3 |
| 1 | 83.5 | 2.37e-3 |
| 5 | 83.5 | 1.99e-3 |
| 10 | 83.5 | 2.34e-3 |
| 50 | 84.0 | 3.62e-3 |
| 100 | 84.0 | 5.79e-3 |

**Table 5: Traditional SVM ($l_2$ regularization) results for different values of $\lambda$**

| $\lambda$ | Accuracy |
|---|---|
| .01 | 50.0 |
| .1 | 81.0 |
| 1 | 83.0 |
| 2 | 84.0 |

**Table 6: Direct optimization of dual problem for different values of $\lambda$**

Figure 2 shows the decision boundary obtained when using a gaussian (RBF) kernel. That is, the kernel is given by

$$k(x, y) = exp(\frac{||x - y||_2^2}{2\sigma^2})$$

where $\sigma$ is a hyperparameter that was set to 1. This gives a nonlinear decision boundary that gets 83.5% accuracy.

## 3.2 Census Data

We also test these algorithms on the Census Income dataset from the UCI machine learning repository [5]. This dataset contains records about 48,842 people from the 1994 census. Each person has 14 features, such as age, gender, occupation, and so on. The task is to predict whether someone makes more or less than $50,000 per year. For these experiments, I used a subset of the data - 600 datapoints for training and 2000 for the test set (1000 from each class).

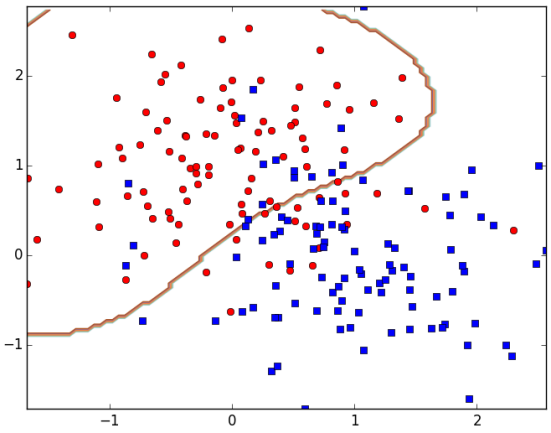


**Figure 2: Toy data with gaussian kernel**

| Algorithm | Train Accuracy | Test Accuracy |
|---|---|---|
| FSV | 75.3 | 72.3 |
| Direct $l_0$ | 56.7 | 53.6 |
| SVM | 68.8 | 67.8 |

**Table 7: Results of different algorithms on the census dataset**

Table 7 shows the results of each of the algorithms on the census dataset. The FSV dataset performs relatively well because it finds which dimensions should be set to 0. Several of the dimensions in this dataset are categorical, and therefore likely to not be helpful for an SVM. The FSV algorithm tends to put a more weight on continuous features (i.e. age, hours worked, years of education, etc.) and near-zero weights on categorical fields. The traditional $l_2$ SVM, on the other hand, puts more weight on all of the features, and its performance suffers as a result. Again, this is a result of the $l_2$ norm not encouraging features to be further decreased once they are already small. Finally, the direct $l_0$ optimization goes too far by setting all but one of its dimensions to 0. The only non-zero weight is for gender, which has the highest predictive value of any single feature.

# 4. CONCLUSION

In this project, I explored two different methods for solving SVMs with the $l_0$ norm for regularization. Because the $l_0$ norm is not actually a norm, it is not tractable to solve the resulting problem directly, so both of these methods are approximations of the true problem. The FSV method [3] is based on a concave approximation of the $l_0$ norm, while the method of Huang et al. [4] directly optimizes the $l_0$ norm but is subject to local minima.

The usefulness of these methods was first demonstrated on a toy dataset with two useful features and one noise feature. On this dataset, both $l_0$ methods and the traditional $l_2$ SVM all got around the same accuracy, although the direct method did get slightly lower accuracy due to its tendency to set all but one of the feature weights to 0. I also showed how different values of each of the various parameters affects performance.

Next, I tested these methods against a real dataset of US census data. Here the difference between the methods was more pronounced. The FSV method achieved the best accuracy by relying almost entirely on continuous (rather than categorical) features. This makes sense, since categorical features, while not quite noise, are less likely to be helpful for SVM classification. Indeed, the $l_2$ SVM was not able to as successfully ignore categorical features, and its performance suffered. The direct $l_0$ optimization approach had the lowest accuracy because it set all but one of the feature weights to 0. Even with larger values of $\lambda$, it tended to minimize the $l_0$ norm over minimizing classification error. Based on these results, it seems that minimizing $l_0$ norm is indeed an effective form of feature selection, and can in some cases lead to better classification performance.

Finally, I also briefly explored the dual formulation of this problem. In this formulation, we are minimizing the number of support vectors rather than the number of feature weights,

so it is not really a form of feature selection. Nevertheless, this approach can be useful when the number of features is large relative to the number of training points. Also, the dual formulation allows us to use the kernel trick to obtain non-linear decision boundaries.

## 5. REFERENCES


[1] E. Amaldi and V. Kann. On the approximability of minimizing nonzero variables or unsatisfied relations in linear systems. *Theoretical Computer Science*, 209(1):237–260, 1998.

[2] P. Bradley, O. Mangasarian, and J. Rosen. Parsimonious least norm approximation. *Computational Optimization and Applications*, 11(1):5–21, 1998.

[3] P. S. Bradley and O. L. Mangasarian. Feature selection via concave minimization and support vector machines. In *ICML*, volume 98, pages 82–90, 1998.

[4] K. Huang, I. King, and M. R. Lyu. Direct zero-norm optimization for feature selection. In *Data Mining, 2008. ICDM'08. Eighth IEEE International Conference on*, pages 845–850. IEEE, 2008.

[5] M. Lichman. UCI machine learning repository - census income. `https://archive.ics.uci.edu/ml/datasets/Census+Income`, 2013.


## APPENDIX

## A. SOURCE CODE FOR FSV

```
import numpy as np
import matplotlib.pyplot as plt
import cvxpy as cvx
import math
import random
from sklearn import svm

def sla(A, B, lambduh, alpha):
    vexp = np.vectorize(math.exp)
    n = A.shape[1]
    m = A.shape[0]
    k = B.shape[0]

    w = cvx.Variable(n)
    b = cvx.Variable()
    y = cvx.Variable(m)
    z = cvx.Variable(k)
    v = cvx.Variable(n)

    wi = np.random.rand(n, 1)
    bi = random.random()
    yi = np.random.rand(m, 1)
    zi = np.random.rand(k, 1)
    vi = np.random.rand(n, 1)

    constraints = [-A*w + np.ones((m)) * b + np.ones((m)) <= y,
        B*w - np.ones((k)) * b + np.ones((k)) <= z,
        y >= 0,
        z >= 0,
        -v <= w,
        w <= v]

    while True:
        obj = cvx.Minimize((1-lambduh) * (sum(y) / m + sum(z) / k) + lambduh * alpha * vexp(-alpha *
        prob = cvx.Problem(obj, constraints)
        prob.solve(solver=cvx.CVXOPT)
        print("status:", prob.status)
        # print('opt val:', prob.value)

        endVal = (1-lambduh)*(sum(y.value - yi) / m +
        endVal = np.squeeze(endVal)
        # print(endVal[0][0])

        wi = w.value
        bi = b.value
        yi = y.value
        zi = z.value
        vi = v.value

        # alpha = 1.2 * alpha

        if abs(endVal) < .0000000001:
            break

    print(wi)
    print(bi)
    return [wi, bi]

def predict(w, b, x):
    pred = np.dot(w.T, x) - b
    if pred >= 0:
        return 1
    else:
        return -1

def plotStuff(A, B, w, b):
    data = np.array([x for x in A] + [x for x in B])
    h = .005
    xMin, xMax = data[:, 0].min(), data[:, 0].max()
    yMin, yMax = data[:, 1].min(), data[:, 1].max()
    xx, yy = np.meshgrid(np.arange(xMin, xMax, h), np.

    Z = np.array([predict(w, b, [x,y,0]) for (x,y) in
    # Z = np.array([predictAvg(w, [x,y, 1.0]) for (x,
    # Z = np.array([predictDual(alphas, combined, dist
    Z = Z.reshape(xx.shape)

    plt.contour(xx, yy, Z, cmap=plt.cm.Paired)

    # xInt = -1.0 * w[2] / w[0]
    # yInt = -1.0 * w[2] / w[1]
    # plt.plot([xInt, 0], [0, yInt])

    # plt.plot([x[0][0] for x in combined if x[1] ==
    # plt.plot([x[0][0] for x in combined if x[1] ==
    plt.plot([x[0] for x in A], [x[1] for x in A], 'ro
    plt.plot([x[0] for x in B], [x[1] for x in B], 'bs
    plt.show()

def toyData():
    numPts = 100
    tempA = np.random.multivariate_normal([0,1], [[.5,
    tempB = np.random.multivariate_normal([1,0], [[.5,
    A = np.zeros((numPts, 3))
    B = np.zeros((numPts, 3))
    for i in range(len(A)):
```

```
        A[i] = np.append(tempA[i], np.random.random() * 500)
    for i in range(len(B)):
        B[i] = np.append(tempB[i], np.random.random() * 500)

    X = np.array([x for x in A] + [x for x in B])
    Y = np.array([1] * numPts + [-1] * numPts)
    # [w,b] = sla(A, B, .5, 2)

    # # compute accuracy
    # aPreds = [predict(w,b,x) for x in A]
    # correct = aPreds.count(1)
    # bPreds = [predict(w,b,x) for x in B]
    # correct += bPreds.count(-1)
    # accuracy = float(correct) / float(2*numPts)
    # print('accuracy: ', accuracy)

    # print([predict(w, b, x) for x in A])
    # print([predict(w, b, x) for x in B])
    # plotStuff(A, B, w, b)

    clf = svm.SVC(kernel='linear', C=.001)
    clf.fit(X, Y)
    print('svm')
    print(clf.coef_)
    print(clf.score(X, Y))
    # plotStuff(A, B, np.array(clf.coef_).reshape((2,1)), clf.intercept_)

def loadAdultData():
    A = []
    B = []
    d = [{} for _ in range(14)]
    for l in open('adult.data'):
        parts = l.split(',')
        if len(parts) < 14:
            continue
        item = []
        for p,i in zip(parts, range(15)):
            if i == 14:
                if p.strip() == '<=50K':
                    A.append(np.array(item))
                else:
                    B.append(np.array(item))
            elif i == 0 or i == 2 or i == 4 or i == 10 or i == 11 or i == 12:
                item.append(int(p.strip()))
            else:
                if not p in d[i]:
                    if not d[i]:
                        d[i][p] = 0
                    else:
                        d[i][p] = max(d[i].values()) + 1
                item.append(d[i][p])

    random.shuffle(A)
    random.shuffle(B)
    A = np.array(A)
    B = np.array(B)

    return(A[:300,:], B[:300,:], A[1000:2000,:], B[1000:2000,:])


def censusData():
    A,B,testA,testB = loadAdultData()
    # [w,b] = sla(A, B, .1, 5)
    # # compute accuracy
    # aPreds = [predict(w,b,x) for x in A]
    # correct = aPreds.count(1)
    # bPreds = [predict(w,b,x) for x in B]
    # correct += bPreds.count(-1)
    # accuracy = float(correct) / float(A.shape[0] + B
    # print('train accuracy: ', accuracy)

    # aPreds = [predict(w,b,x) for x in testA]
    # correct = aPreds.count(1)
    # bPreds = [predict(w,b,x) for x in testB]
    # correct += bPreds.count(-1)
    # accuracy = float(correct) / float(testA.shape[0]
    # print('test accuracy: ', accuracy)

    # svm
    X = np.array([x for x in A] + [x for x in B])
    Y = np.array([1] * A.shape[0] + [-1] * B.shape[0])
    clf = svm.SVC(kernel='linear', C=1)
    clf.fit(X, Y)
    print('train accuracy')
    print(clf.score(X, Y))

    X = np.array([x for x in testA] + [x for x in test
    Y = np.array([1] * testA.shape[0] + [-1] * testB.s
    print('test accuracy')
    print(clf.score(X, Y))

def main():
    random.seed(12345)
    np.random.seed(12345)
    censusData()
    # toyData()


if __name__ == '__main__':
    main()
```

## B. SOURCE CODE FOR DIRECT

```
import numpy as np
import matplotlib.pyplot as plt
import cvxpy as cvx
import math
import random
import copy

def direct(A, B, lambduh):
    vexp = np.vectorize(math.exp)
    n = A.shape[1]
    m = A.shape[0]
    k = B.shape[0]

    w = cvx.Variable(n)
    b = cvx.Variable()
    y = cvx.Variable(m)
    z = cvx.Variable(k)

    wi = np.random.rand(n,1)
    bi = random.random()

    prevWI = copy.deepcopy(wi)
    prevBI = copy.deepcopy(bi)
```

```
    constraints = [-A*w - np.ones((m,1)) * b + np.ones((m,1)) <= y,
        B*w + np.ones((k)) * b + np.ones((k)) <= z,
        y >= 0,
        z >= 0]

    count = 0
    while True:
        # print(1/np.square(np.squeeze(wi.reshape((1,n)))))
        bigLambduh = np.diag(1/np.square(np.squeeze(wi.reshape((1,n)))))
        obj = cvx.Minimize(lambduh * (sum(y) + sum(z)) + cvx.quad_form(w, bigLambduh))
        prob = cvx.Problem(obj, constraints)
        prob.solve(solver=cvx.CVXOPT)
        print("status:", prob.status)
        # print('opt val:', prob.value)

        prevWI = wi
        prevBI = bi
        wi = np.array(w.value)
        bi = b.value

        diff = np.squeeze(wi - prevWI).reshape(n,1)

        for i in range(n):
            if wi[i] < .00001:
                wi[i] = .00001

        diff = sum(sum(abs(diff))) + abs(bi - prevBI)
        # print(diff)
        if diff < .0001:
            break

        if count > 10:
            break
        count += 1

        # print(np.dot(np.dot(wi.T, bigLambduh), wi))


    print(wi)
    print(bi)
    return [wi, bi]

def directDual(A, B, lambduh, K):
    n = A.shape[1]
    m = A.shape[0]
    k = B.shape[0]

    alphas = cvx.Variable(m + k)
    b = cvx.Variable()
    y = cvx.Variable(m)
    z = cvx.Variable(k)

    constraints = [-K[:m,:] * alphas - np.ones(m) * b + np.ones(m) <= y,
        K[m:,:] * alphas + np.ones(k) * b + np.ones(k) <= z,
        y >= 0,
        z >= 0]

    # constraints = [None] * (m + k)

    # for i in range(m):
    #     constraints[i] = cvx.mul_elemwise(K[i,:], alphas) - b + 1 <= y[i]
    # for i in range(k):
    #     constraints[m+i] = cvx.mul_elemwise(K[m+i,:]
    # constraints.append(y >= 0)
    # constraints.append(z >= 0)

    alphasi = np.random.random(m + k)
    bi = np.random.random()
    while True:
        bigLambduh = np.diag(1/np.square(alphasi))
        obj = cvx.Minimize(cvx.quad_form(alphas, bigLa
        prob = cvx.Problem(obj, constraints)
        prob.solve(solver=cvx.CVXOPT)

        diff = np.squeeze(alphasi - alphas.value.T).re

        alphasi = np.squeeze(np.array(alphas.value.T))
        bi = b.value

        for i in range(m+k):
            if alphasi[i] <= .00001:
                alphasi[i] = .00001

        print(sum(sum(abs(diff))))
        if sum(sum(abs(diff))) < .1:
            break

    return [alphasi, bi]

def predict(w, b, x):
    pred = np.dot(w.T, x) + b
    if pred >= 0:
        return 1
    else:
        return -1

def plotStuff(A, B, w, b):
    data = np.array([x for x in A] + [x for x in B])
    h = .05
    xMin, xMax = data[:, 0].min(), data[:, 0].max()
    yMin, yMax = data[:, 1].min(), data[:, 1].max()
    xx, yy = np.meshgrid(np.arange(xMin, xMax, h), np.
    # Z = np.array([predict(w, b, [x,y,0]) for (x,y) i
    Z = np.array([predictDual(w, b, data, [x,y]) for (
    Z = Z.reshape(xx.shape)

    plt.contour(xx, yy, Z, cmap=plt.cm.Paired)

    # plt.plot([x[0][0] for x in combined if x[1] == 1
    # plt.plot([x[0][0] for x in combined if x[1] == -
    plt.plot([x[0] for x in A], [x[1] for x in A], 'ro
    plt.plot([x[0] for x in B], [x[1] for x in B], 'bs
    plt.show()

def predictDual(alphas, b, data, x):
    pred = 0.0
    for i,d in zip(range(len(data)), data):
        if alphas[i] >= .0001:
            # pred = pred + alphas[i] * np.dot(d.T, x)
            pred = pred + alphas[i] * gaussianKernel(
    pred = pred + b
    if pred >= 0:
```

```
        return 1
    else:
        return -1

def gaussianKernel(x, y):
    sigma = 1.0
    return math.exp(-1.0 * np.dot(x-y, x-y) / (2.0 * sigma * sigma))

def makeDotK(A, B):
    m = A.shape[0]
    k = B.shape[0]
    data = np.array([x for x in A] + [x for x in B])
    K = np.zeros((m+k, m+k))
    for i in range(m+k):
        for j in range(m+k):
            # K[i,j] = np.dot(data[i,:].T, data[j,:])
            K[i,j] = gaussianKernel(data[i,:].T, data[j,:])

    return K

def loadAdultData():
    A = []
    B = []
    d = [{} for _ in range(14)]
    for l in open('adult.data'):
        parts = l.split(',')
        if len(parts) < 14:
            continue
        item = []
        for p,i in zip(parts, range(15)):
            if i == 14:
                if p.strip() == '<=50K':
                    A.append(np.array(item))
                else:
                    B.append(np.array(item))
            elif i == 0 or i == 2 or i == 4 or i == 10 or i == 11 or i == 12:
                item.append(int(p.strip()))
            else:
                if not p in d[i]:
                    if not d[i]:
                        d[i][p] = 0
                    else:
                        d[i][p] = max(d[i].values()) + 1
                item.append(d[i][p])

    random.shuffle(A)
    random.shuffle(B)
    A = np.array(A)
    B = np.array(B)

    return(A[:300,:], B[:300,:], A[1000:2000,:], B[1000:2000,:])

def toyData():
    numPts = 100
    tempA = np.random.multivariate_normal([0,1], [[.5, 0], [0, .5]], numPts)
    tempB = np.random.multivariate_normal([1,0], [[.5, 0], [0, .5]], numPts)
    # A = np.zeros((numPts, 3))
    # B = np.zeros((numPts, 3))
    # for i in range(len(A)):
    #     A[i] = np.append(tempA[i], np.random.random() * 500)
    # for i in range(len(B)):
    #     B[i] = np.append(tempB[i], np.random.random() * 500)
    A = tempA
    B = tempB

    # [w,b] = direct(A, B, 10)
    # print([predict(w, b, x) for x in A])
    # print([predict(w, b, x) for x in B])
    # # plotStuff(A, B, w, b)
    # # compute accuracy
    # aPreds = [predict(w,b,x) for x in A]
    # correct = aPreds.count(1)
    # bPreds = [predict(w,b,x) for x in B]
    # correct += bPreds.count(-1)
    # accuracy = float(correct) / float(2*numPts)
    # print('accuracy: ', accuracy)

    K = makeDotK(A,B)
    data = np.array([x for x in A] + [x for x in B])
    [alphas, b] = directDual(A, B, 6, K)
    print(alphas)
    print(b)
    # print([predictDual(alphas, b, data, x) for x in
    # print([predictDual(alphas, b, data, x) for x in
    plotStuff(A, B, alphas, b)

    # compute accuracy
    aPreds = [predictDual(alphas,b, data, x) for x in
    correct = aPreds.count(1)
    bPreds = [predictDual(alphas,b, data, x) for x in
    correct += bPreds.count(-1)
    accuracy = float(correct) / float(A.shape[0] + B.s
    print('train accuracy: ', accuracy)

def censusData():
    A,B,testA,testB = loadAdultData()
    [w,b] = direct(A, B, 5)
    # compute accuracy
    aPreds = [predict(w,b,x) for x in A]
    correct = aPreds.count(1)
    bPreds = [predict(w,b,x) for x in B]
    correct += bPreds.count(-1)
    accuracy = float(correct) / float(A.shape[0] + B.s
    print('train accuracy: ', accuracy)

    aPreds = [predict(w,b,x) for x in testA]
    correct = aPreds.count(1)
    bPreds = [predict(w,b,x) for x in testB]
    correct += bPreds.count(-1)
    accuracy = float(correct) / float(testA.shape[0] -
    print('test accuracy: ', accuracy)
def main():
    random.seed(12345)
    np.random.seed(12345)
    toyData()
    # censusData()

if __name__ == '__main__':
    main()
```