Sachin Ravi
COS 521

Project Report
January 13, 2014

Semi-supervised Learning in Support Vector Machines

1 Introduction

In traditional supervised classification, classifiers are trained using feature/label pairs and the classifier performance is measured on unseen test data. In the current Internet age, as the amount of data produced grows exponentially, we would like to use as much of this data as possible to train classifiers in order to get better performance. This is especially true in models such as neural networks that have millions of parameters and thus can overfit to training sets that contain even millions of labeled items [4]. However, the problem is that labeled data is hard to acquire, having high cost in terms of both time and money. For example, ImageNet, currently the standard image classification dataset, requires manual labor outsourced through the use of Amazon Mechanical Turk to produce the true labels for the training set [2].

Thus, it would be very beneficial if we could use the huge amount of readily available unlabeled data to train our classifiers. Semi-supervised learning is the setting in which a classifier is trained using a small amount of labeled data and a large amount of unlabeled data. The basic idea is to initially train the classifier using the labeled data and then improve the generalization ability (in terms of accuracy) of the classifier by somehow using the large amount of unlabeled data.

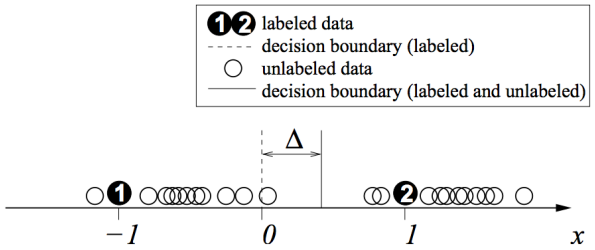

Figure 1: Unlabeled data can be a form of regularization.

The first and most important question to ask is how can unlabeled data even be helpful? Consider the situation in Figure 1 [7]. The hypothesis is that the initial classifier overfits the training data. By using the large amount of unlabeled data, we get a classifier that will have better generalization performance on unseen test data. Thus, we can view using the unlabeled data as a form of regularization designed to avoid overfitting.

2 Background

2.1 Support Vector Machines

In this project we explored using Support Vector Machines (SVMs) in the context of Semi-supervised learning. In Support Vector Machines, the intuition is to try to create a separating hyperplane between the positively and negatively labeled items. Since many choices could exist for this hyperplane, in order to generalize well on test data, we find the hyperplane with the largest margin, which is defined to be the distance from the hyperplane to an item in the training set. We want to maximize the margin that so that the hyperplane is far away as possible from the training data so that it is less likely that a new test item will be misclassified by the hyperplane.

Suppose $L = \{(\mathbf{x}_1, y_1), \dots (\mathbf{x}_m, y_m)\}$ is the training set and v be a hyperplane going through the origin. Let δ be the margin and let $w = \frac{v}{\delta}$. The margin maximizing hyperplane can be formulated as an optimization problem in the following manner:

$$\begin{aligned} &\min \frac{1}{2}||w||^2 \\ &\text{s.t. } \forall i \ : \ y_i(\mathbf{w} \cdot \mathbf{x}_i) \geq 1. \end{aligned}$$

In many cases, it is not possible to perfectly separate the positive and negative examples by any hyperplane. Thus, we want to be able to move over some of the wrongly classified examples but pay a price for this since we do not want it to happen very often. The optimization problem is then changed to be:

$$\begin{aligned} &\min \frac{1}{2}||w||^2 + C\sum_{i=1}^{m} \xi_i \\ &\text{s.t. } \forall i \ : \ y_i(\mathbf{w} \cdot \mathbf{x}_i) \geq 1 - \xi_i \\ &\quad \xi_i \geq 0, \end{aligned}$$

where ξ_i is the slack for each training item and C is some positive constant.
With $z_+ = \max(0, z)$, this problem can be simplified to:

$$\min \frac{1}{2}||w||^2 + C\sum_{i=1}^{m} (1 - y_i(\mathbf{w} \cdot \mathbf{x}_i))_+.$$

The $y_i(\mathbf{w} \cdot \mathbf{x}_i)$ is the margin and the $(1 - y_i(\mathbf{w} \cdot \mathbf{x}_i))_+$ is known as the hinge loss. This loss is convex (see Figure 2) and so the above problem is a convex optimization problem that can be easily solved. The loss is 0 whenever a point is on the correct side of the hyperplane i.e. whenever $y_i(\mathbf{w} \cdot \mathbf{x}_i) \geq 1$.

2.2 Quasi-Newton Optimization

We briefly discuss Quasi-Newton optimization as it is used in the related work. In class, we discussed gradient descent as a method for doing non-convex optimization in which we

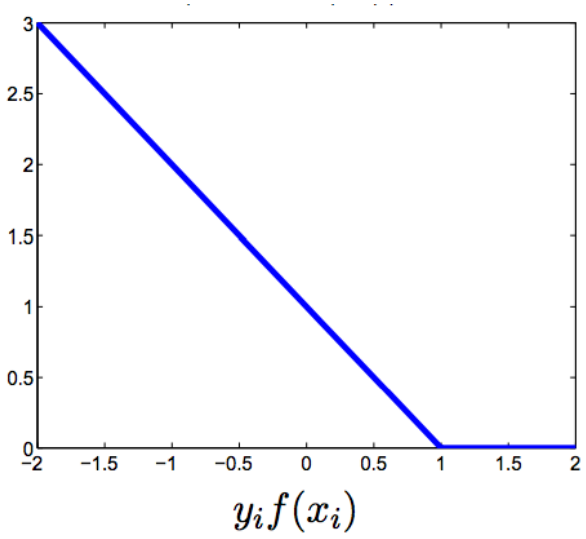

Figure 2: For $f(\mathbf{x}_i) = (\mathbf{w} \cdot \mathbf{x}_i)$, the hinge loss $(1 - y_i(\mathbf{w} \cdot \mathbf{x}_i))_+$ is convex.

try to minimize a function value by moving in a direction related to the gradient. Another method for non-convex optimization is Newton's method. In the single-dimensional case, a function attains its minimum when the derivative $f'(x) = 0$. In Newton's method, we start with a guess of the minimum and we approximate the derivative of the function at that point using a linear function and then solve for the root of that function explicitly. We then iterate repeatedly for some number of steps. To approximate $f'(x)$, we approximate $f(x)$ using a second-order Taylor series which makes use of $f''(x)$ and so this method has higher requirements on the smoothness of $f(x)$ but by using more information about $f(x)$ may converge faster.

We now try to make this intuition more exact. Let $f : \mathbb{R}^n \to \mathbb{R}$ be a twice-differentiable function. We start with an initial point x_0 and produce a sequence of points that converge to the optimum x^*. Let the gradient of f at x_k be ∇f_k and the Hessian be H_k.

Let $p = x - x_k$. The second order Taylor expansion around x_k is

$$m_k(p) = f_k + p^T \nabla f_k + \frac{1}{2} p^T H_k p.$$

The gradient with respect to this function is then

$$\nabla m_k(p) = \nabla f_k + H_k p.$$

Setting this equal to 0, we get that the minimum occurs at $x - x_k = p_k = -H_k^{-1} \nabla f_k$. We thus use $p_k = -H_k^{-1} \nabla f_k$ as the search direction and for some $\alpha \in (0, \infty)$, the next point is

$$x_{k+1} = x_k + \alpha p_k.$$

Computing the inverse of the Hessian can be costly, and so Quasi-Newton methods involve approximating the inverse somehow so as to speed up the process. There are many different methods that fall under the Quasi-Newton class but we discuss one specific method called BFGS (Broyden-Fletcher-Goldfarb-Shanno) algorithm.

2.2.1 BFGS

In this method, we directly try to approximate the inverse of the Hessian H_k^{-1}. To find this approximation, we define several desirable properties:

1. H_k^{-1} is symmetric
2. In the definition of the gradient using a linear approximation, the function's gradient approximation must equal the function's gradient at two points x_k and x_{k-1}. For x_k, we have trivially that

$$\nabla m_k(x_k) = \nabla f_k + H_k^{-1}(x_k - x_k) = \nabla f_k.$$

   For x_{k-1}, we need

$$\begin{aligned} &\nabla m_k(x_{k-1}) = \nabla f_k + H_k^{-1}(x_{k-1} - x_k) = \nabla f_{k-1} \\ \Leftrightarrow &(x_k - x_{k-1}) = H_k^{-1}(\nabla f_k - \nabla f_{k-1}) \end{aligned}$$

   Letting $s_{k-1} = x_k - x_{k-1}$ and $y_{k-1} = \nabla f_k - \nabla f_{k-1}$, we get that we need

$$s_{k-1} = H_k^{-1} y_{k-1}.$$

3. Subject to the above properties, we have H_k^{-1} to be as close as possible to H^{k-1}. We define closeness in terms of the Frobenius norm.

The optimization problem with these properties in mind is the following:

$$\begin{aligned} &\min ||H - H^{k-1}|| \\ &\text{s.t. } H = H^T \\ &\quad\; Hy_{k-1} = s_{k-1} \end{aligned}$$

This problem has the following unique solution:

$$H_k = \left(1 - \rho_{k-1} s_{k-1} y_{k-1}^T\right) H_{k-1} \left(1 - \rho_{k-1} y_{k-1} s_{k-1}^T\right) + s_{k-1} \rho_{k-1} s_{k-1}^T,$$

where $\rho_{k-1} = \left(y_k^T s_k\right)^{-1}$.

3 Related Work

Formally, in the semi-supervised setting, we assume we have some extra unlabeled training data $U = \{x_{m+1}, \ldots, x_{m+u}\}$.

3.1 Semi-Supervised SVMs

In [6], an optimization problem is formulated for finding the optimal labeling for test data using the both labeled and unlabeled data. For unlabeled data, it is assumed that the true label is the one predicted by the model based on what side of the hyperplane the unlabeled point ends up being. Thus, the optimization problem becomes:

$$\min \frac{1}{2}||w||^2 + C_1 \sum_{i=1}^{m}(1 - y_i(\mathbf{w} \cdot \mathbf{x}_i))_+ + C_2 \sum_{i=1}^{l}(1 - sign(\mathbf{w} \cdot \mathbf{x}_i)(\mathbf{w} \cdot \mathbf{x}_i))_+$$
$$= \min \frac{1}{2}||w||^2 + C_1 \sum_{i=1}^{m}(1 - y_i(\mathbf{w} \cdot \mathbf{x}_i))_+ + C_2 \sum_{i=1}^{l}(1 - |(\mathbf{w} \cdot \mathbf{x}_i)|)_+,$$

where C_1 and C_2 positive constants.

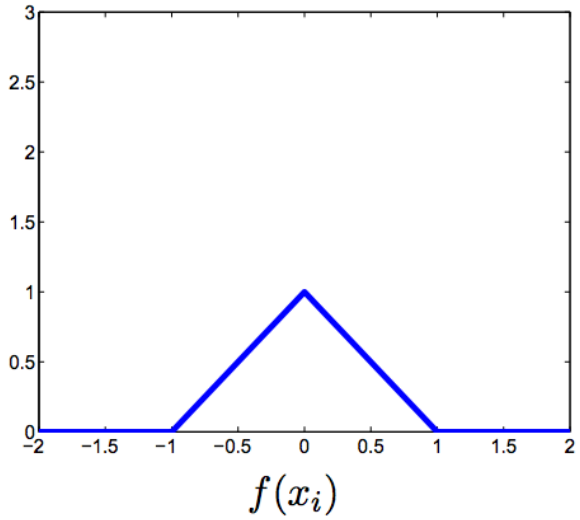

Figure 3: For $f(\mathbf{x}_i) = (\mathbf{w} \cdot \mathbf{x}_i)$, the loss involved in the semi-supervised formulation $(1 - |(\mathbf{w} \cdot \mathbf{x}_i)|)_+$ is non-convex.

The difficulty here is that $(1 - |(\mathbf{w} \cdot \mathbf{x}_i)|)_+$ has a non-convex shape (see Figure 3), making the problem hard to solve. This loss prefers $\mathbf{w} \cdot \mathbf{x}_i \geq 1$ or $\mathbf{w} \cdot \mathbf{x}_i \leq -1$, or the unlabeled instance to be away from the decision boundary.

Since the loss is non-convex, we would like to use some approximate technique to solve the optimization problem. In order to use any gradient based approach, however, we need a differentiable loss function and both the hinge loss and the hat loss are clearly not differentiable. In [3], the authors replace each of these non-differentiable functions with their differentiable counterparts. The hinge loss is replaced with the modified logistic loss and the hat loss is replaced with an exponential loss (see Figure 4).

With the use of these differentiable counterparts, the optimization problem then becomes

$$\min \frac{1}{2}||w||^2 + C_1 \sum_{i=1}^{m} \frac{1}{\gamma_1} \log(1 + exp(\gamma_1(1 - y_i(\mathbf{w} \cdot \mathbf{x}_i)))) + C_2 \sum_{i=1}^{l} exp(-\gamma_2(\mathbf{w} \cdot \mathbf{x}_i)^2).$$

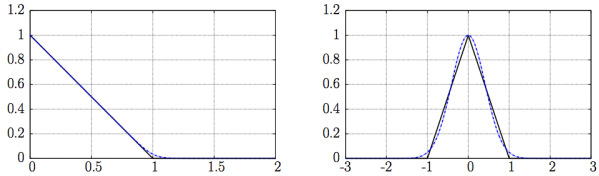

Figure 4: The hinge loss $L(y,t) = (1-yt)_+$ is replaced with its differentiable counterpart, the logistic loss $L(y,t) = \frac{1}{\gamma}\log(1+exp(\gamma(1-yt)))$. The hat loss $L(t) = (1-|t|)_+$ is replaced with its differentiable counterpart, the exponential loss $L(t) = exp(-\gamma t^2)$

The authors then solve this optimization problem approximately by using the BFGS algorithm.

4 Self-Training

In this project, we explore using self-training to learn from the unlabeled data. The idea here is to "prime" the model with labeled data and then use the model's own predictions as labels for the unlabeled data to re-train a new model with the original labeled data and the newly labeled data and then iteratively repeat this process. The problem with this method is that it can suffer from "semantic drift" [1], where considering its own predictions as true labels can cause the model to drift away from the correct model. The model would then continue to mislabel data and use it again and continue to drift farther and farther away from where it should be. Thus, we need to prevent using mislabeled data because it can cause the model error's to continually propagate.

To prevent this problem, we will only use the model's predictions to label the data only when we are highly confident about the predictions. If we only assume labels for items we are highly confident about, it is less likely that we will attempt to learn from incorrect data. The notion of confidence we will use for our SVM model is the distance from the hyperplane used by the SVM. The larger the distance from the hyperplane, the more confident we can be because this means the item is more deeper in the space of the class the SVM thinks the item belongs to and thus likely it should be on the other side of the SVM.

One question we could have is how using highly confident predictions as labels for unlabeled items could help improve our model? For example, in the SVM case, if we only added labels for items that are far away from the hyperplane, then would not the optimal hyperplane remain the same? We just have to ensure that we are not too conservative in picking which points to infer the label for so that we are adding some new information to improve the classifier.

We now describe how exactly our learning algorithm is implemented. We first train an SVM on the labeled training data L. Let μ_+, σ_+ be the mean and standard deviation of the distances to the hyperplane of the positively labeled items and μ_-, σ_- be mean and standard deviation of the distances to the hyperplane of the negatively labeled items. We then iteratively process the unlabeled training data over β rounds. In each round, we label

each item x as y in the unlabeled training set based on our existing SVM and get the distance d for the item from the hyperplane. From Chebyshev's inequality, we know that for a class l

$$\Pr(|X - \mu_y| \geq k \cdot \sigma_y) \leq \frac{1}{k^2}$$
$$\Leftrightarrow \Pr(|X - \mu_y| < k \cdot \sigma_y) \geq 1 - \frac{1}{k^2}.$$

Thus, we have some threshold δ and only infer the label of an item given some distance d if d lies within some k (calculated according to δ) standard deviations from the mean. More concretely, we have that

$$\Pr\left(|X - \mu_y| < \sqrt{\frac{1}{1-\delta}}\sigma_y\right) \geq \delta,$$

and so we skip the item if $|d - \mu_y| \geq \sqrt{\frac{1}{1-\delta}}\sigma_y$ because that is a low probability event; otherwise we add the item $(\mathbf{x}, y)$ to our labeled training set and remove it from the unlabeled set. After processing all these items, we train a new SVM on the labeled training set and repeat for β rounds. Pseudocode for the algorithm is given in Algorithm 1.

Algorithm 1 Pseudocode for Self-Learning Algorithm

```
L is the initial labeled training set, U is the unlabeled set
M ← SVM trained on L
for t = 1 to β do
    μ+, σ+ ← mean, standard deviation of distances to hyperplane of M of positively labeled items
    μ−, σ− ← mean, standard deviation of distances to hyperplane of M negatively labeled items
    for x ∈ U do
        d ← distance of x to hyperplane of M
        y ← label of x according to hyperplane of M
        if |d − μy| < sqrt(1/(1−δ)) σy then
            L ← L ∪ {(x, y)}
            U ← U − {x}
        end if
    end for
    M ← SVM trained on L
end for
Return M
```

5 Experimental

In order to evaluate the method discussed above, we test its effectiveness on the MNIST dataset [5]. To simplify the analysis, since MNIST is a multi-class classification task, we reduce it to a binary classification by only considering data involving only the digits 4 and 5. We use the *sci-kit* library in Python and the code used is attached.

Since MNIST is normally used for supervised learning, we create the labeled and unlabeled sets manually. Considering only the digits 4 and 5, the labeled training set size is 9365 and the test size is 1874. We then split this original labeled training set into a small labeled set and large unlabeled set, where the labels are ignored. The initial labeled training set has 100 digits and so 8365 digits are in the unlabeled set. We pick optimal values of δ and β based on our experiments. If we use the entire training set to learn an SVM model, we can

get an accuracy of 0.983 on the test set. This accuracy serves as an upper bound for our semi-supervised learning algorithm because that accuracy is achieved using all the correct labels for the items that we will use in our unlabeled set.

In the first experiment, we run our self-learning algorithm and show that it shows better performance than the base classifier on the test set and thus is able to learn something useful from the unlabeled data. We see that δ is picked such that the classifier is improved from the base model, meaning that the items we decide to label from the unlabeled set at each step are not too conservatively picked. Our final model trained by the self-learning algorithm achieves an accuracy of 0.981, which is close to 2.4% better than the base model trained purely by the labeled set. Additionally, this accuracy is close to the upper bound from above where the whole set is used as a labeled set. We can see that we end up using more than 75% of the unlabeled set by assigning labels based on our model.

Round	Number Labeled	Accuracy
Base Model	NA	0.9573
Round 1	6023	0.9727
Round 2	108	0.9781
Round 3	66	0.9791
Round 4	61	0.9807
Round 5	49	0.9813
Final Model	NA	0.9813

Figure 5: Performance of self-learning algorithm on test-set.

In the second experiment, we show how the choice of δ can affect the self-training algorithm. We see that as we decrease the values of δ, the model's performance increases and then decreases. This occurs because with a low threshold, we only label the items that we are confident about and so the labels are true but we do not label that many items and so the model cannot improve as much. With a very high threshold, we label more items but some of those labels are incorrect (because we were not that confident about them) and so we end up learning the wrong pattern and hurt our model performance.

We compared the performance of our method to [3] but found that using their code produced an accuracy less than simply training only on the small labeled set. We are not sure if this bad performance is due to this specific dataset or because some necessary condition for the theory was broken.

6 Conclusion

In this project, we consider semi-supervised learning in the context of SVMs. We first introduce the basic SVM formulation and discuss Quasi-Newton Optimization, specifically the BFGS algorithm, as it is relevant to understanding work in the area. Then, in the related work, we discuss semi-supervised SVMs and current work that involves solving the semi-supervised SVM optimization problem by using differentiable equivalents in the objective function and then solving the problem approximately using the BFGS algorithm. We then formalize our own technique of self-learning in which we use the model's highly confident

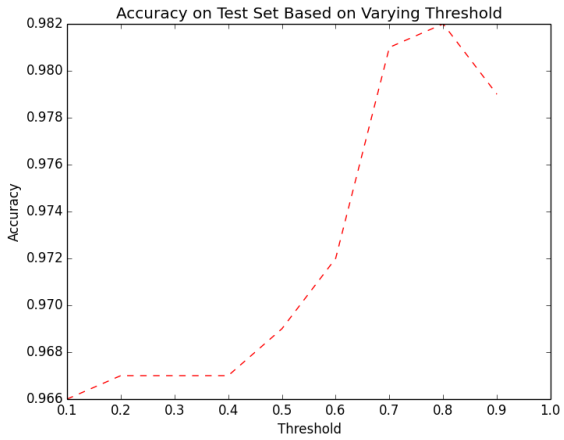

Figure 6: Experimental results displaying how varying threshold affects accuracy on test set.

predictions as assumed true labels in the unlabeled set, train a new model on this new labeled data and repeat until convergence. We carry out experiments on a modified version of the MNIST dataset to test the effectiveness of our self-learning method. Our results show that using our self-learning algorithm we can get better performance using an unlabeled set in addition to the initial labeled one.

References

[1] Xinlei Chen, Abhinav Shrivastava, and Abhinav Gupta. Neil: Extracting visual knowledge from web data. In *Computer Vision (ICCV), 2013 IEEE International Conference on*, pages 1409–1416. IEEE, 2013.

[2] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. IEEE, 2009.

[3] Fabian Gieseke, Antti Airola, Tapio Pahikkala, and Oliver Kramer. Sparse quasi-newton optimization for semi-supervised support vector machines. In Pedro Latorre Carmona, J. Salvador Snchez, and Ana L. N. Fred, editors, *Proceedings of the 1st International Conference on Pattern Recognition Applications and Methods (ICPRAM)*, page 4554. SciTePress, 2012.

[4] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[5] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[6] Vladimir Naumovich Vapnik and Vlamimir Vapnik. *Statistical learning theory*, volume 2. Wiley New York, 1998.

[7] Xiaojin Zhu. Semi-supervised learning literature survey. Technical Report 1530, Computer Sciences, University of Wisconsin-Madison, 2005.

```
"""
Generate dataset of 4's and 5's
"""

from sklearn.datasets import load_digits
import pickle, gzip, numpy

__IMAGE_SIZE__ = 784

__DATA__ = 'data'
__LABELS__ = 'labels'

__CLASS_POS__ = 4
__CLASS_NEG__ = 5

if __name__ == "__main__":

    # Load the dataset
    train_set, valid_set, test_set = pickle.load( open( "mnist.pkl", "rb" ) )

    X = train_set[0]
    Y = train_set[1]
    relvX = numpy.empty((0,__IMAGE_SIZE__), int)
    relvY = numpy.empty((0,1), int)
    for i in range(0, X.shape[0]):
        if Y[i] == __CLASS_POS__:
            relvX = numpy.append(relvX, numpy.matrix(X[i]),
                axis=0)
            relvY = numpy.append(relvY, numpy.matrix([[0]]),
                axis=0)
        elif Y[i] == __CLASS_NEG__:
            relvX = numpy.append(relvX, numpy.matrix(X[i]), axis=0)
            relvY = numpy.append(relvY, numpy.matrix([[1]]), axis=0)

    testX = test_set[0]
    testY = test_set[1]
    relvTestX = numpy.empty((0,__IMAGE_SIZE__), int)
    relvTestY = numpy.empty((0,1), int)
    for i in range(0, testX.shape[0]):
        if testY[i] == __CLASS_POS__:
            relvTestX = numpy.append(relvTestX, numpy.matrix(testX[i]),
                axis=0)
            relvTestY = numpy.append(relvTestY, numpy.matrix([[0]]),
                axis=0)
        elif testY[i] == __CLASS_NEG__:
```

```
            relvTestX = numpy.append(relvTestX, numpy.matrix(testX[i]),
                axis=0)
            relvTestY = numpy.append(relvTestY, numpy.matrix([[1]]),
                axis=0)

train = {__DATA__ : relvX, __LABELS__: relvY}
# Save training set as pickle file
pickle.dump( train, open( "train.p", "wb" ) )

test = {__DATA__: relvTestX, __LABELS__ : relvTestY}
# Save test set as pickle file
pickle.dump( test, open( "test.p", "wb" ) )
```

```
"""
Self-learning algorithm
"""
from sklearn import svm
import numpy
import pickle
import math

__IMAGE_SIZE__ = 3072

__DATA__ = 'data'
__LABELS__ = 'labels'

__TRAINING_SIZE__ = 100
__TEST_SIZE__ = 1874
__UNLABELED_SIZE__ = 7000

__THRESHOLD__ = 0.80
__ROUNDS__ = 5

def load_data():
    train = pickle.load( open( "train.p", "rb" ) )
    test = pickle.load( open( "test.p", "rb" ) )

    X = train[__DATA__][0:__TRAINING_SIZE__:]
    Y = train[__LABELS__][0:__TRAINING_SIZE__:]
    unlabeledX = train[__DATA__][__TRAINING_SIZE__ + 1:__TRAINING_SIZE__
        + __UNLABELED_SIZE__]

    testX = test[__DATA__][0:__TEST_SIZE__]
    testY = test[__LABELS__][0:__TEST_SIZE__:]

    return [X,Y,testX,testY,unlabeledX]

def train(base_svm, rounds, X, Y, unlabeledX, testX, testY):

    for i in range(0, rounds):
        pos_distances = numpy.ravel(base_svm.decision_function(
                X[numpy.ravel(Y[:,0] == 0)]))
        pos_mean = (sum(pos_distances)/len(pos_distances))
        pos_std_dev = numpy.std(pos_distances)

        neg_distances = numpy.ravel(base_svm.decision_function(
                X[numpy.ravel(Y[:,0] == 1)]))
        neg_mean = (sum(neg_distances)/len(neg_distances))
```

```
        neg_std_dev = numpy.std(neg_distances)

        skipped = 0
        to_remove = []

        k = math.sqrt(1.0/(1.0 - __THRESHOLD__))

        j = 0
        for x in unlabeledX:
            pred = base_svm.predict(x)[0]
            dist = base_svm.decision_function(x)[0][0]

            value = 0.0
            if (pred == 0):
                value = (abs(dist - pos_mean)/pos_std_dev)
            elif (pred == 1):
                value = (abs(dist - neg_mean)/neg_std_dev)

            if (value < k):
                X = numpy.append(X, numpy.matrix(x), axis=0)
                Y = numpy.append(Y, numpy.matrix([[pred]]), axis=0)
                to_remove.append(j)
            else:
                skipped = skipped + 1

            j = j + 1

        print "labeled", len(to_remove)

        new_svm = svm.SVC()
        new_svm.fit(X, Y)

        acc = new_svm.score(testX, testY)
        print "accuracy", acc

        base_svm = new_svm
        unlabeledX = numpy.delete(unlabeledX, to_remove, 0)

    print "final traing set size", X.shape[0]
    return base_svm


if __name__ == "__main__":
    array = load_data()
    X,Y,testX,testY,unlabeledX = load_data()
```

```
lin_svm = svm.SVC()
lin_svm.fit(X, Y)
base_acc = lin_svm.score(testX, testY)

final_svm = train(lin_svm, __ROUNDS__, X, Y, unlabeledX, testX, testY)
acc = final_svm.score(testX, testY)

print "threshold", __THRESHOLD__
print "base accuracy", base_acc
print "final accuracy", acc
```