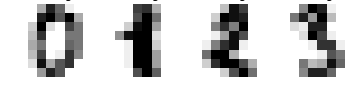
Run times for SVM image recognition using dimensionality reduction

by Ben Hendel and Russell Luttrell

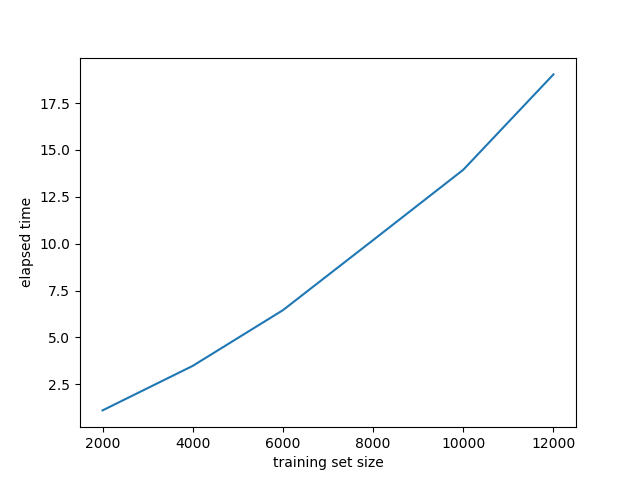


The famous MNIST dataset includes 70,000 images of hand drawn digits for classification into the numbers 0-9. Each 28 by 28 image is a matrix of values that can be anywhere from 0 to 255 depending on how dark the cell is. While there are many different ways of classifying this data, we are going to be looking at Support Vector Machines (SVMs) in particular. SVMs create a boundary by maximizing the margin between points, and use a kernel to represent the data in higher dimensions to create a classifier between points that would not otherwise be linearly separable.

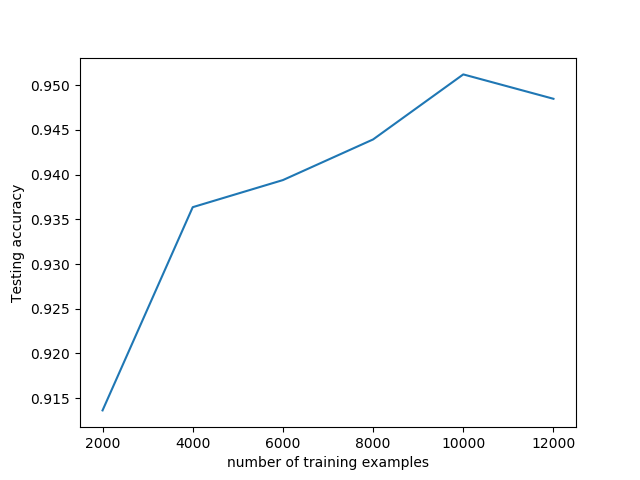
SVM is relatively fast, but Scikit-learn’s svm.SVC function may be unnecessarily slow for a multiple classification problem. SVM is normally built to create a boundary between two different classes (binary classification). For a problem like classifying digits into 10 categories, this doesn’t work. What Scikit-learn does to get around this is to do many “one vs. one” classifications, yielding n(n-1)/2 = 45 different combinations. For example, is it a 5 or a 7, 8 or a 9, etc. This makes it very time intensive to run on all 70,000 images (478.48 seconds). For our analysis, we are looking to find a way to speed up SVM. This will likely result in some loss of accuracy, but is necessary for applications with millions of data points. In our analysis, we will select a subset of 10,000 rows of data and run various procedures to speed up the SVM.

**Baseline SVM**

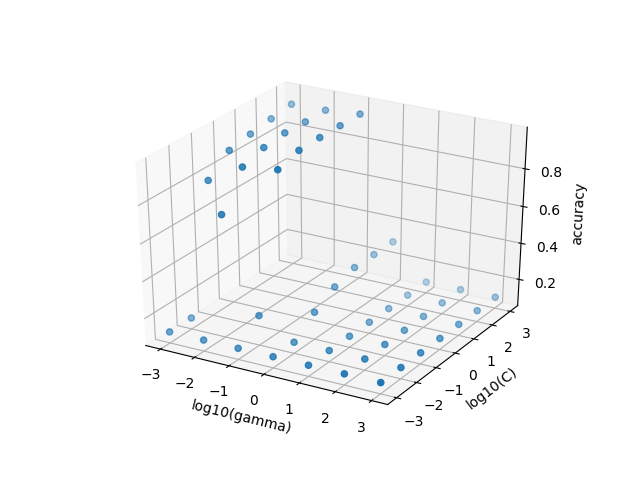
The time complexity of sklearn.svm.SVC is more than quadratic with respect to the number of samples[3]. By executing an SVM with different sized subsets of the original 70,000 data points, we can observe the time it takes. When we plotted time vs number of samples, the resulting plot looked somewhat quadratic.



Increasing the size of our training subset generally improves the performance of our model, though not always since the selection is random.



Support Vector Machines involve a number of hyperparameters. These include the kernel function to pick, the regularization parameter C, the parameter gamma of the kernel, and the degree in the case of a polynomial kernel. To determine the best settings for these parameters for the rbf kernel, we tested all combinations of different values on a logarithmic scale from 10^-3 to 10^3. By doing a grid search (trying out combinations of gammas and Cs) we are able to find the best values for the parameters of gamma and C. Of the values for gamma and C that we tested, the best results were from gamma = .01 and C = 10. We achieved 95.48% accuracy with the Gaussian rbf kernel with these values of gamma and C. We used these values of gamma and C for the rest of our SVM uses.



Starting with a baseline RBF kernel SVM on a random set of 10000 data points, we get an accuracy of 0.9527 in 25.51 seconds. Various other kernels were tried, giving mixed results.

**Kernel Time (seconds) Accuracy**

*RBF* 25.51 0.953

*Sigmoid* 25.29 0.889

*Polynomial degree 2* 23.20 0.948

*Polynomial degree 3* 29.76 0.935

*Polynomial degree 4* 37.55 0.898

Using a Sigmoid kernel on that same set of data gives us accuracy of 0.889 in 25.29 seconds. A Polynomial kernel with degree 2 gets accuracy 0.948 in 23.20 seconds; and with a degree of 3: 0.935 in 29.76 seconds. Going beyond degree 3 into degree 4 is a bad idea, as the runtime goes up to 37.55 seconds and accuracy goes down to 0.898. This is likely due to overfitting, because the training accuracy was 0.9295. Increasing the degree further only broadens this difference. Runtime seems to increase with the degree of the polynomial, and fitting a higher order polynomial can create too many overfitted boundaries. In any case, polynomial kernels do not perform as well as the RBF in our application.

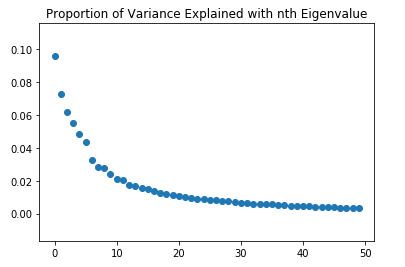
**Linear SVC**

Scikit-learn also includes a “LinearSVC” function which uses liblinear instead of libsvm, and works on a “one vs rest” rather than “one vs one” basis. A basic LinearSVC runs in only 0.75 seconds, which is several times faster. The accuracy, however, is only 0.903 (with an iteratively maximized C = 0.02). From the baseline rbf SVM, we lose 5% accuracy but it takes 1/33 of the time to run. Theoretically, increasing C should soften the margin-maximizing rule if it means more correct classification. By playing around with different values of C, the opposite seems to be the case. Decreasing C seems to increase accuracy and precision, while decreasing the runtime.

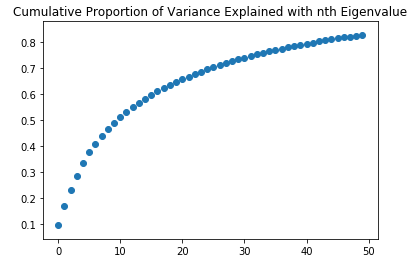
This method is fast enough that it can be run on all 70,000 rows in 9.53 seconds with 0.91 accuracy and precision. The time is saved because instead of doing 45 comparisons (each potential digit vs each other potential digit), they only do 10. For example, Is it a three vs is it not a three?

**Principal Components**

Principal Components Analysis is a widely used dimensional reduction technique where orthogonal linear combinations of variables are taken to maximize the proportion of variance explained. With our data, we can trim down to 20 components and preserve 64.5% of the variance. Trimming down to 50 components preserves 82.7% of the variance. The relationship is summarized in the screeplot of variance below.



Each additional component is less useful, and there is no clear “elbow” in the graph that tells us when we have chosen enough components. Hopefully, this dimensionality reduction will make the SVM run faster without compromising too much in accuracy.



Running an SVM on 10,000 rows of the reduced data runs in an astonishingly fast 1.9 seconds and gives an accuracy and precision of 0.94, which is almost as good as the baseline accuracy of 0.9527 but taking 12 times less time.

If we are to reduce the number of components, accuracy is lost and not much time is saved; 10 components drops accuracy down to 0.87 while saving less than a second of time. If we are to increase the number of components, we notice that accuracy plateaus at around 0.943, meaning there is little to be gained from adding components. The exact number of components you choose is up to interpretation, but anywhere between 30 and 70 runs in 1.5-2 seconds and achieves accuracy of 0.93-0.94.

This method also takes time to fit the PCA on the training set. This is negligible with our chunk of 10,000, but may be significant when applied to the whole dataset. Running a 30-component PCA SVM on the whole dataset takes 27.4 seconds for PCA fitting and another 43.2 seconds for the SVM, with an accuracy of 0.968.

**Conclusion**

**Method Time (seconds) Accuracy**

*SVM (rbf kernel) 478.48 0.981*

*LinearSVC 9.53 0.914*

*PCA SVM (30 principal components) 68.70 0.968*

Three of the best techniques are summarized above. Which one is the best trade-off between time and accuracy?

It depends on how much you value time and accuracy, and also on the size of the data you are working with. Scikit-learn’s default svm.SVC function is a solid and accurate tool for most uses, and its rbf kernel performs very well at classifying hand-drawn digits. However, if you are willing to sacrifice accuracy for time, LinearSVC is many times faster and the better choice. A PCA is extremely fast and accurate for moderately sized data, but takes a while to reduce very large datasets.

All these methods to reduce runtime are on the machine learning side; what has not been mentioned are data cleanup and processing techniques such as deskewing, rotating, and filtering, which have the potential to greatly improve the SVM. Some of professor Yann LeCun’s papers are able to achieve accuracy greater than 99%[1].

-some of the code that was used in our project-

from \_\_future\_\_ import division

import matplotlib.pyplot as plt

import random

import numpy as np

from sklearn import datasets, svm, metrics

import sklearn

from sklearn.model\_selection import train\_test\_split

mnist = datasets.fetch\_mldata("MNIST Original")

r = list(range(70000))

np.random.seed(seed=37)

np.random.shuffle(r)

X,y = mnist.data/255, mnist.target

xtrain, xtest, ytrain, ytest = train\_test\_split(X, y, test\_size=0.33, random\_state=41)

import time

start = time.time()

# Create a classifier: a support vector classifier

classifier = svm.SVC(gamma = 0.01, C = 10)

classifier.fit(xtrain, ytrain)

expected = ytest

predicted = classifier.predict(xtest)

end = time.time()

print(end - start)

print("Classification report for classifier %s:\n%s\n"

% (classifier, metrics.classification\_report(expected, predicted)))

print("Confusion matrix:\n%s" % metrics.confusion\_matrix(expected, predicted))

print(sklearn.metrics.accuracy\_score(expected, predicted))

from sklearn import decomposition

import numpy as np

import matplotlib

import matplotlib.pyplot as plt

start = time.time()

pca = decomposition.PCA(n\_components=30, svd\_solver='full')

pca.fit(xtrain)

pxtrain = pca.fit\_transform(xtrain)

pxtest = pca.transform(xtest)

end = time.time()

print(end - start)

plt.scatter(range(30), pca.explained\_variance\_ratio\_)

plt.title("Proportion of Variance Explained with nth Eigenvalue")

plt.show()

def SVMtime(inxtrain, inytrain, inxtest, inytest, gamma = 0.01, C = 1):

start = time.time()

# Create a classifier: a support vector classifier

classifier = svm.SVC(gamma = gamma, C = C)

classifier.fit(inxtrain, inytrain)

expected = inytest

predicted = classifier.predict(inxtest)

end = time.time()

print(end - start)

print("Classification report for classifier %s:\n%s\n"

% (classifier, metrics.classification\_report(expected, predicted)))

print("Confusion matrix:\n%s" % metrics.confusion\_matrix(expected, predicted))

print("Variance explained", sum(pca.explained\_variance\_ratio\_))

print("Accuracy", sklearn.metrics.accuracy\_score(expected, predicted))

SVMtime(pxtrain, ytrain, pxtest, ytest)

References

1.“THE MNIST DATABASE of handwritten digits” Yann LeCun, 1998 http://yann.lecun.com/exdb/mnist/

2.“Understanding SVMs for Digit Classification using MNIST Dataset” Vijayasaradhi Indurthi, November 26, 2015

<https://www.linkedin.com/pulse/understanding-svms-digit-classification-using-mnist-dataset-indurthi>

3.“sklearn.svm.SVC¶” Scikit Learn 2016

<http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html>