CAPTCHA recognition based on DSP using machine learning techniques

Guang Yang
Dept. of Electrical Engineering
Columbia University
New York, US
gy2237@columbia.edu

Jiayi Fu
Dept. of Electrical Engineering
Columbia University
New York, US
jf3030@columbia.edu

Zhangyu Liu
Dept. of Electrical Engineering
Columbia University
New York, US
zl2513@columbia.edu

Shutian Liu
Dept. of Electrical Engineering
Columbia University
New York, US
sl4039@columbia.edu

***Abstract*— Decoding CAPTCHA images is an interesting but challenging task: until today, no artificial intelligence (AI) can pass the Turing test. However, it is possible to train AIs to "see" and then "understand" the CAPTCHA code. After carefully choosing the CAPTCHA images, they will be converted to gray scale, binarized, de-noised, split and then normalized by a variety of image processing techniques such as the max frequency filtering and connected-component labeling. With the standardized binary file, we will apply two machine learning algorithms: K Nearest Neighbors method and Support Vector Classification to train and test the model. The designed model works very well within the scope of our problem.**

Keywords— CAPTCHA recognizer; Machine learning; K Nearest Neighbors; Support Vector Machine; Optical character recognition; Connected-component labeling

I. INTRODUCTION

Nowadays, CAPTCHA code is widely used by multiple websites for security reasons. As CAPTCHA is the abbreviation of "Completely Automated Public Turing test to tell Computers and Humans Apart", it is self-explanatory that this code is designed to be not interpretable to the artificial intelligence. Usually this intent is achieved by adding noisy pixels, distorting and overlapping the numbers/letters in the codes. However, with the fast-developing of imaging processing techniques and machine learning algorithms, decoding some simple CAPTCHA codes is not an impossible mission for AIs anymore. It is an interesting and meaningful topic, as learning how to decode the CAPTCHA codes is helpful for building safer websites under the attacks from unauthorized Artificial Intelligence(AI). In our project, we will go over the procedures to decode some simple CAPTCHA codes. Here, by "simple" we mean that the codes have no crowded symbols, scaling, and rotations. Showing in Figure 1 below is a simple CAPTCHA code used for our project. As we can see, although having no distortions and overlaps, the simple codes still have noisy pixels and we will need to suppress the noises.

Figure 1 A CAPTCHA code

Many blogs and papers have discussed about the topic on decoding CAPTCHA, and it is actually one of the challenges for "The Kaggle competition for machine learning" [1]. There are basically two steps for designing the CAPTCHA recognizer. The first step is to convert the CAPTCHA images to proper forms that can be digitally processed, and the second step is to implement some machine learning algorithms to train the computers how to "understand" the contents of the CAPTCHA images.

As is shown in Figure 2, we use python to write web crawler to download the CAPTCHA code online, and then use image processing techniques like max frequency filtering and connected-component labeling method to suppress the noises. In the next step, we use the Optical Character Recognition (OCR) tech provided by tesseract to help us make some categorization of the training dataset. After constructing the training dataset, we extract the info contained in the images to get feature vectors of each image and use them in the final verification. There are many useful methods for classification. Because our problem is a multi classification problem, so we choose a powerful method as Support Vector Machine (SVM) to tackle our image recognizer problem. At the same time, we have tried the K Nearest Neighbors method to make some

comparison with SVM. The whole codes used in this project are written in python2.7. To do our project, we have imported the image-processing library Pillow and machine learning library scikit-learn for convenience.

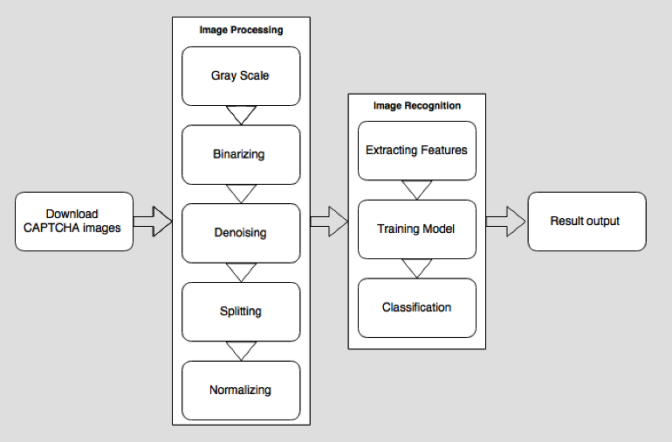

Figure 2 Flow chart of the system. System consists of four parts: data collecting, image processing, image recognition, and result display.

Overall, our result is really good: we have got an accuracy of nearly 99.7% in self testing of SVM model. And we have classified 1000 CAPTCHA codes and found all the verification is correct. Also, we find that K Neighbors method also has a good performance but it will run much slower than SVM model. Also, if the number of classification labels becomes much larger, the K Neighbors method will run much slower and have a worse performance.

II. Technical Approach

A. *Image Pre-Processing*

First, to gather some simple CAPTCHA images, we use Python to write a web crawler to get the images from url ('http://scm.sf-express.com/isc-vmi/loginmgmt/imgcode') and save them locally on our computers. Basically, each of images contains four numbers or letters. To recognize the content in the picture, we need to recognize them one by one, so we need to cut every image into four pieces. Again, as there is no overlap, and the letters and numbers are roughly equally distributed in distance on the images, we can directly cut an image into four equal pieces and then every piece contains only one letter/number. Then we can convert the colorful letters/numbers to gray scale and further to binary bits. At this point, it is necessary to filter out the noisy pixels and lines with certain criteria. If the result is not good enough, it is possible to further enhance the images so that the filter is easier to set. The python web crawler we used requests lib, and the training set is saved in the *pictures* folder, while the test set is save the in the *testpictures* folder and *testpictures_k* folder.

B. *Image Filtering*

In this part, we aim to use some filtering techniques to suppress the background noise combined with lines and points. We have tried several techniques including max frequency filtering and connected-component labeling method. And the process is included in *BinarizingImage* class described in *binary.py*.

In particular, here for each point we use connectivity based filtering followed by connected component labeling. In other words, for the former we considered neighbors of our point of interest to remove line and scattered point noise by running the graph traversal, for the later we take the neighbors to our point of interest to compute connected components. Because we are filtering binary images and value of a bit is either 0 or 255, we define the foreground to be 255. We walked through all pixels from the upper left point after removing the outline of the image to remove noises and small connected components.

C. *Image Categorization*

After getting the filtered, separated binary bits for every number/letter on all of the CAPTCHA images, we could use tesseract, a optical character recognition (OCR) toolbox to do preliminary classifications on them. We run tesseract on mode psm -10 to select, and the process was written in shellOCR.sh. Although tesseract can read the text on the images, it is not perfect as it transfers texts into strings and not all the strings can be categorized.

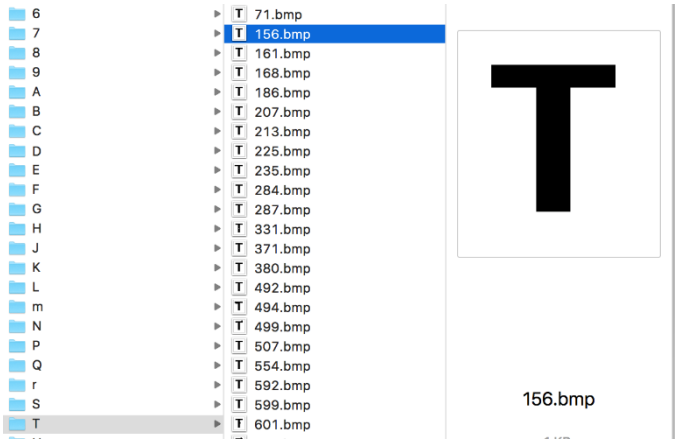

Figure 3 Folders and characters in the folders.

The correctness of the OCR has been nearly 96% after filtering, and then we recognized images manually to create a better training set. In fact, in our case, we have over 10000 images and only more than 346 of them are not automatically saved to a proper dictionary. For those letters/numbers successfully categorized, we put them into different folder with all same letters/numbers going into a single folder. Presumably, there are letters/numbers categorized to wrong directories. As human eyes can easily recognize them, we can point them out and move them into right folders. The CAPTCHA images contain letters in capital and numbers from 0-9. After categorization, we found that the CAPTCHA images do not contains '0', '1', and 'O','I', for these letters will even confuse human eyes. So there are 32 folders in total. We could see this categorized result in *trainningDataset* folder. What we found interesting in this process is that the

CAPTCHA images we used contain lots of 'W' and 'M', which are more difficult for OCR to recognize than other letters/numbers. We use the Categorization result as training dataset. The categorization can be done automatically by *createTrainingDataset.py*. Results of the categorized characters is shown in Figure 3.

D. *Image Feature Detection*

After getting the categorized images, we need to find out what are the features of interest for every image. In other words, we will convert the info contained in the processed images into an array of 1 and 0. Figure 3 shows how we extracted the feature.

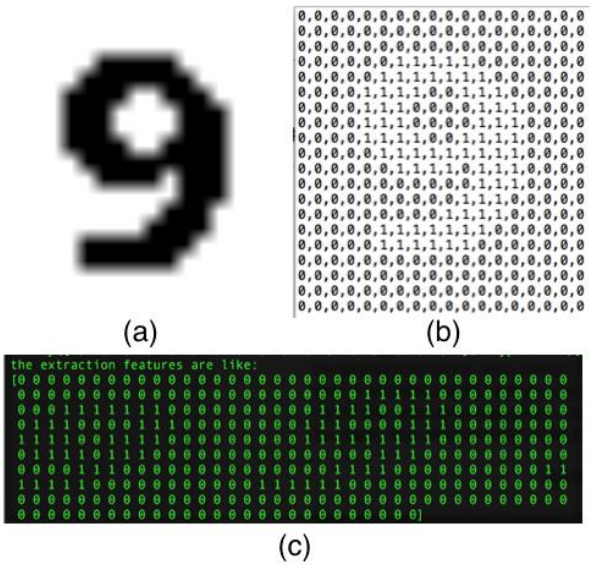

Figure 4 Image feature extraction. Panel a) shows a processed image labeled as '9'; panel b) shows how the image changed to features of 0 and 1; panel c) shows how the final format of feature, saved as an 1D array.

Using *Extraction.py*, we can save all images' features in *traindataX.csv* and saved all images' labels in *traindatay.csv*. In the .csv file, each binary bit is separated by a comma. Then with these bits, we can reconstruct the original image to a clearer image with white background (binary 0) and a well-depicted black letter/number (binary 1). Comparing it to the number 9 in Figure 1, we can see that it looks much clearer now in a black color with a purely white background. Therefore, we believe that it is not necessary to do any image enhancement for our purpose.

At this point, all the numbers and letters are categorized and saved to proper directories, so it is time to program our computers to recognize the text in the images. For training and testing, we will use two different approaches: the first one is based on the K Nearest Neighbors algorithm, the first one is using the Support Vector Machine (SVM) as the supervised learning model.

E. *K Nearest Neighbors*

K Nearest Neighbors algorithm is one of the easiest algorithms to do classification. The key idea is using the Euclidean distances between data points to classify a given point to a proper class, based on the class information of the nearest K points to the given point. This algorithm is convenient as Euclidean distance is suitable for any N-dimensional plane. For our original images, each of them is composed of a 72×20 pixel array, and we have split the image so every single number/letter is represented by approximately a 1×360 binary bit line.

Figure 5 The label and feature stored in .csv file. The up one is the training set X: it contains all the binary bits for every letter/number we have extracted; the lower one is the training set y: it contains all the labels for every binary line.

To apply K Nearest Neighbors algorithm to our project, we will have to define a training set and a test set. The training set is like the base line of classifying the images, and the test set is the challenge we need to solve by our design. Here, we are taking the labels of the binary arrays in the training dataset as the label map. After comparing the binary arrays in the test set to the binary arrays in the training set, we can find out the top k closest neighbors to the numbers/letters in the test set and get their corresponding labels by searching them in the label map. Then, by counting the times of appearance of a particular label and putting them in an descending order, we can assign the most frequently appeared label to be the label of the lines in the training set. As the label is the original number or letter in the image, we hereby get our desired results.

Shown in *KneighborsModelPrediction.py* is the results. To justify, *KneighborsModelPrediction.py* can rename the images

in *testpictures_k* folder by recognition results of the K Neares Neighbors method.

F. *Support Vector Classification (SVC)*

After getting the test result from K Nearest Neighbors algorithm, we can go further by taking a kernelized learning algorithm via Support Vector Machine (SVM). The advantage of using a kernel function is that the kernel function can assign different weights to different neighbors based on the distance between the neighbors and the point of interest. There are a variety of kernel functions available in the Scikit-learn library, and based on the information we find online we determine to use the Gaussian basis function kernel (RBF kernel). The RBF kernel for two points x, and x' is defined as

$$K(x, x') = exp(-\frac{\|x - x'\|^2}{2\delta^2})$$

Here, the $\|x - x'\|^2$ is the Euclidean distance between the two points. It is pretty clear that with the distance increasing to infinity the value of the kernel decreases to 0 and when x =x', the value of the kernel is simply 1000. In other word, the closer neighbors plays dominating roles when determining the label of the testing bits. Indeed, to perform the support vector classification, we can use the Scikit-learn library built-in function to find out which is the best kernel function to perform our task. By *GridSearchCV(svr, parameters)*, we can see that the result showing RBF kernel is the best kernel function for the image classification.

To perform the support vector classification, we need to use the function *clf.fit*(X,y), which fits the array X holding all training samples, and the array y containing all labels. After being fitted, the model then can be used to predict the test samples by *clf.predict*(test data). After printing out the predicted test set values, we can change the file name of each image in the *test* folder to its *CAPTCHA.jpg*. Thus, by checking whether an image is displaying the CAPTCHA code the same as its file name, we can know is the prediction correct or not.

For SVM model, we directly used svm provided by scikit-learn. The training model process is shown in *SVMmodel.py*, and the classification process is shown in *finalPrediction.py*. The *classfication.py* will rename files in *testpictures folder* by the result of recognition.

III. Experiments

A. *Filtering Images*

For a given image, the filtering code *binary.py* will return 4 clean and equal spilt bmp files, which represent 4 characters in the original picture. For example, as shown in the figure below, the original picture, which contains "T3TG", is on the top and followed by results of each step, in which we removed a certain sort of noise.

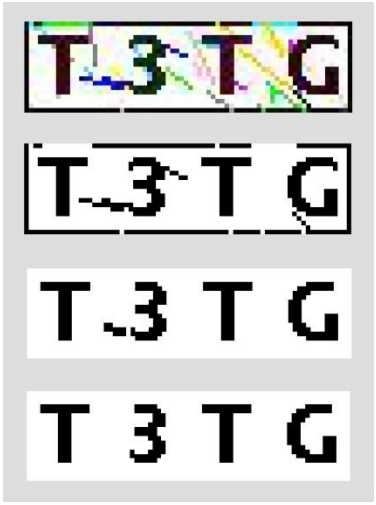

Figure 6 From top to bottom: original image and results from each step.

At first, we converted the picture into binary scale, which filtered most of background noises. Then we used 4 and 8-connectivity based filtering to remove the noise combined with lines and points. However, duo to all characters contained 2x2 like blocks, such kind of noise couldn't be removed only by filtering technique. So in the final step, we used 8-connectivity based labeling to compute connected components, then removed all components that smaller than a threshold size.

B. *K Nearest Neighbors*

For a given *CAPTCHA.jpg*, the K Nearest Neighbor code *KNeighborsModelPrediction.py* will return the numbers and letters showing on the image in the output array. Below Figure 7 displays the result showing for an input CAPTCHA code with the K Nearest Neighbors algorithm.

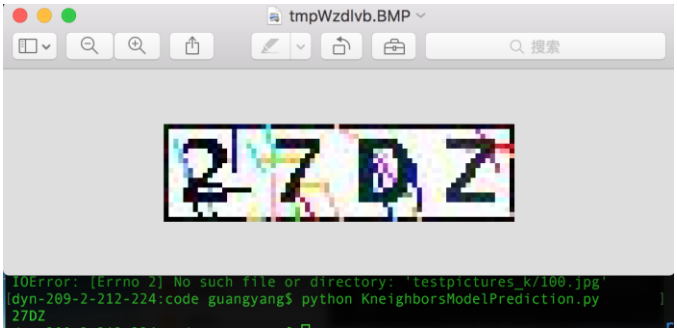

Figure 7 Experiment Result after running the K Nearest Neighbors codes for the input "27DZ" CAPTCHA image.

As everyone can see here, the printing result of the script is exactly the same as the input text. It is not a best case picked by us. We have run over all the codes in test set and they are all correctly recognized by the program. It is still too easy to say that our program has an accuracy of 100%. Our dataset is limited and simple, with every number/letter checked to be in the right place, so the risk of data contamination is reduced to the minimum. Running the algorithm in an ideal environment,

we are surprised that our code does not showing any wrong recognition result. Also, when we read the results we may carelessly miss a wrong result surrounded by all the other correct ones. For the running time, because there is no pre-set searching condition for the K Nearest Neighbors algorithm, for every new inquiry the program must run over all the labels in the label map. In other words, for a new inquiry the program must go over the whole training set. This process may take a pretty long period of time. Once the training set is loaded, for further inquiries the program can quickly compute the results.

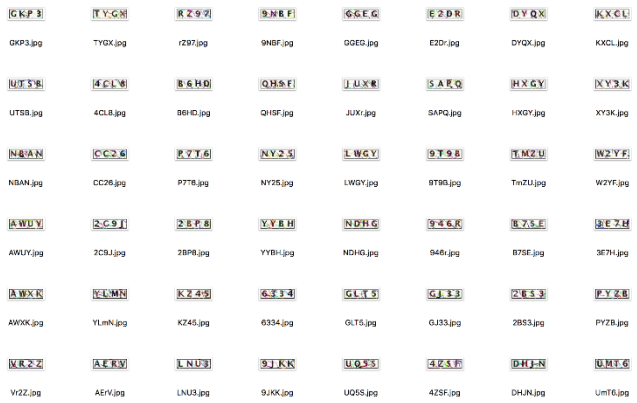

Figure 8 Experiment results using SVM method.

C. *Support Vector Classification*

After getting the experimental result of K Nearest Neighbors, let's look at the experimental result of the Support Vector Classification. Upon requests, we can use the *finalPrediction.py* function to print out the recognition result for a specific CAPTCHA image that is inquired by the user, or use the *classification.py* to print out the recognition results for all the images in the test set, and update their file names to be the same as the CAPTCHA codes on the images. Figure 9 displays the result in the command window for the former request.

Self testing by cross-validation, we know that the accuracy is about 99.7% for the SVC model. We also checked all the images in the output directory of the *classification.py* function manually, and we do not find any mismatch between an image and its file name. In other words, there is no error detection showing in the results. A small fraction of the ~1000 images in the result directory is shown in Figure 7.

Presumably, SVM should have better results than the K Nearest Neighbors algorithm, as the kernel function operates in high dimensions and do not need to know the coordinates of the implicit feature domain. However, here we do not see any improvement of results by replacing the K Nearest Neighbors by the SVM, because the accuracy given by the KNN computing is already 100%. We believe that if we use some more complicate CAPTCHA images to train the model, the advantage of using SVM will be highlighted.

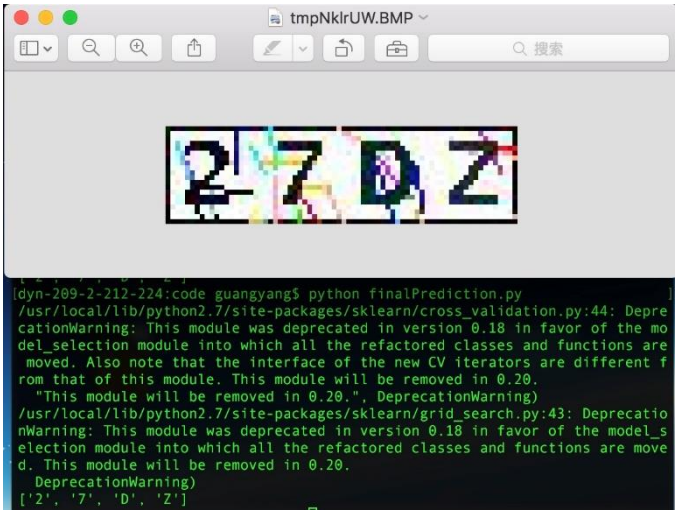

Figure 9 Command window result. Using SVM method.

Figure 10 Experiment results using K nearest neighbors method.

IV. Discussion

As everyone can see here, the printing result of the script is exactly the same as the input text. It is not a best case picked by us. From the very beginning of our project, we determined to do the CAPTCHA recognition, and basically we are not distracted from this goal. We know that it is pretty unrealistic for us to take considerations of complicate CAPTCHA codes in this short period of time, so we focus on the simple ones as defined early. It turns out that both the K Nearest Neighbors and the Support Vector Classification work pretty well for our selected scope of problem. The only trouble that come to our way was when we trying to use the K-means algorithm to do the recognition, there are some clusters get lost in this process. We believe this problem is caused by the ambiguity between some similar symbols and numbers. For example, The Y cluster and the V cluster could be really close to each other so that they are not distinguishable by the K-means algorithm and therefore there is no data left for the Y cluster. Under this condition, we lose the cluster Y. Facing this problem, it is fair for us to say that K-means is not the best algorithm for our

design goal, and then we change to use K Nearest Neighbors instead.

In the future, it is possible to make the design even works for more complicated CAPTCHA images. With the rotations, overlaps, and distortions, the equal cutting strategy will be problematic, and a new approach to separate every single letter/number in the image is needed. Assume that the harder CAPTCHA contains random number of characters, then we have to first count them. And the noise of the images is more confusing that it does not have a evenly distributed background, and has random scratches of a variety of shapes which made the filtering and the binarizing even more difficult. Generally speaking, for more complicated CAPTCHA images, we need to design better signal processing methods for pretreatment before we began classifying. To obtain a better binary image, we need local binarization to handle the images whose background brightness is changing gradually. For the binarized image, it may not be so satisfying that we are only left with a white background and some characters in black, there may still have some scratches. So we may have to use the connected-component method to detect the components whose area is small enough for us to remove. If we are given that the characters is separated, then we can count the number of the left connected-component to determine how many characters are in the image. And by locating them, we can design a proper way to cut the image to obtain single characters. Maybe you think that we can begin classifying in the next step, but the truth may not be that easy, for the noise scratches can be connected with the characters that we cannot remove them. Then, with the scratches shown as part of the characters, the classification will be difficult, Also, there may be transformations either linear or nonlinear performed on the characters. Maybe we have to do some erosion or dilation to let the weight of the scratches become smaller, and try different mapping functions to map the distorted characters back to their normal patterns.

At the same time, the measurement between the standard letters/numbers and the given inputs will change a lot, so the algorithms we used (K Nearest Neighbors and Support Vector Classification) may be problematic under this situation. Especially for the K Nearest Neighbors algorithm, because by simply rotating a line by a small angel, the Euclidean distance between the original and the rotated one will be very large. Again, as CAPTCHA is designed to keep AIs out, a desired trend is that the design of the CAPTCHA will always out run the machine learning speed. Therefore, the challenge of decoding CAPTCHA always exists.

ACKNOWLEDGMENT

The authors would like to thank Professor John Wright for his wholehearted teaching. What we have learned from this course will be the valuable assets for our future study and careers.

The authors also want to thank all TAs for their patient helps. Thanks for Shun Feng Supply Chain Collaboration for providing all the CAPTCHA images.

REFERENCES

[1] Osuna, Edgar, Robert Freund, and Federico Girosit. "Training support vector machines: an application to face detection." *Computer vision and pattern recognition, 1997. Proceedings., 1997 IEEE computer society conference on*. IEEE, 1997.

[2] Tax, David MJ, and Robert PW Duin. "Support vector data description." *Machine learning* 54.1 (2004): 45-66.

[3] Keller, James M., Michael R. Gray, and James A. Givens. "A fuzzy k-nearest neighbor algorithm." *IEEE transactions on systems, man, and cybernetics* 4 (1985): 580-585.

[4] Beyer, Kevin, et al. "When is “nearest neighbor” meaningful?." *International conference on database theory*. Springer Berlin Heidelberg, 1999.

[5] Cortes, Corinna, and Vladimir Vapnik. "Support-vector networks." *Machine learning* 20.3 (1995): 273-297.

[6] Htea. "jianshu blog" http://www.jianshu.com/p/41127bf90ca9

APPENDIX

In this project, all the programs are running on MACBOOK Pro, and the operating system is OS X EI Captain 10.11.6. The program used are written using python2.7. The program has import python libs as scikit-learn, numpy, and pillow, so to run the codes, first use pip command to install these libs. Also, to use tesseract OCR function, we need to use homebrew to install tesseract. All the description of functions in each python code and shell command can be found in *readme.mdown* and *readme.html*. The CAPTCHA images are from Shun Feng Supply Chain Collaboration (http://scm.sf-express.com).