CS342 Machine Learning

Assignment 2: Digit Recogniser

Name: Chris George, ID: 1305455, Department: Mathematics

1. Abstract

This report showcases my participation in the “Digit Recogniser” Kaggle competition and my findings on the MNIST digits data. Each image is stored as a 1x784 vector of pixel intensities (a 28x28 image). There are approximately the same number of images from each class, and their average pixel density varies between classes. I implemented Random Forests, Support Vector Machines, K-Nearest Neighbours and Convolutional Neural Networks to classify this data, and used a hard voting ensemble model to combine several of these base models. I extracted features using Sobel Edge Detection, Gaussian Blurring, Principal Component Analysis and Histogram of Oriented Gradients to produce diversity between base models. For CNN, I expanded the dataset by adding in image rotations. Gaussian blurring and HOG were the most successful features, KNN and CNN were the most accurate base models and the ensembles outperformed most base models. However, CNN achieved the best accuracy - 98.51%.

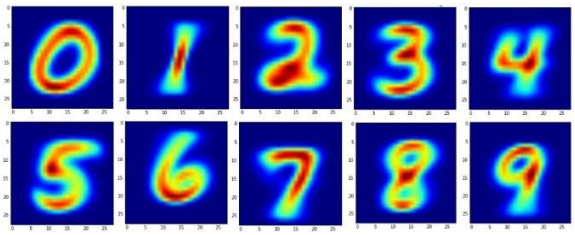
2. Data Exploration and Feature Engineering (FE)

The first step of my data exploration was to check for an imbalanced distribution, but the results suggest that there isn't a significant difference between the numbers of images for each class, so class imbalance shouldn't be a problem.

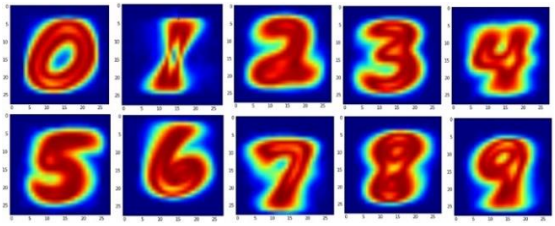
Index	1	7	3	9	2	6	0	4	8	5
label	4684	4401	4351	4188	4177	4137	4132	4072	4063	3795

I split the training data by class and computed the means and standard deviations of each pixel. I plotted these values, as shown below – dark blue pixels show a low pixel value, and as the pixels increase in value the pixel colour in the plot changes to light blue to yellow to red. These plots suggest that perhaps blurring the images to reduce noise would help make images of the same class look similar (so would be a good feature for a classification model).

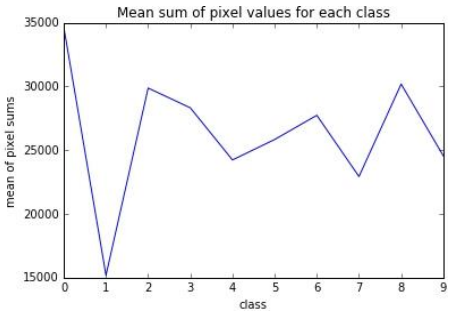
Plot of mean pixel values

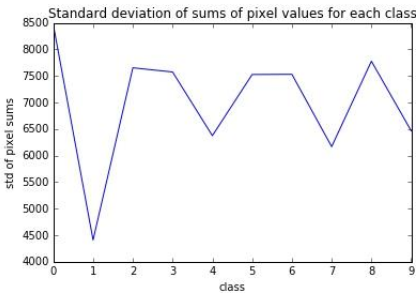


Plot of standard deviations of pixel values



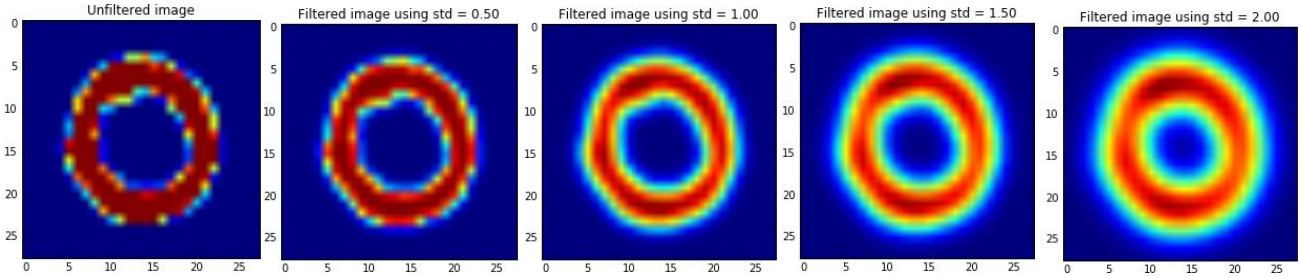
Since the pixel intensities vary between classes I decided to sum up pixel values for each observation and plot the mean and standard deviation of these sums for each class. This shows a distinction between the total pixel intensities of each class. I also investigated various feature engineering methods.





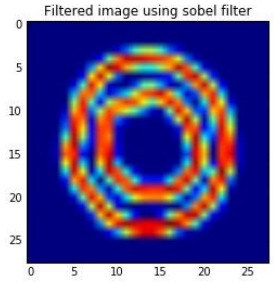
Blurring:

Blurring should make images of the same class look more alike by removing some differences between images which represent the same digit. I used a Gaussian filter, which will recalculate the value of each pixel as a weighted average of itself and its surrounding pixels, using weights which correspond to a 2-D Gaussian distribution centred at the corresponding pixel (as you get further away from the corresponding pixel – the mean, the weights decrease proportionally to a 2-D Gaussian distribution). This blurring method is optimised by tuning the standard deviation of the Gaussian distribution. The plots below show an image representing the digit 0 (the second image in the dataset) filtered for various parameter values.



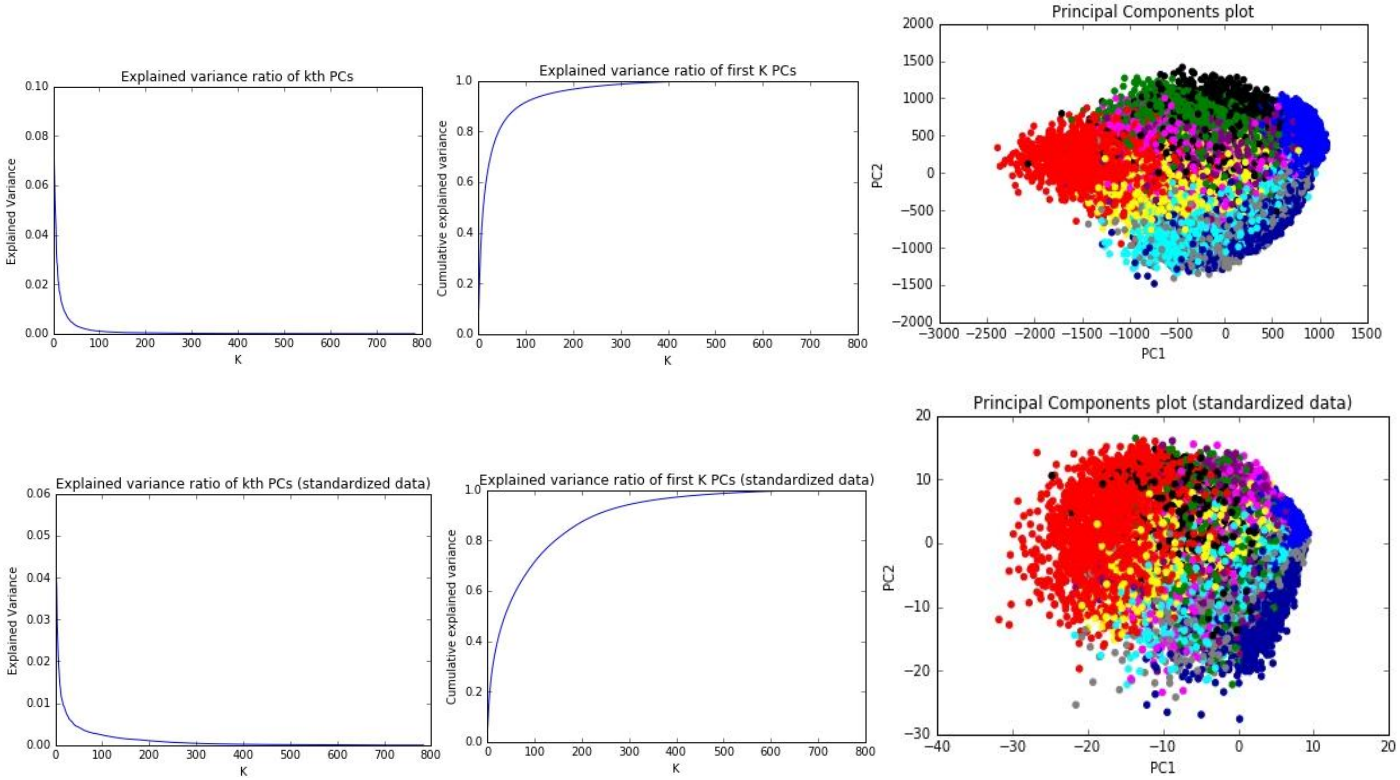
Edge detection:

Since images of the same class should have similar edges, I used a Sobel filter to detect edges which, like Gaussian blurring, recalculates each pixel value as a weighted average of itself and its surrounding pixel values. However, the Sobel filter will approximate the gradients in the image to highlight edges. The Sobel kernel is a specific edge-detection operator and does not require any parameter tuning. The plot to the right shows the result of using the Sobel kernel to filter the same image I used to demonstrate Gaussian blurring.



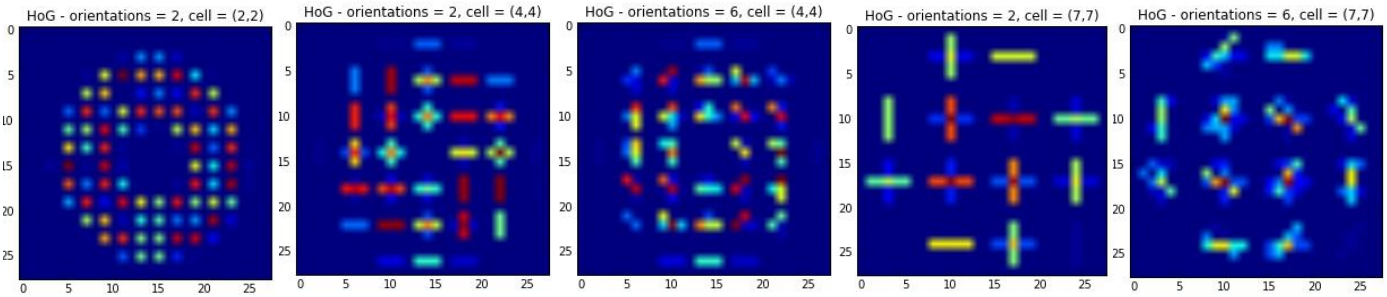
Principal Component Analysis (PCA):

Since the data is high dimensional (784 pixels), I used Principal Component Analysis as a FE and dimensionality reduction method. PCA computes linear combinations of the original (possibly correlated) attributes, in order to return new uncorrelated attributes (called principal components) which preserve the variance in the data. The principal components are computed by projecting the data onto the eigenvectors of the data covariance matrix ($PC_i = Xe_i$ where e_i is the i^{th} eigenvector, with eigenvectors ordered descending by size of their eigenvalues). The eigenvalues represent the variance in the original data which is explained by the corresponding component. The dimension of the data can be reduced by keeping the first K principal components, which explain a certain ratio of variability in the data. I decided to use PCA on both the original data and the standardized data, to see if certain models worked better when preserving the scale in the data. For both cases, the plots below show the amount of variability in the data explained by the first K principal components, and a scatter plot of the first two components, coloured by class. Even just the first two principal components start to introduce clusters of classes. To explain 95% of variance in the data, 151 components are needed in the non-standardized case, and 318 in the standardized case.



Histogram of Oriented Gradients (HOG):

I also implemented the Histogram of Oriented Gradients method. This method divides images into cells, and for each cell computes a histogram of gradient directions of the pixels within the cell. These histograms highlight the appearance of the corresponding cell, simplifying the image and extracting gradient features. It is possible to normalise each cell with pixel intensities across several cells (such a region is called a block). However, this extra step is used to remove variations in the images due to changing illuminations/shadowing. The images in the MNIST dataset are hand-written digits, so I do not think that this extra step is necessary for the MNIST digits data. The tuning parameters are the cell dimensions and the number of orientations (gradient directions). I set the number of cells per block to (1,1) to avoid the normalisation step and simplify the parameter tuning. The plot below shows the HOG image returned using several parameters on the same image as before.



3. Machine Learning Methods

This section highlights my methods, results and analysis for the various classification models and feature engineering approaches I used to predict on the MNIST digits data, with an accuracy performance criterion. The first stage of my approach was to implement and submit predictions for each model-feature engineering combination using sensible parameters, and to analyse which combinations performed best and why. I ran 10-fold cross validation on the training data to test each model before submission.

To allow for a reasonable comparison of models, I used the same FE parameters for each model. For Gaussian Blurring I used a standard deviation of 1.0, to smooth the images sufficiently without losing too much information. For PCA I used 150 principal components, to significantly reduce the dimensionality of the data whilst retaining around 95% of the variance when using the original dataset and 81% when using the standardized data. For HOG, I used cell dimensions of (4,4) and 5 orientations to get a good trade-off between the amount of information represented by the HOG features and the data pre-processing speed.

For the second stage of my approach, I implemented and submitted predictions for a convolutional neural network, trained on an expanded dataset (I added rotations of bootstrap sampled images to the data). I then combined some of my best performing models into an ensemble to boost the score. Finally, I decided to tune the parameters of the best performing models and re-run the ensemble model to try and improve my ensemble performance. I then improved my CNN model to achieve my highest accuracy score of 98.51%.

Random Forest (submissions 1-6)

A random forest is an ensemble model which uses a bagging technique on decision trees. It trains multiple deep decision trees (which tend to over-fit, having high variance and low bias) on bootstrapped samples of the input observations and attributes, and then aggregates the predictions of the decision trees. Taking bootstrap samples of the observations and then taking bootstrap samples of the attributes reduces the covariance of the base decision trees, while aggregating the predictions reduces the variance of the model to achieve a good bias-variance-covariance trade-off.

The main parameters for this model are the bootstrap sample size, bootstrap attribute subset size, and the number of decision trees trained. I used the 'RandomForestClassifier" from the sklearn package, which uses a bootstrap sample size equal to that of the input data. I then decided to use an "entropy" criterion for training the decision trees, and left the other parameters as their default values (bootstrap attribute subset size = sqrt(number of attributes), and using 10 decision trees). Below are the accuracy results I got on the submission data (submitted in the same order, left to right):

Feature Engineering:	None	Gaussian Blurring	Sobel Edge Detection	PCA	PCA (standardized)	Histogram of Oriented Gradients
Accuracy:	0.94600	0.94743	0.91629	0.89057	0.90186	0.94014

This model performed very well with the original dataset. The Gaussian blurring feature engineering method was the most successful, which isn't unexpected since it removes noise between images of the same class. The HoG features also performed well, but the accuracy was slightly lower than when using the original data, which would perhaps not be the case if I used different HOG parameters. The Sobel edge features and principal components yielded the lowest accuracy scores. Perhaps solely using edges isn't an effective FE method, as it may remove other important features in the image. It is also not surprising that predicting on only 150 principal components wasn't very successful for Random Forests, as these models tend to work well for high dimensional data (so maybe reducing the dimension isn't the best idea).

Support Vector Machine (submissions 7-12)

This classifier computes a linear decision boundary in the feature space which maximises the margin between the various classes and the decision boundary, subject to the constraint that all training observations are classified correctly (training involves solving an optimisation problem). Since the decision boundary is defined by the points closest to it (the support vectors), this method induces observation sparsity (only support vectors are needed to compute predictions) and hence prediction times are very fast. A slack variable 'C' can be added to relax the

constraint, which allows the linear boundary to approximate a solution when the data is not linearly separable, and helps to control overfitting. Alternatively, a kernelized SVM can be used to compute a linear decision boundary on a different space to that of the input data by substituting dot products in the decision boundary calculations with kernel functions (i.e. it acts as a feature engineering method by computing the decision boundary on a space which is different to that of the input data).

I implemented a linear SVM rather than a kernelized SVM, as although the latter might yield better results, using a kernelized SVM would introduce more parameters to tune and a linear SVM is usually faster to train. I hoped that the high number of attributes in my data and features should hopefully allow for a good enough approximation to a linear decision boundary. I used the "LinearSVC" implementation from the sklearn package, and used default parameters (C=1, squared-hinge loss, Euclidean distance, and one-vs-rest multiclass strategy). As an SVM is a binary-classification model, the one-vs-rest multiclass strategy finds a decision boundary for each class (labelling all other classes the same) and computes a confidence score for the observation belonging to that class. It then predicts the class which has the highest such score. Below are the accuracy results I obtained on the submission data (submitted in that order, left to right):

Feature Engineering:	None	Gaussian Blurring	Sobel Edge Detection	PCA	PCA (standardized)	Histogram of Oriented Gradients
Accuracy:	0.83971	0.86529	0.81771	0.81486	0.90514	0.94157

These results show a much lower performance than the Random Forest for all FE methods except HOG and standardized PCA. This suggests that it is not possible to accurately separate some of these features with a linear decision boundary. This also suggests that I may have used too low a value for the relaxation parameter C for the first four FE methods, since if C is too low the model tends to under-fit (with a high bias and low variance), as suggested by the first four results. Moreover, maybe using 150 principal components was not enough for this model, as a higher number of components might make the data more linearly separable.

K-Nearest Neighbours (submissions 13-18)

A K-NN model classes a new observation by finding its 'k' closest neighbours in the training data and returning the most common class amongst these neighbours. All of the computation is carried out when calculating the distances between observations to find the nearest neighbours in the prediction stage. The main parameters which need tuning are the number of neighbours, k, and the distance metric used. A higher value of k gives high bias and low variance (using more neighbours will reduce the variance between models but will increase the systematic error in predictions). It is also possible to assign weights to the neighbours, so that closer neighours have a higher weight when computing the prediction – this can help to deal with noisy data and outliers.

I chose to use a K-NN model as the scale of each pixel is the same and my data exploration shows how images of different class tend to have different total pixel densities, so you would expect images of the same class to be closer together than images of different classes. For this reason, I chose to use k=3 (this should give low bias whilst keeping the variance reasonably low too). I used the "KNeighborsClassifier" from the sklearn package, with a Euclidean distance metric.
Below are the accuracy results I obtained on the submission data (submitted in the same order, left to right):

Feature Engineering:	None	Gaussian Blurring	Sobel Edge Detection	PCA	PCA (standardized)	Histogram of Oriented Gradients
Accuracy:	0.96857	0.97343	0.95829	0.97086	0.95100	0.97386

This model yielded very high accuracy results. As before, using Sobel edge features gave the least accurate model. HOG features gave the most accurate model again, closely followed by Gaussian blurring and principal components. For k-NN, principal components on the original data seem to be a good FE method as the first 150 principal components outperformed the original data, and reducing the dimensions of the data would also speed up distance calculations, discard less useful attributes and avoid extreme distances being calculated due to high dimensions. However, principal components on standardized data gave the lowest scores, so perhaps it is best to preserve scale with KNN as standardizing would decrease the magnitude of principal components, and hence decrease distances between different classes.

Convolutional Neural Network (submissions 19 and 24)

I implemented a convolutional neural network, since they are very popular for object recognition in images. CNNs are deep neural networks with multiple hidden layers which are not always fully connected (can have dropout layers between fully connected layers to reduce the number of parameters and control overfitting). CNNs also have convolutional layers made up of filters which act as feature engineering methods, followed by pooling layers to down-sample which can help reduce overfitting and capture translation invariant features. These networks tend to have less parameters than normal neural networks, and are usually easier to fit. Due to the variety of layers which can be used, there are many parameters and layer combinations which can be chosen. I implemented the CNN using the Theano & Lasagne libraries, and using code from online CNN examples as a guideline. I trained for 15 epochs with a learning rate of 0.0005 using 7 layers:

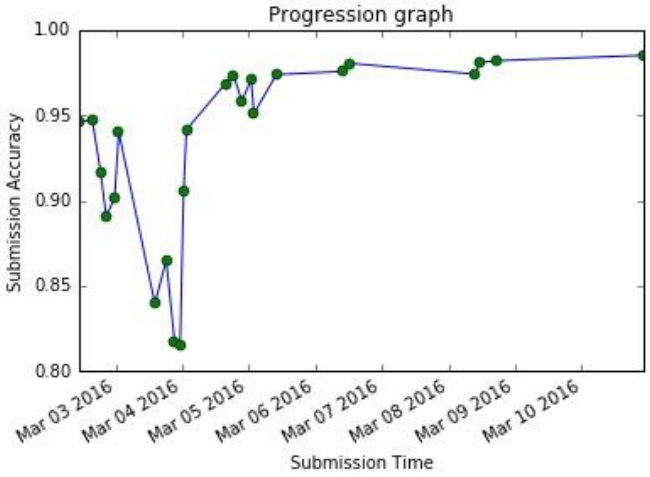
1. Input layer, which takes the 28x28 image as input
2. Convolutional layer, using 10 (3x3) filters and a rectified linear unit activation function (ReLU)
3. Max pooling layer, with a pooling filter of size (2,2)
4. Convolutional layer, using 15 (2x2) filters and a ReLU activation function
5. Fully connected layer, with 200 hidden units
6. Dropout layer, with dropout probability p=0.5
7. Fully connected output Layer, with 10 output units and a softmax activation function

Since the convolution layers act as feature engineering methods, and neural nets require large amounts of data in order to avoid overfitting, I extended the dataset rather than use FE methods. I took 8 bootstrap samples of the data, each of size 10000, rotated the images in these samples by 5,10,15,20,-5,-10,-15,-20 degrees respectively, and joined the rotated samples to the training data. This extended it by 80000 observations, and added information through image rotations. Training this model on the extended data set, and predicting on the submission data, I achieved an accuracy score of 0.97571. My last submission (submission 24) was from rerunning the CNN model for 30 epochs with 750 hidden units in layer 5 and 30 units in the output layer, which yielded my best accuracy score of 0.98514.

Ensemble (submissions 20-23)

I used a majority voting ensemble technique to combine predictions from some of my best performing models. I wanted to use different base models with a variety of feature engineering methods, to achieve a low covariance between base models in order to get a good bias-variance-covariance trade-off. I used the following eight base models: Random Forest (RF) + No FE, RF + blurring, SVM + blurring, SVM + HOG, k-NN + blurring, k-NN + PCA, k-NN + HOG, CNN + extended data.

This ensemble model increased my performance on the submission data to 0.98043 accuracy. This is only a 0.00472 increase to my CNN performance, so perhaps the covariance between the base models might be higher than I expected. To check that using only some of my best performing models was a good idea, I ran a second similar ensemble to combine predictions from 16 of my previous submissions, but as expected this gave me a lower accuracy of 0.97414. I then tuned the parameters for my base models and reran the ensemble, which increased the accuracy score to 0.98100. Finally, I decided to remove the SVM+blurring from this ensemble and replaced it with KNN+Edges to add more diversity between base models. This yielded an accuracy score of 0.98200.



The progression graph illustrates my submissions. As explained above, the first 18 submissions correspond to RFs (1-6), SVMs (7-12), and KNNs (13-18). The next two are for the CNN model (19) and the hard voting ensemble using eight base models (20). Finally, the last four are: the same ensemble as model 20 but using sixteen base models (21), the same ensemble as model 20 but with tuned parameters for the base models (22), the same ensemble as model 22 but with KNN+Edges instead of SVM+Blurring (23), and lastly - the most accurate CNN model (24). This shows how my submissions varied a lot as I was trying out different models, but then levels out as I implement CNNs and ensembles to increase my accuracy.

4. Conclusions

I am pleased to have reached an accuracy score of 98.51% using my chosen methods and feature engineering techniques, and I think that several fundamental machine learning concepts were key to helping me to achieve this. Firstly, it was important for me to explore the data beforehand to compare images of different classes, check for imbalanced distributions and get ideas of different models and feature engineering techniques which I could use for this challenge. This enabled me to choose feature engineering methods which were significantly different, in order to help me better understand which features work well with the different models. Choosing features which would diversify my models was also important when trying to reduce the covariance between base models in my ensembles.

A challenge when trying to compare so many different combinations of models and features was the huge number of parameters which needed to be carefully chosen. Exploring the data and its different features helped me to decide on sensible parameters for both my feature engineering methods and my models (which is obviously crucial, as it is impractical and time consuming to simply try to tune parameters when dealing with so many parameter combinations). Although I had decided on parameters which I thought were sensible and would give a good trade-off between performance and execution speed, I also used 10 fold cross-validation on my training data when running each of my models (computing the mean and standard deviation of accuracy scores across folds) to check that I was not overfitting, and to reassure myself that my parameter decisions were sensible.

However, due to the size of the dataset and the number of models I was comparing, performing 10-fold cross-validation on each one was computationally expensive. Since my data exploration suggested that there is no obvious class imbalance in the dataset, and so random bootstrap samples should still contain an approximately equal number of images of each class, I decided to take a bootstrap sample of 20,000 observations and perform the cross-validation on this sample rather than on the full 42,000 observations in the training set. I used a fixed random state so that the same set is used for each model to allow for a better comparison of models. This technique worked well to speed up my pre-processing and training, and the accuracy scores from my submissions were generally very close to the cross-validation scores I obtained during training (but I of course trained my models on the entire training data to predict on the submission data).

When choosing model parameters, it was also important for me to consider the bias-variance-covariance decompositions of each model in order to understand how changing these parameters would change their likeliness to under-fit (high bias) or over-fit (high variance) on the training data. For example, a random forest trains deep decision trees (with high variance and low bias) on bootstrapped samples of the data and its attributes to over-fit diverse models (which therefore have low covariance), and then averages the predictions across base models to reduce the variance. This knowledge was important in helping me choose the number of trees to use and in understanding that it was best to leave tree depth to its maximum for each tree. It also helped me to understand how this type of model tends to work best on data with high dimension, and hence why it didn't perform as well when using only 150 principal components. Moreover, increasing the number of neighbours used in KNN will increase the bias and decrease the variance, whilst increasing the number of hidden units in neural networks or increasing the relaxation parameter C in SVMs will increase the variance and decrease the bias of the corresponding models. This knowledge was useful in trying to estimate the optimal parameters.

If I were to continue competing in this Kaggle competition, I would look at probabilistic classifiers and ways to further diversify the base models in my ensemble methods, as using a simple ensemble method enabled me to achieve a high accuracy score. I would also like to combine different types of ensemble models to see if I could further improve my performance, and beat my top CNN score.

5. Appendix

Data exploration code

I wrote the code below to carry the first part of my data exploration, as explained in section 2.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
#Read in MNIST digits data
Digits_train_data = pd.read_csv("train.csv",header = 0)
Digits_test_data = pd.read_csv("test.csv",header = 0)
#compute number of image of each class
class_counts = pd.DataFrame(pd.value_counts(Digits_train_data['label']))
class_counts = class_counts.transpose()
#create variables to store the subsets of the training data (by label)
labels = [0,1,2,3,4,5,6,7,8,9]
subsets = {}
subset_means = {}
subset_stds = {}
subset_sums = {}
mean_of_sums = [0]*10
std_of_sums = [0]*10

for i in labels:
    #add a new object into train_subsets, named i, where i is the label.
    #then, store the subset of the data for the corresponding label into this object
    subsets[i] = Digits_train_data[Digits_train_data['label'] == i].copy()
    del subsets[i]['label']
    #for each label, compute the array of means and standard deviations for each pixel
    subset_means[i] = np.asarray(np.mean(subsets[i]))
    subset_stds[i] = np.asarray(np.std(subsets[i]))
    #resize the arrays to store pixel values in a matrix
    #(if the array has length 784, we will get a 28x28 matrix)
    matrix_size = int(np.sqrt(len(subset_means[i])))
    subset_means[i] = np.resize(subset_means[i],(matrix_size,matrix_size))
    subset_stds[i] = np.resize(subset_stds[i],(matrix_size,matrix_size))
    #plot the pixel means for each class
    plt.figure()
    imgplot = plt.imshow(subset_means[i])
    #plot the pixel standard deviations for each class
    plt.figure()
    imgplot = plt.imshow(subset_stds[i])
    #compute the pixel sums for each image
    subset_sums[i] = np.asarray(np.sum(subsets[i].transpose()))
    #compute the mean and variance of pixel sums for each class
    mean_of_sums[i] = np.mean(subset_sums[i])
    std_of_sums[i] = np.std(subset_sums[i])

#plot the average (mean) sum of pixel values for each class
plt.figure()
plt.title('Mean sum of pixel values for each class')
plt.plot(labels,mean_of_sums)
plt.xlabel('class')
plt.ylabel('mean of pixel sums')
#plot the variance of sums of pixel values for each class
plt.figure()
plt.title('Standard deviation of sums of pixel values for each class')
plt.plot(labels,std_of_sums)
plt.xlabel('class')
plt.ylabel('std of pixel sums')
```

I wrote the code below to plot my PCA plots as explained in the relevant section. For the standardized PCA case, I simply standardized my data before applying PCA.

```
import pandas as pd
import numpy as np
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
#Read in MNIST digits training data
Digits_train_data = pd.read_csv("train.csv",header = 0)
#make a copy of the data and retrive the label
X = Digits_train_data.copy()
t = X['label']
del X['label']
#compute the number of attributes
num_attr = X.shape[1]
#initialise arrays to store k and explained variance ratios for the first
explained_variance = np.asarray([0.0]*num_attr)
k = np.asarray([0]*num_attr)
#initialise and fit the pca model
pca = PCA(n_components = num_attr)
pca.fit(X)
#store the explained variance ratios for each principal component
explained_variance_ratios = pca.explained_variance_ratio_
for i in range(num_attr):
    #compute the ratio of explained variance for the first i+1 components
    explained_variance[i] = np.sum(explained_variance_ratios[0:i+1])
    #store k=i+1
    k[i] = i+1
#plot the explained variance ratios over k
plt.figure()
plt.plot(k,explained_variance_ratios)
plt.title("Explained variance ratio of kth PCs")
plt.xlabel("K")
plt.ylabel("Explained Variance")
#plot the cumulative explained variance ratios over k
plt.figure()
plt.plot(k,explained_variance)
plt.title("Explained variance ratio of first K PCs")
plt.xlabel("K")
plt.ylabel("Cumulative explained variance")
#compute principal components
X_PCs = pd.DataFrame(pca.transform(X))
X_PCs = pd.concat([t,X_PCs], axis=1)
colours = [0]*len(X)
#the colours to use - the last three are dark blue, purple and grey
colour_names = ['red','blue','black','green','cyan','magenta','yellow',
                '#0000A0','#800080','#808080']
#create an array of colours - 1 for each class
for i in range(len(X)):
    for j in range(10):
        #check which digit the label corresponds to and assign the colour
        if (X_PCs.iloc[i,0] == j):
            colours[i] = colour_names[j]
#plot the first two principal components, coloured by class
plt.figure()
plt.title('Principal Components plot')
plt.xlabel('PC1')
plt.ylabel('PC2')
plt.scatter(X_PCs[0],X_PCs[1],color=colours)
```

My scripts which produce the figures used to demonstrate Gaussian Blurring, Sobel Edge detection and Histogram of Oriented Gradients use the feature engineering methods below and uses the plt.imshow function to plot the filtered images (after resizing them to a 28x28 matrix).

Feature engineering code

I wrote the function below to pre-process my data before training models (extracting the various features).

```
#function to carry out feature engineering methods on input data and return these features
def feature_engineering(data, flag, pca_fit_data = None, pca_test_flag = False,  n_pca = 150,
                        std_blur = 1.0, pixels_per_cell_hog = (4,4), orientations_hog = 5):
    X = data.copy()
    #compute the number of pixel rows/columns in the image
    image_size = int(np.sqrt(X.shape[1]))

    #method to replace each image in the dataset with the gaussian blurred version of itself
    if (flag == 'gaussian blur'):
        for i in X.index:
            #retrive the ith observation
            obs_unfiltered = np.array(X.iloc[i], dtype=np.float64)
            #resize the current observation into a matrix
            obs_unfiltered = np.resize(obs_unfiltered, new_shape = (image_size,image_size))
            #compute the filtered image
            obs_filtered = gaussian_filter(obs_unfiltered, std_blur)
            #reshape the filtered image into an array
            obs_filtered = np.resize(obs_filtered, new_shape = (image_size*image_size))
            #store the filtered image back into the data frame
            X.iloc[i] = obs_filtered

    elif (flag == 'sobel edge detection'):
        for i in X.index:
            #retrive the ith observation
            obs_unfiltered = np.array(X.iloc[i], dtype=np.float64)
            #resize the current observation into a matrix
            obs_unfiltered = np.resize(obs_unfiltered, new_shape = (image_size,image_size))
            #compute the filtered image
            obs_filtered = sobel(obs_unfiltered)
            #reshape the filtered image into an array
            obs_filtered = np.resize(obs_filtered, new_shape = (image_size*image_size))
            #store the filtered image back into the data frame
            X.iloc[i] = obs_filtered
```

```
    elif (flag == 'principal components'):
        #if applying pca to training data
        if (pca_test_flag == False):
            #initialise a pca model
            pca = PCA(n_components=n_pca)
            #fit the model and compute PCs
            X = pd.DataFrame(pca.fit_transform(X))
        #if applying pca to test data, need to fit model on training data
        else:
            #initialise a pca model
            pca = PCA(n_components=n_pca)
            #fit the model on training data and compute PCs on test data
            pca.fit(pca_fit_data)
            X = pd.DataFrame(pca.transform(X))

    elif (flag == 'hog features'):
        for i in X.index:
            #retrive the ith observation
            obs_unfiltered = np.array(X.iloc[i], dtype=np.float64)
            #resize the current observation into a matrix
            obs_unfiltered = np.resize(obs_unfiltered, new_shape = (image_size,image_size))
            #compute the filtered image
            obs_filtered_features, obs_filtered_image = hog(obs_unfiltered, orientations =
                orientations_hog, pixels_per_cell=pixels_per_cell_hog, visualise = True)
            #reshape the filtered image into an array
            obs_filtered = np.resize(obs_filtered_image, new_shape = (image_size*image_size))
            #store the filtered image back into the data frame
            X.iloc[i] = obs_filtered

    else:
        return 'Error: feature engineering method not found'

    return X
```

```
import numpy as np
import pandas as pd
from skimage.filters import gaussian_filter
from skimage.filters import sobel
from sklearn.decomposition import PCA
from skimage.feature import hog
from sklearn.preprocessing import StandardScaler
```

The code below shows my function which I modified when standardizing the data before applying PCA.

```
    elif (flag == 'principal components'):
        std_scaler = StandardScaler()
        #if applying pca to training data
        if (pca_test_flag == False):
            X = pd.DataFrame(std_scaler.fit_transform(X))
            #initialise a pca model
            pca = PCA(n_components=n_pca)
            #fit the model and compute PCs
            X = pd.DataFrame(pca.fit_transform(X))
        #if applying pca to test data, need to fit model on training data
        else:
            std_scaler.fit(pca_fit_data)
            X = pd.DataFrame(std_scaler.transform(X))
            pca_fit_data = pd.DataFrame(std_scaler.transform(pca_fit_data))
            #initialise a pca model
            pca = PCA(n_components=n_pca)
            #fit the model on training data and compute PCs on test data
            pca.fit(pca_fit_data)
            X = pd.DataFrame(pca.transform(X))
```

Random Forest code

I wrote the code below to perform 10-fold CV on a bootstrap sample of my data using a random forest with my various feature engineering methods. This code is the same for SVMs and KNNs, just with different models trained.

```
#Read in MNIST digits data
Digits_train_data = pd.read_csv("train.csv",header = 0)
Digits_test_data = pd.read_csv("test.csv",header = 0)
#take a copy of the training data
X_all = Digits_train_data.copy()
#take a sample of the training data to use for CV
X_sample = X_all.sample(n=20000, replace = True, random_state = 1)
X = X_sample.copy()
#reset index for the sample
X.index = range(len(X))
#separate labels from data in training set
t = X['label']
del X['label']
#Transform data using one of the feature engineering methods
X = feature_engineering(X, flag = 'gaussian blur')
#X = feature_engineering(X, flag = 'sobel edge detection')
#X = feature_engineering(X, flag = 'hog features')

#initialise folds for 10 fold CV
folds = KFold(len(X), n_folds=10)
#create array to store accuracy scores
accuracy_scores = [0]*10
#initialise counter for iterating in the scores array
i = 0
#fit and score model for each CV fold
for train_index, test_index in folds:
    #for the currect fold, retrieve training data and preprocess it
    X_train_unprocessed = X.iloc[train_index]
    X_train = X_train_unprocessed
    #X_train = feature_engineering(X_train_unprocessed, flag = 'principal components')
    #retrieve training target
    t_train = t.iloc[train_index]
    #retrive testing data and preprocess it
    X_test_unprocessed = X.iloc[test_index]
    X_test = X_test_unprocessed
    #X_test = feature_engineering(X.iloc[test_index], flag = 'principal components',
    #                             pca_fit_data = X_train_unprocessed, pca_test_flag = True)
    #retrive test target
    t_test = t.iloc[test_index]

    #initialise model
    random_forest = RandomForestClassifier(criterion = "entropy")
    #fit model
    random_forest.fit(X_train, t_train)
    #predict on test data
    t_predict = random_forest.predict(X_test)
    #compute accuracy of current fold and store in array
    accuracy_scores[i] = accuracy_score(t_test,t_predict)
    #increment counter
    i += 1

#compute mean and std of accuracy score
mean_accuracy = np.mean(accuracy_scores)
std_accuracy = np.std(accuracy_scores)
```

I wrote the code below to apply the various feature engineering methods on the training and testing data, and then train a random forest and predict on the submission data.

```
#take a copy of the testing data
X_testing_data = Digits_test_data.copy()
#reset the training data as the whole dataset
X = X_all
#separate labels from data in training set
t = X['label']
del X['label']
#preprocess submission data
#X_processed = X
#X_testing_data_processed = X_testing_data
X_processed = feature_engineering(X, flag = 'gaussian blur')
X_testing_data_processed = feature_engineering(X_testing_data, flag = 'gaussian blur')
#X_processed = feature_engineering(X, flag = 'sobel edge detection')
#X_testing_data_processed = feature_engineering(X_testing_data, flag = 'sobel edge detection')
#X_processed = feature_engineering(X, flag = 'hog features')
#X_testing_data_processed = feature_engineering(X_testing_data, flag = 'hog features')
#X_processed = feature_engineering(X, flag = 'principal components')
#X_testing_data_processed = feature_engineering(X_testing_data, flag = 'principal components',
#                                               pca_fit_data = X, pca_test_flag = True)

#initialise a model for submission
random_forest_sub = RandomForestClassifier(criterion="entropy")
#fit model on training dataset
random_forest_sub.fit(X_processed,t)
#compute predictions on submission data
t_predict_sub = pd.DataFrame(random_forest_sub.predict(X_testing_data_processed))

#format and write the predictions to a csv file
t_predict_sub.index += 1
t_predict_sub.index.names = ['ImageId']
t_predict_sub.columns = ['Label']
t_predict_sub.to_csv(path_or_buf = 'results.csv')
```

Support Vector Machine code

I wrote the code below to apply 10-fold CV to the bootstrap sample of my training data using a SVM (as explained above, the only difference between the scripts for RF, SVM and KNN models is the section which fits the model and predicts (lines 126-131).

```
#initialise folds for 10 fold CV
folds = KFold(len(X), n_folds=10)
#create array to store accuracy scores
accuracy_scores = [0]*10
#initialise counter for iterating in the scores array
i = 0
#fit and score model for each CV fold
for train_index, test_index in folds:
    #for the currect fold, retrieve training data and preprocess it
    X_train_unprocessed = X.iloc[train_index]
    #X_train = feature_engineering(X_train_unprocessed, flag = 'principal components')
    X_train = X_train_unprocessed
    #retrieve training target
    t_train = t.iloc[train_index]
    #retrive testing data and preprocess it
    X_test_unprocessed = X.iloc[test_index]
    #X_test = feature_engineering(X.iloc[test_index], flag = 'principal components',
    #                             pca_fit_data = X_train_unprocessed, pca_test_flag = True)
    X_test = X_test_unprocessed
    #retrive test target
    t_test = t.iloc[test_index]

    #initialise model
    SVM = LinearSVC()
    #fit model
    SVM.fit(X_train, t_train)
    #predict on test data
    t_predict = SVM.predict(X_test)
    #compute accuracy of current fold and store in array
    accuracy_scores[i] = accuracy_score(t_test,t_predict)
    #increment counter
    i += 1

    print i, '\n'
```

Like with Random Forests and KNN, I wrote the code below to fit an SVM model to the training data and predict on the submission data.

```
#initialise a model for submission
SVM_sub = LinearSVC()
#fit model on training dataset
SVM_sub.fit(X_processed,t)
#compute predictions on submission data
t_predict_sub = pd.DataFrame(SVM_sub.predict(X_testing_data_processed))
#format and write the predictions to a csv file
t_predict_sub.index += 1
t_predict_sub.index.names = ['ImageId']
t_predict_sub.columns = ['Label']
t_predict_sub.to_csv(path_or_buf = 'results.csv')
```

K-Nearest Neighbours code

10-fold CV part of my KNN implementation.

```
#initialise folds for 10 fold CV
folds = KFold(len(X), n_folds=10)
#create array to store accuracy scores
accuracy_scores = [0]*10
#initialise counter for iterating in the scores array
i = 0
#fit and score model for each CV fold
for train_index, test_index in folds:
    #for the currect fold, retrieve training data and preprocess it
    X_train_unprocessed = X.iloc[train_index]
    #X_train = feature_engineering(X_train_unprocessed, flag = 'principal components')
    X_train = X_train_unprocessed
    #retrieve training target
    t_train = t.iloc[train_index]
    #retrive testing data and preprocess it
    X_test_unprocessed = X.iloc[test_index]
    #X_test = feature_engineering(X.iloc[test_index], flag = 'principal components',
    #                             pca_fit_data = X_train_unprocessed, pca_test_flag = True)
    X_test = X_test_unprocessed
    #retrive test target
    t_test = t.iloc[test_index]

    #initialise model
    knn = KNeighborsClassifier(n_neighbors=3)
    #fit model
    knn.fit(X_train, t_train)
    #predict on test data
    t_predict = knn.predict(X_test)
    #compute accuracy of current fold and store in array
    accuracy_scores[i] = accuracy_score(t_test,t_predict)
    #increment counter
    i += 1
```

Fitting a model on my training data and predicting on the submission data.

```
#initialise a model for submission
knn_sub = KNeighborsClassifier(n_neighbors=3)
#fit model on training dataset
knn_sub.fit(X_processed,t)
#compute predictions on submission data
t_predict_sub = pd.DataFrame(knn_sub.predict(X_testing_data_processed))
#format and write the predictions to a csv file
t_predict_sub.index += 1
t_predict_sub.index.names = ['ImageId']
t_predict_sub.columns = ['Label']
t_predict_sub.to_csv(path_or_buf = 'results.csv')
```

Convolutional Neural Networks code

I wrote the function below to add rotated images from bootstrapped samples to the input data.

```
#function to bootstrap sample n images from the data, rotate them by the given
#angle, and add them to the data
def add_rotated_images(data, n_images, angle):
    X = data.copy()
    #take a sample of the training data to rotate and add to data
    X_sample = X.sample(n=n_images, replace = True).copy()
    #reset index for the sample
    X_sample.index = range(len(X_sample))
    #separate labels from data in training set
    t_sample = X_sample['label']
    del X_sample['label']
    #compute the number of pixel rows/columns in the image
    image_size = int(np.sqrt(X.shape[1]))
    #rotate each image in the sample by the required angle
    for i in X_sample.index:
        #retrive the ith observation
        obs_unfiltered = np.array(X_sample.iloc[i], dtype=np.float64)
        #resize the current observation into a matrix
        obs_unfiltered = np.resize(obs_unfiltered, new_shape = (image_size,image_size))
        #compute the rotated image
        obs_filtered = rotate(image = obs_unfiltered, angle = angle)
        #reshape the rotated image into an array
        obs_filtered = np.resize(obs_filtered, new_shape = (image_size*image_size))
        #store the rotated image back into the data frame
        X_sample.iloc[i] = obs_filtered
    X_sample = pd.concat([t_sample,X_sample],axis=1)
    #make the index of the sample continue from that of X
    X_sample.index += len(X)
    #join the sample onto the dataset
    X2 = pd.concat([X,X_sample], axis=0)
    return X2
```

```
import numpy as np
import pandas as pd
import lasagne
from lasagne import layers
from nolearn.lasagne import NeuralNet
from skimage.transform import rotate
```

I use the convolutional neural network implementation below, which is a modified version of one I found on Kaggle. I got the code for the convolutional neural net from this kaggle script:
https://www.kaggle.com/amandus/digit-recognizer/py-digitrecognizer-cnn/code

```
def CNN(n_epochs):
    net1 = NeuralNet(
        layers=[
        ('input', layers.InputLayer),       #input layer
        ('conv1', layers.Conv2DLayer),      #Convolutional layer
        ('pool1', layers.MaxPool2DLayer),   #max pooling layer for sub-sampling, reduces overfitting
        ('conv2', layers.Conv2DLayer),      #another convolutional layer
        ('hidden3', layers.DenseLayer),     #fully connected layer
        ('dropout1', layers.DropoutLayer),  #dropout layer to reduce overfitting
        ('output', layers.DenseLayer),      #fully connected output layer
        ],

    input_shape=(None, 1, 28, 28),
    conv1_num_filters=10,
    conv1_filter_size=(3, 3),
    conv1_nonlinearity=lasagne.nonlinearities.rectify,

    pool1_pool_size=(2, 2),

    conv2_num_filters=15,
    conv2_filter_size=(2, 2),
    conv2_nonlinearity=lasagne.nonlinearities.rectify,

    hidden3_num_units=200,

    dropout1_p=0.5,

    output_num_units=10,
    output_nonlinearity=lasagne.nonlinearities.softmax,

    update_learning_rate=0.0005,
    update_momentum=0.9,

    max_epochs=n_epochs,
    verbose=1,
    )
    return net1
```

I wrote the code below to extend the training data using rotated images, train the CNN for 15 epochs using the implementation above, and then used this model to predict on the submission data.

```
#Read in MNIST digits data
Digits_train_data = pd.read_csv("train.csv",header = 0)
Digits_test_data = pd.read_csv("test.csv",header = 0)
X = Digits_train_data.copy()
#take bootsrap samples of the data, rotate the images at different angles,
#and add the samples back onto the dataset
X_processed = add_rotated_images(data = X, n_images = 10000, angle = 5)
X_processed = add_rotated_images(data = X_processed, n_images = 10000, angle = 10)
X_processed = add_rotated_images(data = X_processed, n_images = 10000, angle = 15)
X_processed = add_rotated_images(data = X_processed, n_images = 10000, angle = 20)
X_processed = add_rotated_images(data = X_processed, n_images = 10000, angle = -5)
X_processed = add_rotated_images(data = X_processed, n_images = 10000, angle = -10)
X_processed = add_rotated_images(data = X_processed, n_images = 10000, angle = -15)
X_processed = add_rotated_images(data = X_processed, n_images = 10000, angle = -20)
#separate labels from data in training set
t = X_processed['label']
del X_processed['label']
# Read competition data files:
t_train = t.values.ravel()
X_train = X_processed.values
X_test = Digits_test_data.copy().values
# convert to array, specify data type, and reshape
t_train = t_train.astype(np.uint8)
X_train = np.array(X_train).reshape((-1, 1, 28, 28)).astype(np.uint8)
X_test = np.array(X_test).reshape((-1, 1, 28, 28)).astype(np.uint8)

# train the CNN model for 15 epochs
cnn = CNN(15).fit(X_train,t_train)

# use the NN model to classify test data
t_predict_sub = pd.DataFrame(cnn.predict(X_test))
#format and write the predictions to a csv file
t_predict_sub.index += 1
t_predict_sub.index.names = ['ImageId']
t_predict_sub.columns = ['Label']
t_predict_sub.to_csv(path_or_buf = 'results.csv')
```

Ensemble model code

I wrote the code below to implement my hard voting ensemble, which reads in several files corresponding to submitted predictions from various models, concatenates them into a pandas Data Frame, computes the mode prediction for each image in the submission data, and outputs this in the required format.

```
import numpy as np
import pandas as pd
from scipy.stats import mode

#Read in the saved results from the various submissions
t_RF_NoFE = pd.read_csv("RF - No FE.csv",header = 0)['Label']
t_RF_Blurring = pd.read_csv("RF - Blurring.csv",header = 0)['Label']
t_SVM_Blurring = pd.read_csv("SVM - Blurring.csv",header = 0)['Label']
t_SVM_HOG = pd.read_csv("SVM - HOG.csv",header = 0)['Label']
t_KNN_Blurring = pd.read_csv("KNN - Blurring.csv",header = 0)['Label']
t_KNN_PCA = pd.read_csv("KNN - PCA.csv",header = 0)['Label']
t_KNN_HOG = pd.read_csv("KNN - HOG.csv",header = 0)['Label']
t_CNN_extended_data = pd.read_csv("CNN - Extended Dataset.csv",header = 0)['Label']

#concatenate predictions from each base model
t_predict_concat = pd.concat([t_RF_NoFE,t_RF_Blurring,
                              t_SVM_Blurring,t_SVM_HOG,
                              t_KNN_Blurring,t_KNN_PCA,t_KNN_HOG,
                              t_CNN_extended_data],axis=1)

#compute the final predictions as the modes of predictions from each classifier
target_modes = [0]*len(t_predict_concat)
for i in range(len(t_predict_concat)):
    target_modes[i] = mode(np.asarray(t_predict_concat.iloc[i]))[0][0]

#format and write the predictions to a csv file
t_predict_sub = pd.DataFrame(data={'Label': target_modes})
t_predict_sub.index += 1
t_predict_sub.index.names = ['ImageId']
t_predict_sub.to_csv(path_or_buf = 'results.csv')
```