Machine Learning

Assignment 4

Shadab Zafar
2017MCS2076

Q1: Fixed Algorithms

A.) **K Means**

i.) Assign Majority Label to Clusters

The data was normalized (zero mean, unit variance) before running K Means clustering.

ii.) Find Test Labels & Accuracies

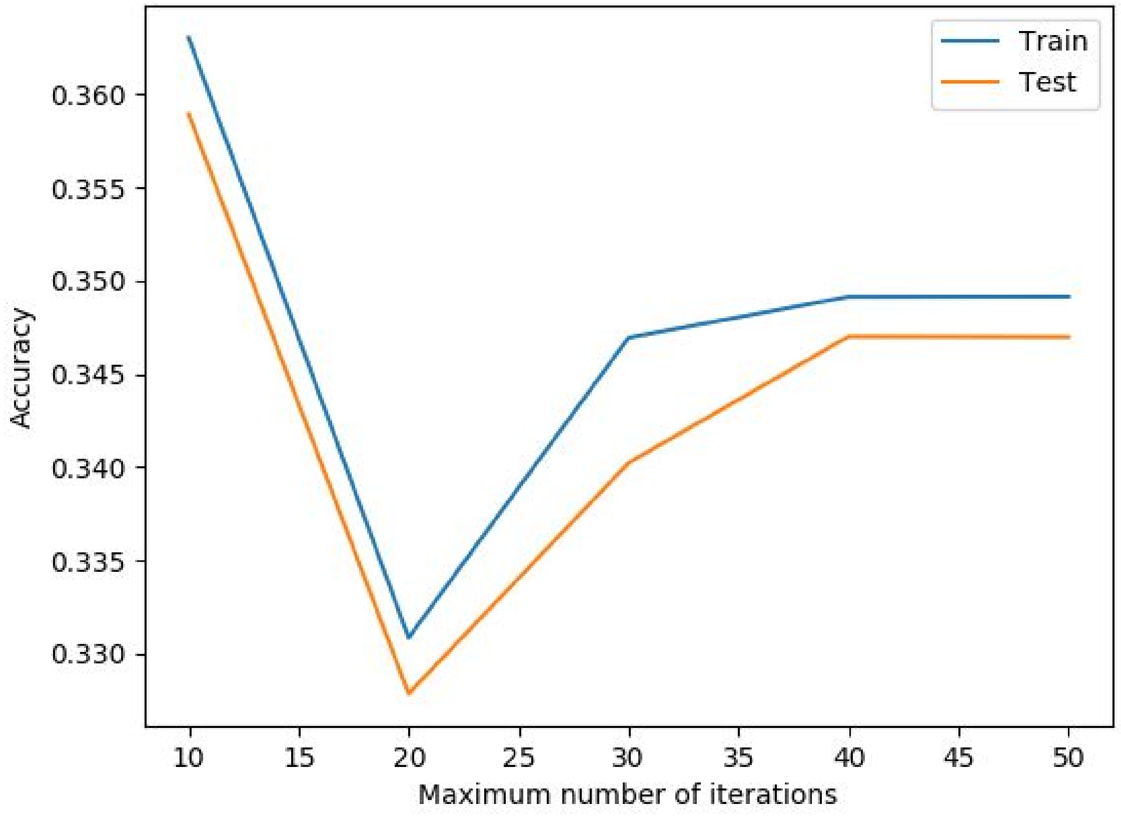
KMeans Parameters: n_init=10, n_clusters=20, max_iter=300

Training Accuracy: 34.894% Testing Accuracy: 34.755%

iii.) Varying max_iter parameter

max_iter	Training Accuracy	Testing Accuracy
10	0.36303	0.35893
20	0.33084	0.32784
30	0.34694	0.34022
40	0.34913	0.34700
50	0.34914	0.34698
300 (default)	0.34894	0.34755

iv.) K Means Accuracy Plot



B.) PCA + SVM

Before applying PCA, there were two different ways of scaling the data. One was to just divide all the entries by 255 (effectively bringing all entries between 0 & 1) or we could normalize the data (zero mean-unit variance.)

I tried out both the ways, and normalization outperformed 0-1 scaling.

It takes around 50 minutes for the Normalize-PCA-SVM pipeline to train (with C=1.0) resulting in a training accuracy of **0.69912** and a testing accuracy of **0.69345**

Whereas with 0-1 scaling, the accuracy was **0.65662**.

For finding out the optimal C value, I split the training data into two parts, 70% of which was used for training while the other 30% was used for validation.

I tried out C values: 0.001, 0.01, 0.1, 1, 10, 100 out of which, C=0.001 gave the best CV score of **0.69404**

Scikit-Learn's default SVC classifier uses the OneVsRest strategy, so OneVsOne needs to be set via parameters.

Note:

For most of these algorithms I used Google Colab for training since there I was able to use GPUs and parallelize on different processors (where applicable.) The machines there also have more RAM than mine, which certainly helped.

C.) Neural Network (PyTorch)

I used PyTorch for implementing the neural network. The architecture was:

```
Net(
  (l1): Linear(in_features=784, out_features=1000, bias=True)
  (l2): Linear(in_features=1000, out_features=20, bias=True)
)
```

The loss criterion used was Negative Log Loss, and the Adam optimizer.

I normalized the data (zero mean and unit variance) before training the net.

To test out the implementation without having to submit the labels again and again, I used a validation set (30%) and during training, validation accuracy was calculated at each epoch.

Initially, I tried keeping the number of hidden units low (less than 100) but that was resulting in lower accuracy (0.7064) but increasing the hidden units also resulted in improved accuracy, with 500 units I got a testing accuracy of 0.78410 while with 1000 units it was 0.7888.

I ran an exhaustive grid search to find a limit on epochs, the best learning rate etc. and the parameters that yielded the best result were.

Hidden units: 1000, Learning Rate: 0.0005, Epochs: 28, Accuracy: **0.7921**

D.) Convolution Neural Network (PyTorch)

I continued with PyTorch for CNN too. The Architecture was:

```
ConvNet(
  (conv): Conv2d(1, 64, kernel_size=(5, 5), stride=(1, 1))
  (mp): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), dilation=(1, 1))

  (fc1): Linear(in_features=9216, out_features=512, bias=True)
  (fc2): Linear(in_features=512, out_features=20, bias=True)
)
```

Using ReLU at hidden layers, and Log Softmax at output layer. The loss criterion used was Negative Log Loss, and the Adam optimizer.

On my machine, one epoch of this network takes around 4 minutes to run, and the network takes around 25 epochs to converge to a good dev set accuracy. But I used Google Colab for training this network (and running grid search) as GPUs were available there.

Grid search was run on various parameters, and the best parameters were:

Epochs: 50, Learning Rate: 0.001, Batch Size: 256
Channels: 64, Kernel Size: 5 x 5, Max Pool Size: 2 x 2, Hidden Layers: 512

Training Accuracy: **0.9436** Testing Accuracy: **0.85422**

E.) Comparison of Algorithms

K Means performed very poorly out of all these algorithms, which was expected, since it is an unsupervised algorithm that does not take into account class information when building the model. It takes around 3-4 minutes for 10 iterations of KMeans to run.

PCA combined with SVM gave fairly decent results (because we were using a linear kernel) but it took around an hour to complete.

A standard neural net (non convolutional) was able to offer better accuracy than SVM (~10% more) while the running time was similar. The key challenge here was to find the best set of parameters (using cross validation.) Before I was trying out small number of hidden units and spent a considerable amount of time searching for a good value less than 100 but when I later increased it to 1000, the accuracy improved.

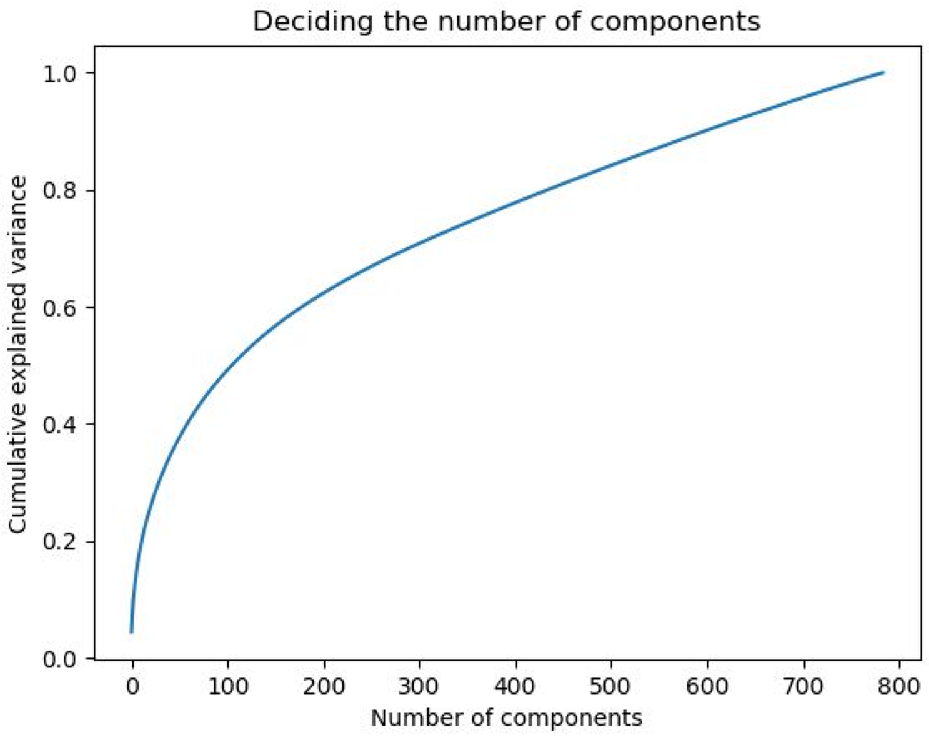
CNNs gave the best accuracy, among all mentioned algorithms, since the training data was images, this sort of result was expected as CNNs are able to utilise feature locality. Like NNs, training CNNs takes a lot of patience, since there’s a large number of hyperparameters to be checked.

Part 2 - Kaggle Competition (misc.)

Apart from the algorithms listed above, I applied two basic classifiers: Logistic Regression & XGBoost which resulted in an accuracy around 0.63.

I also built an majority based ensemble method, which used the final kaggle data labels to aggregate labels of multiple models. This did help me gain around 1% accuracy improvements.

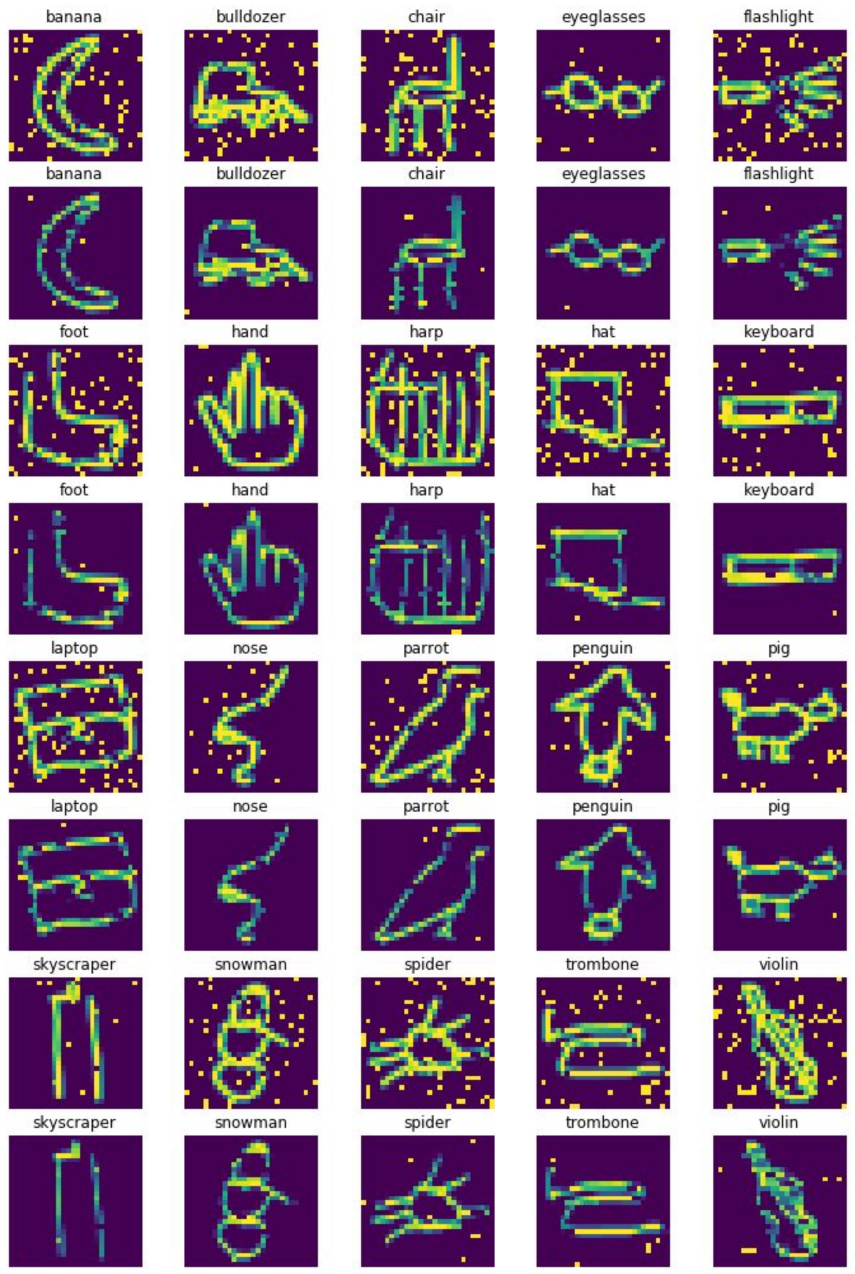
To find a good number of components to scale down the data to, I decided to plot the components vs explained variance curve. Which showed that 50 was too low a number.



With this insight, I tried PCA (200 components) + SVM (with RBF Kernel) resulting in a training accuracy of **0.99068** and a testing accuracy of **0.83005** which was my first good score.

Data Preprocessing

After plotting images in the dataset, I noticed that there was a lot of noisy pixels, which felt as if they were knowingly added. To remove these pixels, I used OpenCV morphological transforms (specifically - erosion.)



Best Performing Model

I switched from PyTorch to Keras, as I found the API cleaner. I tried various different architectures: Alexnet, VGG 13,VGG 19. Some layers were removed (while the others modified) to get them to work with our dataset. The models were trained with SGD using a batch size of 128.

Publick link to my model and label files:
https://drive.google.com/open?id=1yX6gN9Zvxy40KlNOkpbj7U0pEaFTuydw

My best score was: **0.93537** which was obtained by a weighted majority voting of the following models:

Model: **(label test file name)**	**Details:** **(variations etc.)**	**Weight:** **(test accuracy)**
keras_vgg_13_918	Unclean Data, 6 Epochs	0.918
keras_vgg_13_91867	Unclean Data, 7 Epochs	0.91867
keras_vgg_19_small_912	Clean Data, ~12 Epochs	0.91242
keras_vgg_13_909	Unclean Data, 8 Epochs	0.909
keras_alexnet		0.90270

For ensembling, I used the label file generated by a model directly. The weight was assigned as the test accuracy (reported by Kaggle.) We can see how ensembling improves the accuracy as the best individual model was giving 91.8% while the ensemble gives 93.5%

Note: I was not storing the model before, and forgot to note down some of the hyperparameters, so the model that I’ve uploaded are not the exact same as the ones I got the above scores on.

Architectures

VGG 13:

```
Conv2D(64, (3, 3)) -> BatchNormalization() -> Activation('relu')
Conv2D(64, (3, 3)) -> BatchNormalization() -> Activation('relu')
MaxPooling2D((2, 2)) -> Dropout(0.2)

Conv2D(128, (3, 3)) -> BatchNormalization() -> Activation('relu')
Conv2D(128, (3, 3)) -> BatchNormalization() -> Activation('relu')
MaxPooling2D((2, 2)) -> Dropout(0.2)

Conv2D(256, (3, 3)) -> BatchNormalization() -> Activation('relu')
Conv2D(256, (3, 3)) -> BatchNormalization() -> Activation('relu')
MaxPooling2D((2, 2)) -> Dropout(0.2)

Conv2D(512, (3, 3)) -> BatchNormalization() -> Activation('relu')
Conv2D(512, (3, 3)) -> BatchNormalization() -> Activation('relu')
MaxPooling2D((2, 2)) -> Dropout(0.2)

Conv2D(512, (3, 3)) -> BatchNormalization() -> Activation('relu')
Conv2D(512, (3, 3)) -> BatchNormalization() -> Activation('relu')

Flatten()

Dense(20, activation='softmax')
```

VGG 19 (smaller): (no BatchNormalization, dense layers)

Conv2D(64, (3, 3))
Conv2D(64, (3, 3))
MaxPooling2D((2, 2))

Conv2D(128, (3, 3))
Conv2D(128, (3, 3))
MaxPooling2D((2, 2))

Conv2D(256, (3, 3))
Conv2D(256, (3, 3))
Conv2D(256, (3, 3))
Conv2D(256, (3, 3))
MaxPooling2D((2, 2)) -> Dropout(0.3)

Conv2D(512, (3, 3))
Conv2D(512, (3, 3))
Conv2D(512, (3, 3))
Conv2D(512, (3, 3))
MaxPooling2D((2, 2)) -> Dropout(0.3)

Flatten()

Dense(4096) -> Dropout(0.3)
Dense(4096) -> Dropout(0.3)

Dense(20, activation='softmax')

Alexnet:

```
Conv2D(64, (8, 8))
Conv2D(192, (5, 5))
MaxPooling2D((2, 2)) -> Dropout(0.3)

Conv2D(384, (2, 2))
Conv2D(256, (2, 2))
Conv2D(512, (2, 2))

Flatten()

Dense(4096) -> Dropout(0.3)
Dense(4096) -> Dropout(0.5)

Dense(20)
```