Report Assignment 2 INF264

Summary

The project aims to use machine learning to recognize handwritten digits from 0-9 and handwritten letters from a-f, as well as separate out blank inputs. In general, the approach is to use hold-out validation to find good hyperparameters, but in some instances grid searches with cross-validation are also carried out. Using validation data, the best instance of each kind of model was selected, and these best instances were all tested on test data to receive the final scores.

Of the algorithms tested, the selected CNN performed the best on the test data at a weighted f1-score of 0.986. As this algorithm proved reasonably accurate and efficient, I believe that it would be a good choice to apply to the task in real life. The reported score can be seen as a good measure of the algorithm's performance on real-life data, as it was tested on diverse data (with outliers such as very faint characters, or in other ways different from typical instances of handwritten characters).

The code can be run top-down in the Jupyter Notebook for all reported results, but note that running all the code could take a while. However, especially computationally heavy grid searches have been commented out, and the resulting models from these have instead been saved.

Technical report

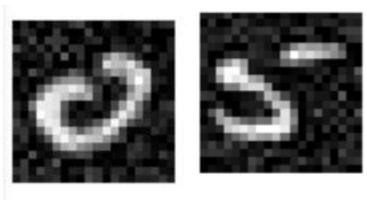
Data exploration

The first steps I took in the project was to have a quick look at the dataset. I noted that it was quite large, with 107802 entries, each with 400 features. This large number of features could prove computationally expensive dependant on the model used. I also checked if there were any missing or NaN values and found this not to be the case. Because of this, I did not see the need to do any significant pre-processing steps.

I also used np.unique(y, return_counts=True) to check out the balance of data in the dataset. Here I found that there was some variation, with some letters having fewer instances in the dataset. Especially the label 14, which corresponds to “E” had a low number of instances with only 1337. This could mean that classifiers could have difficulty detecting this character in particular, as opposed to others.

After looking at an example plot of the data using the provided function, I decided to build out a few functions to look at a few different instances of each label. I used return_images(n) and plot_images(n) for this task.

When plotting out the images, I noted that there was a lot of variation between different instances of the same label. Some characters were not fully “closed”, see here “0” from plot 1 and “5” from plot 2:



The different instances also varied a lot based on line width and slant of the character. Circles drawn on characters which include circles were also often very different from one another.

I next decided to plot some instances of the blank character separately. When plotting these, I found that they were not full black, having some lighter shades mixed in. However, they were still quite consistent and not light enough to likely be confused with any of the other classes. Therefore, I concluded that the blank character would likely not be an issue during classification and moved on to trying classification models on the data.

KNN

The first model that came to mind for classification was KNN, as this was the first one that was taught in the course. Sklearn’s KNN has quite a few hyperparameters that can be chosen, but the most significant one is the choice of *k*. Because of computational demands, I decided not to try any other hyperparameters for this model. I did not expect this kind of model to be best suited for image recognition, but I was curious as to what the results could be, and what I could learn from them.

Here I decided to do an approach where I created a function, multiple_knn, which created a model for every k-value passed in, then computed a weighted f1-score for each model. I settled on using f1 as the dataset was a little unbalanced, which meant that accuracy could be high if the model rarely predicts the label which there is little data of. I wanted to avoid this so that I could attain the best possible results even on labels with less data like “E”.

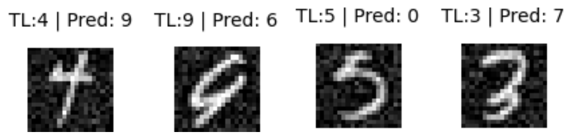
Training with higher k-values was computationally expensive, so I first tried some relatively low k-values (1, 3, 5, 9, 15), hoping they could still give some valuable insight. I then plotted out validation f1 (I initially wanted to plot this out for training as well, but just calculating these values took a long time, so I deemed it not worth the calculation time) and selected the best performing KNN model automatically. The model chosen was with k = 1.

From the plot I carried out, it appeared that validation accuracy steadily decreased for higher values of k. Together with the long training and f1 calculating times, this was a strong indication that KNN was not well suited for the classification task. This also makes sense in the context of the curse of dimensionality – with such a high-dimensional feature space, algorithms that rely on using distance metrics such as KNN often do not perform well. Because of these shortcomings, I decided to not further experiment with KNN.

The final average weighted f1 for the selected model was 0.95. I was surprised at this high performance - it appears that although a model with k=1 could be liable to overfitting, this was not the case here as it generalized reasonably well for the test data.

Next, I figured it was interesting to have a quick look at some misclassified instances before I moved on. I noted the particular difficulty the model had in distinguishing “D” from “0”, often predicting “0” when the true label was “D”. This was a bigger issue than I anticipated, and I was curious to see if the other classifiers performed better in this regard. Mistakes between “8” and “B” was also a common issue. Surprisingly, recall for “E” was quite high, even though this was the label with the lowest amount of data.

The model also had a lot of mistakes on characters which I believe would be easily identifiable by humans, another indication that KNN was not able to capture features in the data accurately. Here are some samples from the notebook:



Random forest

The next model I wanted to try was the random forest classifier. I first wanted to do a grid search to automatically find optimal hyperparameters for this model, but found this to be very computationally expensive. I therefore decided to first experiment with the basic sklearn

RandomForestClassifier, (with default hyperparameters) and then adjust according to performance on the validation data.

After fitting the basic model, I noticed that it performed perfectly on the training data. I therefore thought that it could be overfitting, leading me to the idea of using hyperparameters for pre-pruning to try and improve performance such as max_depth, max_features and max_leaf_nodes.

This was however not a great success, as the values I chose pruned too aggressively, leaving it unable to predict well for any labels. I noted that for “E” in particular precision was 1 but recall was only 0.01, meaning although the model was correct when it predicted “E”, it very rarely predicted “E”. This was another clear sign that the tree was pruning too aggressively.

Following this, I adjusted the hyperparameters to increase max_depth and removed max_leaf_nodes in an attempt to prune less aggressively, but performance on the validation data was still quite poor.

At this point, I all but abandoned random forests, as I could not find a way to improve upon the performance of the base random forest model. However, when I was working on SVC, I discovered the benefits of using sklearn’s PCA. After using PCA, I found that it was no longer infeasible to carry out a grid search, and I utilized a search to test a few different max_depth values for the gini and entropy measures. Unfortunately, this grid search turned out to be unsuccessful in improving validation performance.

Based on the validation results so far, I decided that it would not be useful to experiment further with random forests than this. This kind of model did not seem well suited to the data, and generally performed poorly. The final selected model was the one with max_depth = 50, which achieved an f1-score of 0.937 on the test data. This model had much the same limitations as the KNN model, often predicting “0” when the true label was in fact “D”. As the issues appeared to be similar in nature to the issues with KNN, I will not provide further analysis here.

SVC

When using the right basis function, sklearn can make use of the kernel trick to do computations in the input space. Nevertheless, in an input space with 400 features, there are a lot of computations needed even making use of this. I realized that training such a model was not practical when I initially tried to train an sklearn SVC on the original data. This is when I got the idea of using sklearn’s PCA for feature extraction and thus dimensionality reduction.

My process for this was to first scale the data, then use a PCA model for feature extraction on this data. I first tried to use n_components='mle' and svd_solver='full' to use Minka's MLE to get an appropriate dimensionality. However, this approach barely reduced the feature space, only changing it from 400 to 396.

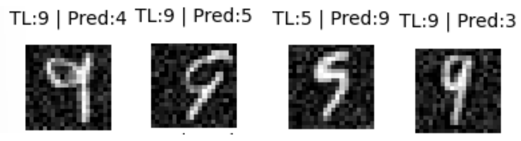
The number one goal at this point was to reduce the feature space such that it would be possible to fit an SVC in a reasonable time. I therefore went for a bit of a "shot in the dark" approach and tried reducing to only 30 components. Although this meant only 0.458 of the variance was retained, the results on validation data were promising for this value, achieving an f1-score of 0.96.

This led me to experiment further with this reduced data, even when much of the variance was lost. I decided to use this to carry out a grid search with cross validation using sklearn's GridSearchCV. As the grid search will use cross validation, I could now split the dataset into only training and test data (as validation data would be separated out from training data for each fold in the CV).

Although the feature extraction process made training times for each model more reasonable, I still felt that I had to limit my search somewhat as each fit took anywhere from 2-15 minutes. Therefore, I decided to focus on the radial basis function kernel with a few different C and gamma values. I also set the number of folds for each to 2 to make the search achievable.

After letting the grid search run for a while, the best estimator was found to be a radial basis function SVC with C=10, gamma=0.01. I saved the model in a file to make sure the model was not lost and performed predictions on the test set using this automatically selected best model. The results were good, achieving 0.968 overall weighted f1-score and 0.97 overall accuracy on the test set.

Interestingly, the model performed perfectly on the digit "1", which none of the earlier models had done. This could be explained by the fact that this is a relatively simple character, which one can imagine can be built with few components. My theory is therefore that the dimensionality reduction from PCA does not affect this character much. Other notable results were that this model also had trouble separating "0" from "D" and "8" from "B", even though it was better at this than the KNN model. The plot of incorrect instances also showed that it had some issues with the number "9", often confusing it with other numbers:



MLP

The next model I wanted to try was sklearn's Multi-Layer Perceptron. Neural networks are well suited for image recognition because of their layered structure and ability to extract patterns/features from image data. I therefore believed that this would be a promising approach. First, I tried training and predicting using the basic MLPClassifier, achieving an average weighted f1-score of 0.9.

Building on this, I noticed that the default parameter for the hidden_layer_sizes in the classifier was only (100,). I reasoned that increasing the deepness of the network by introducing more multiple layers would allow it to better detect features in the data. Next, I trained a model with hidden_layer_sizes(150,100,50) and evaluated this on the validation data. This led to an immediate improvement on the validation data, achieving an f1-score of 0.96.

Next, I tried a few more model variations, introducing early stopping to counteract overfitting. I also tried having more hidden layers and using an adaptive learning rate instead of a constant one. After evaluating all model variations on the validation data, the best was selected to be the model with hidden_layer_size = (150,100,50).

Although the f1-score on test data was slightly worse than for the SVC model, achieving only 0.956, both training time and prediction time was much more efficient. These promising results led me to experiment further with neural networks, moving on to convolutional neural networks.

CNN

After experimenting with sklearn's MLP, I moved on convolutional neural networks using tensorflow/keras. The advantage of using tensorflow was that I was able to introduce convolutional layers into the network. I found it was typical to use either 3x3 or 5x5 convolutional layers, but I figured that 3x3 would be suitable for the small image size of 20x20. The number of filters initially chosen for the convolutional layers were mostly arbitrary based on what I found was common practice. For the first network, I decided to use the stochastic gradient descent optimizer as a starting point.

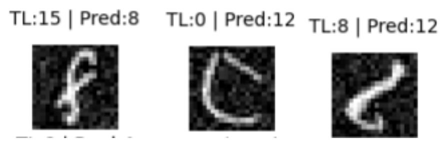
To train the model, I figured that 10 epochs and a batch size of 64 was a reasonable starting point to stop the training process from being too computationally heavy, yet still be able to attain good results. This model achieved a validation accuracy of 0.97.

Next, I wanted to try a model with a higher number of convolutional layers, so I added a layer with 128 filters to the existing architecture. For the training process I decided to adjust the batch size to 128. I was also curious if a network with four convolutional layers would be any good, so I tried this as well, where the last layer had 256 filters, but validation accuracy for this large model did not improve from the model with two convolutional layers.

As having this fourth layer did not improve validation accuracy, I toned down the number of layers, instead opting for a different optimizer (adam). I tried using this optimizer with both two and three layers, collecting accuracy scores on the validation data for these as well.

After testing all these models, I automatically selected the best among them based on accuracy on the validation data. The best model found was model number four, that is the first adam model with two layers. This model yielded an f1-score of 0.984 on the test set, the best so far of the different models tried. I therefore decided to select this model as the final classifier based on its efficiency and the good score.

When checking out a sample of the incorrectly classified characters, some of them seemed to be labelled incorrectly:



For instance, for the first image my classifier has predicted “8”, yet the true label is supposedly 15, “F”. However, as it seemed outside the task description to manually relabel data, I did not do any further analysis or work on this.

Other results of note was that this classifier had little difficulty distinguishing between “8” and “B”. However, there were still 32 times where the predicted label was 13, “D” and the true label was “0”. This was a common problem with the other models, and was what posed the most issues for this final classifier, even though it performed very well on average.

Improvements given time and/or computing resources

Given more computing resources, I would have focused my attentions on either the SVC model or the CNN model. These two both yielded good results, and I believe further experimentation would be fruitful. The KNN and random forest models on the other hand did not seem well-suited for the classification task.

For the SVC, I think it would be interesting to try more hyperparameters and different kernels. The training data could also be adapted to more principal components than the 30 I decided on to keep more of the variance. Additionally, introducing more folds into the cross-validation may help to give more accurate results, making it more likely that we end up picking the best performing model.

There may very well also have been a combination of hyperparameters that would have improved performance for the MLP model, but it would be more interesting to experiment with the convolutional neural network. I felt that my limited knowledge and experience was a limiting factor when trying to find the optimal hyperparameters for this network. Given more time to learn about this, I would perhaps have been able to work more efficiently towards finding the optimal network architecture and hyperparameter values.

Another step I could have taken given more time and resources would be to check through the data, making sure that there were no mislabelled instances. A case could be made for removing outliers as well, although I believe that it could also be argued that outliers could occur in a real-life use case and should therefore be reckoned with.