Classifying digits: Pattern Recognition Assignment

Daniele Di Grandi
7035616

Matthijs Wolters
5983592

ACM Reference Format:
Daniele Di Grandi and Matthijs Wolters. 2020. Classifying digits: Pattern Recognition Assignment. In *Proceedings of (Master Computing Science).* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In this paper we analyze a subset of the famous MNIST dataset, with the objective of automatically recognizing images of handwritten digits, formed by 28x28 pixels, contained in the dataset. The dataset will be fully described in section 2. Various prediction techniques were tried and the results are reflected on and discussed. The algorithms proposed in this paper are divided in two sections: a feature space analysis and a classifier comparison. The feature space analysis describes feature spaces derived from an exploratory analysis of the data. Here we tried to build the predictor variables based on some specific property of the dataset, such as the Ink Feature, the mirror feature and the combined ink + mirror feature. The digit are classiffied with a naïve version of the Logistic Regression, which it produces questionable results, but interesting analysis. The classifier comparison describes models based on some well-known algorithms (Logistic Regression, Support Vector Machines and Feed Forward Neural Networks) where the data were split into training and testing samples and we used all the 784 pixels as features, which produces some interesting and good results, as well as some interesting analysis. We believe that these methods are applicable beyond MNIST, specifically in all the tasks where a number of distinct classes are present and some images, or documents, composed of pixels have to be classified. Though, remaining in the field of images of handwritten characters, an extension of MNIST could be to predict every handwritten character and not only a digit from 0 to 9. Implementations of these algorithms are already in place, in fields like note taking applications (where they also implement a cursive recognition task), analysis of scanned documents, and address recognition for postal services.

2 DATASET

The dataset used in this analysis is a subset of the well known MNIST set of handwritten digits[1]. The original dataset was created for use in research and training of different classifiers for the detection of handwritten digits. The dataset used in this paper contains 42,000 images of digits 0 through 9. Each image is comprised of 28 by 28 pixels, which comes to a total of 784 pixels per digit. Each pixel contains a value between 0 and 255, with 0 being a completely white pixel, 255 being completely black and every value in between a different shade of gray. Within the dataset each image is encoded as a single line of 784 values corresponding to each pixel. Furthermore the first column of every line contains the label of the image encoded by the pixels. Figure 1 shows some examples of digits.

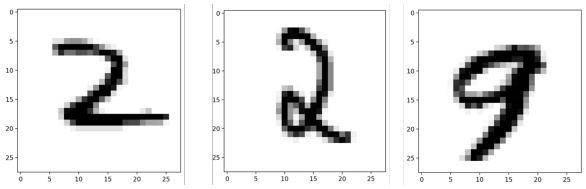

Figure 1: Example of digits in classes 2 and 9.

2.1 Exploratory Analysis

To become familiar with the dataset an initial exploratory analysis was done. The first thing we analysed was the frequency distribution of the classes. Table 2 shows the results. The classes are relatively well distributed in the data, however the class of 1 digits is in the majority. It should be obvious that the sum of the class frequencies equals 42,000, as this is the number of elements in this dataset. The probability distribution gives a clear view of the proportion of classes in the dataset. If we would classify very naively, using a majority class prediction, we can see from this table that we would get an accuracy of 11.15%. This means that the models used to classify this data will probably be better at classifying class '1' digits, more than any other class, because there is more data to train on. One drawback of this is that the models might start overfitting class '1' digits but to combat this we use cross-validation for the tuning and training of the models.

CLASS	FREQUENCY	PROB DISTRIB
0	4132	9.84 %
1	4684	11.15 %
2	4177	9.95 %
3	4351	10.36 %
4	4072	9.70 %
5	3795	9.04 %
6	4137	9.85 %
7	4401	10.48 %
8	4063	9.67 %
9	4188	9.97 %

Figure 2: Frequency and probability distribution of digits.

Another interesting aspect of the data could be the presence of 'useless' variables. In the case of this data, useless variables could be defined as pixels which never assume a value other than 0. In other words these pixels are never touched by ink. If this holds true for any pixel in the whole dataset, we can probably consider that pixel non differential for classification. The result of this analysis can be found in figure 3. We found 76 out of 784 pixels for which this property holds. This means that 9.69% of the data could be 'useless' variables. If each pixel is used as a feature for a classifier, these useless pixels could be removed as features to reduce the amount

Master Computing Science, Pattern Recognition, Assigment 1

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

of data being analysed. It is interesting to note that the pixels that are always white are mainly on the edges, but they are also almost symmetrical, with a slight concentration on the left side. A table with the pixel indexes can be found in the appendix.

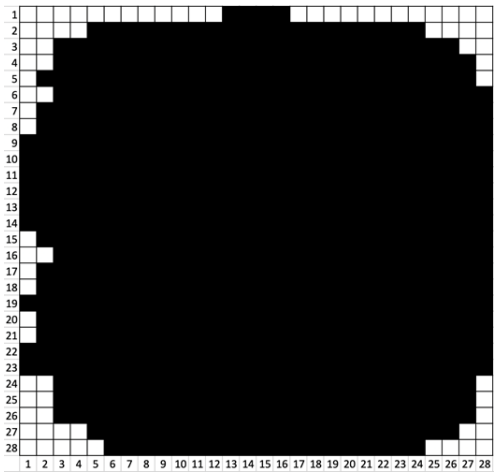

Figure 3: An overview of the useless pixels.

A different interesting analysis that could be done is checking the amount of mirrored pixels in the image. There are of course a number of ways to approach this kind of feature. For example, which axis do you mirror on and how strict do you want the mirroring to be, are important questions to answer. In our case we decided to define a number of different features, to deal with the different axes. Because the digits are represented as pictures of 28 by 28 pixels, we can easily define a horizontal and a vertical axis between pixel 14 and 15 on both axes (given that the first pixel is (1,1)). This allows us to define zones for which we can check the amount of mirrored pixels. See figure 4 for a visualisation.

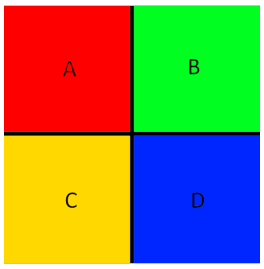

Figure 4: Zones to check mirrored pixels

Now these zones have been defined we can check the amount of mirrored pixels for each zone with respect to all other zones. This gives us six features because the mirror property is symmetric. If we want to check the amount of mirrored pixels across the horizontal axis, for example, we would need to compare zone A to zone B and zone C to zone D. The six features we have defined are:

- zone A to zone B
- zone A to zone C
- zone A to zone D
- zone B to zone C
- zone B to zone D
- zone C to zone D

These features should give the best overview on the amount of mirrored pixels and where they are concentrated. It should also be noted that when checking zones A to D and B to C the pixels are considered mirrored on both axes. The values of these features are found by checking pixels in mirrored locations and seeing if the ink value (the scale from white to black) falls within a particular margin. The reason we allow for a margin for the ink values to fall between is because we are working with noisy data and we are interested in more than just strictly mirrored pixels (i.e. pixels with the exact same ink value). We want to find if the digits are symmetrical in relatively the same area. If a mirrored pixel that falls within the margin has been found, it is counted and this sum of pixels is then finally transformed to a percentage. So we have six features which describe the percentage of mirrored pixels (within a margin) with respect to the zone defined in figure 4. The following table shows the mean of these values, with a margin of 100%. How we chose this margin is explained in section 3.2. The values in the table have been scaled between 0 and 1 for the highest and lowest values.

	AtoB	AtoC	AtoD	BtoC	BtoD	CtoD
0	0.467	0.47	0.414	0.32	0.221	0.205
1	0.97	0.936	1.0	0.524	0.389	0.503
2	0.677	0.667	0.602	0.39	0.236	0.262
3	0.623	0.378	0.556	0.118	0.308	0.438
4	0.531	0.557	0.669	0.335	0.451	0.597
5	0.448	0.336	0.501	0.176	0.226	0.359
6	0.371	0.669	0.59	0.438	0.465	0.562
7	0.65	0.182	0.418	0.0	0.36	0.556
8	0.612	0.542	0.567	0.123	0.021	0.262
9	0.662	0.412	0.59	0.198	0.398	0.593

Table 1: Mean of mirror percentage in zones (scaled)

As can be seen from the table, the class '1' is the most symmetric on average across all the zones. Most classes seem to be symmetric from zone A to other zones and less so from zone B to other zones. Furthermore, the left side of the images seems to be the most mirrored along the vertical axis. These observations could help define a feature to help with classification

3 FEATURE SPACES

In this section we will explain two approaches to the feature space, rather than using the pixels as features. This is interesting because it allows for some different options when classifying the data and gives a better understanding of the dataset. It should be noted that these feature spaces were only tested using logistic regression (henceforth LR) and that no penalty was given to LR. Consequently, there is no need to use cross-validation to tune a hyperparameter and therefore, we will only explain these feature spaces. After we have given the feature spaces and the results of the classification

with LR for these feature spaces, we give the results of using both features together. We used the logistic regression method (*LogisticRegression()*) from the Python package scikit-learn[4]. To allow for the reproduction of our results we passed random state equal to 0 to the method in all our applications of it, in this and the next section.

3.1 Ink Feature

The ink feature is defined as the amount of ink used to write a particular digit. This feature is then used to predict the class of the digit. The cost in ink of a digit is calculated by summing the values in all 784 pixels, thus, the feature contains 42,000 observations of summed ink values. For this feature, each row of digits has the pixel values replaced by a single value corresponding to the ink sum. The new feature space will, for example, look as follows:

Index	Label	Ink-sum
0	1	16649
1	0	44609
2	1	13425
...	...	...
41998	6	26381
41999	9	18178

Table 2: Ink-sum feature space

The results of this analysis are given in table **??**. To better visualise the spread and efficacy of this feature we use a box plot of the mean and standard deviation of the ink-sum for each of the classes. The x-axis denotes the digit classes and the y axis denotes the ink-sum. See figure 5.

Class	Mean	Std Dev
0	34632.40	8461.89
1	15188.46	4409.46
2	29871.09	7653.00
3	28320.18	7574.10
4	24232.72	6374.63
5	25835.92	7526.60
6	27734.91	7530.50
7	22931.24	6168.34
8	30184.14	7777.39
9	24553.75	6465.23

Table 3: Caption

It should be clear that this feature will probably not be effective at distinguishing classes because, apart for the class '1', the mean and the standard deviation are very similar. As a result of this, the boxes are almost on the same level. Moreover, the number of outliers is very high for each class. In conclusion, this model would probably only classify the digits that belongs to the class '1' correctly, whilst it will misclassify almost all the other digit classes, since it has no clear way to distinguish between different classes. In fact, if we reflect more in depth on this feature, it will appear obvious that the ink-sum will depend too much on how big the handwritten digit

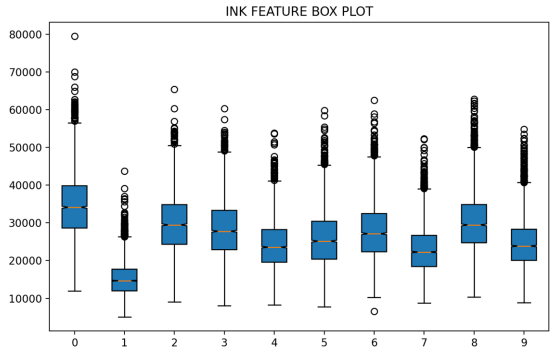

Figure 5: Ink-sum box plot

is, instead of taking into account the position of each pixel while computing the sum. Furthermore, using abstract reasoning, if we consider handwritten digits of the same size, digits like '6' and '9' will be hard to distinguish since the ink-sum will be practically the same.

As was mentioned previously, we used LR to classify the digits using this feature space. The LR package made available by sci-kit learn package for Python was used[4]. Furthermore to allow for accurate reproducibility the random state for LR was set to 0. The other parameters used in the function are: 'newton-cg' as solver, 'multinomial' as multi_class and 'none' as penalty. Before running the regression, we scaled the data in order to have mean equal to 0 and standard deviation equal to 1 because this will improve the accuracy of the logistic regression.

As was expected the accuracy of the model with this feature space is very low at 22.69%. Table 4 gives the precision, recall, f1-score and support for each class of digits.

class	precision	recall	f1-score	support
0	0.25	0.59	0.35	4132
1	0.44	0.82	0.57	4684
2	0.14	0.08	0.10	4177
3	0.12	0.24	0.16	4351
4	0.00	0.00	0.00	4072
5	0.00	0.00	0.00	3795
6	0.00	0.00	0.00	4137
7	0.15	0.39	0.22	4401
8	0.00	0.00	0.00	4063
9	0.13	0.05	0.08	4188

Table 4: Result table of LR and ink

The class '1' has a recall value of 82%, meaning that 4 times out of 5 that '1' should be predicted, it was indeed predicted. A more interesting remark is that the class '4', '5', '6' and '8' seems to be never predicted. The confusion matrix should provide more information about this aspect:

In this matrix, the columns represent the model predictions and the rows represent the actual digit. Hence, it can be seen that for

	pred0	pred1	pred2	pred3	pred4	pred5	pred6	pred7	pred8	pred9
0	2420	83	322	805	0	0	0	384	0	118
1	10	3823	5	101	0	0	0	722	0	23
2	1495	280	327	1039	0	0	0	874	0	162
3	1246	408	335	1037	0	0	0	1141	0	184
4	440	829	196	886	0	0	0	1496	0	225
5	727	671	198	846	0	0	0	1190	0	163
6	1057	450	296	982	0	0	0	1145	0	207
7	325	1190	149	819	0	0	0	1700	0	218
8	1430	192	343	1047	0	0	0	879	0	172
9	484	763	196	870	0	0	0	1651	0	224

Table 5: Confusion matrix for Ink feature

example, the digit '7' was predicted 1651 times when the actual digit to be predicted should have been '9' and the digit '1' is predicted 3823 times when the actual digit to be predicted is '1'. Thus, the main diagonal tells us the number of correct predictions within each class. Moreover, by comparing each digit, this matrix tells us how good the model is at distinguishing between one digit or another, interestingly, it can be seen that the classes '4', '5', '6' and '8' were never predicted. For example, the model is good at distinguish the class '1' from the class '8', in fact, when the true class was '8', the class '1' was only predicted 192 times (which is reasonably small with respect to the other errors) and when the true class was '1', the class '8' was never predicted. However, it's not that good at distinguishing, for example, the class '3' and '8': although '8' was never predicted when the true class was '3', the class '3' was instead predicted 1047 times when the true class was indeed '8'.

The reason behind these errors, besides the fact this feature does not give the classifier much information, is found by looking at the weight factors for each class:

$$
\begin{aligned}
P(t=0|x) &= -0.1978704 + 1.34686018x \\
P(t=1|x) &= -1.96428274 - 3.09059168x \\
P(t=2|x) &= 0.22468821 + 0.7471617x \\
P(t=3|x) &= 0.34207981 + 0.5170475x \\
P(t=4|x) &= 0.27509953 - 0.20760753x \\
P(t=5|x) &= 0.24625231 + 0.10043532x \\
P(t=6|x) &= 0.31113968 + 0.42464827x \\
P(t=7|x) &= 0.26876329 - 0.48613177x \\
P(t=8|x) &= 0.17756873 + 0.79122988x \\
P(t=9|x) &= 0.31656157 - 0.14305187x
\end{aligned}
$$

Table 6: Scaled weight factors per class for LR

If for example we consider only '3' and '8' and the digit predicted was always '3' (using the same input x and given only those 2 equations), this means that:

$P(t=3|x) > P(t=8|x) \forall x \in DATASET$

Thus: $0.34207981 + 0.5170475x > 0.17756873 + 0.79122988x$

Obtaining: $x < 0.6$

This means that all the '8' digits in the dataset have a scaled ink-sum value of less than 0.6 and that's the reason why, when an '8' should be classified, a '3' (or another class) was instead predicted. This is another reason, besides the low accuracy, that makes this feature not so useful.

3.2 Mirror Feature

This feature has already been explained in the exploratory analysis, however we still need to give the results of the classification and explain how we chose the margin. Recall that this feature checks pixels in mirrored location for a number of zones defined in the exploratory analysis. A pixel is considered mirrored if the ink value falls within a particular percentile margin. To find the best margin for this feature, we ran a test classifying the data for different margins ranging from 0 to 100%. The results can be found in figure 6.

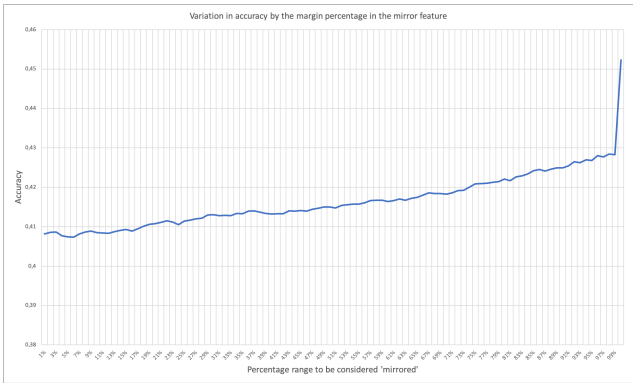

Figure 6: Margin test for mirror feature

As can be seen from the graph, the feature works best when the margin is at 100%. We believe this is because when the margin is so high, a pixel is only considered **not** mirrored when there is a significant difference in the amount of ink in those pixels (i.e. the pixel is either twice as dark or twice as light as the mirrored pixel). This means that the feature concentrates on the important parts of the digits.

class	precision	recall	f1-score	support
0	0.39	0.41	0.40	4132
1	0.79	0.93	0.85	4684
2	0.37	0.40	0.38	4177
3	0.30	0.18	0.22	4351
4	0.36	0.36	0.36	4072
5	0.41	0.28	0.34	3795
6	0.52	0.61	0.56	4137
7	0.48	0.61	0.54	4401
8	0.34	0.35	0.34	4063
9	0.36	0.31	0.33	4188

Table 7: Result table of Mirror feature

Furthermore we can see from this graph, the mirror feature space outperforms the ink feature by a large amount. At it's best, the mirror feature had an accuracy of 45.23%. Another thing to note, which can be seen clearly from the confusion matrix, is that the mirror feature has no trouble classifying digits in the classes '4', '5',

'6' and '8', whereas the ink feature never classified a single sample in those classes. This is due to the fact that the mirror feature contains more features to base it's classification on, in comparison to the ink feature, which has only one. In sum, the mirror feature takes the idea of the ink feature and builds upon it, achieving a much better result. Not only is this because there are more features for LR to make use of, these features also retains some location information.

	pred0	pred1	pred2	pred3	pred4	pred5	pred6	pred7	pred8	pred9
0	1694	57	592	114	113	241	566	252	425	78
1	2	4358	70	5	117	2	25	8	39	58
2	581	295	1664	142	258	70	381	66	557	163
3	154	132	221	775	382	461	153	1077	440	556
4	206	212	318	114	1454	128	713	158	285	484
5	504	76	329	425	366	1070	198	283	379	165
6	396	188	328	14	446	74	2539	3	133	16
7	107	15	251	403	108	101	10	2706	153	547
8	476	107	471	204	286	276	226	290	1421	306
9	217	99	312	361	479	159	73	816	358	1314

Table 8: Confusion matrix Mirror feature

3.3 Ink and Mirror merged

It is also possible to combine feature spaces such as the ink and mirror spaces, especially if they are small. Here we combined these feature spaces and classified the digits with LR once again, to see if the accuracy could be improved even more. The result was not so different from the result of using the mirror feature alone. This is because the mirror feature captures much of the same information as the ink feature, but it also retains some of the location information of the ink used.

	pred0	pred1	pred2	pred3	pred4	pred5	pred6	pred7	pred8	pred9
0	1988	16	617	152	62	288	497	116	353	43
1	0	4408	62	5	91	6	23	18	21	50
2	616	146	1776	168	270	107	353	49	553	139
3	190	65	293	1077	367	506	143	808	412	490
4	116	210	241	185	1522	230	684	176	234	474
5	274	109	212	342	402	1411	194	343	331	177
6	355	113	301	19	484	108	2582	11	138	26
7	44	30	141	200	93	169	10	3295	70	349
8	469	61	502	344	290	424	222	108	1372	271
9	146	91	255	451	493	302	64	773	291	1322

Table 9: Confusion matrix merged features

The accuracy of the merged features (49.41%) did not differ significantly from the mirror feature. We surmise this is due to the fact that the mirror feature measures much of the same information as the ink feature, but with added information. This means that adding the ink feature to the feature space of the mirror feature does not really add anything the classifier did not already know.

class	precision	recall	f1-score	support
0	0.47	0.48	0.48	4132
1	0.84	0.94	0.89	4684
2	0.40	0.43	0.41	4177
3	0.37	0.25	0.30	4351
4	0.37	0.37	0.37	4072
5	0.40	0.37	0.38	3795
6	0.54	0.62	0.58	4137
7	0.58	0.75	0.65	4401
8	0.36	0.34	0.35	4063
9	0.40	0.32	0.35	4188

Table 10: Result table of merged Ink + Mirror feature

4 PIXEL FEATURES

In this section we return to the pixels as our feature space. For this purpose, we will firstly use all the 784 pixels values as features. We will, try for LR only, to use the pixels values that have not previously been classified as 'possibly useless' (since they are always white), and compare the accuracy results from both the scenarios. Furthermore, an operation of scaling the features is made even here, since it will improve the performances of the tested methods. To classify the images a number of different classifiers were used, logistic regression (henceforth LR), support vector machines (henceforth SVM) and, feed forward neural networks (henceforth NN). Due to the fact that LR and SVM are supervised learning algorithms, it is necessary to use labeled data, such as the MNIST dataset. Furthermore, to keep the comparison between algorithms fair, supervised learning was also used for NN. This allows us to compare the models directly using a statistical test, to determine if there is a significant difference in the results. The true accuracy of these classifiers is dependent on the use of unbiased training data and the tuning of hyperparameters. Thanks to the exploratory analysis we know that the data is relatively unbiased. To find optimal values for the hyperparameters cross-validation was used and our approach will be explained in the next section.

4.1 Cross-validation

The tuning of hyperparameters for different classification models can be a tricky thing. Thankfully, cross-validation allows us to compare models with different hyperparameter settings to find which one classifies the data with the highest significant accuracy. We used two different methods for cross-validation. For LR we used manual cross-validation, with which we mean to say we implemented our own cross-validation method. For SVM and NN the GridsearchCV function from the scikit-learn package for python was used[4]. The GridsearchCV method creates a classifier for every combination of the hyperparameters defined by us and then uses cross-validation to find the best parameters. This also helps us to avoid overfitting on the data. The manual cross-validation method does the same thing in principle, but we implemented the function *cross_val_score* from the scikit-learn package ourselves within a for loop, testing different hyperparameters each time, in order to store some intermediate results. We used 10 fold cross-validation on the models because this data set is large and this is an accepted number of folds for larger

data sets according to the literature [3]. Each of these folds is used as a test set for each of the models made by GridsearchCV or our manual method, while the other folds were used as training. This helps avoid overfitting as all of the models are tested on unseen data. The dataset was divided into 5000 training samples and 37000 testing samples and the cross validation method was only used on the 5000 samples: this makes sure the computation time doesn't gets too large and the error (therefore, also the accuracy) of each method can be calculated on new and never seen data: the 37000 test samples. Obviously, in order to have a fair analysis, the data are always split in the same way when analyzing these 3 different models, and this operation is done by providing *random_state=3* to the scikit-learn function: test_train_split[4]. Splitting the data using this function ensures that the various training and testing sets are as homogenous as possible: the distribution of data for each digit will be, for each sub-dataset, as close to the original dataset as possible.

4.2 Logistic Regression

The logistic regression with lasso penalty only has one parameter to tune: the inverse regularization strength. In python, this value is defined as: $C = 1/\lambda$. A link to the code that performed this analysis is given in the appendix and the C-values that were tested are given here:

C-values: [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.74, 0.75, 0.76, 0.77, 0.78, 0.79, 0.8, 0.81, 0.814, 0.815, 0.816, 0.817, 0.818, 0.819, 0.82, 0.823, 0.826, 0.829, 0.83, 0.832, 0.835, 0.838, 0.84, 0.841, 0.844, 0.847, 0.85, 0.86, 0.87, 0.9, 1, 1.1, 1.2, 1.3, 1.4, 2, 3, 4, 5, 6, 7, 8, 9, 10, 50, 100, 200, 220, 260, 300, 320, 360, 400, 420, 460, 500, 520, 560, 600, 1000, 1500, 2000, 3000, 4000]

We want to compare the results of LR while removing the 'possible useless' pixels and without removing them, therefore, a 'manual' cross validation (without using the GridSearchCV function) was performed, in order to store some intermediate results. We tested some possible values for C from a very wide range. To the LR function in python,in the parameters option, the penalty 'l1' corresponds to the lasso penalty and the solver 'saga' was chosen because, among all the available solvers, it is the only one that could handle a multinomial logistic regression with a lasso penalty. Moreover, a random state of 0 was given. By repeating this operation while keeping and removing the possibly 'useless' pixels we obtained the following 2 graphs:

The results show a different scenario from the one expected: even if a pixel is white in all samples, regardless of which class we are looking at, it could still make a difference in classifying the digit. We can see this is the case since the cross validated error for each point is always lower in the scenario when we keep the 'useless' pixels as features.

A possible explanation for this behavior could be found in the overfitting data problem: since we have more features in the second scenario, it is possible that, even though a cross validation for regularizing the data is performed, we still have a bit of overfitting, that causes that roughly 1.5 % less error with respect to the scenario where we have removed the pixels.

This is the reason why, we will perform the rest of the analyses, we always consider all the 784 pixels present in the dataset, without

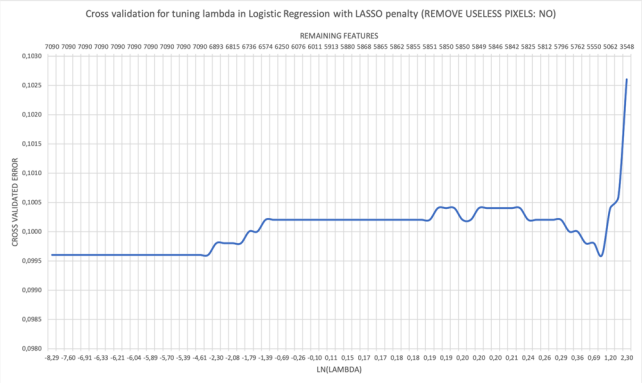

Figure 7: Cross-validation for LR with 'useless' pixels

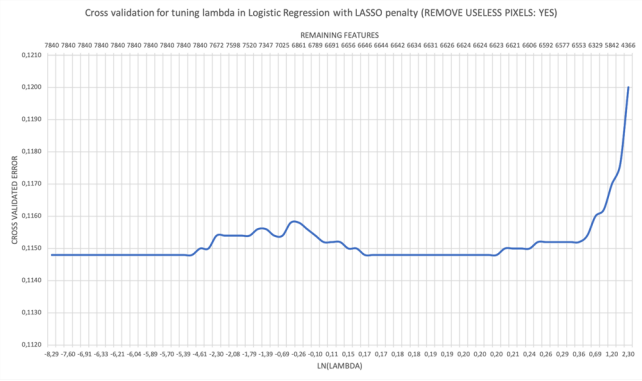

Figure 8: Cross-validation for LR without 'useless' pixels

removing the ones that we had assumed to be useless. Moreover, by looking at the graph with all the pixels considered, we still have to choose the optimal value for λ, which is the value that minimizes the cross validated error. The minimum value is reached in 2 different areas of the graph:

- the interval from negative infinity to -3.91
- the point 0.91629

Where they both reach a cross validated error of 9.96%.

The discriminating part that will tell us whether to choose one area or the other, is the complexity of the model, determined by the number of remaining features. Since, at an equal level of cross validated error, we want to choose the less complex model, the optimal value is the point 0.91629 since it has 5356 remaining features compare to the point -3.91 which has 7090 remaining features.

Thus, $ln\lambda = ln0.91629$ then $\lambda = 2.5$ and C=0.4 are, for this dataset, the cross validated optimal values for the logistic regression with lasso penalty.

4.3 Support Vector Machines

To classify the MNIST data with an SVM we used the SVM (*SVC()*) method from the scikit-learn package for Python[4]. To allow for the reproduction of our results, we passed random state equal to

0 in every application of the SVM. Here, there are more hyperparameters to tune with respect to the analysis of LR. Since we can't plot the cross-validated error with a variation of the hyperparameters (the graph would have more than 2 dimensions, since the number of hyperparameters to tune is 4) we do not need to store intermediate results, therefore we chose to use the pre-built function: 'GridSearchCV' in order to tune all the combinations of hyperparameters to find the optimal values. The list of optimal values that were searched are as follows:

C: *[0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.75, 0.8, 0.81, 0.816, 0.82, 0.83, 0.835, 0.84, 0.844, 0.85, 0.86, 0.87, 0.9, 1, 1.1, 1.2, 1.3, 1.4, 2, 3, 4, 5, 6, 7, 8, 9, 10, 50, 100, 200, 220, 260, 300, 320, 360, 400, 420, 460, 500, 520, 560, 600, 1000, 1500, 2000, 3000, 4000]*
kernel: $['linear', 'poly', 'rbf']$
degree: *[3, 4, 5]*
gamma: *['scale', 'auto']*

As before, we tested some possible values of C, but now we also want to find the correct type of kernel for the SVM. This is because this data might be better separable with non linear kernels. The results of the LR mean that we know the linear kernel works, but we are interested in finding the most optimal model, so checking non-linear kernels is preferred. Furthermore, as we want to keep the complexity of the model as low as possible, we have set the polynomial kernel to go no bigger than 5. We also want to check that gamma is set to an appropriate level, either $\frac{1}{n\_features}$ (gamma is set to 'auto') or $\frac{1}{(n\_features * n\_variables)}$ (gamma is set to scale 'scale')[4].

As output, we obtain these values as optimal:

C: 50, **degree:** 3, **gamma:** 'scale', **kernel:** 'poly'

From this it follows that $\lambda = 0.02$.

4.4 Feed Forward Neural Network

In order to predict digits with a NN we made use of the Multilayer Perceptron (*MLP()*) method from the scikit-learn package for Python[4]. Furthermore, we passed random state equal to 0 in all our applications of this method, to allow for the reproduction of our results. One of the great difficulties of using NN to classify data is that the models are very hard to interpret. This is mainly due to the fact that NN make use of 'hidden layers' of nodes and it is very difficult to visualize these hidden layers and what impact they have on the NN. As such, choosing values for the hyperparameters of the NN is not trivial. We based our decisions on varying the hidden layer size from the size of the output to more than twice the size of the input and also tested, for a number of these values, a different number of hidden layers. Furthermore, we decided to test for different activation functions, different values for the learning rate and whether the learning rate was adaptive. Note the numbers in the tuples in hidden_layer_sizes denote the amount of nodes and number of hidden layers respectively. The complete set of parameters tested are as follows:

alpha: *[1e-3, 1e-2, 0.02, 0.03, 0.04, 0.05, 0.1, 0.2, 0.3, 0.7, 0.9]*
hidden_layer_sizes: *[(10, 1), (50, 1), (100, 1), (200, 1), (300, 1), (397,1), (300, 2), (397, 2), (500, 2), (800, 2), (512, 3)]*
learning_rate_init: *[0.0001, 0.001, 0.01, 0.1]*
learning_rate: *['constant', "adaptive"]*

Here the same approach was taken as with SVM, as the number of parameters exceeds 1, we would need a multidimensional graph to visualise the change in accuracy. We opted not to try this and focus our effforts on finding parameters which give the most accurate results. We used GridsearchCV to test all test different parameters and the results were the following:

alpha: 0.7, **hidden_layer_sizes:** (512, 3), **learning_rate_init:** 0.001, **learning_rate:** 'constant'

5 RESULTS

After having tuned the parameters with the training set of 5000 samples, we tested the three classifiers against our remaining data (37000 samples), to try to accurately describe the error. In the following sections the results are given per classifier.

5.1 Logistic Regression

After this hyperparameter tuning, we finally perform the logistic regression and calculate the error on the 37000 samples of testing data. The output of the model is as follows:

Accuracy Score: 89.18 %

class	precision	recall	f1-score	support
0	0.93	0.95	0.94	3662
1	0.89	0.97	0.93	4086
2	0.92	0.85	0.88	3677
3	0.87	0.87	0.87	3816
4	0.88	0.92	0.90	3582
5	0.86	0.83	0.84	3347
6	0.92	0.95	0.93	3630
7	0.91	0.91	0.91	3885
8	0.87	0.81	0.84	3587
9	0.87	0.86	0.87	3728

Table 11: Result table of LR with Cross-Validation and LASSO

	pred0	pred1	pred2	pred3	pred4	pred5	pred6	pred7	pred8	pred9
0	3476	2	17	8	10	58	62	3	22	4
1	1	3979	15	20	0	12	10	5	41	3
2	45	69	3123	69	94	12	73	63	113	16
3	19	42	85	3306	7	146	26	55	83	47
4	9	34	30	5	3298	11	33	9	17	136
5	63	51	17	166	46	2767	78	23	91	45
6	34	23	30	1	23	51	3436	2	30	0
7	12	81	46	13	51	8	5	3523	4	142
8	48	168	35	149	33	147	25	23	2888	71
9	50	31	14	48	174	22	3	155	29	3202

Table 12: Confusion matrix for LR with cross validation and LASSO

The accuracy achieved for this test data is pretty high, though the model still misclassifies some digits, as seen in the confusion matrix. What is interesting to note about the confusion matrix is that it confirms that the model is most likely to misclassify number that are visually similar. For example when the number is a '9' the algorithm is most likely to classify it correctly but if it makes a mistake, it is most likely to classify a '4' or a '7' in that order. Similar observations can be made for the number '8' and predicting '1', '3' and '5'.

5.2 Support Vector Machine

Performing the analysis on the test set using the best values for the hyperparameters, we obtained these results:

Accuracy Score: 94.28%

class	precision	recall	f1-score	support
0	0.98	0.97	0.97	3662
1	0.98	0.98	0.98	4086
2	0.95	0.90	0.93	3677
3	0.94	0.93	0.93	3816
4	0.91	0.96	0.93	3582
5	0.94	0.92	0.93	3347
6	0.97	0.96	0.97	3630
7	0.96	0.92	0.94	3885
8	0.88	0.95	0.91	3587
9	0.93	0.93	0.93	3728

Table 13: Result table of SVM

	pred0	pred1	pred2	pred3	pred4	pred5	pred6	pred7	pred8	pred9
0	3542	0	13	3	13	26	35	0	27	3
1	0	4022	15	8	4	2	8	7	16	4
2	13	15	3327	51	71	8	15	44	126	7
3	4	3	41	3536	7	59	6	21	117	22
4	2	6	21	2	3431	5	10	5	7	93
5	9	2	4	82	20	3089	38	5	78	20
6	19	2	22	3	23	30	3502	1	28	0
7	8	26	29	8	97	5	0	3591	25	96
8	12	12	23	42	19	49	4	2	3392	32
9	12	7	13	20	104	17	1	60	43	3451

Table 14: Confusion matrix for SVM

It is obvious that the accuracy of the SVM is higher than that of LR. If we take a closer look at the result table and confusion matrix, we see that LR and SVM make similar errors. Another interesting thing to note is that when the digit is a '3', class '8' is often predicted but this is not the case for the opposite situation.

5.3 Feed Forward Neural Network

The results for the best hyperparameter values for the NN are:

Accuracy Score: 88.61%

The NN seems to have performed at a similar level to LR, we will, however, perform a statistical test in the next section in order to find out if the difference is significant. When comparing the result

class	precision	recall	f1-score	support
0	0.96	0.92	0.94	3662
1	0.93	0.97	0.95	4086
2	0.90	0.83	0.87	3677
3	0.84	0.87	0.85	3816
4	0.82	0.85	0.83	3582
5	0.77	0.84	0.80	3347
6	0.93	0.93	0.93	3630
7	0.94	0.90	0.92	3885
8	0.86	0.81	0.83	3587
9	0.87	0.88	0.87	3728

Table 15: Result table of NN

	pred0	pred1	pred2	pred3	pred4	pred5	pred6	pred7	pred8	pred9
0	3368	2	12	152	37	12	34	4	8	33
1	0	3961	26	5	2	26	30	7	28	1
2	7	40	3054	198	30	172	130	22	17	7
3	49	4	92	3301	146	157	6	10	31	20
4	3	2	24	84	3041	76	8	7	78	259
5	13	25	64	126	103	2824	31	5	145	11
6	6	43	75	14	7	86	3389	1	9	0
7	7	79	17	12	16	33	12	3505	93	111
8	13	83	10	34	161	286	8	30	2895	67
9	33	3	3	11	182	14	10	149	60	3263

Table 16: Confusion matrix for NN

tables of these two algorithms we can see that NN out performs LR in some classes but under performs in others.

It is interesting to note that the performance indicator 'recall' found in the merged Ink + Mirror feature on class '1' (table 10) is 94%, very similar to the 'recall' values achieved using the LR, SVM and NN methods. This means that the class '1', among all the classes, is the most recognizable and symmetric, as we also saw in the exploratory analysis

6 CONCLUSIONS

To compare the results of the different algorithms a 5x2 cross-validated t-test using the function paired_ttest_5x2cv from the python library 'mlxtend' was performed on all combinations of classifiers. We found no statistical significance between the accuracies of the models. Table 17 shows the p-values for every combination of algorithm. The table shows that all p-values are under 0.05, so there is no statistical significance between model accuracies found.

classifiers	p-value	statistic
LR vs SVM	0.000	-37.713
SVM vs NN	0.022	3.279
NN vs LR	0.028	3.054

Table 17: P-values for all classifiers

Further research could be done on the tuning of the NN as this is the most complex model to tune. Furthermore, some preprocessing

of the data could significantly increase the accuracy. One method found in the literature was the act of deskewing [2]. This takes digits (or other images) that are at an obvious angle and sets them straight. This homogenises the dataset so that the models would have an easier time classifying the data. Another approach that could be tried, but wasn't in this paper due to time constraints, is the adding of the mirror features to the pixel feature space. Because the pixel data is compressed to a single line, some of the information regarding which pixels are near each other is lost, adding in the mirror feature might give the classifiers the extra information necessary to increase the accuracy.

REFERENCES

[1] THE MNIST DATABASE of handwritten digits. http://yann.lecun.com/exdb/mnist/. Accessed: 2020-12-04.

[2] R. V. Cox, B. G. Haskell, Y. LeCun, B. Shahraray, and L. Rabiner. On the applications of multimedia processing to communications. *Proceedings of the IEEE*, 86(5):755–824, 1998.

[3] T. Hastie, R. Tibshirani, and J. Friedman. *The elements of statistical learning: data mining, inference, and prediction.* Springer Science & Business Media, 2009.

[4] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

A 'USELESS' PIXEL INDEXES

pixel0	pixel11	pixel26	pixel57	pixel196	pixel672	pixel755
pixel1	pixel16	pixel27	pixel82	pixel392	pixel673	pixel756
pixel2	pixel17	pixel28	pixel83	pixel420	pixel699	pixel757
pixel3	pixel18	pixel29	pixel84	pixel421	pixel700	pixel758
pixel4	pixel19	pixel30	pixel85	pixel448	pixel701	pixel759
pixel5	pixel20	pixel31	pixel111	pixel476	pixel727	pixel760
pixel6	pixel21	pixel52	pixel112	pixel532	pixel728	pixel780
pixel7	pixel22	pixel53	pixel139	pixel560	pixel729	pixel781
pixel8	pixel23	pixel54	pixel140	pixel644	pixel730	pixel782
pixel9	pixel24	pixel55	pixel141	pixel645	pixel731	pixel783
pixel10	pixel25	pixel56	pixel168	pixel671	pixel754	

B LINK TO CODE

https://gitlab.com/Saeden/pattern-recognition-assignment