Jeffrey Alberts	3083179
Jur Kersten	4168763
Steven Fleuren	5506425

Project D - Lab-report

Exercise 1

a. For the MostFrequent classifier, the validation and testing accuracy were 14% (14 correct out of 100). The naiveBayes classifier gave different results for validation and testing. These scores were 69% (69 correct out of 100) and 55% (55 correct out of 100) respectively. Furthermore, the naiveBayes classifier used a smoothing parameter (k) of 2.0. As seen in these tests, there is a big difference in accuracy between the MostFrequent- and the naiveBayes-classifier. The MostFrequent classifier is simply a counter which just returns the values that was seen most often in the training data. This however doesn't lead to very accurate results, since the training data might not use all examples the same amount. Therefore, the result using the MostFrequent classifier will not be able to relativate its data. This is something that the naiveBayes classifier can do, since working with the Bayes rule which incorporates the probability of a certain attribute showing in the data when a specific class (or root node) is present. In this way distinction is made between certain data and its attributes which is not made using the MostFrequent classifier.

b. A smoothing factor, like Laplace smoothing which is used in the naiveBayes classifier, is necessary for any classifier that uses Bayes rule. The naiveBayes classifier uses a class and its attribute, which in machine learning it encounters in the training data, to calculate the probability-estimate. This probability-estimate, which is used in testing to accurately choose the correct class, consists of all the probabilities that the specific attributes that are found in the data belong to a certain class. If done correctly, this correct class is chosen so in a way the naiveBayes classifier can choose/make an accurate decision which class is correct. However, it is possible that certain attributes are never linked/initialised to certain classes in the training data and will therefore have a value of zero. Because of the calculation of the probability-estimate (product of all attributes), this would result in a probability-estimate of 0 which means that the data of <u>all</u> attributes is lost. A smoothing factor is implemented to make sure that no attribute ever has the value of zero in each specific class. The cost of this is some accuracy on the one hand, however the benefits in the classification process are greater for no data is lost.

c. The autotune option runs the naiveBayes classifier ten times with different smoothing values (k) ranging from 0.001 to 50.00. For all of these smoothing values, it tests it performance on the validation set and returns the score. Then, the naiveBayes classifier is ran on the test set with the k-value that corresponds to the highest score on the validation set and its results are reported. The results turned out ot be 74% on the validation set and 65% on the test set with k = 0.1. When comparing this with the scores with k = 2.0, it can be seen that the lower k value scores higher on both validating (74% vs 69%) and testing (65% vs 55%) respectively. The only difference between these two instances is the smoothing value, so the differences in scores can only be explained by this k-value. Seeing that it is relatively high at 2.0 makes a case that this value is too high and smooths out the probability estimates too much. These estimates are calculated per class, so when you smooth out the estimates, the scores for each class get closer together. This makes it harder to see the difference between classes, which is more so in testing data (difference of 10%) than in training data (difference of 4%), as seen by the bigger difference between validating and testing for both instances. This because more attributes that haven't been encountered before will appear in the test data, pulling the estimates for each instance in the data set closer to 0 and therefore smaller and harder to correctly interpret for the naiveBayes classifier. All in all, smoothing is necessary, but needs to be done carefully since smoothing out scores will make all scores get closer together, making differences smaller and therefore classification increasingly difficult.

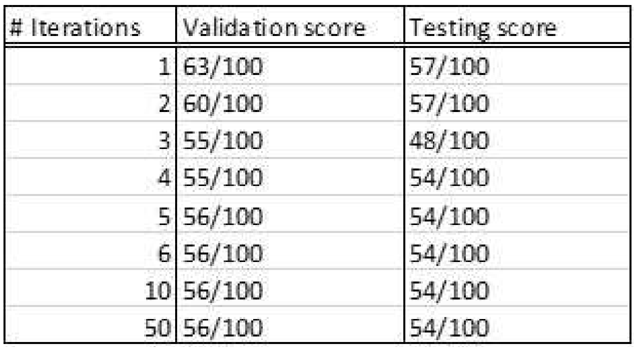
d. The naive Bayes classifier is based on the idea that if all n feature variables F_1, ..., F_n are independent given the label C, then

$$p(C|F_1,\dots,F_n) = \frac{1}{Z}p(C)\prod_{i=1}^{n} p(F_i|C)$$

The classifier can calculate the right hand side for each label, and return the label with the highest value as the best guess. The Z value is the same for each label, so it doesn't need to be calculated. If the number of features is large, like it is in the case of the digit data, the product of the probabilities becomes very small, and rounding errors might start to wreck havoc. Taking the logarithm avoids this problem: one can find the logarithm of the product of the probabilities by calculating the sum of the logarithm of each probability, and since the logarithm is a strictly ascending function the label with the highest product of probabilities will also have the highest logarithm of the product of probabilities.

e. A validation set can be seen as part of a training set which has been held back to check whether the training set works correctly and so it can be used to compare to the test set to see whether there are problems (like overfitting). The advantage of this is that the validation set, even though it is very similar to the training set and contains the same attributes, the classifier has not been allowed to train on this set so the accuracy is more objective. It is almost a check on whether the training data has been learned correctly, so sort of reliability. This is different from the test set in that this checks whether the classifier can generalise and interpret data (and its containing attributes) which are comparable but still structurally different from the training data. An example of this would be handwriting, where the training and validation data train on words written by someone in a specific handwriting, where the testing data would then be these same words but in another person's handwriting making it clear whether the classifier could still make out these words or if it learned to specifically (overfitted).

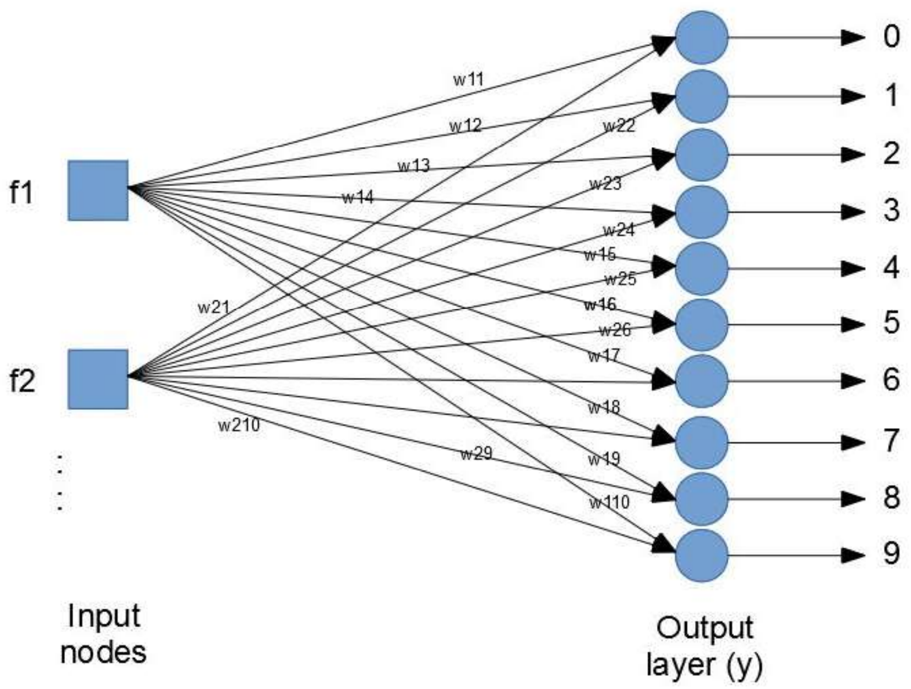
Exercise 2



# Iterations	Validation score	Testing score
1	63/100	57/100
2	60/100	57/100
3	55/100	48/100
4	55/100	54/100
5	56/100	54/100
6	56/100	54/100
10	56/100	54/100
50	56/100	54/100

a. The table above shows the validation and testing scores for each number of iterations. The perceptron goes through the training examples in the order chronologically in the order in which they are presented in trainingData[]. It appears that the designed perceptron is less effective at the classification process than the naive Bayes classifier. Where an optimized naive Bayes classifier (at k = 0.1) was able to score 74/100 correctly on the validation set and 65/100 on the testing set, the optimized perceptron scored significantly lower. However, the naiveBayes classifier is overfitted a lot more than the perceptron. Where the perceptron only has a slight difference in scores between validation and testing, the naiveBayes classifier shows a bigger gap. So the naiveBayes classifier is ultimately better in labeling the data and scores quite a lot higher in this area, but the perceptron isn't as overfitted. This gives the perceptron the advantage that, even though it can less accurately label data, the testing data can vary more from the trainings data and the perceptron could possibly still be able to make out the labels.

b. As seen in the data above, the highest scores are recorded after only one iteration. It would however be foolish to stop after only one iteration, since that would leave the perceptron' standard weights almost intact and learning would only occur in a limited amount. Also, the gap between the validation score and the testing score is quite high after one iteration. Both scores drop a little bit before they reach a score of 56/100 and 54/100 respectively. This is after 5 iterations, and this score doesn't change after 6, after 10 or even after 50 iterations. So, it seems as though the weights have leveled out and found an equilibrium after 5 iterations. However, if you stop after 5 iterations you would not know this has happened, and the smart thing to do would be doing a couple more iterations. If the scores remain the same each iteration, the program can be terminated. The program can be built to automatically terminate after a couple of iterations in which the scores don't change (in similar fashion as the autotune function of naiveBayes, even though this has preset values for k). A good rule of thumb would seem to be to terminate after 4 iterations have produced the same scores. In this example that would be after 8 iterations. This would regulate the program's time consumption, at a reasonably small cost since the perceptron' weights would be only updated by very small amounts and make it unlikely that the scores would change.



c.

Exercise 3

a. The weight sequence that was chosen for our perceptron was sequence ‘a’. This resembles the weight distribution for our perceptron the best. The choice for ‘b’ would have been unrealistic, since ‘b’ is a perfect image of certain representations of the digits, however our perceptron has proven it is not perfect (neither in validation nor in testing) which would then make it unlikely for it to produce the weight distribution in ‘b’. Also, the representation of the digits can come in many different forms and shapes, perfectly representing one of those forms would not lead to all the other forms being interpreted well, so it would not necessarily be beneficial to have this representation of weights.
b. Perceptron can only classify data that is linearly separable if it has one layer. By using a perceptron with multiple layers of neurons you can model data that is not linearly separable. Since the classification of digits demands nonlinear separation of the data, a more complex perceptron utilising multiple layers would be ideal for this problem.

Exercise 4

a. A weight change, especially at the start of training can influence the classification process a lot and push the classifier towards a certain choice. If that is the right choice then that is beneficial, but when it’s the wrong choice this can be very impactful. This is because the change in weight becomes increasingly smaller with each iteration and therefore the impact of each score or feature becomes increasingly small. That is why it is a wise thing to do to limit weight-change and start the learning conservatively (to decrease the risk of letting one score take over the whole learning process).

b. The autotune function works by changing the C value of the MIRA-classifier and checking the validation score for each of these C values. The C value which corresponds with the highest validation score is then computed to calculate the testing score of the MIRA-classifier. After running the autotune function on the MIRA-classifier, the results on validation were 68% (68/100) and the results on testing were 62% (62/100) with a C value of 0.008 . The C value is a constant value that is used in the MIRA-classifier to update the weights when y’ and y (so the label with the highest score and the true label) do not correspond (in other words: when a label is not correctly classified). In the case that a label is incorrectly classified, the weights that are put on both the true label and the label with the highest score are updated (heightened or lowered respectively). They are updated by a factor, but firstly this factor is multiplied by the learning rate (for reasons explained in 4.a). This learning rate is dependent on the weights and the factors already, but it is also capped to not go higher than a certain value. This is where the C value comes in, since this C value is the cap where the learning rate can’t go above. So the C value is in fact the maximum learning rate, and changing this C will have an impact on the weight change and therefore on the classifying process. When the C value is higher, weight changes on updates are stronger and have more impact on the classification process. When comparing the MIRA-classifier (validation: 68% and testing: 62%) to the perceptron (validation: 56% and testing: 54%) and the optimized naiveBayes classifier (validation: 74% and testing: 65%), it seems that the MIRA-classifier is better than the perceptron in validation but more significantly in testing. A logical explanation for this is that the MIRA-classifier uses learning rate to normalise the changes that are made to the weights where the perceptron does not do this. As for the naiveBayes classifier, it scores higher on validation but only slightly higher on testing. It seems that the naiveBayes classifier and the MIRA-classifier are more or less comparable with each other in performance even though their methods are very different. But, the naiveBayes uses a smoothing factor, which can be compared to the learning rate in that it also normalises results. This might be an explanation for the comparability of both systems, as both the smoothing factor and the C value have been optimized in both systems by means of the autotune function.

c. The images of the results for the MIRA classifier can be found below. For the first two labels we added an image 100 highest weight features of the perceptron classifier for comparison. While the two images definitely have differences, it’s hard to tell which classifier is better by just looking at these images.

```
=== Features with high weight for label 0 ===

        #
      ##  #
      #### #
     ####  ##
    ### # #  #
     ######  #
  #  #    #
  #       #
 ##       # #
 ### #     ###
### #      ###
##       ###
###       #
###      #  # #
###        #
###        #
 ####
  ## #    #
   #### ###
   ######
    #####
```

```
=== Features with high weight for label 1 ===

   ###
  #####
  ####  ####
   ## #####
      ###
  #####  #
 ######
   ###
    ###
    ##
   ###
   ####
  ####
  ####
 ###### #  #
 ###### # ##
 ###### # ##
 ### ## ###
  #   #
#  #
#
```

```
Features with high weight for label 0 (perceptron)

      #  ##
      ######
       ######
       #####
          #
      ###    #
   #   ##    ##
   ##       ##
   ##       ##
          ##
 ### #       ##
 ## #       ##
 ###        #
####   #    #  #
 ###
 ####        ##
  ########
   ######   #
   # #####
    ######
      ##
```

```
Features with high weight for label 1 (perceptron)

      # #
      #   ##
     ##  ###
    ### ####
     #  #
    ## #
   ## ##
   ####
##   ####
# #   ###
    ###
    ###
   ####
   ####
  # ###  #
  ######   #
 #######
 #### ##  #
  ####
# #####  #
  #  #  #
 ##  ###
```

```
=== Features with high weight for label 2 ===


     #########  #
    ########### #
    ###### #
     #  #
     #   #
                    #
                 # ##
                   ##

                   #

          ##      #
         ##       ##
      #  ####         ###
     ##   ###          #
      ## #####         ##
      ### ###      ## # #
       # #         #####
                  ######
                 #######
```

```
=== Features with high weight for label 3 ===


                   #

  #     #  #   #     #
  ###    # # ##### ###
  ##       #     # ##
  ###          #  #
               #
          #
         ##     ## #
         ###     # #
                #####
              #  #####
                  # ##
                  ####
                  ####
    ####         #####
    #####        # ##
    #####          #
     #### #  ##
      ######
```

```
=== Features with high weight for label 4 ===

                      #
                    ####
                 ###
                       #
                       #
     ##  ## #          #
    ##    ####     #  ##
    ##  ###
    ##  ###     #     #
    #########  #####  #
    ###### #########  #
    ######## #  ###    #
     ####   #  ####    #
       #       #




               # # #
                 #
         ##
         #
```

```
=== Features with high weight for label 5 ===


                       #
        ##           #####
       ## #         ######
        #      #    ######
        ##      ##### ####
           #  ##        #
            # #
         ## #
           ###
         #####
         ###  #
            #   ##
             #####
     ###      ###
     ##    ######
       ####### #
      #####    #
       #      #
    # #
    ###
```

```
=== Features with high weight for label 6 ===


               ####
              ######
              #####
              ####
             ####



           ###
          ###
          ###   #    ##
         #### ####   ##
         ### ####    ##
         ### ###   ###
         ###      # #
         # #      ####
         ###     #####
         ###     ###
         ###     # #
        ##### #   #
```

```
=== Features with high weight for label 7 ===


      #####
     #####     ###
     #####   # ####
        ###   ##### #
     ####    ###  # ##
     ###          # #
     ##          #####
     #           #  #####
                ####  ###
               #####  #
      #       ####
              ##
               ##
               #
               #


        #
       ##   ####
       ##   ####
     ##
```

```
=== Features with high weight for label 8 ===

                     #
             ##### ####
             #########
           ### #     #
         ###        ####
         ##         ####
         #      #  #  ##
         #   # #####  #
         #     ##
            #

            ##
            ####
           ##  #
         ####    #
       ###      ##
        #      ##
        #     #
     # ####    #
     # ###  ####
      ########
```

```
=== Features with high weight for label 9 ===


          ###
         ###
         ## # #####
         # # #
          ## #     ##
         ####     ###
          ###   ####
          ##########
           ######
       ####    # #
       #####
       # ##  #



             #
           ###
          #####   ##
          # ### ##
          ##  ####
      #  ###     #
        ###      ##
```

Exercise 5

a. The naives Bayes classifier is in theory a suitable instrument for classifying digits. By utilising a proper number of features the classifier is capable of handling the rather simplicity of digit recognition in a nonlinear way. However the assumption that all feature variables are independent of the class variable is rather naive. The features of certain digits are certainly not independent. Compare for instance some of the similar features of a 1 and 7. The other problem that might arise is overfitting on the training data. These issues are mitigated by the fact that digit classification is more simplistic compared to for instance face recognition. The differences between the training data and test data will almost surely be slimmer considering the limited number of ways to write a certain digit.
b. The command python dataClassifier.py -d digits -c naiveBayes -o -1 4 -2 2 runs a naive Bayes classifier (-c) on the default digits dataset (-d). This '-o -1 4 -2 2' part of the command computes a odd ratio analysis between two labels; in this case between 4 and 2. The odds are calculated by computing the frequency that if a digit is labeled as 4 how often are its features classified as 2. The same frequency is calculated for the instance in which it is labeled as 4 and its features not classified as 2. The odds ratio is the ratio between these two. It is being giving as the features (pixels) in a text picture.
The features it gives as output can be seen as the pixels which are more likely corresponding to 4 as opposed to 2.
c. **I.** The number of blank (0) pixels in a digit. We calculate the number of regions which contain only 0 pixels by taking the first pixel and search for all its neibours and remove these from a list. We increase the number of regions by one and repeat it for any remaining pixels. This would give the numbers 1,2,3,5,7 only one region; the digits 4,6,9,0 two regions and 8 three regions. We thought the classifier would be better suited to differentate between 6, 8 and 9 this way.
II. The edges of each digit. We find and store the hard edges of the digits to show the classifier the shape of each digit. This can be calculated by comparingeach digit with the neighbour next to it and storing it as a 1 if they differ. The shapes of each digit are more or less unique thus making this an excellent featureset.
III. Empty rows. We find every row in a digit which does not contain any pixels > 0. The thought process was that this would decrease the amount of false positives to certain digits. It did not.
IV. A row of pixels > 0 which is interrupted by a certain number of blank pixels. Easily implemented with a list and counter. It would store each row and give it a binary value of 1 if the row is interrupted. This would differentiate between almost all digits. Especially the errors between 4 and 9 lessen because a lot of fours are open on top thus giving a clear difference in the features.
V. The number of pixels in a digit that are > 0. This is easily countable. We stored it as member of a percentage. We assumed that the surface area of digits differ greatly. This wound up not making any difference no matter what steps of percentage we used.
d. We coded all five features mentioned above but only implemented the edge detection, empty rows and interrupted rows.
The test performance of our classifier is 84.0%, with a validation performance of 83.0%.

Disabling the features (-f) we implemented gives a test performance of only 78.0%, with a validation performance of 82.0%.
Removing any of the features we implemented at all decreases the test performance by at least 1.0%. The amount of correct guesses are higher. The increase in features did not overfit the data. This would give a higher validation but lower test performance. We can therefore assume the implemented features give the classifier an improved performance.

Exercise 6

a. For the perceptron classifier trained in Q5, the baseline performance is:
   Validation score: 88/100
   Testing score: 84/100
b.
   i. The behavior of the StopAgent is easy to clone: the action is provided as an argument of the function enhancedPacmanFeatures,so we can simply add a feature “stop” that is 1 if the action is stop and 0 otherwise. This captures the behavior of StopAgent well because it stops whenever possible.
   ii. To be able to clone the behavior of FoodAgent it is useful to know which actions let Pacman eat a food dot, and what the distance to the nearest food dot is in the next state. The two features combined should give our agent a good idea about where to find food to eat.
   iii. To behave like SuicideAgent our agent needs to be able to find ghosts. The isLose function is useful to find out which actions let the agent run into a ghost. Another useful feature that is useful is the distance from the position of Pacman in the next state to the nearest ghost.
   iv. Most of the features discussed above are useful for ContestAgent as well. Two features that might help as well describe if taking an action lets Pacman consume a capsule, and the distance to the nearest capsule. We also added a feature with the score of the state as value, since ContestAgent will often prefer action that increase the score.
c. We have implemented all the features mentioned above, with the exception of the “stop” feature, since the StopAgent was easily cloned with just the default features. We used BFS to determine the distance to the nearest food dot, capsule and ghost. Simply adding the distance as a feature didn’t work very well. For instance when the nearest ghost is 10 steps away it isn’t a problem to move closer to it for ContestAgent, but if the ghost is 2 steps away it is. To avoid this problem, the distance features are of the form (“distanceToX”, i), where (“distanceToX”, i) = 1 if i is smaller than the distance to x. When we state something like disable “distanceToX” we refer to disabling (“distanceToX”, i) for all i. We will use the classifier to train each of the four agents:
   i. StopAgent: The classifier achieves a 100% score for both validating and testing, with or without any added features.
   ii. FoodAgent: With all features enabled, the verification score is 88.7% and the testing score is 88.4%. With all features disabled, the results were 80.1%/80.5%.

We also ran the classifier with one feature disabled to determine the effect of this feature on the end result. For FoodAgent, the features with a significant effect (2% or more difference on both scores) were “foodEaten”, “distanceToCapsule”, “distanceToFood” and “score”. Surprisingly, the classifier performed better with “foodEaten” or “score” disabled: 93.9%/91.3% and 92.2%/93.2% respectively. An explanation could be that eating the nearest food dot is often not the best long term strategy. This might also explain why “distanceToCapsule” has an effect on the performance of the classifier: disabling this feature provides a score of 83.8%/84.7%, so apparently this information is useful for the classifier. It is often a good strategy to collect the food dots near the nearest corner first, to avoid having to come back later; the distance to the nearest capsule is the distance to the nearest corner. Finally, disabling “distanceToFood” hurt the score as well: 84.7%/82.9%.

iii. SuicideAgent: With all features enabled the score is 89.04%/99.0%, without features 68.7%/72.5%. Only one feature has a relevant effect, namely “distanceToGhost”. Without this feature, the classifier scores 69%/79.4%. There are only 102 instances in the testing set of this agent, which might explain the large difference between the verification and testing score. Note that the “lose” feature, which returns 1 if Pacman reaches a terminal state where it loses and 0 otherwise, does not seem to have an effect on the performance. It’s possible that the case where this feature is 1 happens too rarely for it to have an significant effect during training.

iv. ContestAgent: Without features the score is 79.8%/82.8%, with features enabled 93.4%/95%. Somewhat surprisingly, only two features significantly changed the performance of the classifier when disabled, namely “distanceToFood” and “score”. The former matters the most, disabling “distanceToFood” sets the score to 80.3%/82.5%. Finding food dots is apparently very important for the strategy of ContestAgent. Without the “score” feature the score is 85.2%/88.2%.

Exercise 7

Autograder scores:

Q1	Q2	Q3	Q4	Q5	Q6	Total
4/4	1/1	6/6	6/6	4/4	4/4	25/25