Supervised Learning - Coursework II

Antonios Anagnostou, Kefei Hu
{antonios.anagnostou.18, kefei.hu.18}@ucl.ac.uk

December 14, 2018

Part I

Introduction of Kernelised Perceptron with Dual Form

In a general binary linear problem such as $\hat{y} = sign(wx)$, the weight vector w can be expressed as a linear combination of $x_i \in S_m$.

$$\begin{aligned} w &= \sum_{i=1}^{m} \alpha_i y_i x_i \\ \hat{y_t} &= sign(wx_t) \\ \hat{y_t} &= sign(\alpha_i y_i x_i x_t) \\ \hat{y_t} &= sign(\alpha_i y_i < x_i, x_t > \end{aligned}$$

Replacing x by $\phi(x)$ we obtain the dual form with Kernel:

$$\hat{y_t} = sign(\alpha_i y_i K(x_i, x_t))$$

In the computation of 'just a little bit problem', dual form is adopted for convenience, since Gaussian Kernel is involved. Details on how this binary dual form is generalized to multi-class can be found in section 'Understanding multi-class prediction with a'.

The role of coefficient α in Dual Form Percetpron

Consider the classifier for a paritcular class, k. $\alpha^{(k)}$ is a vector of size (m). As shown by the formula of prediction in dual form, $\alpha_i^{(k)}$ is the coefficient associated with the Kernel of x_i and new data x, for class k.

During training, $\alpha^{(k)}$ is only updated when the perceptron fails to predict k for training data with true label being k. Denote $\alpha_t^{(k)}$ to be the coefficients for class-k that are already trained on t samples. Suppose perceptron makes a mistake on x_{t+1}, that instead of predicting k it incorrectly predicts k'. $\alpha^{(k)}$ is updated in the following way:

$$\alpha_{t+1}^{(k)} = \alpha_t^{(k)} + 1$$

Given a mistake, it becomes likely that the coefficients associated with class k' is too high, hence it must be penalized. $\alpha_{t+1}^{(k')} = \alpha_t^{(k')} - 1$.

To generalize the role of $\alpha_t^{(k)}$ for any class k, $\alpha_t^{(k)}$ can be also regarded as a counter of mistakes associated with class k after reviewing t training samples:

$$\sum_{i=1}^{t} I\big[y_i = k \bigcap \hat{y}_i \neq k\big] - I\big[y_i \neq k \bigcap \hat{y}_i = k\big]$$

In our algorithm, $\alpha_t^{(k)}$ for all 10 classes are stored in a matrix denoted by α of size $(m, 10)$.

Intuition of multi-label perceptron in dual form

There are two approaches to convert Binary Perceptrons into Multi-label perceptrons: One vs. All and One vs. One. This section provides more insight on how the coefficients α, updated such way in the previous section affect the algorithm in either approahces.

One vs. All

To generalize binary linear classifier to multi-class, consider building a linear classifier for all 10 digits, and take the class with maximum of confidence. Denote $C_t \in R^10$ to be the confidence of prediction on x_t for every class. If x_i has label $= k$, $y_i^{(k)} = 1$.

$$C_t = (\sum_{i=0}^{t-1} \alpha_i K(x_i, x_t))$$
$$\hat{y}_t = Argmax_k C_t^{(k)}$$

We already know that $\alpha_{(t,k)}$ is a counter of mistakes associated with class k so far at stage $= t$.

$C_t^{(k)}$ for class k increases by an amount of $K(x_i, x_t)$ if the perceptron has failed to correctly predict x_i when $y_i = k$. In cases of Polynomial and Gaussian Kernel, $K(x_i, x_t)$ increases as the similarity between x_i, x_t increases. Hence, the more similar our new data x_t is to x_i, higher the perceptron's confidence to predict $\hat{y}_t = k$.

Similarly, If the perceptron has mistaken x_i as a point in class k, and x_i, x_t are very similar, it is unlikely that x_t is also in class k, hence $C_t^{(k)}$ decreases accordingly.

Details on how to implement the OvA perceptron will be outlined in 'Multiclass Perceptron algorithm - One vs. All'.

Multiclass Perceptron algorithm - One vs. All

As we have seen the intuition of Kernalised perceptron with OvA approach in the previous section, we will now proceed to define the training and testing procedures for this setup.

Training the perceptrons

1. Create a mapping matrix called y_arr of size (m_{train},10), using y_{train}. $y\_arr[t,:]$ is 1 for y_{train} else -1.
2. Initialize a weight matrix α, of dimensions $(m_{train}, 10)$.
3. Initialize also an array for storing the mistakes made by the perceptron, as it is being iteratively trained. Let also *error_current* be the error rate observed so far by the algorithm.
4. Calculate the kernel matrix $K(X_{train}, X_{train})$ for the training set.
5. Do:
   (a) Do a full *epoch* (definition of an epoch procedure to follow) of the data set and obtain the update weights *alpha* and number of mistakes made.
   (b) Calculate $error\_next = mistakes/number\_of\_examples$

   Until $error\_next - error\_current < 0.01$
6. Return the weights matrix α and the error rates observed in every epoch

In every *epoch*, the algorithm performs the following steps:

1. For every $x_t \in S_{m_{train}}$:
   (a) Calculate the prediction confidences for every label, by calculating the quantity $\alpha^T K(X_{train}, x_t)$.
   (b) Take the argmax of this confidence vector. Create a vector of $\hat{y_t}$ of size 10, where $\hat{y_t}$ is 1 for the class with the highest confidence, 0 elsewhere.
   (c) If the $\hat{y_t} \neq y_t$:
      i. Update the weights α_t using the following formula: $\alpha_t = \alpha_{t-1} + where(y\_arr_t > 0, 1, 0) + where(\hat{y_t} > 0, -1, 0)$ Note that the *where* operator here is a transformation of the real and predicted labels so that they are in the $0, 1^n$ and $0, -1^n$ spaces accordingly. This update will result in penalizing the classifiers which have been mistaken about t.
2. Return the update weights matrix α and the number of mistakes done.

Table 1: Basic results for Perceptron (Polynomial kernel)

d	Training set error rate	Test set error rate
1	0.0807 +- 0.0210	0.1003 +- 0.0229
2	0.0153 +- 0.0042	0.0470 +- 0.0054
3	0.0074 +- 0.0031	0.0403 +- 0.0052
4	0.0058 +- 0.0036	0.0358 +- 0.0055
5	0.0032 +- 0.0016	0.0325 +- 0.0043
6	0.0025 +- 0.0013	0.0331 +- 0.0044
7	0.0074 +- 0.0235	0.0394 +- 0.0237

Using the perceptrons to predict

Given a set of n perceptrons trained in a set X_{train}, we can now use them to predict an unlabelled set of data X_test.

1. Calculate the kernel matrix $K(X_{train}, X_{test})$.
2. For every example t in the unlabelled set:
   (a) Calculate the prediction confidences for every label, by calculating the quantity $\alpha^T K_t$.
   (b) Translate the confidences into a vector of $0, 1^n$, where 1 denotes that label j has the maximum confidence for this example. Let the predicted label be $\hat{y_t}$.
3. Return the predictions $\hat{y}$.

Question 1. Basic Results, O.v.A Polynomial

For a basic set of results for the perceptron with polynomial kernel, we will perform the following process:

- Initialize two arrays for training and test set error rates respectively.
- For every run out of 20:
  - For every value of d in the specified range $[1, 7]$:
    - Split the data set randomly into 80% and 20% for training and testing respectively.
    - Train the perceptron in 80% of the data.
    - Predict on the training set to retrieve the training error rate after the perceptron has been trained.
    - Predict on the test set to retrieve the test error rate.
    - Record these two error rates in the two arrays specified above.

Table 1 contains the mean training and test error rate, as well as the standard deviations, as observed of a series of 20 runs for the kernel perceptron, using d value in range $[1, 7]$.

Table 2: Confusion matrix for Perceptron (Polynomial kernel)

	0	1	2	3	4	5	6	7	8	9
0	0.00 +- 0.00	0.00 +- 0.00	0.02 +- 0.02	0.01 +- 0.02	0.01 +- 0.01	0.01 +- 0.01	0.01 +- 0.01	0.00 +- 0.01	0.01 +- 0.01	0.00 +- 0.01
1	0.00 +- 0.00	0.00 +- 0.00	0.00 +- 0.01	0.00 +- 0.00	0.03 +- 0.04	0.00 +- 0.00	0.01 +- 0.01	0.00 +- 0.01	0.01 +- 0.02	0.00 +- 0.01
2	0.01 +- 0.02	0.00 +- 0.01	0.00 +- 0.00	0.02 +- 0.02	0.02 +- 0.02	0.00 +- 0.01	0.01 +- 0.01	0.02 +- 0.02	0.01 +- 0.02	0.00 +- 0.01
3	0.01 +- 0.01	0.00 +- 0.01	0.03 +- 0.02	0.00 +- 0.00	0.00 +- 0.00	0.05 +- 0.02	0.00 +- 0.00	0.01 +- 0.01	0.03 +- 0.03	0.00 +- 0.00
4	0.00 +- 0.01	0.02 +- 0.02	0.01 +- 0.02	0.00 +- 0.00	0.00 +- 0.00	0.00 +- 0.01	0.01 +- 0.02	0.01 +- 0.02	0.01 +- 0.01	0.03 +- 0.03
5	0.02 +- 0.02	0.00 +- 0.01	0.01 +- 0.01	0.05 +- 0.04	0.01 +- 0.01	0.00 +- 0.00	0.02 +- 0.02	0.00 +- 0.01	0.02 +- 0.02	0.01 +- 0.01
6	0.02 +- 0.02	0.01 +- 0.01	0.01 +- 0.01	0.00 +- 0.00	0.01 +- 0.01	0.01 +- 0.01	0.00 +- 0.00	0.00 +- 0.00	0.00 +- 0.01	0.00 +- 0.01
7	0.00 +- 0.00	0.01 +- 0.01	0.01 +- 0.02	0.00 +- 0.01	0.01 +- 0.02	0.00 +- 0.00	0.00 +- 0.00	0.00 +- 0.00	0.01 +- 0.01	0.03 +- 0.02
8	0.01 +- 0.02	0.01 +- 0.02	0.01 +- 0.02	0.04 +- 0.02	0.01 +- 0.02	0.03 +- 0.02	0.01 +- 0.01	0.02 +- 0.02	0.00 +- 0.00	0.01 +- 0.01
9	0.00 +- 0.01	0.01 +- 0.01	0.00 +- 0.01	0.00 +- 0.01	0.03 +- 0.02	0.00 +- 0.01	0.00 +- 0.00	0.04 +- 0.03	0.00 +- 0.01	0.00 +- 0.00

Question 2. 5-Fold Cross-Validation, O.v.A Polynomial

For this question, we performed 20 runs of the following algorithm:

- Initialize two arrays, one for holding the test set error rates and one for storing the best values of d.
- For every run out of 20:
  - We randomly divide the data set into 80% and 20%, for training and testing respectively.
  - We perform a 5-Fold Cross-Validation on the 80% of the data, evaluating different values of d.
  - Upon finding the optimal value $d*$, we train our classifier in the entirety of 80% of the data set.
  - We then try to predict on the remaining 20% and record the test error rate as well as the value d*.
- Return the test error rates observed and the set of $d*$ values.

Using the process defined above, we observe the following results when doing 5-Fold cross validation over 20 runs:

- Mean d*: 5.4 +- 0.86
- Mean test error: 0.0335 +- 0.0049

Question 3. Confusion Matrix, O.v.A Polynomial

In order to calculate the confusion matrix, we will add one more step in the CV process which was defined in the previous question. That is to say, the process we will use is the following:

- We initialise an empty matrix which will contain the confusions over the twenty runs (20x(10x10) matrix)
- For ever run out of 20:
  - We initialize a new 10x10 matrix to zeros for the confusions recorded in this run.
  - We randomly divide the data set into 80% and 20%, for training and testing respectively.
  - We perform a 5-Fold Cross-Validation on the 80% of the data, evaluating different values of d.

- Upon finding the optimal value $d*$, we train our classifier in the entirety of 80% of the data set.
- We then try to predict on the remaining 20% and record the test error rate as well as the value d*.
- We iterate through the examples of the test set (20% of the data) and for every misclassification we will be incrementing the corresponding cell ($real\_label, predicted$) by $1/mistakes$. This will ensure that the confusions of this run will be proportionate to the total number of mistakes in this run, i.e. what is the contribution of mistaking digit i with j in the number of mistakes we did in this run. *(A more naive approach would be to just increment by 1, but according to the guidelines of this assignment, errors should be reported in percentage and not totals)*

Table 2 contains the mean confusion rates and the standard deviation as recorded using the process above. We note that some of the highest confusions are mistaking the digit 3 with 5 (5% of mistakes on average +- 2%), the digit 5 with 3 (5% of mistakes on average +- 4 %), 9 with 7 (4% of mistakes on average +- 3 %), 1 with 4 (3% of mistakes on average +- 4 %) and 4 with 9 (3% of mistakes on average +- 3%).

Question 4. Hardest-to-predict, O.v.A Polynomial

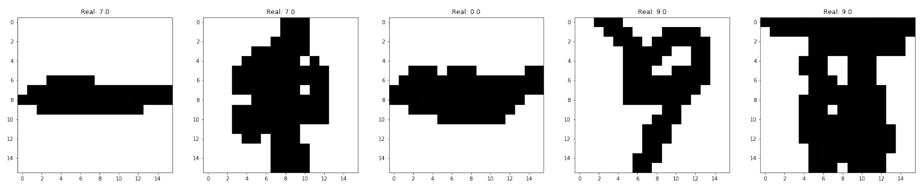

Figure 1: Hardest to predict elements in the data set

In order to find the hardest elements in the data set to predict, we will execute the following procedure 20 times:

1. Initialize a least_confidences array which will be of dimensions (number of examples X 1)
2. Using a 5-Fold split, divide the data set into 80% and 20% accordingly.
3. For each fold, train a polynomial perceptron on the training set using the mean of $d*$ as calculated in the previous question and then predict the labels of the test set.
4. For every example in the test set, increment the confidence in the least_confidences array by the confidence of the predicted label, divided by k=5 (so as to obtain the mean of confidences at the end of the 5-Fold process).
5. At the end of the 20 runs, also divide the least_confidences array by 20 to have the mean confidences.

Table 3: Basic results for Perceptron (Gaussian kernel)

d	Training set error rate	Test set error rate
1	0.0000 +- 0.0000	0.0603 +- 0.0036
2	0.0000 +- 0.0000	0.0624 +- 0.0050
3	0.0000 +- 0.0000	0.0606 +- 0.0043
4	0.0000 +- 0.0000	0.0646 +- 0.0050
5	0.0000 +- 0.0000	0.0639 +- 0.0051
6	0.0000 +- 0.0000	0.0741 +- 0.0058
7	0.0000 +- 0.0001	0.0963 +- 0.0060

6. Sort the array of least_confidences in ascending order and return the indices of the first five elements.

Figure 1 contains these five examples and their corresponding labels. As observed, none of these examples are easy to comprehend even by a human, so observing least confidences in our predictions is to be expected.

Question 5.i. Basic results, O.v.A Gaussian

To obtain some basic results for the Gaussian kernel of the kernel perceptron, we have followed the same process as described in length in Question 1.

Table 3 contains the mean training and test error rate, as well as the standard deviations, as observed of a series of 20 runs for the kernel perceptron, using d value in range $[1, 7]$.

We observe that the Gaussian Kernel seems to be overfitting on the training data, which leads as a result to a higher test error rate, than the Polynomial Kernel.

Question 5.ii. 5-Fold Cross-Validation, O.v.A Gaussian

Using the process defined in Question 2, we observe the following results when doing 5-Fold cross validation over 20 runs:

- Mean d*: 2.4 +- 0.9165
- Mean test error: 0.0621 +- 0.0047

Similar to the observations above, even in the 5-Fold Cross-Validation setup, the Gaussian Kernel seems to be underperforming in comparison with the Polynomial Kernel, in which we observed a mean test error rate of 0.0335 +- 0.0049.

Question 6. Alternative generalization method, O.v.O

The one-vs-one approach for multilabel perceptron classification

As we have mentioned earlier on, an alternative approach to one-versus-all classification is the one-versus-one, i.e. O.v.O. This approach requires training 45 binary classifiers to distinguish between each pair of classes. The binary classifier between any pair of classes (k_1, k_2) makes prediction as the following:

Table 4: Basic results for one-vs-one Perceptron (Polynomial kernel)

d	Training set error rate	Test set error rate
1	0.1298 +- 0.0100	0.1383 +- 0.0128
2	0.0621 +- 0.0066	0.0819 +- 0.0092
3	0.0370 +- 0.0036	0.0634 +- 0.0067
4	0.0264 +- 0.0030	0.0583 +- 0.0066
5	0.0211 +- 0.0026	0.0552 +- 0.0051
6	0.0169 +- 0.0040	0.0521 +- 0.0079
7	0.0140 +- 0.0027	0.0506 +- 0.0060

$$\hat{y_t} = \begin{cases} k_1 & sign(\sum_{i=0}^{t-1} \alpha_i y_i K(x_i, x_t)) = 1 \\ k_2 & sign(\sum_{i=0}^{t-1} \alpha_i y_i K(x_i, x_t)) = -1 \end{cases}$$

If the perceptron has mistaken x_i as a point in class k_1, and x_i, x_t are very similar, $\sum_{i=0}^{t-1} \alpha_i y_i K(x_i, x_t)$ is reduced by $K(x_i, x_t)$. This reduces its 'chances' of being recongised as k_1.

The training procedure follow a similar pattern with the ones described in the one-vs-all approach, except that in this case, we have a total of 45 binary classifiers as opposed to only 10. At prediction time, we use all 45 classifiers to predict the class of the example $t+1$ that we receive. Afterwards, we use a voting procedure, during which we take the summation of classifiers which have predicted each class. The class which has received the maximum votes will be the predicted label $\hat{(y_{t+1})}$

Basic results

Table 4 contains the mean training and test error rate, as well as the standard deviations, as observed of a series of 20 runs for the one-versus-one perceptron, using d value in range $[1, 7]$.

Comparing the results with the one-vs-all polynomial kernel perceptron, we observe that both the training and test set error rates seem to be higher in this case, even for higher values of d.

5-Fold Cross-Validation

Using the process defined in Question 2, we observe the following results when doing 5-Fold cross validation over 20 runs:

- Mean d*: 6.15 +- 0.909
- Mean test error: 0.04935 +- 0.00445

Question 7. Kernel Perceptron vs. other algorithms

We have chosen to compare the kernel perceptron to Softmax Regression and Support Vector Machines. In the following subsections, we will present the set up for each of these two algorithms, as well as their time complexities and test errors and we will later on compare the three algorithms.

Softmax Regression

For Softmax Regression, we have have used the implementation of *scikit-learn*. Since implementation of classification algorithms can vary with different implementations of logistic regression, we will now describe the arguments used in scikit-learn's implementation, which describe the overall setup of the logistic regression. Those are:

- penalty='l2': We use a standard l2 regulariser for our classifier. It should be noted that scikit-learn's documentation actually states that in the multinomial setup, the available solvers only support 'l2' penalty setup.
- dual=False: Since the number of examples are more than the number of features, we will not be using the dual form formation.
- multi_class='multinomial', since we are concerned about multi-class classification. This is a key factor for our setup, as if the keyword 'multinomial' is specified, then the softmax function is used to find the predicted probability of each class.
- C: This is the inverse of regularization strength, i.e. smaller values specify stronger regularization. We will treat this as a hyperparameter and compute results for different c values in the range [0.001, 0.01, 0.1, 1, 10, 100, 1000]
- solver='newton-cg': In order to minimise the cost function, a "solver" algorithm must be used. The choice of solver varies in different libraries and implementations. Scikit-learn offers a variety of different solvers available for this optimisation problem, but specifically in the case of multinomial classification, the applicable solvers are 'sag', 'saga' and 'lbfgs'.
- max_iter=100: The maximum number of iterations that the solver can perform while trying to converge.
- tol=1e-4: This is the threshold value used as a stopping criterion.

Time complexity and choice of the solver algorithm Since Softmax Regression is an optimisation problem, different implementations and libraries use different solving algorithms. In this section we will try to describe in a bit more detail of each of the solvers available in scikit-learn package:

- **Newton-CG**: Newton's method for minimisation combines the method of Gradient Descent with the Hessian matrix. This solver uses both the first and second partial derivatives the algorithm will try to converge towards the minimum of the cost function. A caveat is that this algorithm is prone to saddle points, i.e. points which are not local extremes of the function. For a more in-depth analysis of this method as well as complexity analysis, please refer to [1].
- **Limited-memory Broyden–Fletcher–Goldfarb–Shanno Algorithm**: Similar method to Newton-CG except for using approximations for the Hessian matrix, so as to reduce memory and computational complexity.

Table 5: Basic results of Linear Regression

C	Training set error rate	Test set error rate
0.001	0.0817 +- 0.0014	0.0847 +- 0.0054
0.01	0.0510 +- 0.0009	0.0635 +- 0.0054
0.1	0.0285 +- 0.0009	0.0551 +- 0.0033
1	0.0088 +- 0.0011	0.0644 +- 0.0063
10	0.0007 +- 0.0002	0.0753 +- 0.0057
100	0.0001 +- 0.0001	0.0834 +- 0.0047
1000	0.0000 +- 0.0000	0.0864 +- 0.0052

- **Stochastic Average Gradient (SAG)**: The main difference of this solver when compared to the standard stochastic gradient descend is that it incorporates some memory of previous gradient values, which helps to achieve faster convergence. [3]
- **SAGA**: A variation of SAG, which also support $L1$ regularisation.

We should note that scikit-learn documentation [8] recommends SAGA for large datasets. In practice, we have observed that SAGA, even after normalising the data, results in non-convergence errors. For this reason and so as to aim for the best possible approximation, we preferred the **Newton-CG solver**. Since the solving algorithm is the main source of time complexity in softmax regression, it is not possible to provide an overall bound in terms of *O()* notation. Individual algorithms can be analysed in their respective papers, as cited above, but as the number of operations might depend on the number of iterations, threshold values and other possible parameters of the solver, this goes beyond the scope of this experiment.

In the experiment below, we will evaluate C as hyperparameter and try all values in the range [0.001, 0.01, 0.1, 1, 10, 100, 1000] in two setups:

1. We will perform 20 runs of going through all the possible values of C, divide the data set into 80% and 20% randomly, and then using the first set to train the LR classifier. We will be recording the classification error as observed for every combination of C, in these 20 runs, on the test set.
2. We will perform 20 runs of going through all the possible values of C, divide the data set into 80% and 20% randomly, and then performing a 5-fold cross-validation to find the best possible C value. Then, we would train the classifier on the whole 80% of the data set using this optimal value of C and will be recording the classification error as observed on the test set.

performing a 5-fold cross validation over 20 runs to find how each of them perform as well as the best combination over 20 runs.

Basic results of Softmax Regression Table 5 contains the training and test error rates for different values of C, as described in the experimental setup above. We observe that for $C = 0.1$, we observe a "sweet" spot where test set error rate is at its lowest and training set error rate is at a low enough value.

Results of CV setup for Linear Regression Using the process defined above, we observe the following results when doing 5-Fold cross validation over 20 runs:

- Mean test error: 0.0548 +- 0.0043
- Cost value: 0.10 +- 0.00

Support Vector Machines

Support Vector Machines were originally introduced by Cortes and Vapnik in 1995 [5] and have been widely used ever since in the field of machine learning. We will use SVMs to compare their performance with the kernel perceptron. For a formal mathematical definition of the problem as well as an in-depth analysis, please refer to [4].

For this experiment, we will be using the implementation of SVM (SVC) found in *scikit-learn*. Similar to our approach with softmax regression, we will now begin to analyse the specific parameters used in our classification approach with SVMs. Those are:

- C: This parameter controls the amount of error that the SVM is allowed to do in the training error. As described in [4], in Equation (4), C is a quantity that controls the compromise which is introduced by allowing some slack variables to exist, i.e. examples of the training set which are misclassified when finding the separating hyperplane. We will treat this quantity as a hyperparameter and evaluate the test set error rate for C in the range of [0.1, 1, 10, 100, 1000].
- kernel: This is the choice of kernel for the SVM classifier. The available kernels in the scikit learn library can be found in the library's documentation [7]. We will treat this as a hyperparameter as well and evaluate two possible values:
  - 'poly' (polynomial kernel): $(\gamma\langle x, x'\rangle + r)^d$
  - 'rbf' (Gaussian): $\exp(-\gamma\|x - x'\|^2)$
- gamma='scale': This is the value observed in the kernels above. The default value of parameter γ in scikit-learn's SVM package is 'auto', which is $1/n$, where n is the number of features. Newer versions of this library also introduce the value 'scale', whose purpose is to reduce the parameter's dependence on the scale of X. To reduce complexity, we do not treat γ as a hyperparameter, but we use 'scale' which would set the parameter to be $1/(n * std(X))$. It should be noted that there is actually open discussion whether this quantity is correct for γ as contributors of the library have argued that gamma should instead be calculated according to variance rather than standard deviation. [1]
- degree=3: This only applies to the polynomial kernel and denotes the order of polynomial used for the kernel function (see also above the mathematical definition of the polynomial kernel).
- tol=1e-3: This is the threshold value used as a stopping criterion.
- max_iter=-1: We do not specify a maximum iteration number for the solver of our optimisation function.

[1]https://github.com/scikit-learn/scikit-learn/issues/12741

Table 6: Basic results of SVM using polynomial kernel

C	Training set error rate	Test set error rate
0.01	0.1717 +- 0.0040	0.1737 +- 0.0066
0.1	0.0380 +- 0.0009	0.0485 +- 0.0035
10	0.0002 +- 0.0001	0.0211 +- 0.0027
100	0.0001 +- 0.0000	0.0207 +- 0.0034
1000	0.0000 +- 0.0000	0.0210 +- 0.0029

Table 7: Basic results of SVM using Gaussian kernel

C	Training set error rate	Test set error rate
0.01	0.2166 +- 0.0040	0.2241 +- 0.0123
0.1	0.0477 +- 0.0012	0.0563 +- 0.0059
10	0.0004 +- 0.0001	0.0224 +- 0.0018
100	0.0001 +- 0.0001	0.0228 +- 0.0036
1000	0.0000 +- 0.0000	0.0233 +- 0.0039

Time complexity of SVMs Bottou and Lin [4] show that the training examples in the SVM problem can be split into the below categories: examples which are not support vectors, examples which are *bounded* support vectors (i.e. they appear in the discriminant as $a_k = C$) and *free* support vectors (i.e. they appear in the discriminant with $a_k \in [0, C]$).

To analyse the complexity of SVMs, let's consider the following observations:

- If we are given the knowledge of which examples in our training set constitute support vectors then we can find out the weights for the remaining R free support vectors, by solving a linear system of R equations. The latter are derived by obtaining the derivatives of the loss function. Typically this can be done with complexity of $O(R^3)$.
- In order to verify that this is indeed a solution for SVM, we need to compute the gradient of the dual form and validate the conditions of optimality as set in the mathematical definition of the problem. Therefore, if our data set comprises n examples, out of which S are support vectors, this set of operations would cost $O(nS)$.

The authors also show that the number of support vectors grows asymptotically linearly with the number of examples. Therefore, based on the observations above, the computational cost of SVM is either $O(n^3)$ for large numbers of C or $O(n^2)$ for smaller values.

Having defined the parameters and hyper-parameters used in the SVM classifier, we will proceed by performing the same type of experiment as in softmax Regression.

Basic results for SVM Table 6 and 7 present the mean training and test error rates as observed for the polynomial and gaussian kernel experiments and various values of C, over 20 runs. We observe, that even for higher values of C, there is no significant increase in test error. That being said, for $C = 10$, we observe the lowest combination of test error rate and variance (i.e. for higher values of C, variance increases).

Results of CV setup for SVM classifiers Using the process defined above, we observe the following results when doing 5-Fold cross validation over 20 runs:

- Mean test error: 0.0221 +- 0.0033
- C (in log10 scale): 1.55 +- 0.7399
  Interpretation: For 12/20 runs, we have observed C* value=10. For 5/20 runs, we have observed C*=100. For 3/20 runs, we have observed $C^* = 1000$.
- Kernel: For 18/20 runs, we have observed better values of test error using the polynomial kernel and respectively for 2/20 runs we have observed better values while using the Gaussian kernel.

Comparing accuracy of three algorithms

Table 8 summarises the mean test error rates observed in the different classification algorithms examined in this paper. These are the error rates as calculated, over a series of 20 runs, in the randomly allocated 20% test set, after using the cross-validation process defined in Question 2.

As we can see, the SVM seems to outperform the other methods, with the polynomial O.v.A. perceptron being close in the difference of mean test error rates. This aligns with the results of the results of the cross-validation process of SVM, as we have observed that in the majority of the runs, the CV process preferred a polynomial kernel of degree 3 as opposed to a Gaussian kernel. Therefore, both methods tackle the classification issue with a polynomial of degree 3.

Discussion on result

The reason of SVM being a better performer than Perceptron in this case lies within their objectives.

SVM: Minimise $\parallel w \parallel_2$, and minimize $\sum_{i=0}^{m} 1 - y_i(wx_i + b)$

Perceptron: Minimize $\sum_{i=0}^{m} -y_i(wx_i + b)$

A noticeable similarity between SVM and Perceptron is that they both include hinge loss in their objective to minimize. But SVM has a quadratic objective due to the L2 Regulariser. By applying a quadratic optimiser on SVM, the margin of the seperating hyperplane, $\gamma = \frac{1}{\|w\|}$ is also maximized, ensuring that the sum of square distances between each pair of points from different sides of the hyperplane is maximum.

On the other hand, the cost function of Perceptron is just hinge loss, and it is updated with stochastic gradient descent with learning rate = 1. E.g.when predicting t^{th} training data, $l_t = max\{0, -y_t(wx_t + b)\}$. The gradient of this loss w.r.t w is $-y_t x_t$, which is the amount that w is updated when perceptron misclassfies x_t. Being updated this way, Perceptron may find any linear hyperplane, as long as it divides the training data such that training mistakes are minimized, but there is no objective to maximise the margin. Hence, SVM is expected to give a more mathematically concrete solution, and generalize better with unseen data.

Softmax Regression is the 3rd worst performer, just behind Perceptron with polynomial kernel. The similarity between Softmax Regression and Perceptron is that they both outputs 'confidence' when they make predictions. One of the main differences of Softmax from Kernelised Perceptron is that its 'confidence' vector computed assumes the conditional distribution of each class, $P(\hat{y}_i = y_i|x_i) = \frac{e^{w_{y_i} x_i}}{\sum_{k=1}^{10} e^{w_k x_i}}$. Another difference is that Softmax Regression considers all n individual features seperately, while Kernelised Perceptron has a mistake-driven approach that considers the similarity between current data point and the data it has seen before. Since there are too many features, Softmax Regression may be more prone to over-fitting, hence does not generalize as well for unseen data as Perceptron.

Comparing time complexity of three algorithms

Table 9 summarizes the time complexities needed for the training of the algorithms mentioned above. In order to be consistent with the rest of this paper, we will denote m to be the number of examples in the training set, n to be the number of dimensions and T to be the number of examples in the test set. A brief analysis of these complexities follows:

- **Perceptron**: For the O.v.A. perceptron, the main cost during the training phase occurs when calculating the kernel matrix. In both the polynomial and the gaussian kernel, this has complexity $O(m^2 n)$ as it involves matrix multiplications of sizes (m, n) and (n, m). Part two of this assignment goes in length of analysing the cost of the perceptron.

  Similar to the training phase, in testing the most costly operation is the calculation of the kernel matrix. If we denote with T the number of examples in the test set, then the complexity becomes $O(mTn)$, as it involves matrix multiplications of sizes (m, n) and (n, T).

- **Softmax Regression**: As we have noted above the time complexity of the training phase relies upon our decision for a solver method. Testing complexity scales linearly with the number of examples in the test set.

- **SVM**: As discussed above and as explained in the paper of Bottou [4], the training complexity of SVMs is either $O(m^2)$ or $O(m^3)$ depending on the value of parameter C. For testing, the complexity can be shown to be dependent on the number of support vectors, i.e. $O(n_{SV} n)$. [6]

Tables 10 and 11 contain some empirical times measured for the basic results and the 5-fold Cross-Validation runs as observed while executing the experiments. In Table 10, we observe that the O.v.A. perceptron with polynomial kernel requires less time for training than the Gaussian kernel perceptron. This is because the Gaussian kernel is computed upon the pairwise-distances of the

Table 8: Comparison of different classification algorithms' test error rates

Algorithm	Test set error rate
Perceptron O.v.A. with polynomial kernel	0.0335 +- 0.0049
Perceptron O.v.A. with gaussian kernel	0.0621 +- 0.0047
Softmax Regression	0.0548 +- 0.0043
SVM	0.0221 +- 0.0033

Table 9: Comparison of different classification algorithms' time complexity

Algorithm	Training complexity	Testing complexity
Perceptron O.v.A. with polynomial kernel	$O(m^2n)$	$O(mTn)$
Perceptron O.v.A. with gaussian kernel	$O(m^2n)$	$O(mTn)$
Softmax Regression	Depends on the solver	$O(m)$
SVM	$O(m^2)$ or $O(m^3)$	$O(n_{SV}n)$

examples found in the training set which requires more computations than in the case of the polynomial kernel.

Table 10: Comparison of empirical time complexity when calculating the basic results

Algorithm	Time observed
Perceptron O.v.A. with polynomial kernel	0:12:53 hrs
Perceptron O.v.A. with gaussian kernel	0:28:48 hrs
SVM with polynomial kernel	0:28:51 hrs
SVM with gaussian kernel	0:32:16 hrs
Softmax Regression	0:30:25 hrs

Table 11: Comparison of empirical time complexity in 5-Fold CV setup

Algorithm	Time observed
Perceptron O.v.A. with polynomial kernel	0:32:27 hrs
Perceptron O.v.A. with gaussian kernel	1:01:32 hrs
Softmax Regression	1:02:16 hrs
SVM (polynomial and gaussian kernel)	2:04:09 hrs

References

[1] Royer, Clément W., Michael O'Neill, and Stephen J. Wright. "A Newton-CG Algorithm with Complexity Guarantees for Smooth Unconstrained Optimization." arXiv preprint arXiv:1803.02924 (2018).

[2] Defazio, Aaron, Francis Bach, and Simon Lacoste-Julien. "SAGA: A fast incremental gradient method with support for non-strongly convex composite objectives." Advances in neural information processing systems. 2014.

[3] Mark Schmidt, Nicolas Le Roux, Francis Bach. Minimizing Finite Sums with the Stochastic Average Gradient. Mathematical Programming B, Springer, 2017, 162 (1-2), pp.83-112.

[4] Bottou, Léon, and Chih-Jen Lin. "Support vector machine solvers." Large scale kernel machines 3.1 (2007): 301-320.

[5] Corters, C., and V. Vapnik. "Support vector network." Machine learning 20 (1995): 273-297.

[6] Claesen, Marc, et al. "Fast prediction with SVM models containing RBF kernels." arXiv preprint arXiv:1403.0736 (2014).

[7] Scikit-Learn documentation for SVM Kernels `https://scikit-learn.org/stable/modules/svm.html#svm-kernels`

[8] Scikit-Learn documentation of Logistic Regression `https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression`