INDIAN INSTITUTE OF TECHNOLOGY, DELHI

REPORT

Assignment 2

ARNAV TULI | ENTRY NO. 2019CS10424

Course - COL774

Prof. Parag Singla

Due October 6, 2021

Contents

1 Text Classification **2**

1.1 Naive Bayes algorithm . . . 2

1.2 Random and Majority prediction . . . 3

1.3 Confusion Matrix . . . 4

1.4 Stopwords removal and Stemming . . . 5

1.5 Feature Engineering . . . 6

1.6 F1-Score . . . 7

1.7 Incorporating Summary for Classification . . . 8

2 MNIST Digit Classification **10**

2.1 Binary Classification . . . 10

2.1.1 Linear Kernel (CVXOPT) . . . 11

2.1.2 Gaussian Kernel (CVXOPT) . . . 11

2.1.3 Linear Kernel (LIBSVM) . . . 12

2.1.4 Gaussian Kernel (LIBSVM) . . . 13

2.2 Multi-class Classification . . . 14

2.2.1 Gaussian Kernel (CVXOPT) . . . 14

2.2.2 Gaussian Kernel (LIBSVM) . . . 14

2.2.3 Confusion Matrices . . . 15

2.2.4 5-Fold Cross Validation . . . 17

Chapter 1

Text Classification

1.1 Naive Bayes algorithm

Before training the model, I processed the text using `nltk` library. Exact procedure:

1. Tokenize each reviewText using `word_tokenize()`
2. Convert all words/tokens to lower case and remove punctuations within words (like ')
3. Filter out non-alphabetical words from the list

Next, I created a vocabulary from the text corpus (training data) and also added an *UNKNOWN* token to handle any unseen word in the test data. After this, I trained a Naive-Bayes classifier with Laplace Smoothing to avoid zero probabilities ($\alpha = 1$).
Before testing the model, I processed the text (test set) using the same procedure as above. I present my observations below.

Observations:
Training accuracy: 70.12 %
Test accuracy: 66.20714285714286 %
Macro F1-score (test data): 0.27155741884672746
F1-score for class 1: 0.13138686131386862
F1-score for class 2: 0.017391304347826087
F1-score for class 3: 0.08563134978229318
F1-score for class 4: 0.31430165519806585
F1-score for class 5: 0.8090759235915834

1.2 Random and Majority prediction

Let m be the number of test examples, and let $X_i \in \{0,1\}$ be the random variable denoting prediction status on i^{th} test example, $1 \leq i \leq m$. $X_i = 1$ for correct prediction and $X_i = 0$ otherwise. Since, in random predictor, a label is chosen from the set $\{1,2,3,4,5\}$ uniformly at random, the $P(X_i = 1) = \frac{1}{5}$ and $P(X_i = 0) = \frac{4}{5}$. Hence, $E[X_i] = \frac{1}{5}$ for all $1 \leq i \leq m$. Now, define X to be the random variable denoting the number of test examples for which the prediction was correct. Then,

$$X = X_1 + X_2 + .. + X_m$$

$$\implies E[X] = \frac{1}{5} + \frac{1}{5} + .. + \frac{1}{5} = \frac{m}{5}$$

Hence, *expected* test accuracy $= \frac{E[X]}{m} = \frac{1}{5} = 20$ %.

Similarly, majority accuracy can be determined by observing that the majority class is class *5*, and that the number of examples of class *5* in the test set is 9252. Hence, majority accuracy $= \frac{9252}{14000} = 66.085$ %.
I also made random/majority predictions and verified the above claims.

Majority accuracy: 66.08571428571429 %	**Random accuracy:** 20.357142857142858 %
Macro F1-score: 0.15916050232238085	**Macro F1-score:** 0.14242848646032663
F1-score for class 1: 0 (zero recall)	F1-score for class 1: 0.030193633081719722
F1-score for class 2: 0 (zero recall)	F1-score for class 2: 0.038101388440426216
F1-score for class 3: 0 (zero recall)	F1-score for class 3: 0.1191983122362869
F1-score for class 4: 0 (zero recall)	F1-score for class 4: 0.21347184986595175
F1-score for class 5: 0.7958025116119043	F1-score for class 5: 0.31117724867724866

Note: My program displays a *zero* F1-Score if precision or recall is 0, even if the other quantity is not defined. (possibly due to zero denominator)

Observations:
The Naive-Bayes algorithm gives a significant improvement over the random model, as 66.2 % is much greater than the expected 20 % in the latter. Also, the algorithm performs better than the majority model, but not by much ($66.2 > 60.08$). This can be explained as the training data is skewed towards class 5, as almost 51 % of the examples correspond to 5 star reviews. Hence, the Naive-Bayes model is unable to effectively learn the data patterns for other classes, and ends up making a lot of incorrect predictions for classes other than 5. This makes the performance of algorithm comparable with the majority prediction paradigm.

1.3 Confusion Matrix

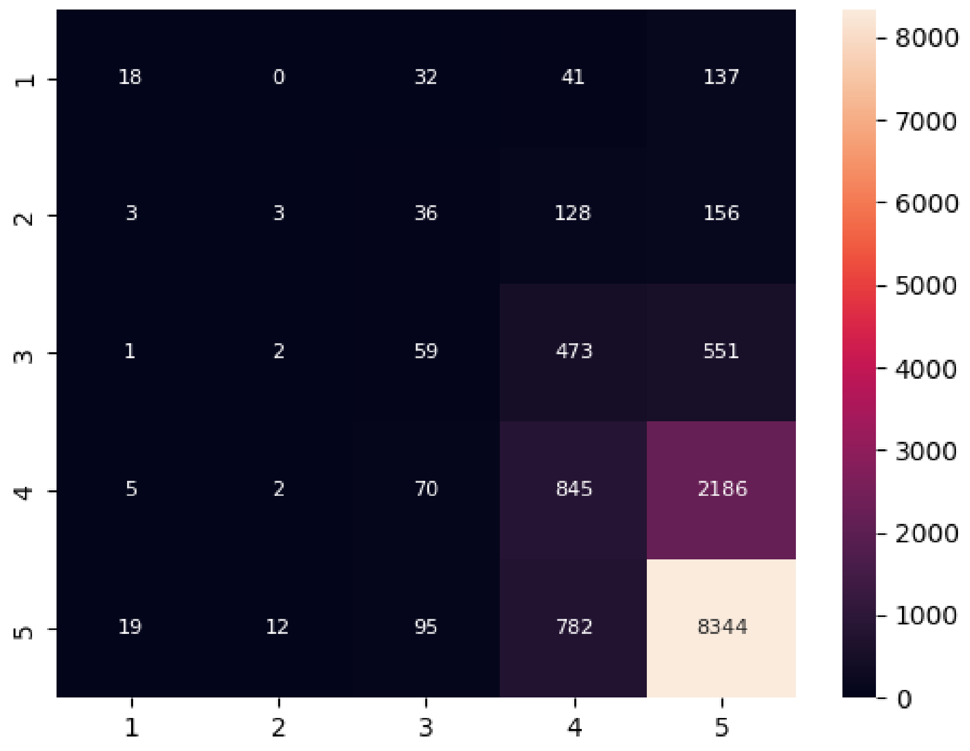


Figure 1.1: Confusion Matrix for 5-class classification (model 1.1)

Observations:

- Class 5 has the highest value of the diagonal entry (8344). This means that most of the correct predictions (true positives) belong to the class of 5 star reviews. Going in decreasing order, next class in number is class 4, then class 3, class 1 and finally class 2, which has the least value of diagonal entry (3).
- For each class, the model predicts the review to be of class 5 for majority of the time. For e.g, 2186 reviews of class 4 are predicted to be of class 5. Similarly, 551 reviews of class 3 are predicted to be of class 5 and so on.
- Observation in the previous point indicates that the model is heavily skewed towards class 5 and consequently has low recall values for other classes. This can be explained as the training data itself is heavily skewed and 51 % of the examples are of class 5. This makes the class prior, or $\phi_5 > 0.5$, which then plays a dominant role in classification, and can potentially overshadow the already less data patterns that are available for other classes, and lead to incorrect prediction. (Less data + less ϕ factor $\implies$ low recall for other classes)
- The highest off-diagonal entry corresponds to fourth row and fifth column (2186). This indicates that the model most often confuses a review of class 4 to be of class 5. This can be

explained as both 4 star and 5 star reviews come under the *good* category, and have almost similar data patterns. This can lead to confusion and more so, when the model itself is skewed towards class 5. Another reason is that there is a large number of reviews of class 4, and hence, the model is confused more often between class 4 and 5, as compared to other classes.

1.4 Stopwords removal and Stemming

Before training the model, I processed the text using `nltk` library. Exact procedure:

1. Tokenize each reviewText using `word_tokenize()`
2. Convert all words/tokens to lower case and remove punctuations within words (like ')
3. Filter out non-alphabetical words from the list
4. Stem words using Porter Stemming algorithm
5. Filter out stop words from the list

Next, I created a vocabulary from the text corpus (training data) and also added an *UNKNOWN* token to handle any unseen word in the test data. After this, I trained a Naive-Bayes classifier with Laplace Smoothing to avoid zero probabilities ($\alpha = 1$).
Before testing the model, I processed the text (test set) using the same procedure as above. I present my observations below.

Observations:
Test accuracy: 65.60714285714286 %
Macro F1-score: 0.2660489368223934
F1-score for class 1: 0.13380281690140844
F1-score for class 2: 0.01680672268907563
F1-score for class 3: 0.07831762146482958
F1-score for class 4: 0.29696969696969694
F1-score for class 5: 0.8043478260869565

As clear from data, the test set accuracy decreases on removing stopwords and applying stemming ($65.6 < 66.2$). This can be explained as in our case, stopwords can actually play a crucial role in classification. For e.g. consider a 1 star review that just says: "The music is not good". Once, I process the data using the procedure specified above, I will be left with a list containing two tokens- ["music", "good"] (as all other words are stopwords). Hence, analysing this list, the algorithm will classify it to be a *good* review (class 4, 5), rather than a *bad* one, as it can see only two tokens- "good music". Hence, the example will be miss-classified.
I tested this intuition, by only performing stemming and not removing stopwords, and the model accuracy increased to 65.9 %. Thus, in our problem of review classification, removing stopwords is not a good choice.
Next, I will try to explain, why stemming of words is leading to a decrease in accuracy ($65.9 < 66.2$). Again, consider two sentences: "I liked this", "I like this". The first one is having a past-tense meaning, in the sense that the reviewer used to like the music (say) before, but it may not be the case currently. In the other sentence, the reviewer is claiming that he/she still likes

the music. Clearly both the sentences will have the same stemmed token list, but the former can correspond to a lower rating review and the latter can correspond to a higher one. This can lead to *confusion* and miss-classification. In other words, stemming cuts of context-related information and can lead to confusion. Hence, it is decreasing the accuracy in our problem.

1.5 Feature Engineering

Before training the model, I processed the text using `nltk` library. Exact procedure:

1. Tokenize each reviewText using `word_tokenize()`
2. Store punctuations like '!' and '?' in a separate list
3. Convert all words/tokens to lower case and remove punctuations within words (like ')
4. Filter out non-alphabetical words from the list
5. Stem words using Porter Stemming algorithm
6. Filter out stop words from the list
7. Append punctuations ('!', '?') to the list
8. Generate bigrams for the given list of tokens and append it to the original list

Features introduced/manipulated:

1. Manipulated existed feature (unigram). Unigrams originally (section 1.4) consisted of only alphabetical words. In this model, I expanded their domain space to include some special punctuations like '!' and '?'. The intuition behind this was that many low rating reviews have heavy usage of exclamations and questions, possibly because the user did not like the music/product and want to put stress on some specific aspects of their review, and in some cases, to ask questions (about the performance of the product) and voice their opinions. On the other hand, even high ratings use '!' to express their excitement over the product, but its rare for high rating reviews to ask questions and use '?'. Hence, I believe that these two values and their inclusion in the feature domain can further help in classifying reviews and improving overall accuracy.
2. Adding new feature (bigram). Naive-Bayes assumes independence of words used at different positions given a class. This in the strict sense is *false*, as some unigrams may be more commonly used after some other words/unigrams. Also, bigrams help in making use of contextual information to some degree. For e.g. "not good" can be treated as "bad" in a bigram model, but it will be treated as "not" and "good" in a unigram model, which may or may not come out as "bad" in the overall prediction. Hence, I believe that including bigrams as features to my Naive-Bayes model will benefit the model and help in increasing accuracy.

Next, I created a vocabulary (unigrams + bigrams) from the text corpus (training data) and also added an *UNKNOWN* token to handle any unseen word/bigram in the test data. After this, I trained a Naive-Bayes classifier with Laplace Smoothing to avoid zero probabilities ($\alpha = 1$). (Bag of unigrams and bigrams)
Before testing the model, I processed the text (test set) using the same procedure as above. I present my observations on the following page.

Observations:
Test accuracy: 66.30714285714286 %
Macro F1-score: 0.2059790373886376
F1-score for class 1: 0.00823045267489712
F1-score for class 2: 0 (zero recall)
F1-score for class 3: 0.008976660682226214
F1-score for class 4: 0.2082004555808656
F1-score for class 5: 0.8044876180051991

Improvement in Accuracy:
I first tried using these two features in two different models. In both situations, I noticed an increase in model accuracy. Model trained using only unigrams and bigrams (no domain modifications) gave an accuracy of 66.14 % on the test data. On the other hand, model trained using only domain modifications gave an accuracy of 65.67 %. Both these accuracies are greater than the one obtained in section 1.4, but are lesser than the one obtained in section 1.1. Also, it can be seen that addition of bigrams have a more impact on the classification accuracy as compared to simply modifying the unigram domain space.
After this, I tried combining both these features and using them in a single model. With the amalgamated model (section 1.5), I was able to achieve a test accuracy of 66.3 %, which was even better than the one obtained in section 1.1. Hence, I was able to successfully engineer new features (bigrams and modified unigrams) which lead to an increase in test accuracy.

1.6 F1-Score

Best performing model above (w.r.t test accuracy): Model in section 1.5
F1-statistics for this model:-
Macro F1-score: 0.2059790373886376
F1-score for class 1: 0.00823045267489712
F1-score for class 2: 0 (zero recall)
F1-score for class 3: 0.008976660682226214
F1-score for class 4: 0.2082004555808656
F1-score for class 5: 0.8044876180051991

I think that F1-score metric is more suited for this kind of dataset because:

- Both training and test data are heavily skewed towards class 5 reviews (51 % and 66 % respectively), hence, high test accuracies can be obtained by simply doing majority predictions, as seen in section 1.2. Hence, it is important that the model also has a high precision and recall, and equivalently high F1-score.
- We are classifying reviews into star ratings, which can later on possibly influence decisions of a larger audience, hence, it is very important that we get things right, or equivalently good *recall*. Therefore, it is more important to minimize individual off-diagonal entries (confusion), than to maximize the sum of diagonal entries (accuracy), as *false negatives* can be meaning-changing (e.g. 1 star review classified as a 5 star one). This in turn is equivalent to having a high recall and precision, and consequently high F1-score.

1.7 Incorporating Summary for Classification

First, I processed the data (reviewText), just as I did in section 1.1. Then, before training the model, I also processed the summary using `nltk` library, and incorporated it in my training set. Exact procedure:

1. Tokenize each summary using `word_tokenize()`
2. Convert all words/tokens to lower case and remove punctuations within words (like ')
3. Filter out non-alphabetical words from the list
4. Extend the reviewText data by including summary data $\frac{m}{\min(8n,m)}$ times

where m = number of tokens in processed reviewText, and, n = number of tokens in processed summary. I explain the intuition behind this choice below:

1. Just adding summary text to review text once will not be sufficient in general, as summary speaks for the whole review. Hence, single addition may not assign suitable weight to the summary when determining probabilities in Naive-Bayes classification. I tested this out, and I was only able to increase the accuracy to 66.6 %, with a single addition of summary text to review.
2. So, now the problem was to determine how many times the summary should be added to the original review. Clearly, shorter the summary, more precise and powerful it can be when performing classification. Hence, we should add it more to the review text. On the other hand, larger the summary, more explicit it is in conveying the information, hence, additional enhancement in weight is not really required. This gives the *inverse* relation with the size of the summary (n).
3. Next, observe that the summary speaks for the review, and hence, it should never overshadow the review itself. Therefore, assigning it too much weight is out of the question, as it can lead to not-that-good fits (66.8 %). Since, summary speaks for the review itself, it is logical to assume that its weight is dependent on the size of the review itself. This is where the direct relation with review size comes in (m).
4. Now, the denominator should always be less than or equal to m, as I am adding the summary to review at least *once*. This is where the *min* comes in the denominator.
5. Last thing was to determine the constant of proportionality, and I experimented with different values like 1, 2, 3, 4, .., 15, and found that 8 gives the best accuracy-F1 tradeoff. Overall, 9 was giving the highest accuracy, but only by a small margin. On the other hand, F1-score in that case was significantly smaller. Hence, I decided to use 8 (or $\frac{1}{8}$) as the proportionality constant.

Next, I created a vocabulary from the text corpus (training data) and also added an *UNKNOWN* token to handle any unseen word in the test data. After this, I trained a Naive-Bayes classifier with Laplace Smoothing to avoid zero probabilities ($\alpha = 1$).
Before testing the model, I processed the text (test set) and incorporated the summary using the same procedure as above. I present my observations on the following page.

Observations:
Test accuracy: 67.57857142857142 %
Macro F1-score: 0.3319887607643829
F1-score for class 1: 0.2348993288590604
F1-score for class 2: 0.053475935828877004
F1-score for class 3: 0.17358958462492252
F1-score for class 4: 0.3772143105244877
F1-score for class 5: 0.8207646439845668

I was able to increase the test set accuracy of my model to **67.58 %**, which is by far the highest among all the models trained in previous sections (increase of 1.2 % over model trained in 1.5). Not surprisingly, this model also has the highest macro F1-score (0.332) among all the models.

Chapter 2

MNIST Digit Classification

2.1 Binary Classification

The dual optimization problem in case of binary-classification (soft-margin SVM) is to:

$$\max_{\alpha \in \mathbb{R}^m} \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{m} \alpha_i \alpha_j y^{(i)} y^{(j)} K(x^{(i)}, x^{(j)})$$

subject to the following constraints:

$$\forall 1 \leq i \leq m : 0 \leq \alpha_i \leq C$$

$$\sum_{i=1}^{m} \alpha_i y^{(i)} = 0$$

Note: The symbols follow classroom notation and have their usual meanings.
The optimal parameters can be found using QP-algorithm which CVXOPT uses, however, the package requires the problem to be expressed as a minimization problem, instead of a maximization one. Also, all the inequality constraints have to be of $\leq$ type. Hence, I expressed the problem equivalently as follows:

$$\min_{\alpha \in \mathbb{R}^m} \frac{1}{2} \sum_{i,j=1}^{m} \alpha_i \alpha_j y^{(i)} y^{(j)} K(x^{(i)}, x^{(j)}) - \sum_{i=1}^{m} \alpha_i$$

subject to the following constraints:

$$\forall 1 \leq i \leq m : -\alpha_i \leq 0 \wedge \alpha_i \leq C$$

$$\sum_{i=1}^{m} \alpha_i y^{(i)} = 0$$

From these expressions, I can see that the objective function can be written as: $\frac{1}{2}\alpha^T P \alpha + q^T \alpha$, where $P \in \mathbb{R}^{m \times m}$, such that $P_{ij} = y^{(i)} y^{(j)} K(x^{(i)}, x^{(j)})$, and $q \in \mathbb{R}^{m \times 1}$, with $q = [-1, -1, .., -1]^T$. Similarly, $G \in \mathbb{R}^{2m \times m}$, $h \in \mathbb{R}^{2m \times 1}$ and $A \in \mathbb{R}^{1 \times m}$ can be defined such that: $G\alpha \leq h$ and $A\alpha = 0$.

Thus, expressing the dual optimization problem in this format, I was able to train a binary classifier (4(-1) vs 5(1)) using CVXOPT package, for both linear and Gaussian kernel.

Note: My entry number ends with **4** (2019CS10424), hence, I trained a 4-vs-5 binary classifier, using CVXOPT and LIBSVM. Training/Test data was filtered accordingly.
Note: In all my observations, I only mention statistical data. Support vector indices, coefficients, weight and bias parameters are omitted and can be found in `Q2_data.txt` for each part below.

2.1.1 Linear Kernel (CVXOPT)

C: 1
Kernel Function: $K(x^{(i)}, x^{(j)}) = (x^{(i)})^T x^{(j)}$
Training Time: 98.23 seconds
Number of support vectors, nSV: 150 (α-threshold: 10^{-4})
Training Accuracy: 100.0 %
Macro F1-score: 1.0
F1-score for class 4: 1.0
F1-score for class 5: 1.0
Test accuracy: 98.66595517609392 %
Macro F1-score: 0.9866251804601451
F1-score for class 4: 0.9873031995937024
F1-score for class 5: 0.985947161326588

2.1.2 Gaussian Kernel (CVXOPT)

C: 1, $\boldsymbol{\gamma}$: 0.05
Kernel Function: $K(x^{(i)}, x^{(j)}) = e^{-\gamma||x^{(i)}-x^{(j)}||^2}$
Training Time: 65.18 seconds
Number of support vectors, nSV: 1459 (α-threshold: 1.2×10^{-3})
Training Accuracy: 100.0 %
Macro F1-score: 1.0
F1-score for class 4: 1.0
F1-score for class 5: 1.0
Test accuracy: 99.89327641408751 %
Macro F1-score: 0.998930296913958
F1-score for class 4: 0.9989816700610998
F1-score for class 5: 0.9988789237668162

Comparison with Linear Kernel:
The training accuracy is 100 % for both the above models. Hence, learning step is perfect for both kernels. The test accuracy in case of Gaussian kernel (99.89 %) is higher than in case of linear kernel (98.67 %). Also, the per-class F1-scores and consequently macro F1-score is higher in the classifier using Gaussian kernel. This can be explained as Gaussian kernel is a non-linear kernel and maps the input feature vectors into higher-dimension and learns a hyperplane in the larger space. This mapping can help bifurcating the data with a hyperplane more confidently, and learning a more accurate hypothesis function, which may be a non-linear function in the original space. Also, as the number of training examples are much greater than the number of features, SVM is less prone to over-fitting and is therefore, able to correctly account for *general* non-linear characteristics of features, resulting in higher test accuracy. (less *false* predictions)

In the next two sections, I present observations that were collected using the LIBSVM package. `svm_train()` and `svm_predict()` interface functions were used with appropriate flags to carry out training and testing respectively.

2.1.3 Linear Kernel (LIBSVM)

C: 1
Kernel Function: $K(x^{(i)}, x^{(j)}) = (x^{(i)})^T x^{(j)}$
Training Time: 1.76 seconds
Number of support vectors, nSV: 150
Training Accuracy: 100.0 %
Macro F1-score: 1.0
F1-score for class 4: 1.0
F1-score for class 5: 1.0
Test accuracy: 98.66595517609392 %
Macro F1-score: 0.9866251804601451
F1-score for class 4: 0.9873031995937024
F1-score for class 5: 0.985947161326588
Maximum element-wise difference in "w" parameter: 0.0002981 (absolute difference)
Difference in "b" parameter: 0.00085964

Comparison with classifier trained using CVXOPT:
The number of support vectors (150) and training/test accuracies and F1-score are the same as for model trained using CVXOPT. This validates the observations present in section 2.1.1. Though not mentioned here, the support vector indices for both the models are also identical and can be looked up in `Q2_data.txt` file. This is expected as model hyperparameters, kernel, training and test data were the same for both of them. Next coming to the weight and bias parameters. I have specified above the *absolute* difference in "b" parameter and maximum element-wise difference in "w" parameter. Both of these are of the order 10^{-4} and hence, are negligible. The almost-identical nature of "w" and "b" parameters can be solidified by noticing that the test accuracies and F1-scores for both models are completely identical. Hence, it is safe to say that the two models (2.1.1 and 2.1.3) are in good agreement, when it comes to model parameters.
Next, I compare the computational costs (training time) of the two models. Model trained used LIBSVM took 1.76 seconds as compared to 98.23 seconds taken by model trained using CVXOPT. This means that training was around **55** times faster with LIBSVM. This is expected as LIBSVM uses a more specific algorithm for computing the optimal value of α, namely **SMO**, which is tuned to the needs of an SVM by taking into account its machine learning aspects like sparsity of support vectors. On the other hand, CVXOPT provides a general purpose quadratic programming routine and though highly optimized, is not finely tuned to the needs of SVM. Hence, it is unable to make use of extra information provided in case of SVMs and therefore, slower.

2.1.4 Gaussian Kernel (LIBSVM)

C: 1, $\boldsymbol{\gamma}$: 0.05
Kernel Function: $K(x^{(i)}, x^{(j)}) = e^{-\gamma||x^{(i)}-x^{(j)}||^2}$
Training Time: 8.35 seconds
Number of support vectors, nSV: 1459
Training Accuracy: 100.0 %
Macro F1-score: 1.0
F1-score for class 4: 1.0
F1-score for class 5: 1.0
Test accuracy: 99.89327641408752 %
Macro F1-score: 0.998930296913958
F1-score for class 4: 0.9989816700610998
F1-score for class 5: 0.9988789237668162
Maximum element-wise difference in SV-coefficient: 0.00074583

Comparison with classifier trained using CVXOPT:
The number of support vectors (1459) and training/test accuracies and F1-score are the same as for model trained using CVXOPT. This validates the observations present in section 2.1.2. Though not mentioned here, the support vector indices for both the models are also identical and can be looked up in `Q2_data.txt` file. This is expected as model hyperparameters, kernel, training and test data were the same for both of them. Next coming to the weight and bias parameters. Since, explicit expression for weight parameter is difficult to obtain, I have equivalently compared the support vector coefficient for each support vector. I have specified above the maximum element-wise difference in SV-coefficient. This difference is of the order 10^{-4} and hence, is negligible (as the coeffecents are all greater than 10^{-3}). The almost-identical nature of coefficients can be solidified by noticing that the test accuracies and F1-scores for both models are completely identical. Hence, it is safe to say that the two models (2.1.2 and 2.1.4) are in good agreement, when it comes to model parameters.
Next, I compare the computational costs (training time) of the two models. Model trained used LIBSVM took 8.35 seconds as compared to 65.18 seconds taken by model trained using CVXOPT. This means that training was around **8** times faster with LIBSVM. This is expected as LIBSVM uses a more specific algorithm for computing the optimal value of α, namely **SMO**, which is tuned to the needs of an SVM by taking into account its machine learning aspects like sparsity of support vectors. On the other hand, CVXOPT provides a general purpose quadratic programming routine and though highly optimized, is not finely tuned to the needs of SVM. Hence, it is unable to make use of extra information provided in case of SVMs and therefore, slower.

2.2 Multi-class Classification

For multi-class classification, I followed the one-vs-one method and trained 45 binary-classifiers (each class vs every other class). During prediction, I output the class with maximum number of votes from all the 45 classifiers. For tie breaking, I used the maximum score for a given class, where score is the absolute value of $w^T x + b$.

2.2.1 Gaussian Kernel (CVXOPT)

For this part, I used the binary-classifier (section 2.1.2) for training 45 models.
C: 1, $\boldsymbol{\gamma}$: 0.05
Kernel Function: $K(x^{(i)}, x^{(j)}) = e^{-\gamma||x^{(i)}-x^{(j)}||^2}$
Training Time: 2487.39 seconds
Test accuracy: 97.23 %
Macro F1-score: 0.9721508007907497
F1-score for class 0: 0.9832572298325724
F1-score for class 1: 0.9889722099691223
F1-score for class 2: 0.9629272989889263
F1-score for class 3: 0.9718796250616676
F1-score for class 4: 0.9766497461928934
F1-score for class 5: 0.9735806632939854
F1-score for class 6: 0.9786347055758208
F1-score for class 7: 0.9681216282491417
F1-score for class 8: 0.958290946083418
F1-score for class 9: 0.9591939546599496

2.2.2 Gaussian Kernel (LIBSVM)

For this part, I used the `svm_train()` and `svm_predict()` interface of the LIBSVM module for training and testing respectively.
C: 1, $\boldsymbol{\gamma}$: 0.05
Kernel Function: $K(x^{(i)}, x^{(j)}) = e^{-\gamma||x^{(i)}-x^{(j)}||^2}$
Training Time: 593.42 seconds
Test accuracy: 97.23 %
Macro F1-score: 0.9721441482606101
F1-score for class 0: 0.983756345177665
F1-score for class 1: 0.9889722099691223
F1-score for class 2: 0.9633911368015413
F1-score for class 3: 0.9718796250616676
F1-score for class 4: 0.9766497461928934
F1-score for class 5: 0.9730337078651685
F1-score for class 6: 0.978125
F1-score for class 7: 0.9671729544341009
F1-score for class 8: 0.9592668024439918
F1-score for class 9: 0.9591939546599496

Comparison with classifier trained using CVXOPT:
The test accuracy is the same as that for model trained using CVXOPT. The per-class F1-scores for classes 0, 2, 5, 6, 7 and 8 are slightly different for the two models. Consequently the macro F1-scores also differ by a small value (of the order of 10^{-5}). This is likely due to how the tie-breaking is carried out in the two models, in case of equal number of votes. Hence, it is safe to say that the two classifiers are in good agreement, when it comes to model parameters and testing. This is expected as the underlying models (section 2.1.2 and 2.1.4) were almost-identical and in good agreement.
Next, I compare the computational costs (training time) of the two models. Model trained using LIBSVM took 593.42 seconds as compared to 2487.39 seconds taken by model trained using CVXOPT. This means that training was around **4** times faster with LIBSVM. This is expected as LIBSVM uses a more specific algorithm for computing the optimal value of α, namely **SMO**, which is tuned to the needs of an SVM by taking into account its machine learning aspects like sparsity of support vectors. On the other hand, CVXOPT provides a general purpose quadratic programming routine and though highly optimized, is not finely tuned to the needs of SVM. Hence, it is unable to make use of extra information provided in case of SVMs and therefore, slower.

2.2.3 Confusion Matrices

For binary classification model (section 2.1.4)

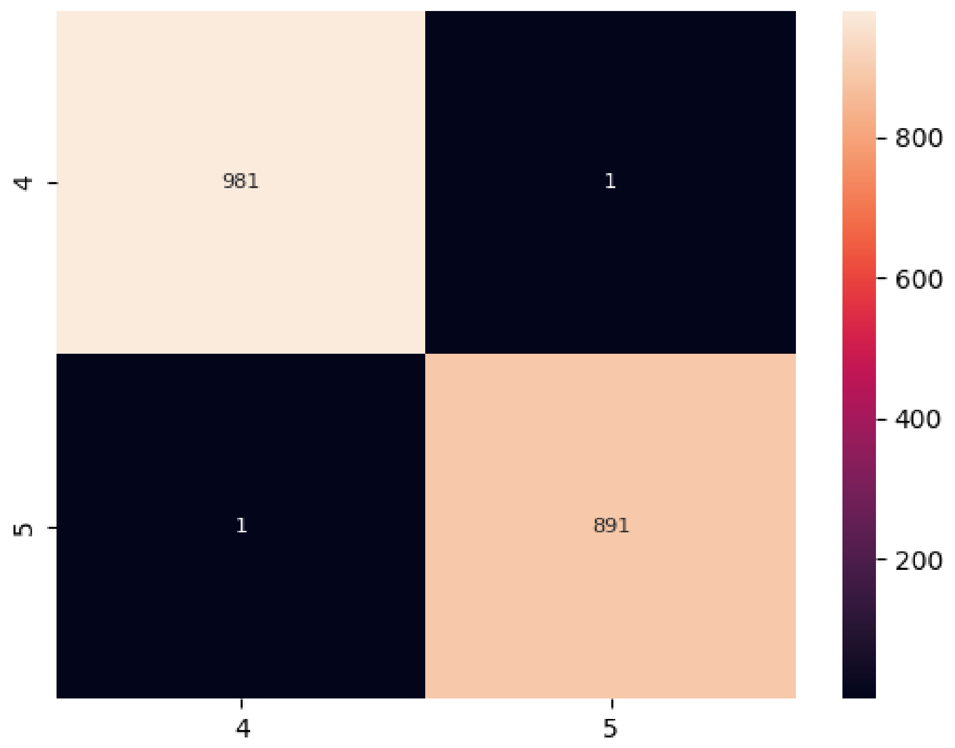
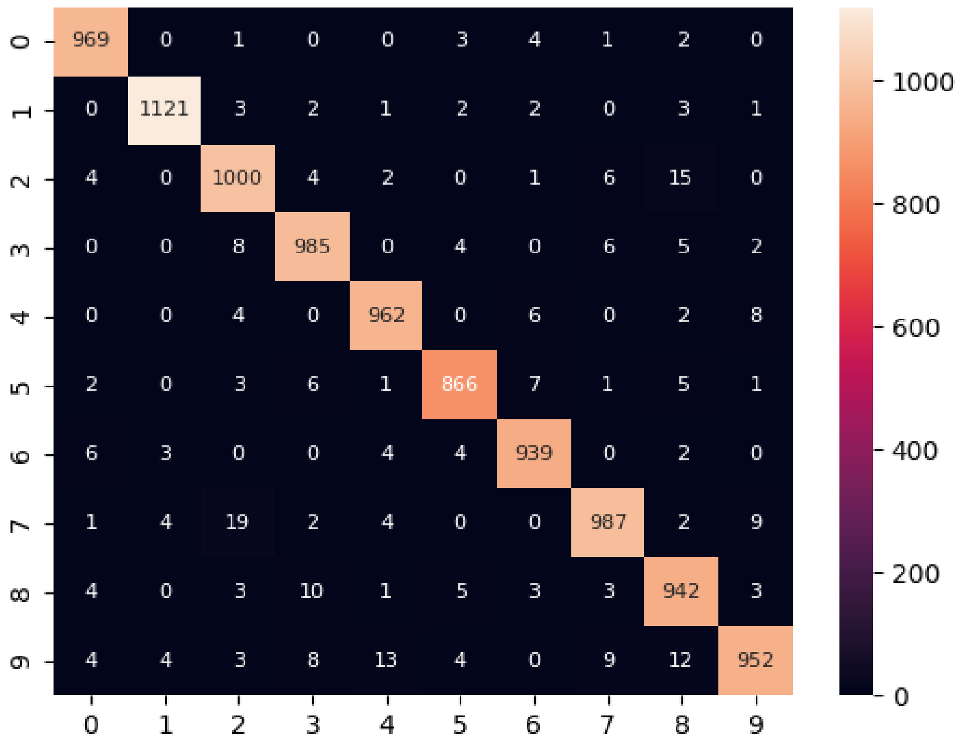


Figure 2.1: Confusion Matrix for binary classification (4 vs 5)

Observations:

- Majority of the test examples lie on the main diagonal of the confusion matrix indicating that the classifier correctly classifies majority of the test examples (99.89 %).
- Only one *4* is confused to be *5*, and one *5* is confused to be *4*. Consequently, the matrix is symmetric.

For multi-class classification model (section 2.2.2)



	0	1	2	3	4	5	6	7	8	9
0	969	0	1	0	0	3	4	1	2	0
1	0	1121	3	2	1	2	2	0	3	1
2	4	0	1000	4	2	0	1	6	15	0
3	0	0	8	985	0	4	0	6	5	2
4	0	0	4	0	962	0	6	0	2	8
5	2	0	3	6	1	866	7	1	5	1
6	6	3	0	0	4	4	939	0	2	0
7	1	4	19	2	4	0	0	987	2	9
8	4	0	3	10	1	5	3	3	942	3
9	4	4	3	8	13	4	0	9	12	952

Figure 2.2: Confusion Matrix for 10-class classification

Observations:

- Majority of the test examples lie on the main diagonal of the confusion matrix indicating that the classifier correctly classifies majority of the test examples (97.23 %). We refer to these entries as *true positives*.
- Other off diagonal entries are much less in number and many entries are also zero indicating *no-confusion* between those classes. Examples include 0-1, 0-3, 0-4, 6-7 etc.
- The matrix is not symmetric, and there is no direct relationship between miss-classification of class *a* into class *b* and vice-versa. This is actually dependent on the number and quality of test examples.
- The digit **7** is miss-classified to be digit **2** most often (19 times). This is not totally unexpected, as poor handwriting can make a *7* look like *2*, if its written with a dash.

Some examples of miss-classified digits:

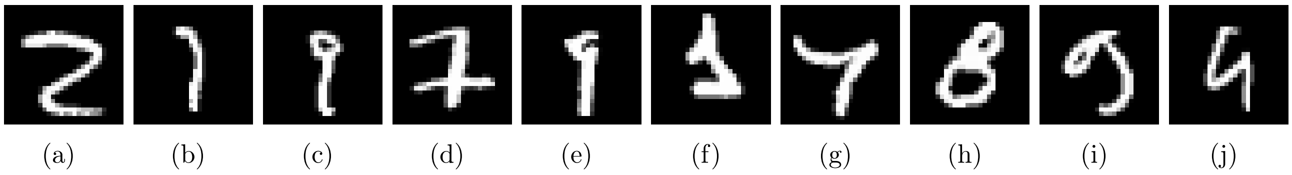


Figure 2.3: 2-but-7, 7-but-1, 9-but-7, 7-but-2, 9-but-7, 1-but-6, 7-but-4, 8-but-3, 9-but-5, 9-but-4

Notation: *a-but-b* denotes that the actual class is *a*, but predicted class is *b*.

The miss-classification of the above digits is reasonable. For e.g. (b) clearly looks more like a *1* than *7*. The same thing can be said for examples like (g) and (j), where the digit looks more like *4*, due to lack of proper writing. *7* being miss-classified into *2* ((d)) can also have another reason: there are two commonly used ways of writing *7*, one without a horizontal dash, and other with one. If however, the horizontal dash is more to the bottom of the digit, there are chances that people may think of it as a *2*, than a *7*, which is so the case in (d). This also explains why *7* is miss-classified into *2* most often. (poor handwriting and multiple-formats). The miss-classification of other digits can be argued on similar grounds. Hence, the results make sense.

2.2.4 5-Fold Cross Validation

To do 5-Fold cross validation, I performed the following steps:

1. Randomly shuffle the data set
2. Split the data set into 5 parts of equal examples
3. For every value of C in the set $\{10^{-5}, 10^{-3}, 1, 5, 10\}$, train 5 models, leaving out one different split in each of them. (use model 2.2.2)
4. For each of these 5 models, determine validation accuracy on the left out data, and determine cross validation accuracy (average of the five accuracies). Also, store the model that gave the highest accuracy (best model).
5. Use the best model obtained above to predict on test data and note down the test accuracy.

Data collected:

C	**CV Accuracy (in %)**	**Test Accuracy (in %)**
10^{-5}	9.39	10.09
10^{-3}	9.39	10.09
1	97.275	97.12
5	97.37	97.23
10	97.37	97.23

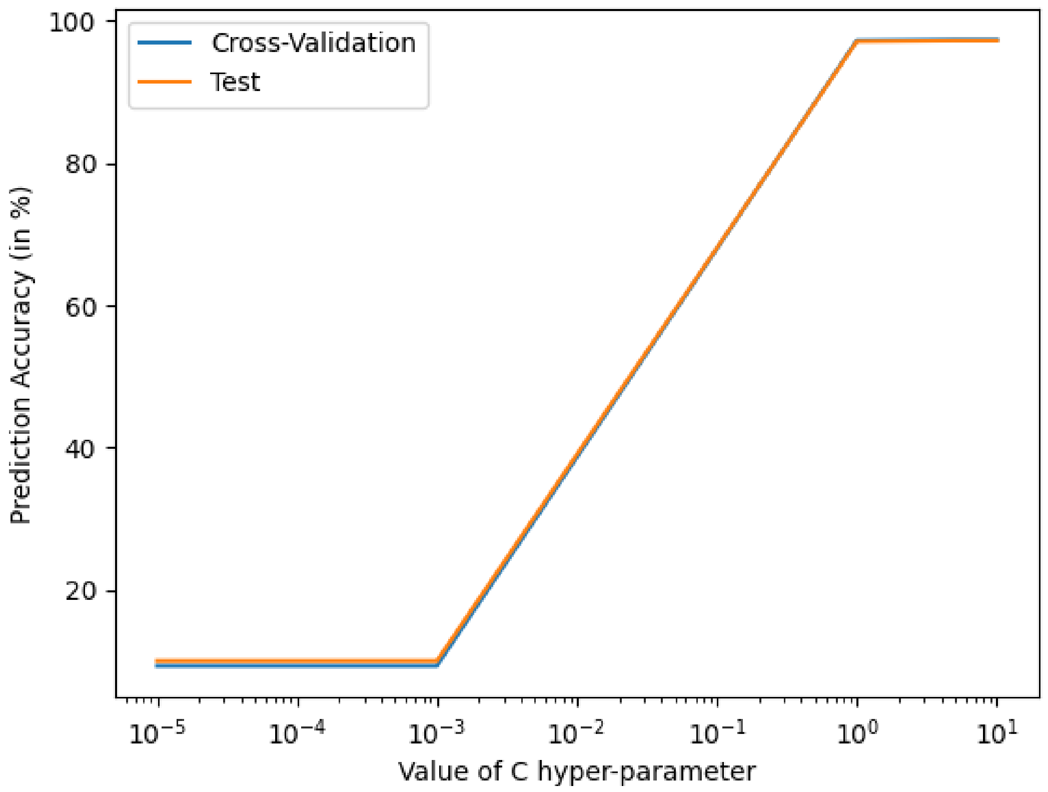


Figure 2.4: Variation of prediction accuracy with C

Observations:

- $C = 5$ and $C = 10$ give the best value of cross-validation accuracy (97.37 %).
- These values of C (5 and 10) also give the best value of test accuracy (97.23 %) as determined by the best 5-fold model.
- The CV accuracy and test accuracy are very close to each other for all values of C, that were used. This indicates that the validation set is a good proxy for the test set, and the parameter value is able to generalize well.
- For small values of C (10^{-5} and 10^{-3}), both the CV accuracy and test accuracy are extremely low (9-10 %). This indicates that the model is *under-fitting* the data. Increasing C can help mitigate under-fitting
- From the data obtained, I can conclude that $C = 5$ and $C = 10$ are the best model parameters for multi-class classification using Gaussian kernel (with $\gamma = 0.05$), from the set $\{10^{-5}, 10^{-3}, 1, 5, 10\}$