Assignment 2

October 7, 2021

1 Text classification

Reading the data

[35]:
```
data_dir = "reviews_Digital_Music_5.json"
files = os.listdir(data_dir)

"""
Reading train data
"""
train_data = pd.read_json(f"{data_dir}/{files[0]}", lines=True)
X_train_text = train_data["reviewText"].to_numpy()
Y_train = train_data["overall"].to_numpy()

"""
Reading test data
"""
test_data = pd.read_json(f"{data_dir}/{files[1]}", lines=True)
X_test_text = test_data["reviewText"].to_numpy()
Y_test = test_data["overall"].to_numpy()
```

1.1 Naive Bayes

1.1.1 Tokenization

This function converts list of document tokens into term document matrix

[37]:
```
def countVectorizer(docs, vocabulary=None):

    indptr = [0]
    indices = []
    term_freq = []

    if vocabulary is None:
        fixed_vocabulary = False
        vocabulary = {}
    else:
        fixed_vocabulary = True
```

```
    for doc in docs:
        for term in doc:
            if not fixed_vocabulary or term in vocabulary:
                index = vocabulary.setdefault(term, len(vocabulary))
                indices.append(index)
                term_freq.append(1)
        indptr.append(len(indices))

    term_doc_matrix = csr_matrix((term_freq, indices, indptr),
                                 shape=(len(docs), len(vocabulary)),
                                 dtype=int)

    return vocabulary, term_doc_matrix
```

convert to lowercase and remove punctuations and characters

[39]:
```
def text_cleaning(data):
    clean_data = []
    for i, doc in enumerate(data):
        clean_doc = re.sub(r'[^\w\s]', ' ', str(doc).lower().strip())
        clean_doc = re.sub(r' [\w\s] ', ' ', clean_doc).strip()
        clean_data.append(clean_doc)
    return clean_data
```

[40]:
```
X_train_clean = text_cleaning(X_train_text)
X_test_clean = text_cleaning(X_test_text)
```

Tokenization code for converting string to tokens

[41]:
```
def tokenize(data):
    return word_tokenize(data)

def text_processor_1(data):
    data_tokens = []
    for doc in data:
        data_tokens.append(tokenize(doc))
    return data_tokens
```

Here we are doing the following: * convert documents into tokens * Forms a term document matrix representation of the collection

[55]:
```
classes = np.unique(Y_train)

X_train_tokens = text_processor_1(X_train_clean)
vocab, X_train = countVectorizer(X_train_tokens)

X_test_tokens = text_processor_1(X_test_clean)
vocab, X_test = countVectorizer(X_test_tokens, vocab)
```

With this we have obtained a vocabulary of size 101167. Size of the term document matrix is mentioned below.

[77]:
```
(<50000x101167 sparse matrix of type '<class 'numpy.int64'>'
        with 9738796 stored elements in Compressed Sparse Row format>,
 <14000x101167 sparse matrix of type '<class 'numpy.int64'>'
        with 2772505 stored elements in Compressed Sparse Row format>)
```

1.1.2 Training

This is the code for finding the model parameters

[57]:
```python
def train_mnb(X_train, Y_train, classes):
    phi = []
    theta = []

    for k in classes:
        phi_k = np.mean(Y_train == k)
        phi.append(phi_k)

        term_k = X_train[Y_train == k].sum(axis=0) + 1
        theta_k = np.squeeze(np.asarray(term_k))/term_k.sum()
        theta.append(theta_k)

    return np.array(phi), np.array(theta)
```

[58]:
```python
phi, theta = train_mnb(X_train, Y_train, classes)
```

1.1.3 Accuracy

Code for model prediction

[59]:
```python
def accuracy(Y_pred, Y):
    return np.mean(Y_pred == Y)
```

[60]:
```python
def predict_mnb(phi, theta, X_test, classes):
    log_theta = np.log(theta).T
    Y_score = X_test@log_theta + phi
    Y_pred = np.argmax(Y_score, axis=1)

    Y_pred = np.array([classes[y] for y in Y_pred])
    return Y_pred
```

Obtained the following accuracy in test and train data

[62]:

	train	test
Accuracy	0.70038	0.666286

1.2 Accuracy comparision

[38]:
```
pd.DataFrame([[test_acc, random_acc, m_acc]],
             columns=['test', 'random', 'majority'], index=['Accuracy'])
```

[38]:
```
              test    random  majority
Accuracy  0.666357  0.203714  0.660857
```

There is improvement in the test accuracy than random prediction.

But very less improvement from the majority prediction this is because 50% of the review have rating 5.

1.3 Confusion Matrix

[64]:
```
Predicted :   1   2    3    4     5
Actual :
1            34  11   41   42   100
2             4   5   75  112   130
3             5   5  131  467   478
4             8   3   97  934  2066
5            43  17  102  866  8224
```

Category 5 has the highest value in the diagonal entries.

[59]:
```
Actual :          1         2         3      4         5
perc       0.016286  0.023286  0.077571  0.222  0.660857
```

As we can see in the actual collection majority of file are from Category 5.

[60]:
```
Predicted :      1       2         3         4         5
perc         0.205  0.1975  0.196714  0.198143  0.202643
```

And what naive bayes is predicting, is its trying to distribute everything equally to each Category.

1.4 Stemming and Stopword removal

Following are the functions for stemming and stopword removal

[50]:
```
def stemming(data):
    stems = [PorterStemmer().stem(token) for token in data]
    return stems
```

[51]:
```
def remove_stop_words(data, stopwords):
    tokens = [word for word in data if word not in stopwords]
    return tokens
```

[52]:
```
def text_processor_2(data, stopwords):
    data_tokens = []
    for i, doc in enumerate(data):
        if i%1000 == 0:
```

```
            print(f"Processing : {i}", end='\r', flush=True)
        tokens = remove_stop_words(doc, stopwords)
        tokens = stemming(tokens)
        data_tokens.append(tokens)
    return data_tokens
```

[69]:
```
stopwords = nltk.corpus.stopwords.words('english')

X_train_stems = text_processor_2(X_train_tokens, stopwords)
vocab, X_train = countVectorizer(X_train_stems)

X_test_stems = text_processor_2(X_test_tokens, stopwords)
vocab, X_test = countVectorizer(X_test_stems, vocab)
```

```
Processing : 13000
```

Here we obtained vocabulary size of 76039, which is less than what we obtained before. Shape of the term document matrix below:

[70]:
```
(<50000x76039 sparse matrix of type '<class 'numpy.int64'>'
        with 5379545 stored elements in Compressed Sparse Row format>,
 <14000x76039 sparse matrix of type '<class 'numpy.int64'>'
        with 1540199 stored elements in Compressed Sparse Row format>)
```

Obtained the following accuracy in test and train data

[95]:

	train	test
Accuracy	0.67634	0.654714

We can observe that the test accuracy has gone up by 0.002, which is very less.

Confusion matrix on the test data

[114]:

Predicted :	1	2	3	4	5
Actual :					
1	37	12	42	37	100
2	7	11	73	105	130
3	3	9	134	448	492
4	13	9	122	895	2069
5	66	28	132	937	8089

1.5 Feature engineering

function for creating term document matrix

[139]:
```
def extract_feature(X_train_input, X_test_input, extract_func):
    X_train_tokens = extract_func(X_train_input)
    vocab, X_train = countVectorizer(X_train_tokens)

    X_test_tokens = extract_func(X_test_input)
```

```
    vocab, X_test = countVectorizer(X_test_tokens, vocab)
    return X_train, X_test, X_train_tokens, X_test_tokens, vocab
```

function for training and prediction

[170]:
```
def train_predict(X_train, Y_train, X_test, Y_test, classes):
    phi, theta = train_mnb(X_train, Y_train, classes)


    Y_pred = predict_mnb(phi, theta, X_train, classes)
    train_acc = accuracy(Y_pred, Y_train)
    print(f"Accuracy : {train_acc}")

    Y_pred = predict_mnb(phi, theta, X_test, classes)
    test_acc = accuracy(Y_pred, Y_test)
    print(f"Accuracy : {test_acc}")

    return phi, theta, Y_pred, train_acc, test_acc
```

The above method is not helping much

1.5.1 Bi-gram model

Code for adding bigrams

[156]:
```
def concat_bigram(data):
    data_tokens = []
    for doc in data:
        doc_bi_gram = []
        for k in range(1, len(doc)):
            doc_bi_gram.append(f"{doc[k-1]} {doc[k]}")
        data_tokens.append(doc+doc_bi_gram)
    return data_tokens
```

[171]:
```
X_train, X_test, X_train_bigrams, X_test_bigrams, vocab = extract_feature(
    X_train_stems, X_test_stems, concat_bigram)
```

Here we are adding considering bigrams as tokens, and obtained vocabulary size of 2105996. Shape of the term document matrix below:

[172]:
```
(<50000x2105996 sparse matrix of type '<class 'numpy.int64'>'
        with 10709090 stored elements in Compressed Sparse Row format>,
 <14000x2105996 sparse matrix of type '<class 'numpy.int64'>'
        with 2603303 stored elements in Compressed Sparse Row format>)
```

[174]:
```
             train      test
Accuracy  0.83056  0.661857
```

Confusion matrix of the test data points

```
[175]: Predicted :  1  2  3   4     5
       Actual :
       1            1  0  1  15   211
       2            0  0  0  29   297
       3            0  0  1  93   992
       4            0  0  1  82  3025
       5            0  0  0  70  9182
```

1.5.2 Tri-gram model

Code for adding bigrams

```python
[176]: def concat_trigram(data):
           data_tokens = []
           for doc in data:
               doc_tri_gram = []
               for k in range(2, len(doc)):
                   doc_tri_gram.append(f"{doc[k-2]} {doc[k-1]} {doc[k]}")
               data_tokens.append(doc+doc_tri_gram)
           return data_tokens
```

```python
[177]: X_train, X_test, X_train_trigrams, X_test_trigrams, vocab = extract_feature(
           X_train_stems, X_test_stems, concat_trigram)
```

Here we are adding considering trigrams as tokens, and obtained vocabulary size of 4628313. Shape of the term document matrix below:

```
[178]: (<50000x4628313 sparse matrix of type '<class 'numpy.int64'>'
               with 10659098 stored elements in Compressed Sparse Row format>,
        <14000x4628313 sparse matrix of type '<class 'numpy.int64'>'
               with 1795153 stored elements in Compressed Sparse Row format>)
```

```
[180]:             train   test
       Accuracy  0.89614  0.661
```

Confusion matrix of the test data points

```
[181]: Predicted :  1  2  3  4     5
       Actual :
       1            0  0  0  1   227
       2            0  0  0  6   320
       3            0  0  0  6  1080
       4            0  0  0  9  3099
       5            0  0  0  7  9245
```

1.6 F1 score

[65]:
```python
def F1_score(precision, recall):
    n = 2*precision*recall
    d = precision + recall
    return n/d
```

[66]:
```python
def compute_F1_from_confusion(conf_df):
    conf_matrix = conf_df.to_numpy()
    f1_scores = []

    for c in range(len(conf_matrix)):
        fp_tp = conf_matrix[:,c].sum()
        if fp_tp:
            precision = conf_matrix[c, c]/fp_tp
        else:
            precision = 1

        recall = conf_matrix[c, c]/conf_matrix[c, :].sum()
        f1 = F1_score(precision, recall)
        f1_scores.append(f1)

    return pd.DataFrame(f1_scores, index=conf_df.columns.to_list(),␣
↪columns=['F1 score'])
```

I obtained best model in the section 1a.

[67]:
```python
f1_df = compute_F1_from_confusion(conf_df).T; f1_df
```

[67]:

	1	2	3	4	5
F1 score	0.21118	0.027248	0.171018	0.337855	0.812247

[68]:
```python
f1_df.mean(axis=1).to_numpy()[0]
```

[68]: 0.31190964352289685

The classes are highly skewed so macro f1 error is better than test error.

1.7 Summary

Only summary Reading data

[110]:
```python
train_data = pd.read_json(f"{data_dir}/{files[0]}", lines=True)
X_train_sum_text = train_data["summary"].to_numpy()
Y_train = train_data["overall"].to_numpy()

test_data = pd.read_json(f"{data_dir}/{files[1]}", lines=True)
X_test_sum_text = test_data["summary"].to_numpy()
Y_test = test_data["overall"].to_numpy()
```

cleaning data

[111]:
```
X_train_sum_clean = text_cleaning(X_train_sum)
X_test_sum_clean = text_cleaning(X_test_sum)
```

tokenization

[112]:
```
X_train_sum_tokens = text_processor_1(X_train_sum_clean)
vocab, X_train_sum = countVectorizer(X_train_sum_tokens)

X_test_sum_tokens = text_processor_1(X_test_sum_clean)
vocab, X_test_sum = countVectorizer(X_test_sum_tokens, vocab)
```

stemming and stopword removal

[113]:
```
stopwords = nltk.corpus.stopwords.words('english')

X_train_sum_stems = text_processor_2(X_train_sum_tokens, stopwords)
vocab, X_train_sum = countVectorizer(X_train_sum_stems)

X_test_sum_stems = text_processor_2(X_test_sum_tokens, stopwords)
vocab, X_test_sum = countVectorizer(X_test_sum_stems, vocab)
```

```
Processing : 13000
```

Training and prediction

[115]:
```
phi, theta = train_mnb(X_train_sum, Y_train, classes)
Y_pred = predict_mnb(phi, theta, X_test_sum, classes)
acc = accuracy(Y_pred, Y_test)
print(f"Accuracy : {acc}")
```

```
Accuracy : 0.6508571428571429
```

Summary and reviewText Reading data

[88]:
```
train_data = pd.read_json(f"{data_dir}/{files[0]}", lines=True)
X_train_text = train_data["reviewText"].to_numpy()
X_train_sum = train_data["summary"].to_numpy()
Y_train = train_data["overall"].to_numpy()

test_data = pd.read_json(f"{data_dir}/{files[1]}", lines=True)
X_test_text = test_data["reviewText"].to_numpy()
X_test_sum = test_data["summary"].to_numpy()
Y_test = test_data["overall"].to_numpy()
```

combining text

[98]:
```
X_train_comb = X_train_text + X_train_sum
X_test_comb = X_test_text + X_test_sum
```

cleaning

[99]:
```
X_train_clean = text_cleaning(X_train_comb)
X_test_clean = text_cleaning(X_test_comb)
```

tokenization

[101]:
```
X_train_sum_tokens = text_processor_1(X_train_clean)
vocab, X_train_sum = countVectorizer(X_train_sum_tokens)

X_test_sum_tokens = text_processor_1(X_test_clean)
vocab, X_test_sum = countVectorizer(X_test_sum_tokens, vocab)
```

stemming and stopword removal

[105]:
```
stopwords = nltk.corpus.stopwords.words('english')

X_train_sum_stems = text_processor_2(X_train_sum_tokens, stopwords)
vocab, X_train = countVectorizer(X_train_sum_stems)

X_test_sum_stems = text_processor_2(X_test_sum_tokens, stopwords)
vocab, X_test = countVectorizer(X_test_sum_stems, vocab)
```

```
Processing : 13000
```

Training and prediction

[106]:
```
phi, theta = train_mnb(X_train, Y_train, classes)
Y_pred = predict_mnb(phi, theta, X_test, classes)
acc = accuracy(Y_pred, Y_test)
print(f"Accuracy : {acc}")
```

```
Accuracy : 0.6635714285714286
```

2 MNIST Digit Classification

Reading data function for loading the MNIST dataset given the path.

[13]:
```
def load_mnist(mnist_dir):
    train_data = pd.read_csv(f"{mnist_dir}/train.csv", header=None).to_numpy()
    X_train = train_data[:, :-1]/255
    Y_train = train_data[:, -1].reshape(-1,1)

    test_data = pd.read_csv(f"{mnist_dir}/test.csv", header=None).to_numpy()
    X_test = test_data[:, :-1]/255
    Y_test = test_data[:, -1].reshape(-1,1)

    return X_train, Y_train, X_test, Y_test
```

the below function returns dataset of two labels, which can then used for binary classification

[14]:
```
def two_class_data(X, Y, i, j):
    pos_ij = np.where(np.logical_or(Y == i, Y == j))[0]

    X_ij = X[pos_ij]
    Y_ij = np.where( Y[pos_ij] == i, 1, -1)
    return X_ij, Y_ij
```

The parameters of the SVM used for training are $C = 1.0$, $\gamma = 0.05$ and *threshold* = *1e-6* which decides the number support vectors, datapoints with $threshold < \alpha_i < C - threhold$ are considered support vectors

2.1 Binary classification

Code for expressing the SVM optimization problem

[14]:
```
def QP_coefficients(X_train, Y_train, kernel_func, C=1.0):
    K = kernel_func(X_train)
    YTY = Y_train@Y_train.T

    P = matrix(YTY*K, tc='d')
    N = X_train.shape[0]
    q = matrix(np.full(N, -1), tc='d')

    G1 = np.diag(np.full(N, -1))
    h1 = np.zeros(N)
    G2 = np.eye(N)
    h2 = np.full(N, C)
    G = matrix(np.vstack([G1, G2]), tc='d')
    h = matrix(np.hstack([h1, h2]), tc='d')

    A = matrix(Y_train.T, tc='d')
    b = matrix(0, tc='d')
    return P, q, G, h, A, b
```

2.1.1 Linear Kernel

Class for linear SVM

[109]:
```
class LinearSVM:

    def __init__(self):
        self.alpha = None
        self.X = None
        self.y = None
        self.c = None

    def linear_kernel(self, X):
        return X@X.T
```

```
    def train(self, X_train, Y_train, C=1.0):
        self.X = X_train
        self.y = Y_train
        self.c = C

        P, q, G, h, A, b = QP_coefficients(X_train, Y_train,
                                   self.linear_kernel, C)
        sol = solvers.qp(P, q, G, h, A, b)
        self.alpha = np.array(sol['x'])

    def get_coeff_sv(self, threshold):
        sv_pos, _ = np.where(self.alpha > threshold)
        alpha_sv = self.alpha[sv_pos]
        X_sv = self.X[sv_pos]
        return alpha_sv, X_sv

    def compute_Wb(self, threshold):
        sv_pos, _ = np.where(self.alpha > threshold)

        alpha_sv = self.alpha[sv_pos]
        X_sv = self.X[sv_pos]
        Y_sv = self.y[sv_pos]
        W = np.sum(Y_sv*alpha_sv*X_sv, axis=0, keepdims=True).T

        msv_pos, _ = np.where(alpha_sv < C - threshold)
        X_msv = X_sv[msv_pos]
        Y_msv = Y_sv[msv_pos]
        alpha_msv = alpha_sv[msv_pos]
        b = np.mean(Y_msv - X_msv@W)
        return W, b


    def predict(self, X_test, theshold=1e-6):
        W, b = self.compute_Wb(threshold)
        Y_score = X_test@W + b
        Y_pred = np.where(Y_score > 0, 1, -1)
        return Y_pred
```

Training

[92]:
```
svm = LinearSVM()
svm.train(X_train_3, Y_train_3, C)
```

[99]:

	Training	Testing	Number of SV
Accuracy	1.0	0.9949	134

2.1.2 Gaussian Kernel

Class for gaussian SVM

[117]:
```
class GaussianSVM:

    def __init__(self):
        self.alpha = None
        self.X = None
        self.y = None
        self.c = None

    def gaussian_kernel_fast(self, X, Y, gamma=0.05):
        sq_dists = cdist(X, Y, 'sqeuclidean')
        K = np.exp(-gamma*sq_dists)
        return K

    def gaussian_kernel(self, X, gamma=0.05):
        sq_dists = squareform(pdist(X, 'sqeuclidean'))
        K = np.exp(-gamma*sq_dists)
        return K

    def train(self, X_train, Y_train, C=1.0, gamma=0.05):
        self.X = X_train
        self.y = Y_train
        self.c = C

        kernel = lambda X: self.gaussian_kernel(X, gamma)
        P, q, G, h, A, b = QP_coefficients(X_train, Y_train, kernel, C)

        sol = solvers.qp(P, q, G, h, A, b)
        self.alpha = np.array(sol['x'])


    def get_coeff_sv(self, threshold):
        sv_pos, _ = np.where(self.alpha > threshold)
        alpha_sv = self.alpha[sv_pos]
        X_sv = self.X[sv_pos]
        return alpha_sv, X_sv


    def predict(self, X_test, theshold=1e-6):
        sv_pos, _ = np.where(self.alpha > threshold)
        alpha_sv = self.alpha[sv_pos]
        X_sv = self.X[sv_pos]
        Y_sv = self.y[sv_pos]

        msv_pos, _ = np.where(alpha_sv < C - threshold)
```

```
        X_msv = X_sv[msv_pos]
        Y_msv = Y_sv[msv_pos]
        alpha_msv = alpha_sv[msv_pos]

        #compute b
        p1 = np.where(Y_msv == 1)[0][0]
        p2 = np.where(Y_msv == -1)[0][0]
        K = self.gaussian_kernel_fast(X_sv, X_msv[[p1, p2]])
        b = - np.sum(alpha_sv*Y_sv*K)/2

        #computing prediction
        K = self.gaussian_kernel_fast(X_sv, X_test)
        Y_score = np.sum(alpha_sv*Y_sv*K, axis=0) + b
        Y_score = Y_score.reshape(-1, 1)
        Y_pred = np.where(Y_score > 0, 1, -1)
        return Y_pred, Y_score
```

Training

[118]:
```
gsvm = GaussianSVM()
gsvm.train(X_train_3, Y_train_3, C, gamma)
```

[121]:

	Training	Testing	Number of SV
Accuracy	1.0	0.998996	1463

We can see there that using gaussian kernel our accuracy has improved, as it tranforms the data-points into infinite dimension, so we can capture non-linearities in a better way.

2.1.3 LIBSVM

Linear kernel

[105]:
```
prob  = svm_problem(Y_train_3.reshape(-1), X_train_3)
param = svm_parameter('-t 0 -c 1')
m = svm_train(prob, param)

Y_pred_lib, p_acc, p_vals = svm_predict(Y_train_3.reshape(-1), X_train_3, m)
Y_pred_lib, p_acc, p_vals = svm_predict(Y_test_3.reshape(-1), X_test_3, m)
```

```
Accuracy = 100% (4000/4000) (classification)
Accuracy = 99.498% (1982/1992) (classification)
```

comparision

[107]:

```
               Training Accuracy  Testing Accuracy  Training time  \
Linear CVOPTX                1.0           0.99497         50.769
Linear LIBSVM                1.0           0.99497          0.349

               Number of SV
Linear CVOPTX           134
Linear LIBSVM           134
```

Comparing the weights and the bias
below is a function to compute weight and bias of the LIBSVM

[193]:
```
def compute_libsvm_Wb(X, y, m, threshold=1e-10):
    coeff_sv = np.array(m.get_sv_coef())
    X_sv = X[m.get_sv_indices()]
    Y_sv = y[m.get_sv_indices()]
    W = np.sum(coeff_sv*X_sv, axis=0, keepdims=True).T

    msv_pos = np.where(coeff_sv < 1.0 - threshold)[0]
    X_msv = X_sv[msv_pos]
    Y_msv = Y_sv[msv_pos]
    b = np.mean(Y_msv - X_msv@W)
    return W, b
```

Computing the norm of the difference of the parameters obtained from LIBSVM and CVOPTX.

[199]:
```
W_lib, b_lib = compute_libsvm_Wb(X_train_3, Y_train_3, m, 1e-10)

print(f'Weight difference : {np.linalg.norm(W_lib - W)}')
print(f'Bias difference   : {np.linalg.norm(b_lib - b)}')
```

```
Weight difference : 12.221917886255408
Bias difference   : 12.269557835556052
```

Gaussian kernel

[122]:
```
prob  = svm_problem(Y_train_3.reshape(-1), X_train_3)
param = svm_parameter('-t 2 -c 1 -g 0.05')
m_g = svm_train(prob, param)

Y_pred_libg, p_acc, p_vals = svm_predict(Y_train_3.reshape(-1), X_train_3, m_g)
Y_pred_libg, p_acc, p_vals = svm_predict(Y_test_3.reshape(-1), X_test_3, m_g)
```

```
Accuracy = 100% (4000/4000) (classification)
Accuracy = 99.8996% (1990/1992) (classification)
```

Comparision

[124]:

	Training Accuracy	Testing Accuracy	Training time	\
Gaussian CVOPTX	1.0	0.99890	31.8130	
Gaussian LIBSVM	1.0	0.99899	2.9467	

	Number of SV
Gaussian CVOPTX	1463
Gaussian LIBSVM	1344

As we can see the computational cost of our implementation is way more than that of LIBSVM.

2.2 Multiclass Classification

2.2.1 OneVsOne Implementation

Below is the class which performs the one vs one multiclass SVM classification

[40]:
```
class OvO_GSVM:

    def __init__(self):
        self.ovo_svms = {}
        self.classes = None

    def train(self, X_train, Y_train, X_val, Y_val, C=1.0, gamma=0.05,
              threshold=1e-6):

        self.classes = np.unique(Y_train)

        for i in range(len(self.classes)):
            for j in range(i+1, len(self.classes)):

                #data collection
                X_train_ij, Y_train_ij = two_class_data(X_train, Y_train,
                                                        self.classes[i],
                                                        self.classes[j])
                X_val_ij, Y_val_ij = two_class_data(X_val, Y_val,
                                                    self.classes[i],
                                                    self.classes[j])

                #training
                svm = GaussianSVM()
                svm.train(X_train_ij, Y_train_ij, C, gamma)
                self.ovo_svms[(self.classes[i], self.classes[j])] = svm

                #accuracy
                Y_pred_ij, _ = svm.predict(X_val_ij, theshold=threshold)
                acc = accuracy(Y_pred_ij, Y_val_ij)
                print(f"{self.classes[i]}-{self.classes[j]} Accuracy : {acc}")


    def ovo_svm_prediction(self, Y_pred, Y_score):
        ovo_pred = []
        for i in range(Y_pred.shape[0]):
            count = np.bincount(Y_pred[i])
            max_count = np.max(count)

            labels = np.where(count == max_count)[0]
            label_score = Y_score[i][labels]

            label = labels[np.argmax(label_score)]
            ovo_pred.append(label)
```

```python
        return np.array(ovo_pred).reshape(-1, 1)

    def predict(self, X_test, threshold=1e-6):
        Y_pred_ijs = []
        Y_score_ijs = []
        for i in range(len(self.classes)):
            for j in range(i+1, len(self.classes)):

                svm = self.ovo_svms[(self.classes[i], self.classes[j])]

                Y_pred_ij, Y_score_ij = svm.predict(X_test, threshold)
                Y_pred_ij = np.where( Y_pred_ij == 1, self.classes[i],
                                     self.classes[j])

                Y_pred_ijs.append(Y_pred_ij)
                Y_score_ijs.append(np.abs(Y_score_ij))

        return self.ovo_svm_prediction(np.hstack(Y_pred_ijs),
                                       np.hstack(Y_score_ijs))
```

Training the model Below is the accuracy obtained on each of the one vs one svm classifiers

[37]:
```python
ovo_gsvm = OvO_GSVM()
ovo_gsvm.train(X_train, Y_train, X_test, Y_test, C, gamma, threshold)
```

```
0-1 Accuracy : 0.9981087470449173
0-2 Accuracy : 0.9960238568588469
0-3 Accuracy : 0.9979899497487437
0-4 Accuracy : 0.9984709480122325
0-5 Accuracy : 0.9951923076923077
0-6 Accuracy : 0.9927760577915377
0-7 Accuracy : 0.9970119521912351
0-8 Accuracy : 0.9943705220061413
0-9 Accuracy : 0.9914529914529915
1-2 Accuracy : 0.9958467928011075
1-3 Accuracy : 0.9967365967365968
1-4 Accuracy : 0.9971658006613132
1-5 Accuracy : 0.9960532807104094
1-6 Accuracy : 0.9947443860487338
1-7 Accuracy : 0.9953767914932964
1-8 Accuracy : 0.9952584163110479
1-9 Accuracy : 0.9953358208955224
2-3 Accuracy : 0.9926542605288933
2-4 Accuracy : 0.9955312810327706
2-5 Accuracy : 0.9963617463617463
2-6 Accuracy : 0.9959798994974874
2-7 Accuracy : 0.9854368932038835
2-8 Accuracy : 0.9900299102691924
```

```
2-9 Accuracy : 0.9911807937285644
3-4 Accuracy : 0.998995983935743
3-5 Accuracy : 0.9926393270241851
3-6 Accuracy : 0.9994918699186992
3-7 Accuracy : 0.9921491658488715
3-8 Accuracy : 0.9899193548387096
3-9 Accuracy : 0.9886082218920258
4-5 Accuracy : 0.9989327641408752
4-6 Accuracy : 0.9938144329896907
4-7 Accuracy : 0.9955223880597015
4-8 Accuracy : 0.9959100204498977
4-9 Accuracy : 0.988950276243094
5-6 Accuracy : 0.9918918918918919
5-7 Accuracy : 0.996875
5-8 Accuracy : 0.992497320471597
5-9 Accuracy : 0.9889531825355077
6-7 Accuracy : 0.999496475327291
6-8 Accuracy : 0.9953416149068323
6-9 Accuracy : 0.9984748347737672
7-8 Accuracy : 0.9925074925074925
7-9 Accuracy : 0.9877270495827197
8-9 Accuracy : 0.9843671205244579
```

The overall accuray of the multiclass classifier

[202]:

	Train time	Test accuracy
OvO CVOPTX	2473.6748 secs	0.9725

2.2.2 OneVsOne LIBSVM

Training

[58]:
```
prob  = svm_problem(Y_train.reshape(-1), X_train)
param = svm_parameter('-t 2 -c 1 -g 0.05 -q')
m = svm_train(prob, param)
```

prediction

[87]:
```
Y_pred_libovo, p_acc, p_vals = svm_predict(Y_test.reshape(-1), X_test, m)
```

```
Accuracy = 97.23% (9723/10000) (classification)
```

comparision

[61]:

	cvoptx	libsvm
Training time	2473.6740	417.42
Test accuracy	0.9725	97.23

Both the accuracies are almost the same, but training time difference is huge.

2.2.3 Confusion matrix & Comparision

Code for finding the most misclassified class

[9]:
```
def most_misclassified(conf_matrix):
    mask = np.ones(conf_matrix.shape, dtype=bool)
    np.fill_diagonal(mask, 0)

    max_count = np.max(conf_matrix[mask])
    return np.where(conf_matrix == max_count)
```

Binary linear CVOPTX

```
- Binary linear CVOPTX

Confusion Matrix

Predicted :   -1     1
Actual :
-1           978     4
 1             6  1004

Most misclassified : (array([1]), array([0]))
```

Binary gaussian CVOPTX

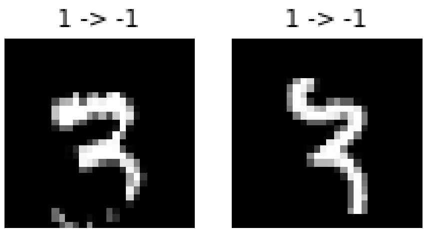
```
- Binary gaussian CVOPTX

Confusion Matrix

Predicted :   -1     1
Actual :
```

```
-1            982     0
 1              2  1008
```

Most misclassified : (array([1]), array([0]))



Binary linear LIBSVM

- Binary linear LIBSVM

Confusion Matrix

```
Predicted :   -1     1
Actual :
-1            978     4
 1              6  1004
```

Most misclassified : (array([1]), array([0]))

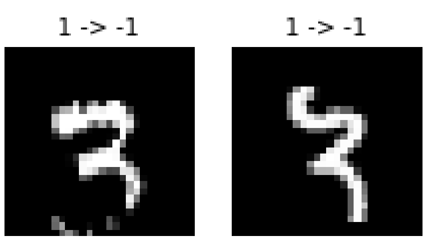
Binary gaussian LIBSVM

```
- Binary gaussian LIBSVM

Confusion Matrix

Predicted :   -1     1
Actual :
-1           982     0
 1             2  1008

Most misclassified : (array([1]), array([0]))
```



Multiclass CVOPTX

[25]:

Predicted :	0	1	2	3	4	5	6	7	8	9
Actual :										
0	969	0	1	0	0	3	4	1	2	0
1	0	1122	3	2	0	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	3	5	1	3	942	3
9	5	4	3	8	13	3	0	8	12	953

Here 7 is being misclassified as 2 most often.

The result makes sense, as in the first image it looks like 1 but is actually 9.

Multiclass LIBSVM

[90]:

Predicted :	0	1	2	3	4	5	6	7	8	9
Actual :										
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

Here 7 is being misclassified as 2 most often.

2.2.4 Cross validation

Code for K fold cross validation

[14]:
```python
def Kfold_validation_C(X_train, Y_train, C, K=5):
    accuracies = {}
    batch_size = int(np.ceil(X_train.shape[0]/K))

    for c in C:
        print(f"Parameter value : {c}")

        fold_accuracy = []
        for k in range(K):
            val_start = k*batch_size
            val_end = (k+1)*batch_size

            X_val = X_train[val_start:val_end]
            y_val = Y_train[val_start:val_end]

            X_train_fold = np.vstack([X_train[:val_start], X_train[val_end:]])
            y_train_fold = np.vstack([Y_train[:val_start], Y_train[val_end:]])

            prob  = svm_problem(y_train_fold.reshape(-1), X_train_fold)
            param = svm_parameter(f'-t 2 -c {c} -g 0.05')
```

```
        m = svm_train(prob, param)
        p_labels, p_acc, p_vals = svm_predict(y_val.reshape(-1), X_val, m)
        fold_accuracy.append(p_acc[0])

    accuracies[c] = fold_accuracy
return accuracies
```

The following validation accuray was obtained for each fold for each value of the parameter in the cross validation process.

```
Parameter value : 1e-05
Accuracy = 9.4% (376/4000) (classification)
Accuracy = 9.3% (372/4000) (classification)
Accuracy = 15.725% (629/4000) (classification)
Accuracy = 17.15% (686/4000) (classification)
Accuracy = 8.625% (345/4000) (classification)
Parameter value : 0.001
Accuracy = 9.4% (376/4000) (classification)
Accuracy = 9.3% (372/4000) (classification)
Accuracy = 15.725% (629/4000) (classification)
Accuracy = 17.15% (686/4000) (classification)
Accuracy = 8.625% (345/4000) (classification)
Parameter value : 1
Accuracy = 97.2% (3888/4000) (classification)
Accuracy = 97.175% (3887/4000) (classification)
Accuracy = 97.2% (3888/4000) (classification)
Accuracy = 97.475% (3899/4000) (classification)
Accuracy = 97.525% (3901/4000) (classification)
Parameter value : 5
Accuracy = 97.2% (3888/4000) (classification)
Accuracy = 97.325% (3893/4000) (classification)
Accuracy = 97.45% (3898/4000) (classification)
Accuracy = 97.55% (3902/4000) (classification)
Accuracy = 97.625% (3905/4000) (classification)
Parameter value : 10
Accuracy = 97.2% (3888/4000) (classification)
Accuracy = 97.325% (3893/4000) (classification)
Accuracy = 97.45% (3898/4000) (classification)
Accuracy = 97.55% (3902/4000) (classification)
Accuracy = 97.625% (3905/4000) (classification)
```

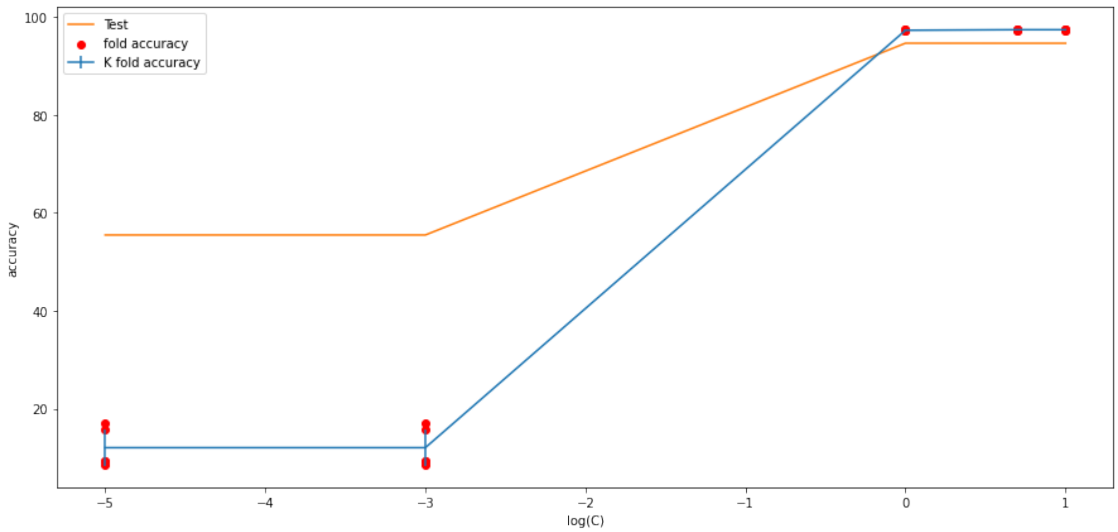
The following accuray was obtained on the test dataset.

```
Parameter value : 1e-05
Accuracy = 55.49% (5549/10000) (classification)
Parameter value : 0.001
Accuracy = 55.49% (5549/10000) (classification)
Parameter value : 1
Accuracy = 94.68% (9468/10000) (classification)
```

```
Parameter value : 5
Accuracy = 94.67% (9467/10000) (classification)
Parameter value : 10
Accuracy = 94.67% (9467/10000) (classification)
```

Plot of the K cross validation accuracy and corresponding test accuracy

[88]: <matplotlib.legend.Legend at 0x7fe7bc05a190>



observation The cross validation accuracy is following the test accuracy trend. * We get best cross validation accuracy C = [1, 5, 10] * We get best test accuracy C = [1, 5, 10]. * Yes, we get the same value of C for both best accuracies.