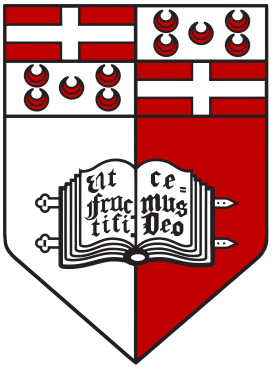

ICS3206 - Machine Learning, Expert Systems, and Fuzzy Logic

Course Project 2017

Full name: Miguel Dingli
I.D: 49997M
Course: B.Sc. (Hons) in Computing Science

FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY

Declaration

Plagiarism is defined as "the unacknowledged use, as one's own, of work of another person, whether or not such work has been published, and as may be further elaborated in Faculty or University guidelines" (University Assessment Regulations, 2009, Regulation 39 (b)(i), University of Malta).

I / ~~We*~~, the undersigned, declare that the [assignment / ~~Assigned Practical Task report / Final Year Project report~~] submitted is my / ~~our*~~ work, except where acknowledged and referenced.

I / ~~We*~~ understand that the penalties for committing a breach of the regulations include loss of marks; cancellation of examination results; enforced suspension of studies; or expulsion from the degree programme.

Work submitted without this signed declaration will not be corrected, and will be given zero marks.

* Delete as appropriate.

(N. B. If the assignment is meant to be submitted anonymously, please sign this form and submit it to the Departmental Officer separately from the assignment).

Miguel Dingli	M. Dingli
Student Name	Signature
Student Name	Signature
Student Name	Signature
Student Name	Signature

ICS3206	ICS3206 - Machine Learning, Expert Systems, and Fuzzy Logic Course Project 2017
Course Code	Title of work submitted

02/01/2018

Date

Table of Contents

Statement of Completion

Item	**Completed (Yes/No/Partial)**
Implementation of bag of words model	Yes
Implementation of SVM	Yes
Used cross validation for parameter tuning	Yes
Experiments and their evaluation	Yes
Overall conclusions	Yes

1 – Introduction

In this assignment, a spam filter will be constructed by considering the messages in the public spam filtering dataset available at http://spamassassin.apache.org/old/publiccorpus/ to come up with a model that is able to classify new messages as being either ham (i.e. non-spam), or spam.

Using the bag of words model, the messages will be converted to a simpler form and a subset of these messages, i.e. the training set, will be used to train Support Vector Machines (SVMs). In general, the SVMs will use machine learning algorithms to produce the above-mentioned model used to classify new messages. From the remaining messages, another subset, i.e. the testing set, will be used to test the accuracy of the classification.

The different types of SVMs that will be explored vary by the kernel used, e.g. linear kernel, and the parameters set for the SVM, some of which directly relate to the kernel being used. By going through different kernels and parameters and using a cross-validation process, one out of the many configurations will be identified as the best configuration to use such that the SVM produces the optimal model for the purpose of spam filtering

2 – Bag of Words Model

The Bag of Words model [1, 2] is used to represent text data instances in a simpler form so that these can be easily processed by machine learning algorithms. Before operating on the text data, the initial step is to come up with a dictionary of words of interest typically formed by going through all of the text instances that will be used in the training process and taking note of all the unique words.

Next, in a feature extraction process, each text is typically transformed into a vector of integers (features) representing the number of times each unique word appears in a document, i.e. the term frequency. This is done in accordance with the defined dictionary, i.e. the n^{th} count in the vector is the number of times that the n^{th} dictionary word is present in the text. In this conversion, the order and structure of the words are eliminated given that these do not help in determining from which category a document originated.

To reduce the space requirements of the resultant features and to improve overall performance and accuracy, stemming or lemmatization can be applied to the words so that these are converted to a more basic form (i.e. stems and lemmas, respectively), meaning that some words (and hence their counts) can be joined together.

Usually, the feature extraction is followed by a normalization process called tf-idf whereby the term frequencies are converted to weights based on the inverse of the document frequency. This eliminates the issue that common words such as “a”, “the”, “and” have a very high frequency compared to other less common words which might be of much more importance with respect to classifying what the document consists of.

Example: consider the following three texts:

1. “This assignment is about spam filtering”
2. “This assignment is about support vector machines”
3. “This assignment is about spam filtering and support vector machines and spam filtering”

The dictionary resulting from these three texts is the following:

- [this, assignment, is, about, spam, filtering, support, vector, machines, and]

Thus, the features resulting from the three texts and the dictionary are the following:

1. [**1**, **1**, **1**, **1**, **1**, **1**, 0, 0, 0, 0]
2. [**1**, **1**, **1**, **1**, 0, 0, **1**, **1**, **1**, 0]
3. [**1**, **1**, **1**, **1**, **2**, **2**, **1**, **1**, **1**, **2**]

3 – SVMs

Support Vector Machines (SVMs) [3, 4] are machine learning models based on the concept of decision planes. They are a supervised learning form of learning, meaning that they deal with labelled data, which is data for which we know which category or class each instance in the data belongs to. Besides classification of data, SVMs are also used in regression analysis and outlier detection.

SVMs are capable of performing both linear and non-linear classification. In general, given a collection of labelled multi-dimensional training data lying in an n-dimensional space, the SVM comes up with an optimal hyperplane which splits the training data into the correct categories. The hyperplane should also be able to classify new unseen examples into a category, based on which side of the hyperplane the examples fall.

The accuracy of a classification of a new example is dependent on many aspects. Firstly, it depends on the size of the training set, since the more training examples the SVM is provided with, the more refined the resultant hyperplane will be. It is also dependent on the gap between the categories of training examples. The wider the gap between training examples belonging to different categories, the more accurate the hyperplane be.

Furthermore, the effectiveness of SVM also depends on the selection of an SVM kernel and the kernel parameters. The best combination of the parameters is selected by performing cross-validation, where each combination of the provided parameters is checked and the combinations resulting in the highest score are selected.

Note that when the training data is not labelled, an unsupervised learning approach is used instead. This approach, called support vector clustering, attempts to find natural clustering of the data to groups, and then map new data to these formed groups. This process is called support vector clustering.

4 – Setup and Implementation

Note that the corpus is expected to be located in **`Resources/publiccorpus/`** from the working directory. The expected folders inside this directory are **`easy_ham`**, **`easy_ham_2`**, **`hard_ham`**, **`spam`**, and **`spam_2`**. Furthermore, the program takes a single argument which will be mentioned in section 4.1.

Among the modules imported, two important ones are `email` and `sklearn`. The `email` module will be used to parse the message files found in the corpus, so that rather than using the whole file, there is more control over which parts of the message are included or excluded in the learning process that follows . The `sklearn` module includes various important machine learning components which will be used throughout the program.

In the following subsections, snippets from the script along with descriptions will be presented.

4.1 – Initializing

The initialization part of the program consists of multiple steps, which will be presented as a bulleted list:

- Firstly, the directory containing spam filtering dataset is set as `Resources/publiccorpus/`.
- The `CATEGORY_MAP` sets each folder's category to distinguish between *ham* (0) and *spam* (1). This map also indicates the folders that are expected to be found in the directory `RESOURCES_DIR`.
- The `CATEGORY_NAMES` list contains the two main category names, *ham* (at 0) and *spam* (at 1).

```
# Main directory of corpus
RESOURCES_DIR = "Resources/publiccorpus/"

# Categories
CATEGORY_MAP = {
    'easy_ham': 0, 'easy_ham_2': 0, 'hard_ham': 0, 'spam': 1, 'spam_2': 1
}
CATEGORY_NAMES = ['ham', 'spam']
```

- Next, the number of training examples *per category* `TRN_PER_CAT` is set as the first (and only) program argument. For the maximum number of testing examples, `TST_PER_CAT` is set to -1, so that after the training examples are picked, all remaining messages will be used for the testing part.
- The lists `trn_strs` and `tst_strs` will hold the training and testing examples in string form.
- Two other lists `trn_cats` and `tst_cats` will hold the category of each training and testing example.

```
# Setting sizes of training and testing set
ALL = -1                            # used to indicate a maximum test set size
TRAIN_PER_CAT = int(sys.argv[1])    # training samples from each category
TESTS_PER_CAT = ALL                 # testing samples from each category

# Lists for training and testing msgs and their categories
train_strings = []    # list of training messages
tests_strings = []    # list of testing messages
train_cats = []       # corresponding categories of training messages
tests_cats = []       # corresponding categories of testing messages
```

- Lastly, two tools which for the bag of words process are defined. Given a list of strings, which in this case will be a list messages, the `CountVectorizer` produces a list of features in the form of a vector of term frequencies. This is essentially the feature extraction part of the bag of words model.
- The `TfidfTransformer` tool will apply tf-idf (refer to section 2) to the resultant list of features.

```
# Count vectorizer and tf-idf transformer
count_vectorizer = CountVectorizer()
tfidf_transformer = TfidfTransformer()
```

4.2 – Bag of Words

In this section, the snippets corresponding to the bag of words task will be presented.

4.2.1 – Function to Convert Message to String

As previously discussed, the `email` module was imported so as to have more control over the parsed message.

The following function takes a message parsed by this module and converts it to a string. In the case that the payload of a message contains multiple parts, the function is recursively applied to the sub-parts.

It was decided that only parts having a `text/plain` or `text/html` content type are to be included. In the former, the content was converted to lowercase, for standardisation. In the latter, however, the content was replaced with just "html" so that the resultant string is not contaminated with a lot of `html` tags, but the fact that `html` was present is acknowledged, since this is significant in distinguishing between *ham* and *spam* [?].

```
# Converts a message to a string
# - Plain parts are converted to lowercase
# - Html  parts are converted to "html"
# - Other parts are ignored
def msg_to_string(msg):
    if msg.is_multipart():  # convert each sub-part and join all strings together
        text_list = list(map(msg_to_string, msg.get_payload()))
        return (" ").join(text_list)
    elif msg.get_content_type() == 'text/plain':
        return msg.get_payload().lower()
    elif msg.get_content_type() == 'text/html':
        return "html"
    else:
        return ""
```

4.2.2 – Function to Convert to String and Add to Set

The second function to be discussed is `convert_and_add_files`. For use in this function, two lists to keep track of the number of training and testing examples (respectively) collected for each category were defined.

Given the name of a folder from the corpus directory, the function goes through each message in the folder in random order and adds the string obtained using `msg_to_string` to either the list of training or testing examples. The message is added to the list of testing examples only if the training-examples-per-category quota for the particular category has been reached.

```
# List to keep track of no. of train/test examples collected for each category
train_counts = [0] * len(CATEGORY_NAMES)
tests_counts = [0] * len(CATEGORY_NAMES)

# Converts each message in a folder to a string and adds to training/testing sets
def convert_and_add_files(folder_name):
    # Obtain category of folder and folder's directory
    folder_cat = CATEGORY_MAP[folder_name]
    folder_dir = os.path.join(RESOURCES_DIR, folder_name)

    # Go through all files randomly and fill up training and testing sets
    files_list = os.listdir(folder_dir)
    for file in sample(files_list, len(files_list)):

        # Open message file and convert the message to a string
        with open(os.path.join(folder_dir, file), 'r', encoding='ISO-8859-1') as
         input_file:
            msg_text = msg_to_string(email.message_from_file(input_file))
            input_file.close()

        # If training set filled up, start filling up testing set
        if train_counts[folder_cat] < TRAIN_PER_CAT:
            train_strings.append(msg_text)
            train_cats.append(folder_cat)
            train_counts[folder_cat] += 1
        elif tests_counts[folder_cat] < TESTS_PER_CAT or TESTS_PER_CAT == ALL:
            tests_strings.append(msg_text)
            tests_cats.append(folder_cat)
            tests_counts[folder_cat] += 1
        else:
            break  # training and testing set filled up
```

4.2.3 – Processing Each Folder

The following code simply applies the discussed `convert_and_add_files` to each folder expected to be in the dataset's main directory. Afterwards, the actual number of training and testing examples are computed.

```
# For each folder in main directory, convert all files in that folder
print("Going through folders...")
for folder in CATEGORY_MAP.keys():
    convert_and_add_files(folder)
    print(folder + "...DONE.")

# Total actual size of training and testing sets
total_train = len(train_strings)
total_tests = len(tests_strings)
```

4.2.4 – Count Vectorizer and TF-IDF Transformation

Finally, the `CountVectorizer` is applied on both the list of training and testing examples. The output of the vectorizer is fed into the `TfidfTransformer` to apply tf-idf to the training and testing examples. Refer to sections 2 and 4.1 for discussions about `TfidfTransformer` and `CountVectorizer`, respectively.

```
# For training set: obtain bag of counts and apply tfidf
X_train = count_vectorizer.fit_transform(train_strings)
X_train = tfidf_transformer.fit_transform(X_train)

# For testing set: obtain bag of counts and apply tfidf
X_tests = count_vectorizer.transform(tests_strings)
X_tests = tfidf_transformer.transform(X_tests)
```

4.3 – Training SVMs

In this section, the implementation for the training of SVMs will be presented. The three SVM kernels picked are the **linear** (`linear`), **radial basis function** (`rbf`), and **polynomial** (`poly`). The kernel parameters were picked based on running the program and analysing the cross-validation results for these three kernels.

In the below snippet, each SVM is fitted to the training data. Afterwards, a score for each SVM is obtained by passing the testing data. The score indicates the fraction of test examples that were correctly classified by the SVM. This is converted to a percentage and output.

```
# Fitting SVMs
svms = [
    SVC(kernel='linear', C=1).fit(X_train, train_cats),
    SVC(kernel='rbf', C=100, gamma=0.01).fit(X_train, train_cats),
    SVC(kernel='poly', C=1, gamma=0.0001, degree=8).fit(X_train, train_cats)
]
# Calculating scores
for svm in svms:
    kernel = svm.get_params()['kernel']
    score = svm.score(X_tests, tests_cats)
    accuracy = score * 100
    correct = score * total_tests
    print("Accuracy when using %s kernel: %.2f%% (%d out of %d)" % (kernel, accuracy,
        correct, total_tests))
```

4.4 – Cross-Validation

In the below snippet, the implementation for cross-validation is presented, where GridSearchCV was used. Given a parameter grid (a dictionary of parameters), GridSearchCV goes through every combination of parameters and applies k-fold cross-validation, where the number of subsets k is defined by its argument cv. From the GridSearchCV result, the optimal parameters are obtained simply by accessing the `best_params_` field.

```
# Parameters for GridSearch
param_grid = [
    {'kernel': ['linear'], 'C': [1, 1e1, 1e2, 1e3]},
    {'kernel': ['rbf'],    'C': [1, 1e1, 1e2, 1e3], 'gamma': [1e-4, 1e-3, 1e-2]},
    {'kernel': ['poly'],   'C': [1, 1e1, 1e2, 1e3], 'gamma': [1e-4, 1e-3, 1e-2],
        'degree': [2, 8, 10]},
]

# Perform GridSearch
print("Performing cross-validation...")
cv = GridSearchCV(SVC(), param_grid, cv=5, n_jobs=1).fit(X_train, train_cats)
print("...cross-validation done.")

# Print all GridSearch scores
print("GridSearch scores:")
means = cv.cv_results_['mean_test_score']
stdevs = cv.cv_results_['std_test_score']
params = cv.cv_results_['params']
for mean, stdev, param in zip(means, stdevs, params):
    print("%0.3f (+/-%0.03f) for %r" % (mean, stdev * 2, param))

# Print best parameters for greatest score
print("\nBest parameters: %r" % cv.best_params_)
```

5 – Experiments and Results

For this section, various experiments were performed and are presented below:

- **Experiment 1**: Observing the effects of increasing the training set size on the prediction scores.
    - **Setup**: the testing set size was kept at 2000 (1000 from each category) whilst the training set sizes were 20, 100, 1000, and 1700 (10, 50, 500, and 850 from each category, respectively). Note that for a greater training set size, there would not be enough examples left for 2000 testing examples. The kernel parameters used are the ones presented in the code snippet of the previous section.
    - **Expected results**: with a larger training set size, the score should be higher.
    - **Actual results**: based on the mean of three readings for each result (full results in Appendix 1), the below table shows that the score for each kernel increases with the number of training examples.

	Linear kernel score	RBF kernel score	Polynomial kernel score
20 training examples	78.65%	84.85%	87.33%
100 training examples	94.78%	95.52%	88.92%
1000 training examples	96.15%	96.30%	90.40%
1700 training examples	98.92%	99.00%	94.57%

    - **Conclusions**: as expected, the scores increased when the training set size was increased.

- **Experiment 2**: Observing the effect of increasing the value of SVM parameter C on the prediction scores.
    - **Setup**: the testing set size was kept as 2000 (1000 from each) whilst the training set size was kept at 200 (100 from each). For each of the three kernels, the C values used were 1, 10, 100, and 1000, whilst all other parameters were kept as shown in the code snippet of the previous section.
    - **Expected results**: since larger values of C means a more accurate resultant hyperplane, the scores are expected to be higher the greater the value of C. However, for very large values of C, the accuracy could suffer due to overly strict fitting of the models.
    - **Actual results**: based on the mean of three readings for each result (full results in Appendix 2), the below table shows some expected as well as unexpected results. In the linear kernel's case, the results are very close to each other, meaning that no real conclusion can be made. In the RBF and polynomial kernels' case, the accuracy increased drastically from C=10 to C=100. However, an un-expected result which could be due to the type of kernel is the decrease from C=1 to C=10 for the polynomial kernel. In general, from C=100 to C=1000, there was a decrease in accuracy, which can be attributed to the overly strict fitting of the models but might also be caused by result error.

	Linear kernel score	RBF kernel score	Polynomial kernel score
C=1	95.80%	72.80%	89.37%
C=10	96.07%	73.52%	84.25%
C=100	95.87%	95.88%	91.77%
C=1000	94.40%	95.12%	86.17%

    - **Conclusions**: the significant increase in accuracy from C=10 to C=100 for the majority of kernels indicates that the accuracy does increase with the value of C. However, the result did include various unexpected aspects which one should keep in mind.

- **Experiment 3**: Observing the effect of using hard_ham as training data on the prediction scores.

    - **Setup:** in the first half of the experiment, the order of the mappings in the CATEGORY_MAP were left as default, i.e. easy_ham, easy_ham_2, hard_ham, spam, and spam_2. In the second half, the order was changed such that hard_ham was placed first followed by easy_ham, easy_ham_2, and so on. The training and testing set sizes were kept at a constant 500 and 2000 (250 and 1000 from each category), respectively. The value of '250' examples from each category is very important, especially in the second half, since the number of messages in the hard_ham folder is 250, meaning that all of these messages will be used as the training data representing ham messages.

    - **Expected results**: since the hard_ham folder contains ham messages that are not as easily classifiable when compared to easy_ham, the accuracy of the model is expected to decrease significantly due to the factors that distinguish ham from spam being less obvious to the model.

    - **Actual results**: based on the mean of three readings for each result (full results in Appendix 3), the below table clearly shows the expected results. Since hard_ham provides less information about what a non-spam message should look like, the accuracy using any of the three kernels decreased drastically when hard_ham was used as ham training data instead of easy_ham.

	Linear kernel score	**RBF kernel score**	**Polynomial kernel score**
CATEGORY_MAP = { 'easy_ham': 0, 'easy_ham_2': 0, 'hard_ham': 0, 'spam': 1, 'spam_2': 1}	96.55%	96.70%	91.10%
CATEGORY_MAP = { 'hard_ham': 0, 'easy_ham': 0, 'easy_ham_2': 0, 'spam': 1, 'spam_2': 1}	67.82%	68.53%	64.03%

    - **Conclusions**: this experiment has shown that it is of paramount importance to use training data with features that enable easy classification of the data into the actual categories.

6 – Conclusions

Overall, the experiments showed how important it is to select a suitable training set size and a training set in which the examples best describe the category that they belong to. The penalty value C also plays an important role, but its importance seems to vary across kernels.

7 – References

[1] A Gentle Introduction to the Bag-of-Words Model. Available: https://machinelearningmastery.com/gentle-introduction-bag-words-model/.

[2] Bag of Words Natural Language Processing. Available: https://ongspxm.github.io/blog/2014/12/bag-of-words-natural-language-processing/.

[3] Introduction to Support Vector Machines. Available: https://docs.opencv.org/2.4/doc/tutorials/ml/introduction_to_svm/introduction_to_svm.html.

[4] Support Vector Machines (SVM). Available: http://www.statsoft.com/Textbook/Support-Vector-Machines.

8 – Appendix 1 – Experiment 1 Results

For 10 training examples per category:

1. Accuracy when using linear kernel: 72.70% (1454 out of 2000)
   Accuracy when using rbf kernel: 79.25% (1585 out of 2000)
   Accuracy when using poly kernel: 88.15% (1763 out of 2000)

2. Accuracy when using linear kernel: 87.75% (1755 out of 2000)
   Accuracy when using rbf kernel: 90.00% (1800 out of 2000)
   Accuracy when using poly kernel: 87.60% (1752 out of 2000)

3. Accuracy when using linear kernel: 75.50% (1510 out of 2000)
   Accuracy when using rbf kernel: 85.30% (1706 out of 2000)
   Accuracy when using poly kernel: 86.25% (1725 out of 2000)

For 50 training examples per category:

1. Accuracy when using linear kernel: 95.90% (1918 out of 2000)
   Accuracy when using rbf kernel: 96.20% (1924 out of 2000)
   Accuracy when using poly kernel: 87.50% (1750 out of 2000)

2. Accuracy when using linear kernel: 93.35% (1867 out of 2000)
   Accuracy when using rbf kernel: 95.00% (1900 out of 2000)
   Accuracy when using poly kernel: 93.45% (1869 out of 2000)

3. Accuracy when using linear kernel: 95.10% (1902 out of 2000)
   Accuracy when using rbf kernel: 95.35% (1907 out of 2000)
   Accuracy when using poly kernel: 85.80% (1716 out of 2000)

For 500 training examples per category:

1. Accuracy when using linear kernel: 96.55% (1931 out of 2000)
   Accuracy when using rbf kernel: 96.70% (1934 out of 2000)
   Accuracy when using poly kernel: 89.80% (1796 out of 2000)

2. Accuracy when using linear kernel: 96.10% (1922 out of 2000)
   Accuracy when using rbf kernel: 96.30% (1926 out of 2000)
   Accuracy when using poly kernel: 90.00% (1800 out of 2000)

3. Accuracy when using linear kernel: 95.80% (1916 out of 2000)
   Accuracy when using rbf kernel: 96.00% (1920 out of 2000)
   Accuracy when using poly kernel: 91.50% (1830 out of 2000)

For 850 training examples per category:

1. Accuracy when using linear kernel: 98.30% (1966 out of 2000)
   Accuracy when using rbf kernel: 98.45% (1969 out of 2000)
   Accuracy when using poly kernel: 92.45% (1849 out of 2000)

Kindly refer to 2 and 3 overleaf.

2. Accuracy when using linear kernel: 99.55% (1991 out of 2000)
   Accuracy when using rbf kernel: 99.60% (1992 out of 2000)
   Accuracy when using poly kernel: 95.05% (1901 out of 2000)

3. Accuracy when using linear kernel: 98.90% (1978 out of 2000)
   Accuracy when using rbf kernel: 98.95% (1979 out of 2000)
   Accuracy when using poly kernel: 96.20% (1924 out of 2000)

9 – Appendix 2 – Experiment 2 Results

For C = 1:

1. Accuracy when using linear kernel: 96.60% (1932 out of 2000)
   Accuracy when using rbf kernel: 72.25% (1445 out of 2000)
   Accuracy when using poly kernel: 95.45% (1909 out of 2000)

2. Accuracy when using linear kernel: 95.15% (1903 out of 2000)
   Accuracy when using rbf kernel: 72.70% (1454 out of 2000)
   Accuracy when using poly kernel: 93.35% (1867 out of 2000)

3. Accuracy when using linear kernel: 95.65% (1913 out of 2000)
   Accuracy when using rbf kernel: 73.45% (1469 out of 2000)
   Accuracy when using poly kernel: 79.30% (1586 out of 2000)

For C = 10:

1. Accuracy when using linear kernel: 96.45% (1929 out of 2000)
   Accuracy when using rbf kernel: 72.25% (1445 out of 2000)
   Accuracy when using poly kernel: 83.85% (1677 out of 2000)

2. Accuracy when using linear kernel: 95.95% (1919 out of 2000)
   Accuracy when using rbf kernel: 72.35% (1447 out of 2000)
   Accuracy when using poly kernel: 84.30% (1686 out of 2000)

3. Accuracy when using linear kernel: 95.80% (1916 out of 2000)
   Accuracy when using rbf kernel: 75.95% (1519 out of 2000)
   Accuracy when using poly kernel: 84.60% (1692 out of 2000)

For C = 100:

1. Accuracy when using linear kernel: 96.35% (1927 out of 2000)
   Accuracy when using rbf kernel: 96.20% (1924 out of 2000)
   Accuracy when using poly kernel: 87.90% (1758 out of 2000)

2. Accuracy when using linear kernel: 95.45% (1909 out of 2000)
   Accuracy when using rbf kernel: 95.45% (1909 out of 2000)
   Accuracy when using poly kernel: 94.90% (1898 out of 2000)

3. Accuracy when using linear kernel: 95.80% (1916 out of 2000)
   Accuracy when using rbf kernel: 96.00% (1920 out of 2000)
   Accuracy when using poly kernel: 92.50% (1850 out of 2000)

For C = 1000:

1. Accuracy when using linear kernel: 94.65% (1893 out of 2000)
   Accuracy when using rbf kernel: 94.65% (1893 out of 2000)
   Accuracy when using poly kernel: 77.70% (1554 out of 2000)

2. Accuracy when using linear kernel: 93.30% (1866 out of 2000)
   Accuracy when using rbf kernel: 93.25% (1865 out of 2000)
   Accuracy when using poly kernel: 90.20% (1804 out of 2000)

3. Accuracy when using linear kernel: 95.25% (1905 out of 2000)
   Accuracy when using rbf kernel: 97.45% (1949 out of 2000)
   Accuracy when using poly kernel: 90.60% (1812 out of 2000)

10 – Appendix 3 – Experiment 3 Results

For `CATEGORY_MAP = {'easy_ham':0, 'easy_ham_2':0, 'hard_ham':0, 'spam':1, 'spam_2':1}`:

1. Accuracy when using linear kernel: 97.05% (1941 out of 2000)
   Accuracy when using rbf kernel: 97.30% (1946 out of 2000)
   Accuracy when using poly kernel: 88.00% (1760 out of 2000)

2. Accuracy when using linear kernel: 96.55% (1931 out of 2000)
   Accuracy when using rbf kernel: 96.60% (1932 out of 2000)
   Accuracy when using poly kernel: 94.50% (1890 out of 2000)

3. Accuracy when using linear kernel: 96.05% (1921 out of 2000)
   Accuracy when using rbf kernel: 96.20% (1924 out of 2000)
   Accuracy when using poly kernel: 90.80% (1816 out of 2000)

For `CATEGORY_MAP = {'hard_ham':0, 'easy_ham':0, 'easy_ham_2':0, 'spam':1, 'spam_2':1}`:

1. Accuracy when using linear kernel: 68.30% (1366 out of 2000)
   Accuracy when using rbf kernel: 68.35% (1367 out of 2000)
   Accuracy when using poly kernel: 60.70% (1214 out of 2000)

2. Accuracy when using linear kernel: 67.75% (1355 out of 2000)
   Accuracy when using rbf kernel: 68.60% (1372 out of 2000)
   Accuracy when using poly kernel: 65.95% (1319 out of 2000)

3. Accuracy when using linear kernel: 67.40% (1348 out of 2000)
   Accuracy when using rbf kernel: 68.65% (1373 out of 2000)
   Accuracy when using poly kernel: 65.45% (1309 out of 2000)