Assignment Group 2
Felix Dunér (18 h)
Sara Hillström (18 h)

DAT 405 – Lab 4

1. b) Code for splitting the data in test and training sets (it's on the same line in our code, just structured this way to be visible).

```
X_train, X_test, y_train, y_test =
train_test_split(all_emails.messages, all_emails.labels, test_size=0.35, random_state = 42)
```

2. b) A classifier based on Bernoulli Naïve Bayes focuses on the presence or absence of words in a mail. This presumes that all features are binary, representing if a word either occurs or not. Using this algorithm for classification means ignoring the frequency of which this word in question occurs in the mail being analyzed, something Multinomial Naïve Bayes takes into account. Although Multinomial Naïve Bayes can be seen as an upgrade due to this frequency measurement, it doesn't take any notion about absent words. With it's binary way of working, a classifier using Bernoulli Naïve Bayes utilizes the full vocabulary by noting either presence or absence and therefore takes absent words into account as well.

3. a) **Spam vs. easy ham:**
   Multinomial naive bayes: 90.65 %
   Bernoulli naive bayes: 92.61 %

   b) **Spam vs. hard ham:**
   Multinomial naive bayes: 80.23%
   Bernoulli naive bayes: 85.55 %

   As expected, the model does better on detecting spam vs. 'ham' when we use the easy ham data instead of the hard ham data.

4. a) It's useful to detect and remove common words from the data set since a lot of these words are connective words, such as 'and', 'if' etc. and they are therefore probably not informative enough about the content of an email on their own. Both a spam email and a 'ham' email can contain these kinds of common words, while the subject of the email (and therefore most likely the indicator of it being spam or not) differ substantially. Hence, words that appear frequently are often insignificant for the detection of spam emails.

```
count_vect = CountVectorizer(preprocessor = pre_processor)

#counting freq. of different words
bag_of_words = count_vect.fit_transform(all_emails.messages)
sum_words = bag_of_words.sum(axis=0)
words_freq = [(word, sum_words[0, idx]) for word, idx in count_vect.vocabulary_.items()]
words_freq =sorted(words_freq, key = lambda x: x[1], reverse=True)

#Top 20 most common words
print(f'The 20 most common words and their freq. are: {words_freq[:20]}')

#20 most uncommon words
print(f'The 20 least common words and their freq. are: {words_freq[-20:]}')
```

Code for finding the most and least common words in the email data set

b)

```
#min_df eliminates too uncommon words, max_df remove too frequent words (in % of total)
count_vect = CountVectorizer(preprocessor = pre_processor, max_df=0.8, min_df=0.005, stop_words='english')
```

Code for filtering out unwanted words using CountVectorizer

In 4 a), we looked at the top N/bottom N most common words. When switching N for these tables, we could get a somewhat perception of how many common/uncommon words would be good to filter out before running our final model. Based on the different words we saw, and the reasoning from 4a) on common words not giving any information on their own, we decided that the model should be more sensitive to common words than to uncommon words (min_df removes words occurring in less than .5% of the emails, max_df removes words occurring in more than 85% of the emails). Although there's most likely a big overlap, we decided to use the already existing list of English stop words (common words such as 'the', 'from', 'what' together with all words who occur in more than 80% of the emails.

Other than filtering out common/uncommon words and stop words, we also decided to try and improve our model by removing HTML code, words longer than 25 characters (since this is most likely not an actual word) and punctuations and other special signs. The code for doing so can be seen below, and this method is called in the CountVectorizer.

```
#removing HTML tags and punctuation
def pre_processor(messages):
    messages = messages.lower()
    no_html = re.compile('<.*?>')
    messages = re.sub(no_html, '', messages)
    no_long_words = re.compile('\w{25,}')
    messages = re.sub(no_long_words, '', messages)
    no_special_signs = re.compile('\W')
    messages = re.sub(no_special_signs, ' ', messages)

    return messages
```

Code for preprocessing the email set

The results of this cleaning were:

Spam vs. easy ham:
Multinomial Naive Bayes: 98.32 % vs 90.65 % before cleaning data.
Bernoulli Naive Bayes: 94.86 % vs 92.61 % before cleaning data.

Spam vs. hard ham:
Multinomial Naive Bayes: 87.07 % vs 80.23 % before cleaning data.
Bernoulli Naive Bayes: 85.93 % vs 85.55 % before cleaning data.

5. a) ***Accuracy for different cases***

   Spam vs. easy ham (without headers):
   Multinomial naive bayes: 98.32 % vs 98.22 % with headers.
   Bernoulli naive bayes: 94.86 % vs 98.97 % with headers.

   Spam vs. hard ham (without headers):
   Multinomial naive bayes: 87.07 % vs 85.93% with headers.
   Bernoulli naive bayes: 85.93 % vs 89.35 % with headers.

   When filtering out headers, we saw a pattern that the headers usually were the first paragraph of each email. Therefore, we decided to use .split('\n\n',1), and remove the first paragraph for all emails consisting of more than one paragraph.

```
#Method for removing headers
for message in all_emails.messages:
    temp = message.split("\n\n",1)
    if len(temp) > 1:
        all_emails = all_emails.replace(message,temp[1])
```

When looking for ways of filtering out footers, we couldn't find any trend for detecting footers. We also noticed that a lot of the 'ham' emails have previous emails from the same email chain, which is useful for detecting that it is in fact a 'ham' email. Lastly, a lot of the text in the footers (e.g. signatures and aliases) are often unique/somewhat unique words, and thus it will probably be filtered out by our min_df that filters out words that occur in less than .5% of the emails. With all of this in mind, we decided to keep the footers and focus on the headings instead.

b) Often when we deal with datasets used for binary classification problems like this one, a majority of the samples belong to one of the two possible classes resulting in an imbalanced or skewed dataset. This increases the risk of the model overfitting to the majority class, decreasing in reliability. This imbalance is probably a rather accurate representation of the real-life problem your model is working to solve, which means that although it might not have been sufficiently exposed to the other class in its training and thereby lacks quality in its classification, it might still provide decent results. The reason behind this is rather simple; a model trained with an imbalanced dataset has mostly been trained to classify the majority class, which means that even if it lacks quality in classifying the minority class it will be correct in quite a lot of cases.

Looking at the split of data into a training and a test set using a random splitting method might increase, or even create this imbalance which risks to make a model error prone. In this particular case we have an imbalance in the original dataset with a majority of non-spam mails (i.e. ham), which might be increased by our random splitting into test and training sets.

There are different ways to deal with an imbalanced datasets, with resampling being one of the most common techniques. This can be done in two ways; undersampling and oversampling. Undersampling implies removing random samples from the class to which the majority belongs, which might result in

loss of critical information. On the contrary, oversampling involves creating more samples from the minority class. In its simplest form this is done by creating duplicates of random samples, although this might lead us back to the original problem of overfitting. Resampling should only be used on the training set, and both these methods aim to create a more balanced dataset.

c) Using already mentioned logic with the problem regarding an imbalanced dataset, this might lead to a model which overfit to the spam messages which in this case are the majority class in the training dataset. A model overfitted to the majority class could result in a potential inability to detect and classify the minority class (ham messages). This could have a significant effect on the accuracy (lowering it) if the dataset meant for testing the model consists mostly of samples belonging to this minority class.