

Experiment 3:

1. There are 5572 records in the data set. The distribution of the labels is

Ham	4825
Spam	747

Ham is right skewed; Spam is left skewed and the histogram of sms length is right skewed.

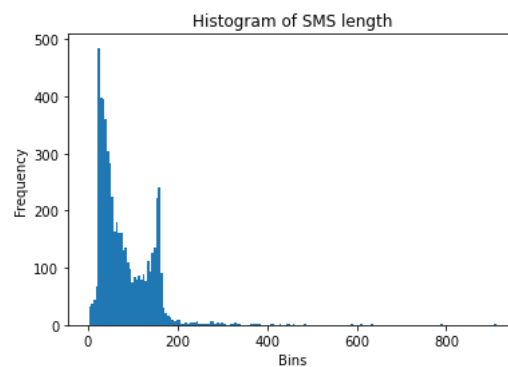
- 2.

5169 unique sms message. Most frequent sms message is “Sorry, I’ll call later” with a frequency of 30.

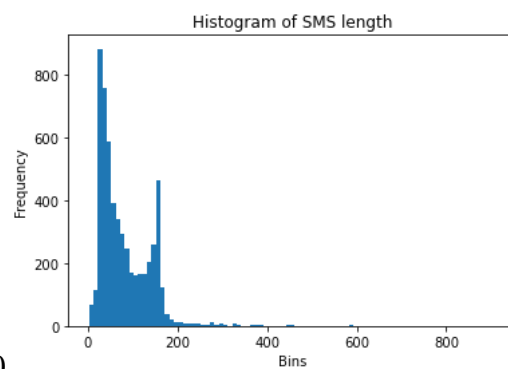
```
In [85]: df_sms.sms.value_counts()

Out[85]: Sorry, I'll call later
30
I cant pick the phone right now. Pls send a message
12
Ok...
```

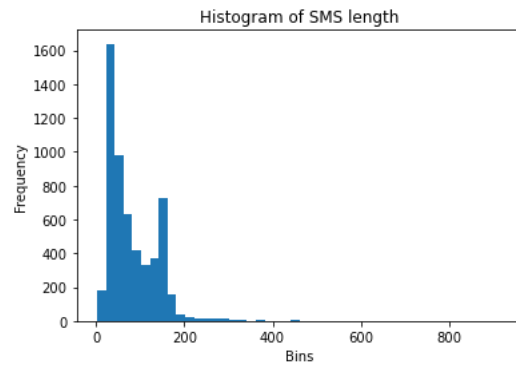
3. Max length of SMS is 910 and Min length of SMS is 2.



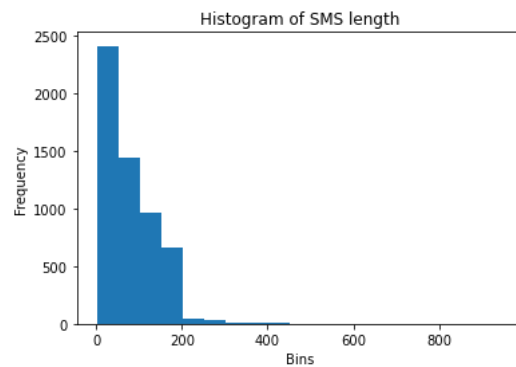
Bin size - 5



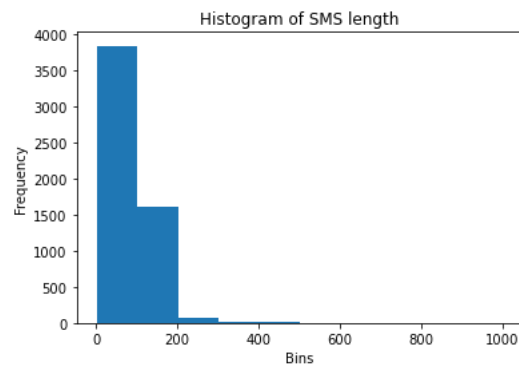
Bin size - 10



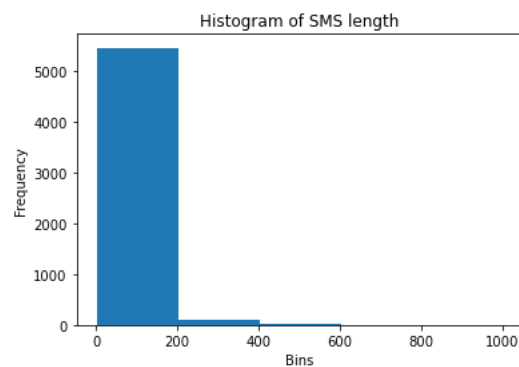
Bin size – 20



Bin size -50



Bin size – 100



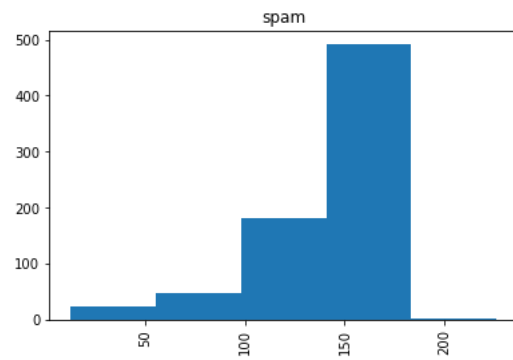
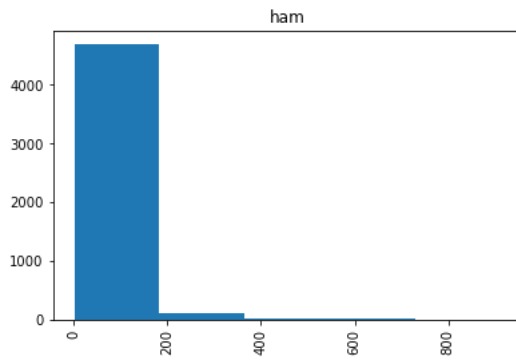
Bin size – 200

The bin width changes the ability of a histogram to identify local regions of higher incidence and the granularity of the data. So as the bin size increases, the histogram

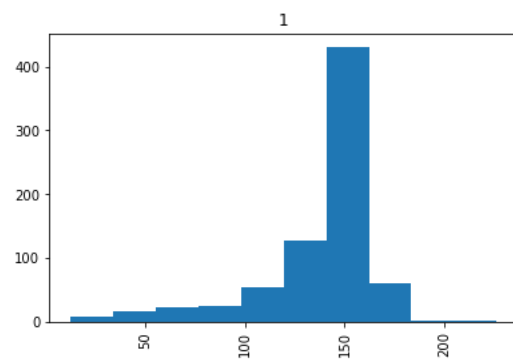
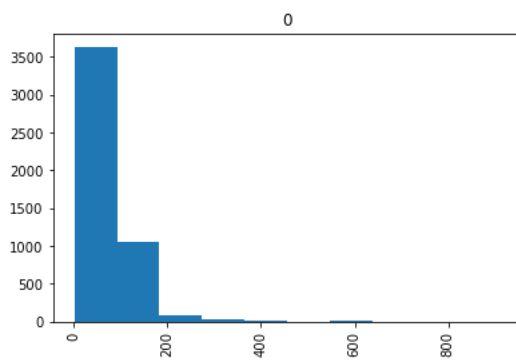
will lose the granularity of data. In particular here, we can observe that as the size increases, we are losing any indication of a second peak.

4.

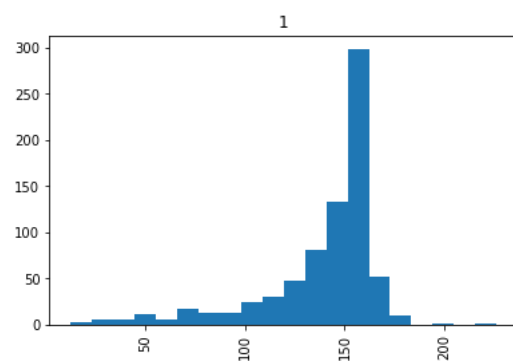
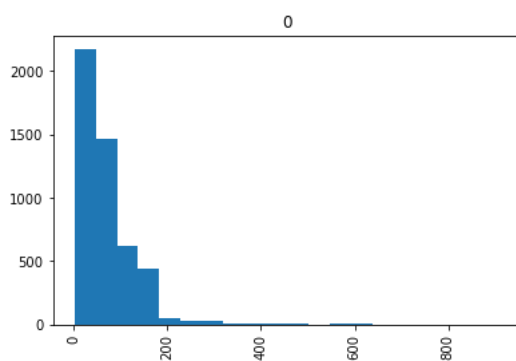
Bins = 5



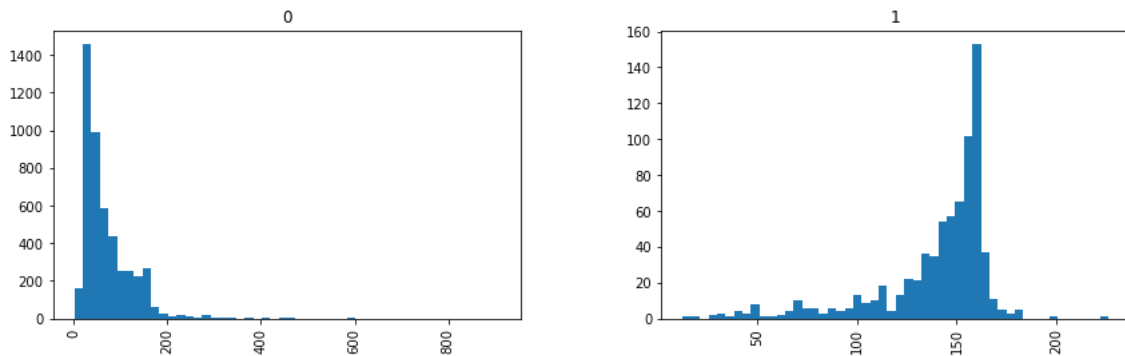
Bins = 10



Bins = 20



Bins = 50



The number of bins increase with decrease in bin width. As bins increase the granularity of the data improves.

5. We convert the words to lowercase to normalize, to avoid duplicate words due to variation in letter case. Say the words “Is” and “is”, due to difference in letter case the two words would be considered as different. Converting all upper-case letters would also work to fulfill the original goal.
6. Count vectorization does the following:
 - i. It tokenizes the string (separates the string into individual words) and gives an integer ID to each token.
 - ii. It then counts the occurrence of each of those tokens.

If we set stop words to English then all the words in the document set that matches the list of the stop words defined by scikit learn will be removed because they are insignificant. Examples of stop words are: ‘a’, ‘as’, ‘at’, ‘be’, ‘by’

7. We first separate the data into train/test and then generate a document-term matrix based on the training dataset and afterward generate a matrix for the test set.

We first initialize the Count Vectorizer method and then fit the training data using “fit_transform” to get the document matrix. We first separate the train and test dataset and then generate the document term matrix because we set the parameters through training and use these settings to generate the document matrix.

8.

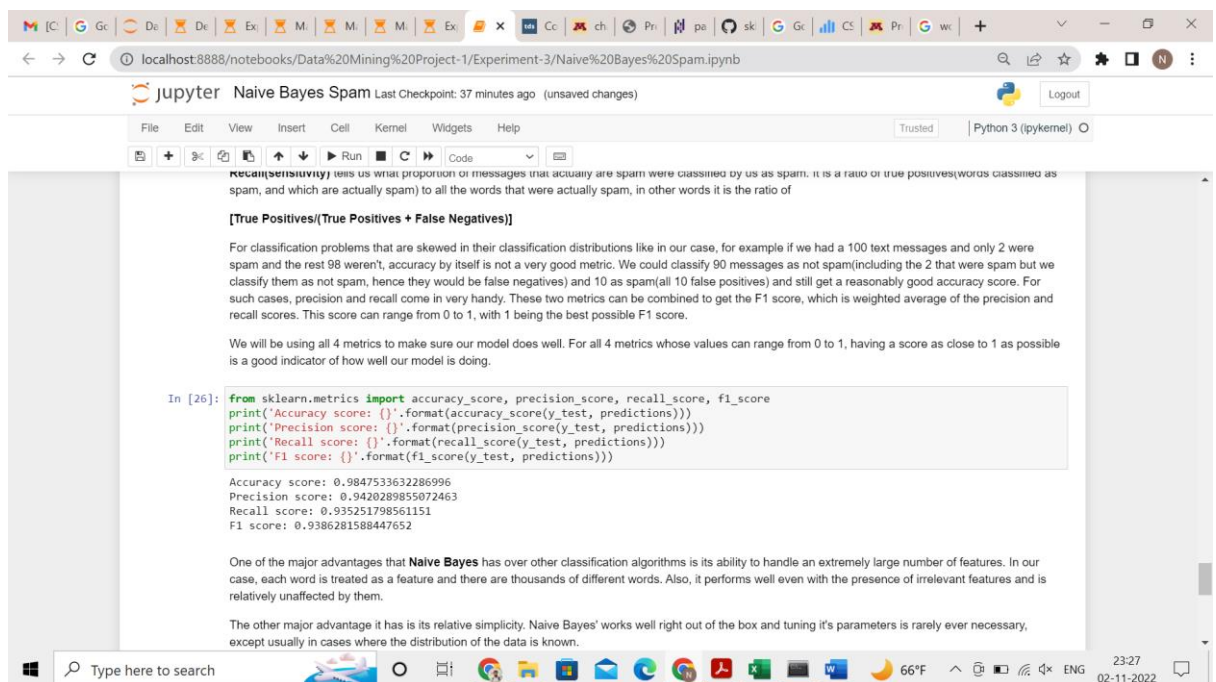
```
documents1 = ['Hi, how are you?', 'Win money, win from home. Call now.', 'Hi., Call you now or tomorrow?']
count_vector.fit(documents1)
count_vector.get_feature_names_out()
doc_array = count_vector.transform(documents).toarray()
frequency_matrix = pd.DataFrame(doc_array, columns = count_vector.get_feature_names_out())
frequency_matrix
```

	are	call	from	hi	home	how	money	now	or	tomorrow	win	you
0	1	0	0	1	0	1	0	0	0	0	0	1
1	0	1	1	0	1	0	1	1	0	0	2	0
2	0	1	0	1	0	0	0	1	1	1	0	1

9. There are 7777 features created while making the document term matrix for the SMS dataset.

The features have to be combined or irrelevant features have to be excluded to reduce the number of features. The advantage is the complexity of the model decreases and the disadvantage is certain features might have their part in the model settings after training.

10. The input data here is discrete (word counts for text classification), hence we should use multinomial Naive Bayes implementation as it is suitable for classification with discrete features



The screenshot shows a Jupyter Notebook interface in a web browser. The browser address bar shows the URL: `localhost:8888/notebooks/Data%20Mining%20Project-1/Experiment-3/Naive%20Bayes%20Spam.ipynb`. The notebook title is "Naive Bayes Spam" and it shows "Last Checkpoint: 37 minutes ago (unsaved changes)". The interface includes a menu bar (File, Edit, View, Insert, Cell, Kernel, Widgets, Help) and a toolbar with icons for file operations, cell execution, and code execution. The notebook content includes a paragraph explaining the F1 score, a code cell with Python code to calculate accuracy, precision, recall, and F1 score using `sklearn.metrics`, and the resulting output of these metrics. The output shows: Accuracy score: 0.9847533632286996, Precision score: 0.9420289855072463, Recall score: 0.935251798561151, and F1 score: 0.9386281588447652. The notebook also includes a paragraph about the advantages of Naive Bayes.

recall (sensitivity) tells us what proportion of messages that actually are spam were classified by us as spam. It is a ratio of true positives (words classified as spam, and which are actually spam) to all the words that were actually spam, in other words it is the ratio of

[True Positives / (True Positives + False Negatives)]

For classification problems that are skewed in their classification distributions like in our case, for example if we had a 100 text messages and only 2 were spam and the rest 98 weren't, accuracy by itself is not a very good metric. We could classify 90 messages as not spam (including the 2 that were spam but we classified them as not spam, hence they would be false negatives) and 10 as spam (all 10 false positives) and still get a reasonably good accuracy score. For such cases, precision and recall come in very handy. These two metrics can be combined to get the F1 score, which is weighted average of the precision and recall scores. This score can range from 0 to 1, with 1 being the best possible F1 score.

We will be using all 4 metrics to make sure our model does well. For all 4 metrics whose values can range from 0 to 1, having a score as close to 1 as possible is a good indicator of how well our model is doing.

```
In [26]: from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
print('Accuracy score: {}'.format(accuracy_score(y_test, predictions)))
print('Precision score: {}'.format(precision_score(y_test, predictions)))
print('Recall score: {}'.format(recall_score(y_test, predictions)))
print('F1 score: {}'.format(f1_score(y_test, predictions)))
```

Accuracy score: 0.9847533632286996
Precision score: 0.9420289855072463
Recall score: 0.935251798561151
F1 score: 0.9386281588447652

One of the major advantages that **Naive Bayes** has over other classification algorithms is its ability to handle an extremely large number of features. In our case, each word is treated as a feature and there are thousands of different words. Also, it performs well even with the presence of irrelevant features and is relatively unaffected by them.

The other major advantage it has is its relative simplicity. Naive Bayes' works well right out of the box and tuning its parameters is rarely ever necessary, except usually in cases where the distribution of the data is known.