Problem Description -

This dataset represents the output of the OCR stage of our data pipeline. We need to train a document classification model. Deploy the model to a public cloud platform (AWS/Google/Azure/Heroku) as a webservice with a simple UI.

Data Profiling/ Processing –

The preliminary step in any data science project should be to develop familiarity with the data at hand. Transitioning right away to model development without knowing the basic characteristics of data can cause issues. So in this step, I computed a series of profiles so as to identify issues, perform quality checks and to get an idea of how the data is distributed.

I started with looking the data –

```
In [4]: #Check the data
        df.describe()
Out[4]:
```

	type	text
count	62204	62159
unique	14	60176
top	BILL	bf064c332aa1 079935e500e5 1a4dd36c6de0 7efa289...
freq	18968	11

Then I looked if there are any null values in the dataset –

```
In [5]: #Look for Missing values
        df.isna().sum()

Out[5]: type      0
        text     45
        dtype: int64
```

There are around 45 null values out of 62204, since there are very less number of null values we can go ahead and drop those rows.

Then I looked if there are any duplicates rows in the data

```
In [8]: #Lets Look if there are any duplicates in the data
        sum(df.duplicated())

Out[8]: 1617
```

There were around 1600 duplicate rows, it certainly won't help our model, I went ahead and dropped these as well.

Let's look at the data again, we are down to 60542 rows –

```
In [10]: df.describe()

Out[10]:
```

	type	text
count	60542	60542
unique	14	60176
top	BILL	84884d80641d
freq	18449	3

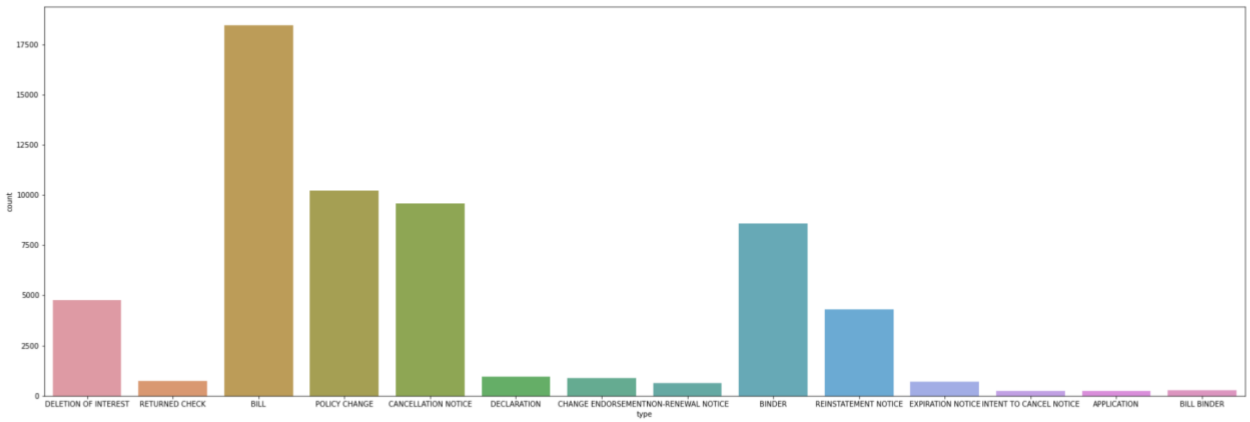
We can see that there are 14 categories for document type, let's look at their counts.

Data Distribution

```
In [11]: # Lets look at the document category and count
         df.type.value_counts()

Out[11]: BILL                        18449
         POLICY CHANGE               10229
         CANCELLATION NOTICE          9571
         BINDER                       8590
         DELETION OF INTEREST         4779
         REINSTATEMENT NOTICE         4295
         DECLARATION                   966
         CHANGE ENDORSEMENT            866
         RETURNED CHECK                730
         EXPIRATION NOTICE             719
         NON-RENEWAL NOTICE            618
         BILL BINDER                   277
         INTENT TO CANCEL NOTICE       227
         APPLICATION                   226
         Name: type, dtype: int64
```

Let’s visualize it –

```
In [12]: plt.figure(figsize=[30,10])
         sns.countplot(x=df.type)
Out[12]: <AxesSubplot:xlabel='type', ylabel='count'>
```



We can see that the data is not at all evenly distributed, that is not good for any classification problem.

Possible solution can be –
1. Down-sample the Majority Class
2. Up-sample the Minority Class
3. Combine certain categories of documents.

Let’s analyze the vocabulary richness of our data. If we just analyze the lexical richness of the text description data, it will still give us much idea about how rich our data is in terms of unique vocabulary words. Since the data is hashed, we can’t do much with it, like stemming, lemmatization or filtering the stop words out.

```
In [13]: #Lets See how many unique words we have in our dataset
         all_words = [w.split() for w in df.text.values]
         total_flat_words = [ewords for words in all_words for ewords in words]

         print('total unique words in the dataset: ',len(set(total_flat_words)))
         print('total words in the dataset: ',len(total_flat_words))

         total unique words in the dataset:  1037934
         total words in the dataset:  20250777
```

We are dealing with 1 million unique words in our dataset, that is certainly a lot, we are restricted a lot to bring this down due to the hashed data.

Let’s look at distribution of number of words in document across various document types
For that I will generate a new feature which is word_count of each document. This will help us filter the data easily and to better visualize the data.

Let’s look at our new feature, grouped by different across different document types.

In [16]:
```
#Check the grouping by the doc type
df.groupby('type').word_count.agg(['count','min','mean','median','max','std','sum'])
```

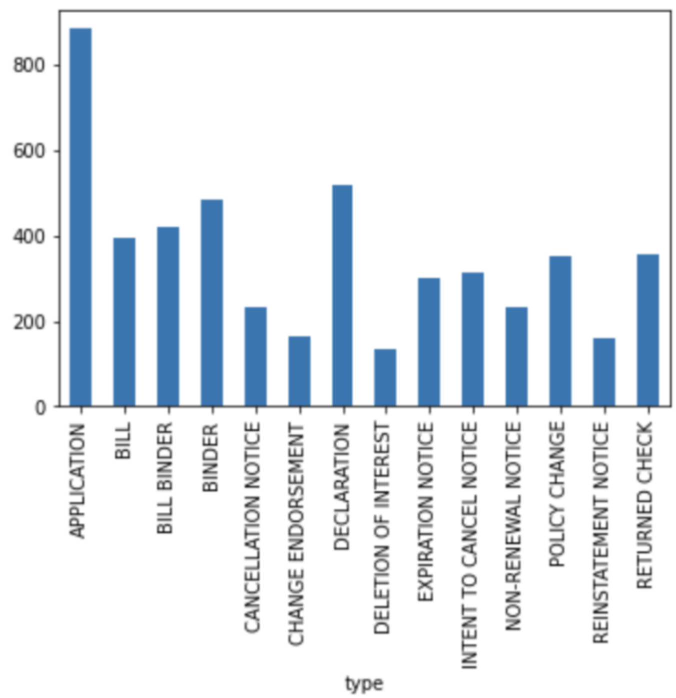
Out[16]:

type	count	min	mean	median	max	std	sum
APPLICATION	226	34	884.026549	766.5	3465	661.126257	199790
BILL	18449	1	395.256545	330.0	7030	249.893216	7292088
BILL BINDER	277	5	418.606498	317.0	3360	371.339565	115954
BINDER	8590	1	483.029453	358.0	7426	439.725545	4149223
CANCELLATION NOTICE	9571	7	231.965312	179.0	4107	171.399429	2220140
CHANGE ENDORSEMENT	866	10	163.112009	111.0	4833	215.160368	141255
DECLARATION	966	4	516.761905	404.5	4734	426.754326	499192
DELETION OF INTEREST	4779	66	135.516008	104.0	1941	95.745271	647631
EXPIRATION NOTICE	719	70	302.303199	203.0	1798	270.202116	217356
INTENT TO CANCEL NOTICE	227	8	312.960352	266.0	2979	243.826253	71042
NON-RENEWAL NOTICE	618	87	234.312298	163.0	1071	159.959125	144805
POLICY CHANGE	10229	1	352.781113	230.0	9076	454.899502	3608598
REINSTATEMENT NOTICE	4295	7	158.863329	134.0	2665	97.001223	682318
RETURNED CHECK	730	42	358.061644	289.5	4460	280.876207	261385

We can see that there are many outliers on both sides, some documents have only one word which I think is not enough to determine the document category, these documents might not be scanned properly from OCR stage and certainly wouldn’t help our model.
The same goes to the one which have lots of words, there document will skew the data.

Let's plot the Mean word count of different document type –

```
In [17]: # Plotting the mean number of words per document type
         df.groupby('type').word_count.mean().plot(kind='bar')

Out[17]: <AxesSubplot:xlabel='type'>
```



Let's look at the number of documents with less that 10 words -

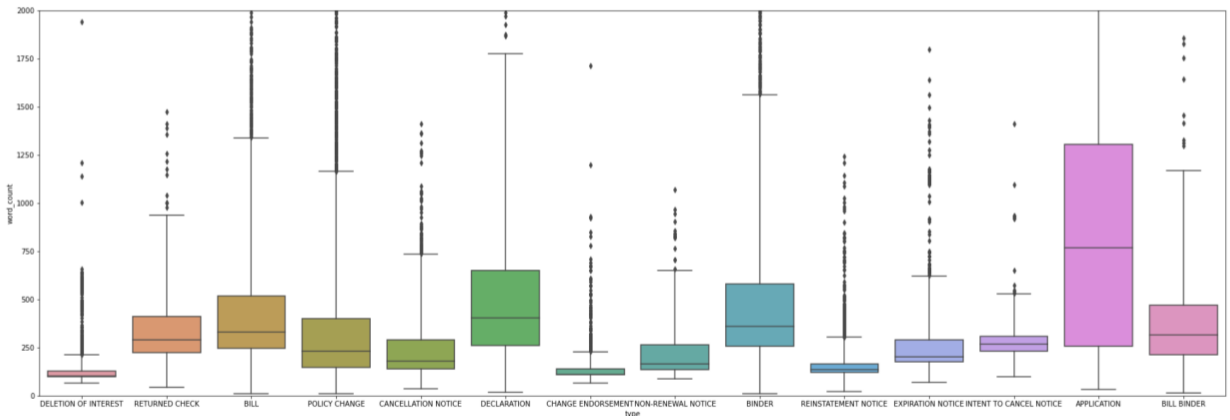
We would need to investigate the documents where we have less than 10 words with some of them having only 1 word, which seems quite sketchy and might be bad data from OCR since both mean and median seems to be very high for the doc type

```
In [18]: df[df['word_count'] <= 10].groupby('type').size().sort_values()

Out[18]: type
         BILL BINDER                1
         CANCELLATION NOTICE        1
         CHANGE ENDORSEMENT         1
         INTENT TO CANCEL NOTICE    1
         REINSTATEMENT NOTICE       1
         DECLARATION                2
         BILL                      14
         BINDER                    28
         POLICY CHANGE             57
         dtype: int64
```

It looks like there are around 100 documents like this, let's go ahead and drop these as well.

Let’s visualize the distribution of number of words for each document type, we see that there are lot of outliers. I am using whis=3 instead of default 1.5

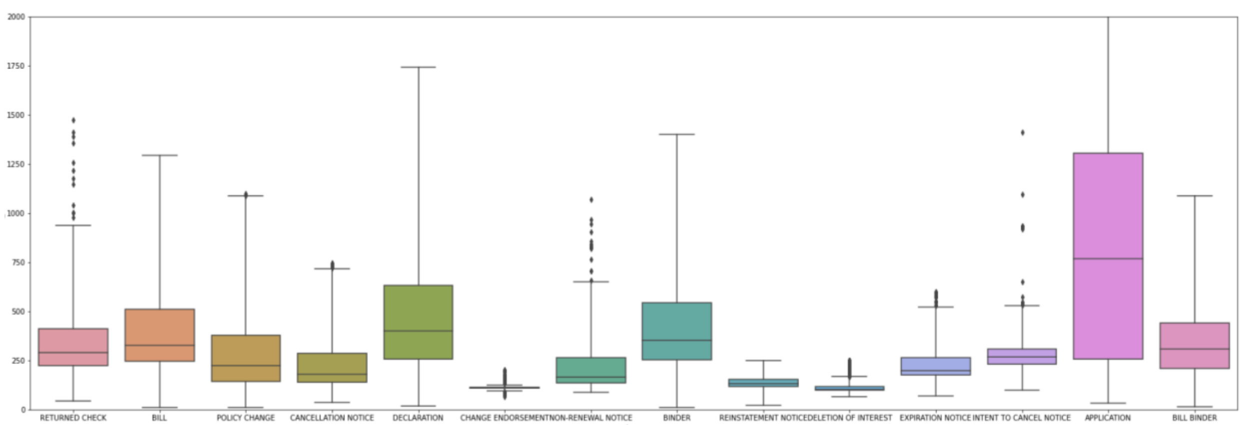
```
In [20]: #Lets look at the word count distribution
         plt.figure(figsize=[30,10])
         plt.ylim(0, 2000)
         sns.boxplot(x='type',y='word_count',data=df,whis=3)

Out[20]: <AxesSubplot:xlabel='type', ylabel='word_count'>
```



I went ahead and dropped these outliers.

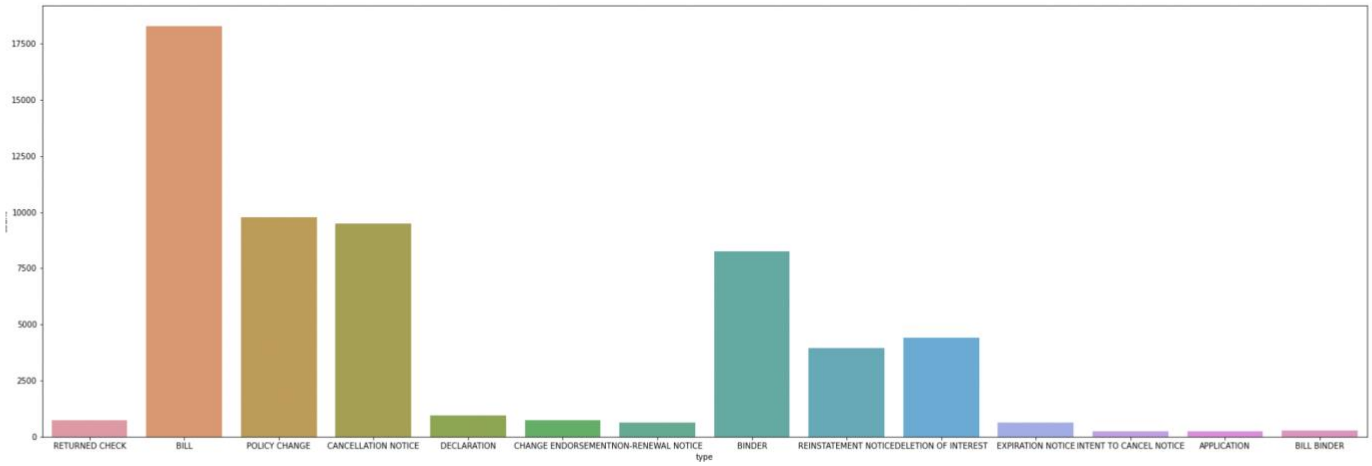
Recommendation –
We could also try to get first n words out of the document instead of directly dropping it.

Let’s look at the data again after dropping –



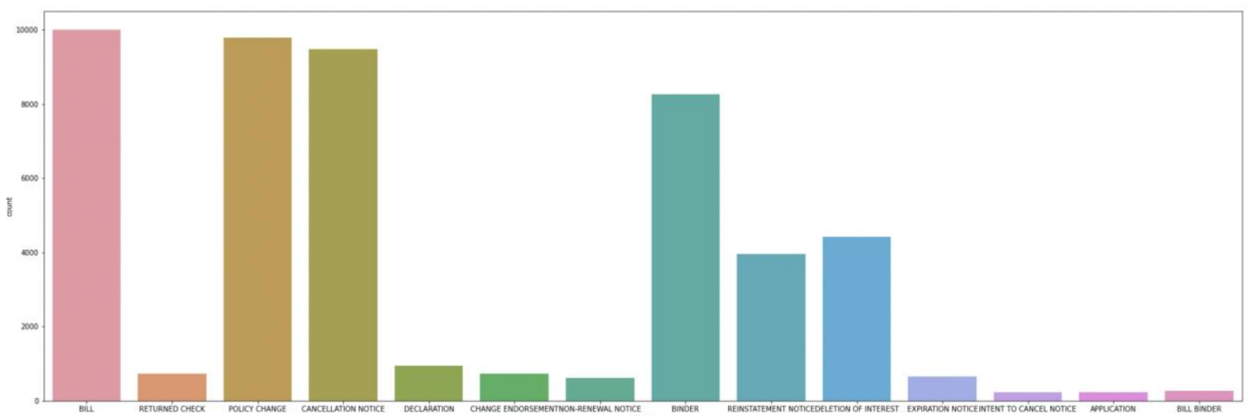
Seems pretty reasonable now, I didn’t drop some categories which had very less number of rows. Dropping the data will help us with the skewness as well as number of dimensions in data.

Let’s look at the distribution again



We still have to deal with the imbalance in the data –
I went with down sampling the Document Type – BILL as it’s almost 2 times as big as the next closest class.

Let’s look at the data again after down sampling –



It improved a little but still we have lot of imbalance in the data. I didn’t want to upsample the minority class as there are so many classes and we will end up with lot of synthetic data.

Recommendation –

1. We can look for combining certain document types like Intent to Cancel and Cancellation Notice, or bill and bill binder etc.
2. We can come back and try upsampling certain categories which we are not able to classify properly.
3. We can further downsample more categories and see if it helps the model.

Modeling -

Let's look at the length of dataset after all the previous steps

```
In [6]: #Total rows of data
        len(df)

Out[6]: 50262
```

Now I went ahead and split the data into train and test set, I kept 25% of the data for test set

```
In [8]: #Lets split the data into train and test
        # Split the dataset into train and test sets
        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=100)
```

I used **sklearn Pipeline** for modeling process –

To convert all the text document to vector I used Scikit-learn's **CountVectorizer**
This functionality makes it a highly flexible feature representation module for text.

I followed it with **TF-IDF transformer**, it will enable us to give us a way to associate each word in a document with a number that represents how relevant each word is in that document. Then, documents with similar, relevant words will have similar vectors, which is what we are looking for in a machine learning algorithm.

```
Pipeline([('vect', CountVectorizer(min_df=5, ngram_range=(1,2))),
          ('tfidf', TfidfTransformer()),
```

After the following two step I tried different models, I experimented with several Machine Learning algorithms: Logistic Regression, Linear SVM, Multinomial Naive Bayes, Random Forest, KNeighbour Classifier, Stochastic Gradient Descent and MLP.

Applying SGD Classifier

In [140]:
```
# Create SGDClassifier model pipeline
model_nm = 'SGDClassifier'
model = Pipeline([('vect', CountVectorizer(min_df=5, ngram_range=(1,2))),
                  ('tfidf', TfidfTransformer()),
                  ('model',SGDClassifier(max_iter=1000,loss='modified_huber',class_weight='balanced')), ],verbose=Tr
```

In [141]:
```
#Lets fit the model
model.fit(X_train,y_train)
```

```
[Pipeline] .............. (step 1 of 3) Processing vect, total=  21.9s
[Pipeline] ............. (step 2 of 3) Processing tfidf, total=   0.8s
[Pipeline] ............. (step 3 of 3) Processing model, total=   3.1s
```

Out[141]:
```
Pipeline(steps=[('vect', CountVectorizer(min_df=5, ngram_range=(1, 2))),
                ('tfidf', TfidfTransformer()),
                ('model',
                 SGDClassifier(class_weight='balanced',
                               loss='modified_huber'))],
         verbose=True)
```

SGD Classifier and LinearSVC performed very well among all the classifiers.

After trying different models, I also tried to see if I could extract the k-best features out of the all the features, which I tried with sklearn's selectkbest using chi2, but it didn't improve the performance.

Applying Chi-Square Feature Selection

125]:
```
#SGDClassifier(alpha=0.0001,penalty='elasticnet',n_iter=50)
#k=73000
# Create SGDClassifier model pipeline
model_nm = 'SGDClassifierKbest'
model = Pipeline([('vect', CountVectorizer(min_df=5, ngram_range=(1,2))),
                  ('tfidf', TfidfTransformer()),
                  ('selectkbest',SelectKBest(chi2, k=30000)),
                  ('model',SGDClassifier(max_iter=1000,loss='modified_huber',class_weight='balanced')), ],verbose=Tr
```

126]:
```
#Lets fit the model
model.fit(X_train,y_train)
```

```
[Pipeline] .............. (step 1 of 4) Processing vect, total=  19.7s
[Pipeline] ............. (step 2 of 4) Processing tfidf, total=   0.8s
[Pipeline] ....... (step 3 of 4) Processing selectkbest, total=   0.5s
[Pipeline] ............. (step 4 of 4) Processing model, total=   2.0s
```

126]:
```
Pipeline(steps=[('vect', CountVectorizer(min_df=5, ngram_range=(1, 2))),
                ('tfidf', TfidfTransformer()),
                ('selectkbest',
                 SelectKBest(k=30000,
                             score_func=<function chi2 at 0x7fb403608c80>)),
                ('model',
                 SGDClassifier(class_weight='balanced',
                               loss='modified_huber'))],
         verbose=True)
```

Here is how the model compared with each other on accuracy.

Model	Embeddings	Accuracy
Naive Bayes	CV+TF-IDF	0.73
Random Forest	CV+TF-IDF	0.85
SGD	CV+TF-IDF	0.88
Logistic Regression	CV+TF-IDF	0.86
LinearSVM	CV+TF-IDF	0.88
KNeighbour	CV+TF-IDF	0.82
NN	Tokenizer	0.84

SGD and LinearSVM performance were really close, I went ahead with SGD for the final model.

I did grid search for hyperparameter tuning with modifying loss, alpha, and penalty. It worked best with the default configrations.

Model Metrics –

This is how the metric looked for the test set.
We were able to achieve the accuracy close to 88%, with high F-Score

Classification Report –

accuracy 0.8723539710329461

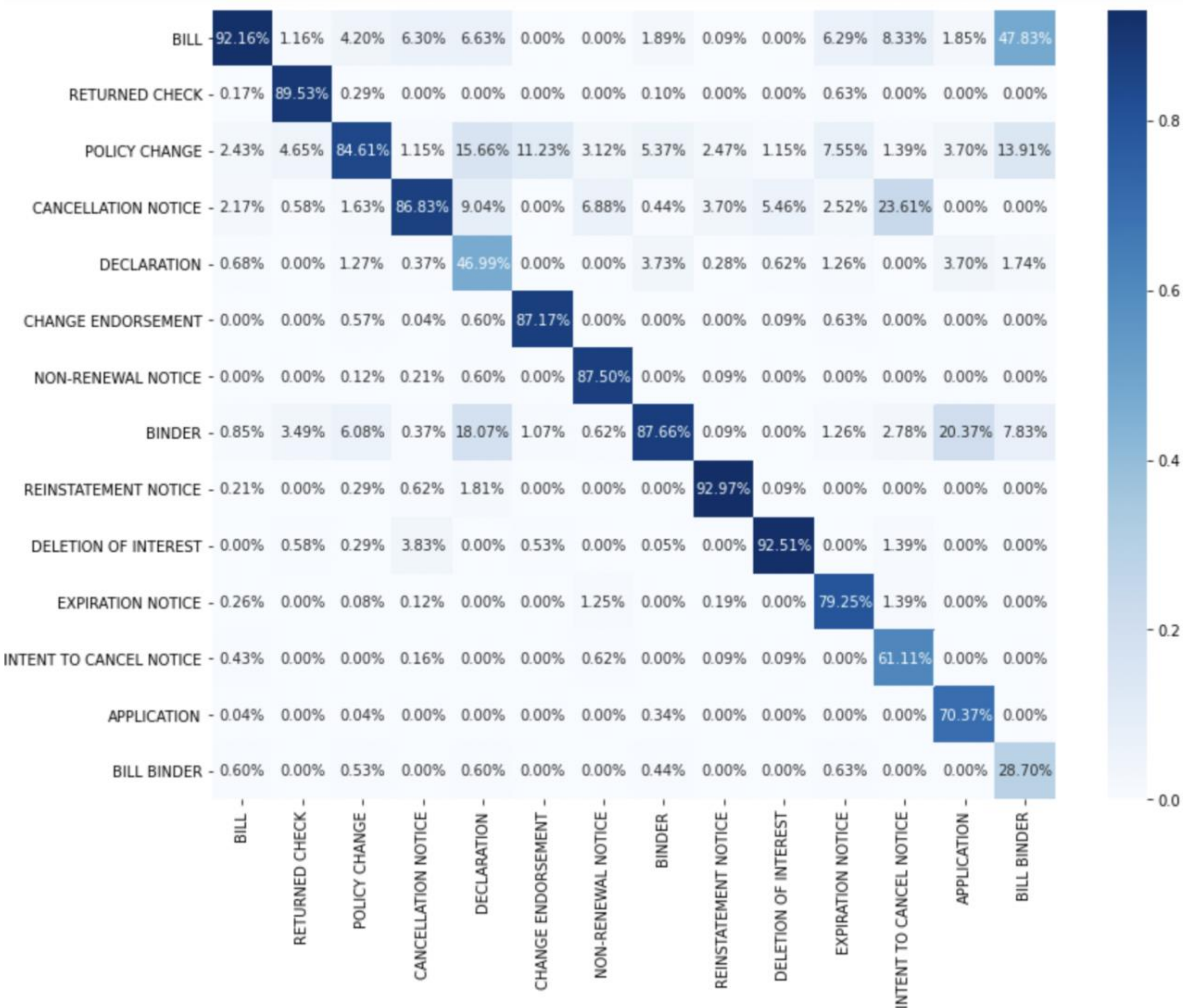
	precision	recall	f1-score	support
APPLICATION	0.70	0.81	0.75	47
BILL	0.92	0.85	0.88	2544
BILL BINDER	0.29	0.46	0.35	71
BINDER	0.88	0.88	0.88	2054
CANCELLATION NOTICE	0.87	0.89	0.88	2358
CHANGE ENDORSEMENT	0.87	0.90	0.89	181
DECLARATION	0.47	0.34	0.40	227
DELETION OF INTEREST	0.93	0.91	0.92	1154
EXPIRATION NOTICE	0.79	0.89	0.84	142
INTENT TO CANCEL NOTICE	0.61	0.72	0.66	61
NON-RENEWAL NOTICE	0.88	0.93	0.90	150
POLICY CHANGE	0.85	0.86	0.86	2399
REINSTATEMENT NOTICE	0.93	0.97	0.95	1010
RETURNED CHECK	0.90	0.92	0.91	168
accuracy			0.87	12566
macro avg	0.78	0.81	0.79	12566
weighted avg	0.87	0.87	0.87	12566

Though out model is working really well for lot of categories.

Our model is performing poorly in classifying Bill Binder and not performing that well with Intent to cancel.

Let’s check the Confusion Matrix to identify which categories our model is misclassifying it to.

Confusion Matrix –



We can make few observations here –

- Lot of our Intent to Cancel Notice are classified as Cancellation Notice which makes as sense as those documents are bound to have lot of similarity.
- Same we can see for Bill binder and observe that our model is misclassifying it a lot with either Bill or Binder.

Recommendation –

- It would certainly help if we could get more data for these categories.
- We could also see the raw ocr data to see if we can identify the relevant differences among these documents.
- We can try upsampling these classes to see if our model performance increases.

I also tried testing the model against the full dataset.
Here are the results –

We were able to achieve overall approximate 90% accuracy

accuracy 0.8962821152206438

	precision	recall	f1-score	support
APPLICATION	0.57	0.96	0.71	229
BILL	0.97	0.86	0.91	18959
BILL BINDER	0.37	0.81	0.51	289
BINDER	0.92	0.92	0.92	8952
CANCELLATION NOTICE	0.84	0.93	0.88	9729
CHANGE ENDORSEMENT	0.92	0.84	0.88	889
DECLARATION	0.61	0.73	0.66	967
DELETION OF INTEREST	0.94	0.88	0.91	4826
EXPIRATION NOTICE	0.81	0.88	0.84	734
INTENT TO CANCEL NOTICE	0.60	0.92	0.73	229
NON-RENEWAL NOTICE	0.92	0.98	0.95	624
POLICY CHANGE	0.87	0.91	0.89	10616
REINSTATEMENT NOTICE	0.95	0.93	0.94	4367
RETURNED CHECK	0.91	0.98	0.94	749
accuracy			0.90	62159
macro avg	0.80	0.90	0.83	62159
weighted avg	0.91	0.90	0.90	62159

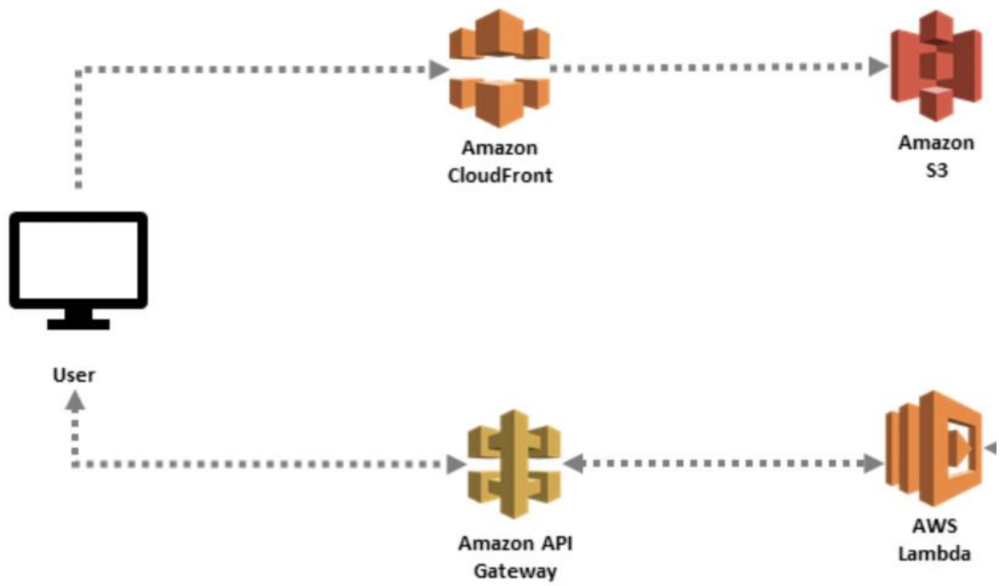
Model Recommendations:
Here are some recommendations that can be explored to further improve the analysis:

- Using Functional programming, we can form a general function which can be used to pass classifiers and to derive the results.
- The current version of the code can be made much better by making it more modular and defining classes.
- It certainly would help I could see the raw data instead of hashed data to further refine the model.
- I wanted to try passing limited subset of words per document instead of full document for training.
- We can use advanced methodology like word2vec to find out words that occur together and can use them in the feature's extraction process as well.
- I wanted to try a CNN model, I believe that would improve our model scores.
- I was restricted by the processing power, certain algorithms like XGBOOST, RandomForest took a lot of time to run, didn't give me chance to tweak hyperparameters to improve their performance.

Model Deployment –

I followed the serverless deploy model of AWS.
I used lambda function + API gateway for the RestAPI, docker image in ECR for model libraries. and Amazon S3 for model storage as well as hosting the static UI.

I used AWS SAM CLI to create the full backend-stack using template.yaml to define the required resources needed.



Rest API -
https://et9cl4lp4l.execute-api.us-east-1.amazonaws.com/Prod/predict/

Swagger API Documentation –
https://app.swaggerhub.com/apis-docs/ladip21235/bk-doc-class/0.2

Sample Curl Request -

- GET -
curl --request GET 'https://et9cl4lp4l.execute-api.us-east-1.amazonaws.com/Prod/predict?words=putDocumentTextHere'

- POST -
curl --location --request POST 'https://et9cl4lp4l.execute-api.us-east-1.amazonaws.com/Prod/predict' --header 'Content-Type: application/json' \
--data-raw '{"words":"putDocumentTextHere"}'

Prediction UI –
https://bkfsmodel.s3.amazonaws.com/index.html