Spoiler Classification in Book Reviews

Team Members: Rohan Saraogi (rsaraogi@seas.upenn.edu), Akash Sundar (aka2000@seas.upenn.edu), Shikha Reddy (shikham@seas.upenn.edu)

home pod: lovable-lemur

Abstract

Book spoilers on review websites can detract from a user's experience consuming media. In this project we seek to build a system that automatically detects spoiler sentences. We frame the problem as a sentence-level binary classification task and experiment with different machine learning models (logistic regression, perceptrons, random forests) and transfer learning with large-language models. We address a class imbalance problem in our dataset by using Area Under Curve (AUC) as our primary evaluation metric, and experimenting with different class weightings, sampling strategies, and loss functions. Our best performing model proves to be logistic regression, which achieves a test AUC of 0.785.

1 Motivation

Book spoilers on review websites can be a cause of concern for users who want to fully experience the fun, uncertainty, and suspense associated with reading books. Certain review websites allow review authors to annotate whether their reviews (or sentences in their reviews) contain spoilers. However, we observe that this feature isn't always used by review authors, and therefore review authors cannot be fully relied on to annotate spoiler sentences. A potential solution would be crowd-sourcing: allow the readers of reviews to annotate spoilers. However, this just shifts the task over to the readers and raises potential scalability concerns as it can be difficult to ensure that spoiler annotations are done in a timely manner.

We seek to address this problem by building a system that automatically detects spoiler sentences in reviews. At face value, this seems to be a fairly challenging task given the breadth and richness of natural language. How could a system use the raw text of a sentence to detect whether it is a spoiler? However, with the proliferation of machine learning methods in the domain of natural language processing (NLP), time has shown that these methods can be used to solve a wide range of language related tasks. We intend to train machine learning models specifically for the task of spoiler classification, and to also experiment with models trained on other language related tasks via methods like transfer learning. A successful design of such a system would help automatically detect spoiler sentences and also address the aforementioned scalability concerns: once deployed, the system could automatically annotate spoiler sentences when authors post their reviews.

2 Related Work

Although sentence classification is a common task in NLP, there is little work done specifically for spoiler classification. Early work for this task suffers from potential generalizability concerns owing to reliance on hand-crafted domain-specific features extracted from relatively well-curated datasets. In [3], the authors manually annotate IMDB (Internet Movie Database) reviews, and approach this task using Latent Dirichlet Allocation (LDA) based topic models using linguistic dependency features instead of simple bag-of-words (BOW) representations. In [1], the authors create a spoiler dataset from the TV Tropes website, and emphasize the use of meta-data based features eg. genre for spoiler classification. The authors in [4], with the help of human annotators, create a spoiler dataset from Amazon.com reviews. They highlight the importance of personal names and peculiar words for spoiler classification and observe that a Naive Bayes based model performs best on their dataset.

One of the first applications of deep learning for this task was in [2]. The authors use the dataset created by the authors in [1], and develop a GRU/CNN based deep-learning model for this task, with an emphasis

on modeling external genre information. In [8], the authors create the dataset we have used for our project, build their own custom RNN/attention based deep learning model for spoiler classification, and emphasize the role of biases pertaining to the review author and the book being reviewed for this task. These features enable them to achieve an impressive 0.919 AUC score on the dataset.

To the best of our knowledge, there hasn't been any noteworthy work done with an emphasis on the application of modern large-language models to this task, whether in the form of contextual embeddings that can be used by downstream machine learning models, or fine-tuning/transfer learning.

3 Data Set

3.1 Summary Statistics, Visualizations & Preprocessing

Our dataset (`https://drive.google.com/uc?id=196W2kDoZXRPjzbTjM6uvTidn6aTpsFnS`) was created by the authors of [8] by scraping `www.goodreads.com`.

Name	Type
review_id	string
user_id	string
book_id	string
timestamp	date
rating	integer
review_sentences	list
has_spoiler	boolean

(a)

Statistic	Original	Reduced
No. of reviews	1,378,033	172,004
No. of users	18,892	15,605
No. of books	25,475	23,450
No. of sentences	17,672,655	199,000
% of spoiler sentences	3.22%	3.16%

(b)

Table 1: (a) Dataset column names and types (b) Summary statistics for original and reduced datasets

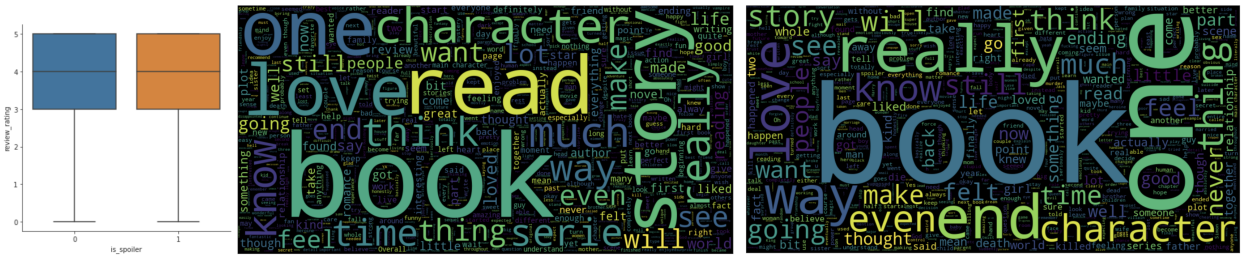
Each row in the original dataset corresponds to a review. We note that :-

- *review_id*, *user_id*, and *book_id* are anonymized IDs for the review, the review author, and the review book respectively; *timestamp* is the review date; *rating* is the review author's rating for the book (0-5 scale); *has_spoiler* is a boolean indicating whether the review contains any spoiler sentences.
- *review_sentences* is a list of tuples. Each tuple is a pair (*is_spoiler*, *sentence*), where *sentence* is a sentence in the review and *is_spoiler* is a boolean indicating whether the sentence is a spoiler or not. The spoilers have been self-annotated by the review authors. In this project we primarily focus on the contents of this column.

Table 1(b) provides summary statistics for the data. We note that :-

- There is a class imbalance problem as only 3.22% of the sentences are spoilers. We discuss different approaches we have used to deal with this in subsequent sections.
- Although not shown here, ∼8% of the spoiler sentences are spoilers in one review and not spoilers in another. Some potential reasons for this are (1) whether a sentence is a spoiler can partly depend on the context provided by the other sentences in the review (2) as the sentences are annotated by the review authors, the annotations can be subjective and depend on the author's judgement. Owing to this ambiguity and concerns about the impact of these sentences on model performance, we have removed these sentences from the dataset. We recognize this as a limitation of our dataset/approach and provide further thoughts on the same in the Conclusion and Discussion section.

- The original dataset contains 17,672,655 sentences. We have reduced the size of the dataset to ∼200,000 sentences owing to computational resource constraints. The reduced dataset has been generated via random sampling to preserve the class distributions.

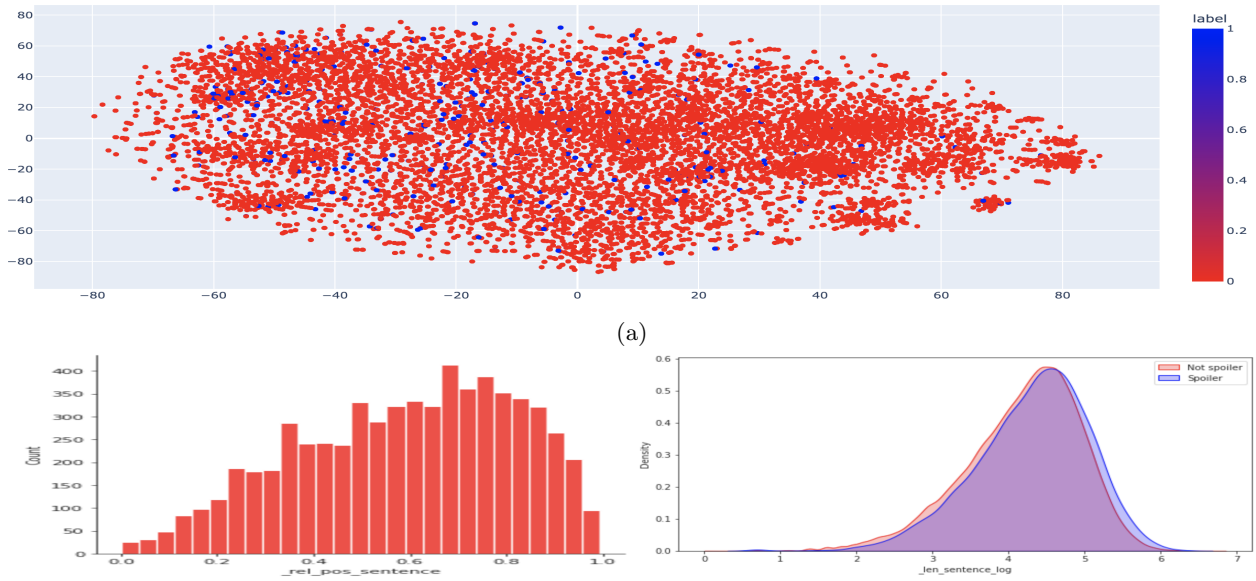


(a) (b) (c)

Figure 1: (a) Box plot of review ratings for not spoiler sentences (blue) and spoiler sentences (orange) (b) Wordcloud of not spoiler sentences (c) Wordcloud of spoiler sentences

The boxplot in Figure 1(a) suggests that there is no discernible difference between not spoiler and spoiler sentences on the basis of review ratings. The wordclouds in Figure 1(b) and 1(c) look quite similar, although the spoiler wordcloud does highlight some words that could be associated with spoilers eg. 'never', 'end'.

3.2 Feature Engineering



(a)

(b) (c)

Figure 2: (a) t-SNE plot of Sentence-BERT embeddings of a random sample of 10000 not spoiler sentences (red) and spoiler sentences (blue) (b) Histogram of relative positions of spoiler sentences in reviews (c) Density plot of log of lengths in characters of not spoiler sentences (red) and spoiler sentences (blue)

- Given that machine learning models cannot directly work with text, it is necessary to convert it to a numeric format. Converting text to dense contextual embedding vectors via pretrained large language models is a very common trend these days in NLP. The idea is that large language models pretrained on vast amounts of text can (without any manual preprocessing) encode the text into

meaningful representations that can be used by downstream machine learning models. We have used the *sentence-transformers* Python package (`https://www.sbert.net/`) to convert each sentence to a 384 dimensional dense embedding. The package is based on [6], and we have used the default model *all-MiniLM-L6-v2*.

- *rel_pos_sentence* is the relative position of a sentence in a review i.e. the index of the sentence in the review divided by the no. of sentences in the review. The skew in Figure 2(b) suggests that spoiler sentences tend to appear later in the review. Hence this feature could be informative for spoiler classification and we have decided to include it.
- *__len_sentence_log* is the log of the length of a sentence in characters. The role of the logarithm is to make the feature magnitude more manageable. Figure 2(c) suggests that spoiler sentences can be slightly longer than not spoiler sentences. Given how marginal the difference is, we are skeptical about the informativeness of this feature. However, we have included it anyways with the understanding that it can be regularized out in the training process if it proves to be redundant.

4 Problem Formulation

We frame the problem as a sentence-level binary classification task: given the raw text of a sentence s in a review, we aim to predict if it contains spoilers ($y_s = 1$) or not ($y_s = 0$).

We have split the dataset into 99,000 training samples, 50,000 validation samples, and 50,000 test samples. Owing to the class imbalance problem, we have used reasonably large validation and test sets to ensure that they contain enough spoiler sentences.

For our non-deep learning models (other than the baseline), we use sentence-BERT embeddings, *rel_pos_sentence*, and *__len_sentence_log* as features. For our transfer learning models we use sentence raw texts, *rel_pos_sentence*, and *__len_sentence_log* as features (details of model architecture are given in the Methods section). To reiterate what has already been mentioned in the Data Set section, the purpose of the sentence-BERT embeddings is to convert raw text to dense embeddings that can be used by downstream machine learning models. Also, Figure 2(b) suggests that *rel_pos_sentence* could prove to be a useful feature, and we include *__len_sentence_log* with the expectation that it can be regularized out if it proves to be redundant.

As this is a binary classification task, our primary loss function is binary cross-entropy (BCE) loss i.e. $\mathcal{L} = \sum(y_s \log(p_s) + (1 - y_s) \log(1 - p_s))$, where p_s is the prediction probability of a sentence s being a spoiler and y_s is the true label. Owing to the class imbalance problem, for the transfer learning models we also experiment with two modifications of BCE loss :-

- *Weighted BCE loss*: To increase the importance of spoiler sentences when computing the loss, we introduce a weight η to the spoiler term in the loss i.e. replace $y_s \log(p_s) + (1 - y_s) \log(1 - p_s)$ with $y_s \eta \log(p_s) + (1 - y_s) \log(1 - p_s)$ (`https://pytorch.org/docs/stable/generated/torch.nn.BCEWithLogitsLoss.html`). To balance the impact of the classes, we set $\eta = 32 \approx 97/3$ i.e. to approximately the ratio of the percentage of not spoiler sentences and the percentage of spoiler sentences.
- *Focal loss*: The authors in [5] introduce a factor $\alpha(1 - p_t)^\gamma$ to BCE loss to deal with a class imbalance problem in their dataset. The purpose of the factor is to put more focus on the extent of misclassification (regardless of class) when computing the loss, and α and γ are hyperparameters. We implement a version of focal loss ourselves and use the default values $\alpha = 0.25$ and $\gamma = 2$ used by the authors.

5 Methods

5.1 Baseline

For our baseline we use a TF-IDF vectorizer (`https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html`) to generate sparse embeddings for sentences,

and train a logistic regression classifier with an L2 penalty (`https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html`) on it. TF-IDF vectorization is a classic technique to generate text embeddings and has been a staple in pre-deep learning NLP. And logistic regression is the prototypical machine learning method for binary classification. So it is fitting to use this combination as a baseline.

5.2 Logistic Regression

We use our engineered features (sentence-BERT embeddings, *rel_pos_sentence*, *__len_sentence_log*), and train a logistic regression classifier on it with hyperparameter tuning. This allows us to compare TF-IDF vectorization with our feature engineering, and the hyperparameter tuning gives us a better sense of the capabilities of logistic regression for our problem.

5.3 Multi-Layer Perceptron Classifier (MPC)

We use our engineered features and train an MPC (`https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html`) on it with hyperparameter tuning. This serves as an initial foray into the application of deep learning methods to our problem.

5.4 Random Forest Classifier (RFC)

We use our engineered features and train an RFC (`https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html`) on it with hyperparameter tuning. As an ensemble method, RFCs can learn a much more complex decision surface compared to logistic regression, and would therefore be an interesting candidate to experiment with.

5.5 Transfer Learning

We experiment with transfer learning with DistilBERT. DistilBERT is a transformer model proposed by the authors in [7], and we obtain a pretrained version of the model from Hugging Face using the *transformers* package. As indicated in Figure 3, we pass the raw text of a sentence through DistilBERT and extract a 768 dimensional dense embedding corresponding to the [CLS] token (the embedding associated with this token is specifically intended to be used for downstream classification tasks). We combine the embedding with *rel_pos_sentence* and *__len_sentence_log*, and pass it through a classification head composed of a series of dropout, dense, and ReLU layers. The model output is a single logit which can be passed through a sigmoid function to convert it to a probability of the input sentence being a spoiler.

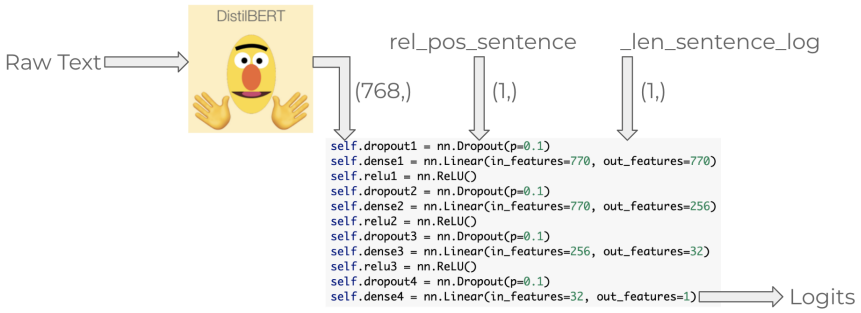


Figure 3: Model architecture

6 Experiments and Results

6.1 Class Imbalance

Our dataset has a class imbalance problem, with only ∼3% of the sentences being spoilers. It is necessary to address this problem as otherwise we would expect the not spoiler sentences to dominate the training process. As such, for our transfer learning models we experiment with BCE loss, weighted BCE loss, and focal loss. Also, for the logistic regression (not baseline), MPC, and RFC models, we experiment with the following three methods (we only use one at a time and not a combination of them):-

- *Underweighting the majority class*: The sklearn implementations of logistic regression and RFCs have a *class_weight* parameter to vary the importance of different classes during the training process. We set the class weight for the spoiler class to 1 and vary the weight for the not spoiler class in the range [0.01, 0.05, 0.1, 0.3, 0.5, 1].
- *Undersampling the majority class*: We undersample the majority class using a random undersampler from the imblearn package (`https://imbalanced-learn.org/stable/references/generated/imblearn.under_sampling.RandomUnderSampler.html`). This is done by varying the *sampling_strategy* parameter (the ratio of the no. of spoiler sentences to the no. of not spoiler sentences) in the range [0.05, 0.1, 0.3, 0.5, 1].
- *Oversampling the minority class*: We oversample the minority class using a SMOTE oversampler from the imblearn package (`https://imbalanced-learn.org/stable/references/generated/imblearn.over_sampling.SMOTE.html`). This is done by varying the *sampling_strategy* parameter (the ratio of the no. of spoiler sentences to the no. of not spoiler sentences) in the range [0.05, 0.1, 0.3, 0.5, 1].

6.2 Evaluation Metrics

Owing to the class imbalance problem, accuracy may not be the best evaluation metric as even a majority classifier can achieve ∼97% accuracy by classifying all sentences as not spoilers. We instead use the Receiver Operating Characteristic (ROC) curve and particularly the Area Under Curve (AUC) score as our primary evaluation metric as it better accounts for the class imbalance. The ROC curve also highlights the model performance for different probability thresholds, which is more holistic compared to just using a 0.5 threshold. As secondary metrics, we also report the accuracy, precision, recall, and F1-score.

6.3 Hyperparameter Tuning

We experiment with the following hyperparameter grids :-

- *Logistic Regression (not baseline)*: {`"penalty": ["elasticnet"], "l1_ratio": [0, 0.5, 1]`}
- *MPC*: {`"hidden_layer_sizes": [(128, 64, 32, 1)], "activation": ["tanh", "relu", "logistic"], "alpha": [0.0001, 0.05]`}
- *RFC*: {`"n_estimators": [100, 150], "max_depth": [5, 7], "min_samples_leaf": [1, 5]`}

6.4 Transfer Learning

We experiment with the following :-

- We freeze the DistilBERT weights and train the classification head for 5 epochs. The models corresponding to BCE loss, weighted BCE loss, and focal loss are called *BCE(5)*, *WBCE(5)*, and *Focal(5)* respectively below.

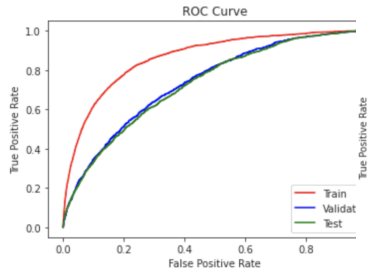
- Focal loss has the best AUC performance on the validation set. Hence we freeze the DistilBERT weights and train the classification head for 10 epochs with focal loss. We then unfreeze the DistilBERT weights and train the entire model for an additional 5 epochs. The corresponding model is called *Focal(15)*.

6.5 Results

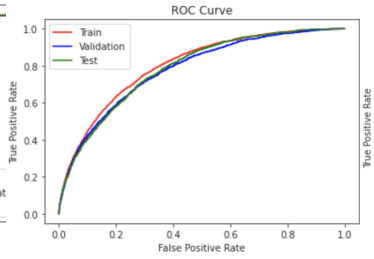
The following table and figures summarize the results.

Model	Accuracy			Precision			Recall			F1			AUC		
	Train	Val	Test	Train	Val	Test	Train	Val	Test	Train	Val	Test	Train	Val	Test
Baseline	0.967	0.970	0.969	nan	1.00	0.50	0.00	0.00	0.00	nan	0.00	0.00	0.865	0.738	0.729
Logistic	0.917	0.917	0.918	0.16	0.14	0.14	0.35	0.34	0.32	0.21	0.20	0.19	0.801	0.779	0.785
MPC	0.904	0.902	0.901	0.15	0.12	0.11	0.40	0.35	0.32	0.22	0.18	0.16	0.802	0.757	0.762
RFC	0.941	0.933	0.932	0.27	0.13	0.13	0.47	0.22	0.22	0.35	0.16	0.16	0.893	0.767	0.766
BCE(5)	0.967	0.970	0.969	nan	nan	nan	0.00	0.00	0.00	nan	nan	nan	0.756	0.759	0.755
WBCE(5)	0.826	0.828	0.826	0.09	0.09	0.09	0.50	0.51	0.50	0.16	0.15	0.15	0.759	0.762	0.758
Focal(5)	0.954	0.957	0.957	0.20	0.21	0.19	0.14	0.15	0.13	0.17	0.17	0.16	0.769	0.769	0.769
Focal(15)	0.945	0.948	0.947	0.19	0.19	0.18	0.20	0.23	0.20	0.19	0.21	0.19	0.783	0.777	0.780

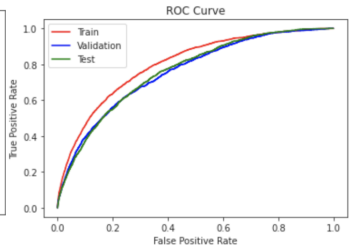
Figure 4: Performance metrics



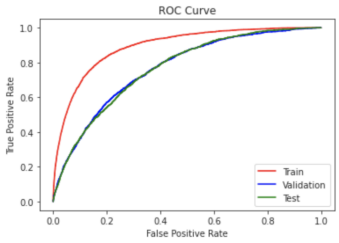
(a) Baseline ROC curves



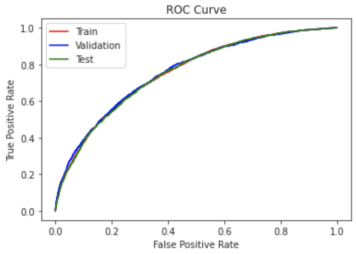
(b) Logistic ROC curves



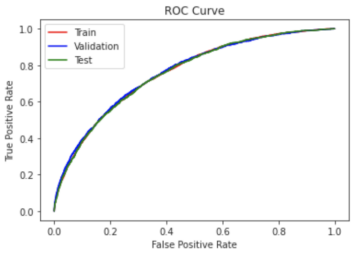
(c) MPC ROC curves



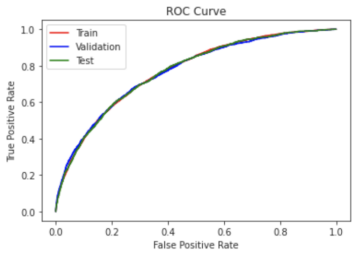
(d) RFC ROC curves



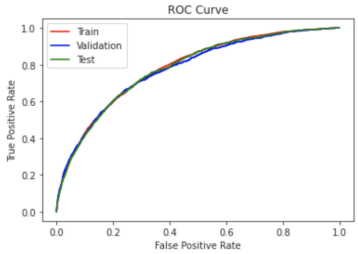
(e) BCE(5) ROC curves



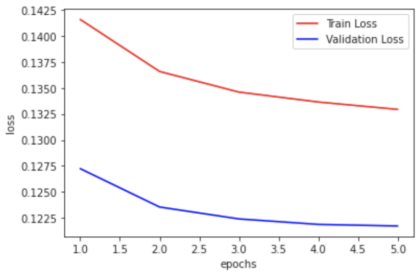
(f) WBCE(5) ROC curves



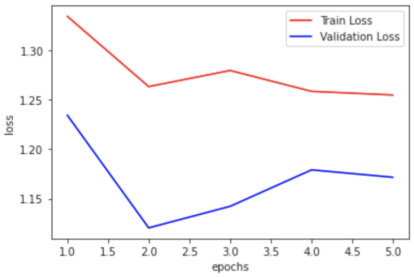
(g) Focal(5) ROC curves



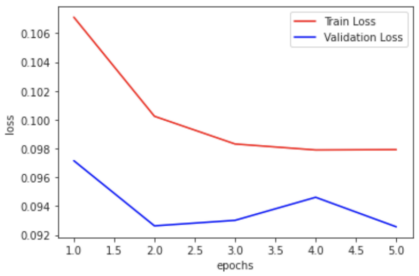
(h) Focal(15) ROC curves



(i) BCE(5) Train/Val loss



(j) WBCE(5) Train/Val loss



(k) Focal(5) Train/Val loss

Figure 5: ROC and train/val curves

We note that :-

- The summary and curves for logistic regression, MLP, and RFC are for the model parameters that yielded the highest AUC on the validation set. These were {`"penalty": "elasticnet", "l1_ratio": 0.5, "class_weight": {0: 0.1, 1: 1}`} for logistic regression, {`"hidden_layer_sizes": (128, 64, 32, 1), "activation": "tanh", "alpha": 0.05, "sampling_strategy": 0.3`} for MLP, and {`"n_estimators": 150, "max_depth": 7, "min_samples_leaf": 1, "class_weight": {0: 0.05, 1: 1}`} for RFC. Underweighting the majority class did the best for logistic regression and RFC, and oversampling the minority class did the best for MPC.
- Despite trying a fairly large no. of iterations, the MPC model couldn't fully converge. Our reasoning for this is that, given that the MPC model is fairly simplistic, it may not have the capacity to fully learn the complex decision surface for our problem.
- As expected, the training and validation loss curves for transfer learning have a generally downward trend. Interestingly, the train losses are higher than the validation losses. Two potential reasons that come to mind for this are (1) perhaps the validation set just happens to be 'nice' for the models to work with (2) the models have a large no. of regularization layers eg. dropout that are activated during training and deactivated during validation. Perhaps these layers are skewing the predictions.
- Owing to Colab repeatedly timing us out due to GPU overuse, we have unfortunately not been able to include the train/val loss curves for *Focal(15)*. However, the performance metrics and ROC curves for this model have been included and were recovered using model checkpoint files saved during the training.

7 Conclusion and Discussion

7.1 Key Findings

We make the following observations based on the table in Figure 4 :-

- The baseline model predicts (almost) all sentences to be not spoilers, and hence achieves an accuracy of ∼97% and an F1-score of 0.00 on the test set. This highlights the impact of the class imbalance problem.
- Although not great, the logistic regression, MPC, and RFC models achieve better F1-scores on the test set without substantial tradeoffs in test accuracy (all $> 90\%$). This highlights the value of our attempts to overcome the class imbalance problem via methods like underweighting the majority class, undersampling the majority class, and oversampling the minority class.
- *BCE(5)* behaves similarly to the baseline, predicting almost all sentences to be not spoilers. *WBCE(5)* goes too far the other direction, taking a significant hit in test accuracy by predicting too many sentences to be spoilers. Despite this, the test recall of 0.5 indicates that it can only correctly classify 50% of the spoiler sentences in the test set. *Focal(5)* achieves the highest test F1 and AUC scores out of the three loss functions, and therefore seems to have the best balance. However, training the transfer learning model for an additional 10 epochs with focal loss only yields a 1% improvement in AUC.
- Our best performing model is logistic regression, which achieves a ∼19% improvement in F1-score and ∼6% improvement in AUC score over the baseline.

7.2 Feature Importances

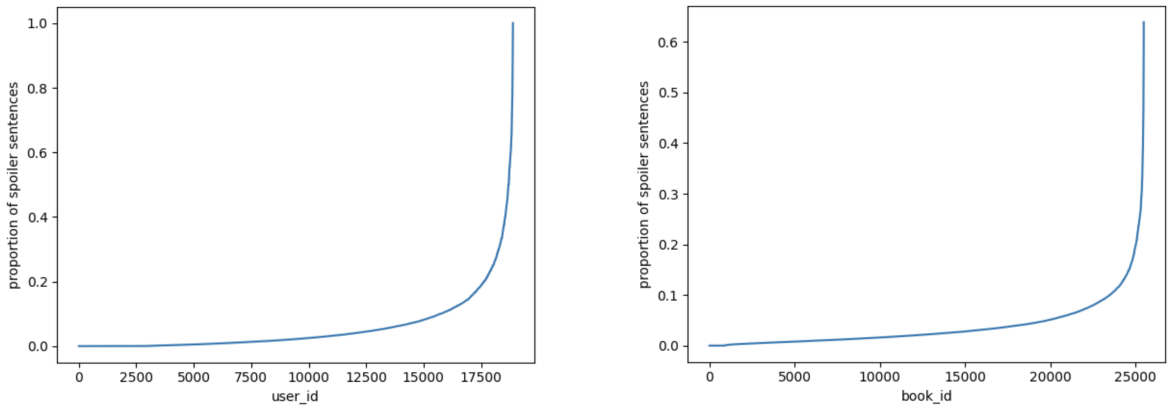
It is difficult to quantify the importance of features given that almost all our features come from dense embeddings and therefore don't have an easy to understand meaning. Regardless, for our best performing

model (logistic regression), as a rough approximation, we quantify the importance of a feature by dividing the magnitude of its coefficient in the model by the sum of the magnitudes of the coefficients for all features. If done this way, the average importance of a feature would be 100/#features ∼0.259%. Here, the feature importance is 1.830% for *rel_pos_sentence*, and 0.184% for *__len_sentence_log* (sentence-BERT accounts for all remaining importance). This suggests that most of the heavy-lifting is being done by sentence-BERT, which is to be expected especially since it accounts for almost all the features. However, it is reassuring to see that the feature importance of *rel_pos_sentence* is 1.830/0.259∼7 times the average and the feature importance of *__len_sentence_log* is less than the average. This lends some credence to our earlier suspicions that *rel_pos_sentence* can contribute some value to the models, while *__len_sentence_log* would be less meaningful.

7.3 Challenges & Future Scope

This project has presented us with some significant challenges such as a class imbalance problem and an overlap between the sets of spoiler and not spoiler sentences (mentioned in the Data Set section). Despite only working with ∼1% of the ∼17 million sentences in our dataset, our use of feature engineering, different class weightings, sampling strategies, and loss functions, has enabled us to significantly outperform random guessing/a majority classifier and beat our baseline AUC score by ∼6%.

Another challenge with the dataset is highlighted in Figure 6. The first plot shows the proportion of spoiler sentences corresponding to each user (i.e. the no. of spoiler sentences written by the user divided by the total no. of sentences written by the user). The second plot shows the proportion of spoiler sentences corresponding to each book (i.e. the no. of spoiler sentences written about the book divided by the total no. of sentences written about the book). Both plots follow the same exponential trend suggesting that, relative to others, there are certain users who are highly prone to writing spoilers and there are certain books that are highly prone to having spoilers written about them.



(a) (b)

Figure 6: (a) Users sorted in increasing order of the proportion of spoiler sentences (b) Books sorted in increasing order of the proportion of spoiler sentences

This brings us back to the emphasis of past work for this task on non-textual meta-features. We would expect that adding demographical information about the users and information about the books eg. genre would add significant predictive power to our models. In the real world, while information about users might not be easily accessible (or ethical to use for that matter), information about books should be fair game. In our case we only had access to anonymized IDs, which we feel are difficult to work with and don't easily generalize to unseen data. We have done the best that we could do with the text data and our efforts have yielded some meaningful results. An obvious next step would be to draw on datasets from earlier work for this task that do contain user/book specific information, and to marry that data with modern NLP approaches such as large-language model based dense embeddings and transfer learning. It would be interesting to see how

well these approaches can work with and augment the capabilities of that data.

Acknowledgments

We are sincerely thankful to Dr. Ungar for making such a difficult course enjoyable and informative by offering us so many resources eg. lecture notes, quizzes, worksheets, homework assignments etc. We are also very thankful to the course TAs for being there for us when needed, and in particular to our home pod TA Matthew Pressimone for his meaningful advice throughout the duration of the project.

References

[1] Jordan L. Boyd-Graber, Kimberly Glasgow, and Jackie Sauter Zajac. Spoiler alert: Machine learning approaches to detect social media posts with revelatory information. In *Beyond the Cloud: Rethinking Information Boundaries - Proceedings of the 76th ASIS&T Annual Meeting, ASIST 2013, Montreal, Canada, November 1-5, 2013*, volume 50 of *Proc. Assoc. Inf. Sci. Technol.*, pages 1–9. Wiley, 2013.

[2] Buru Chang, Hyunjae Kim, Raehyun Kim, Deahan Kim, and Jaewoo Kang. A deep neural spoiler detection model using a genre-aware attention mechanism. In *PAKDD*, 2018.

[3] Sheng Guo and Naren Ramakrishnan. Finding the storyteller: Automatic spoiler tagging using linguistic cues. In *Proceedings of the 23rd International Conference on Computational Linguistics (Coling 2010)*, pages 412–420, Beijing, China, August 2010. Coling 2010 Organizing Committee.

[4] Hidenari Iwai, Yoshinori Hijikata, Kaori Ikeda, and Shogo Nishida. Sentence-based plot classification for online review comments. In *2014 IEEE/WIC/ACM International Joint Conferences on Web Intelligence (WI) and Intelligent Agent Technologies (IAT)*, volume 1, pages 245–253, 2014.

[5] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr DollÃ¡r. Focal loss for dense object detection, 2017.

[6] Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. *CoRR*, abs/1908.10084, 2019.

[7] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of BERT: smaller, faster, cheaper and lighter. *CoRR*, abs/1910.01108, 2019.

[8] Mengting Wan, Rishabh Misra, Ndapa Nakashole, and Julian McAuley. Fine-grained spoiler detection from large-scale review corpora. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 2605–2610, Florence, Italy, July 2019. Association for Computational Linguistics.

Introduction

Our project is about spoiler classification in book reviews. Our dataset is based on book reviews from www.goodreads.com. We frame the problem as a sentence-level binary classification task: given a sentence in a review, we aim to predict if it contains spoilers or not.

Our best performing model is logistic regression. Starting from the original dataset, we provide an end-to-end walkthrough of the same below.

Install/Import Packages

In [ ]:
```
# We install the sentence-transformers package to generate
# dense embeddings for sentences.
!pip install sentence-transformers
```

In [ ]:
```
from collections import defaultdict
from google.colab import drive
import imblearn
import itertools
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import plotly.express as px
import random
import seaborn as sns; sns.set_style("ticks")
from sentence_transformers import SentenceTransformer
from sklearn import metrics
from sklearn.linear_model import LogisticRegression
from sklearn.manifold import TSNE
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from tqdm import tqdm; tqdm.pandas()
from wordcloud import WordCloud
```

Data Preprocessing & Feature Engineering

The original dataset is review-level i.e. one review per row. We convert the dataset to sentence-level ie. one review sentence per row.

In [ ]:
```
# Load original dataset
!gdown --id ########## Put ID here ##########
```

In [ ]:
```
df = pd.read_json(
    path_or_buf="/content/goodreads_reviews_spoiler.json",
    lines=True
)
df.head(5)
```

In [ ]:
```
# review_id, user_id, and book_id are in string format. We convert them to a
# numeric format for space saving.
df.review_id = LabelEncoder().fit_transform(df.review_id)
df.user_id = LabelEncoder().fit_transform(df.user_id)
df.book_id = LabelEncoder().fit_transform(df.book_id)
```

In [ ]:
```
# Convert dataset from review-level to sentence-level
cache = defaultdict(list)

for i in tqdm(range(len(df))):
    _pos_sentence, _num_sentences = 0, len(df.review_sentences.iloc[i])

    for is_spoiler, sentence in df.review_sentences.iloc[i]:
        cache["timestamp"].append(df.timestamp.iloc[i])
        cache["review_id"].append(df.review_id.iloc[i])
        cache["user_id"].append(df.user_id.iloc[i])
        cache["book_id"].append(df.book_id.iloc[i])
        cache["review_rating"].append(df.rating.iloc[i])
        cache["is_spoiler"].append(is_spoiler)
        cache["sentence"].append(str(sentence))
        cache["_pos_sentence"].append(_pos_sentence)
        _pos_sentence += 1
        cache["_num_sentences"].append(_num_sentences)

# Save sentence-level dataset
drive.mount('drive')
pd.DataFrame(data=cache).to_csv("data.csv")
!cp data.csv "drive/My Drive/"
```

We load the sentence-level dataset, perform some preprocessing, create sentence embeddings, and split the data into train/val/test sets.

In [ ]:
```
# Load sentence-level dataset
!gdown ########## Put ID here ##########
```

In [ ]:
```
df = pd.read_csv(filepath_or_buffer="/content/data.csv")
df.head(5)
```

In [ ]:
```
# Drop unnamed column
df.drop(columns=["Unnamed: 0"], inplace=True)

# Drop rows with missing values
print(df.isnull().sum())
df.dropna(inplace=True)

# Drop sentences that have been marked as both spoilers and not-spoilers
spoiler_sentences = set(df.sentence[df.is_spoiler == 1])
not_spoiler_sentences = set(df.sentence[df.is_spoiler == 0])

df["flag"] = df.sentence.apply(
    lambda s: not ((s in spoiler_sentences) and (s in not_spoiler_sentences))
)
df = df[df.flag]

df.drop(columns=["flag"], inplace=True)
```

In [ ]:
```
# Truncate dataset to reduce computational/storage requirements
# Use random sampling to preserve class distributions
df_trunc = df.sample(n=199000, random_state=42, ignore_index=True)
```

In [ ]:
```
# Create pretrained sentence embeddings
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

sentence_transformer = SentenceTransformer(
    model_name_or_path="all-MiniLM-L6-v2", device=device
)
embeddings = sentence_transformer.encode(
    sentences=df_trunc.sentence.values,
    show_progress_bar=True,
```

```
    convert_to_numpy=True,
    device=device
)
df_trunc["sentence_embedding"] = pd.Series(list(embeddings))
```

In [ ]:
```
# Split dataset into train/val/test datasets
X_train, X_test_, y_train, y_test_ = train_test_split(
    df_trunc.loc[:, df_trunc.columns != "is_spoiler"],
    df_trunc["is_spoiler"],
    train_size=99000,
    test_size=100000,
    random_state=42,
    shuffle=True
)
X_val, X_test, y_val, y_test = train_test_split(
    X_test_,
    y_test_,
    train_size=50000,
    test_size=50000,
    random_state=42,
    shuffle=True
)

# Save datasets
drive.mount("drive")
X_train.to_pickle("X_train.pkl")
!cp X_train.pkl "drive/My Drive/"
X_val.to_pickle("X_val.pkl")
!cp X_val.pkl "drive/My Drive/"
X_test.to_pickle("X_test.pkl")
!cp X_test.pkl "drive/My Drive/"
y_train.to_pickle("y_train.pkl")
!cp y_train.pkl "drive/My Drive/"
y_val.to_pickle("y_val.pkl")
!cp y_val.pkl "drive/My Drive/"
y_test.to_pickle("y_test.pkl")
!cp y_test.pkl "drive/My Drive/"
```

We load the train/val/test sets, perform feature engineering, and setup the data to be used for training.

In [ ]:
```
# Load train/val/test datasets
!gdown ########## Put ID here ########## # X_train
!gdown ########## Put ID here ########## # X_val
!gdown ########## Put ID here ########## # X_test
!gdown ########## Put ID here ########## # y_train
!gdown ########## Put ID here ########## # y_val
!gdown ########## Put ID here ########## # y_test
```

In [ ]:
```
X_train = pd.read_pickle(filepath_or_buffer="/content/X_train.pkl")
y_train = pd.read_pickle(filepath_or_buffer="/content/y_train.pkl")
X_val = pd.read_pickle(filepath_or_buffer="/content/X_val.pkl")
y_val = pd.read_pickle(filepath_or_buffer="/content/y_val.pkl")
X_test = pd.read_pickle(filepath_or_buffer="/content/X_test.pkl")
y_test = pd.read_pickle(filepath_or_buffer="/content/y_test.pkl")

y_train = pd.DataFrame(y_train)
y_val = pd.DataFrame(y_val)
y_test = pd.DataFrame(y_test)
```

In addition to our pretrained sentence embeddings, we generate two additional features :- the relative position of the sentence in a review (i.e. the index of the sentence in the review divided by the no. of sentences in the review), and the log of the length of the sentence in characters.

In [ ]:
```
# Relative position of the sentence in the review
X_train["_rel_pos_sentence"] = X_train._pos_sentence / X_train._num_sentences
X_val["_rel_pos_sentence"] = X_val._pos_sentence / X_val._num_sentences
X_test["_rel_pos_sentence"] = X_test._pos_sentence / X_test._num_sentences

# Log of the no. of characters in the sentence
X_train["_len_sentence"] = X_train.sentence.apply(lambda s: len(s))
X_val["_len_sentence"] = X_val.sentence.apply(lambda s: len(s))
X_test["_len_sentence"] = X_test.sentence.apply(lambda s: len(s))
X_train["_len_sentence_log"] = X_train._len_sentence.apply(lambda x: np.log(x))
X_val["_len_sentence_log"] = X_val._len_sentence.apply(lambda x: np.log(x))
X_test["_len_sentence_log"] = X_test._len_sentence.apply(lambda x: np.log(x))
```

In [ ]:
```
# Create feature array of sentence embeddings, relative positions of sentences,
# and logs of lengths of sentences

X_train_sentence_embedding = np.array(X_train.sentence_embedding.tolist())
X_val_sentence_embedding = np.array(X_val.sentence_embedding.tolist())
X_test_sentence_embedding = np.array(X_test.sentence_embedding.tolist())

X_train_values = np.hstack(
    tup=(
        np.array(X_train.sentence_embedding.tolist()),
        np.expand_dims(X_train._rel_pos_sentence.values, axis=1),
        np.expand_dims(X_train._len_sentence_log.values, axis=1),
    )
)

X_val_values = np.hstack(
    tup=(
        np.array(X_val.sentence_embedding.tolist()),
        np.expand_dims(X_val._rel_pos_sentence.values, axis=1),
        np.expand_dims(X_val._len_sentence_log.values, axis=1),
    )
)

X_test_values = np.hstack(
    tup=(
        np.array(X_test.sentence_embedding.tolist()),
        np.expand_dims(X_test._rel_pos_sentence.values, axis=1),
        np.expand_dims(X_test._len_sentence_log.values, axis=1),
    )
)

y_train_values = y_train.values.ravel()
y_val_values = y_val.values.ravel()
y_test_values = y_test.values.ravel()
```

Data Exploration

In [ ]:
```
df = pd.concat(
    [
        pd.concat([X_train, X_val, X_test], axis=0, ignore_index=True),
        pd.concat([y_train, y_val, y_test], axis=0, ignore_index=True)
    ], axis=1
)
```

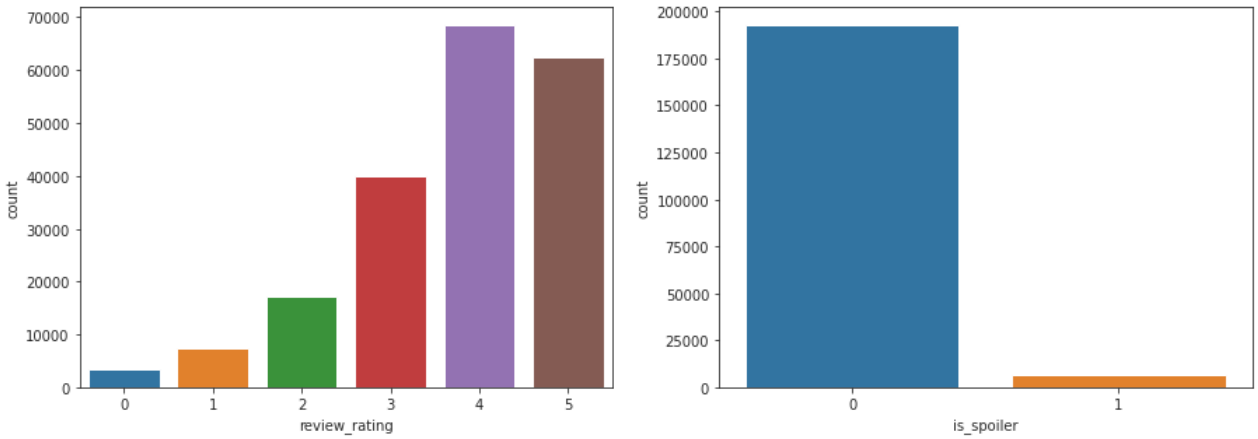
Summary Statistics

The following is a summary of our dataset. We note that our dataset suffers from a class imbalance problem, with only ~3% of the sentences being spoilers.

In [ ]:
```
print(f"No. of reviews              : {df.review_id.nunique():,}")
print(f"No. of users                : {df.user_id.nunique():,}")
print(f"No. of books                : {df.book_id.nunique():,}")
print(f"No. of sentences            : {len(df):,}")
print(f"% of sentences with spoilers: {round(df.is_spoiler.sum() * 100 / len(df), 2)}%\n"

fig, ax = plt.subplots(1, 2, figsize=(15, 5))
sns.countplot(x=df.review_rating, ax=ax[0])
sns.countplot(x=df.is_spoiler, ax=ax[1])
fig.show()
```

```
No. of reviews              : 172,004
No. of users                : 15,605
No. of books                : 23,450
No. of sentences            : 199,000
% of sentences with spoilers: 3.16%
```

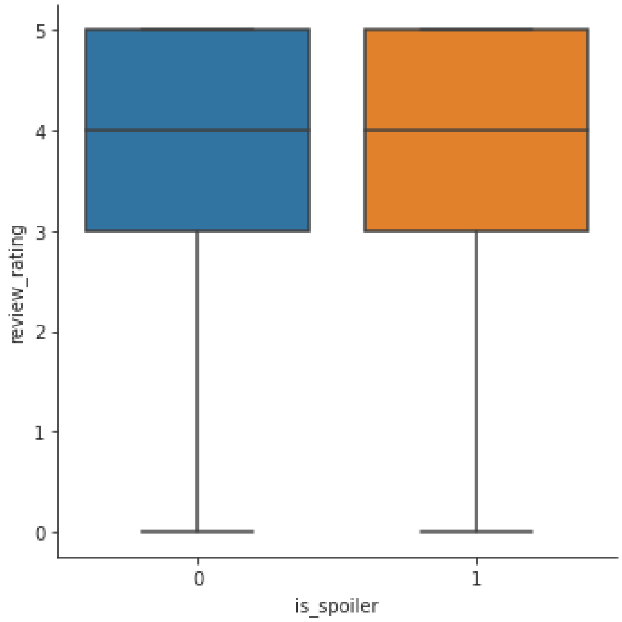


is_spoiler vs. review_rating

The following plot suggests that there is no discernible difference between not spoiler and spoiler sentences on the basis of review ratings

In [ ]:
```
sns.catplot(data=df, x="is_spoiler", y="review_rating", kind="box")
```

Out[ ]:
```
<seaborn.axisgrid.FacetGrid at 0x7fb4075e5be0>
```

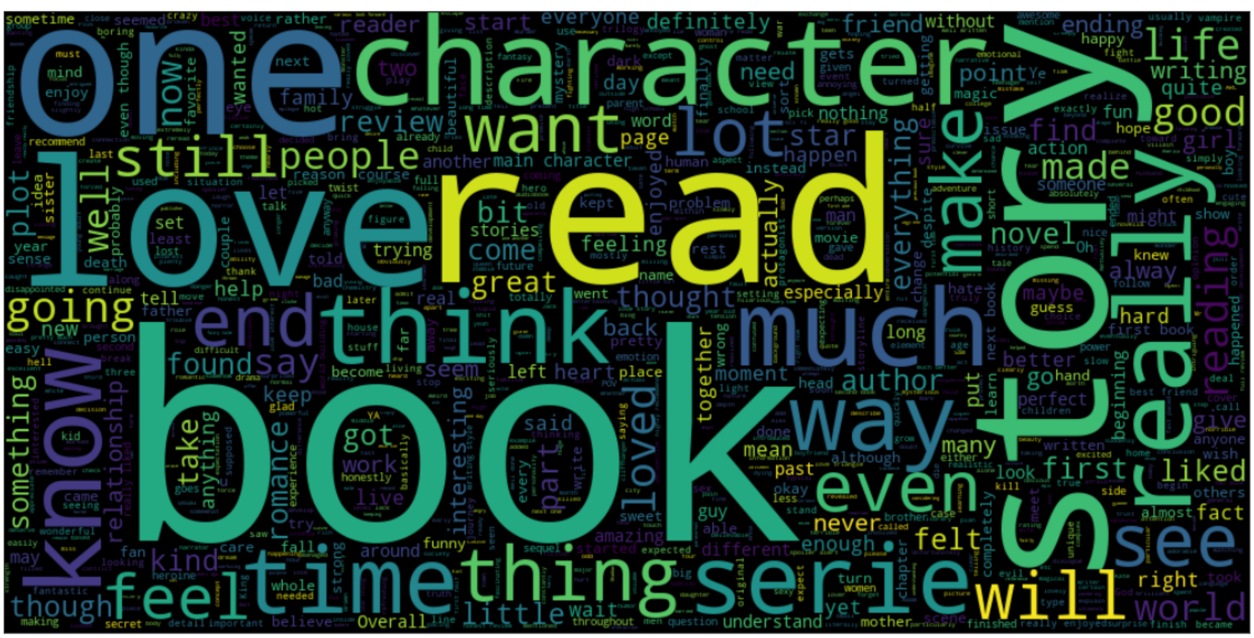


Wordclouds of not spoiler/spoiler sentences

In [ ]:
```
# Not spoiler sentences wordcloud
wc = WordCloud(max_words=2000, width=1000, height=500)
wc.generate(" ".join(df[df.is_spoiler == 0]["sentence"]))

plt.figure(figsize=(20, 20))
plt.imshow(wc, interpolation="bilinear")
plt.xticks([])
plt.yticks([])
```

Out[ ]:
```
([], <a list of 0 Text major ticklabel objects>)
```

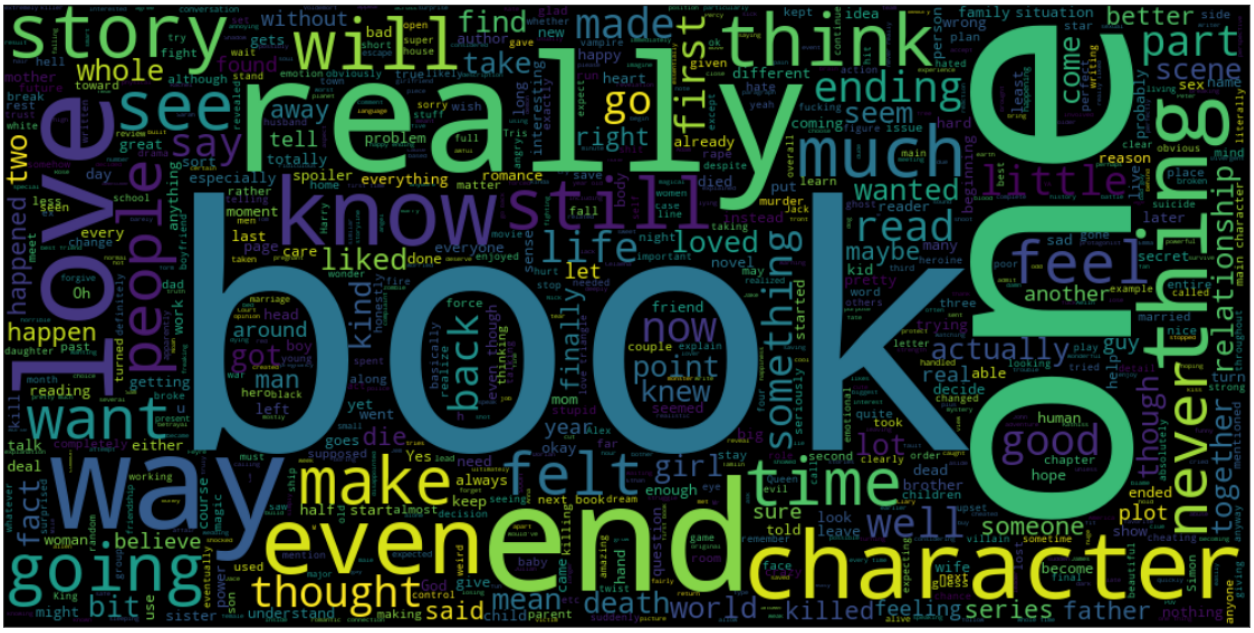


In [ ]:
```
# Spoiler sentences wordcloud
wc = WordCloud(max_words=2000, width=1000, height=500)
wc.generate(" ".join(df[df.is_spoiler == 1]["sentence"]))

plt.figure(figsize=(20, 20))
plt.imshow(wc, interpolation="bilinear")
```

```
plt.xticks([])
plt.yticks([])
```

Out[ ]:
```
([], <a list of 0 Text major ticklabel objects>)
```



Histogram of relative positions of sentences

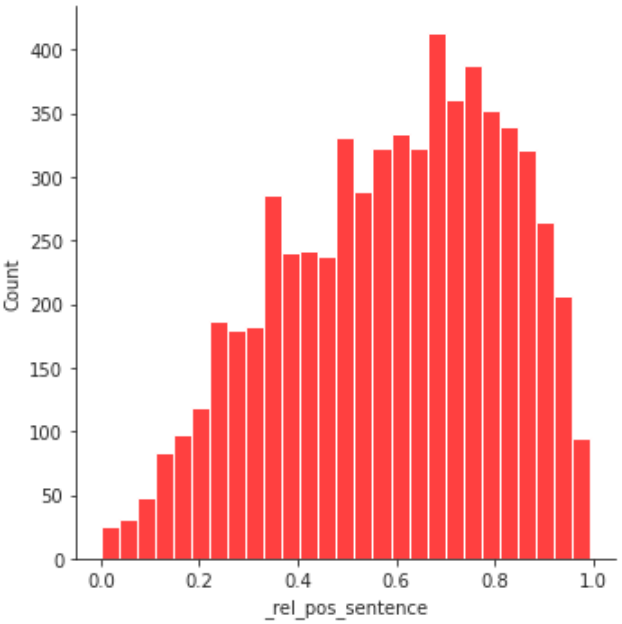
The skew in the histogram below suggests that spoiler sentences tend to appear later in the review. Hence this feature could be informative for spoiler classification.

In [ ]:
```
plt.figure(figsize=(10, 5))

sns.displot(
    data=df[df.is_spoiler == 1], x="_rel_pos_sentence", kind="hist", color="red"
)
```

Out[ ]:
```
<seaborn.axisgrid.FacetGrid at 0x7fbf4c2dbca0>

<Figure size 720x360 with 0 Axes>
```



Density plots of the logarithms of the lengths of sentences

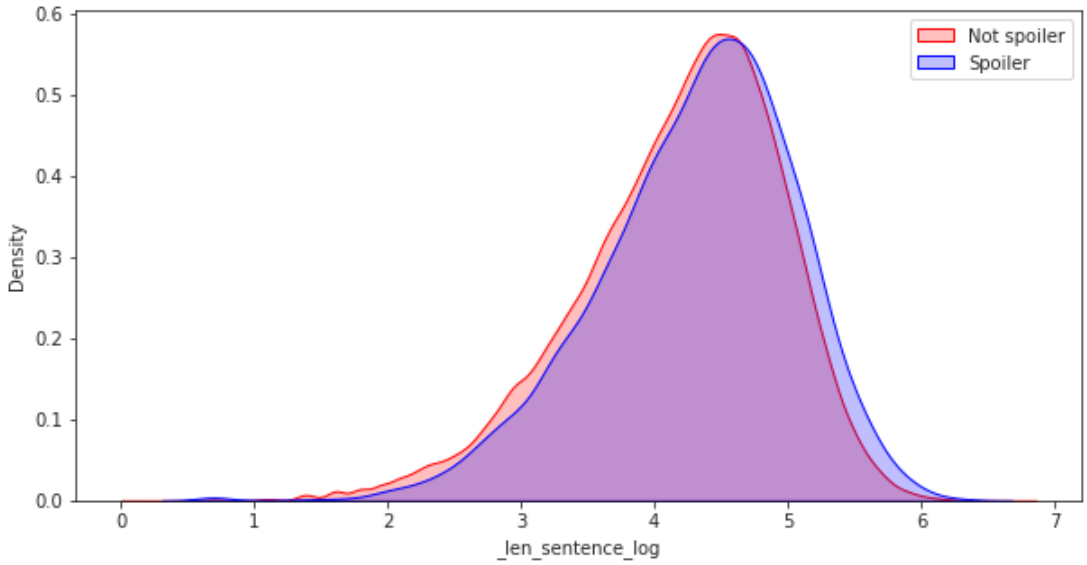
Given how marginal the difference is in the plot below, we are skeptical about the informativeness of this feature. However, we have included it in our models anyways with the understanding that it can be regularized out in the training process if it proves to be redundant.

In [ ]:
```
plt.figure(figsize=(10, 5))

sns.kdeplot(
    data=df[df.is_spoiler == 0],
    x="_len_sentence_log",
    shade=True,
    color="red",
    label="Not spoiler",
    clip=[0, 2000]
)
sns.kdeplot(
    data=df[df.is_spoiler == 1],
    x="_len_sentence_log",
    shade=True,
    color="blue",
    label="Spoiler",
    clip=[0, 2000]
)

plt.legend(loc="best")
```

Out[ ]:
```
<matplotlib.legend.Legend at 0x7fbf4c6c9bb0>
```



Utility Functions

We define some utility functions that are used during model training.

In [ ]:
```
def print_metrics(model, X, y):
    """Prints performance metrics"""
    y_pred = model.predict(X) # Predicted labels
    y_pred_proba = model.predict_proba(X) # Predicted probabilities

    # Test accuracy
    print(f"Accuracy: {metrics.accuracy_score(y_true=y, y_pred=y_pred)}\n")
```

```
    # Classification report
    print(metrics.classification_report(y_true=y, y_pred=y_pred))

    fig, (ax1, ax2) = plt.subplots(nrows=1, ncols=2, figsize=(15, 5))

    # Confusion matrix
    cm = metrics.confusion_matrix(y_true=y, y_pred=y_pred)
    sns.heatmap(cm, annot=True, fmt="d", ax=ax1, cmap=plt.cm.Blues, cbar=False)
    ax1.set(
        xlabel="Pred", ylabel="True",
        xticklabels=[0, 1], yticklabels=[0, 1],
        title="Confusion Matrix"
    )

    # ROC curve/AUC score
    fpr, tpr, thresholds = metrics.roc_curve(
        y_true=y, y_score=y_pred_proba[:, 1], pos_label=1
    )
    auc_score = metrics.roc_auc_score(y_true=y, y_score=y_pred_proba[:, 1])
    ax2.plot(
        fpr, tpr, color="red", label=f"ROC Curve (AUC = {round(auc_score, 3)})"
    )
    ax2.set_xlabel("False Positive Rate")
    ax2.set_ylabel("True Positive Rate")
    ax2.set_title("ROC Curve")
    ax2.legend(loc="best")
```

In [ ]:
```
def print_metrics_hyper(model, X, y):
    """Truncated version of print_metrics for hyperparameter tuning"""
    y_pred = clf.predict(X) # Predicted labels
    y_pred_proba = clf.predict_proba(X) # Predicted probabilities

    accuracy = metrics.accuracy_score(y_true=y, y_pred=y_pred)
    auc_score = metrics.roc_auc_score(y_true=y, y_score=y_pred_proba[:, 1])
    precision, recall, fbeta_score, _ = metrics.precision_recall_fscore_support(
        y_true=y, y_pred=y_pred, average="binary"
    )
    print(f"accuracy: {accuracy:.3f}, auc: {auc_score:.3f}, precision: {precision:.3f}, r
```

In [ ]:
```
def product_dict(**kwargs):
    """Helper function for hyperparameter tuning"""
    keys = kwargs.keys()
    vals = kwargs.values()
    return [dict(zip(keys, instance)) for instance in itertools.product(*vals)]
```

Logistic Regression

We experiment with the following three cases :-

1) Underweighting majority class + Hyperparameter tuning

In [ ]:
```
class_weights = [{0: w, 1: 1} for w in [0.01, 0.05, 0.1, 0.3, 0.5, 1]]
params = (product_dict(**{
    "penalty": ["elasticnet"],
    "class_weight": class_weights,
    "l1_ratio": [0, 0.5, 1]
}))

for param in params:
    penalty = param["penalty"]
    class_weight = param["class_weight"]
    l1_ratio = param["l1_ratio"]
```

```
    # Fit model
    clf = LogisticRegression(
        penalty=penalty,
        class_weight=class_weight,
        random_state=42,
        solver="saga",
        max_iter=1000,
        l1_ratio=l1_ratio
    )
    clf.fit(X_train_values, y_train_values)

    print(f"penalty: {penalty}, class_weight: {class_weight}, l1_ratio: {l1_ratio}")
    print_metrics_hyper(clf, X_train_values, y_train_values)
    print_metrics_hyper(clf, X_val_values, y_val_values)
```

2) Undersampling majority class + Hyperparameter tuning

In [ ]:
```
params = (product_dict(**{
    "sampling_strategy": [0.05, 0.1, 0.3, 0.5, 1],
    "penalty": ["elasticnet"],
    "l1_ratio": [0, 0.5, 1]
}))

for param in params:
    sampling_strategy = param["sampling_strategy"]
    penalty = param["penalty"]
    l1_ratio = param["l1_ratio"]

    # Undersample majority class
    X_train_rs_values, y_train_rs_values = \
    imblearn.under_sampling.RandomUnderSampler(
        sampling_strategy=sampling_strategy,
        random_state=42
    ).fit_resample(X_train_values, y_train_values)

    # Fit model
    clf = LogisticRegression(
        penalty=penalty,
        random_state=42,
        solver="saga",
        max_iter=1000,
        l1_ratio=l1_ratio
    )
    clf.fit(X_train_rs_values, y_train_rs_values)

    print(f"sampling_strategy: {sampling_strategy}, penalty: {penalty}, l1_ratio: {l1_rat
    print_metrics_hyper(clf, X_val_values, y_val_values)
```

3) Oversampling minority class + Hyperparameter tuning

In [ ]:
```
params = (product_dict(**{
    "sampling_strategy": [0.05, 0.1, 0.3, 0.5, 1],
    "penalty": ["elasticnet"],
    "l1_ratio": [0, 0.5, 1]
}))

for param in params:
    sampling_strategy = param["sampling_strategy"]
    penalty = param["penalty"]
    l1_ratio = param["l1_ratio"]

    # Oversample minority class
    X_train_rs_values, y_train_rs_values = imblearn.over_sampling.SMOTE(
        sampling_strategy=sampling_strategy,
        random_state=42
```

```
    ).fit_resample(X_train_values, y_train_values)

    # Fit model
    clf = LogisticRegression(
        penalty=penalty,
        random_state=42,
        solver="saga",
        max_iter=1000,
        l1_ratio=l1_ratio
    )
    clf.fit(X_train_rs_values, y_train_rs_values)

    print(f"sampling_strategy: {sampling_strategy}, penalty: {penalty}, l1_ratio: {l1_rat
    print_metrics_hyper(clf, X_val_values, y_val_values)
```

Below we show the metrics for the best performing model

In [ ]:
```
# Train the model
clf = LogisticRegression(
    penalty="elasticnet",
    class_weight={0: 0.1, 1: 1},
    random_state=42,
    solver="saga",
    max_iter=1000,
    l1_ratio=0.5
)
clf.fit(X_train_values, y_train_values)
```

Out[ ]:
```
LogisticRegression(class_weight={0: 0.1, 1: 1}, l1_ratio=0.5, max_iter=1000,
                   penalty='elasticnet', random_state=42, solver='saga')
```

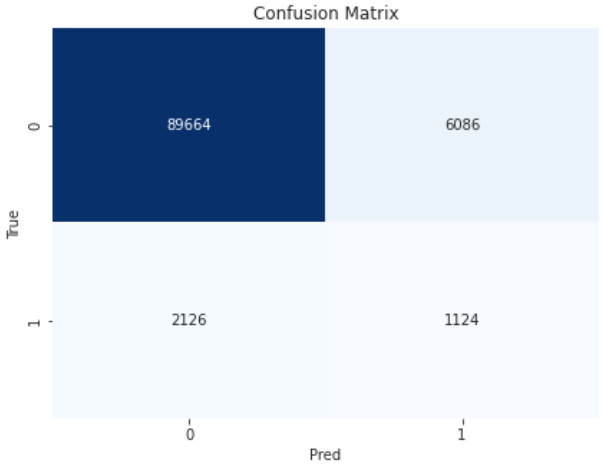
In [ ]:
```
# Print metrics on train dataset
print_metrics(clf, X_train_values, y_train_values)
```

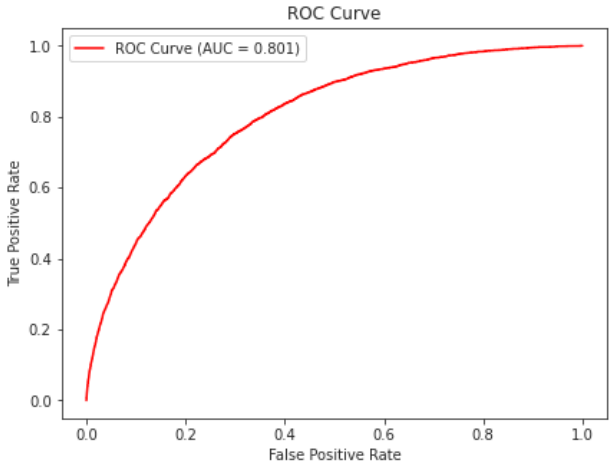
```
Accuracy: 0.917050505050505

              precision    recall  f1-score   support

           0       0.98      0.94      0.96     95750
           1       0.16      0.35      0.21      3250

    accuracy                           0.92     99000
   macro avg       0.57      0.64      0.59     99000
weighted avg       0.95      0.92      0.93     99000
```





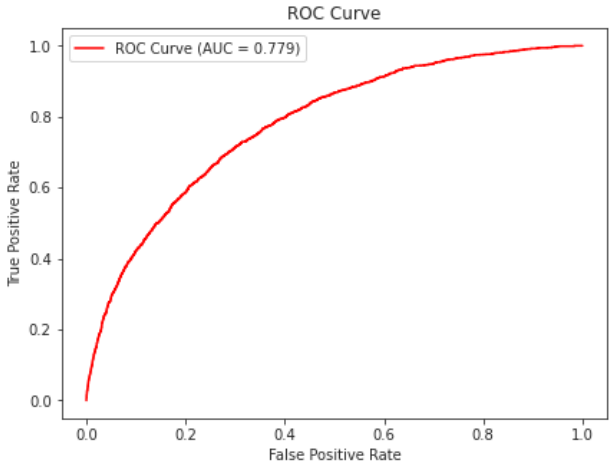
In [ ]:
```
# Print metrics on val dataset
print_metrics(clf, X_val_values, y_val_values)
```

```
Accuracy: 0.91702

              precision    recall  f1-score   support

           0       0.98      0.94      0.96     48488
           1       0.14      0.34      0.20      1512

    accuracy                           0.92     50000
   macro avg       0.56      0.64      0.58     50000
weighted avg       0.95      0.92      0.93     50000
```



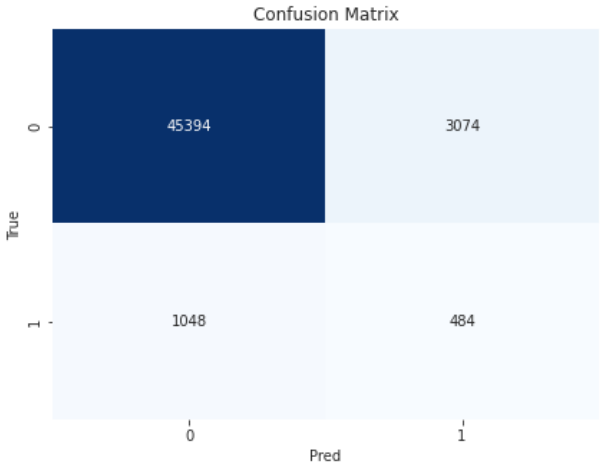
In [ ]:
```
# Print metrics on test dataset
print_metrics(clf, X_test_values, y_test_values)
```

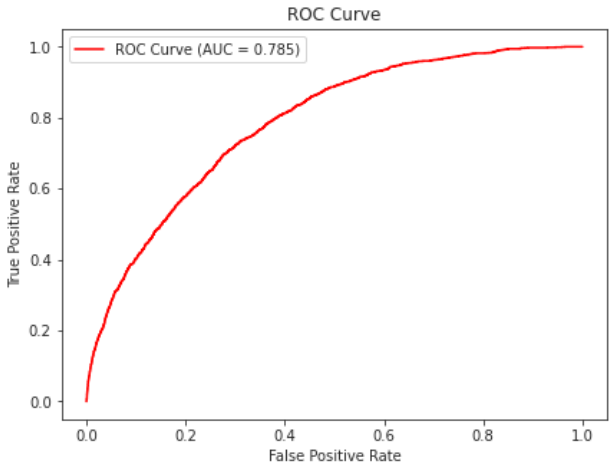
```
Accuracy: 0.91756

              precision    recall  f1-score   support

           0       0.98      0.94      0.96     48468
           1       0.14      0.32      0.19      1532

    accuracy                           0.92     50000
   macro avg       0.56      0.63      0.57     50000
weighted avg       0.95      0.92      0.93     50000
```





In [ ]:
```
# Combined ROC curve for train/val/test datasets

y_pred = clf.predict(X_train_values)
y_pred_proba = clf.predict_proba(X_train_values)
fpr, tpr, thresholds = metrics.roc_curve(
    y_true=y_train_values, y_score=y_pred_proba[:, 1], pos_label=1
)
```

```
auc_score = metrics.roc_auc_score(
    y_true=y_train_values, y_score=y_pred_proba[:, 1]
)
plt.plot(fpr, tpr, color="red", label=f"Train")

y_pred = clf.predict(X_val_values)
y_pred_proba = clf.predict_proba(X_val_values)
fpr, tpr, thresholds = metrics.roc_curve(
    y_true=y_val_values, y_score=y_pred_proba[:, 1], pos_label=1
)
auc_score = metrics.roc_auc_score(
    y_true=y_val_values, y_score=y_pred_proba[:, 1]
)
plt.plot(fpr, tpr, color="blue", label=f"Validation")

y_pred = clf.predict(X_test_values)
y_pred_proba = clf.predict_proba(X_test_values)
fpr, tpr, thresholds = metrics.roc_curve(
    y_true=y_test_values, y_score=y_pred_proba[:, 1], pos_label=1
)
auc_score = metrics.roc_auc_score(
    y_true=y_test_values, y_score=y_pred_proba[:, 1]
)
plt.plot(fpr, tpr, color="green", label=f"Test")

plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Curve")
plt.legend(loc="best")
```

Out[ ]: <matplotlib.legend.Legend at 0x7fbf4bf07ee0>