Assignment Data Mining 2025: Classification for the Detection of Opinion Spam

Evangelos Skoulas 9591834
Radis Marios Toumpalidis - 7646089
Thijme Bouwmeester - 7045492

Course code: INFOMDM
Lecturer: Ad Feelders

October 16, 2025

1 Introduction and Problem Motivation

Online reviews strongly influence which hotels consumers choose and how much revenue flows to those hotels, which makes review platforms attractive targets for deceptive opinion spam designed to look genuine while misleading readers. [2,3] Beyond manipulation for competitive gain, recent reporting shows that criminal groups also weaponize fake reviews for extortion, flooding businesses with one-star ratings and then demanding payment to stop or remove the damage. [1]

In this research we build on the Chicago hotel benchmark introduced by Ott et al, which contains balanced sets of truthful and deceptive reviews for hotels. This study has shown that straightforward n-gram text classifiers can outperform human judges by a wide margin. [2,3] In our study we focused on the negative reviews because the assignment targets that subset and because negative deception poses a direct risk to reputation and revenue. We used the provided folds so that our evaluation is comparable to the original work and easy to reproduce and so that our claims about model quality rest on a stable test split.

The purpose of this paper is to clarify which choices matter most for detecting deception in negative hotel reviews. We assessed whether stronger models such as logistic regression, decision trees, random forests and gradient boosting yield gains over standard linear baselines and whether adding bigrams to unigrams improves performance in a consistent way. We also examined which terms were most associated with deceptive reviews and which with truthful reviews so that our findings are interpretable and useful in practice. We report the accuracy of our tests on the provided folds, and we support the main comparisons with paired significance tests so the reader can judge whether differences are likely to matter beyond chance.

2 Data Description

In this section we describe the dataset and key exploratory patterns.

2.1 Dataset Overview

The dataset contains 800 human-generated hotel reviews split evenly into 400 deceptive and 400 truthful negative reviews. Deceptive reviews were written on Amazon Mechanical Turk with one fake review per worker across 20 Chicago hotels (20 reviews each). Truthful negative reviews (1–2 stars out of 5) were collected from travel sites like Expedia, TripAdvisor and Yelp. This resulted in a balanced and well-labeled dataset suited for classification tasks. [2]

2.2 Exploratory Analysis

I. Word and Character Count
Analysis was conducted on the full text before preprocessing.

Label	Mean	Std Dev	25%	50%	75%	Max
Deceptive	177.5	93.5	115	160	222	784
Truthful	178.7	100	112	157	220	749

Table 1: Word Count Statistics by Label

Label	Mean	Std Dev	25%	50%	75%	Max
Deceptive	947.5	947.5	604	865	1175	4075
Truthful	964.3	964.3	606	842	1159	4159

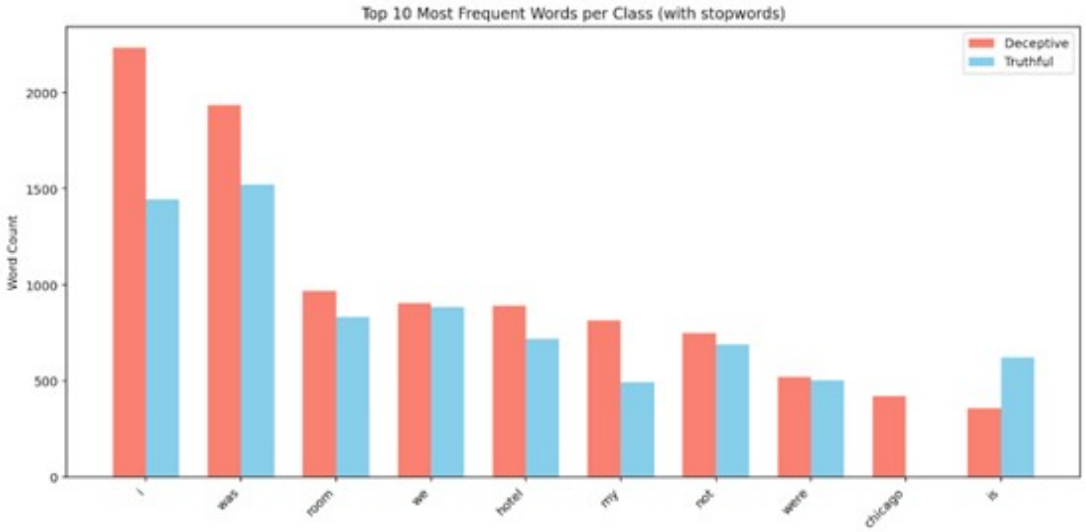
Table 2: Character Count Statistics by Label

The word and character length distributions are very similar between classes, indicating that text length alone probably does not contain much discriminative information.

II. Unique Word Count
Deceptive reviews contain slightly fewer unique words (5,200 vs 6,000) indicating lower vocabulary diversity. This could serve as a discriminative feature either by analyzing the total number of unique words per review or by examining the uniqueness of specific terms within each review.

III. Most common words
A)Include full vocabulary: Top 10 most frequent words were dominated by common stopwords (a, the and) offering little insight.

B)Include specific stop words: As suggested by Ott et al [2], deceptive reviews show more personal pronouns and experiential words ("I", "we", "my", "Chicago") thus providing background information unrelated to the hotel being reviewed while truthful reviews are more neutral.



C)All stopwords removed: The top descriptive words are almost identical across classes meaning they cannot be used as differentiators.

To sum up, deceptive reviews emphasize personal narrative (pronouns, self-reference) whereas truthful reviews use more neutral language. Removing stopwords confirms that the actual descriptive content is very similar across classes.

3 Pre-Processing

In this section we detail the preprocessing pipeline, including text cleaning, tokenization, and feature construction.

3.1 Dataset Construction

In order to achieve optimal performance across all trained algorithms and ensure their comparability, we applied the same filtering and pre-processing steps in the initial reviews. Before generating the document-term matrix, we converted all reviews to lowercase and punctuation marks were removed using the built-in Python class string, which provides a predefined list of English punctuation symbols. We then removed all numerical characters and any extra white spaces. Finally, we filtered out several non-informative tokens such as regular common English stop words ("a","and","the") and topic-related words like "hotel".

3.2 Text Representation

One of the simplest and most common methods for representing text as numerical vectors is the bag-of-words method, which represents each document as a collection of token counts indicating the frequency of each term across the corpus. In our implementation, we utilized the text vectorization functionalities provided by the *sklearn.feature_extraction.text.CountVectorizer* class. This function can exclude tokens with a document frequency of less than a p1% or more than a p2%. The class can also be parameterized to retain only the top N features ranked by term frequency, as well as to specify the range of n-grams to consider. Another option we explored was the *sklearn.feature_extraction.text.TfidVectorizer* class, which transforms text into numerical features by combining term frequency (TF) with inverse document frequency (IDF). This gives higher weight to terms that are more distinctive across documents, helping the model focus on important words. We used similar configuration options as with CountVectorizer such as controlling document frequency, number of features and n-gram range. Finally, we tested the *TfidVectorizer* with use_idf=False which only considers the term frequency component without applying the IDF weighting. This effectively produces a normalized version of raw term counts while keeping the same configurable parameters as the previous vectorizers.

3.3 Feature Extraction

Regardless of its simplicity and effectiveness, the bag of words model does not capture higher-level information that can be derived from lexical analysis. As a result, to potentially enhance the performance of all algorithms we extracted the 9 lexical features presented below from each document into a matrix of lexical features. Then, we combined the document term matrix with the normalized lexical features matrix, which was normalized to fit its corresponding training algorithm.

1. **Number of unique words per text**: motivated by the exploratory analysis, which suggested that truthful reviews tend to use a more diverse vocabulary.
2. **Number of words per text**.
3. **Average word length**.
4. **Number of characters per text**: to test if the total number of characters provides similar insights.
5. **Number of sentences per text**.
6. **Average sentence length**.
7. **Standard deviation of sentence length**.
8. **Overall negativity score**: inspired by the work of Ott et al. which indicates that fake hotel reviewers exaggerate sentiment relative to truthful reviews.
9. **Punctuation ratio**: motivated by the hypothesis above.

These features were designed to complement the vector that is derived from representing the text as a bag of words where each group of n words constitutes a feature.

4 Algorithms

In this section we describe the algorithms used for the assignment.

4.1 Multinomial Naive Bayes

Multinomial Naive Bayes (MNB) is a probabilistic classifier commonly used for text classification tasks such as deception detection. It represents the frequency distribution of words in documents for each class while assuming that features (i.e. the number of occurrences of a text token) are conditionally independent given the class label. The model calculates the likelihood of a document belonging to a particular class based on the product of the probabilities of its text tokens within that class.To resolve the issue of zero probabilities for unseen features, Laplace smoothing is utilized by adding a small constant to every word count prior to normalization, which is also the only configurable hyper-parameter of MNB.

4.2 Logistic Regression

Logistic Regression is a linear classification algorithm that estimates the likelihood of a sample belonging to a specific class using the logistic function. It estimates a set of weights for each feature to find a decision boundary that best separates the classes based on their feature values. A logistic regression on the task of minimizing the log-loss function via an optimization algorithm. To avoid overfitting, a regularization function is applied, in our case the LASSO L1 penalty, to constrain the size of the learned weights. The final model generates class probabilities, facilitating reliable decision-making even in high-dimensional feature spaces, such as those found in text data.

4.3 Classification Tree

A classification tree is a predictive model that keeps splitting the feature space into regions until it can assign each data point to a specific category. At each split the algorithm picks a feature and a suitable threshold value for it in order to reduce impurity. The goal is to separate the data into groups that are as "pure" as possible meaning they contain data points that belong to the least possible number of categories.The impurity at each node is commonly calculated using Gini or Entropy measure.
The performance of a classification tree depends mainly on the quality of the data but also on a set of important hyperparameters that control how the model grows and splits: [6]

1. **max_depth**: limits how deep the tree can grow.
2. **min_samples_split**: minimum number of samples required to split a node.
3. **min_samples_leaf**: minimum number of samples required in a leaf node; very small values can lead to overfitting.
4. **criterion**: impurity measure used to evaluate splits (commonly `gini` or `entropy`).
5. **ccp_alpha**: complexity parameter for cost-complexity pruning, used to remove branches and prevent overfitting.

Getting the right combination of these parameters helps balance bias and variance. Deeper trees usually reduce bias but can overfit while shallower ones might underfit but generalize better.

4.4 Random Forest

A Random Forest is an extension of decision trees that builds many trees and combines their predictions. Each tree is trained on a random sample of the training data (sometimes with replacement) and at each split considers only a random subset of features. The final prediction comes from aggregating the votes or averages of all the trees.
Key hyperparameters as described in *sklearn.ensemble.RandomForestClassifier* include:

1. **n_estimators**: number of trees in the forest; more trees usually improve performance but increase training time.
2. **max_depth**: maximum depth of each tree.
3. **min_samples_split** / **min_samples_leaf**: control how trees grow and split, similar to decision trees; prevent overly complex trees.
4. **max_features**: number of features considered at each split.
5. **bootstrap**: indicates whether each tree is trained using bootstrap sampling (sampling with replacement).
6. **ccp_alpha**: controls the amount of post-pruning applied to reduce overfitting.

Random Forests, compared to a single decision tree, reduce variance by averaging the predictions of many trees and can also handle large feature sets efficiently. As a result, random forests generally achieve better performance and generalize more effectively on unseen data.

4.5 Gradient Boosting

Gradient Boosting builds an ensemble of shallow trees where each new tree corrects the mistakes of the previous ones. This lets the model capture non-linear patterns in sparse text features while staying relatively simple to tune. We use scikit-learn's GradientBoostingClassifier.
Key hyperparameters

1. **n_estimators**: number of boosting stages (number of trees added sequentially).
2. **learning_rate**: controls how much each new tree contributes to the model.
3. **max_depth**: maximum depth of each individual tree.
4. **subsample**: fraction of the training data used per boosting stage (introduces randomness to reduce overfitting).
5. **min_samples_leaf** / **min_samples_split**: control the minimum number of samples required to create leaves or splits.
6. **max_features**: number of features randomly selected when looking for the best split.

Within our five-model set, Gradient Boosting is among the strongest performers because it captures non-linear interactions between terms. It usually needs a bit more tuning than the linear baselines, but it delivers better accuracy than several of the other algorithms.

5 Experimental Setup

In this section, we explain the experimental setup, covering data splits, hyperparameter tuning and evaluation metrics.

5.1 Multinomial Naive Bayes

We implemented the Multinomial Naive Bayes (MNB) algorithm using the "MultinomialNB" class from the _scikit-learn_ library. This model is particularly suitable for text classification tasks where features represent term frequencies, as it assumes that feature values follow a multinomial distribution conditioned on the class label. The model was trained on the bag-of-words representation [2] described in the previous section combined with the lexical features set. Prior to training, the feature vectors were normalized to ensure comparability across documents of different lengths. The primary hyperparameter that can be fine-tuned in the multinomial Naive Bayes model is the smoothing parameter, commonly denoted as 'a' (alpha) in the literature, which prevents "zero probabilities for unseen features in the bag-of-words representation by adding a constant to all term frequencies.

The smoothing value can either be equal for all features or feature-specific. In our study we adopted a feature-specific approach, as it provides a way to incorporate prior knowledge about a feature before the model is trained. For example, highly discriminative words may benefit by smoothing values lower than 1, reflecting their importance in the

classification task. Several methods can be used to generate a feature-specific smoothing array, such as basing it on feature frequency across the corpus, or on each feature's class distribution. In our implementation, we trained multinomial Naive Bayes under two distinct hyperparameter configurations. In the first, a uniform smoothing value of 1 was assigned to all features. In the second configuration a feature-specific smoothing array was computed based on each feature's class distribution for the *_bag-of-words_* features, while extracted lexical features were assigned a fixed smoothing value of 1. To identify an optimal smoothing array based on a feature's class distribution, we implemented a function that assigned a smoothing value of 0.3 to a feature it was highly discriminative—defined as appearing in more than 80% of the documents within a class—and a value of 2.0 if it appeared in fewer than 30% of the documents of that class.

Each model configuration was evaluated using k-fold cross-validation with several k values, chosen such that k evenly divided the training set. Model performance was assessed using the mean accuracy, precision, recall and F1-score across folds and retrained the resulting model in all the training data. Throughout experiments with different configurations of the smoothing array and the maximum size of feature sets, the feature-specific approach proved to always increase test accuracy by 1-3%. Further details are provided in the Results section.

5.2 Logistic Regression

The Logistic Regression classifier was implemented with the scikit-learn's *linear_model.LogisticRegression* class, using the same process of combining the bag-of-words representation with the lexical feature matrix as in the Naive Bayes model. Since we applied the L1 LASSO penalty for all the possible configurations of our model, the only configurable parameters were the inverse regularization strength, denoted as *C* , the optimization algorithm, denoted as *solver*, which must support the LASSO penalty; and the class weights, which we did not adjust since the dataset had equal frequency among classes.
There are only two optimization algorithms supported for the LASSO penalty: "liblinear", "saga". The first is more robust and performs better on smaller datasets, while the second is designed to better fit larger datasets or sparse representations.
As a result we applied the same cross validation strategy with Naive Bayes within a varying number of splits between 5,10 and the total number of bag-of-words features based on term frequency, ranging between 150 and 600, combined with the derived lexical features. In addition, we tuned Logistic Regression across the 0.1, 0.7, 1, 2 values for the inverse regularization strength and across 'liblinear', 'saga' for the optimization algorithm.

5.3 Decision Trees, Random Forests

First, we picked reasonable ranges for the main hyperparameters of each model and ran experiments varying the vectorizer type (Count or TF-IDF), number of features and n-gram choice (unigrams or unigrams+bigrams). The goal was to see which combinations gave the highest average 5-fold cross-validation accuracy. Next, we ran a Grid Search over all remaining hyperparameters for the best-performing setups to find the overall best model for each algorithm. Finally, we trained the selected model on the full dataset and evaluated it using accuracy, precision, recall, F1-score and its confusion matrix.

For the classification tree, we explored a wide range of hyperparameter values to control model complexity and prevent overfitting. Specifically, the maximum depth of the tree (max_depth) was tested with values from 5 up to 30, as well as allowing unlimited depth. The minimum number of samples required to split a node (min_samples_split) was varied between 2, 5, 10 and 15, while the minimum samples allowed in a leaf node (min_samples_leaf) ranged from 1 to 10. We also compared both impurity measures available in scikit-learn (criterion = gini and entropy) and experimented with cost-complexity pruning (ccp_alpha) using values between 0.0 and 0.1.

For the random forest model, we used similar hyperparameters but extended them to account for ensemble learning. We tested different numbers of trees (n_estimators) with values 100, 200 and 500, and examined different limits on tree depth (max_depth) including 10, 20, 30 and no depth limit. We also tuned the splitting behavior using min_samples_split values of 2, 5, 10 and 15, and min_samples_leaf values of 1, 2, 5 and 10. To control feature selection at each split, we used max_features="log2", and evaluated both bootstrap sampling strategies (bootstrap=True and False). Finally, the pruning parameter ccp_alpha was tested with values 0.0, 0.001 and 0.01.

5.4 Gradient Boosting

We turn reviews into features with Count or TF-IDF using unigrams and unigrams plus bigrams, capped at 500 or 1,000 terms with a minimum document frequency of two percent. For each representation we run RandomizedSearchCV over GradientBoosting-Classifier with five-fold stratified cross-validation, shuffling, accuracy scoring, and a fixed random seed of 42. The search draws 60 configurations under logistic loss and varies the number of trees (80–280), the learning rate (0.03–0.2), the maximum depth (2–4), the subsample fraction (0.75–1.0), the minimum samples required in a leaf (1–5), the minimum samples required to split (2–11) and the max features(None, sqrt, log2, 0.5). We refit the best model by mean cross-validated accuracy on folds one to four and evaluate it on fold five to report test accuracy, precision, recall, and F1.

6 Results

In this section we present the results, compare models across settings and summarize the main findings.

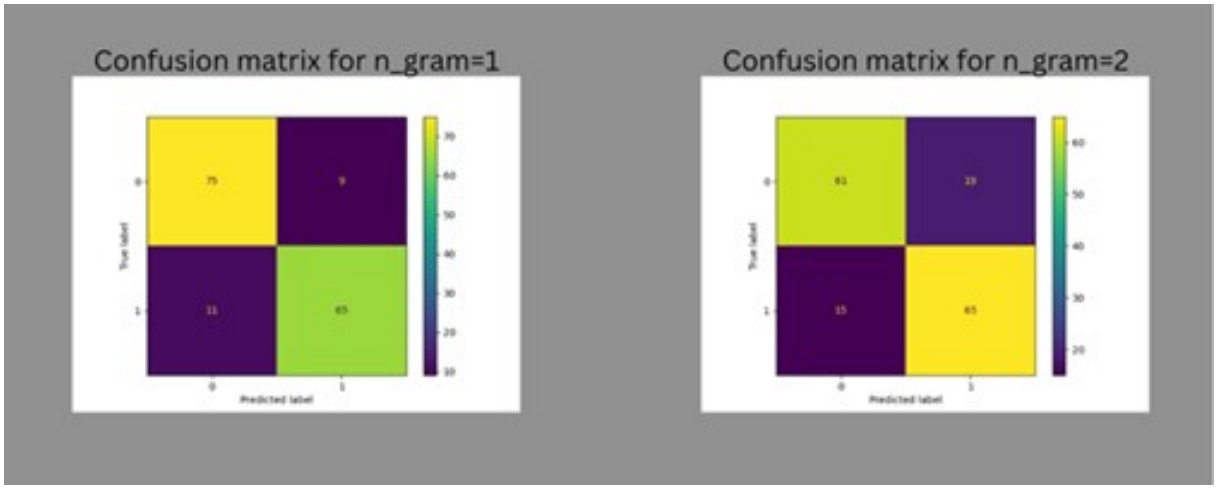
6.1 Model Performances

6.1.1 Multinomial Naive Bayes

Training iterations were conducted both with and without the inclusion of bi-grams, each time calculating the feature-specific smoothing array as described in Section 5.2. The number of bag-of-words features was varied according to term frequency thresholds to assess the effect of feature dimensionality on model performance.

On the test set, the best accuracy obtained for the model without bi-grams was 87.5% using 390 bag-of-words features, while the best accuracy for the model with bi-grams was 87.0% using 550 features. Both configurations achieved these accuracies consistently

across cross-validation with 5 and 10 splits. During hyperparameter fine-tuning the inclusion of feature specific smoothing array increased the cross-validation accuracy from approximately 82% to 87.5%, with precision recall and f1-score as 0.88,0.85,0.87 respectively for the uni-gram configuration and from 85% to 87%, with precision value of 0.774 recall value of 0.81 and f1-score as 0.79 for the bi-gram configuration, indicating that the feature-specific smoothing array and term frequency thresholds contributed positively to model generalization.



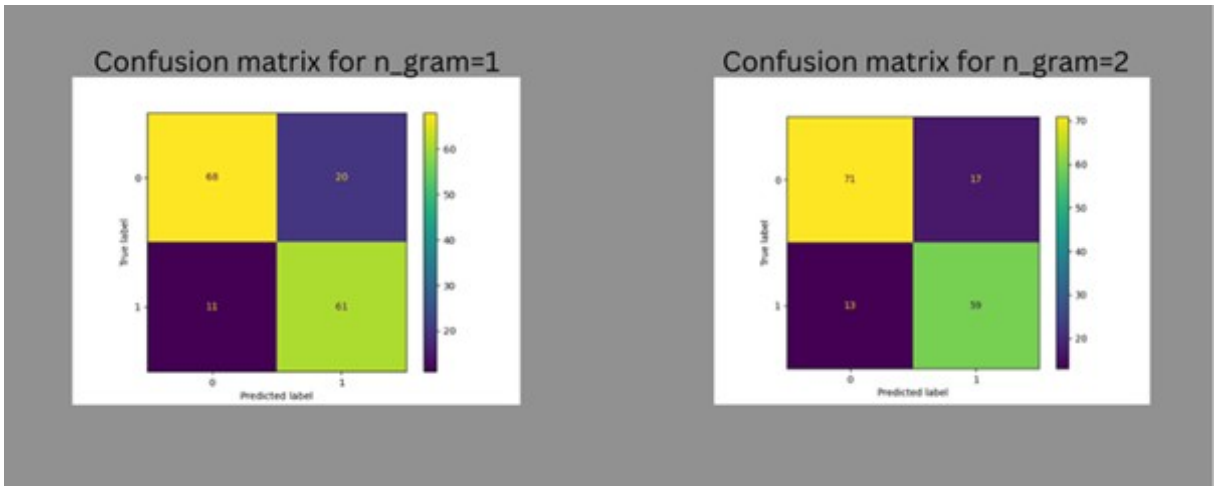
6.1.2 Logistic Regression

The Logistic Regression experiments followed the same feature configuration logic as the Naive Bayes model, varying the bag-of-words n-gram range and the number of features based on term frequency thresholds. Cross-validation was performed with 5 and 10 folds to identify the best hyperparameter configuration for the inverse regularization strength and the optimization algorithm . Across all tested configurations, the "saga" solver consistently outperformed "liblinear", likely due to its improved handling of high-dimensional sparse feature spaces typical of text data.

For 5-fold cross-validation, the best overall accuracy was 87.5% for uni-grams, with precision recall and f1-score as 0.75,0.85,0.80 respectively, obtained with 300 bag-of-words features and an inverse regularization strength of C = 0.1. While inspecting the regression model's attributes, particularly the coefficients array, we discovered that the top 5 terms pointing towards a deceptive review are ['luxury', 'recently', 'smelled', 'seemed', 'finally'] and the top 5 terms pointing towards a truthful review are .

The best performance for bi-grams under the same configuration was 86.0%, using 250 features, resulting in a precision value of 0.77, recall value of 0.81 and f1-score 0.79. Additionally, by analyzing the coefficient array of the model we discovered that the top 5 terms pointing towards a deceptive review are ['finally', 'seemed', 'decided', 'room and', 'chicago'], and the top 5 terms pointing towards a truthful review are ['door', 'many', 'manager', 'bed', 'am'].

For 10-fold cross-validation, the best overall accuracy reached 84.0% for both uni-grams and bi-grams, with bag-of-words feature sizes of 300 and 350, respectively, and an inverse regularization strength of C = 0.1.



Both the Multinomial Naive Bayes (MNB) and L1-regularized Logistic Regression classifiers demonstrated strong performance in the deception detection task, achieving comparable results overall. The highest test accuracy for the MNB model was 87.5% using uni-grams and 87.0% with bi-grams, while Logistic Regression achieved a maximum accuracy of 87.5% for uni-grams and 86.0% for bi-grams. Despite their peak performances being almost identical, the Logistic Regression model accomplished its best result with a smaller set of features (300 compared to MNB's 390), indicating that the LASSO regularization effectively minimized model complexity by emphasizing the most discriminative features. On the other hand, the Naive Bayes classifier showed more stability across different feature dimensions and gained more from the use of feature-specific smoothing, which improved its generalization ability. Overall, the results indicate that while both models are well-suited for textual deception detection, Naive Bayes offers slightly more robust performance, whereas Logistic Regression provides improved feature efficiency and interpretability due to its sparsity-inducing regularization. Overall, the findings suggest that while both models are suitable for detecting textual deception, Naive Bayes tends to offer slightly stronger performance with less computation time, whereas Logistic Regression enhances feature efficiency and interpretability due to its sparsity-inducing regularization.

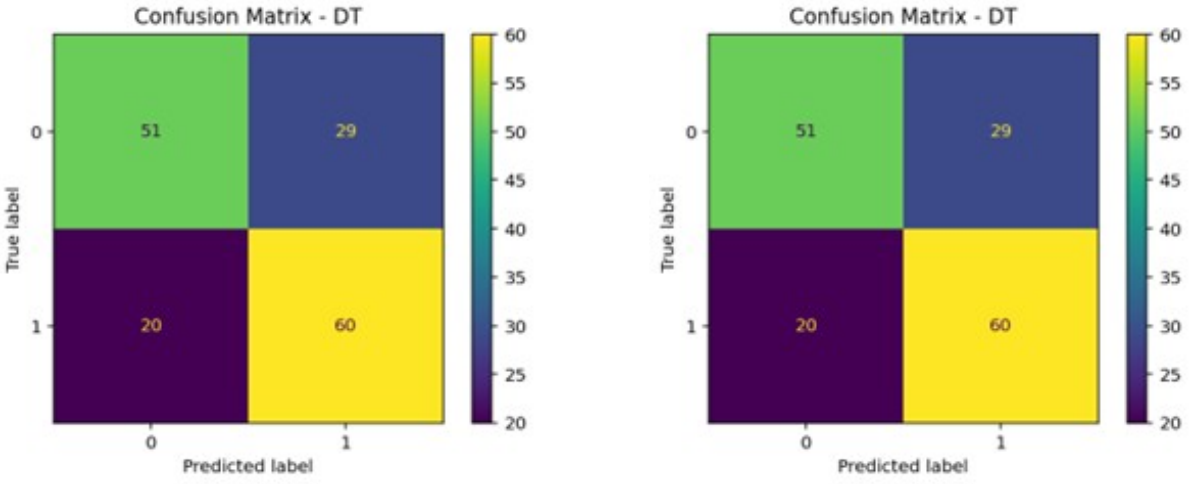
6.1.3 Decision Trees

At first, we tested different vectorizers, feature counts and n-gram ranges, evaluating each combination using 5-fold stratified cross-validation accuracy. CountVectorizer consistently achieved the best performance, outperforming the other vectorizers by 2–4%, suggesting that frequent word patterns are more informative than rare terms in this dataset. TF–IDF without IDF weighting performed worst, while TF–IDF with IDF weighting improved slightly but still lagged behind.Within CountVectorizer, unigrams barely outperformed the combination of unigrams and bigrams by about 0.5–1% and performance plateaued beyond 500 features which means that bigrams practically do not provide extra discriminative information for decision trees.

Next, we selected the five most promising configurations from the previous stage and performed an exhaustive grid search over additional hyperparameters, keeping the best feature and vectorizer settings fixed. The tuned parameters included max_depth, min_samples_split, min_samples_leaf, criterion, and ccp_alpha. The best-performing decision tree used a CountVectorizer with unigrams and 200 features, having also the following configuration:

max_depth = 5, min_samples_leaf = 10, min_samples_split = 2, ccp_alpha = 0.01, criterion = entropy.

Its performance metrics on the test set were:
Accuracy = 69.4%, Precision = 67.4%, Recall = 75.0%, F1-score = 71.0% which are also depicted in the confusion matrix shown below. We should note that the performance was identical when we included bigrams which suggests that they were largely ignored as features by the tree. These results indicate that a single decision tree tends to struggle with high-dimensional sparse text features and will easily overfit regardless of its configuration.



(a) Decision Tree - Unigrams

(b) Decision Tree - Unigrams + Bigrams

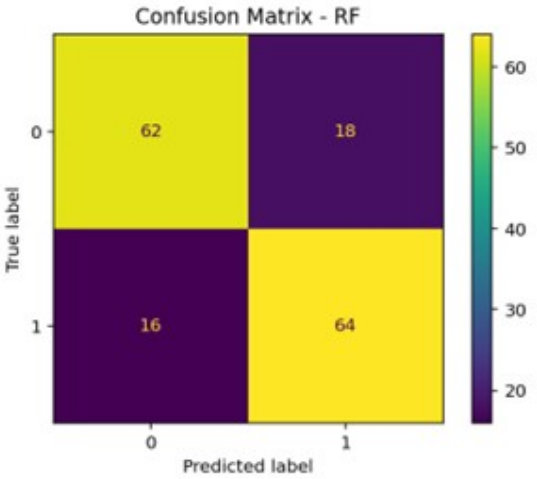
Note:
When we added extra features (sentiment score, word count etc) they did not improve the results. Instead, they generally reduced accuracy. This likely happened because the models were already learning enough from the text itself and the extra features added noise rather than useful information

6.1.4 Random Forests

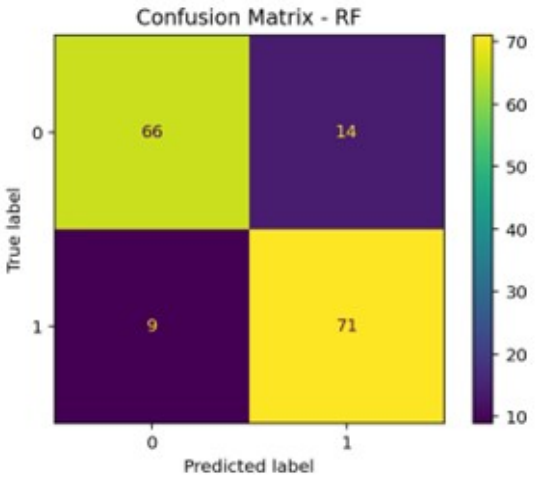
Like in the case of classification trees, we experimented with different vectorizers, feature counts, and n-gram configurations evaluating each combination using 5-fold stratified cross-validation accuracy and focusing on the top 5 performers. For random forests, increasing the number of features (from 500 up to 2000) consistently improved performance, especially when using CountVectorizer. Unlike a single decision tree, a random forest can handle more features and thus capture more complex patterns. While unigrams performed slightly better during cross-validation the gap was again marginal to unigrams+bigrams. Therefore, we evaluated both options during final model selection. Including bigrams ultimately improved performance on the test set as random forests discovered additional information provided by word pairs.

Overall, random forests showed a significant performance improvement compared to a single decision tree (by approximately 10–15%) demonstrating better generalization on unseen data due to averaging over multiple trees and random feature selection at each split.

The best-performing random forest configuration used CountVectorizer with a combination of unigrams and bigrams and 2000 features having also the following configuration: max_depth = None, min_samples_split = 10, min_samples_leaf = 2, max_features = "log2", n_estimators = 200 and a ccp_alpha = 0.001. Its performance metrics on the test set were: Accuracy = 85,6%, Precision = 83,53%, Recall = 88,75%, F1-score = 86,1% which are also depicted in the right confusion matrix shown below. At the right figure we can also see how for the same model configuration leads to considerably lower accuracy if we only consider unigrams.



(a) Random Forest - Unigrams



(b) Random Forest - Unigrams + Bigrams

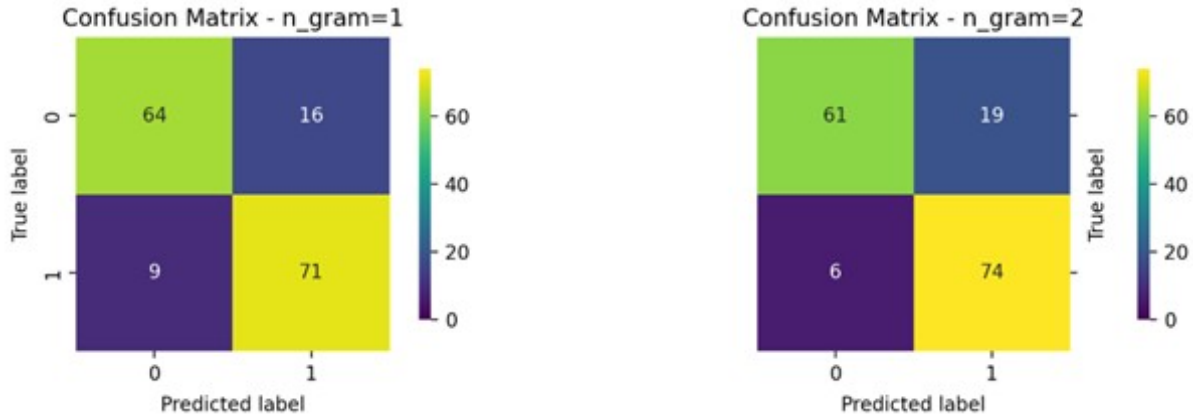
The addition of additional engineered featured rather decreased accuracy as was the case with classification trees.

6.1.5 Gradient Boosting

Training runs were conducted with and without bigrams, holding the feature cap at one thousand and comparing TF-IDF for unigrams against unigrams plus bigrams. On the validation folds, the best unigram configuration reached a mean cross-validated accuracy of 85.16% with a standard deviation of 2.84%. The best uni+bi configuration performed slightly better on average, achieving a mean cross-validated accuracy of 85.78% with a standard deviation of 3.37%.

On the held-out test split, the selected unigram model obtained an accuracy of 84.38%, with precision 0.8161, recall 0.8875, and F1-score 0.8503. The highest test accuracy observed for the unigram test was 88.28%. For the uni+bi testing, which used CountVectorizer and one thousand features. The selected configuration also produced 84.38% test accuracy, with precision 0.7957, recall 0.9250, and F1-score 0.8555. The uni+bi family delivered the strongest peak test result overall, with a best observed test accuracy of 92.19%.

During hyperparameter tuning, the cross-validated the best result used logistic loss with 192 estimators, a learning rate of 0.18625, maximum depth 3, subsample 0.82121, minimum samples per leaf 2, minimum samples per split 4. Taken together, these results indicate that adding bigrams modestly improves average validation accuracy and clearly boosts peak test performance, primarily through higher recall, while TF-IDF unigrams remain competitive and slightly more stable across folds.



6.2 Model Comparisons and Statistical Tests

A statistical significance test checks whether differences in performance across models are real or just due to chance. We use the Wilcoxon signed-rank test, as implemented in scipy.stats library [6], because it works with small sample sizes and does not assume a normal distribution of differences. It ranks the differences in accuracy between two models across folds and tests whether positive and negative differences are balanced. We will apply it to the cross-validation accuracies of the best models for pairs of different algorithms to confirm that observed performance differences are meaningful.

The performance of tree-based models was evaluated against the Multinomial Naive Based and regularized logistic regression classifiers using 5-fold cross validation accuracies. The decision tree model achieved mean accuracy of approximately 71% on the 5 splits, performing significantly lower than both MNB (89%) and logistic regression (79%). Despite these results, the p-value of 0.0625 for both classifiers indicates that their better performance against the decision tree was almost at the commonly used statistical significance threshold of 0.05.

The random forest model obtained a higher mean accuracy of 81%, showing better generalization than the single decision tree but still trailing behind MNB. Its comparison with LogReg resulted in a p-value of 1.0, suggesting statistically indistinguishable performance, whereas the difference with MNB (p=0.0625) again approached but did not meet the significance threshold.

Finally, the gradient boosting classifier achieved a mean accuracy of 82% outperforming the decision tree and slightly exceeding random forest. It displayed marginally non-significant differences with the MNB and decision tree models (p=0.0625) while its comparison with the regression model (p=0.31) and random forest (p=0.19) demonstrated no significant difference.

Overall, while Naive Bayes consistently produced greater accuracy, none of the pairwise comparisons achieved statistical significance above the confidence level threshold of 95%, but approached significance at approximately 93.8% on all its comparisons against tree based models. This suggests a mild tendency for MNB to perform better, but the evidence is insufficient to draw a definitive conclusion.

7 Conclusion

From our experiments both Multinomial Naive Bayes (MNB) and L1-regularized Logistic Regression performed strongly. Their results were close but Naive Bayes reached slightly higher accuracy (up to 87.5%) and was more stable across folds while Logistic Regression achieved similar performance with fewer features thanks to regularization. Analysis of the Logistic Regression coefficients showed that deceptive reviews tended to include more expressive or evaluative terms such as "finally" or "seemed", whereas truthful reviews relied on more concrete, situational words like "business" or "bed," reflecting a linguistic distinction between emotional description and factual reporting.

Tree-based models showed different behavior. A single Decision Tree performed poorly (69% accuracy) due to overfitting and difficulty handling sparse text features. However, when using ensemble methods, performance improved significantly. Random Forests reached around 85–86% accuracy and Gradient Boosting achieved the highest performance overall, with test accuracy up to 92%. These results show that ensemble tree models can match or outperform linear models on this task.

Regarding n-gram range, adding bigrams did not consistently improve performance. For Naive Bayes and Logistic Regression unigrams worked best. For Decision Trees bigrams hurt performance because of sparsity. However, Random Forests and Gradient Boosting handled bigrams better and gained small improvements in some settings. Overall, unigrams were usually enough and bigrams helped only more complex models. Finally, Wilcoxon significance testing showed that differences between models were not statistically significant at the 95% level although Naive Bayes and Gradient Boosting showed an advantage over the others. This suggests that while some models performed better in practice we cannot claim strong statistical evidence probably due to the limited dataset size.

8 Gen AI Contribution

-Exploring different vectorization techniques.
-Studying and understanding the different statistical significance testing methods.
-Help with coding and algorithm implementation. Set up vectorizers and n-grams.
-How to combine the bag-of-words matrix with extracted lexical features from the text such as unique word ratio, average word and sentence length, verb ratio etc.
-Brainstorm ways of applying a custom alpha smoothing array in multiNB instead of using a uniform value.
-Grammar and Vocabulary corrections on the report.

9 References

References

1. NL Times. (2025, September 24). *Scammers blackmailing Dutch businesses with fake Google reviews*. https://nltimes.nl/2025/09/24/scammers-blackmailing-dutch-businesses- fake-google-reviews

2. Ott, M., Cardie, C., & Hancock, J. T. (2013). Negative deceptive opinion spam. *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies* (pp. 497–501). Association for Computational Linguistics. Retrieved from `https://aclanthology.org/N13-1053/`

3. Ott, M., Choi, Y., Cardie, C., & Hancock, J. T. (2011). Finding deceptive opinion spam by any stretch of the imagination. *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies* (pp. 309–319). Association for Computational Linguistics. Retrieved from `https://aclanthology.org/P11-1032/`

4. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., & Duchesnay, É. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12, 2825–2830.

5. Manning, C. D., Raghavan, P., & Schütze, H. (2008). *Introduction to Information Retrieval*. Cambridge University Press.

6. SciPy Documentation. (n.d.). *Wilcoxon signed-rank test*. Retrieved from `https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.wilcoxon.html`