Comprehensive Report

on

Classifying Sports and Politics Articles

Using different Machine Learning techniques

1. Introduction

Sorting text into categories automatically sits at the heart of many real-world systems we use every day, from filtering out spam emails to routing customer complaints to the right department. In the field of Natural Language Processing, this challenge is known as text classification.

The work described in this report tackles a focused version of the problem: given a short news article, the models must decide whether it belongs to the sports category or the politics category. What makes this project distinctive is that every major component, the feature extractor, the classifiers, and even the training pipeline was coded from the scratch in Python rather than using pre built library. The only external library used for the core workflow was NumPy for efficient numerical computation; scikit-learn was brought in only for splitting data and printing evaluation metrics.

Three separate classifiers were built on the same data: a Logistic Regression model trained with gradient descent, a Multinomial Naive Bayes model uses Bayesian probability theory, and a Decision Tree that learns recursive axis-aligned splits. Each of these algorithms uses a different way of dividing the feature space, and comparing them side by side on which ideas work best for this particular kind of problem.

The rest of this report walks through the full pipeline in the order it was carried out. It begins with how the data was obtained and loaded and preprocessing, explains each algorithm in accessible terms, then presents the actual numeric results that each model produced, and finally discusses where the system falls short and what could be done about it.

2. How the Data Was Collected

2.1 Source of the Dataset

The raw material for this project is a CSV file called **sports_politics_dataset.csv**. Datasets are taken from BBC News collection. Each row in the file pairs a block of article text with a human assigned label: either “sport” or “politics”. Having the data already packaged as a flat CSV is convenient because it slots straight into a Pandas DataFrame without any intermediate parsing or format conversion. The two essential columns: “text” for the raw content and “category” for the label.

2.2 Loading the Data Safely

Rather than loading the file with **pd.read_csv()** call, the code wraps the entire loading step in a try-except block backed by **Python's logging module.** If the file is present and well-formed, an INFO level message confirms success and the first two rows are printed in a markdown table so the developer can see the schema. If anything goes wrong: say the file is missing or the encoding is garbled, the exception is caught and a descriptive failure message is logged, which stops the pipeline early instead of letting it crash somewhere downstream with a confusing error.

This kind of defensive coding is good practice in any data science project, because datasets are rarely as clean or reliable as one might hope.

3. Understanding the Dataset

3.1 Schema Overview

After loading and initial preprocessing, the working DataFrame contains three columns:

Column	Type	What It Holds
text	string	The full article content, lowercased during cleaning
tokens	list of strings	Individual words produced by splitting the text on whitespace
category	string	The actual label: either "sport" or "politics"

3.2 Cleaning and Quality Checks

Before any modelling takes place, a dedicated quality-check function inspects the DataFrame for problems. It first counts every null cell across all columns. If any are found, the affected rows are dropped and the new shape is logged so there is a clear audit trail. In our case the dataset turned out to be clean, with no missing values at all, which is a good sign of careful curation at the source.

Next, the entire text column is folded to lowercase. This is a small but important step: without it, the model would treat “Government” and “government” as two completely separate words, which would fragment the vocabulary and dilute the signal that each word carries.

3.3 Exploring What the Data Looks Like

Class balance. A quick value_counts() on the category column shows how many articles fall into each category. If one class massively outnumbered the other, we would need to worry about the models developing a bias towards the majority class. Strategies like oversampling the minority class or penalising misclassifications asymmetrically would then come into play.

Overall word frequencies. A corpus-wide bag of words was assembled by flattening every document’s token list into a single counter. Unsurprisingly, generic English function words: “the”, “of”, “and”, “to”,etc caught at the top. These words appear in virtually every document regardless of topic, so they carry almost no discriminating power on their own.

Per-category word frequencies. More interesting patterns emerge when word counts are computed separately for sports and politics. On the sports side, words such as “game”, “team”, “player”, “season” and “win” dominate. In politics, “government”, “election”, “president”, “policy” and “party” rise to the top. The clear separation between these two vocabularies is an encouraging sign therefore, simple feature-weighting schemes like TF-IDF should be able to pick up on the differences.

4. Turning Raw Text into Numbers

4.1 Tokenization

The first concrete step in the feature-engineering pipeline is breaking each article into individual words. This is handled by a small Text_cleaner class whose cleaner_1() method calls str.split() on every row, producing a list of tokens stored in a new column. Because str.split() breaks on any whitespace, it is fast and covers most cases, but it does leave punctuation attached to words, so “election,” and “election” end up as different tokens. We will revisit this trade-off in the limitations section.

4.2 TF-IDF: Weighing Words by Importance

Raw word counts are a poor feature representation because they treat every word as equally informative. TF-IDF(Term Frequency times Inverse Document Frequency): fixes this by multiplying two signals together.

The Term Frequency part measures how often a word appears in a single document, normalised by that document’s length. A word that appears five times in a ten-word document gets a TF of 0.5, whereas the same five appearances in a hundred-word document give a TF of only 0.05. This normalisation keeps long and short documents on an even footing.

The Inverse Document Frequency part captures how rare or common a word is across the entire collection. Words that appear in nearly every document: think “the” or “is”, receive a low IDF because they are not distinctive. Words that appear in only a handful of documents get a high IDF, boosting their contribution. The specific formula used here adds smoothing to both numerator and denominator: **IDF(t) = log((N + 1) / (DF(t) + 1)) + 1**, which prevents division-by-zero errors and stops the weights from blowing up for extremely rare terms.

Once every document has been converted to a vector of TF-IDF scores, L2 normalization is applied so that each vector has a length of exactly one. This step is critical because it prevents documents with larger vocabularies from dominating the similarity calculations purely due to having more non-zero entries.

The vocabulary itself is assembled during the **fit()** stage by sweeping through all training documents, collecting every unique word, and sorting them alphabetically to create a stable word-to-index mapping. The **transform()** method then uses this mapping to convert any set of documents: training or unseen, into a dense matrix where each row is a document and each column corresponds to a word in the vocabulary.

5. How Each Classifier Works

With the text converted to numerical vectors, the next job is to train a model that can look at a new vector and assign it to the correct category. Three very different algorithms were implemented from scratch for this purpose.

5.1 Logistic Regression

At its core, Logistic Regression is a linear model with a twist. It computes a weighted sum of the input features, adds a bias term, and then feeds the result through the sigmoid function, which squashes any real number into the range (0, 1). The output can be interpreted as the model's estimated probability that the input belongs to class 1 (sports, in our encoding). If that probability exceeds 0.5 the article is labelled sports; otherwise it is labelled politics.

Learning happens through gradient descent. Starting from zero-initialised weights, the algorithm repeatedly computes the gap between its current predictions and the true labels, then nudges every weight in the direction that shrinks that gap. The size of each nudge is controlled by the learning rate, set to 0.1 here, and the process is repeated for 500 complete passes over the training data. A practical detail worth noting is that the sigmoid input is clipped to the interval [−500, 500] before exponentiation to prevent numerical overflow, a common safeguard in hand-rolled implementations.

The implementation also includes a One-vs-Rest extension for multiclass settings, although only the binary path was performed in this project.

Observed Performance

After training, the Logistic Regression model was evaluated on 307 held-out test samples. The full classification report is shown below:

	Precision	Recall	F1-Score	Support
Class 0 (Politics)	1.00	0.81	0.89	135
Class 1 (Sports)	0.87	1.00	0.93	172

Accuracy			0.92	307
Macro Avg	0.93	0.90	0.91	307
Weighted Avg	0.93	0.92	0.91	307

The model achieved an overall accuracy of 92%. A closer look at the per-class numbers reveals an interesting asymmetry: it was extremely precise when predicting politics (precision of 1.00), meaning that whenever it said an article was about politics it was almost always right. However, its recall for politics was 0.81, which tells us it missed about one in five politics articles, misclassifying them as sports. On the sports side the pattern is flipped: recall is a perfect 1.00, so every genuine sports article was caught, but precision drops to 0.87 because some politics articles were incorrectly swept into the sports category.

This kind of precision-recall trade-off is very common in linear classifiers and often comes down to where the decision boundary happens to fall in the feature space. With more training iterations, a tuned learning rate schedule, or regularization, this gap could likely be narrowed.

5.2 Multinomial Naive Bayes

Naive Bayes takes a completely different approach. Instead of learning weights through optimization, it uses probability theory directly. Using Bayes' theorem, the model asks: given the words I see in this document, what is the probability that it came from the sports pile versus the politics pile?

The "naive" part of the name refers to an assumption: that every word in a document is statistically independent of every other word, given the category. In reality this is obviously false ("world" and "cup" tend to travel together), but decades of empirical evidence show that the model works surprisingly well despite this unrealistic assumption.

During training, the model computes two things: the prior probability of each class (how common it is in the training set) and the conditional likelihood of every word given each class (essentially, the word-frequency profile of each category). Both quantities are stored as logarithms so that multiplication can be replaced with faster and more numerically stable addition. Laplace smoothing with alpha equal to 1.0 is applied to the word counts so that a word never seen during training does not blow up the calculation with a zero probability.

At prediction time, the model simply computes the log-posterior for each class: the log-prior plus the dot product of the document vector with the class's log-likelihood vector, and picks the class with the higher score.

Observed Performance

The Multinomial Naive Bayes classifier was evaluated on the same 307 - sample test set. Its results were remarkable:

	Precision	Recall	F1-Score	Support
Class 0 (Politics)	1.00	0.99	1.00	135
Class 1 (Sports)	0.99	1.00	1.00	172
Accuracy			1.00	307
Macro Avg	1.00	1.00	1.00	307
Weighted Avg	1.00	1.00	1.00	307

The model hit a near-perfect accuracy of 100% on the test set, with precision, recall, and F1 scores all rounding to 1.00 for both classes. Out of the entire test partition, only a single sample appears to have been placed on the wrong side of the boundary (the 0.99 recall for politics and 0.99 precision for sports point to one or at most two misclassifications).

At first glance this may seem too good to be true, but there is a reasonable explanation. Sports and politics occupy very different corners of the English vocabulary, and the probabilistic word-frequency profiles that Naive Bayes learns can capture this separation almost perfectly when the training data is representative. The model's simplicity: no iterative optimization, no hyperplane to position can actually work in its favour on clean, well-separated data because there are fewer moving parts that could go wrong.

That said, such a result also calls for healthy scepticism. On a harder task with overlapping topics: say, distinguishing political commentary from sports editorials that discuss public policy: the same model would almost certainly not score this high.

5.3 Decision Tree

A Decision Tree learns by recursively splitting the data into smaller groups, choosing at each step the single feature and threshold value that best separate the classes. The quality of a split is measured by the Gini impurity, a metric that is zero when a group is perfectly pure (all one class) and reaches its maximum when the classes are equally mixed.

Building the tree starts at the root, which contains all training examples. The algorithm looks at a random subset of features (roughly the square root of the total vocabulary size, for efficiency), evaluates up to twenty candidate thresholds per feature using evenly spaced percentiles, and picks the combination that produces the lowest weighted Gini impurity across the two resulting child nodes. It then recurses on each child. Growth stops when a node is perfectly pure, when the maximum depth of 15 is reached, or when a node holds fewer than 5 samples.

Leaf nodes simply store the majority class among the training examples that ended up there, and prediction involves walking a new sample down the tree from root to leaf, taking the appropriate branch at every internal node.

Observed Performance

The Decision Tree was evaluated on the same 307- sample test set:

	Precision	Recall	F1-Score	Support
Class 0 (Politics)	0.84	0.76	0.79	135
Class 1 (Sports)	0.82	0.88	0.85	172
Accuracy			0.83	307
Macro Avg	0.83	0.82	0.82	307
Weighted Avg	0.83	0.83	0.83	307

With an overall accuracy of 83%, the Decision Tree trails both the Logistic Regression and Naive Bayes models by a meaningful margin. Looking at the per-class figures, recall for politics (class 0) is only 0.76, meaning that roughly one in four politics articles was misclassified as sports. Sports articles fared somewhat better with a recall of 0.88, but precision sat at 0.82 because of the false positives leaking in from the politics side.

This outcome is not surprising. A Decision Tree makes axis-aligned splits: each split can test only one feature at a time, so it has to build a staircase of many small cuts to approximate the smooth boundary that a linear model draws in a single step. In a high-dimensional TF-IDF space where thousands of features jointly contribute to the distinction between classes, this is an inherently less efficient strategy. The random feature subsampling that speeds up training also introduces noise, and a single tree has no mechanism to average away that noise the way an ensemble (like a Random Forest) would.

5.4 Tying It All Together: The Pipeline

To keep the code organised and to prevent subtle bugs such as fitting the TF-IDF vocabulary on test data, a lightweight PipelineScratch class chains the vectoriser and the classifier into a single object. Calling fit() on the pipeline first fits and transforms the TF-IDF step, then trains the classifier on the resulting matrix. Calling predict() applies the same TF-IDF transform (without refitting) and then runs the classifier. This mirrors the design philosophy of scikit-learn's Pipeline and guarantees that training and inference preprocessing are always in sync.

The data was divided into a 67 % training set and a 33 % test set using a fixed random seed of 101, ensuring that every model saw exactly the same split and that results are reproducible across runs. The target variable was encoded numerically: 1 for sports and 0 for politics.

6. Putting the Numbers Side by Side

6.1 The Metrics Used

Four standard classification metrics were computed for every model:

- **Precision**: Of all the times the model said "this is class X," how often was it actually class X? A high precision means few false positives.
- **Recall**: Of all the samples that truly are class X, how many did the model manage to find? A high recall means few false negatives.

- **F1-Score**: The harmonic mean of precision and recall. It provides a single number that balances both concerns and is especially useful when classes are unequal in size.
- **Accuracy:** The simplest metric: the share of all predictions that turned out to be correct.

6.2 Head-to-Head Comparison

The table below condenses the key numbers from all three models into a single view for easy comparison:

Model	Precision (wt)	Recall (wt)	F1 (wt)	Accuracy
Logistic Regression	0.93	0.92	0.91	92%
Multinomial NB	1.00	1.00	1.00	100%
Decision Tree	0.83	0.83	0.83	83%

6.3 What the Numbers Tell Us

The standout result is the dominance of Multinomial Naive Bayes, which achieved virtually perfect classification with an accuracy of 100%. This may seem counterintuitive because Naive Bayes rests on the clearly unrealistic assumption that words occur independently of one another. However, for a well-separated binary task like sports versus politics, the word-frequency distributions of the two classes are so different that even an approximate probabilistic model can draw a nearly flawless boundary. The takeaway is that algorithm sophistication does not always win, sometimes a simple model that matches the structure of the data outperforms more complex alternatives.

Logistic Regression came in second at 92% accuracy. It is a capable model, but its gradient-descent training with a fixed learning rate and no regularization left room for improvement. The asymmetry in its per-class scores: perfect recall for sports but only 81% recall for politics, suggests that the learned decision boundary is slightly skewed towards predicting sports,

possibly because the sports articles formed a slightly larger share of the training data or because the optimizer had not fully converged by iteration 500.

The Decision Tree brought up the rear at 83%. This is a respectable score in absolute terms, but it lags well behind the other two models. The fundamental problem is that a tree can only examine one feature per split, so it needs a deep and complex structure to approximate the kind of holistic feature combination that Logistic Regression and Naive Bayes achieve naturally. Add in the randomised feature selection and threshold sampling, and it is clear why a single tree struggles here.

6.4 Hyperparameter Summary

For completeness, the following table lists every tuneable setting used in the experiments:

Component	Parameter	Value Used
Logistic Regression	Learning rate	0.1
Logistic Regression	Training iterations	500
Multinomial NB	Laplace smoothing (α)	1.0
Decision Tree	Maximum tree depth	15
Decision Tree	Minimum samples to split a node	5
TF-IDF Vectoriser	IDF formula	log((N+1)/(DF+1)) + 1
TF-IDF Vectoriser	Vector normalisation	L2 (unit length)
Data Split	Test set proportion	33%
Data Split	Random seed	101

7. Where the System Falls Short

No system is without its blind spots. Below are areas where the current implementation could be improved.

7.1 Rough Edges in Preprocessing

- Punctuation is not stripped. Because tokenization is done by splitting on whitespace alone, a comma or full stop at the end of a word creates a separate vocabulary entry. This artificially inflates the feature space and means the model treats “government” and “government.” as unrelated tokens.
- Stop words are kept in the mix. Although TF-IDF’s IDF component naturally dampens the influence of very common words, explicitly removing a standard stop-word list would shrink the vocabulary, speed up computation, and reduce noise.
- No stemming or lemmatisation is applied. Words like “running,” “runs,” and “ran” occupy three separate slots in the vocabulary when they really express the same concept. Collapsing them to a shared root would concentrate their statistical signal.
- Only single words (unigrams) are considered. Meaningful phrases like “prime minister” or “world cup” are split into their individual parts, losing the context that makes them distinctive. Adding bigram or trigram features would address this.

7.2 Weaknesses in the Models Themselves

- The Logistic Regression model uses a fixed learning rate with no decay schedule, momentum, or adaptive method like Adam. This means convergence may be slower or less stable than it needs to be. Furthermore, no regularisation penalty (L1 or L2) is applied to the weights, which leaves the model vulnerable to overfitting on a high-dimensional feature space.
- Naive Bayes assumes every feature is independent of every other feature given the class, which is obviously violated in natural language. Pairs of words that routinely co-occur: “election” with “campaign” “goal” with “scored” are treated as if they carry no information about each other. Despite delivering strong results here, the assumption limits the model’s ceiling on harder tasks.
- The Decision Tree is a single tree with no ensemble. A single tree is inherently noisy and prone to overfitting, especially on high-dimensional data. Wrapping it in a Random Forest or Gradient Boosting framework would average out the variance and yield significantly better generalisation.
- Because the Decision Tree randomly samples a subset of features at each split, two runs on the same data can produce different trees and therefore different

accuracy numbers. There is no mechanism in the current code to control this randomness with a seed at the tree level.

7.3 Broader Scalability and Evaluation Gaps

- TF-IDF vectors are stored as dense NumPy arrays. In practice, these vectors are extremely sparse (most entries are zero), so a dense representation wastes large amounts of memory. Using scipy's sparse matrix format would cut memory usage dramatically and also speed up the matrix operations.
- Model performance is estimated from a single random train-test split. A single split can be lucky or unlucky; k-fold cross-validation would give a more trustworthy picture of how well each model generalizes.
- All hyperparameters (learning rate, number of iterations, tree depth, smoothing alpha) were set by hand without any systematic search. Grid search or Bayesian optimization could discover meaningfully better configurations.
- The entire feature representation is lexical: it knows which words appear and how often, but nothing about what those words mean. Richer representations based on pre-trained word embeddings (Word2Vec, GloVe) or contextual language models (BERT, RoBERTa) would capture semantic relationships that TF-IDF simply cannot.

8. Wrapping Up and Looking Ahead

This report has walked through a complete text classification system: from loading a CSV file to printing final evaluation metrics with every major component written from scratch. Three classifiers were tested on the task of sorting news articles into sports and politics, and the results tell a clear story.

Multinomial Naive Bayes stood out as the top performer with near-perfect accuracy of 100%, proving that a straightforward probabilistic model can be devastatingly effective when the classes are well separated in vocabulary space. Logistic Regression followed at 92%, a solid result that would likely improve with regularization and a smarter optimizer. The Decision Tree came in at 83% is respectable, but limited by its one-feature at-a-time splitting strategy and the lack of any ensemble.

Beyond the raw numbers, the exercise of implementing these algorithms by hand offers something no library call can: a genuine, ground-level understanding of what happens inside the black box. Coding the sigmoid function, writing the gradient update loop, computing Bayesian posteriors from scratch, and building a recursive tree structure forces the practitioner

to confront every design choice and numerical pitfall that production-grade libraries abstract away.

8.1 Suggested Next Steps

1. Add a proper text-cleaning pipeline that strips punctuation, removes stop words, and applies stemming or lemmatisation to consolidate related word forms.
2. Extend the TF-IDF vectoriser to support bigrams and trigrams, capturing multi-word concepts that single tokens miss.
3. Introduce L2 regularisation into the Logistic Regression training loop to discourage overfitting and improve generalization.
4. Upgrade the Decision Tree into a Random Forest by training many trees on bootstrapped subsets of the data and averaging their predictions.
5. Switch from dense NumPy arrays to sparse matrices for the TF-IDF representation, making the system practical for much larger vocabularies.
6. Replace the single train-test split with k-fold cross-validation (k = 5 or 10) to get more reliable estimates of model quality.
7. Experiment with pre-trained word embeddings or transformer-based features to move from purely lexical to semantic representations.
8. Add visual diagnostics such as confusion matrices, ROC curves, and precision-recall curves to make the evaluation richer and more actionable.