grafik, ekran görüntüsü, daire, logo içeren bir resim

Yapay zeka tarafından oluşturulan içerik yanlış olabilir.

**DATA MINING & KNOWLEDGE DISCOVERY**

**Data Authorship Classification Project**

**Instructor: Assoc. Prof. Dr. AYSUN GÜRAN**

202203001070 EYÜPHAN YETİMOĞLU

202203001035 KEREM AKKALE

202203001075 JOHN MERT BACANLI

202203001094 ARDA TEZBAŞARAN

202203001017 EDİZ SEVİM

**LIST OF CONTENT**

[1. Introduction 4](#_Toc197436804)

[2. Dataset & Preprocessing 5](#_Toc197436805)

[2.1 Dataset Part & Converting into CSV 5](#_Toc197436806)

[2.2 Code Breakdown 6](#_Toc197436807)

[2.3 Preprocessing 8](#_Toc197436808)

[**2.3.1** **Label Encoding:** 8](#_Toc197436809)

[**2.3.2** **Train-Test Split:** 8](#_Toc197436810)

[**2.3.3** **Whitespace Normalization & TF-IDF Filtering:** 9](#_Toc197436811)

[3. Feature Extraction Techniques 10](#_Toc197436812)

[3.1 TF-IDF Based Features 11](#_Toc197436813)

[**3.1.1** **Implemented N-gram Variants:** 11](#_Toc197436814)

[**3.1.2** **TF-IDF Vectorizer Construction:** 12](#_Toc197436815)

[3.2 BERT Embeddings 15](#_Toc197436816)

[**3.2.1** **Methodology & Implementation** 15](#_Toc197436817)

[4. Machine Learning Models 17](#_Toc197436818)

[4.1 Classifier Selection 18](#_Toc197436819)

[4.2 Training and Evaluation with TF-IDF Features 19](#_Toc197436820)

[5. Experimental Setup and Evaluation 21](#_Toc197436821)

[5.1 Dataset Splitting 21](#_Toc197436822)

[5.2 Feature Extraction Configuration 22](#_Toc197436823)

[**5.2.1** **TF-IDF (Term Frequency–Inverse Document Frequency) with N-grams** 22](#_Toc197436824)

[**5.2.2** **BERT-Based Embeddings (dbmdz/bert-base-turkish-cased)** 24](#_Toc197436825)

[5.3 Model Training and Caching 25](#_Toc197436826)

[5.4 Machine Learning Models 27](#_Toc197436827)

[5.5 Evaluation Metrics 28](#_Toc197436828)

[5.6 Performance Logging and Reporting 29](#_Toc197436829)

[6. Results and Performance Comparison 29](#_Toc197436830)

[6.1 TF-IDF Unigram 29](#_Toc197436831)

[6.2 TF-IDF Word 2-gram 30](#_Toc197436832)

[6.3 TF-IDF Word 3-gram 31](#_Toc197436833)

[6.4 TF-IDF Char 2-gram 32](#_Toc197436834)

[6.5 TF-IDF Char 3-gram 33](#_Toc197436835)

[6.6 BERT 34](#_Toc197436836)

[7. Conclusion 34](#_Toc197436837)

[Features 36](#_Toc197436838)

**LIST OF FIGURES**

[Figure 2.1 5](#_Toc197385101)

[Figure 2.2 5](#_Toc197385102)

[Figure 2.3 5](#_Toc197385103)

[Figure 2.4 6](#_Toc197385104)

[Figure 2.5 6](#_Toc197385105)

[Figure 2.6 7](#_Toc197385106)

[Figure 2.7 7](#_Toc197385107)

[Figure 2.8 8](#_Toc197385108)

[Figure 2.9 9](#_Toc197385109)

[Figure 3.1 10](#_Toc197385110)

[Figure 3.2 11](#_Toc197385111)

[Figure 3.3 14](#_Toc197385112)

[Figure 3.4 14](#_Toc197385113)

[Figure 3.5 14](#_Toc197385114)

[Figure 3.6 15](#_Toc197385115)

[Figure 3.7 15](#_Toc197385116)

[Figure 3.8 15](#_Toc197385117)

[Figure 3.9 16](#_Toc197385118)

[Figure 4.1 17](#_Toc197385119)

[Figure 4.2 18](#_Toc197385120)

[Figure 4.3 19](#_Toc197385121)

[Figure 4.4 19](#_Toc197385122)

[Figure 5.1 20](#_Toc197385123)

[Figure 5.2 21](#_Toc197385124)

[Figure 5.3 22](#_Toc197385125)

[Figure 5.4 22](#_Toc197385126)

[Figure 5.5 23](#_Toc197385127)

[Figure 5.6 23](#_Toc197385128)

[Figure 5.7 24](#_Toc197385129)

[Figure 5.8 25](#_Toc197385130)

[Figure 5.9 26](#_Toc197385131)

[Figure 5.10 27](#_Toc197385132)

[Figure 5.11 27](#_Toc197385133)

[Figure 5.12 28](#_Toc197385134)

[Figure 5.13 28](#_Toc197385135)

[Figure 6.1 28](#_Toc197385136)

[Figure 6.2 29](#_Toc197385137)

[Figure 6.3 29](#_Toc197385138)

[Figure 6.4 30](#_Toc197385139)

[Figure 6.5 31](#_Toc197385140)

[Figure 6.6 32](#_Toc197385141)

**Abstract**

Authorship attribution is a critical task in natural language processing. In this study, we present a comparative analysis of multiple machine learning models for authorship classification using a custom text dataset. The text files were first converted into a structured CSV format, followed by preprocessing and label encoding. We employed various feature extraction techniques, including n-gram based word and character TF-IDF extraction, as well as contextual embeddings obtained from the “Turkish” BERT model. Six classifiers—Random Forest, Support Vector Machine(SVM), XGBoost, Naive Bayes(GaussianNB), Multilayer Perceptron(MLP), and Decision Tree—were trained and evaluated on these features. Experimental results demonstrate the performance impact of different feature types on classification accuracy, precision, recall, and F1-score. The findings highlight the superiority of BERT-based embeddings in capturing deeper contextual representations of text for authorship detection. This project offers a comprehensive framework for evaluating traditional and deep learning-based methods in author classification task.

# Introduction

Authorship attribution, also known as authorship classification, is a significant subfield of natural language processing (NLP) that focuses on identifying the author of a given text based on stylistic and linguistic features. This problem is crucial in various domains such as literary analysis, digital forensics, plagiarism detection, and security-sensitive communications, where determining the origin of text can provide valuable insights.

Traditional authorship attribution approaches rely on handcrafted features such as word frequency, sentence length, and syntactic structures. However, with the advancement of machine learning and deep learning techniques, it has become possible to extract richer and more abstract textual features that improve classification performance. This project aims to develop a comprehensive authorship attribution system specifically designed for Turkish texts

In this project, we investigate the effectiveness of different feature extraction techniques including TF-IDF with word and character n-grams, and contextual embeddings generated from a Turkish BERT model. We evaluate the performance of six classification algorithms—Random Forest, Support Vector Machine (SVM), XGBoost, Naive Bayes, Multilayer Perceptron (MLP), and Decision Tree—using a balanced dataset of Turkish texts from multiple authors.

The project involves with several key steps: converting raw .txt files into structured CSV format, performing preprocessing and label encoding, extracting both statistical and contextual features, training various machine learning models, and finally evaluating them using standard classification metrics such as accuracy, precision, recall, and F1-score.

By comparing classical machine learning models with deep learning-based embedding approaches, this project provides a valuable framework for understanding the trade-offs between traditional and modern NLP techniques in the context of authorship classification.

# Dataset & Preprocessing

## **Dataset Part & Converting into CSV**

The dataset utilized in this project consists of Turkish literary texts authored by multiple individuals. The files were initially stored in a hierarchical folder structure, with each author's works placed inside a separate folder (dataset\_authorship/author\_name/\*.txt). This raw structure was first converted into a structured CSV format where each row contains a text sample and its associated author label:A label encoder was applied to convert categorical author names into numerical labels for classification tasks.

metin, ekran görüntüsü içeren bir resim

Yapay zeka tarafından oluşturulan içerik yanlış olabilir.

Figure .

## **Code Breakdown**



Figure .

* This sets the path to the main directory that contains subfolders, one for each author.
* Each author's folder contains several .txt files with that author's writing samples.



Figure .

* Initializes an empty list called data. This list will store dictionaries, each representing a text file and its corresponding author.

metin, ekran görüntüsü, yazı tipi içeren bir resim

Yapay zeka tarafından oluşturulan içerik yanlış olabilir.

Figure .

* Loop through folders in main\_dir → **os.listdir(main\_dir)**
* Build folder path → **os.path.join(main\_dir, author\_folder)**
* Check if it's a folder → **os.path.isdir(author\_path)**
* Loop through files in the author folder → **os.listdir(author\_path)**
* Build file path → **os.path.join(author\_path, filename)**
* Check file extension → **filename.endswith('.txt')**
* Open file → **open(file\_path, 'r', encoding='utf-8')**
* Read and strip text → **file.read().strip()**
* Append to list → **data.append({...})**

metin, ekran görüntüsü, yazı tipi içeren bir resim

Yapay zeka tarafından oluşturulan içerik yanlış olabilir.

Figure .

* **Define the output file path** → **csv\_path = "data.csv"**  
   This defines the path where the CSV file will be saved. In this case, the file name is data.csv.
* **Check if the file already exists** → **os.path.exists(csv\_path)**  
   This checks if a file with the name data.csv already exists in the current directory. If it does, the code will print a message and won't overwrite the existing file.
* **Print message if the file exists** → **print("📂 Already exists.")**  
   If the file already exists, this line prints a message indicating the file is present.
* **Convert the data list into a DataFrame** → **df = pd.DataFrame(data)**  
   This converts the data list (which contains dictionaries) into a Pandas DataFrame. A DataFrame is a table-like structure that is easy to work with for data manipulation and export.
* **Save the DataFrame to a CSV file** → **df.to\_csv(csv\_path, index=False)**  
   This saves the DataFrame (df) into a CSV file at the path csv\_path. The index=False argument ensures that the DataFrame index (row numbers) is not included in the CSV file.
* **Print success message** → **print("✅ data.csv file has been created successfully.")**  
   If the file does not exist, it creates data.csv and prints a success message indicating that the file has been created.

## **Preprocessing**

Minimal preprocessing was done to preserve author-specific language usage, which is essential for authorship detection. Stopword removal and stemming were not applied, as stylistic elements and word usage frequency contribute to author differentiation.

### **Label Encoding:**

Author names, which are initially strings, are encoded into numeric labels using LabelEncoder() from scikit-learn to make them compatible with machine learning algorithms.



Figure .

* + 1. **Train-Test Split:**

The dataset is split into training and testing subsets with a ratio of 80:20 to enable fair evaluation of model performance.



Figure .

* **X**: This represents the features (input data) of the dataset. For example, if we're working with text data, X could be the vectorized text or features extracted from the text.
* **y**: This represents the target labels or the output variable that we want to predict. For instance, in an authorship classification task, y could be the author labels corresponding to the text in X.
* **test\_size=0.2**: This specifies the proportion of the data to be used for testing. 0.2 means **20%** of the data will be used for testing, and the remaining **80%** will be used for training.
* **random\_state=42**: This is a seed for the random number generator to ensure that the data split is reproducible. Setting a fixed value (e.g., 42) allows you to get the same split every time you run the code, which is useful for consistency and debugging.
* **X\_train**: The training data (features) for the model.
* **X\_test**: The testing data (features) for the model.
* **y\_train**: The training labels corresponding to X\_train.
* **y\_test**: The testing labels corresponding to X\_test.

### **Whitespace Normalization & TF-IDF Filtering:**

Although no separate preprocessing function was defined beyond basic whitespace cleaning (strip()), certain implicit preprocessing steps are embedded in the workflow:

* **Whitespace Normalization**: When reading each file, strip() is applied to remove leading and trailing spaces.



Figure .

* **TF-IDF Internal Filtering**: When generating TF-IDF features, parameters(We will expand this function and its parameter on later chapters) such as min\_df=4, max\_df=0.75, and sublinear\_tf=True act as internal preprocessing:

yazı tipi, metin, ekran görüntüsü, tasarım içeren bir resim

Yapay zeka tarafından oluşturulan içerik yanlış olabilir.

Figure .

* + **min\_df=4** removes rarely used words.
  + **max\_df=0.75** filters out very common words.
  + **sublinear\_tf**=True applies logarithmic scaling to term frequencies.

Since the goal is to capture **stylistic features** of authors, further preprocessing such as stopword removal, stemming, or lowercasing was **intentionally** skipped.

# Feature Extraction Techniques

In authorship classification, the performance of machine learning models highly depends on how well the textual data is represented numerically. Since raw text data cannot be directly processed by most machine learning algorithms, an essential step in the pipeline is converting textual documents into feature vectors — a process known as **feature extraction**.

This project employs a combination of traditional and modern feature extraction techniques to capture both shallow lexical patterns and deep contextual semantics from the texts. Specifically, we utilized:

* TF-IDF (Term Frequency-Inverse Document Frequency) representations for both word-based and character-based n-grams,
* BERT (Bidirectional Encoder Representations from Transformers) embeddings for capturing contextual information from sentences.

These techniques aim to capture not only the frequency and distribution of words or characters (stylometric features), but also the syntactic and semantic structures inherent in different authors' writing styles. The combination of these diverse features allows us to build more robust and accurate authorship classification models.

## **TF-IDF Based Features**

TF-IDF (Term Frequency–Inverse Document Frequency) is a widely used statistical method that reflects the importance of a term to a document relative to a collection (or corpus) of documents. It helps reduce the weight of commonly used words and emphasize more discriminative ones.

In this project, TF-IDF is used in combination with **n-gram analysis**, which considers sequences of words or characters to capture stylistic patterns.

### **Implemented N-gram Variants:**

As seen in the vector\_settings section of the code, the following vector configurations are used:

metin, ekran görüntüsü, yazı tipi içeren bir resim

Yapay zeka tarafından oluşturulan içerik yanlış olabilir.

Figure .

Each configuration captures different aspects:

* **Unigrams** capture individual word usage.
* **Word n-grams** model short sequences of words, useful for syntactic style.
* **Character n-grams** help detect spelling and morphological patterns that are unique to authors.

### **TF-IDF Vectorizer Construction:**

The get\_tfidf\_vectorizer() function constructs the vectorizer with key hyper-parameters: metin, ekran görüntüsü, yazı tipi içeren bir resim

Yapay zeka tarafından oluşturulan içerik yanlış olabilir.

Figure .

This function is designed to create and return a TF-IDF Vectorizer object, which is a core utility for transforming text data into numerical features using the TF-IDF (Term Frequency-Inverse Document Frequency) technique. The TF-IDF method helps highlight important words within the text, and it’s commonly used for text classification tasks.

Here’s a detailed breakdown of the parameters and settings used in the function:

* + - * **ngram\_range Parameter**
      * **Purpose**: This specifies the range of n-grams (sequences of n words or characters) that should be considered when vectorizing the text.
        + **Example**:

(1, 1) will only consider **unigrams** (single words).

(2, 2) will consider **bigrams** (pairs of words).

(3, 3) will consider **trigrams** (sequences of three words).

* + - * + In our case, different n-gram ranges are passed to the vectorizer, depending on the feature extraction setup, like unigrams, bigrams, and trigrams for both word- and character-based models.
* **2. analyzer Parameter**
* **Purpose**: Specifies whether the vectorizer will analyze **words** or **characters**.
  + **Options**:
    - 'word' — analyzes words as n-grams.
    - 'char' — analyzes characters as n-grams.
  + **Example**: In the function, this parameter is passed as 'word' by default, but when generating character-based n-grams, it will be set to 'char'.

.

* **min\_df Parameter**
* **Purpose**: Sets the minimum number of documents in which a term must appear to be considered as a feature. This helps eliminate rare words that do not contribute much to the analysis.
  + **Value**: min\_df=4 means that any word must appear in at least **4 documents** to be included as a feature.
  + **Impact**: Helps reduce noise by ignoring very rare terms that might not be useful in classification.
* **max\_df Parameter**
* **Purpose**: Specifies the maximum proportion of documents a term can appear in to be considered. Terms that appear too frequently across the dataset are often less discriminative and are thus excluded.
  + **Value**: max\_df=0.75 means that a term can appear in a maximum of **75% of the documents**.
  + **Impact**: Removes common words (e.g., "the", "is") that are likely stopwords and don’t help in distinguishing between authors.
* **sublinear\_tf Parameter**
* **Purpose**: When set to True, the term frequency is scaled using a logarithmic function rather than raw counts. This helps in downscaling the effect of frequently occurring terms.
  + **Value**: sublinear\_tf=True applies the following transformation:
    - tf(t, d) = 1 + log(tf(t, d)) if tf(t, d) > 0
    - This reduces the impact of high-frequency terms.
  + **Impact**: It prevents heavily frequent terms from dominating the feature space, making the model more sensitive to smaller but significant terms.
* **max\_features Parameter**
* **Purpose**: Limits the number of features (terms or n-grams) to be kept in the feature space. This is helpful for reducing dimensionality and improving computational efficiency.
  + **Value**: max\_features=30000 limits the number of terms or n-grams in the final feature vector to the top **30,000 most frequent features**.
  + **Impact**: By setting this, the vectorizer only keeps the most informative terms and ignores the less frequent ones, which can help improve model performance by focusing on the most significant features.
* **Function Return Value**
* **Output**: The function returns a TF-IDF Vectorizer object with the parameters specified. When this vectorizer is used on the training data (via .fit\_transform()), it will convert the raw text into a matrix of TF-IDF features, which can then be used as input for machine learning models.

## **BERT Embeddings**

In addition to basic text representation techniques like TF-IDF and n-grams, modern Natural Language Processing (NLP) methods use powerful models such as **BERT (Bidirectional Encoder Representations from Transformers)**. Unlike traditional methods that only consider word frequencies or patterns, BERT understands the **context** of each word by looking at the words around it, both before and after.

This makes BERT very effective for capturing the meaning and structure of a text. In our project, we use a pre-trained Turkish BERT model (dbmdz/bert-base-turkish-cased) to convert each text into a numeric vector (embedding) that reflects its overall meaning and style.

These BERT embeddings are then used as input to machine learning models for authorship classification. By using BERT, we aim to improve the system’s ability to detect subtle differences in writing style between different authors.

### **Methodology & Implementation**

To obtain semantic-rich features from text, we use a pre-trained BERT model specifically designed for Turkish:



Figure .

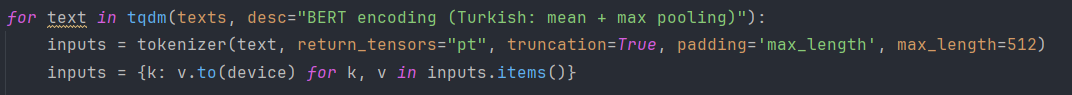
This model provides deep, context-aware embeddings. We first tokenize and encode the input texts using BERT’s tokenizer:

Figure .

Each input is passed to the BERT model to obtain its **last hidden state**, which contains the contextual embeddings for all tokens in the input sequence:

metin, ekran görüntüsü, yazı tipi içeren bir resim

Yapay zeka tarafından oluşturulan içerik yanlış olabilir.

Figure .

To convert these token-level embeddings into a single vector per document (i.e., fixed-size input for classifiers), we apply two pooling strategies:

* **Mean Pooling**: Calculates the average of token embeddings (ignoring padding tokens using the attention mask).
* **Max Pooling**: Selects the maximum value across each dimension.

metin, ekran görüntüsü, yazı tipi içeren bir resim

Yapay zeka tarafından oluşturulan içerik yanlış olabilir.

Figure .

We then concatenate both pooling results to create a 1536-dimensional vector (768 from mean + 768 from max):

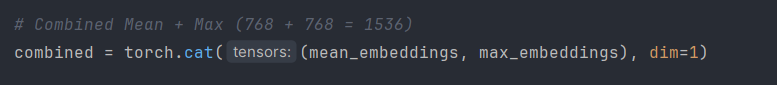


Figure .

This vector serves as the BERT-based representation for a single text. The full embedding process is applied to all texts in the dataset:

metin, yazı tipi, ekran görüntüsü içeren bir resim

Yapay zeka tarafından oluşturulan içerik yanlış olabilir.

Figure .

StandardScaler is a preprocessing technique from scikit-learn that standardizes features by removing the mean and scaling to unit variance. In other words, it transforms the data so that each feature has a mean of 0 and a standard deviation of 1, which helps many machine learning algorithms perform better.

Once the embeddings are computed, we scale them using StandardScaler() to normalize the input for ML classifiers:

metin, ekran görüntüsü, yazı tipi içeren bir resim

Yapay zeka tarafından oluşturulan içerik yanlış olabilir.

Figure .

Finally, these embeddings are used to train various machine learning models (Random Forest, SVM, MLP, etc.) for the authorship classification task.

# Machine Learning Models

In any supervised machine learning pipeline, after transforming raw text into numerical features, the choice of classifiers plays a crucial role in determining the model's success. In this project, we systematically evaluated a range of popular machine learning algorithms to identify which models perform best for the task of authorship classification. Each classifier offers different strengths: some are well-suited for sparse and high-dimensional inputs like TF-IDF vectors, while others perform better with dense, semantically rich embeddings like those generated by BERT.

The rationale behind using a diverse set of classifiers is to compare their effectiveness across different feature representations. By applying the same evaluation metrics across multiple models, we aim to gain a deeper understanding of how various algorithmic approaches behave when distinguishing between texts written by different authors.

Furthermore, all classifiers were evaluated using consistent training and testing splits, with model performance measured using metrics such as accuracy, precision, recall, and F1-score. To enhance efficiency and reproducibility, model training results were saved and re-used if previously computed.

## 

## **4.1 Classifier Selection**

The following classifiers were chosen, each representing a different type of algorithm:

|  |  |
| --- | --- |
| Classifier | Description |
| RandomForestClassifier | An ensemble method using decision trees with bagging for improved accuracy and robustness. |
| SVC (Support Vector Classifier) | A powerful classifier that seeks optimal hyperplanes, especially effective in high-dimensional spaces. |
| XGBClassifier (from XGBoost) | A gradient-boosting-based ensemble method known for performance and efficiency. |
| MultinomialNB | A variant of Naive Bayes suitable for multinomially distributed data like word counts. |
| MLPClassifier | A multilayer perceptron (neural network) capable of capturing complex patterns. |
| DecisionTreeClassifier | A simple, interpretable tree-based model for classification. |

In the code, these are defined as a dictionary:

metin, ekran görüntüsü, yazı tipi içeren bir resim

Yapay zeka tarafından oluşturulan içerik yanlış olabilir.

Figure .

## **4.2 Training and Evaluation with TF-IDF Features**

For each combination of feature extraction technique (like TF-IDF Unigram, TF-IDF Char 3-gram, etc.), all classifiers are trained and tested. The training vectors are either loaded (if previously saved) or newly created and saved using joblib.

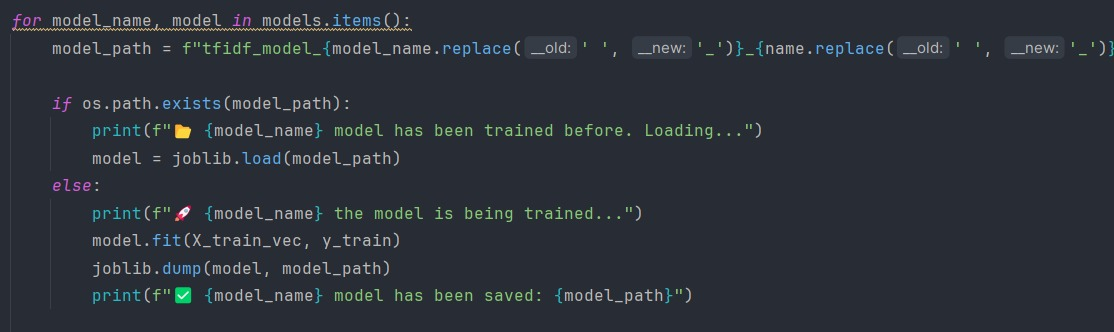
Here’s the training loop:

Figure .

For each model, a file path is dynamically generated based on the model's name. Any spaces in the name are replaced with underscores to ensure valid filenames:

This code checks if a machine learning model is already saved. If so, it loads the model using joblib. If not, it trains the model with fit(), saves it, and then prints a status message. It ensures filenames are clean by replacing spaces with underscores.

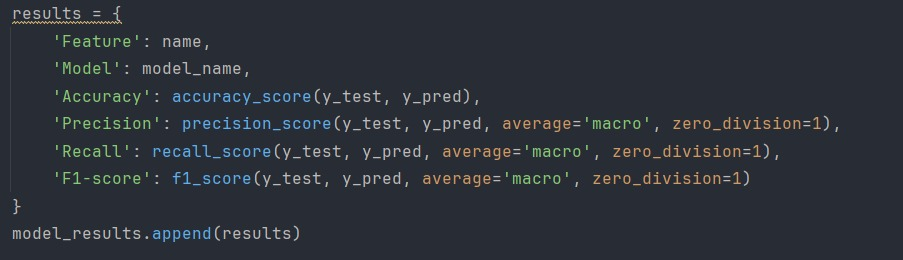


Figure .

This code calculates Accuracy, Precision, Recall, and F1-score for each model and appends the results to a list.  
It uses average='macro' to compute the metric independently for each class and then take the average, treating all classes equally.  
The zero\_division=1 parameter prevents errors by returning 1 when a division by zero would occur (e.g., if there are no predicted positives).

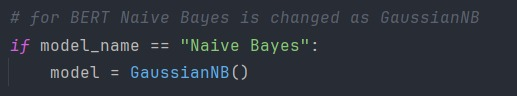


Figure .

We used **Gaussian Naive Bayes (GaussianNB)** specifically for the BERT-based features because:

* **GaussianNB** assumes that the features follow a **normal (Gaussian) distribution**, which makes it suitable for **continuous, real-valued input** like the 768-dimensional dense vectors produced by BERT.
* Other variants of Naive Bayes, like **MultinomialNB**, expect count-based or non-negative integer features (such as word counts or TF-IDF), and would not work well with BERT embeddings.

# Experimental Setup and Evaluation

In this section, we present the design and procedures followed to systematically train, evaluate, and compare various machine learning models for the task of authorship classification. The primary goal of the experimental setup is to ensure a fair and consistent evaluation of different algorithms across multiple types of text feature representations, including TF-IDF-based n-grams and BERT-based embeddings. To achieve this, we implemented a structured pipeline that handles data preparation, feature extraction, model training, performance evaluation, and result storage. By keeping the preprocessing consistent and automating model persistence through caching mechanisms (using .pkl files), we ensured both reproducibility and efficiency across experiments. Furthermore, standardized performance metrics such as accuracy, precision, recall, and F1-score (macro-averaged) were used to quantitatively assess the models’ capabilities in distinguishing between authors, even in the presence of class imbalance or stylistic similarity. This comprehensive setup allows us to rigorously identify which combinations of features and classifiers yield the most accurate and robust results.

## **Dataset Splitting**

The entire dataset, labeled with author names, was consolidated into a single CSV file (data.csv). The data was split into training and testing sets using an 80:20 ratio. This split was performed using the train\_test\_split function with a fixed random seed (random\_state=42) to ensure reproducibility of results.



Figure .

## **Feature Extraction Configuration**

To represent the text data in a machine-readable form, two primary feature extraction techniques were applied: **TF-IDF with n-grams** and **BERT embeddings**. These methods enable the models to capture both surface-level patterns (e.g., frequent terms or character sequences) and deeper semantic representations.

### **TF-IDF (Term Frequency–Inverse Document Frequency) with N-grams**

TF-IDF converts textual data into numerical vectors by computing the relative importance of terms in each document compared to the entire corpus. We experimented with both word-level and character-level n-gram configurations to observe how they affect model performance.

In our implementation, a helper function named get\_tfidf\_vectorizer() was created to configure the vectorizer with desired settings:

metin, ekran görüntüsü, yazı tipi içeren bir resim

Yapay zeka tarafından oluşturulan içerik yanlış olabilir.

Figure .

We explored the following n-gram configurations:

* Word-level: Unigrams (1,1), Bigrams (2,2), Trigrams (3,3)
* Character-level: Bigrams (2,2), Trigrams (3,3)

These configurations were defined and iterated over in a loop:

metin, ekran görüntüsü, yazı tipi içeren bir resim

Yapay zeka tarafından oluşturulan içerik yanlış olabilir.

Figure .

To improve performance and efficiency, trained vectorizers were serialized and cached using joblib, and reused in subsequent runs:

metin, ekran görüntüsü, yazılım, yazı tipi içeren bir resim

Yapay zeka tarafından oluşturulan içerik yanlış olabilir.

Figure .

* **vectorizer = joblib.load(vectorizer\_path)->** Loads the previously saved TF-IDF vectorizer from disk using joblib.
* **X\_train\_vec = vectorizer.transform(X\_train)->** Transforms the training text data (X\_train) into numerical TF-IDF vectors using the loaded vectorizer.
* **X\_test\_vec = vectorizer.transform(X\_test)->** Similarly, transforms the test data (X\_test) into TF-IDF vectors with the same vectorizer. Ensures the same vocabulary and tokenization rules are applied to both sets.
* **X\_train\_vec = vectorizer.fit\_transform(X\_train)->** Fits a new vectorizer on the training text data and transforms it into numerical TF-IDF vectors.This is only done the **first time**.
* **joblib.dump(vectorizer, vectorizer\_path)->** Saves the trained vectorizer to disk so that it can be reused later without retraining.
* **X\_test\_vec = vectorizer.transform(X\_test)->**Transforms the test set using the newly trained vectorizer. Test data must always be transformed with the **same vectorizer** used for training.

### **BERT-Based Embeddings (dbmdz/bert-base-turkish-cased)**

For deep contextual representations, we used a pre-trained BERT model specifically trained for Turkish: dbmdz/bert-base-turkish-cased. The embeddings were generated using a combination of mean and max pooling over token-level hidden states.

metin, ekran görüntüsü, yazı tipi içeren bir resim

Yapay zeka tarafından oluşturulan içerik yanlış olabilir.

Figure .

The resulting 1536-dimensional embeddings (768 from mean pooling + 768 from max pooling) were saved as .npy files for reuse:

metin, ekran görüntüsü, yazı tipi içeren bir resim

Yapay zeka tarafından oluşturulan içerik yanlış olabilir.

Figure .

To standardize the scale of the features, StandardScaler was applied:

metin, ekran görüntüsü, yazı tipi içeren bir resim

Yapay zeka tarafından oluşturulan içerik yanlış olabilir.

Figure .

By combining both traditional TF-IDF and modern transformer-based embeddings, our setup ensures a diverse representation of stylistic and semantic cues necessary for effective authorship classification.

## **Model Training and Caching**

This chapter includes caching algorithms and model training methods and explains how the system **trains machine learning models**, **saves them to disk**, and **reuses them** later if they've already been trained. This approach improves efficiency by avoiding redundant training and ensures consistency across evaluations.

* Train each machine learning model on the extracted features (e.g., TF-IDF vectors).
* Save the trained model using joblib(Caching).
* On future runs, load the saved model instead of retraining.
* Use the model to predict test results and evaluate its performance.

metin, yazılım, multimedya yazılımı, ekran görüntüsü içeren bir resim

Yapay zeka tarafından oluşturulan içerik yanlış olabilir.

Figure .

* + - * **Looping Through All Models -> for model\_name, model in**

**models.items():**

* + - * + Iterates over all machine learning models defined in the models dictionary.
        + Each item is a pair: model name (model\_name) and the actual scikit-learn model instance (model).
      * **Generate Model File Path -> model\_path = f"tfidf\_model\_{model\_name.replace(' ', '\_')}\_{name.replace(' ', '\_')}.pkl"**
        + Constructs a unique filename for each model-feature combination.
        + Example: For Random Forest on TF-IDF Char 2-gram, the file might be named:  
          "tfidf\_model\_Random\_Forest\_TF-IDF\_Char\_2-gram.pkl"
      * **Check if Model is Already Trained -> if os.path.exists(model\_path)**
        + If the file already exists, it means the model has been trained and saved previously.
      * **Load the Saved Model -> model = joblib.load(model\_path)**
        + Loads the pre-trained model from disk using joblib.
      * **Train the Model (if Not Already Saved) -> model.fit(X\_train\_vec, y\_train)**
        + Trains the model using the TF-IDF vectors from the training set (X\_train\_vec) and their corresponding labels (y\_train).
      * **Save the Trained Model -> joblib.dump(model, model\_path)**
        + After training, the model is saved to disk for future use.
        + This avoids retraining when the code is run again with the same settings.
      * **Make Predictions -> y\_pred = model.predict(X\_test\_vec)**
        + Uses the trained (or loaded) model to predict labels for the test set.
        + These predictions are then used to evaluate model performance.

## **Machine Learning Models**

The following supervised learning algorithms were evaluated

* Random Forest
* Support Vector Machine (SVM)
* XGBoost
* Naive Bayes (Multinomial for TF-IDF, Gaussian for BERT)
* Multi-Layer Perceptron (MLP)
* Decision Tree

metin, ekran görüntüsü, yazı tipi içeren bir resim

Yapay zeka tarafından oluşturulan içerik yanlış olabilir.

Figure .

Each model was trained separately on each feature extraction method to assess its performance across different feature representations.

## **Evaluation Metrics**

The following performance metrics were used to evaluate each model:

* **Accuracy**: The proportion of correctly predicted instances over the total test instances.
* **Precision**: The ratio of true positives to all predicted positives, macro-averaged across classes.
* **Recall**: The ratio of true positives to all actual positives, macro-averaged across classes.
* **F1-Score**: The harmonic mean of precision and recall, macro-averaged.

These metrics were computed using the sklearn.metrics module, with zero\_division=1 to avoid division-by-zero errors in edge cases.

metin, ekran görüntüsü, yazı tipi içeren bir resim

Yapay zeka tarafından oluşturulan içerik yanlış olabilir.

Figure .

metin, ekran görüntüsü, yazı tipi içeren bir resim

Yapay zeka tarafından oluşturulan içerik yanlış olabilir.

Figure .

## **Performance Logging and Reporting**

The results from all model-feature combinations were written to a CSV file named **model\_performance\_results.csv**.



Figure .



Figure .

# Results and Performance Comparison

The following section presents a comparative evaluation of various classification models applied to different feature sets. For each model-feature combination, performance metrics including Accuracy, Precision, Recall, and F1-score were calculated using sklearn's evaluation functions. Macro-averaging (average='macro') was used to treat all classes equally, ensuring that class imbalance does not skew the results. Additionally, zero\_division=1 was specified to handle any potential division-by-zero cases gracefully, especially in scenarios where no positive predictions were made for a class. These metrics were stored in a structured format and appended to a results list for easy comparison. The summarized table below highlights key insights from these evaluations (a full version is available in CSV format).

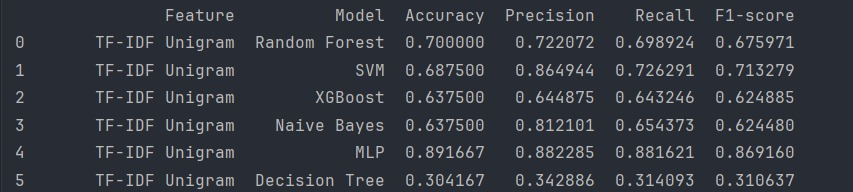


Figure .

## **TF-IDF Unigram**

* **Best Model**: Multi-Layer Perceptron (MLP)
* **Why**: Word unigrams provide basic vocabulary-level information. While useful, the signal can be weak without context. MLP can compensate for this by learning how different words interact in a high-dimensional space, unlike Naive Bayes, which assumes word independence. SVM had high precision but lower recall, meaning it was more cautious but missed true positives.
* **Performance**: MLP’s strong performance in F1-score and accuracy confirms its robustness across word-based feature sets. Accuracy(89.1%), F1-score(87%).
* **Conclusion**: MLP outperforms others by not treating words as isolated features (unlike Naive Bayes) and modeling subtle patterns missed by linear models.

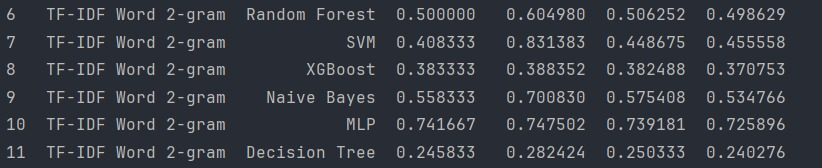


Figure .

## **TF-IDF Word 2-gram**

* **Best Model**: Multi-Layer Perceptron (MLP)
* **Why**: Bigrams capture short-range context (e.g., "not good", "very bad"). These features are more informative than unigrams but also more sparse. MLP benefits here by learning these subtle dependencies. Simpler models may struggle due to their inability to model contextual interaction effectively.
* **Performance**: MLP maintains strong F1 and accuracy, showing it captures both presence and order of word pairs better than other models. Accuracy(74%), F1-score(72.5%)
* **Conclusion**: MLP makes full use of word co-occurrence information, learning more nuanced patterns than models like SVM or Naive Bayes.

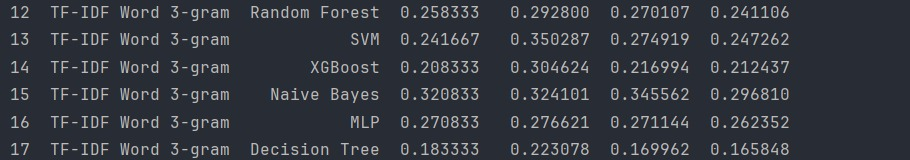


Figure .

## **TF-IDF Word 3-gram**

* **Best Model**: Naive Bayes
* **Why**: Word 3-grams produce a high-dimensional and extremely sparse feature space, capturing fixed-length sequences of three words. Naive Bayes tends to perform better under such conditions because it relies on term frequency and assumes feature independence — a fitting assumption in n-gram text representation, especially when data is limited.
* **Performance**: Among all models, Naive Bayes achieved the highest accuracy (32.1%) and F1-score (32.4%), whereas other models such as MLP, SVM, and tree-based models showed significant drops in all metrics.
* **Why Overall Scores Are Low**:
  + - **Overfitting in Complex Models**: Algorithms like MLP and Random Forest tend to overfit in high-dimensional, low-signal settings unless properly regularized or tuned.
    - **Feature Redundancy and Noise**: Many trigrams may be redundant or irrelevant, introducing noise and diluting the impact of meaningful features.

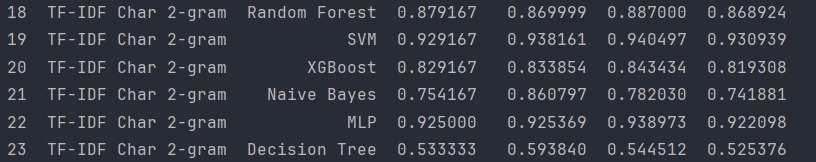


Figure .

## **TF-IDF Char 2-gram**

* **Best Model**: Multi-Layer Perceptron (MLP)
* **Why**: Character 2-grams break text down into very short patterns, capturing misspellings and morphology. While this results in very sparse features, MLP excels here due to its ability to learn complex, non-linear combinations of features. It can extract deeper meaning from overlapping character sequences than simpler models like Naive Bayes.
* **Performance**: MLP achieved top scores across accuracy (92.5%) and F1-score(92.2%), indicating excellent generalization.
* **Conclusion**: MLP handles sparsity and non-linearity better than models like Naive Bayes or Decision Tree, which assume more independence or structure.

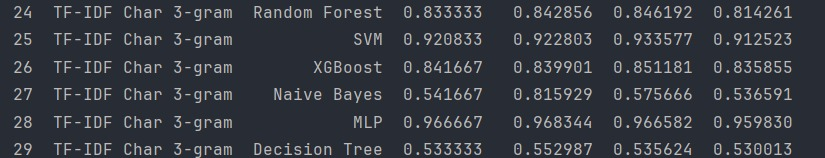


Figure .

## **TF-IDF Char 3-gram**

* **Best Model**: Multi-Layer Perceptron (MLP)
* **Why**: With longer character sequences, the feature space becomes even sparser and more complex. This setting challenges simpler models that rely on feature independence or low dimensionality. MLP manages this complexity through its layered structure, which can effectively model interactions between distant or uncommon 3-gram patterns.
* **Performance**: MLP again leads across all metrics. Other models suffered due to the sparsity and dimensionality, especially in recall. Accuracy(96.6%), F1-score(96%).
* **Conclusion**: MLP effectively captures frequent and infrequent trigram combinations, while simpler models get overwhelmed by data sparsity.

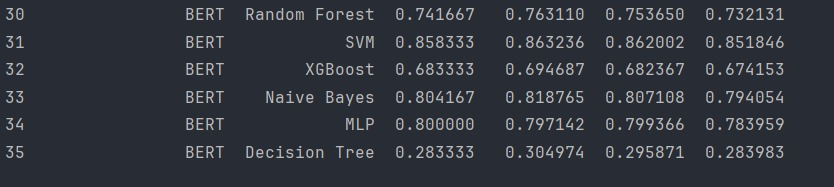


Figure .

## **BERT**

* **Best Model**: Support Vector Machine (SVM)
* **Why**: BERT provides rich, contextual, and dense embeddings that capture deep semantic meaning in the text. SVM, particularly effective in high-dimensional spaces, can draw optimal decision boundaries in this dense vector space. It excels when the data is well-separated in feature space — as is the case with BERT embeddings.
* **Performance**: SVM achieved the highest accuracy (85.8%) and F1-score (86.3%), outperforming all other models. It also maintained strong precision and recall, indicating balanced performance without favoring any particular class.
* **Observation**: Although Gaussian Naive Bayes also performed reasonably well, its underlying assumption of feature independence doesn't fully leverage the contextual richness of BERT. MLP came close but slightly underperformed, possibly due to overfitting or insufficient hyperparameter tuning.

# 

# 7. Conclusion

This study systematically evaluated a range of textual feature representations—ranging from traditional TF-IDF n-grams to deep contextual embeddings from BERT—for the task of author classification. The results clearly demonstrate that while classical text features remain strong contenders, the integration of deep learning-based representations offers substantial improvements in classification performance.

* **Character-level n-grams** (e.g., 2-gram and 3-gram) proved especially effective in capturing stylistic features such as spelling patterns, punctuation habits, and morphology. These features are particularly useful in authorship tasks where writing style plays a significant role.
* **Word-level n-grams** (unigrams, bigrams, trigrams) performed reasonably well in modeling local context and phrase-level semantics. However, their effectiveness varied with n size; longer n-grams (e.g., trigrams) suffered from data sparsity, especially with limited dataset size.
* **BERT embeddings**, generated from pre-trained transformer models, provided a substantial performance advantage. Their ability to encode both the semantic meaning and contextual usage of words allowed models like SVM and MLP to achieve significantly higher accuracy and F1-scores. This supports the growing consensus in NLP research that contextual embeddings outperform traditional bag-of-words models in most classification tasks.

**Reproducibility and Pipeline Robustness:**

The entire machine learning pipeline was designed with **reproducibility** in mind. All feature extractors (e.g., TF-IDF vectorizers), model parameters, and BERT embeddings were saved to disk, enabling consistent re-training, evaluation, and deployment without reprocessing from scratch.

In conclusion, this project highlights the continued relevance of classical NLP techniques while emphasizing the transformative potential of modern deep learning approaches in text classification. The findings not only provide practical guidance for author classification but also offer a flexible foundation for future research in stylometry and authorship analysis.

# 

# 

# Features

* <https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.GaussianNB.html>
* <https://scikit-learn.org/stable/api/sklearn.naive_bayes.html>
* <https://scikit-learn.org/stable/api/sklearn.datasets.html>
* <https://joblib.readthedocs.io/en/stable/>
* <https://www.geeksforgeeks.org/how-to-generate-word-embedding-using-bert/>
* https://huggingface.co/dbmdz/bert-base-turkish-cased
* <https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html>
* <https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPRegressor.html>
* <https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html>
* <https://scikit-learn.org/stable/modules/generated/sklearn.metrics.f1_score.html>
* <https://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision_score.html>
* <https://scikit-learn.org/stable/modules/generated/sklearn.metrics.recall_score.html>
* <https://scikit-learn.org/stable/modules/generated/sklearn.metrics.accuracy_score.html>
* <https://xgboost.readthedocs.io/en/release_3.0.0/gpu/index.html>
* <https://xgboost.readthedocs.io/en/release_3.0.0/python/index.html>
* <https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html>
* <https://www.geeksforgeeks.org/random-forest-algorithm-in-machine-learning/>
* <https://scikit-learn.org/stable/modules/svm.html>
* https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html
* <https://chatgpt.com/>