# Emotion Classification Project

## Project Report


| | |
|---|---|
| Group IDs: | Antonios Chnarakis – U244N1084<br>Evros Vasileiou – U244N1089<br>Muhammad Faisal - U244N1091 |

| | |
|---|---|
| Program: | Data Science (MSc) – DL |
| Semester: | Spring, 2025 |
| Course: | COMP-544DL Machine Learning |
| Instructor: | Dr. Ioannis Katakis |

## 1. Executive Summary

The project '**Designing a Machine Learning Model for Emotions Classification**' has been assigned by Professor Ioannis Katakis as a capstone project for COMP-544DL Machine Learning.

This report aims at furnishing a detailed walkthrough of the activities performed by the project team during analysis, design, implementation, evaluation and optimization phases of the project. The report is appended with Python Jupyter Notebooks where all the essential coding required for the performance of the model is presented. Wherever necessary, references have been provided for the relevant theoretical inferences.

## 2. Data Analysis and Project Planning

The project was kicked off with the analysis of the given dataset and defining the project scope and objectives. The dataset namely 'train.csv' contains two attributes 'text' and 'emotion' and the project outcome will be a machine learning model capable of predicting emotion labels for the new, unknown data provided via an anonymous 'test.csv' file while storing the predicted labels in a new 'predictions.txt' file. In order to execute the project, we devised the following project schedule:

| Start Date | End Date | Activities & Milestones |
|---|---|---|
| 28-Apr-25 | - | Project Assigned |
| 29-Apr-25 | 5-May-25 | Team Development, Data Exploration & Preprocessing |
| 6-May-25 | 6-May-25 | Webex Session for QnA |
| 7-May-25 | 10-May-25 | Model Selection & Development of Algorithms |
| 11-May-25 | 13-May-25 | Model Evaluation |
| 14-May-25 | 16-May-25 | Model Optimization and Fine-tuning |
| 10-May-25 | 18-May-25 | Drafting Technical Report & Documentation of Lessons Learned |
| - | 19-May-25 | Project Submission |

## 3. Project Execution

Although the annexed Jupyter notebooks contain self-explanatory steps explained with marked-down comments, the rationale behind deploying the codes has been explained in this document, which may be read in conjunction with the Python codes.

## a. Data Exploration & Preprocessing

Upon detailed review of the given dataset, below preprocessing steps were deemed necessary before fetching the data for training of algorithms:

- **Basic Data Analysis**: Checking for null values and missing values and performing basic text statistics
- **Identify duplicated texts with conflicting emotions**: There were multiple instances of duplicate text having dissimilar emotion labels. This could be due to different context in the hindsight and removal of these texts could skew the algorithm for similar instances in the test data. Therefore, after a mutual consensus, we decided to accept this anomaly as-is in the training dataset.
- **Noise Reduction**: The meaningless expressions like web URLs, special characters, hashtags, digits, numerics, mentions, handles and unnecessary multiple spaces were removed.

- **Conditional Lemmatization**: This being a large dataset required some optimization to reduce workload on the model and reduce run time. Hence, lemmatization was skillfully performed with minimum impact on context.

- **Conditional Stemming**: Stemming simplifies words to their root forms, reducing vocabulary size and potentially improving generalization and model performance in emotion classification tasks. Therefore, conditional stemming during text preprocessing was applied based on parameters in the preprocess_text function using the Porter Stemmer and only non-stopwords are stemmed.

- **Contractions**: Identified phrases that include contractions and resolved them by utilizing the contractions library of Python.

- **Stop-words Removal**: Deployed NLTK Library to remove stop-words. However, words like 'not' and 'no' which if removed, could misinterpret the context of the text were purposely kept in the learning dataset. This is quite important as we identified during the exploratory data analysis phase were in particular many of the most frequent bigrams and trigrams contain 'not' and 'no'.

- **Non-English Word Removal**: Initially performed, later revoked as some words were either nouns / adjectives or had secondary relation with context and their removal could deprive the text from emotional references.

- **Miscellaneous Data Exploration Activities**: The dataset was further explored via plethora of data exploration and visualization techniques like plotting word-cloud, by emotions to have a visual representation of our input file, visualizing Bigrams and Trigrams to explore linguistic patterns.

- **Word Tokenization & Vectorization**: Deployed time-tested TF-IDF (tf-idfVectorizer) method along with Bag of Words (CountVectorizer) which are self-explanatory vectorization procedures to while working with textual data by converting it into fixed-length numeric vectors during such emotion classification projects.

**EDA_&_Preprocessing_Analysis_of_Data.ipynb:** This file contains the relevant codes for the data exploration steps explained above. This doesn't connect with the final deliverables (main.ipynb and our methods) but deemed necessary before modelling and/or deploying the algorithms.

Prepossessing was then expanded and segregated to two jupyter notebooks where the next phase of program was developed in parallel and seamlessly integrated. These jupyter notebooks, where the next playing field was set, were named 'methods.py' and 'main.ipynb':

**methods.py:** The methods.py file defines the TextPreprocessor class, which classify text preprocessing further. It includes the method preprocess_text that performs several cleaning steps: expanding contractions, converting text to lowercase, removing URLs, mentions, hashtags, punctuation, numbers, and extra whitespace. It also removes common English stopwords, with the exception of "no" and "not". The method allows optional lemmatization using WordNet or stemming using Porter's algorithm. This class is designed to standardize and clean raw text for further use in machine learning workflows. An instance is also exposed as preprocess_text for convenience.

**main.ipynb:** In main.ipynb, preprocessing is applied to the datasets within the train_test function. An instance of TextPreprocessor is created and used to generate three versions of the text column: text_clean, which applies only basic cleaning; text_lemmatized, which includes lemmatization; and text_stemmed, which applies stemming. These three processed variants are created for both training and testing datasets and stored as new columns. The resulting preprocessed texts are then used as inputs for model training and evaluation. The preprocessing here ensures that models are tested across different text normalization strategies.

This concluded the data preprocessing phase and we proceeded to model selection after documenting lessons learned.

## b. The Framework for Model Selection

Based on the project charter, this phase brought new challenges to project team which were addressed with synergy in the light of theoretical principles learned throughout the semester:

- **Selecting model parameters**: We were required to select optimal values for critical model parameters influencing model complexity, performance and run-time, keeping in view the large size of dataset.

- **Defining Classifiers:** In the code, several machine learning models are used to classify emotions from text. ***Logistic Regression*** finds the optimal boundary to separate different emotion classes based on word features and is both fast and effective for text data. ***Support Vector Machine (SVM)*** uses a linear hyperplane to maximize the separation between classes, making it highly accurate for high-dimensional inputs like TF-IDF vectors. SVM with polynomial kernel, radial basis function kernel, sigmoid kernel and custom kernel were not used. ***Naive Bayes*** relies on word occurrence probabilities and is especially efficient for handling large-scale text classification tasks. ***Decision Trees*** split data based on features to form a hierarchy of decisions, providing interpretability but requiring tuning to avoid overfitting. ***Random Forest*** improves on decision trees by combining multiple trees and averaging their predictions, increasing robustness and accuracy. These models are compared in the code to select the best one for emotion classification.

- **Experimentation & Selection of Best Performing Algorithm:** A comparative analysis was performed using text features transformed via TF-IDF and, for testing suitability and also Bag of Words. These features were derived from various preprocessing strategies, including raw, lemmatized, and stemmed variants. Each classifier was trained and evaluated across these variants using either direct accuracy computation. Among the tested models, TF-IDF consistently outperformed CountVectorizer, which was found to deliver inferior results across all classifiers and preprocessing variants. As such, we adopted TF-IDF as the default feature extraction method due to its superior performance with sparse, high-dimensional text data. Cross-validation revealed **SVM as the top-performing classifier**. We then fine-tuned its hyperparameters using GridSearchCV, particularly exploring different regularization **parameters (C)** to control margin width and decision boundary flexibility. The optimal configuration was found to be an **SVM with a linear kernel** and **C=0.1**, effectively balancing margin maximization with misclassification tolerance. This combination proved robust and efficient, especially in synergy with TF-IDF features, making it the best-suited model for our classification task.

## c. Implementation Phase

As explained earlier, the implementation was performed via seamlessly integrated notebooks ‘methods.py’ and ‘main.ipynb’:

**methods.py**

In line with the attached Jupyter notebook defined as ‘methods’, the plan devised in model selection is fundamentally implemented through the TextModelEvaluator class and compare_models() methods.

**main.ipynb**

The notebook acts as the front-end application of the TextModelEvaluator. It starts by loading and preprocessing the training and test datasets. Three processed versions of the text are created:

- Basic cleaned text
- Lemmatized text
- Stemmed text

These versions are passed into the evaluator as a dictionary of variants. The notebook then runs compare_models() with cross-validation enabled. Once complete, the notebook prints out the best model and best preprocessing method selected by the evaluator.

This segment is also equipped with the required function train_test() responsible for learning parameters from the train.csv file and generating predictions for the test.csv file.

### d. Model Evaluation & Optimization

**methods.py**

Three additional methods support this stage:

**1. Hyperparameter Tuning**: tune_best_model() uses GridSearchCV to systematically search through combinations of hyperparameters for the selected model. This ensures the model is not only good by default but also fine-tuned for the task at hand.

**2. Validation Evaluation**: evaluate_on_validation() splits the training data into a training set and a validation set. The selected model is retrained and tested, and performance metrics like accuracy, macro F1-score, and precision/recall are calculated. Optional visualizations like confusion matrices and ROC curves help in validating the algorithms results.

**3. Final Model Training & Prediction**: The best ranking model in terms of accuracy is selected and further used to retrain the full dataset (train_final_model()), and predictions are generated on the unseen test set using the predict() method.

**main.ipynb**

Following the model selection, the notebook checks the accuracy of the best model and defines a relevant hyperparameter grid. This grid is passed to tune_best_model() to perform tuning.

Next, the notebook evaluates the tuned model using evaluate_on_validation(). This step helps in validating that the model generalizes well beyond the training set.

Finally, the tuned model is trained on the entire dataset, and predictions are generated for the test set. These predictions are then saved into the **predictions.txt** file for submission or further use.

### e. Model Results

After a manual split on the **train.csv** file, the dry run of the model generated the best variant and algorithm successfully on all occasions. To ensure consistent and reproducible results, a fixed random_state was used throughout the training and evaluation process. This allowed for stable data splits and helped in identifying the most reliable classifier. Later this switched from fixed state to random state. Multiple models were evaluated through cross-validation, and their performance was compared across different text preprocessing variants. Once the best-performing model and variant were selected based on cross-validation accuracy, hyperparameter tuning was conducted using GridSearchCV to further improve the model's performance. The code automatically selected the model with the highest accuracy, trained it on the full training set, and used it to generate predictions on the test data. These predictions were then saved in a **predictions.txt** file, with one emotion label per line, ready for final submission and evaluation.

The illustration below represents one simulation and the outcome of cross-validation performed on the model - as long as we explicitly specify the split files manually to **train2.csv** file with approximately 5,398 records and **test.csv** file with 2,602 records (without labels). Below illustration represents one simulation and the outcome of cross-validation performed on the model:

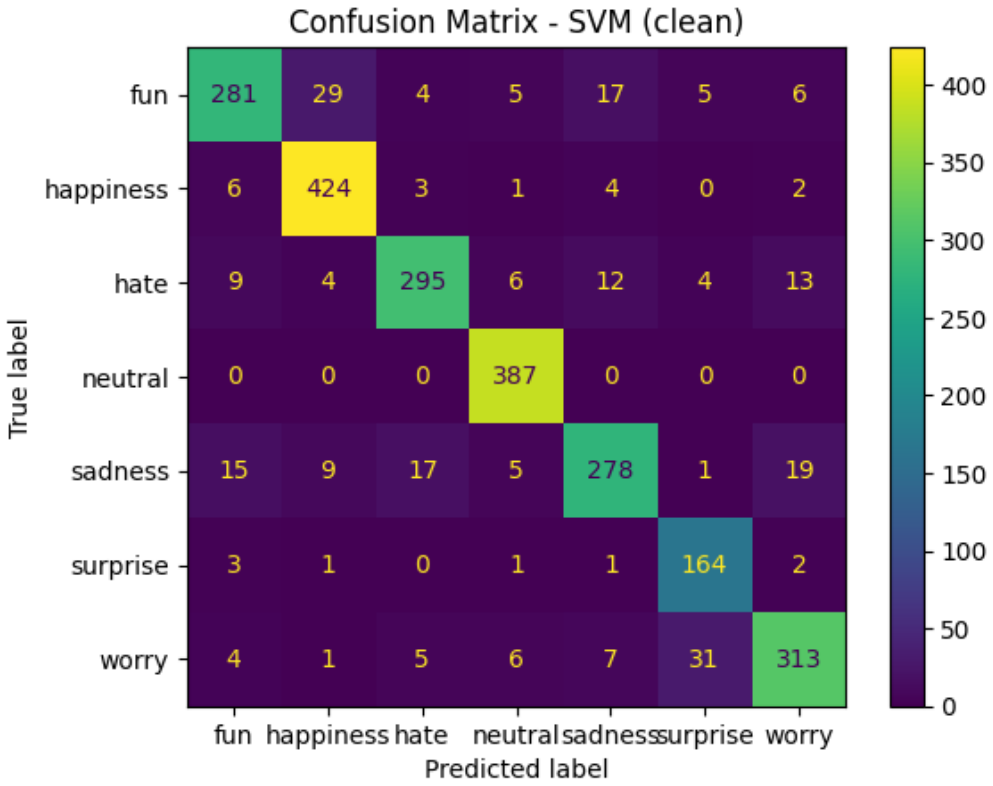
| Variant | Model | Accuracy % | Rank |
|---|---|---|---|
| Clean | SVM | 88.6807 | 1 |
| Clean | Logistic Regression | 87.1988 | 2 |
| Lemmatized | SVM | 86.5325 | 3 |
| Lemmatized | Logistic Regression | 86.0134 | 4 |
| Stemmed | SVM | 85.4389 | 5 |
| Stemmed | Logistic Regression | 84.7537 | 6 |
| Stemmed | Random Forest | 83.1419 | 7 |
| Clean | Random Forest | 82.7713 | 8 |
| Lemmatized | Random Forest | 82.5121 | 9 |
| Clean | Naive Bayes | 78.6399 | 10 |
| Lemmatized | Naive Bayes | 78.4922 | 11 |
| Lemmatized | Decision Tree | 76.4185 | 12 |
| Stemmed | Decision Tree | 78.1398 | 13 |
| Stemmed | Naive Bayes | 76.9727 | 14 |
| Clean | Decision Tree | 76.7128 | 15 |

Below you can see the best result obtained post tuning:

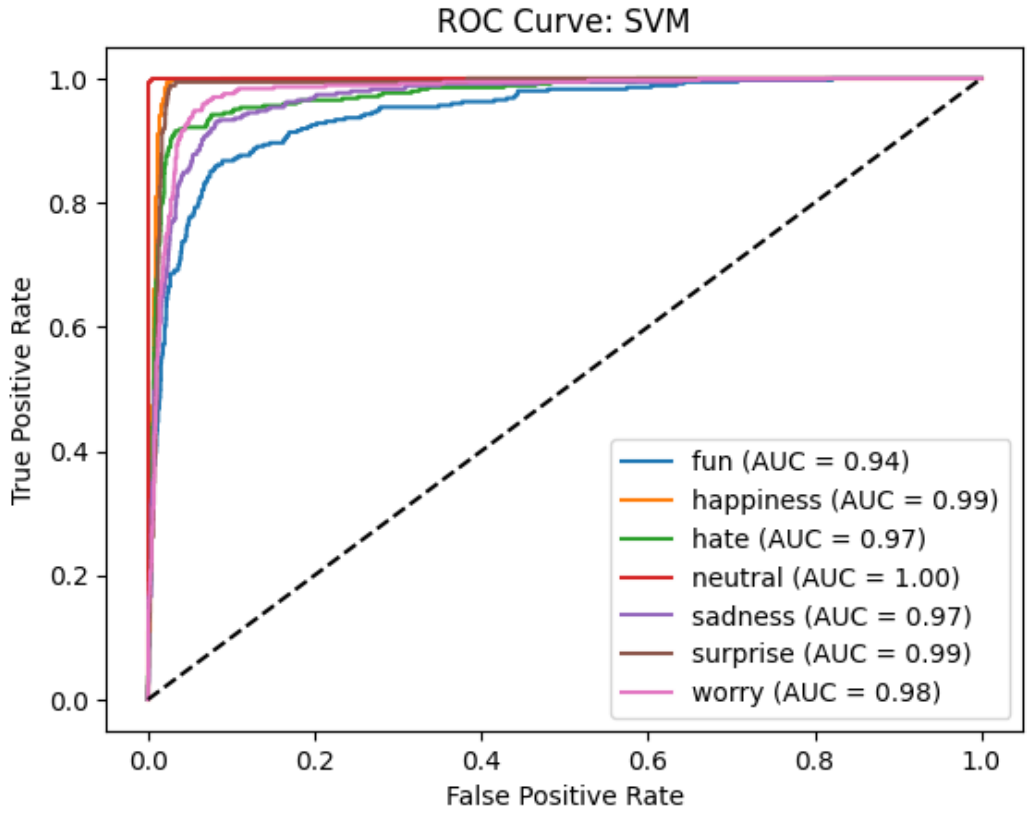
***Best hyperparameters: {'C': 0.1}***
***Best cross-validated score after tuning: 0.889218***

These hyperparameters were selected using GridSearchCV for tuning. This method systematically tested multiple values to find the configuration that yielded the highest accuracy.

A Confusion Matrix for the best performing algorithm is provided below:




In addition, the ROC Curve for the best performing algorithm is provided below:



## 4. Project Close-out & Lessons Learned

The union of methods and classifiers successfully yielded the intended project deliverables with reasonable accuracy for every simulation of random test.csv datasets. The tuned model is trained on the entire train.csv dataset, and predictions are being generated and then saved into a predictions.txt file for submission or further use. As a result, we concluded the project documenting following lessons learned:

- This project provided us a valuable opportunity to collaborate with the cohorts of MDATA program. Separated by long distances, divided by time zones yet focused at a common goal, we enjoyed the pleasure of teamwork and inclusion which is by far the best experience of our master's journey. We remain grateful to our faculty for concluding the coursework with a project assignment which enabled us to implement the theory and witness the challenges of data mining and machine learning projects first-hand.

- Machine learning projects are highly sophisticated and complex endeavors especially when you are dealing with large dataset having ambiguous context. Hence, systematic experimentation with multiple classifiers and text variants is found efficient for identifying the most effective model configuration. This structured approach supported us for the selection of the best model-variant combination.

- Preprocessing text with attention to task-specific linguistic cues, such as preserving negations (e.g., "no", "not") and accepting data irregularities impacting context, significantly improves model performance in such emotion classification tasks. Thus, careful preprocessing of data, ensured we are feeding the algorithm fit-for-purpose training dataset without compromising on crucial contextual signals i.e sentimental laden phrases.

## 5. Attachments

1. Readme.txt [This file contains necessary instructions to run the program. It is suggested to be read first]
2. Requirements.txt [This file contains list of all libraries to run the codes]
3. libraries.py [This file contains list of Python Libraries required to run the codes]
4. EDA_&_Preprocessing_Analysis_of_Data.ipynb [Jupyter notebook having all preprocessing codes]
5. methods.py [Python program file having all classification methods]
6. main.ipynb [Jupyter notebook having the train_test function and classifiers]
7. Multiple simulation results [annexed within the report pdf]

## 6. References


1. Book: Introduction to Data Mining
2. Book: Hands-on Machine Learning with Scikit Learn, Keras & Tensorflow
3. The slide-deck available at Moodle Platform