

Deep Learning for NLP

Konstantinos Malonas

sdi: 7115112200020

Fall Semester 2023

University of Athens Department of Informatics and Telecommunications

Artificial Intelligence II (M138, M226, M262, M325)

Abstract	3
Data processing and analysis	4
Pre-processing	4
Analysis	6
Data partitioning for train, test and validation	10
Vectorization	11
Algorithms and Experiments	12
Experiments	12
Experiments with the max_features hyperparameter	12
Experiments with the ngram_range hyperparameter	13
Experiments with the max_iter hyperparameter	15
Experiments with the solver and penalty hyperparameters	17
Experiments with the C hyperparameter	20
Experiments with the multi_class hyperparameter	23
Experiments with the k fold cross validation	25
Use of CountVectorizer	26
Table of trials	26
Hyper-parameter tuning	27
Optimization	27
Evaluation	28
Results and Overall Analysis	29
Bibliography	29

Abstract

For this project we will build a sentiment classifier that classifies tweets written in Greek language about the Greek elections. The classifier will classify the tweets as NEGATIVE, POSITIVE and NEUTRAL classes. First, we will start by exploring my data, which includes plotting barplots, word clouds, etc., to recognize potential patterns, understand the predominant sentiment classifications for tweets associated with each party, and so on. Then we will continue with experiments to find the best hyperparameters for the Logistic Regression classifier and also we will plot the learning curves to observe the performance of the classifier I am building.

Data processing and analysis

Pre-processing

First we start with the pre-processing of our dataset. Initially we check if there are any null values in any column with the **info** method. We observe that there are no null values.

```
RangeIndex: 36630 entries, 0 to 36629
Data columns (total 4 columns):
#   Column      Non-Null Count  Dtype
---  -
0   New_ID      36630 non-null  int64
1   Text        36630 non-null  object
2   Sentiment   36630 non-null  object
3   Party       36630 non-null  object
dtypes: int64(1), object(3)
memory usage: 1.1+ MB
```

Turn categorical values to numerical with LabelEncoder

It is a best practice to turn the categorical values of our classes from our dataset to numerical. To achieve that, we use the **LabelEncoder** object. We turn our classes from NEGATIVE to 0, from NEUTRAL to 1, and lastly from POSITIVE to 2.

```
df_train_set['Sentiment'] = le.fit_transform(df_train_set['Sentiment'])
df_valid_set['Sentiment'] = le.fit_transform(df_valid_set['Sentiment'])
```

The preprocess_tweet function

In the data preprocessing stage, the **preprocess_tweet** function plays a crucial role in standardizing and cleaning the tweet data. This function performs several key operations to ensure that the text data is in a suitable format for analysis and modeling.

Firstly, we convert all text to lowercase, which helps in maintaining consistency as text data often contains a mix of uppercase and lowercase letters, and in most contexts, these variations are not meaningful. By standardizing the case, we reduce the complexity of the text and avoid treating the same words in different cases as different tokens.

Next, we remove mentions, which are words starting with the '@' character. Mentions in tweets usually refer to usernames and do not contribute meaningful information for our analysis. Similarly, URLs are removed since they often act as noise in the text analysis, being unique and not contributing to the overall understanding of the tweet's sentiment.

We also remove all non-Greek characters, focusing our analysis strictly on Greek text. This step is crucial as it eliminates irrelevant characters or symbols that might be present in the tweets but do not contribute to their semantic meaning.

Lastly, we remove Greek stopwords using the **stopwords-el.json** file. Stopwords are common words that appear frequently in the text but do not carry significant meaning and are often filtered out before processing text data. Removing these words helps in reducing the dimensionality of the data and focuses the analysis on the words that carry more meaning and sentiment.

```
nlp =
spacy.load('/kaggle/input/el-core-news-lg-2/el_core_news_lg/el_core_news
_lg-3.7.0')

def lemmatize_tokenize_text(text):
    doc = nlp(text)
    return ' '.join([token.lemma_ for token in doc])

df_train_set['Text'] =
df_train_set['Text'].apply(lemmatize_tokenize_text)
df_test_set['Text'] = df_test_set['Text'].apply(lemmatize_tokenize_text)
df_valid_set['Text'] =
df_valid_set['Text'].apply(lemmatize_tokenize_text)
```

The unique_words_num function

After we pre-processed our text we printed the unique words of each set with the use of **unique_words_num** function

```
def unique_words_num(tweets):
    # Function that counts the number of the unique words from the Text
    column of each dataframe
    words = set()
    for tweet in tweets:
        words.update(tweet.split())
    return len(words)
```

Result:

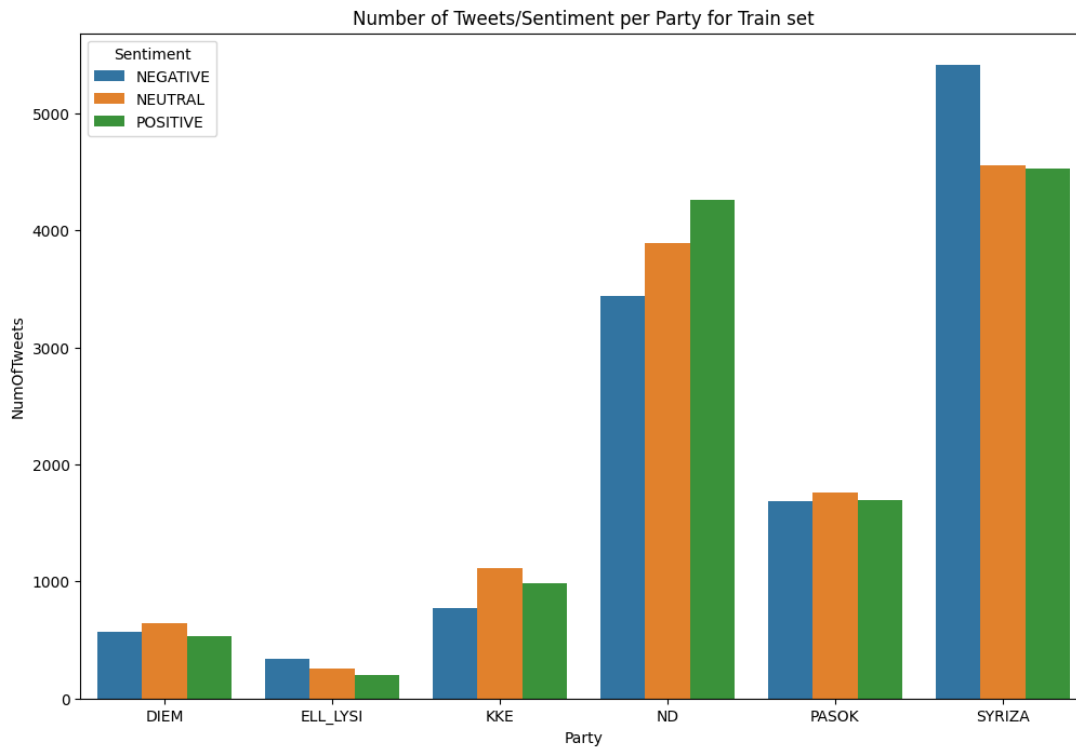
```
Num of unique words in df_train_set: 58389
Num of unique words in df_test_set: 26263
Num of unique words in df_valid_set: 16639
```

Analysis

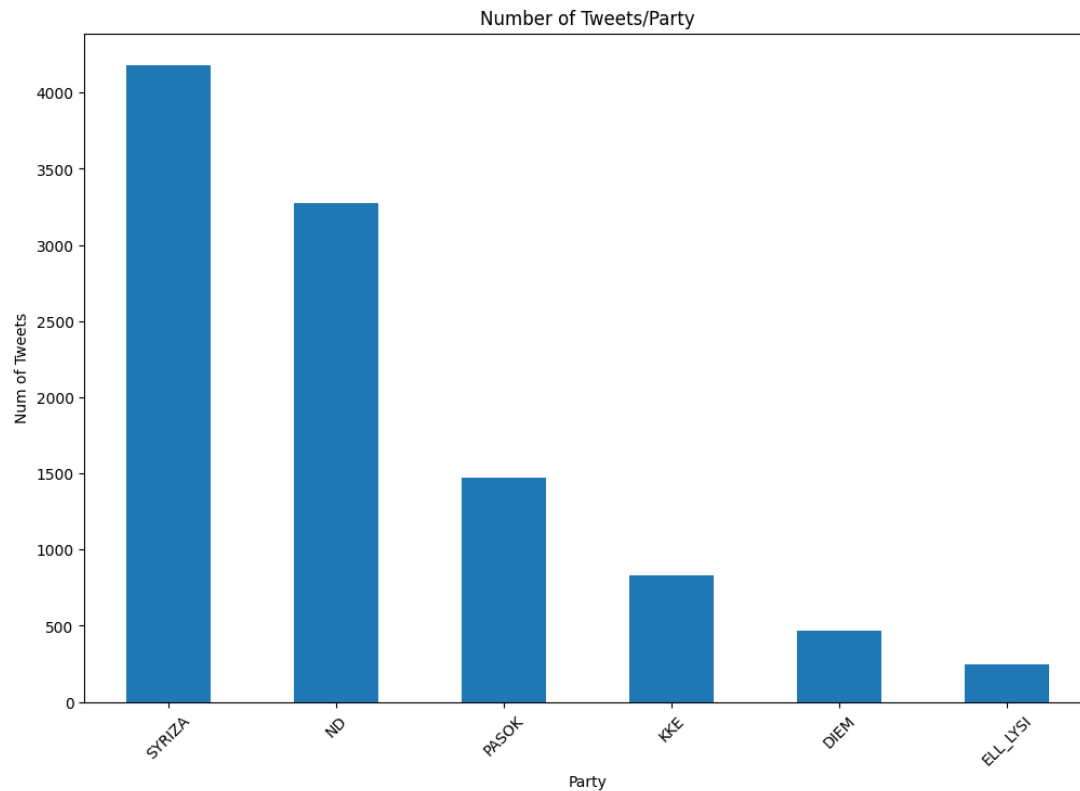
For each set (train, validation and test) we plotted some barplots to visualize the number of tweets about each party and the emotion of these tweets. Also we plot the total number of

tweets for each party. Also after lemmatization and the general preprocessing of the tweets we plotted the corresponding wordcloud for each set.

Number of tweets and sentiment for each party



Number of tweets for each party for the train set



Word cloud for train set

In our analysis, we have utilized word clouds as a visual tool to represent the data from our train, test, and validation sets. Word clouds, or tag clouds, are graphical representations where the frequency of each word in the text data is depicted with various font sizes. The more frequently a word appears in the dataset, the larger and bolder it appears in the word cloud. This visualization technique is particularly useful for quickly identifying key themes and terms that are most prominent in large volumes of text.

The word clouds generated for our datasets reveal significant insights, especially concerning the political context of the tweets. Given that our data is centered around the Greek general elections, it is not surprising to find that the names of the two dominant parties at the time, Syriza (Σύριζα) and Nea Dimokratia (Νέα Δημοκρατία), along with the names of their party leaders, are among the most prominently featured words in the tweets. This observation aligns with the expected discourse during an election period, where discussion and commentary are often focused on the major political parties and their leaders..

[illegible]

In our project, we initially worked with the given datasets as they were originally partitioned into separate training, validation, and testing sets. This conventional partitioning is a standard practice in machine learning, ensuring that models are trained, tuned, and tested on distinct data subsets. However, to explore potential improvements in model performance, particularly in terms of the F1 score, we adopted a strategy of combining the training and validation sets.

However, combining the training and validation sets necessitates a different approach to model validation to avoid overfitting and ensure the model's generalizability. To address this, we implemented cross-validation

Vectorization

In our text analysis project, we employed the Term Frequency-Inverse Document Frequency (TF-IDF) vectorization technique to convert our textual data into a format suitable for machine learning algorithms. Specifically, we used the `TfidfVectorizer` from Python's `scikit-learn` library, which is a widely used tool for text vectorization.

The TF-IDF vectorization technique is a numerical statistic that reflects the importance of a word in a document relative to a collection (corpus) of documents. It has two components:

Term Frequency (TF): This represents how frequently a term occurs in a document. In TF-IDF, terms that occur more frequently in a specific document but less frequently across multiple documents are given higher weightage. This is under the assumption that frequent terms in a specific document are more informative than those commonly used in all documents.

Inverse Document Frequency (IDF): This measures the importance of the term across a set of documents. Words that are common across all documents have lower IDF scores. IDF diminishes the weight of terms that occur very frequently in the dataset and increases the weight of terms that occur rarely.

By combining these two metrics, TF-IDF allows us to evaluate the relevance of words in each document, giving more weight to terms that are unique to a particular document. This technique is particularly useful in text classification tasks, as it helps highlight the most distinctive words in each document.

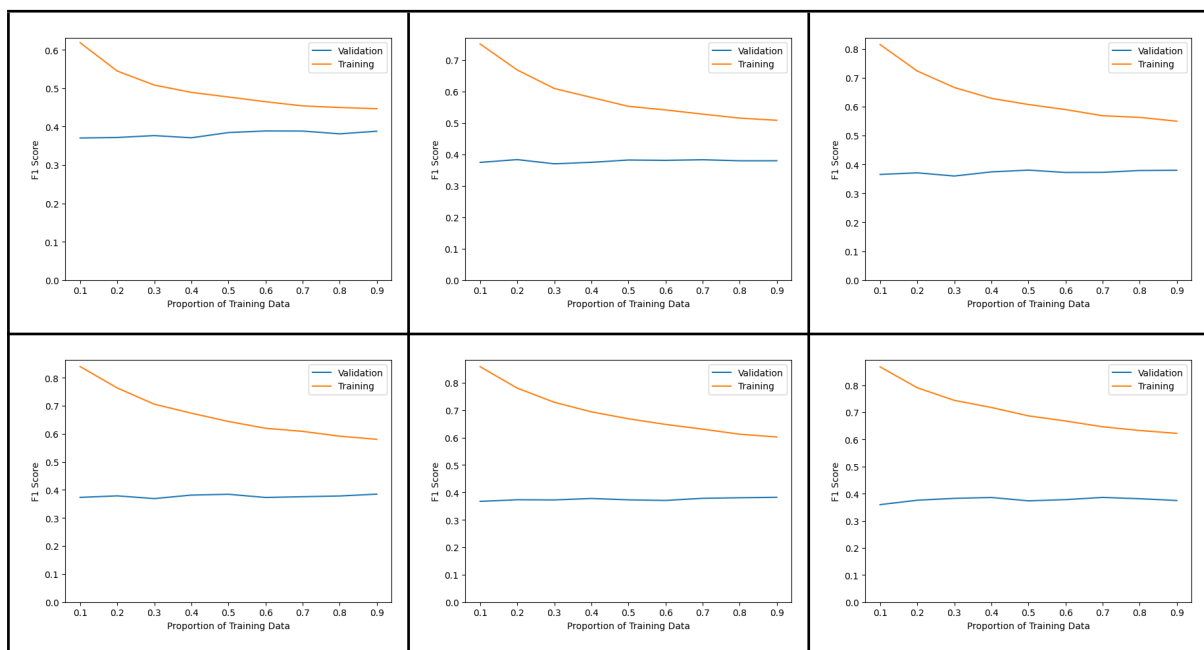
In the following experiments initially we experiment with the number of the **max_features** hyperparameter which limits the number of features (unique words) to the top n most frequent words in the dataset.

Algorithms and Experiments

Experiments

Experiments with the max_features hyperparameter

First we conduct experiments with the max_features hyperparameter with the use of the **choose_max_features** function. We choose values from 1000 up to 11000 with step 2000. We plot the learning curves each time regarding the f1. Below you can observe the training f1 score with orange and the validation f1 score with blue. At the first row it is for values **1000, 3000, 5000** starting from the left. At the second row it is for values **7000, 9000, 11000**.



Below we can observe the mean training f1 and the mean validation f1 for each value of the max_features.

max_features	mean_f1_train	mean_f1_validation
1000	0.494911	0.380094
3000	0.583620	0.378347
5000	0.634697	0.372879
7000	0.669604	0.377662
9000	0.691336	0.375037

11000	0.708648	0.377531
-------	----------	----------

Trend Observation:

As `max_features` increases, the mean F1 score on the training data generally increases. This suggests that with more features, the model is able to learn more details from the training data, possibly capturing more nuances in the text.

Overfitting Indication:

There is a noticeable gap between the mean F1 scores on the training data and the validation data. This gap widens as `max_features` increases, which could indicate a trend towards overfitting. The model might be becoming too complex, fitting the noise in the training data rather than underlying patterns.

Optimal `max_features` Selection:

The mean F1 validation score does not improve consistently as `max_features` increases. In fact, it slightly decreases after 1000 features, suggesting that adding more features does not necessarily benefit the model's performance on unseen data.

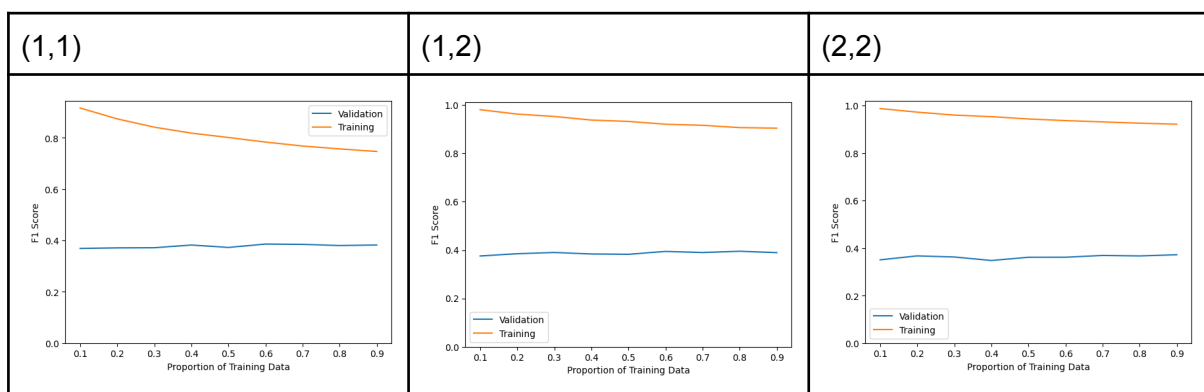
Interestingly, the highest mean F1 validation score is observed at `max_features=1000`, hinting that a simpler model with fewer features might generalize better in this case.

Model Complexity and Generalization:

The best generalization performance on the validation set (as measured by the mean F1 validation score) does not correspond with the highest complexity model (as measured by `max_features`). This emphasizes the importance of balancing model complexity with the ability to generalize.

Experiments with the `ngram_range` hyperparameter

This hyperparameter is pivotal as it dictates the breadth of word combinations considered as features by our model. Specifically, an `ngram_range` of (1, 1) restricts the feature set to unigrams, capturing individual words only, while a range of (1, 2) extends this to include both unigrams and bigrams, thereby encompassing pairs of consecutive words. Such inclusion of bigrams allows the model to recognize context and sentiment that may be expressed through two-word phrases, which could be critical in discerning the nuanced language often found in social media.



ngram_range	mean_f1_train	mean_f1_validation
(1, 1)	0.811498	0.377663
(1, 2)	0.934916	0.387331
(2, 2)	0.947562	0.361919

Unigrams Only (1, 1):

The model with only unigrams has a moderately high mean F1 score on the training set and the lowest gap between training and validation F1 scores. This suggests that while the model is relatively good at generalizing, it may not be capturing as much contextual information as could be gleaned from considering word pairs. The smaller gap indicates less overfitting compared to the other ngram_range settings.

Unigrams and Bigrams (1, 2):

The inclusion of bigrams alongside unigrams significantly increases the mean F1 score on the training data, suggesting that the model is better able to fit to the nuances in the training set. The validation score also sees an improvement, which is the highest among the three settings. However, the larger gap between the training and validation scores could indicate that the model is starting to overfit, as it may be capturing noise in addition to the relevant signals.

Bigrams Only (2, 2):

This configuration leads to the highest mean F1 score on the training set, indicating that the model fits very closely to the training data's bigram patterns. However, the validation F1 score drops significantly compared to the (1, 1) and (1, 2) settings. This drop suggests that bigrams alone may not generalize well and could be overfitting to the training data's specific bigram distributions.

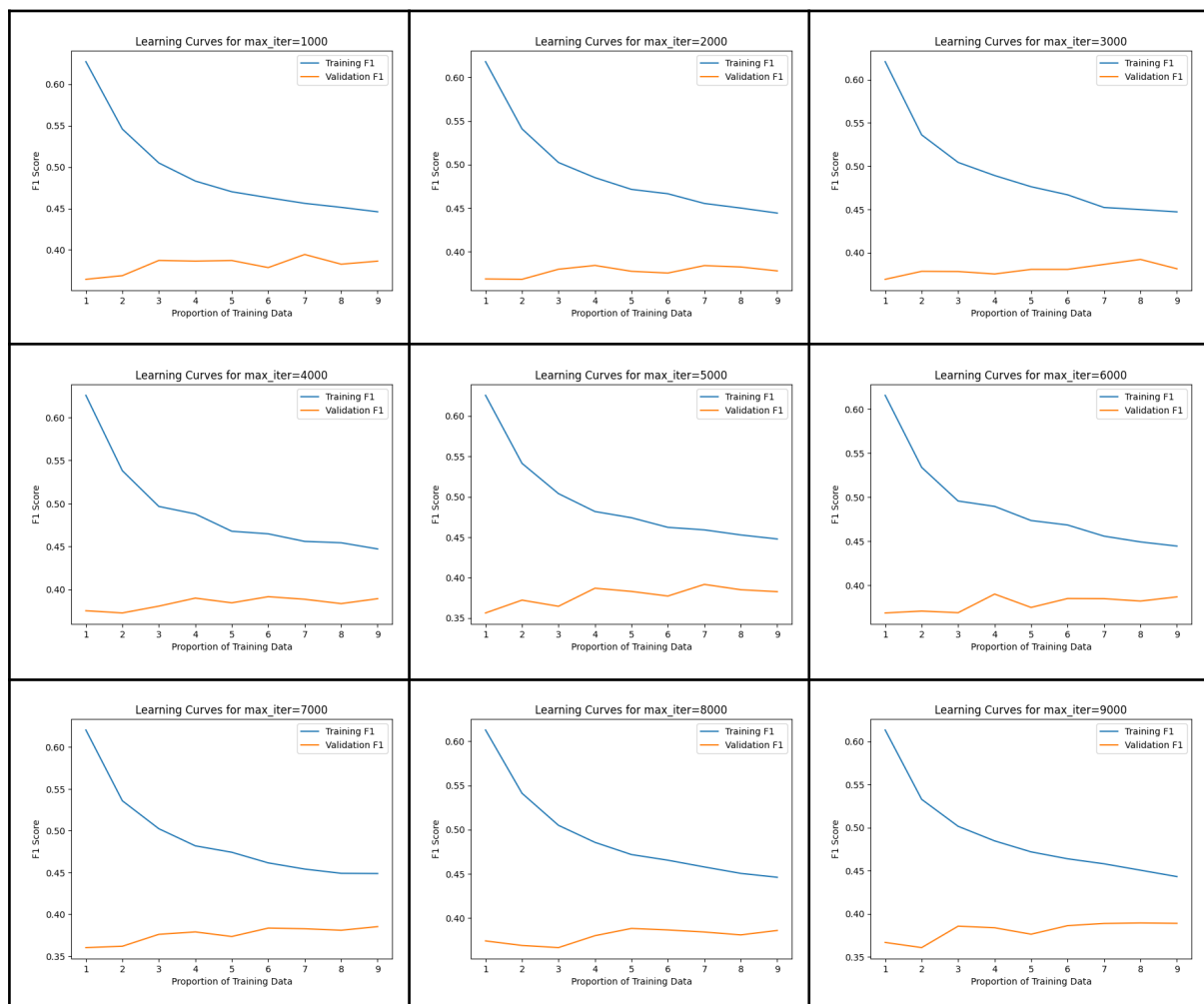
We chose to use **(1,1)** because the gap between training and validation score is the lowest.

Experiments with the max_iter hyperparameter

The `max_iter` hyperparameter in the `LogisticRegression` model dictates the maximum number of iterations the solver will perform during the optimization process. In machine learning, particularly in models that use gradient-based optimization techniques like logistic regression, the number of iterations is a crucial factor that affects both the performance and the computational cost of the model. If the number of iterations is too low, the model may not converge to a solution, leading to underfitting. Conversely, too many iterations may lead to overfitting, especially if the training data contains noise.

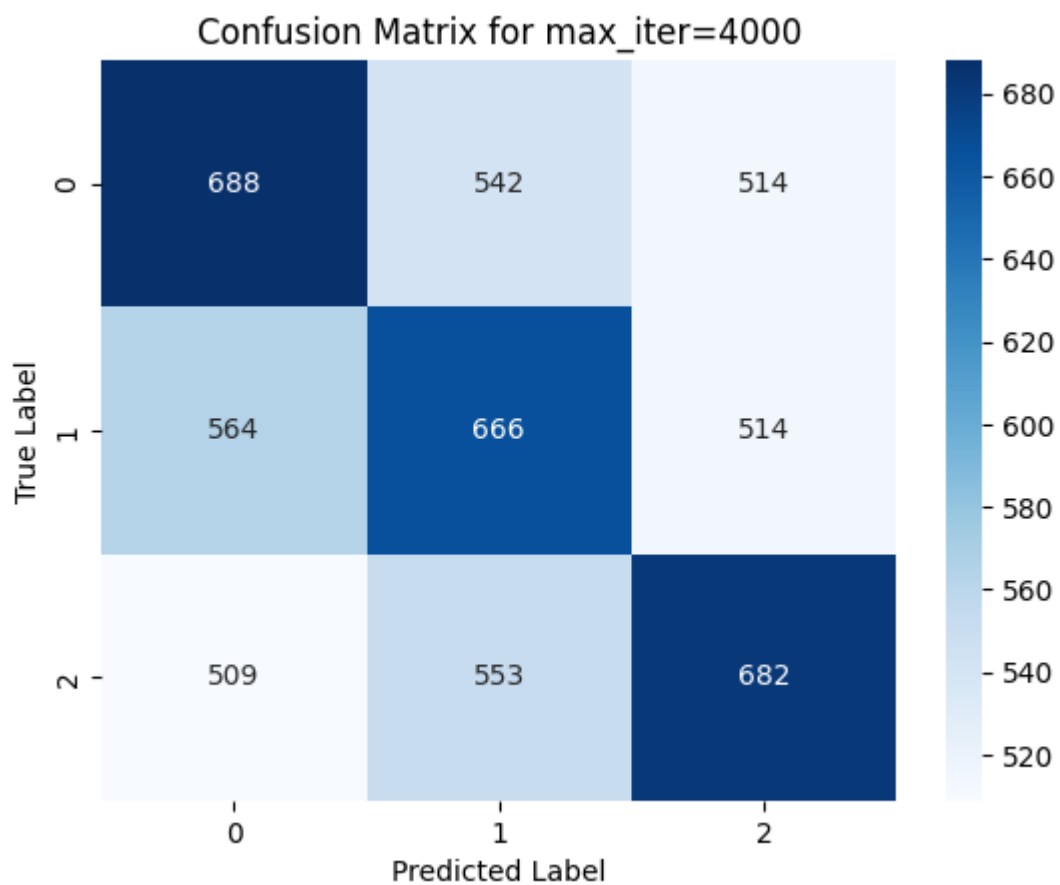
To determine the optimal number of iterations for our logistic regression model, we designed an experimental setup using the `test_iterations` function. This function systematically evaluates the model's performance across a range of `max_iter` values, starting from 1000 up to 9000, increasing in steps of 1000 for each experiment.

Below we can see the learning curves for each value of the `max_iter` hyperparameter. From a first look at the following table it is clear that our model is **overfitting** since the training score and the validation score are not even remotely close.



Mean f1 training score and mean f1 validation score for each max_iter value

max_iter	mean_f1_train	mean_f1_validation
1000	0.494175	0.381803
2000	0.492695	0.377636
3000	0.493601	0.380279
4000	0.493067	0.383903
5000	0.494250	0.377630
6000	0.491899	0.379112
7000	0.491998	0.375778
8000	0.492851	0.379429
9000	0.491048	0.380584



From the confusion matrix above we can observe that our model is not making correct predictions and has a low accuracy.

Experiments with the solver and penalty hyperparameters

We have conducted also experiments with the solver and penalty hyperparameters. For that reason we have created the **test_solvers_and_penalty** function which finds the best solver and penalty regarding the f1 score. It also plots the learning curves for f1, recall and precision using the **plot_learning_curves** function.

The solver and penalty hyperparameters in Logistic Regression are crucial as they define the optimization algorithm used for minimizing the cost function and the regularization technique applied to prevent overfitting, respectively.

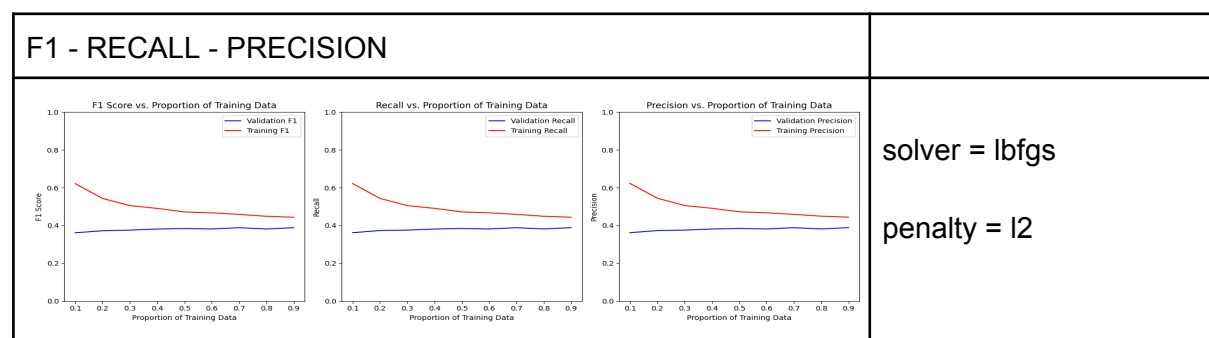
Solver: specifies the algorithm to use in the optimization problem:

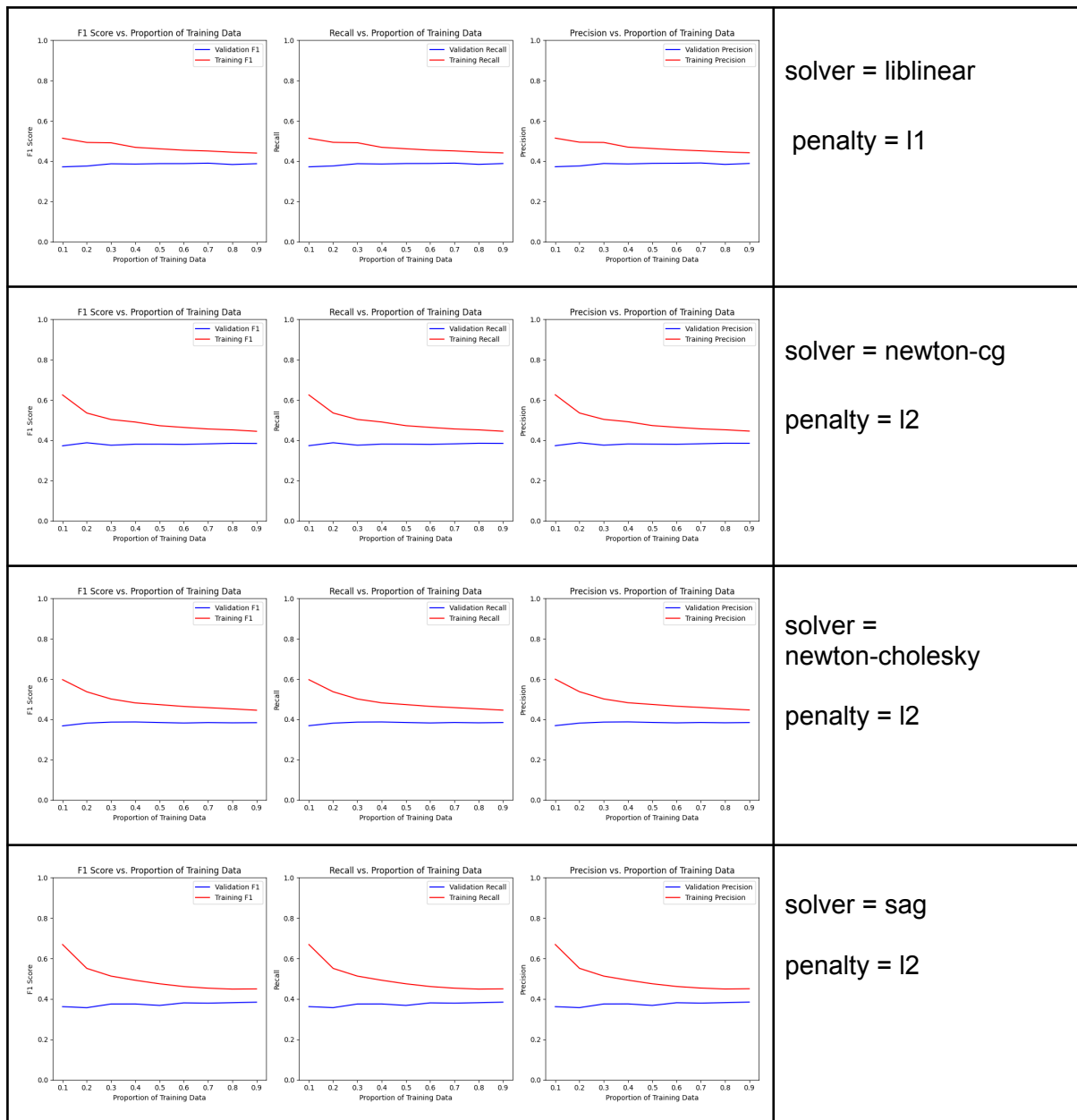
- **lbfgs**, **newton-cg**, and **sag** are more suitable for large datasets as they handle multinomial loss and produce more robust results for multi-class problems.
- **liblinear** is a good choice for small datasets and supports both l1 and l2 regularization.
- **saga** is a variant of sag that also supports the l1 penalty, and is generally faster for large datasets.

Penalty: specifies the norm used in the penalization. Regularization is applied to the cost function to avoid overfitting by discouraging complex models:

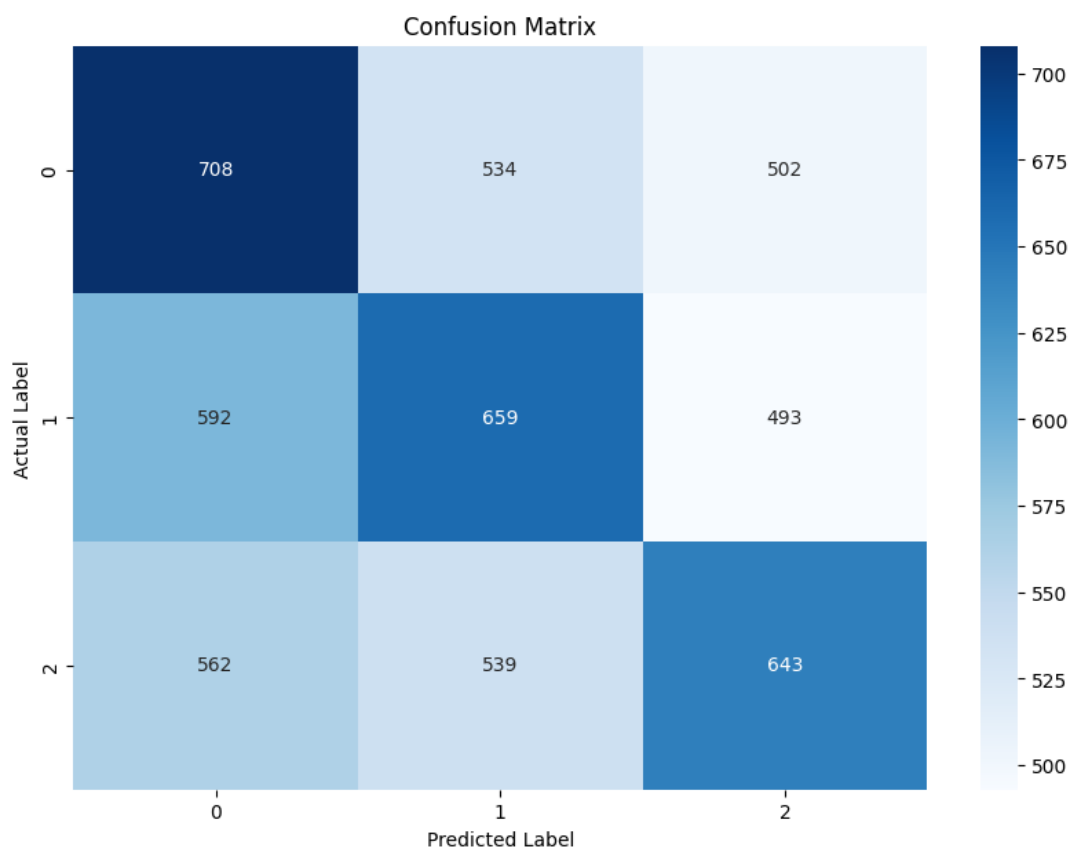
- **l1** can lead to sparse models with coefficients that can be exactly zero. It's useful for feature selection as it tends to shrink coefficients for less important features to zero.
- **l2** tends to shrink coefficients evenly but typically does not set them to zero. This is often more appropriate for problems with many correlated features.

Again from the following learning curves plots and also from the table with the f1 scores it's evident that our model is overfitting since the training score is higher than the validation score for all solvers and penalties and they are not remotely close for any value of the aforementioned hyperparameters.





Solver	Penalty	F1 train score	F1 validation score
lbfgs	l2	0.494502	0.379387
liblinear	l1	0.468746	0.384253
newton-cg	l2	0.493694	0.381039
newton-cholesky	l2	0.490145	0.382411
sag	l2	0.501811	0.373934



Experiments with the C hyperparameter

The C hyperparameter plays a critical role in controlling the trade-off between achieving a low error on the training data and maintaining a model that generalizes well to unseen data.

This hyperparameter is directly tied to the regularization strength with its value being the inverse of the regularization strength.

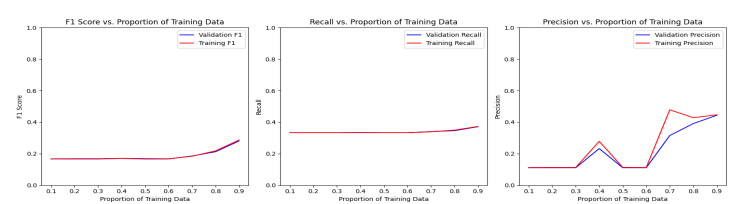
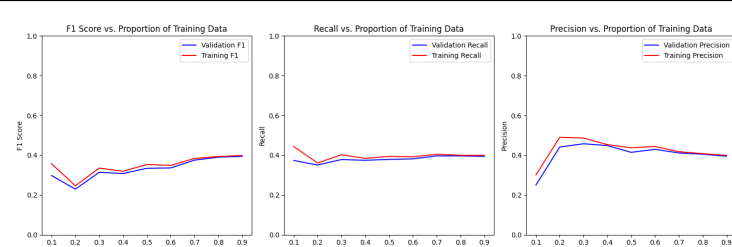
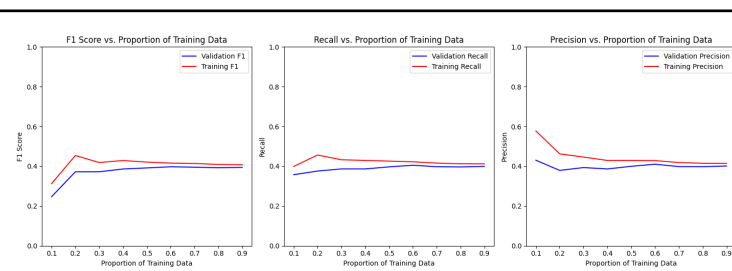
Low values of C:

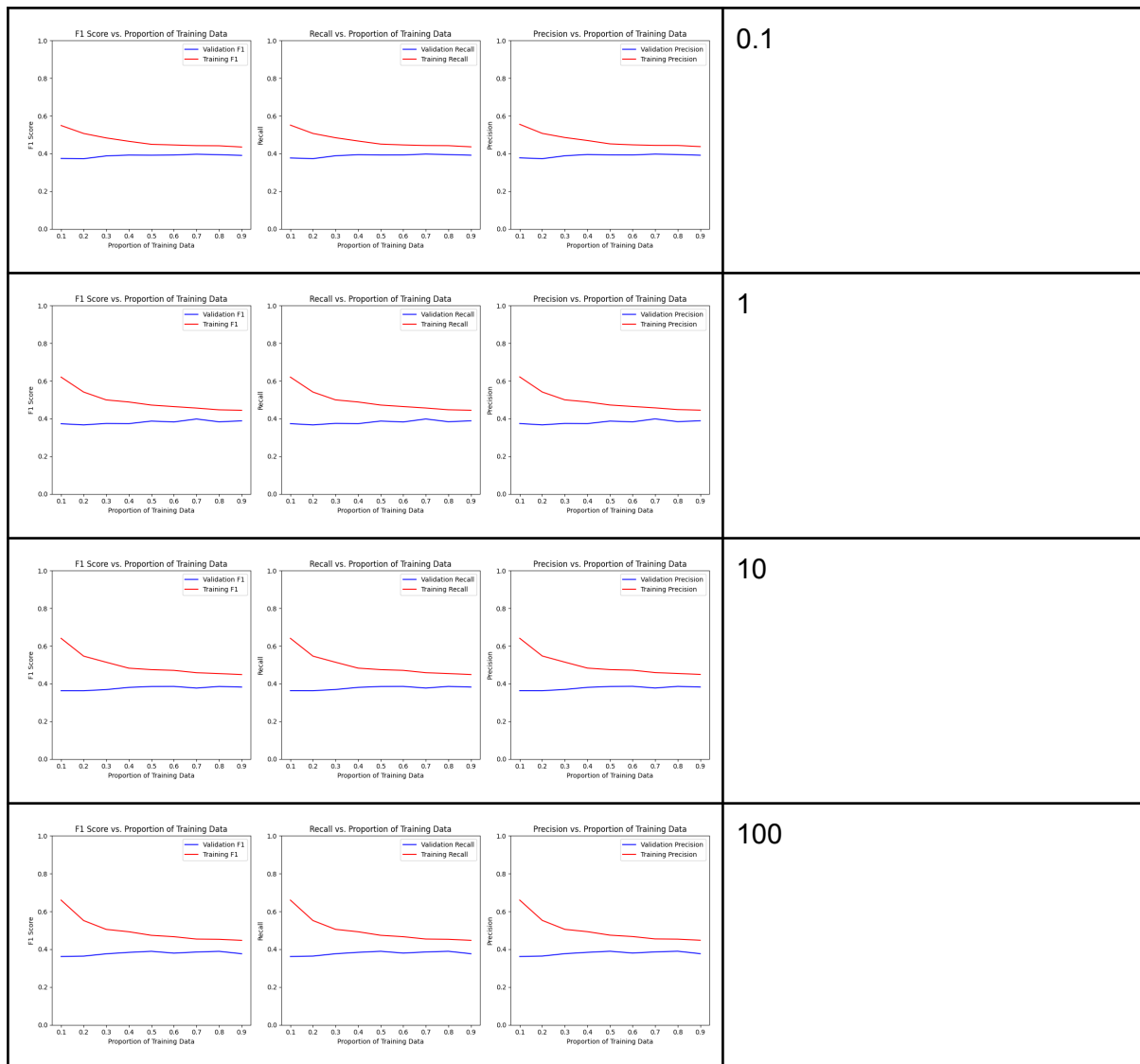
Increase the regularization strength, which creates simpler models that may underfit the training data. This is because the optimization function will prioritize the simplicity (smaller coefficients, depending on the norm used in penalization) of the model over fitting the training data perfectly. Useful when we believe the data is very noisy and we need to prevent the model from learning the noise.

High values of C:

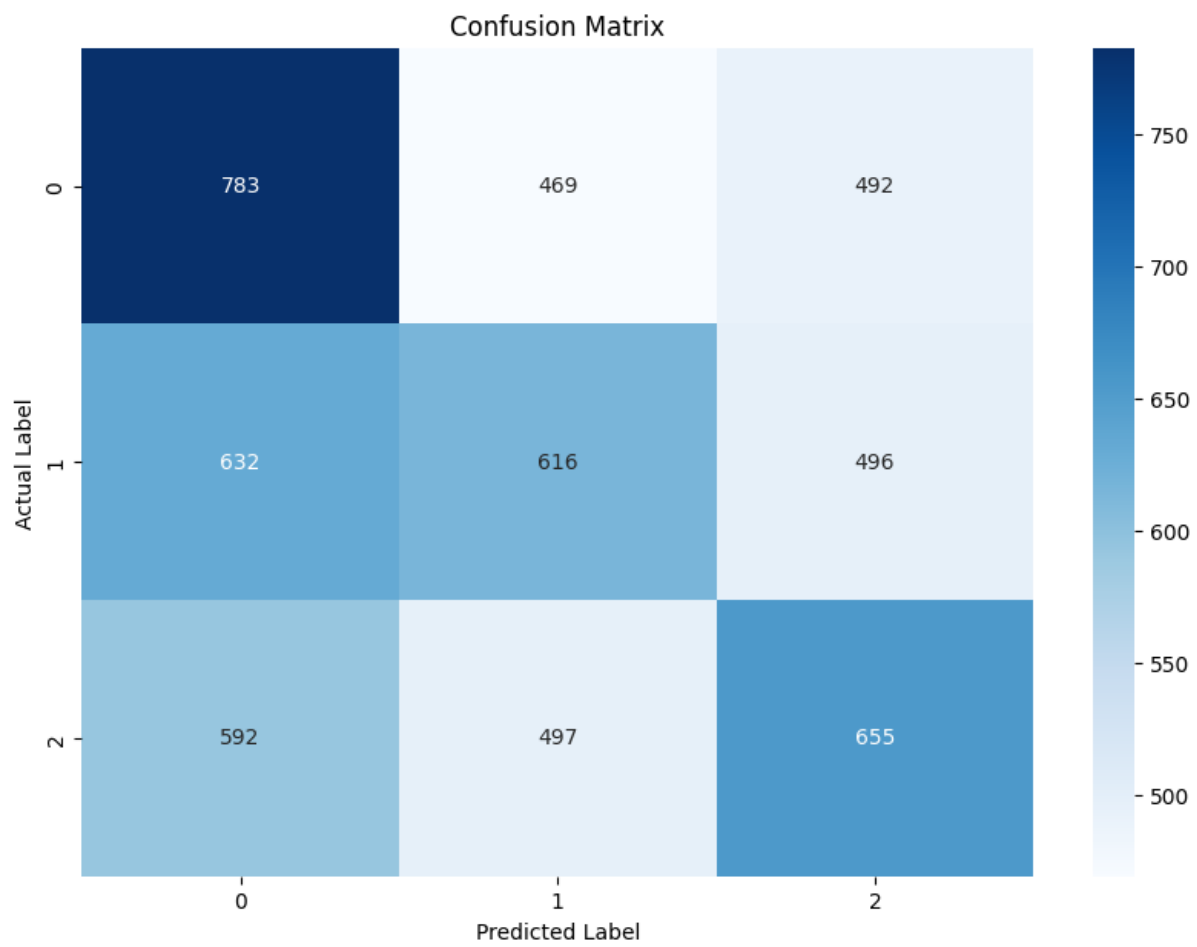
Decrease the regularization strength, allowing the models to become more complex and fit the training data better. This can lead to overfitting if the model starts to learn the noise and detailed fluctuations within the training data. Useful when the model suffers from high bias, i.e., it is too simple and does not capture the underlying trends well.

From the following plots it is evident that our model still **overfitting** despite the various changes in it's hyperparameters.

F1 - RECALL - PRECISION			C
			0.0001
			0.001
			0.01



C	F1 Train Score	F1 Validation Score
0.0001	0.189055	0.186880
0.0010	0.348544	0.331149
0.0100	0.408975	0.371928
0.1000	0.468608	0.387898
1.0000	0.492486	0.380626
10.0000	0.498787	0.376485
100.0000	0.500383	0.378299

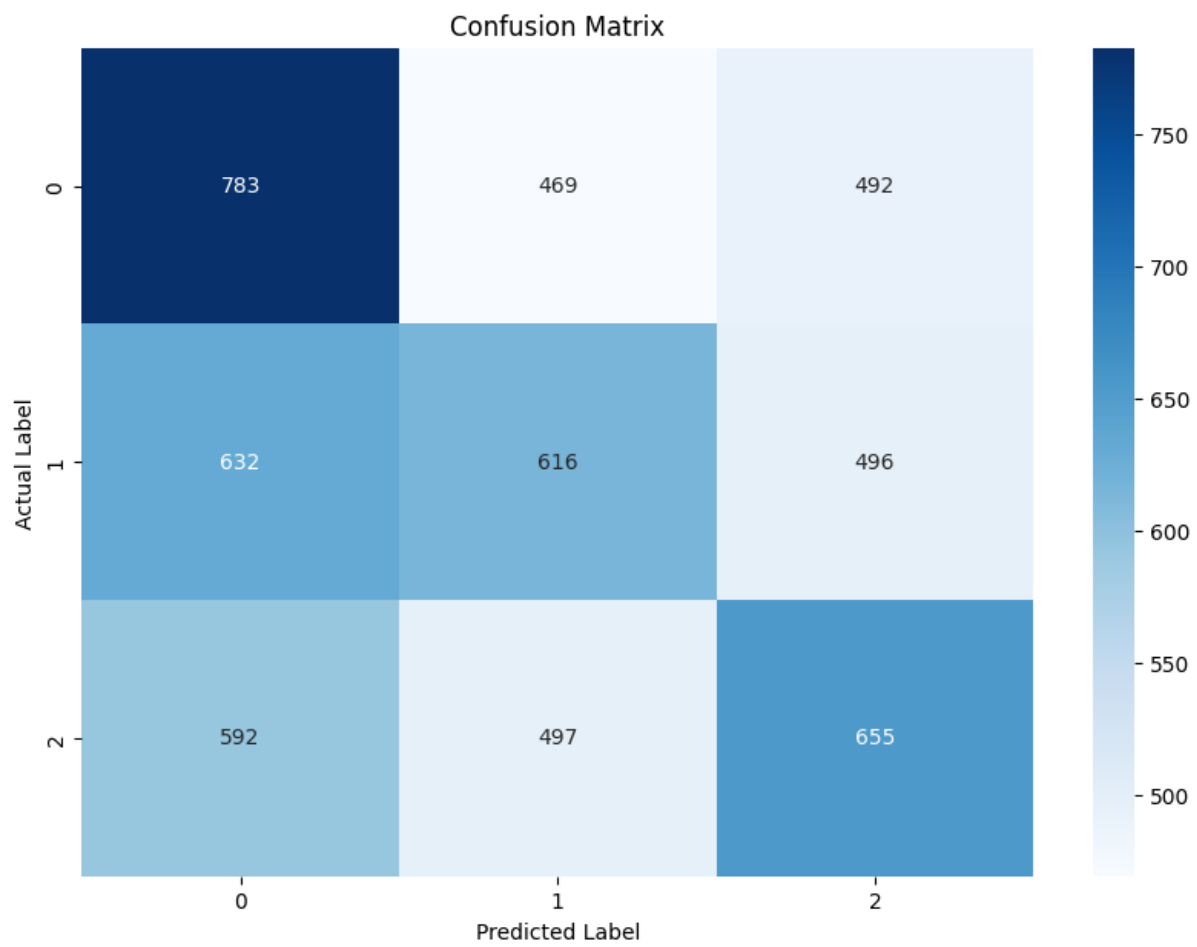


Experiments with the multi_class hyperparameter

We also conducted experiments for the multi_class hyperparameter of LogisticRegression, the best f1 score achieved with the value multinomial, although as it is evident from the graphs, tables and the confusion matrix the model once again overfits, we can observe that the training score is always higher than the validation score across all multi_class values.

F1 - RECALL - PRECISION	multi_class
<p>Three line graphs for multi_class=auto. Each graph plots a metric (F1 Score, Recall, Precision) on the y-axis (0.0 to 1.0) against the Proportion of Training Data on the x-axis (0.1 to 0.9). Each graph contains two lines: a blue line for Validation and a red line for Training. In all three graphs, the training score is consistently higher than the validation score, and both show a slight downward trend as the proportion of training data increases.</p>	auto
<p>Three line graphs for multi_class=ovr. Each graph plots a metric (F1 Score, Recall, Precision) on the y-axis (0.0 to 1.0) against the Proportion of Training Data on the x-axis (0.1 to 0.9). Each graph contains two lines: a blue line for Validation and a red line for Training. In all three graphs, the training score is consistently higher than the validation score, and both show a slight downward trend as the proportion of training data increases.</p>	ovr
<p>Three line graphs for multi_class=multinomial. Each graph plots a metric (F1 Score, Recall, Precision) on the y-axis (0.0 to 1.0) against the Proportion of Training Data on the x-axis (0.1 to 0.9). Each graph contains two lines: a blue line for Validation and a red line for Training. In all three graphs, the training score is consistently higher than the validation score, and both show a slight downward trend as the proportion of training data increases.</p>	multinomial

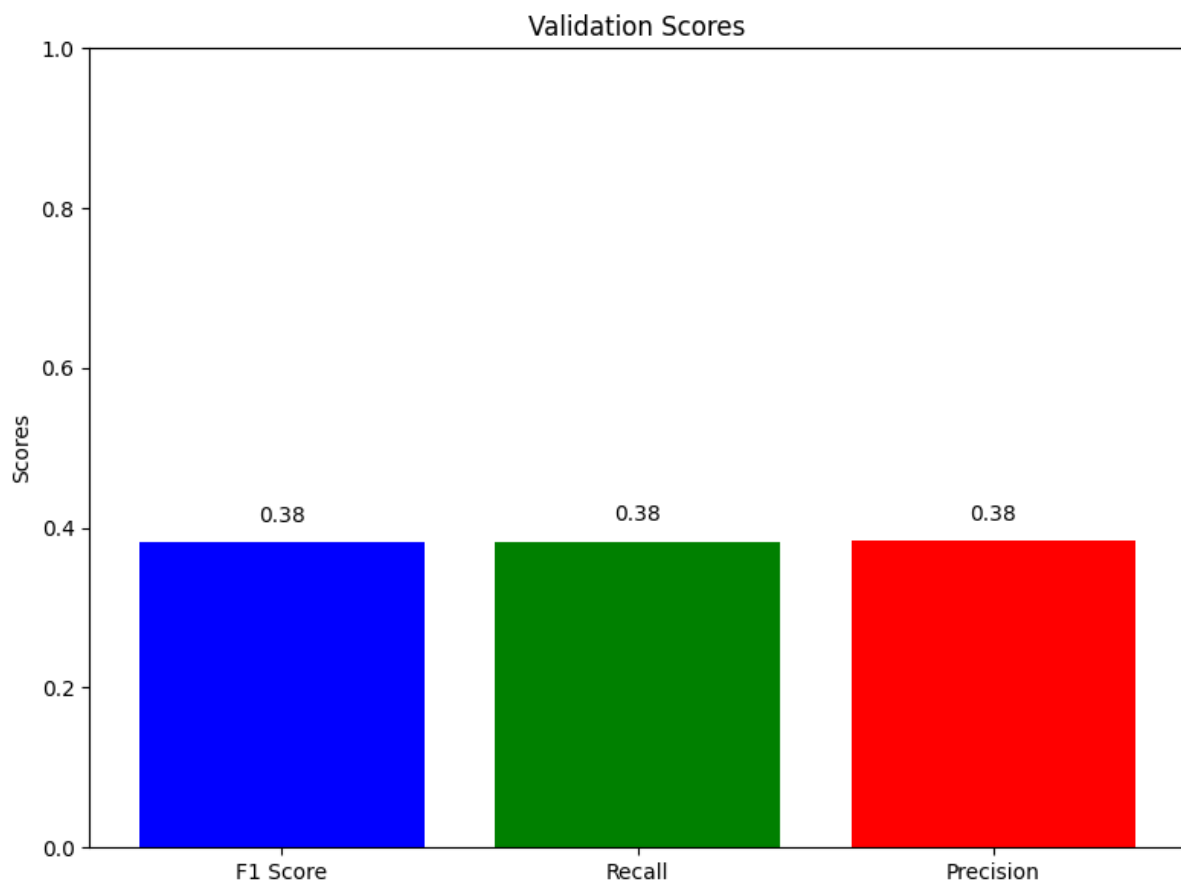
multi_class	F1 Train Score	F1 Validation Score
auto	0.467963	0.386582
ovr	0.457535	0.386851
multinomial	0.468596	0.387168



Experiments with the k fold cross validation

Additionally we **combined** our training data and validation data in order to train our model with a greater number of instances so we can achieve higher f1 score, once again the results we took were close to **0.38** for all evaluation metrics (f1, recall, precision). Merging the training and validation sets effectively increases the dataset's size, allowing the model to 'see' more examples during training. This approach is particularly useful when the initial training set is small and might not capture the full complexity of the data. More data can lead to a more robust model by reducing overfitting, as the model is less likely to learn noise specific to a smaller dataset.

To train our model we used **k-fold cross validation**. K-fold cross-validation works by partitioning the data into **k** distinct subsets, or folds. The model is trained on k-1 folds, while the remaining fold serves as a validation set. This process is iterated k times, with each fold having the opportunity to be used as the validation set. This method is instrumental in mitigating the risks of overfitting, as it ensures that the model's performance is tested on various subsets of the data, promoting generalizability.



Use of CountVectorizer

Additionally we experimented with the use of **CountVectorizer** in order to see if we get higher evaluation scores. The CountVectorizer essentially creates a term-document matrix where the entries are the counts of occurrences of each word in the document. This method offers a straightforward numerical representation of text data, which is often suitable for classification algorithms.

The rationale behind experimenting with CountVectorizer lies in its simplicity and potential effectiveness in certain contexts. Unlike TfidfVectorizer, which weights the term frequencies inversely to their occurrence across documents, CountVectorizer does not account for term importance. Sometimes, this simplicity can yield better results, especially when specific terms that are frequent in the corpus have a strong predictive power for the target variable. The scores we achieved were similar with the results we had with the **TfidfVectorizer**.

```
print(f'f1: {f1} Recall: {recall} Precision: {precision}')
Result:
f1: 0.37644658634592315 Recall: 0.3765938894550059 Precision:
0.3773589571296596
```

Table of trials

Below is the table of trials where you can observe the values which led our model to perform slightly better, even though the change was insignificant.

Hyperparameter	Value	F1 train	F1 validation
max_features	1000	0.493363	0.380777
ngram_range	(1, 2)	0.934373	0.383210
max_iter	4000	0.493218	0.375952
solver, penalty	newton-cg, l2	0.493557	0.376375
C	0.1	0.467461	0.385066
multi_class	multinomial	0.468596	0.387168
k-fold cross validation	5 folds	0.515601	0.380951

Hyper-parameter tuning

For the hyper-parameter tuning we conducted experiments with the `max_features` and the `ngram_range` hyper-parameters of `TfidfVectorizer`. Additionally we experimented with the `max_iter`, `solver`, `penalty`, `C` and multi-class hyperparameters of the `LogisticRegression` object. We found that the values for the hyper-parameters that led the model to perform slightly better were 1000 for `max_feature`, (1, 2) for `ngram_range`, 4000 for `max_iter`, `newton-cg`, l2 for `solver` and `penalty` respectively, 0.1 for `C` and `multinomial` for `multi_class`. Finally we saw that even with the use of k-fold cross validation or the use of `CountVectorizer` we didn't get better results.

In all our trials, the evident **overfitting**, since every time the train score is greater than the validation score, of our model suggests that the unique characteristics of tweet data present a unique challenge. This overfitting may derive from the inherently noisy and sparse nature of text data derived from social media, where slang, abbreviations, and diverse linguistic expressions prevail. Such characteristics can lead the model to learn patterns specific to the training set that do not generalize well to unseen data. Additionally, the limitation of a relatively small dataset could exacerbate the issue, as the model might not be exposed to a sufficiently varied representation of the problem space.

Optimization

Regarding optimization we used various solvers for our Logistic Regression model. More specifically we conducted experiments with `lbfgs`, `liblinear`, `newton-cg`, `newton-cholesky`, `sag`. The solver that succeeded a little better f1 score was `newton-cg`. Although as was mentioned the overfitting is evident.

lbfgs: In a nutshell, it is analogue of the Newton's Method, yet here the Hessian matrix is approximated using updates specified by gradient evaluations (or approximate gradient evaluations). In other words, using an estimation to the inverse Hessian matrix.

The term Limited-memory simply means it stores only a few vectors that represent the approximation implicitly.

It may perform better when the dataset is small as it saves a lot of memory but there are some serious drawbacks such that if it is not paid attention to, it may not converge to anything.

liblinear: It's a linear classification that supports logistic regression and linear support vector machines.

The solver uses a Coordinate Descent (CD) algorithm that solves optimization problems by successively performing approximate minimization along coordinate directions or coordinate hyperplanes.

`liblinear` applies L1 Regularization and it's recommended when we have high dimension dataset (recommended for solving large-scale classification problems).

newton-cg: Refers to Newton's method for optimization with the conjugate gradient method. It's more suitable for larger datasets.

Newton's method uses in a sense a better quadratic function minimisation. A better because it uses the quadratic approximation (i.e. first AND second partial derivatives).

newton-cholesky: A variant of Newton's method that utilizes the Cholesky decomposition to solve optimization problems. This method is typically more efficient for certain types of problems, but it's not a standard solver in scikit-learn as of my last training data.

sag (Stochastic Average Gradient): SAG method optimizes the sum of a finite number of smooth convex functions. Like stochastic gradient (SG) methods, the SAG method's iteration cost is independent of the number of terms in the sum.

It is faster than other solvers for large datasets, when both the number of samples and the number of features are large.

saga: The SAGA solver is a variant of SAG that also supports the non-smooth penalty L1 option (i.e. L1 Regularization). Hence, is therefore the solver of choice for sparse multinomial logistic regression and is also suitable for very Large datasets.

Evaluation

For evaluation of every hyperparameter and for every experiment we were plotting the learning curves that was showing the f1, recall and precision training and validation scores. We focused mostly on increasing the f1 score since we are building a classifier.

f1: F1 Score is a measure that combines recall and precision. As we have seen there is a trade-off between precision and recall, F1 can therefore be used to measure how effectively our models make that trade-off. One important feature of the F1 score is that the result is zero if any of the components (precision or recall) fall to zero. Thereby it penalizes extreme negative values of either component.

Precision: Precision is a metric that gives you the proportion of true positives to the amount of total positives that the model predicts. It answers the question "Out of all the positive predictions we made, how many were true?"

Recall: Recall focuses on how good the model is at finding all the positives. Recall is also called true positive rate and answers the question "Out of all the data points that should be predicted as true, how many did we correctly predict as true?"

Results and Overall Analysis

From the experiments conducted as mentioned in other sections before it is clear that our model overfits since for every hyperparameter the training score is higher and not at least close to the validation score.

Our model performed slightly better for the following hyperparameter values:

max_iter: 4000

solver: newton-cg

penalty: l2

C: 0.1

multi_class: multinomial

Bibliography

1. <https://machinelearningmastery.com/hyperparameters-for-classification-machine-learning-algorithms/>
2. <https://medium.com/@rithpansanga/logistic-regression-and-regularization-avoiding-overfitting-and-improving-generalization-e9afdcddd09d>
3. https://en.wikipedia.org/wiki/Limited-memory_BFGS
4. <https://medium.com/@arnavr/scikit-learn-solvers-explained-780a17bc322d>
5. <https://www.labeled.ai/blog/what-is-accuracy-precision-recall-and-f1-score>
6. https://www.youtube.com/watch?v=a63PL4PdFVc&ab_channel=TylerCaraza-Harter