Hate Speech Classification - 50.007 Machine Learning

Team 10 : Numpy Pandas

Chua Wei Yang (1007828)
Linda Emilia Shalash (1008508)
Raj Narayan Sikder (1008002)
Shonim Shaheen (1008471)
Luvyn Sequira (1007081)

Task 3 Report

Overview:

Final Best Model

Individually, both SVM and XGBoost are highly effective models. With proper feature engineering and hyperparameter tuning, they can achieve comparable ROC AUC performance scores. In addition, both offer fast training speeds, enabling efficient cross-validation and parameter grid searches. This makes the process of fine-tuning parameters quick and streamlined.

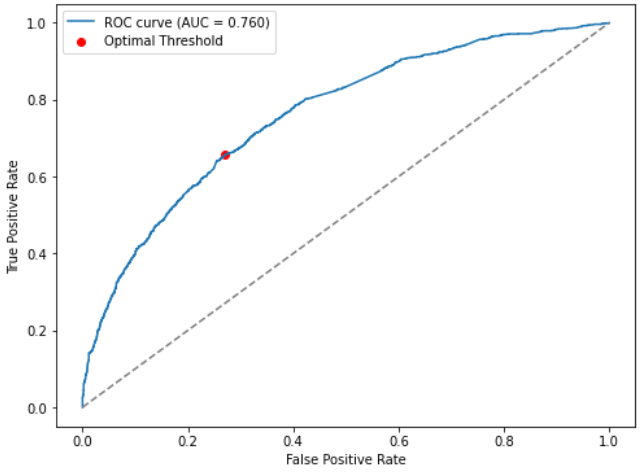

Fig 1 : Tuned XGboost model ROC AUC score: 0.769

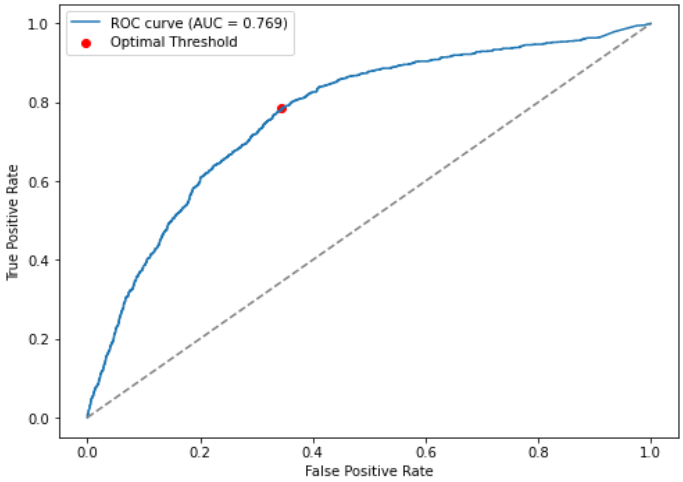

Fig 2 : Tuned Linear SVM model : ROC AUC score: 0.769

To take it a step further we used stacking ensembles with both multiple base models, like XGboost, Linear SVM, RBF SVM and Logistic regression. With a simple Logistic regression model as the meta model for the final prediction.

However, in the end, the final Stack only include XGboost and Linear SVM for 2 reasons:

- Logistic regression and RBF SVM takes very long to train and the trade off of accuracy with train efficiency is not enough
- Prevent overfitting; Using too many similar base model usually tend towards overfitting

Thus, the final model only include XGboost and Linear SVM giving use a consistent ROC AUC of 0.80 ~0.81
And a f1 macro score of 0.72, with heavy bias towards class 0.

validation set classification report

	precision	recall	f1-score	support
0	0.78	0.84	0.81	2262
1	0.69	0.60	0.64	1347
accuracy			0.75	3609
macro avg	0.73	0.72	0.72	3609
weighted avg	0.74	0.75	0.74	3609

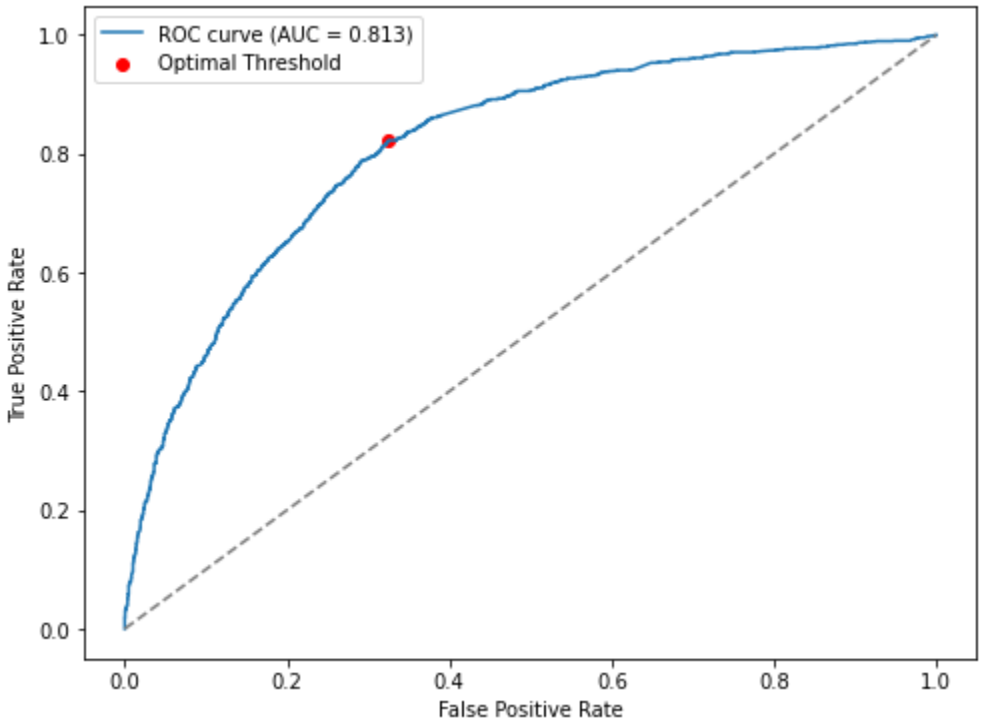

Fig 3 : Tuned Stacked ensembles : ROC AUC score: 0.813

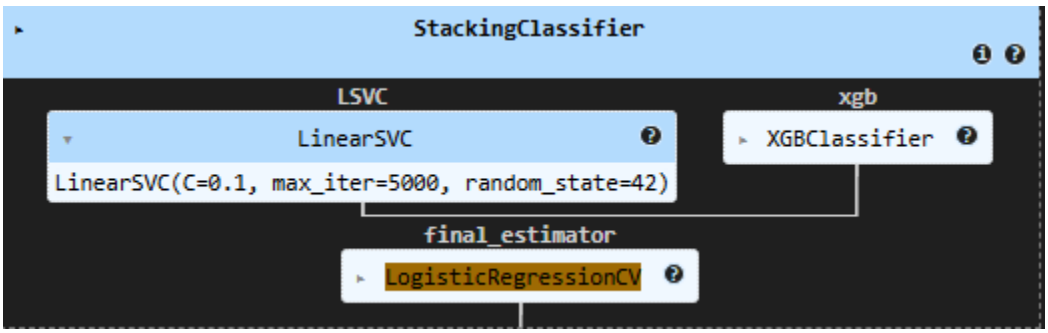

Fig 4: Final model stack

Tuning of Model

Step 1 : Feature Engineer

There's 4 steps of feature engineering in our model.

Step 1: Feature reduction/selection

In our project we selected 3 feature reduction methods and tested them against each other.

- PCA
- LSA
- Chi^2 test

We selected an arbitrary k value to reduce to and run a base Logistic Regression model and compared their f1 scores. For example,

```
Accuracy with all features:     0.5247 ± 0.0205
Accuracy with Chi2:               0.5636 ± 0.0122
Accuracy with PCA:               0.5254 ± 0.0151
Accuracy with LSA:               0.5250 ± 0.0174
```

In this case we chose to reduce our feature to 2000 features and perform an evaluation of the data set. As seen above, Chi^2 performs better than the other two methods by a decent amount.

** Note: We perform the same test on other basic model and reached the same outcome*

Why did we choose these 3 algorithms?

Step 2: Optimal K value

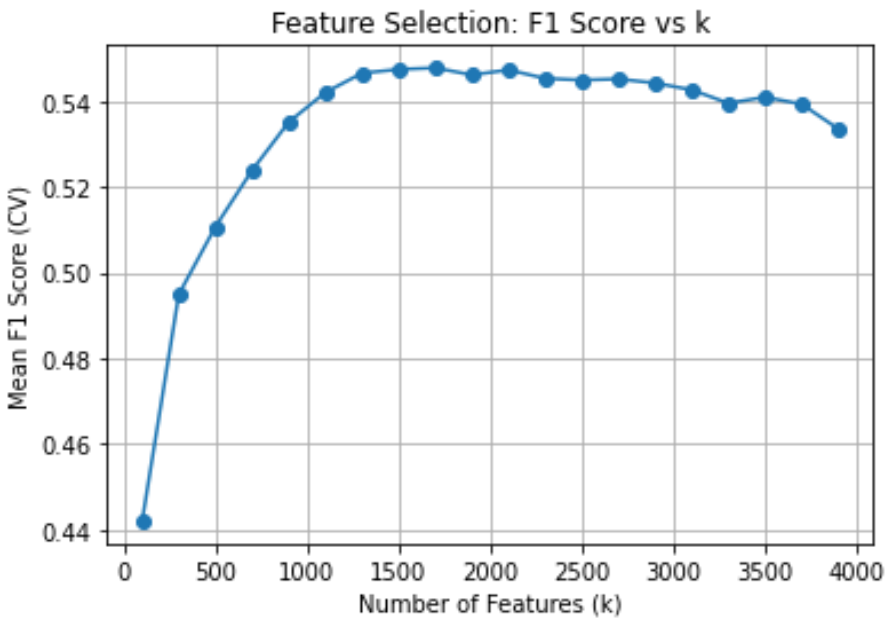

Fig 5: graph of K-value vs Mean F1 score

We evaluated a range of *k* values to identify the optimal number of features that maximizes information capture without introducing redundancy. As illustrated in Figure 5, the performance peaks at *k* = 1700. However, in our model, we selected *k* = 1400 because the performance curve begins to plateau at *k* ≥ 1500. At *k* = 1500, the model already explains approximately 90% of the variance in the data, so using fewer features provides a more efficient representation without sacrificing accuracy.

After feature reduction, we scaled the values across the data set, as scaling brings all features to a similar range, allowing them to contribute more equally to the model's learning process

Step 3: Class imbalance

While training previous models for testing, it is noticeable that the training data have a decent class imbalance in favour of class 0. Thus, the recall score of class 1 is always lower than class 0. To rectify this we tried to perform over/undersampling methods to the training data set. However, surprisingly adding synthetic data in this case actually hurt the models.

```
SMOTE Validation F1: 0.7348415592105804
   SMOTE Test F1: 0.7357553478753733

ADASYN Validation F1: 0.7300301938577398
   ADASYN Test F1: 0.7336941448948315

SVMSMOTE Validation F1: 0.7464360189137431
   SVMSMOTE Test F1: 0.746148059194778
```

```
Normal Validation F1: 0.7491475641288506
   Normal Test F1: 0.7498928872376787
```

By running the experiment, it shows that creating synthetic data hurts the f1 score of our models. Thus, we decide not to perform over/under sampling methods to account for class imbalance.
**Note: We perform the experiment with different classification models, and sometimes the SVM-SMOTE result is similar to the untouched data. (RBF SVM and Decision trees). But it never out performs the untouched data.*

Step 4: Highlighting Outliers

To highlight outliers in the dataset during model training, we employed an unsupervised approach using clustering methods. Initially, we applied K-means clustering and labeled data points that exceeded a certain threshold distance from cluster centroids as outliers. However, this approach yielded inconsistent results due to the presence of noisy samples in the dataset.

To address this issue, we switched to Density-Based Spatial Clustering of Applications with Noise (DBSCAN). This method inherently identifies and separates noise or outliers without requiring the specification of the number of clusters, thus simplifying the hyperparameter tuning process.

```
Estimated number of clusters: 44

Estimated number of noise points: 15284
```

After running DBSCAN with our initial hyperparameters, the model identified 44 clusters and classified 15,284 points as noise. Our goal was to achieve a balanced outcome with a relatively low number of clusters and a high number of noise points, without misclassifying too many points as noise.

```
Estimated number of clusters: 30
Estimated number of noise points: 15525
```

Following further tuning, we settled on 30 clusters and 15,525 noise points, as reducing the cluster count further caused a sharp increase in noise points.

Finally, we assigned the distance of each data point from its nearest cluster centroid back into the dataset. This allows the model to recognize and treat points with large distances as potential outliers.

Step 2: Experimenting with different models

We started experimenting with individual models first before deciding to stack them in an ensemble.
The models we decided to use Linear SVC. Logistic Regression , XGboost, and RBF SVC.

Reasons:

- **XGBoost** – A highly robust and flexible gradient boosting algorithm that iteratively builds an ensemble of decision trees. By using boosting, it focuses on correcting errors from previous iterations, leading to higher predictive accuracy.
- **Linear SVC** – Scikit-learn's implementation of a linear Support Vector Classifier is optimized for speed and scalability, making it well-suited for handling large datasets efficiently while maintaining strong linear classification performance.
- **Logistic Regression** – A simple yet effective linear model that performs well for binary classification when there is a clear linear relationship between features and the target variable. It is easy to interpret but may struggle with complex, non-linear boundaries.
- **RBF SVC** – A Support Vector Classifier with a Radial Basis Function (RBF) kernel, capable of mapping input data into a higher-dimensional space to identify complex, non-linear decision boundaries that linear models, such as Logistic Regression or Linear SVC, cannot capture

For hyperparameter tuning, we used RandomizedSearchCV from the sklearn library. It offers comparable results to the traditional GridSearchCV while significantly reducing computation time, making the tuning process faster and more efficient.

XGboost

XGboost is a very powerful, and its speed and scalability, making it suitable for handling large datasets and complex computations
It is based on the gradient boosting principle, where multiple decision trees are built sequentially, with each tree correcting the errors of its predecessors

Without any tuning, it was able to achieve a high ROC AUC score: 0.769. (shown Fig 1)
However, it has the tendency to overfit without any tuning.

```
Training set classification report from the Hypertune Model
              precision    recall  f1-score   support

           0       0.76      0.92      0.83      5218
           1       0.80      0.53      0.63      3201

    accuracy                           0.77      8419
   macro avg       0.78      0.72      0.73      8419
weighted avg       0.77      0.77      0.76      8419
```

Training set classification report from the **Baseline Model**

	precision	recall	f1-score	support
0	0.78	0.94	0.85	5218
1	0.85	0.57	0.68	3201
accuracy			0.80	8419
macro avg	0.81	0.75	0.77	8419
weighted avg	0.81	0.80	0.79	8419

Thus, the tuning process primarily aimed to reduce overfitting. As shown earlier, the tuned model achieved a lower score on the **training dataset**, while maintaining the same score as the baseline model on the **validation dataset**. The validation set classification report is as follows:

val set classification report

	precision	recall	f1-score	support
0	0.72	0.88	0.79	2262
1	0.68	0.43	0.53	1347
accuracy			0.71	3609
macro avg	0.70	0.66	0.66	3609
weighted avg	0.71	0.71	0.69	3609

Compared to the training results, the variance between **training** and **validation** scores has decreased, indicating that the overfitting has been mitigated to some extent. While class 0 is predicted with higher recall and precision, class 1 performance remains lower, suggesting that further imbalance handling or feature refinement may be beneficial.

```
param_distributions = {
    'max_depth': randint(3, 10),           # Tree depth from 3 to 9
    'learning_rate': uniform(0.01, 0.1),   # Learning rate between 0.01 and 0.11
    'n_estimators': randint(50, 300),      # Number of trees from 50 to 299
    'subsample': uniform(0.5, 0.5),        # Subsample ratio from 0.5 to 1.0
    'colsample_bytree': uniform(0.5, 0.5), # Colsample ratio from 0.5 to 1.0
    'scale_pos_weight': randint(1, 5)      # Class imbalance weight between 1 and 4
}
```

Fig 6. Hyperparameter search grid for XGboost

SVM (Linear and RBF)

RBFSVM

To address the computational challenges of training an RBF SVM on a large dataset, we employed a bagging/bootstrap approach. This method reduces the sample size used for training each individual SVM model by drawing multiple smaller, random subsets of the data with replacement (bootstrap samples).

By training several SVM models on these subsets (bagging), the approach helps manage the computational load and training time while preserving most of the predictive power of a full SVM trained on the entire dataset. This ensemble technique benefits from the variance reduction effect of bagging, which often improves model robustness and generalization. We are trading computation speed against accuracy

```
n_estimators = 10
base_svm = SVC(kernel='rbf', probability=True)  # Adjust class weights as needed

# BaggingClassifier will train 10 SVMs each on ~10% of the dataset
bagging_svm_1 = BaggingClassifier(base_svm,
                                  n_estimators=n_estimators,
                                  max_samples=1.0 / n_estimators, # split sample size by number of SVM models

                                  )
```

Fig 7. Splitting Data set by number of SVM model

```
param_grid = {
    'estimator__C': [0.1, 1, 10, 100],
    'estimator__gamma': ['scale', 'auto', 0.001, 0.01, 0.1, 1],
    'estimator__tol': [1e-4, 1e-5, 1e-6],
    'n_estimators': [5, 10, 20, 30],
}
```

Fig8. Hyperparameter search grid for SVM Bagging

Linear SVM

```
param_grid = {
    'C': [0.01, 0.1, 1, 10, 100],   # Regularization strength
    'penalty': ['l2'],        # Type of regularization
    'loss': ['hinge', 'squared_hinge'],  # Loss function

    'tol': [1e-2, 1e-3, 1e-4],       # Tolerance for stopping
'max_iter': [1000, 2000, 5000],
}
```

Fig 9. Hyperparameter search grid for SVM Linear

val set classification report for **linear SVM**

	precision	recall	f1-score	support
0	0.76	0.86	0.81	2262
1	0.71	0.55	0.62	1347

```
    accuracy                           0.75      3609
   macro avg       0.74      0.71      0.72      3609
weighted avg       0.74      0.75      0.74      3609
```

val set classification report for **RBF SVM Bagging**

```
              precision    recall  f1-score   support

           0       0.68      0.94      0.79      2262
           1       0.72      0.27      0.39      1347

    accuracy                           0.69      3609
   macro avg       0.70      0.60      0.59      3609
weighted avg       0.70      0.69      0.64      3609
```

Surprisingly, the overall Linear SVM outperformed the RBF SVM with bagging, indicating that the dataset may exhibit a strong degree of linear separability. This suggests that a linear decision boundary is sufficient to differentiate the classes effectively.

However, the RBF SVM showed superior performance specifically for class 0, achieving a high recall rate of 0.94. This implies that the RBF high-dimensional feature mapping might better capture the characteristics of class 0, effectively enlarging its feature representation. Conversely, this transformation might reduce the distinction or representation of class 1 features, leading to comparatively lower performance for that class.

In summary, the results hint at two possible data characteristics: either the dataset is largely linearly separable, favoring the Linear SVM overall, or the nonlinear mapping of RBF SVM emphasizes class 0 features at the expense of class 1. Further analysis or feature engineering could help clarify and potentially improve classification for both classes.

Logistic Regression

LogisticRegressionCV in Sklearn library ther are 2 interesting solver provider:

- SAGA
- LBFGs

While experimenting with logistic regression we want to find out which is better.

SAGA

- Particularly well-suited for large-scale and high-dimensional datasets.
- Handles sparse data efficiently.
- It benefits significantly from feature scaling (normalization)
- Typically faster on very large datasets, especially when using penalties like L1 or elastic net.

- **LBFGs**
  A quasi-Newton optimization method that approximates the Hessian matrix for faster convergence.
- Works well with *L2* regularization (default) and is the default solver in sklearn.
- Suitable for small to medium-sized datasets and dense data.
- Can require more memory on very large datasets due to its approximation of the Hessian.

Why consider LBFG when SAGA looks most suitable in this case?

Reproducibility:
As a stochastic solver, SAGA may yield slightly different results across runs due to its inherent randomness. In contrast, LBFGS is deterministic and will converge to the same solution regardless of the starting point. This consistency is particularly valuable when classifying test samples that closely resemble those in the training set, and is especially important in stacking, where even small variations in base model predictions can influence the performance of the meta-model across epochs.

Ultimately, the grid search results indicated that SAGA is the preferred solver.

Final Stacking

As mentioned in the overview, the final stacking model contains only two base learners. During prediction, we experimented with adjusting the classification threshold to favor class 1 by lowering it to 0.45. Changing the decision threshold can sometimes improve performance in cases such as:

- Imbalanced class distributions in the training set

- Test datasets containing many outliers or samples that differ significantly from the training data

However, when the adjusted predictions were submitted to Kaggle, performance was actually worse than with the default threshold of 0.5. Further experiments revealed that increasing the threshold to favor class 0 instead improved the accuracy score.

This suggests two possible explanations:

1. The test dataset likely contains a higher proportion of class 0 samples than class 1.

2. The training data distribution may differ significantly from that of the test set, leading to a shift in class proportions or data characteristics.

Update: The submission to kaggle is wrong, to get a score of 100% just label everything as class 0. Thus, for the threshold paradox.

Possible Improvements:

Tuning of hyperparameter of meta model:

We did not tune the hyperparameter of the meta model as we did not see any improvements in result when we did. Although that could be due to the small search grid we use for tuning. The only regularization we perform is Cross-validation.

Using a different meta model:

Although Logistic Regression is commonly used as a meta model due to it being simplistic and quick (with a small data). There could have been a better meta model that we did not identify. For example, a potential better model would be another Decision Tree model like XGboot or Catboost. As the prediction of level 0 base model could have a non-linear relationship with the log-odds of the label. Thus, in this case Decision tree is better suited to capture non-linear relations.

Feature Extraction:

If we are allowed to perform additional feature extraction such as sentence length, repeated word, popular words used etc. The model could have done better, as just purely using TF-IDF data is difficult as it does not capture position in text, text semantics easily.

Use of Efficient Base Models in Stacking – Due to the computational demands of training larger and more complex stacking ensembles, we opted to remove two of the base models. In their place, we selected faster and more efficient algorithms. This could have affected the predictive power of the stack.

Did you self-learn anything that is beyond the course?

- **Boosting / Bagging Ensemble** – Bagging reduces variance by training models on random data subsets and aggregating results. Boosting trains models sequentially, focusing on previous errors to reduce bias and improve accuracy.

- **Stacking Ensemble (Meta Model / Voting)** – Combines predictions from multiple base models using a meta-model. Can use soft voting (probability averages) or hard voting (majority class) to improve generalization. **This should be taught more in depth in class**

- **DBSCAN + One-Class SVM & Isolation Forest** – DBSCAN clusters dense regions and marks sparse points as noise. One-Class SVM models normal data boundaries, and Isolation Forest isolates anomalies, providing robust outlier detection.**This should be taught more in depth in class.**

- **Chi² Test & LSA** – Chi² selects features most related to the target variable. LSA reduces dimensionality in text data, revealing hidden semantic structures and minimizing noise.